



البرمجة بلغة سي

تأليف

Mike Banahan

Declan Brady

Mark Doran

ترجمة

ناصر داخل

أكاديمية
حسوب

البرمجة بلغة سي

دليلك لتعلم البرمجة و برمجة التطبيقات بلغة البرمجة C

Book Title: The C book

Author: Mike Banahan - Declan Brady - Mark
Doran

Translator: Naser Dakhel

Editor: Ghefar Alrefai - Jamil Bailony

Cover Design: Jamil Bailony

Publication Year: 2023

Edition: 1.0

اسم الكتاب: البرمجة بلغة سي

المؤلف: مايك بانهان - ديكلان برادي - مارك
دوران

المترجم: ناصر داخل

المحرر: غفار الرفاعي - جميل بيلوني

تصميم الغلاف: جميل بيلوني

سنة النشر:

رقم الإصدار:

بعض الحقوق محفوظة - أكاديمية حسوب.

أكاديمية حسوب أحد مشاريع شركة حسوب محدودة المسؤولية.

مسجلة في المملكة المتحدة برقم 07571594.

<https://academy.hsoub.com>

academy@hsoub.com

**أكاديمية
حسوب** 

Copyright Notice

The author publishes this work under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

This license is acceptable for Free Cultural Works.

The licensor cannot revoke these freedoms as long as you follow the license terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Read the text of the full license on the following link:

<https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>



The illustrations used in this book is created by the author and all are licensed with a license compatible with the previously stated license.

إشعار حقوق التأليف والنشر

ينشر المصنّف هذا العمل وفقاً لرخصة المشاع الإبداعي نَسب المصنّف - غير تجاري - الترخيص بالممثل 4.0 دولي (CC BY-NC-SA 4.0).

لك مطلق الحرية في:

- المشاركة — نسخ وتوزيع ونقل العمل لأي وسط أو شكل.
- التعديل — المزج، التحويل، والإضافة على العمل.

هذه الرخصة متوافقة مع أعمال الثقافة الحرة. لا يمكن للمرخص إلغاء هذه الصلاحيات طالما اتبعت شروط الرخصة:

- نَسب المصنّف — يجب عليك نَسب العمل لصاحبه بطريقة مناسبة، وتوفير رابط للترخيص، وبيان إذا ما قد أُجريت أي تعديلات على العمل. يمكنك القيام بهذا بأي طريقة مناسبة، ولكن على ألا يتم ذلك بطريقة توحي بأن المؤلف أو المرخص مؤيد لك أو لعملك.
- غير تجاري — لا يمكنك استخدام هذا العمل لأغراض تجارية.
- الترخيص بالممثل — إذا قمت بأي تعديل، تغيير، أو إضافة على هذا العمل، فيجب عليك توزيع العمل الناتج بنفس شروط ترخيص العمل الأصلي.

منع القيود الإضافية — يجب عليك ألا تطبق أي شروط قانونية أو تدابير تكنولوجية تقيد الآخرين من ممارسة الصلاحيات التي تسمح بها الرخصة. اقرأ النص الكامل للرخصة عبر الرابط التالي:

الصور المستخدمة في هذا الكتاب من إعداد المؤلف وهي كلها مرخصة برخصة متوافقة مع الرخصة السابقة.

عن الناشر

أنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

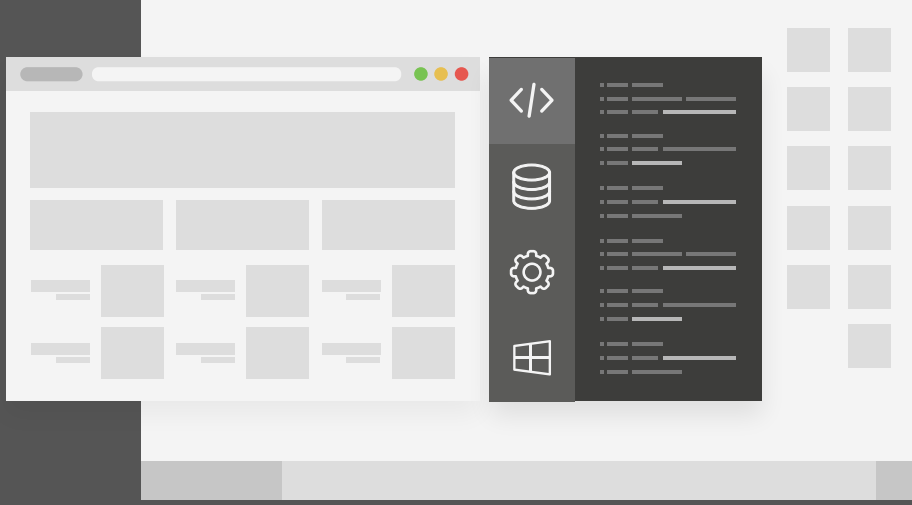


تهدف أكاديمية حسوب إلى تعليم البرمجة باللغة العربية وإثراء المحتوى البرمجي العربي عبر توفير دورات برمجة وكتب ودروس عالية الجودة من متخصصين في مجال البرمجة والمجالات التقنية الأخرى، بالإضافة إلى توفير قسم للأسئلة والأجوبة للإجابة على أي سؤال يواجه المتعلم خلال رحلته التعليمية لتكون معه وتؤهله حتى دخول سوق العمل.



حسوب شركة تقنية في مهمة لتطوير العالم العربي. تبني حسوب منتجات تركز على تحسين مستقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

دورة علوم الحاسوب



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حاسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



المحتويات باختصار

21	مقدمة
28	1. مقدمة إلى لغة سي
50	2. المتغيرات والعمليات الحسابية
103	3. التحكم بالتدفق والتعابير المنطقية
124	4. الدوال Functions
154	5. المصفوفات Arrays والمؤشرات Pointers
202	6. هياكل البيانات
237	7. المعالج المسبق Preprocessor
255	8. مواضيع مخصصة عن لغة سي
280	9. المكتبات Libraries
342	10. تطبيقات عملية
374	11. حلول التمارين

جدول المحتويات

21	مقدمة
21	عن الكتاب
22	نجاح لغة سي
24	المعايير Standards
25	البيئات المستضافة والمستقلة
26	الاصطلاحات المطبعية
26	تسلسل الأفكار
26	أمثلة عن بعض البرامج
27	احترام المعيار
27	المساهمة في النسخة العربية
28	1. مقدمة إلى لغة سي
28	1.1 بنية برنامج لغة سي
29	1.2 الدوال Functions
31	1.3 شرح تمرين 1.1
31	1.3.1 ما الذي احتواه التمرين السابق؟
31	1.3.2 التنسيق والتعليق
32	1.3.3 تعليمات المعالج المسبق
33	1.3.3.1. تعريف Define
34	1.3.4 تعريف وتصريح الدالة
34	1.3.4.1. التصريح
34	1.3.4.2. تعريف
36	1.3.5 السلاسل النصية
37	1.3.6 دالة main
37	1.3.7 التصريح Declaration
38	1.3.8 تعليمة الإسناد Assignment Statement
38	1.3.9 تعليمة الحلقة التكرارية While
39	1.3.10 تعليمة الإعادة return

40	1.3.11 الملخص
40	1.4 بعض البرامج البسيطة بلغة سي
41	1.4.1 برنامج لإيجاد الأعداد الأولية
43	1.4.2 عامل القسمة
44	1.4.3 مثال عن تنفيذ عملية الدخل
45	1.4.4 المصفوفات البسيطة
47	1.5 مصطلحات
48	1.6 خاتمة
48	1.7 تمارين
50	2. المتغيرات والعمليات الحسابية
50	2.1 بعض الأساسيات
50	2.2 المحارف المستخدمة في لغة سي
51	2.2.1 الأبجدية الاعتيادية
52	2.2.2 ثلاثيات المحارف
53	2.2.3 Multibyte Characters المحارف متعددة البايت
55	2.3 البنية النصية للبرامج
55	2.3.1 تخطيط البرنامج
56	2.3.2 التعليق
57	2.3.3 مراحل الترجمة
57	2.4 الكلمات المفتاحية والمعرفات
57	2.4.1 Keywords الكلمات المفتاحية
58	2.4.2 Identifier المعرفات
59	2.5 التصريح عن المتغيرات
61	2.5.1 تمارين
61	2.6 الأنواع الحقيقية Real Types
64	2.6.1 طباعة الأعداد الحقيقية
65	2.6.2 تمارين
65	2.7 الأنواع الصحيحة Integral types
65	2.7.1 الأعداد الصحيحة البسيطة

66	2.7.2 متغيرات المحارف
69	2.7.3 المزيد من الأنواع المعقدة
71	2.7.4 طباعة أنواع الأعداد الصحيحة
72	2.8 التعبيرات والعمليات الحسابية
73	2.8.1 التحويلات
74	ا. الترقية العددية الصحيحة
74	ب. الأعداد الصحيحة ذات الإشارة وعديمة الإشارة
75	ج. الأعداد العشرية والصحيحة
76	د. التحويلات الحسابية الاعتيادية
78	هـ. المحارف العريضة
80	و. التحويل بين الأنواع Cast
82	2.8.2 العوامل Operators
82	ا. عوامل المضاعفة
83	ب. عوامل الجمع
83	ج. عوامل العمليات الثنائية
86	د. عوامل الإسناد
87	هـ. عوامل الزيادة والنقصان
89	و. الأسبقية والتجميع
93	2.8.3 الأقواس
94	2.8.4 الآثار الجانبية Side Effects
95	2.9 الثوابت
95	2.9.1 الأعداد الصحيحة الثابتة
99	2.9.2 الأعداد الحقيقية الثابتة
100	2.10 خاتمة
101	2.11 تمارين
101	2.11.1 تمرين 17.2
103	3. التحكم بالتدفق والتعبير المنطقية
103	3.1 التعبيرات المنطقية والعوامل العلاقية
106	3.2 التحكم بالتدفق

106	3.2.1	تعليلة إذا if الشرطية
108	3.2.2	تعليلة do و while التكرارية
108		ا. اختصار عملية الإسناد والتحقق في تعبير واحد
109	3.2.3	تعليلة for التكرارية
112	3.2.4	أهمية تعليمات التحكم بالتدفق
112	3.2.5	تعليلة switch
113		ا. أكبر قيود تعليلة Switch
114		ب. تعبير العدد الصحيح الثابت
114	3.2.6	تعليلة break
115	3.2.7	تعليلة continue
116	3.2.8	تعليلة goto والعناوين labels
117	3.2.9	خلاصة
118	3.3	عوامل منطقية أخرى
120	3.4	عوامل غربية
120	3.4.1	عامل الشرط: ?
122	3.4.2	عامل الفاصلة
123	3.5	خاتمة
123	3.6	تمارين
124	4. الدوال Functions	
124	4.1	ما التغيرات التي طرأت على لغة سي المعيارية بخصوص الدوال؟
125	4.2	أنواع الدوال
126	4.2.1	التصريح عن الدوال
127	4.2.2	تعليلة الإعادة return
128	4.2.3	وسطاء الدوال
130	4.2.4	نماذج الدوال الأولية function prototypes
133	4.2.5	تحويلات الوسطاء
135	4.2.6	تعريف الدوال
136	4.2.7	التعليمات المركبة والتصريحات
138	4.3	مفهوم التعاود Recursion وتمرير الوسطاء إلى الدوال

138	4.3.1	استدعاء الوسيط بقيمته call by value
140	4.3.2	استدعاء الوسيط بمرجهه call by reference
140	4.3.3	التعاود Recursion
145	4.4	مفهوم النطاق Scope والربط Linkage على مستوى الدوال
145	4.4.1	الربط Linkage
149	4.4.2	تأثير النطاق
150	4.4.3	الكائنات الداخلية الساكنة
151	4.5	الخاتمة
152	4.6	تمارين
154	5.	المصفوفات Arrays والمؤشرات Pointers
154	5.1	تمهيد الفصل
154	5.1.1	ما أهمية هذا الفصل؟
155	5.1.2	تأثير لغة سي المعيارية
155	5.2	المصفوفات Arrays
156	5.2.1	المصفوفات متعددة الأبعاد
157	5.3	المؤشرات Pointers
157	5.3.1	التصريح عن المؤشرات
163	5.3.2	المصفوفات والمؤشرات
166	5.3.3	الأنواع المؤهلة Qualified
168	5.3.4	عمليات المؤشرات الحسابية
169	5.3.5	مؤشرات void و null والمؤشرات الإشكالية
171	5.4	التعامل مع المحارف والسلاسل النصية
171	5.4.1	التعامل مع المحارف
173	5.4.2	السلاسل النصية Strings
176	5.4.3	المؤشرات وعامل الزيادة
177	5.4.4	المؤشرات عديمة النوع
179	5.5	عامل sizeof وحجز مساحات التخزين
192	5.5.1	ما الأشياء التي لا يستطيع العامل sizeof فعلها؟
192	5.5.2	نوع قيمة sizeof

193	5.6 مؤشرات الدوال
195	5.7 المؤشرات في التعابير
195	5.7.1 التحويلات
196	5.7.2 العمليات الحسابية
197	5.7.3 التعابير العلاقية
198	5.7.4 الإسناد
198	5.7.5 العامل الشرطي
198	5.8 المصفوفات وعامل & والدوال
200	5.9 خاتمة
201	5.10 تمارين
202	6. هياكل البيانات
202	6.1 لمحة تاريخية
203	6.2 الهياكل Structures
207	6.2.1 المؤشرات والهياكل
211	6.2.2 القوائم المترابطة وهياكل أخرى
218	6.2.3 الأشجار
223	6.3 الاتحادات Unions
226	6.4 حقول البتات Bitfields
228	6.5 المعددات enums
229	6.6 المؤهلات والأنواع المشتقة
229	6.7 التهيئة Initialization
230	6.7.1 أنواع التهيئة
230	6.7.2 التعابير الثابتة
231	6.7.3 استكمال عن التهيئة
235	6.8 خاتمة
236	6.9 التمارين
237	7. المعالج المسبق Preprocessor
237	7.1 أثر المعيار
238	7.2 كيف يعمل المعالج المسبق؟

239	7.3	الموجهات Directives
240	7.3.1	الموجه الفارغ
240	7.3.2	موجه تعريف الماكرو define
241		ا. استبدال الماكرو
243		ب. التنصيب
243		ج. لصق المفتاح Token pasting
244		د. إعادة المسح
245		ه. ملاحظات
246	7.3.3	موجه التراجع عن تعريف ماكرو undef
247	7.3.4	موجه تضمين ملف مصدري include
248	7.3.5	الأسماء مسبقاً التعريف
250	7.3.6	موجه التحكم بتقارير الأخطاء line
250	7.3.7	التصريف الشرطي
252	7.3.8	موجه التحكم المعتمد على التنفيذ pragma
253	7.3.9	موجه عرض رسالة خطأ قسرية error
253	7.4	الخاتمة
253	7.5	التمارين
255		8. مواضيع مخصصة عن لغة سي
255	8.1	مقدمة
255	8.2	التصاريح declarations والتعاريف definitions وإمكانية الوصول accessibility
256	8.2.1	محددات صنف التخزين
256		ا. المدة الزمنية
259	8.2.2	النطاق Scope
260	8.2.3	الربط
261	8.2.4	الربط والتعاريف
262	8.2.5	الاستخدام العملي لكل من الربط والتعاريف
265	8.3	معرف النوع typedef
268	8.4	المؤهلات const و volatile
269	8.4.1	المؤهل const

271	8.4.2 المؤهل volatile
276	ا. العمليات غير القابلة للتجزئة
277	8.5 نقاط التسلسل Sequence points
278	8.6 الخاتمة
280	9. المكتبات Libraries
280	9.1 مقدمة
280	9.1.1 ملفات الترويسات والأنواع القياسية
281	9.1.2 مجموعات المحارف والاختلافات اللغوية
282	9.1.3 ملف ترويسة <stdio.h>
283	9.1.4 ملف الترويسة <error.h>
284	9.2 تشخيص الأخطاء
285	9.3 التعامل مع المحارف
287	9.4 التوطين Localization
290	9.4.1 دالة setlocale لضبط الإعدادات المحلية
291	9.4.2 دالة localeconv
292	9.5 القيم الحدية
292	9.5.1 ملف الترويسة <limits.h>
293	9.5.2 ملف الترويسة <float.h>
295	9.6 الدوال الرياضية
297	9.7 القفزات اللامحلية Non-local jumps
299	9.8 التعامل مع الإشارة
302	9.9 أعداد متغيرة من الوسطاء
305	9.10 الدخل والخرج I/O
305	9.10.1 مقدمة
306	9.10.2 نموذج الدخل والخرج
306	ا. المجاري النصية
307	ب. المجاري الثنائية
307	ج. المجاري الأخرى
307	9.10.3 ملف الترويسة <stdio.h>

309	9.10.4 العمليات على المجاري
309	أ. فتح المجرى
309	ب. إغلاق المجرى
310	ج. التخزين المؤقت للمجری
310	د. التلاعب بمحتويات الملف مباشرة
311	9.10.5 فتح الملفات بالاسم
313	9.10.6 الدالة freopen
313	9.10.7 إغلاق الملفات
313	9.10.8 الدالتان setbuf و setvbuf
314	9.10.9 دالة fflush
315	9.11 الدخل والخرج المنسق Formatted I/O
315	9.11.1 الخرج: دوال printf
316	9.11.2 الرايات
319	9.11.3 الدخل: دوال scanf
321	9.12 عمليات الإدخال والإخراج على المحارف
321	9.12.1 دخل المحرف
322	9.12.2 خرج المحرف
322	9.12.3 خرج السلسلة النصية
322	9.12.4 دخل السلسلة النصية
322	9.13 الدخل والخرج غير المنسق Unformatted I/O
324	9.14 الدوال عشوائية الوصول Random access functions
326	9.14.1 التعامل مع الأخطاء
327	9.15 أدوات مكتبة stdlib
327	9.15.1 دوال تحويل السلسلة النصية
329	9.15.2 توليد الأرقام العشوائية
329	9.15.3 حجز المساحة
330	9.15.4 التواصل مع البيئة
331	9.15.5 البحث والترتيب
332	9.15.6 دوال العمليات الحسابية الصحيحة

333	9.15.7	الدوال التي تستخدم المحارف متعددة البايت
334	9.16	التعامل مع السلاسل النصية
334	9.16.1	النسخ
335	9.16.2	مقارنة السلسلة النصية والبايت
336	9.16.3	دوال بحث المحارف والسلاسل النصية
337	9.16.4	دوال متنوعة أخرى
338	9.17	التاريخ والوقت
341	9.18	الخاتمة
342		10. تطبيقات عملية
342	10.1	وسطاء الدالة main
345	10.2	تفسير وسطاء البرنامج
348	10.3	برنامج لإيجاد الأنماط
354	10.4	مثال أكثر طموحا
373	10.5	الخاتمة
374		11. حلول التمارين
374	11.1	الفصل الأول
374	11.1.1	التمرين الثاني
375	11.1.2	التمرين الثالث
376	11.1.3	التمرين الرابع
377	11.1.4	التمرين الخامس
379	11.2	الفصل الثاني
379	11.2.1	التمرين الأول
380	11.2.2	التمرين الثاني
380	11.2.3	التمرين الثالث
380	11.2.4	التمرين الرابع
380	11.2.5	التمرين الخامس
380	11.2.6	التمرين السادس
380	11.2.7	التمرين السابع
380	11.2.8	التمرين الثامن

380	11.2.9	التمرين التاسع
380	11.2.10	التمرين العاشر
381	11.2.11	التمرين الحادي عشر
381	11.2.12	التمرين الثاني عشر
381	11.2.13	التمرين الثالث عشر
381	11.2.14	التمرين الرابع عشر
381	11.2.15	التمرين الخامس عشر
382	11.2.16	التمرين السادس عشر
382	11.2.17	التمرين السابع عشر
384	11.3	الفصل الثالث
384	11.3.1	التمرين الأول
384	11.3.2	التمرين الثاني
384	11.3.3	التمرين الثالث
384	11.3.4	التمرين الرابع
384	11.3.5	التمرين الخامس
384	11.3.6	التمرين السادس
385	11.3.7	التمرين السابع
385	11.4	الفصل الرابع
385	11.4.1	التمرين الأول
385	11.4.2	التمرين الثاني
387	11.4.3	التمرين الثالث
387	11.4.4	التمرين الرابع
390	11.5	الفصل الخامس
390	11.5.1	التمرين الأول
390	11.5.2	التمرين الثاني
390	11.5.3	التمرين الثالث
390	11.5.4	التمرين الرابع
390	11.5.5	التمرين الخامس
392	11.5.6	التمرين السادس

392	11.6	الفصل السادس
392	11.6.1	التمرين الأول
392	11.6.2	التمرين الثاني
392	11.6.3	التمرين الثالث
392	11.6.4	التمرين الرابع
392	11.6.5	التمرين الخامس
393	11.6.6	التمرين السادس
393	11.6.7	التمرين السابع
393	11.6.8	التمرين الثامن
393	11.6.9	التمرين التاسع
393	11.7	الفصل السابع
393	11.7.1	التمرين الأول
393	11.7.2	التمرين الثاني
393	11.7.3	التمرين الثالث
394	11.7.4	التمرين الرابع
394	11.7.5	التمرين الخامس
394	11.7.6	التمرين السادس
394	11.7.7	التمرين السابع
394	11.7.8	التمرين الثامن

فهرس الأشكال

29	الشكل 1: طريقة عمل المصرف في لغة سي
155	الشكل 2: مصفوفة ذات 100 عنصر
156	الشكل 3: هيكل مصفوفة ثنائية البعد
158	الشكل 4: مصفوفة ومؤشر
159	الشكل 5: مصفوفة ومؤشر مهياً
162	الشكل 6: عند استدعاء الدالة date
162	الشكل 7: عندما تصل الدالة date إلى تعليمة return
174	الشكل 8: أثر استخدام السلسلة النصية
211	الشكل 9: مخطط تخزين الهيكل
212	الشكل 10: قائمة مترابطة باستخدام المؤشرات
218	الشكل 11: شجرة
238	الشكل 12: المعالج المسبق في لغة سي
263	الشكل 13: هيكل ملف الشيفرة المصدرية
344	الشكل 14: وسطاء البرنامج
345	الشكل 15: وسطاء البرنامج بعد زيادة argv

فهرس الجداول

51	الجدول 1: أبجدية لغة سي
52	الجدول 2: ثلاثيات المحارف
58	الجدول 3: كلمات مفتاحية
64	الجدول 4: رموز التنسيق للأعداد الحقيقية
72	الجدول 5: المزيد من رموز التنسيق
82	الجدول 6: التحويل بين الأنواع
83	الجدول 7: عوامل العمليات الثنائية
87	الجدول 8: عوامل الإسناد المركبة
91	الجدول 9: أسبقية العوامل وترابطها
97	الجدول 10: سلاسل التهريب في لغة C
104	الجدول 11: العوامل العلاقية
147	الجدول 12: الربط وقابلية الوصول
151	الجدول 13: ملخص الربط
177	الجدول 14: معاني المؤشرات
177	الجدول 15: المزيد من معاني المؤشرات
239	الجدول 16: موجّهات المعالج المُسبق
259	الجدول 17: التصريحات الخارجية (خارج الدوال)
259	الجدول 18: التصريحات الداخلية
292	الجدول 19: أسماء ملف الترويسة <limits.h>
293	الجدول 20: أسماء ملف الترويسة <float.h>
312	الجدول 21: أنماط فتح الملف
314	الجدول 22: أنواع التخزين المؤقت
317	الجدول 23: التحويلات
318	الجدول 24: الدوال التي تطبع خرجًا مُنَسَّقًا
320	الجدول 25: محددات الحجم
340	الجدول 26: محددات الحجم

دورة تطوير التطبيقات باستخدام لغة بايثون



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



مقدمة

عن الكتاب

هذا الكتاب مترجم عن الإصدار الثاني من الكتاب [The C Book](#) الذي نُشر أول إصدار منه عام 1991 وقد نفذت كل الإصدارات المطبوعة منه وبقيت النسخة المنشورة على الإنترنت والتي ترجمناها إلى العربية. ورغم قدمه، إلا أن محتواه متوافق مع لغة سي المستقرة والتي تُستعمل في مجال هندسة البرمجيات بكثرة خصوصًا في مجال الأنظمة والأنظمة المدمجة `embedded programming` وتعلمها يعطي فهمًا واسعًا عن عملية البرمجة لذلك تجد أن أغلب الجامعات ودورات البرمجة المتوسعة والشاملة تبدأ بتعليم البرمجة بلغة سي.

الكتاب موجّه لفئتين من الأشخاص، الأشخاص المبتدئين في لغة سي C الذين يريدون تعلّمها من البداية، والفئة التي تعرف التعامل مع لغة سي مسبقًا بمعيارها القديم وتريد التعرف على الإضافات الجديدة في المعيار الجديد. نتمنى أن يكون هذا الكتاب مفيدًا وممتعًا لك في نفس الوقت.

نريد الإشارة هنا إلى أن الكتاب غير مناسب للمبتدئين تمامًا الذين ليس لديهم أي خبرة سابقة بالتعامل مع أي لغة من لغات البرمجة الإجرائية `Procedural` عالية المستوى `High-level`، إذ أن لغة سي ليست بالخيار الأمثل للمبتدئين في البرمجة على الرغم من نجاح البعض في تعلّمها وكانت لغتهم الأولى. سنفترض خلال تكلمنا ومناقشتنا للمفاهيم خلال الفصول أن القارئ على إلمام بالمفاهيم الأساسية، مثل التعليمات البرمجية `Statements` والمتغيرات `Variables` والتنفيذ الشرطي `Conditional Execution` والمصفوفات `Arrays` والإجرائيات `Procedures` (أو البرامج الفرعية `Subroutines`) وغيرها. يركّز هذا الكتاب على مميزات لغة سي وخصائصها، بدلًا من تضييع وقتك بالخوض في التفاصيل الرتيبة عن شرح كيفية جمع رقمين، أو شرح رمز عملية الضرب في البرمجة "*"، وركّز هنا على الطريقة التي تُطبّق فيها لغة سي.

سيستمتع الأشخاص الذين يعرفون كيفية التعامل مع لغة سي مسبقاً بالقراءة عن آخر التغييرات في النسخة المعيارية، وكيف سيؤثر في عمل برامج مكتوبة بهذه اللغة؛ فبالرغم من أن التغييرات بين المعيار الجديد والقديم قد تكون غير مهمة للمتعلّمين الجدد، إلا أنه يجب معرفتها. سنصادف كثيرًا من البرمجيات المكتوبة بخليطٍ من المعيار القديم والحديث بعد عدّة سنوات من الموافقة على هذا المعيار الجديد، وذلك وفقًا لعمر البرنامج ووقت برمجته. يركز الكتاب على التمييز بين المعيار الجديد والقديم وتوضيح الفرق الكبير بينهما للأسباب التي ذكرناها، إذ يعد المعيار الجديد بعض هذه التغييرات ضرورية، وأن بعض الجوانب من المعيار القديم قد عفى عنها الزمن ويجب ألا نستمر باستخدامها؛ ولهذا السبب لن نتطرّق إليها بالتفصيل ولكننا سنشرح فقط ما يعني كل تغيير. وبالنسبة لمبرمج ينوي كتابة برنامجه بالمعيار القديم بالاعتماد على الخصائص القديمة، فالأفضل له قراءة كتاب مختلف.

هذه النسخة الثانية من الكتاب، وقد دُفقت وجرى مراجعتها لتتطابق آخر معايير اللغة، إذ استمدت النسخة الأولى محتوياتها من مسودة عن المعيار الجديد، التي كانت مختلفة بعض الشيء عن المسودة المقبولة في النهاية، وخلال عملية مراجعة الكتاب للنسخة الثانية ضمنا بعض الملخصات وفقرة إضافية توضح استخدامات لغة سي والمكتبة القياسية في حل بعض من المشكلات البسيطة.

نجاح لغة سي

كانت لغة سي C الاستثنائية نتاج عمل رجلٍ واحدٍ يعمل في مختبرات بيل AT&T Bell Laboratories يدعى "دينيس ريتشي Dennis Ritchie"، وشهدت هذه اللغة شعبيةً متزايدةً منذ وقت ظهورها، وينظر إليها البعض بكونها أكثر لغات البرمجة استخدامًا حول العالم. لا يوجد عامل أساسي في نجاح لغة سي، بل هناك مجموعة من العوامل المختلفة الملائمة، ولعل أبرزها هو تطوير اللغة على يد ممارسين للبرمجة (مبرمجين) يوميًا بهدف استخدامها عمليًا وليس بغرض الاستعراض النظري، وكانت متعددة الاستخدامات وتؤدي العديد من الأغراض، إذ ركزت على تزويد المبرمج بالقوة والتحكم بدلًا من احتوائها على حدودٍ وقواعد صارمة تحدّ من حريته.

ساهم هذا السبب الكبير في جعل لغة سي مناسبةً للمحترفين أكثر من المبتدئين، إذ يتطلب تعلّم البرمجة في البداية بيئةً آمنةً وبمبسطة، تعطيك استجابةً لأخطائك وكيفية حلّها بسرعة. بمعنى آخر، برامج تعمل حتى لو لم تنفذ ما تريده -بسبب خطأ من طرفك- ولكن لغة سي لا تعمل بهذه الطريقة. الأمر مماثل لاستخدام حطاب محترف للمنشار الآلي لقطع الأشجار، فهو يعي جيّدًا خطورة الأداة هذه إن لمس شفراتها بينما تعمل، ولكنه يستطيع استخدامها بثقة وبمهارة. تعمل لغة سي C بصورةً مشابهة، وعلى الرغم من أن مصرّف لغة سي قد ينبّه المبرمج عند حدوث بعض الأخطاء ويزوده بإرشاداتٍ محدودة تدلّه على سبب الخطأ، إلا أن المبرمج يملك خيار تجاهل تحذيرات المصرّف وإجباره على استخدام التعليمات التي كتبها، ويفرض أنك أردت الحصول على البرنامج الذي كتبته فعلاً، فستحصل عليه دون أيّ قيود. البرمجة باستخدام لغة سي مماثلة لتناول مزيح من اللحم الأحمر النيء مع الخل ولكن سنتنحو من هذه التجربة لا تقلق!

نجاح هذه اللغة القوية غير مرتبطٍ بأسباب تتعلق بحلقة المبرمجين المحترفين، بل هناك العديد من الاستخدامات التي زادت من نجاح هذه اللغة، فلطالما ارتبطت لغة سي بنظام تشغيل يونكس UNIX، وارتبطت شعبيتها بنجاح هذا النظام؛ وعلى الرغم من أنها ليست اللغة التي يقصدها المبرمجون أولًا لكتابة برامج تجارية كبيرة لمعالجة البيانات، إلا أن توافر لغة سي ضمن تطبيقات يونكس التجارية كان من ضمن ميزاتهما، نظرًا لأن نظام يونكس قد كُتب باستخدامها، إذ تطلب استخدام هذا النظام على أي قطعة من العتاد الصلب وجود مصرّف سي أولًا، ولذلك يكاد من المستحيل أن تجد أي نظام يونكس لا يدعم لغة سي. نتيجةً لذلك، بدأت شركات توزيع البرمجيات التي تريد استهداف الأجهزة التي تعمل بنظام يونكس بالتركيز على لغة سي كونها أفضل خيار للحصول على تبني واسع النطاق لبرامجها. إذا أردنا التكلّم بموضوعيّة، يمكننا القول أن لغة سي هي الخيار الأول لسهولة التنقل على جميع بيئات يونكس.

ازدادت شعبية لغة C أيضًا بفضل نمو سوق أجهزة الحواسيب الشخصية، فقد صُممت لغة سي خصيصًا لتطوير البرمجيات عليها، إذ تمنح هذه اللغة عالية المستوى القدرة للمبرمجين على كتابة برنامج سهل القراءة والكتابة، والقوة للتحكم بمختلف موارد الحاسوب ومعمارياته دون اللجوء لاستخدام شيفرّة تجميعية Assembly Code.

تُعرف لغة سي بقدرتها الفريدة على الجمع بين مستويين من البرمجة في وقتٍ واحد، مع المحافظة على قدرة التحكم بالتدفق وهياكل البيانات data structures والإجراءات procedures -وهي جميع الأمور التي تتوقعها من أي لغة حديثة عالية المستوى-، كما تسمح لمبرمجي الأنظمة بالتعامل والتلاعب بخانات الآلة bits، لضمان الوصول الأقرب للعتاد الصلب في حال الحاجة. مجموعة الخصائص هذه مرغوبة جدًا في سوق برمجيات الحواسيب التنافسي، ونتيجة لذلك اتخذ العديد من مطوري البرمجيات لغة سي أدواتهم الأساسية.

وآخر الأسباب في شعبية لغة سي -وليس أقلها أهمية- هو تعدد استخدامات هذه اللغة، إذ فشلت العديد من لغات البرمجة بتزويد العديد من المزايا المهمة في مجال تطوير التطبيقات التجارية والمرتبطة بالوصول للملفات والتعامل مع الدخل والخرج عامةً، إذ كانت معظم لغات البرمجة تزود أدوات الدخل والخرج I/O على أنها حلولًا مضمّنة Built-in تُفهم تلقائيًا بواسطة المصرّف، وهذه هي النقطة التي تتفوق فيها لغة سي، وواحدة من نقاط قوّة نظام تشغيل يونكس أيضًا؛ إذ تنظر لغة سي إلى هذا الأمر على النحو التالي: إذا لم تكن قادرًا على تزويد المبرمج بحل كامل لمتطلب ما، فعليك السماح له ببناء حلّه الخاص بدلًا من تقديم نصف حل، والذي لن يرضي أي أحد. وعلى مصممي البرمجيات حول العالم أن يحذوا حذو لغة سي ويتعلموا منها، إذ أن هذا الأمر لا ينطبق فقط على حلول الدخل والخرج، بل تزداد استخدامات اللغة عن طريق استخدام دوال المكتبات بطريقة لم يفكر مطوّرو اللغة بها.

نصحت مكتبة الدخل والخرج القياسية Standard I/O Library -أو اختصارًا stdio بخطى أبطأ من اللغة نفسها، ولكنها أصبحت مكتبة تقليديّة إلى حدٍ ما بعد أن منحتها لجنة المعايير مباركتها؛ وهذه المكتبة هي مثالٌ حي على استخدام دوال المكتبات، فقد أثبتت أنه من الممكن تطوير نموذج من ملف دخل/خرج يحتوي على

العديد من الخصائص وقابل للتطبيق على عدة أنظمة بجانب يونكس -النظام الذي شكّلت فيه لأول مرة. على الرغم من قدرة لغة سي على تقديم تحكّم بخصائص العتاد الصلب بمستوى منخفض، فإن رونقها المميز واستخدامها حزمة stdio يقدّم لنا العديد من البرامج متعددة الاستخدامات التي تعمل على أنظمة تشغيل مختلفة كثيرًا عن بعضها. إذا لم تُزق لك بعض الحلول الموجودة ضمن هذه المكتبة وأردت تطبيق حلك الخاص الذي يقدم المزيد من الخصائص أو يتخلص من بعضها فيمكنك فعل ذلك في حال معرفتك بالأمر التقني، وهذا هو الشيء المميز بخصوص هذه المكتبة.

المعايير Standards

من الملفت للانتباه تحقيق لغة سي كل هذا النجاح مع غياب معايير رسمية، والملفت للانتباه أكثر أنها لم تؤثر على شعبية أي من اللغات الأخرى التي عالجتها مشاكلها خلال فترة انتشارها، مثل لغة بيسك BASIC، وهذا غير مفاجئ في حقيقة الأمر، إذ وُجد على الدوام كتاب دليل مرجعي للغة معروف جدًا باسم "لغة البرمجة سي The C Programming Language"، والمكتوب بواسطة الثنائي "بريان كيرنغان Brian Kernighan" و"دينيس ريتشي Dennis Ritchie" في عام 1987، وأُطلق عليه تسمية "K&R"، وقد كان هذا الكتاب بمثابة كتاب قوانين صارم لضبط معايير اللغة باختلاف تنوعها.

كان هناك مصرّف واحد على يونكس فقط، وقد عُرف باسم "مصرّف سي المحمول Portable C Compiler"، إذ كُتب على يد "ستيف جونسون Steve Johnson"، ومثّل مرجعًا تطبيقيًا للغة سي. إذا كان مرجع K&R غامضًا بالنسبة لك، فسيساعدك في فهمه حقيقة أنه أُخذ من طريقة بناء مصرّف يونكس.

مع أن هذا الوضع كان مثاليًا، إذ تُعدّ حيازة دليل مرجعي وتطبيق مرجعي طريقةً ممتازةً لتحقيق دوام اللغة ورسوخها بأقل التكاليف، إلا أن عدد الطرق البديلة التي تُطبّق فيها لغة سي أخذ بالازدياد في عالم الحواسيب الشخصية، مما بدأ بتهديد رسوخ هذه اللغة.

بدأت لجنة X3J11 التابعة للمعهد الأمريكي للمعايير الوطنية American National Standards Institute -أو اختصارًا ANSI-، في بدايات ثمانينيات القرن الماضي 1980 بتشكيل معايير رسمية للغة سي، إذ اتخذت اللجنة من K&R دليلًا مرجعيًا لها، وبدأت بهذه المهمة الشاقة، وكان الهدف من ذلك هو إبعاد أي غموض، وتعريف ما ليس معرفًا، وتصحيح أوجه قصور لغة سي والحفاظ على روح اللغة، إضافةً للعمل على تحقيق أكبر قدر ممكن من التوافق ما بينها وبين الممارسات البرمجية التي اعتاد عليها المبرمجون. لحسن الحظ مُثّلت جميع إصدارات سي التي تتنافس مع بعضها الآخر ضمن اللجنة، الأمر الذي شكّل نقطة تلاقٍ قوية لجميع وجهات النظر المختلفة.

كما هي العادة، أخذت عملية تطوير المعايير وقتًا طويلًا، فالعمل لا يقتصر فقط على الجانب التقني، على الرغم من كونه أكثر الأجزاء استهلاكًا للوقت، كما أن إغفال الجوانب الأخرى والإجراءات المترافقة مع ضبط المعايير أمرٌ سهل فعله، فهذه الأجزاء الأخرى مساويةً في الأهمية للجوانب التقنية. أي معيار لا يحظى بالموافقة

الجماعية من طرف كبار الحرفة معرّض كثيرًا للإهمال وعلى الأغلب لن ينال تبنّيًا واسع النطاق. إذًا، العملية المضنية في الحصول على الموافقة ضمن أفراد اللجنة مهمة جدًا في نجاح أي معيار، وهذا يعني في بعض الأحيان التضحية بتحسينات التقنية على حساب ذلك، فهذه العملية عملية ديمقراطية، مفتوحة لآراء الجميع، مما يعني عدولًا بسيطًا عن الأهداف بقدرٍ مساوٍ يتسبب به الهوس بالكمال التقني؛ ولسوء الحظ فقد تأثر موعد وصول المعيار في اللحظات الأخيرة بسبب بعض الإجراءات البعيدة عن المشاكل التقنية. اكتمل العمل على المعيار تقنيًا في كانون الأول من عام 1988، ولكن الأمر استغرق سنةً إضافيةً لحلّ بعض الاعتراضات، وأخيرًا وصلت الموافقة في إطلاق المستند معيارًا رسميًا صادرًا عن المعهد الأمريكي للمعايير الوطنية في السابع من كانون الأول من عام 1989.

البيئات المستضافة والمستقلة

كان للاعتماد على استخدام المكتبات بهدف زيادة إمكانيات اللغة أثرًا كبيرًا على الجانب التطبيقي، إذ لا تقتصر أهمية دوال مكتبة الدخل والخرج القياسية على برمجة التطبيقات، بل هناك العديد من الاستخدامات الأخرى الأساسية لها، وتؤخذ بكونها من المسلمات في بعض الأحيان، مثل التعامل مع السلاسل النصية والفرز والموازنة، والتلاعب بالمحارف ومزيجًا من التطبيقات الأخرى المشابهة التي تتوقع وجودها في معظم مجالات البرمجة.

كان من المهم للمعيار أن يتكفل أيضًا تزويد الدوال الموجودة في هذه المكتبات بشرح مفصّل بسبب الاعتماد الكبير على المكتبات الإضافية، إذ كانت هذه المهمة أكثر تعقيدًا من مهمة توفير شرح مفصّل عن اللغة بحد ذاتها نظرًا لإمكانية توسعة المكتبة أو تعديلها بواسطة مبرمج يمتلك الخبرة اللازمة، وقد كانت مشروحةً جزئيًا في دليل K&R. تسببت هذه المشكلة بظهور العديد من التطبيقات المشابهة ولكن المختلفة عن المكتبات المستخدمة والمدعومة من اللغة، وكانت عملية التوصل لتوصيف المكتبة ودعمها من أصعب المهام التي نفذتها اللجنة، لكنها كانت من أكثر المهام قيمةً ونفعًا ضمن عملية إصدار المعيار لمستخدم اللغة.

لا تقتصر جميع تطبيقات لغة سي على مجالٍ واحد طبعًا، إذ المكتبة القياسية مفيدة في تطبيقات معالجة البيانات، التي يكثر فيها استخدام ملفات الدخل والخرج والأرقام والبيانات على هيئة السلاسل النصية. تُستخدم لغة سي ضمن مجالٍ مساوٍ في الأهمية ألا وهو الأنظمة المدمجة embedded system، الذي يتطلب تطبيقات مختلفة، مثل التحكم بالعمليات وحوسبة الزمن الحقيقي وتطبيقات أخرى مشابهة.

يعي المعيار هذه الاستخدامات ويعمل على تزويدها بالأدوات اللازمة، إذ تشغل دوال المكتبات التي يجب أن تكون موجودة في البيئات المُستضافة جزءًا كبيرًا من المعيار؛ إذ تُعرف البيئات المستضافة بأنها البيئة التي تحتوي على جميع المكتبات القياسية، ويسمح المعيار باستخدام كلٍّ من البيئات المستضافة والبيئات المستقلة، ولكن مع التفريق بينهما. لكن من الذي قد يريد استخدام بيئة لا تحتوي على أي من المكتبات؟ يستخدمها أي شخص يريد كتابة "برمجيات مستقلة Standalone programs"، مثل نظم التشغيل والنظم

المدمجة مثل متحكمات الآلة، أو برامج تعريف العتاد Firmware، إذ تُعد كل هذه الاستخدامات مثالاً مناسباً على استخدام بيئةٍ مستقلة.

بينما تحتوي البيئة المستضافة على بعض الكلمات المفتاحية المحجوزة لمكتبة معينة، تتميز البيئة المستقلة بعدم وجود هذا النوع من القيود على الرغم من أن كتابة الشيفرة البرمجية باستخدام كلمات مفتاحية من مكتبات قياسية أمرٌ غير محبذ لأنه قد يكون مشتتاً للقارئ. سنتكلم بصورةٍ موسعة أكثر عن أسماء واستخدامات كل من دوال المكتبة في الفصل التاسع.

الاصطلاحات المطبعية

يهدف نمط الطباعة في هذا الكتاب لتحقيق أسلوب متناسق ضمن جميع الفصول عند استخدام المصطلحات المميزة أو التقنية، ونقصد بهذا الكلمات التي لها معنى معين في سياق لغة سي، مثل الكلمات المفتاحية المحجوزة وأسماء المكتبات، إذ ستجدها مكتوبة بصورةٍ مميزة، مثل "int" و "printf". ستُكتب المصطلحات المُستخدمة في هذا الكتاب المرتبطة بالمعيار وليس بلغة سي بضرورة الأمر بخط غامق إن لم تذكر مسبقاً وذلك للتشديد على المصطلحات المذكورة حديثاً؛ وستُستخدم الأحرف الكبيرة (الإنجليزية) بغض النظر عن موضع ورود اسم الدالة أو الكلمة المفتاحية في النص؛ أما علامات الاقتباس فهي مستخدمة من حينٍ إلى آخر إذا كان من الممكن ضياع معناها ضمن سياق الكلام وعدم الانتباه لبعض "الكلمات المميزة". كل أسلوب تنسيق مغاير لذلك قد يكون بسبب نزوة من القارئ أو خطأً كتابياً.

تسلسل الأفكار

يبدأ الكتاب بنظرةٍ عامة على الأجزاء الأساسية من اللغة التي ستسمح لك بكتابة برامج مفيدة بسرعة، ويلي هذه المقدمة تغطيةً دقيقةً عن تفاصيل أهملت في البداية، ويليه نقاشٌ عميق حول المكتبة القياسية. هذا يعني أنك ستتعلم جزءاً لا بأس فيه من اللغة أينما توقفت عن قراءة الكتاب وشعرت أنك وصلت للقدر الكافي من التفصيل. ربما ستجد الفصل الأول بطيء التقدم بعض الشيء، ولكن الأمر يستحق انتباهك وفهمك للمواضيع المذكورة ضمنه.

أمثلة عن بعض البرامج

جُرِّبت جميع الأمثلة المذكورة في هذا الكتاب على مصرّف متوافقٍ مع معيار سي الجديد، هذا يعني أن معظم الأمثلة صحيحة، إلا في حالة لم يكن المصرّف يتبع المعيار بصورةٍ خاطئة أو طبقت شيئاً لا يندرج تحت المعيار. ومع ذلك **بعض** الأخطاء ممكنة الحدوث، نرجو أن تكون صبوراً عند حدوثها.

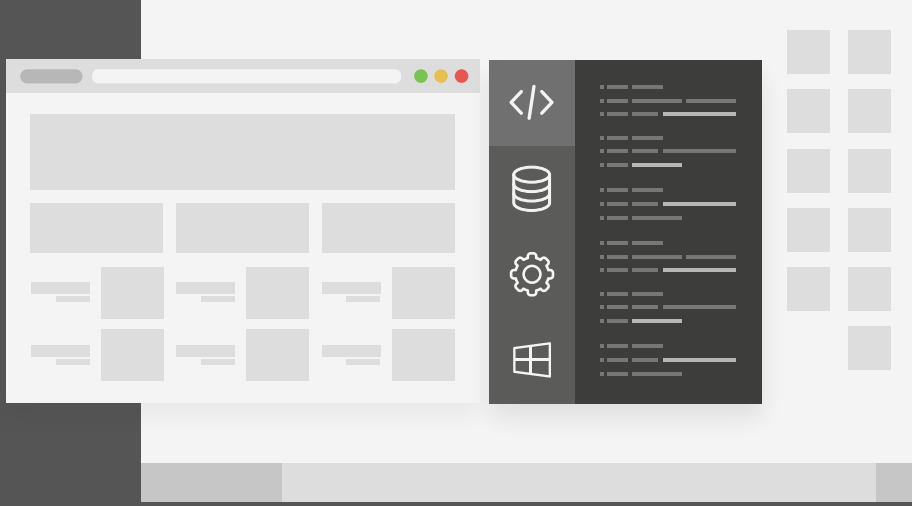
احترام المعيار

هذا الكتاب هو محاولة لتقديم شرح مقروء ومفهوم عن اللغة وفقاً لمعيارها، ويهدف لشرح ما قصده المعيار بلغة بسيطة. بذلنا جهدنا لنقل المعلومة على النحو الصحيح، ولكن عليك أن تعلم أن المصدر الوحيد الذي يعرّف ويصف اللغة هذه بالكامل هو المعيار بنفسه، فمن الممكن أننا أسأنا نقل ما أرادت اللجنة التي أشرفت على هذا المعيار بقوله، أو الطريقة التي نشرح بها بعض المفاهيم أكثر بساطة وعمامة أكثر من الشرح الموجود في المعيار، فإذا كان لديك أيّ شك **اقرأ المعيار**. نعلم أن قراءته ليست التجربة الأكثر متعة بالضرورة، لكنه دقيقٌ وبعيدٌ عن الغموض، ولا تأخذ المعلومة النهائية من أي مصدرٍ غيره.

المساهمة في النسخة العربية

يرجى إرسال بريد إلكتروني إلى academy@hsoub.com إذا كان لديك اقتراح أو تصحيح على النسخة العربية من الكتاب أو أي ملاحظة حول أي مصطلح من المصطلحات المستعملة. إذا ضمنت جزءاً من الجملة التي يظهر الخطأ فيها على الأقل، فهذا يسهّل علينا البحث، ويفضل أيضاً إضافة أرقام الصفحات والأقسام.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن



1. مقدمة إلى لغة سي

1.1 بنية برنامج لغة سي

إذا كنت معتادًا على لغات تتبع بنية الكتل مثل لغة باسكال Pascal، فستجد بنية برنامج لغة سي C مفاجئًا لك؛ وإذا كانت خبرتك في مجال لغات مشابهة للغة فورتران FORTRAN، فستجد البنية مشابهةً لما اعتدت عليه -بالرغم من اختلافها بصورة كبيرة في التفاصيل، وفي الحقيقة استعارت لغة سي من كلا الأسلوبين المُستخدمين بصورة واضحة، ومن أماكن أخرى أيضًا. نتيجةً لأخذ بعض القواعد من مصادر مختلفة، تبدو لغة سي أشبه بنتيجة تزاوج فصيلة كلاب ترير Terrier غير الأنيفة والمعروفة بعنادها وقوتها لكنها متسامحة مع أفراد العائلة. يطلق علماء الأحياء على هذا النوع من الفصائل "القوة الهجينة"، قد يذكرك كلامنا أيضًا بمخلوق كميير Chimera الأسطوري الذي يبدو خليطًا من الخرفان والماعز، قد يمنحنا الحليب والصوف، ولكنه سيزعجنا بثغائه المرتفع ورائحته غير اللطيفة.

إذا نظرنا للأمر عمومًا نلاحظ أن ميزة لغة سي C العامة هي بنية البرنامج الموزعة على عدة ملفات، لأنها تسمح بتصريف منفصل لهذه الملفات، إذ تسمح لغة سي بتوزيع أجزاء من برنامج مكتمل على عدة ملفات مصدرية والتصريف على نحو متفرق عن بعضها بعضًا. مبدأ العمل هنا هو أن جميع عمليات التصريف هذه ستعطينا ملفات يمكن ربطها Linked سويًا عن طريق أي محرر ربط، أو محمّل ربط يستخدمه نظامك، ولكن بنية الكتل لبعض لغات البرمجة المشابهة للغة ألغول ALGOL تجعل هذه الطريقة صعبة التنفيذ، نظرًا لأن البرنامج مكتوب بطريقة تجعل منه كتلة واحدة مترابطة، إلا أن هناك بعض الطرق للتغلب على هذه المشكلة.

تتغلب لغة سي على هذه المشكلة بطريقة مثيرة للاهتمام، إذ من المفترض أن تسرع عملية التصريف، لأن تصريف البرنامج إلى تعليمات مصرفة Object Code بطيء ومكلف من ناحية الموارد، فالتصريف عملية شاقة. أتت من هنا فكرة استخدام المحمّل في ربط مجموعة من التعليمات المصرفة، إذ تتطلب هذه العملية

ترتيب التعليمات حسب عنوانها للتوصل للبرنامج الكامل ببساطة. يُفترض أن يكون هذا الحل خفيف الاستهلاك للموارد، وعلى المحمّل طبقاً فحص **المكتبات** المستخدمة في التعليمات المصرفة واختيارها أيضاً.

الفائدة المكتسبة هنا هو أننا لسنا بحاجة تصريف كامل البرنامج بعد تعديل جزء بسيط منه، في هذه الحالة نحن بحاجة إعادة تصريف الجزء المعدّل من البرنامج فقط؛ ولكن قد يصبح المحمّل في بعض الأحيان أبطأ عمليات تصريف البرنامج وأكثرها استهلاكاً للموارد بزيادة العمل المطلوب منه، وقد تكون العملية أسرع في بعض الأنظمة إذا صُرّف كل شيء دفعةً واحدة، وتُعد لغة أدا Ada إحدى الأمثلة المعروفة باتباعها لهذا الأسلوب. بالنسبة للغة سي فالعمل المنجز من المصرف ليس ضخماً ومعقول إلى حدٍّ ما، يوضح الشكل 1 طريقة عمل المصرف في لغة سي.



الشكل 1: طريقة عمل المصرف في لغة سي

هذه الطريقة مهمة في لغة سي، إذ من الشائع أن تجد جميع البرامج باستثناء الصغيرة منها مؤلفةً من عددٍ من ملفات الشيفرة المصدرة، هذا يعني أيضاً أن جميع البرامج مهما كانت بسيطة ستمرّ بالمحمّل، نظراً لاعتماد لغة سي المكثف على المكتبات، وهذا ما قد يكون غير واضح عند النظرة الأولى أو للمتعلم الجديد.

1.2 الدوال Functions

تتكون لغة C من مجموعة عناصر تشكل لبنات البناء الأساسية لها، مثل **الدوال Functions** وما نطلق عليه تسمية **المتغيرات العامة global variables**، إذ تُسمى هذه العناصر في نقطة ما من البرنامج عند تعريفها، وتحتوي طريقة الوصول لهذه العناصر باستخدام اسمائهم ضمن البرنامج على بعض القواعد، وتُوصف هذه القواعد في المعيار بمصطلح **الربط Linkage**. سنتكلم في الوقت الحالي فقط عن **الربط الخارجي External Linkage** و**انعدام الربط No linkage**، إذ تُدعى العناصر التي يمكن الوصول إليها ضمن البرنامج كاملاً، مثل دوال مكتبة معينة، بعناصر الربط الخارجي، وتُستخدم العناصر عديمة الربط بكثرة أيضاً ولكن الوصول إليها محدودٌ بصورةٍ أكبر.

تُسمى المتغيرات المستخدمة داخل الدالة بالمتغيرات "المحلية Local" وهي عديمة الربط، وعلى الرغم من أننا نتفادى المصطلحات المعقدة قدر الإمكان في هذا الكتاب مثل المصطلحات التي ذكرناها سابقاً، ولكن لا توجد طريقة أبسط من شرح هذه المصطلحات. ستألف مصطلح الربط ضمن هذا الكتاب، والنوع الوحيد من الربط الخارجي الذي ستراه حالياً هو استخدام الدوال.

تكافئ الدوال في لغة سي الدوال والبرامج الفرعية في لغة فورتران FORTRAN والدوال والإجراءات في لغة باسكال Pascal وألغول ALGOL، بينما لا تمتلك لغة بيسك BASIC ومعظم طفراتها mutations البسيطة أو لغة كوبول COBOL مقدار الدوال التي تمتلكه لغة سي.

الفكرة من الدالة بسيطة وتتمثل بإعطائك الإمكانية بتضمين فكرة واحدة أو عملية ضمن قالب، وإعطائها اسمًا ما واستدعائها ضمن أجزاء مختلفة من برنامجك باستخدام اسمها فقط. تكون تفاصيل العملية هذه غير واضحة عند ذكر الاسم ضمن البرنامج، وهذا الأمر طبيعي. وفي البرامج المصممة المهيكلة بصورة جيدة، يمكنك تغيير طريقة عمل دالة ما (شرط ألا يغير ذلك من طبيعة العمل) دون أن تؤثر على الأجزاء الأخرى من البرنامج.

هناك دالة ذات اسم مميز في **البيئات المستضافة** ألا وهي دالة "main"، إذ تمثل هذه الدالة نقطة بداية البرنامج عند تشغيله، أما في **البيئات المستقلة** فالذي يحدد أولى خطوات البرنامج هي **دالة معرفة مسبقًا** **Implementation defined**؛ وهذا يعني أنه على الرغم من أن المعيار لا يحدد ما الذي سيحدث، إلا أن سلوك البرنامج يجب أن يكون محددًا وموثقًا. يتوقف البرنامج عندما يغادر دالة "main". ألق نظرة على البرنامج البسيط التالي الذي يحتوي على دالتين:

```
#include <stdio.h>

/*
 * Tell the compiler that we intend
 * to use a function called show_message.
 * It has no arguments and returns no value
 * This is the "declaration".
 */

void show_message(void);
/*
 * Another function, but this includes the body of
 * the function. This is a "definition".
 */
main(){
    int count;

    count = 0;
    while(count < 10){
        show_message();
    }
}
```

```

        count = count + 1;
    }

    return(0);
}

/*
 * The body of the simple function.
 * This is now a "definition".
 */
void show_message(void){
    printf("hello\n");
}

```

[تمرين 1.1]

1.3 شرح تمرين 1.1

1.3.1 ما الذي احتواه التمرين السابق؟

يمكن لمثال صغير جدًا أن يقدم لك الكثير عن لغة سي، إذ احتوى التمرين السابق على دالتين وتعليمة `#include`، وبعض التعليقات، بالإضافة لأشياء أخرى، وبما أن التعليق هو أبسط الأجزاء في البرنامج دعونا نبدأ به.

1.3.2 التنسيق والتعليق

تخطيط برنامج مكتوب بلغة سي ليس مهمًا للمصرف، ولكنه هام لنقل بعض المعلومات عن البرنامج للقارئ البشري ولتسهيل عملية قراءته، إذ تسمح لك لغة سي باستخدام محرف المسافة `space` ومسافات الجدولة `tabs` والسطور الجديدة `newline` دون أن تؤثر على عمل البرنامج. تُدعى المحارف الثلاث المذكور سابقًا باسم **المسافة الفارغة white space**، ولا يميّز بينها المصرف لأنها ببساطة تغير من موضع الكلمات دون التأثير "مرئيًا" على ما يُعرض على جهاز الخرج. يمكن أن نلاحظ المسافة البيضاء في أي مكان من البرنامج عدا وسط **المعرّفات Identifiers** و**السلاسل النصية Strings** و**المحارف الثابتة Character constants**؛ إذ نقصد بالمعرفات هنا اسم الدالة أو كائن `Object` آخر، لا تشغل بالك بالسلاسل النصية والمحارف الثابتة إذ سنناقشها في الفصول القادمة.

يُعد الفصل بين شيئين مختلفين من الحالات التي يصبح فيها استخدام المسافات الفارغة ضروريًا، إذ قد يتسبب تلاصقهما بتفسيرهما شيئًا آخر، مثل الجزء `void show_message`، إذ نحتاج لمسافة فارغة للفصل

بين الكلمتين `void` و `show_message`، لكن احتواء `show_message` على مسافة بيضاء بينها وبين القوس المفتوح غير ضروري، ووجودها مجرد أسلوب تنسيقي في كتابة الشيفرة لا أكثر.

يبدأ التعليق في لغة C باستخدام المحرفين `/*` / المتلاصقين دون أي فراغ بينهما، ويُعد أي شيء يأتي بعدهما إضافةً للمحرفين `/*` مسافةً فارغةً واحدة. لم يكن الأمر مماثلًا في معيار سي القديم، إذ كانت تنص القاعدة على إمكانية استخدام التعليق في أي مكان يمكن أن تُستخدم فيه المسافة الفارغة، أما الآن يُعد التعليق بحد ذاته مسافةً فارغةً؛ التغيير الحاصل طفيف وستكون الأمور أكثر وضوحًا في **الفصل السابع** عندما نناقش **المعالج المُسبق preprocessor**. يجعل تنسيق تعليقات لغة سي بهذا الشكل تضمين تعليق بداخل تعليق آخر غير ممكن، نظرًا لإغلاق أول زوج محارف `/*` في التعليق الثاني كتلة التعليق، وهذا إزعاج بسيط ستعتاد عليه لاحقًا.

من الممارسات الشائعة وضع زوج المحارف `/*` في بداية كل سطر من تعليق متعدد الأسطر لإبراز كلٍّ منهم، كما هو موضح في مثالنا السابق.

1.3.3 تعليمات المعالج المسبق

التعليمة الأولى في مثالنا السابق هي **موجه للمعالج المُسبق Preprocessor directive**، إذ تضمّن مصرّف لغة سي سابقًا مرحلتين، هما **المعالجة المُسبقة** ومرحلة التصريف الاعتيادية؛ إذ تمثل المعالجة المُسبقة **معالج ماكرو macro processor** تجري بعض عمليات التلاعب النصي البسيطة على البرنامج قبل أن تمرّ النص الناتج إلى المصرّف، وقد أصبحت عملية المعالجة المسبقة جزءًا أساسيًا من عمل المصرّف ولهذا تُعد جزءًا لا يتجزأ من اللغة.

على الرغم من أن لغة سي حساسة لما يقع في نهاية الأسطر إلا أن عملية المعالجة المسبقة تلاحظ الأسطر فقط، ومع ذلك من الممكن كتابة موجه معالجة مسبقة متعدد الأسطر، ولكنه غير شائع، وستشعر بالغرابة عندما تجد هذه الطريقة متبعة. يُعد أي سطر يبدأ بالمحرف `#` توجيهًا للمعالج المسبق.

في التمرين 1.1 يؤدي موجه المعالجة المسبقة `#include` إلى تبديل السطر هذا بمحتويات ملف آخر، وفي هذه الحالة اسم الملف يوجد ما بين القوسين `<>`، وهذه طريقة شائعة لتضمين محتوى **ملفات الترويسات Header files** القياسية ضمن برنامجك، دون الاضطرار لكتابتها بنفسك. يحتوي ملف `<stdio.h>` المهم العديد من المعلومات الضرورية التي تسمح لك باستخدام مكتبة الدخل والخرج القياسية، بهدف الحصول على الدخل وإظهار الخرج، فإذا أردت استخدام هذه المكتبة عليك أن **تتأكد** من وجود `<stdio.h>`. كان معيار سي السابق متساهلاً أكثر بهذا الشأن.

1. تعليمات التعريف Define

تُعدّ تعليمة `#define` من الإمكانيات الأخرى والمستخدمَة كثيرًا للمُعالج المُسبق، إذ تُستخدم على النحو التالي:

```
#define IDENTIFIER replacement
```

تعني التعليمة السابقة أنه على المُعالج المُسبق استبدال جميع الكلمات المطابقة إلى `IDENTIFIER` بالكلمة `replacement` عند جميع نقاط ورودها ضمن البرنامج. يُكتب دائمًا المعرّف بأحرف كبيرة `IDENTIFIER`، وهذا اصطلاحٌ برمجي يساعد قارئ الشيفرة على فهمها، أما الجزء المُبدّل به `replacement` فقد يكون أي سلسلة نصية. تذكر أن المُعالج المُسبق لا يعرف قواعد لغة سي، إذ أن مهمته هي التلاعب النصي فقط. يكون الاستخدام الأكثر شيوعًا لهذه التعليمة هو التصريح `Declare` عن أسماء القيم العددية الثابتة كما هو موضح فيما يلي:

```
#define PI 3.141592
#define SECS_PER_MIN 60
#define MINS_PER_HOUR 60
#define HOURS_PER_DAY 24
```

واستخدام القيم على النحو التالي:

```
circumf = 2*PI*radius;
if(timer >= SECS_PER_MIN){
    mins = mins+1;
    timer = timer - SECS_PER_MIN;
}
```

سيكون الخرج الناتج عن المُعالج المُسبق مماثلًا فيما لو كتبت الشيفرة التالية:

```
circumf = 2*3.141592*radius;
if(timer >= 60){
    mins = mins+1;
    timer = timer - 60;
}
```

أخيرًا، نستطيع تلخيص ما سبق على النحو التالي:

- تتعامل تعليمات المُعالج المُسبق مع الملفات النصية سطرًا بسطر، على نقيض لغة سي.

- تُستخدم التعليمات من النوع `#include` لقراءة محتوى من ملفٍ معين، عادةً لتسهيل التعامل مع دوال مكتبةٍ ما.
- تُستخدم تعليمات `#define` لتسمية الثوابت، وتُكتب الأسماء اصطلاحًا بحروفٍ كبيرة.

1.3.4 تعريف وتصريح الدالة

أ. التصريح

نلاحظ وجود ما يُسمى **تصريح الدالة function declaration** بعد تضمين ملف `<stdio.h>`، الذي يخبر المصرّف أن `show_message` دالة لا تأخذ أي وسيط ولا تُعيد أي قيمة، ويوضح لنا هذا تغييرًا جرى على المعيار، ألا وهو **النموذج الأولي للدالة function prototype**، وسنناقش هذا الموضوع بتوسّع لاحقًا. ليس من الضروري التصريح عن الدالة مسبقًا، إذ ستستخدم لغة سي بعض القواعد القديمة الافتراضية في هذه الحالة، إلا أنه ينصح بشدة التصريح عن الدالة في البداية.

الفرق بين **التصريح والتعريف** هو أن التصريح يصف نوع الدالة والوسطاء المُمرّرة له، بينما يحتوي التعريف على بنية الدالة بالكامل. سيهّمنا فهم الفرق بين المصطلحين لاحقًا.

يستطيع المصرّف تفقد استعمال الدالة `show_message` فيما إذا كان صحيحًا أم لا بالتصريح المُسبق عنها قبل استخدامها، ويصف التصريح ثلاث خصائص عن الدالة، هي: اسمها ونوعها وعدد الوسطاء ونوعهم؛ إذ يشير الجزء `void show_message` إلى نوع الدالة والقيمة التي تُعيدها بعد استدعائها وهي `void` (سنناقش معناها لاحقًا). نستطيع رؤية الاستخدام الثاني لكلمة `void` في قائمة الوسطاء للدالة `(void)`، والذي يعني أن الدالة لا تقبل أي وسطاء.

ب. التعريف

تلاحظ في نهاية البرنامج تعريف الدالة، وبالرغم من أنّ أن طولها ثلاث أسطر فقط، إلا أنها تُعد مثالًا على تعريف دالة متكامل.

تنفّذ دوال لغة سي المهام التي قد تقسّمها بعض اللغات الأخرى إلى جزأين، إذ تستخدم لغات البرمجة الدوال لإعادة قيمةٍ ما، مثل دالة الجيب المثلثي `sin` وجيب التمام `cos` أو ربما دالة تُعيد الجذر التربيعي لعددٍ ما، وهذه الطريقة التي تعمل بها دوال لغة سي، بينما تجري بعض لغات البرمجة هذه العملية باستخدام ما يشبه الدوال ولكن الفرق هنا هو عدم إعادة القيمة، مثل استخدام فورتران للبرامج الفرعية واستخدام باسكال `Pascal` وألغول `Algol` للإجراءات. تنجز لغة سي كل هذه المهام باستخدام الدوال عن طريق تحديد **نوع** القيمة المُعادَة عند التصريح عن الدالة، ولا تُعيد الدالة `show_message` أي قيمة، لذلك نوعها `void`.

إما أن يكون استخدام القيمة `void` بديهيًا لك، أو صعب الفهم حسب الطريقة التي تنظر لها للأمر. ففي الحقيقة، يمكننا الدخول في نقاشات فلسفيّة جانبية وغير مثمرة عن كون `void` يصف نوع قيمةٍ أو لا، لكن

أفضل تجنب ذلك. بغض النظر عن رأيك، استخدام النوع `void` التي تعني: "أنا لا أهتم بأي قيمة ترجعها هذه الدالة (أو لا ترجعها)".

إذًا، نوع الدالة هو `void` واسمها `show_message`، أما القوسان () اللذان يتبعان هذه المعلومات، فهما لتنبية المصرّف أننا نقصد التعريف، أو التصريح عن دالة. إذا كانت الدالة تقبل أي وسيط، فيجب وضع اسمه داخل القوسين. الدالة التي نتكلم عليها في مثالنا لا تأخذ أي وسطاء وهذا الأمر موضح عن طريق استعمال الكلمة `void` بداخل القوسين. وبالتالي أتضح أن للكلمة التي تصف الفراغ والرفض أهمية بالغة.

يشكّل متن الدالة **تعليلة مركبة compound statement**، وهي مجموعة من التعليمات المحاطة بأقواس معقوفة `{}`، على الرغم من وجود تعليلة واحدة فقط إلا أن الأقواس مطلوبة، وعمومًا، تسمح لك لغة سي باستخدام تعليلة مركبة في أي مكان تسمح به عادةً باستخدام تعليلة واحدة بسيطة، وتهدف الأقواس المعقوفة لتحويل عدة تعليمات متتالية إلى تعليلة واحدة.

إذا سألت السؤال المبرّر "هل يتوجب استخدام الأقواس المعقوفة في كل مكان، إذا كان الهدف منها جمع عدّة تعليمات لتعليلة واحدة؟" الإجابة: نعم، **عليك** استخدام الأقواس المعقوفة، والمكان الوحيد الذي لا تستطيع فيه استخدام تعليلة واحدة عوضًا عن تعليلة مركبة هو عند تعريف دالة ما. بالتالي، أبسط دالة يمكننا إنشاؤها هي دالة فارغة، لا تفعل أي عمل إطلاقًا:

```
void do_nothing(void){}
```

التعليلة الموجودة بداخل دالة `show_message` هي استدعاء لدالة من مكتبة `printf`، إذ تُستخدم هذه الدالة لتنسيق وطباعة الأشياء، والاستخدام الموجود في المثال هو أبسط استخدامات هذه الدالة. تقبل الدالة `printf` وسيطًا واحدًا أو أكثر، التي تُمرّر قيمهم من استدعاء الدالة إلى الدالة نفسها، في مثالنا هذا، الوسيط هو **السلسلة النصية**. يُفسّر محتوى السلسلة النصية من الدالة `printf` وتتحكم الدالة بطريقة عرض الخرج حسب الوسطاء الممرّرة له، يماثل عمل الدالة عمل تعليلة `FORMAT` في فورتران إلى حدّ ما.

خُلاصة القول:

- تُستخدم **التصريحات** في التصريح عن اسم الدالة ونوع القيمة المُعادة ونوع قيم الوسطاء إن وُجدت.
- **تعريف** الدالة هي تصريح للدالة مع محتواها أيضًا.
- يُصرّح عن الدالة التي لا تعيد أي قيمة باستخدام الكلمة `void`، على سبيل المثال:

```
void func(/* list of arguments */);
```

- يجب أن يصرّح عن الدالة التي لا تأخذ أي وسيط باستخدام الكلمة `void` ضمن لائحة الوسطاء، على سبيل المثال:

```
void func(void);
```

1.3.5 السلاسل النصية

تُعرّف السلاسل النصية في لغة سي بأنها سلسلة من المحارف المحتواة داخل علامتي تنصيص على النحو التالي:

```
"like this"
```

من غير المسموح أن تشغّر السلسلة النصية عدة أسطر، إذ تُعدّ عنصرًا واحدًا بصوريّة مشابهة للمعرفات. إلا أنه من الممكن استخدام محرّف المسافة ومسافة الجدولة داخل السلسلة النصية. يوضح المثال أدناه سلسلة نصيةً صالحةً وأخرى غير صالحة في السطرين الثاني والثالث:

```
"This is a valid string"
"This has a newline in it
and is NOT a valid string"
```

يمكنك الحصول على سلسلة نصية طويلة جدًا بالاستفادة من حقيقة أن المحرف "" في نهاية السطر يختفي تمامًا ضمن برنامج سي عند التنفيذ:

```
"This would not be valid but doesn't have \
a newline in it as far as the compiler is concerned"
```

الحل الثاني هو باستخدام مِيّزة ضم السلاسل النصية، التي تنظر لكل سلسلتين نصيتين متجاورتين على أنهما سلسلة نصية واحدة:

```
"All this " "comes out as "
"just one string"
```

بالعودة للمثال 1.1، تُعدّ السلسلة النصية "n" مثالاً على ما يُدعى **محرّف التهريب escape character** ويمثّل في حالتنا هذه حالة إنشاء سطر جديد، وستطبع الدالة printf محتوى السلسلة النصية على ملف برنامج الخرج، بحيث سيكون الخرج "Hello" متبوعًا بسطر جديد.

يسمح المعيار الجديد باستخدام **المحارف ذات البايتات المتعدّدة multibyte characters** لدعم الأشخاص العاملين بيئة تستخدم مجموعة محارف أوسع من معيار آسكي ASCII الأميركي، مثل معيار Shift JIS المُستخدم في اليابان. يعرّف المعيار الجديد 96 محرّفًا تمثّل أبجدية لغة سي (المتطرّق لها في فصل أبجدية لغة C)، وفي حال كان نظامك يستخدم مجموعة محارف موسّعة extended، فسيكون المكان الوحيد الذي قد

تستخدمها هو بداخل سلسلة نصية، أو متغيرات من نوع محرف، أو ضمن التعليقات وأسماء **ملفات الترويسة Header files**. ينبغي عليك تفقد ملفات توثيق نظامك إذا أردت استخدام ميزة دعم المحارف الموسعة.

1.3.6 دالة main

يحتوي المثال 1.1 على دالتين، هما: دالة `show_message` ودالة `main`؛ فإذا صرفنا النظر عن طول دالة `main` موازنةً مع الدالة `show_message`، فسلاحظ أن الدالتين مبنيتان بالشكل نفسه، إذ تحتوي كلا الدالتين على اسم وقوسين () متبوعين بقوس معقوص، وتعلیمیة مركبة محتواة داخل القوسين المعقوصين تتبّع تعريف الدالة. على الرغم من أن الدالة المركبة مختلفة عن الدالة الأخرى، إلا أنك ستجد قوس الإغلاق المعقوص نفسه { الذي يتماشى مع القوس الأول }.

يُعد هذا المثال دالةً حقيقيةً يمكن استخدامها في التطبيقات الواقعية، إذ تحتوي على عدة تعليمات ضمن متن الدالة بدلاً من تعليمة واحدة في الدالة السابقة، ولربما لاحظت أيضًا أن الدالة غير مصرّح عنها باستخدام الكلمة `void`، لأن الدالة في الحقيقة تمرّر قيمة معينة، ولكننا سنتكلم عنها لاحقًا.

الشيء المميز بخصوص دالة `main` هو كونها أول دالة تُستدعى عند تنفيذ البرنامج في بيئة مستضافة، إذ تستدعي لغة سي الدالة `main` أولًا عند تشغيل البرنامج، وهذا هو السبب في تسمية الدالة بهذا الاسم (`main` تعني رئيس)، وعلى هذا فهي دالة هامة، ولكن محتوى الدالة هام بقدر مساوٍ، وكما ذكرنا سابقًا يمكن أن تحتوي الدالة على عدّة تعليمات بداخل التعليمة المركبة، دعونا نأخذ نظرة على هذه التعليمات.

1.3.7 التصريح Declaration

التعليمة الأولى هي:

```
int count;
```

وهي ليست تعليمة لتنفيذ أمرٍ ما، بل تعلن عن متغيرٍ جديد ضمن البرنامج، واسمه `count` من نوع "عدد صحيح integer" والكلمة التي تحدد هذا النوع من المتغيرات ضمن برنامج سي مُختصرة إلى الكلمة `int`. لا تدل جميع أسماء أنواع المتغيرات على نوعها بوضوح في لغة سي، إذ يُستخدم اسم نوع المتغير في بعض الأحيان مثل كلمة مفتاحية مختصرة وفي أحيان أخرى يُكتب كاملاً. لحسن الحظ يمكن تخمين نوع البيانات للكلمة `int` بقراءتها.

بفضل هذه التعليمة، يعلم المصرّف الآن أن هناك متغيرًا باسم `count` لتخزين قيم الأعداد الصحيحة. يجب التصريح عن جميع المتغيرات قبل استخدامها في لغة سي، على نقيض لغة FORTRAN، التي يجب أن يأتي التصريح قبل استخدام المتغير ضمن أي تعليمة.

يُعد التصريح عن المتغير count **تعريفًا** عنه في ذات الوقت، وسنناقش الفرق بين الاثنين وسنلاحظ أن الفرق مهم.

1.3.8 تعليمة الإسناد Assignment Statement

بالانتقال إلى السطور التالية، نلاحظ **تعليمة الإسناد assignment statement**، وهي التعليمة التي أسندت أول قيمة للمتغير count (القيمة هي صفر في حالتنا هذه). كانت قيمة المتغير count قبل تعليمة الإسناد غير معرّفة undefined وغير آمنة الاستخدام، وربما تتفاجئ بحقيقة أن رمز الإسناد -أي **عامل الإسناد assignment operator**- يُمثّل بإشارة مساواة واحدة =، وهذا التمثيل مستخدمٌ في معظم لغات البرمجة الحديثة، ولكنه ليس بمشكلة كبيرة.

إدًا، بالوصول لهذه النقطة في برنامجنا نكون قد صرّحنا عن متغير وأسندنا له قيمة الصفر، ماذا بعد؟

1.3.9 تعليمة الحلقة التكرارية While

سنتكلم عن واحدة من تعليمات التحكم بالحلقات، ألا وهي تعليمة while لنلقي نظرةً على شكلها العام:

```
while(expression)
    statement
```

هل هذا كل ما هنالك؟ نعم، هذه بنية تعليمة while، الجزء التالي في مثالنا هو:

```
count < 10
```

ويُدعى باسم **التعبير العلائقي relational expression**، وهو مثال عن إحدى التعبيرات الصالحة الممكن استخدامها في جملة الحلقة التكرارية -مكان expression- ويُتبع التعبير بتعليمة مركّبة، وهي مثال عن التعليمات الصالحة أيضًا الممكن استخدامها -مكان statement-. وهكذا، نستطيع تشكيل تعليمة while صحيحة.

إذا برمجت برنامجًا في السابق، فستلاحظ أن متن الحلقة سينقذ بصورةً متكررة طالما أن التعبير `count < 10` صحيح؛ وإذا أردنا برمجة حلقةٍ منتهية، فعلينا كتابة تعليمة ما تتسبّب بجعل التعبير السابق خاطئًا في مرحلة من المراحل، وهذا موجود في مثالنا فعلاً.

هناك تعليمتان فقط داخل متن الحلقة، إذ تمثّل الأولى استدعاءً لدالة `show_message` وتُكتب تعليمة استدعاء الدالة منتهيةً بقوسين () تحتويان ضمنهما لأحة الوسطاء؛ فإذا لم تحتوي الدالة على أي وسيط، نكتب قوسين فارغين؛ وإذا احتوت على وسيط أو عدة وسطاء نكتب ذلك ضمن القوسين على النحو التالي:

```
/* call a function with several arguments */
function_name(first_arg, second_arg, third_arg);
```

استدعاء الدالة `printf` هو مثال آخر، وسنشرح المزيد عن هذا الموضوع في **الفصل الرابع**.

تمثل التعليمات الأخيرة ضمن الحلقة تعليمة إسناد تضيف الرقم واحد إلى قيمة المتغير `count`، لكي نتوصل في نقطة ما إلى قيمة تجعل من التعبير الخاص بالحلقة خاطئًا.

1.3.10 تعليمة الإعادة `return`

التعليمة الأخيرة المتبقية في المثال هي تعليمة `return`، وتبدو للوهلة الأولى أنها استدعاءً لدالةٍ ما، ولكن التعليمة تُكتب على النحو التالي:

```
return expression;
```

والتعبير `expression` اختياري، إذ يتتبع مثالنا تنسيقًا شائعًا ألا وهو وضع التعبير ضمن قوسين، ولكن الأقواس غير ضرورية ولا تؤثر على عمل التعليمة.

تعيد تعليمة `return` قيمةً من الدالة الحالية الواقعة ضمنها إلى مستدعي `caller` الدالة؛ فإذا لم تُرَوّد التعليمة بالتعبير `expression` فستعاد قيمةً غير معروفة `unknown` إلى المستدعي، وهو أمرٌ خاطئ، عدا حالة إعادة `void`.

الدالة `main` غير مصرّح بنوع خاص بها على الإطلاق على عكس دالة `show_message`، فأَيُّ نوع من القيم تعيدها هذه الدالة؟ الإجابة هي `int`، إذ تفترض لغة سي نوع البيانات المُعاد `int` في حالة عدم تخصيص النوع بوضوح، لذا من الشائع أن تجد الدالة `main` مكتوبةً بهذه الطريقة:

```
int main(){
```

وهو تعبير مماثل لما ورد في مثالنا، ويعطي النتائج نفسها، لكنك لا تستطيع استعمال هذه الميزة للحصول على نوع اعتيادي للمتغيرات بالطريقة نفسها، ويجب أن تصرّح عن نوعها بوضوح.

لكن ما الذي تعني القيمة المُعادة من الدالة `main` وإلى أين تُرسل؟ كانت القيمة تُرسل إلى نظام التشغيل أو الجهة التي شغلت البرنامج في لغة سي القديمة، وفي البيئات المشابهة لنظام يونيكس UNIX، كانت القيمة "0" تعني "نجاح" العملية، بينما يدلّ على "فشل" العملية أي رقم آخر (غالبًا "1-"). حافظ المعيار عند قدومه على هذا التقليد، إذ يدل "0" على التنفيذ الناجح للبرنامج، ولكن هذا **لا يعني** أن القيمة "0" تُمرّر إلى البيئة المُضيفة، بل تُمرّر القيمة المناسبة للدلالة على نجاح البرنامج ضمن هذا النظام. يسبب هذا الكثير من الالتباس، لذا قد تحبذ استخدام القيمتين المعرفتين "EXIT_SUCCESS" و "EXIT_FAILURE" ضمن ملف الترويسة `<stdlib.h>`.

يمثل استخدام التعليمة `return` ضمن الدالة `main` استخدام الدالة `exit` مزودًا بالقيمة المُعادة وسيطًا، والفرق هنا أن الدالة `exit` قد تُستدعى من **أي مكان** ضمن البرنامج، وأن توقف البرنامج في النقطة التي

استُديت فيها بعد إنجاز بعض العمليات النهائية. إذا أردت استخدام الدالة `exit`، **يجب** عليك تضمين ملف الترويسة `<stdlib.h>`، ومن هذه اللحظة فصاعدًا سنستخدم `exit` بدلًا من استخدام `return` ضمن الدالة `.main`.

1.3.11 الملخص

تُعيد الدالة `main` قيمةً من نوع `int`.

يُمائل استخدام التعليمة `return` ضمن الدالة `main` استدعاء الدالة `exit`، الفرق هنا أنه من الممكن استدعاء `exit` في أي نقطة ضمن البرنامج.

إعادة القيمة 0 أو `EXIT_SUCCESS` يعني نجاح البرنامج، بينما تُعد أي قيمة أخرى فشلًا للبرنامج. مع أنَّ المثال المُناقش قصير، إلا أنه سمح لنا بمناقشة العديد من مزايا اللغة المهمة، وهي:

- بنية البرنامج `Program structure`.
- التعليق `Comment`.
- تضمين الملفات `File inclusion`.
- تعريف الدوال `Function definition`.
- التعليمات المركبة `Compound statements`.
- استدعاء الدوال `Function calling`.
- التصريح عن المتغيرات `Variable declaration`.
- العمليات الحسابية `Arithmetic`.
- الحلقات التكرارية `Looping`.

لكننا طبعًا لم نقاش هذه المواضيع بتعمق بعد.

1.4 بعض البرامج البسيطة بلغة سي

بما أننا ما زلنا في مرحلة التكلم عن لغة سي عمومًا، دعونا نتناول مثالين آخرين، إذ ستجد أن بعض التعليمات الموجودة ضمن هذين المثالين أصبحت معروفةً بالنسبة لك ولن نتطرق إليها، ولكننا سنتكلم عن المزايا والتعليمات الجديدة التي لم نشرحها بعد عن اللغة.

1.4.1 برنامج لإيجاد الأعداد الأولية

```
/*
 *
 *
 * Dumb program that generates prime numbers.
 */
#include <stdio.h>
#include <stdlib.h>

main(){
    int this_number, divisor, not_prime;

    this_number = 3;

    while(this_number < 10000){
        divisor = this_number / 2;
        not_prime = 0;
        while(divisor > 1){
            if(this_number % divisor == 0){
                not_prime = 1;
                divisor = 0;
            }
            else
                divisor = divisor-1;
        }

        if(not_prime == 0)
            printf("%d is a prime number\n", this_number);
        this_number = this_number + 1;
    }
    exit(EXIT_SUCCESS);
}
```

[مثال 2.1]

هناك الكثير من الأمور الجديدة المثيرة للاهتمام في هذا البرنامج. أولاً، البرنامج يعمل بغناء بعض الشيء، إذ يقسم العدد على جميع الأرقام الواقعة في المجال بين نصف قيمته والرقم 2 للتحقق ما إذا كان أولياً أم لا، وإذا كان هناك أي قسمة دون باقي فهذا يعني أنه ليس أولياً. هذه المرة الأولى التي نستخدم فيها كلاً من العامل % وعامل المساواة المُمثل بإشارتي مساواة ==، إذ يتسبب هذا العامل بكثير من الأخطاء في برنامجك عادةً. نظراً للخلط بينه وبين عامل الإسناد =، فكن حذراً.

المشكلة في فحص المساواة هذا وأيّ فحص آخر، هي أن استبدال == بالرمز = هو تعليمة صالحة، ففي الحالة الأولى (استخدام ==) تُوازَن القيمتان وتُفحص حالة المساواة، كما في المثال التالي:

```
if(a == b)
while (c == d)
```

كما يمكن استخدام عامل الإسناد = في الموضع ذاته، ولكن لإسناد القيمة الواقعة على يسار الإشارة للمتغير على يمينها. تصبح المشكلة مهمة أكثر إذا كنت معتاداً على البرمجة باللغات التي تستخدم عامل المساواة بصورة مماثلة لما تستخدمه لغة سي لعامل الإسناد، إذ لا يوجد أي شيء يمكنك فعله لتغيير ذلك سوى الاعتياد على هذا الأمر من الآن فصاعداً، ولحسن الحظ تعطيك بعض المصرفات الحديثة بعض التحذيرات عندما تلاحظ استخداماً غريباً لعامل الإسناد في غير موضعه المعتاد، ولكن يصبح الأمر مزعجاً أحياناً إذا كان هذا فعلاً ما تريد فعله وليس خطأً.

نلاحظ في مثالنا السابق أيضاً أول استخدام للتعليمة if، إذ تفحص هذه التعليمة تعبيراً expression موجوداً داخل القوسين على نحوٍ مشابه لتعليمة while، إذ تتطلب جميع التعليمات الشرطية التي تتحكم بتدفق البرنامج تعبيراً محتوياً داخل قوسين بعد الكلمة المفتاحية، وفي هذه الحالة الكلمة المفتاحية هي if. تُكتب التعليمة if على النحو التالي:

```
if(expression)
    statement

if(expression)
    statement
else
    statement
```

يوضّح المثال المبين أعلاه أن التعليمة تأتي بنموذجين، إذ يحتوي الأول على الكلمة if والثاني على الكلمة else؛ فإذا كان التعبير صحيحاً وفقاً للنموذج الأول، فستنفذ مجموعة التعليمات الموجودة داخل متن if، أما إذا كان التعبير خاطئاً فلن تُنفذ؛ بينما تنفذ التعليمات الموجودة ضمن else وفقاً للنموذج الثاني فقط في حالة كان التعبير خاطئاً (أي تعبير if السابق لها).

يتميز استخدام تعليمة `if` بمشكلة شائعة، ألق نظرةً على المثال التالي وأجب، هل ستُنَفَّذ التعليمة

statement-2 أم لا؟

```
if(1 > 0)
    if(1 < 0)
        statement-1
else
    statement-2
```

الإجابة هي **نعم** ستُنَفَّذ التعليمة. لا تركّز تفكيرك على مسافة الإزاحة فهي مضللةٌ غالبًا، فقد تكون تعليمة `else` تابعةً لتعليمة `if` الأولى أو الثانية بحسب وصف كلٍّ منهما، لذا علينا اللجوء لقاعدةٍ ما لجعل الأمور أوضح. القاعدة ببساطة هي أن `else` تابعةً لتعليمة `if` الأقرب (التي تسبق `else`)، والتي لا تحتوي على تعليمة `else` مسبقًا. لنجعل البرنامج يعمل كما نريد وفق تنسيق مسافات الإزاحة، علينا إنشاء تعليمة مركبة باستخدام الأقواس المعقوفة كما يلي:

```
if(1 > 0){
    if(1 < 0)
        statement-1
}
else
    statement-2
```

تتبع لغة سي -على الأقل في هذه الحالة- لعمل معظم لغات البرمجة واصطلاحاتهم. في الحقيقة، قد يشعر بعض القراء أن هذه القاعدة "بديهية" إن سبق وتعاملوا مع لغة برمجة مشابهة للغة سي وأنها لا تستحق الذكر. لنأمل أن يفكر الجميع بهذه الطريقة.

1.4.2 عامل القسمة

يُشار إلى عامل القسمة بالرمز `/`، وعامل باقي القسمة بالرمز `%`. تفعل عملية القسمة المتوقع منها، إلا أن إجراء القسمة على أعداد صحيحة `int` ستعطي نتيجةً مقربةً باتجاه الحد الأدنى (الصفر)، إذ تعطي العملية `5/2` مثلًا النتيجة 2، وتعطي التعليمة `5/3` النتيجة 1؛ وللحصول على القيمة المقطعة من الناتج نستخدم عامل باقي القسمة، إذ تعطينا العملية `5%2` النتيجة 1، والعملية `5%3` النتيجة 2. تعتمد إشارة باقي القسمة وناتج القسمة على المقسوم والمقسوم عليه وهي مُعرّفة في المعيار وسنناقشها لاحقًا في الفصل الثاني.

1.4.3 مثال عن تنفيذ عملية الدخل

من المفيد أن نكون قادرين على الحصول على الدخل وكتابة برامج قادرة على طباعة وعرض النتائج ضمن قوائم أو جداول في الخرج، والطريقة الأبسط لتنفيذ هذا هي عن طريق استخدام الدالة `getchar`، وهي الطريقة الوحيدة التي سنناقشها حالياً؛ إذ تقرأ هذه الدالة كلَّ محرف على حدة من دخل البرنامج وتعيد قيمةً عدديةً `int` تمثِّله، ويمكن استخدام هذه القيمة الممثلة للمحرف في طباعة المحرف ذاته في خرج البرنامج، ويمكن أيضاً موازنة هذا المحرف (القيمة) مع محارف معيّنة أو محارف قُرأت مسبقاً، إلا أن الاستخدام الأكثر منطقية هو موازنة المحرف المُدخل مع محرف معيّن.

لا تُعد موازنة قيمة المحارف ما إذا كانت أصغر أو أكبر خياراً جيّداً، إذ لا يوجد أي ضمان بأن قيمة `a` أصغر من قيمة `b`، مع أن ذلك الأمر محقق في معظم الأنظمة، ولكن الضمان الوحيد هنا الذي يقدمه لك المعيار هو أن القيمة ستكون متتابعة من 0 إلى 9. ألق نظرةً على المثال التالي:

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int ch;

    ch = getchar();
    while(ch != 'a'){
        if(ch != '\n')
            printf("ch was %c, value %d\n", ch, ch);
        ch = getchar();
    }
    exit(EXIT_SUCCESS);
}
```

[مثال 3.1]

هناك ملاحظتان جديرتان بالاهتمام، هما:

- سنجد في نهاية كل سطر دخل المحرف `\n` (محرف ثابت)، وهو المحرف ذاته الذي نستخدمه في دالة `printf` عندما نريد طباعة سطر جديد. نظام الدخل والخرج الخاص بلغة سي غير مبني على مفهوم الأسطر بل على مفهوم المحارف؛ فإذا كنت تريد التفكير بالأمر من منطلق مفهوم الأسطر، فانظر للمحرف `\n` على أنه إعلان لنهاية السطر.

- استخدام %c لطباعة المحرف بواسطة الدالة printf، إذ يسمح لنا هذا الأمر بطباعة المحرف على أنه محرف على الشاشة، بالموازنة مع استخدام %d الذي سيطبع المحرف ذاته ولكن بتمثيله العددي المُستخدم ضمن البرنامج.

إذا جربت تنفيذ هذا البرنامج بنفسك، قد تجد أن بعض الأنظمة لا تمرّر المحرف الواحد تلو الآخر عند كتابته، بل تجبرك على كتابة سطر كاملٍ للدخل أولاً، ثم تبدأ معالجة المحرف الواحد تلو الآخر. قد يبدو الأمر مشوّشاً لبعض المبتدئين عندما يكتبون محرّفًا ما ولا يحدث شيء بعد ذلك، وليس للغة سي أي علاقةٍ بذلك، بل يعتمد الأمر على حاسوبك ونظام تشغيله.

1.4.4 المصفوفات البسيطة

يكون غالبًا استخدام **المصفوفات arrays** في لغة سي للمبتدئين بمثابة تحدٍ، إذ أن التصريح عن المصفوفات ليس صعبًا، بالأخص المصفوفات أحادية البعد one-dimensional، ولكن سبب الأخطاء هنا هو بدء الدليل index الخاص بها من الرقم 0. للتصريح عن مصفوفة مؤلفة من 5 أعداد من نوع int، نكتب:

```
int something[5];
```

تستخدم لغة سي الأقواس المعقوفة square brackets للتصريح عن المصفوفات، ولا تدعم أيّ مصفوفة لا تقع أدلتها بين 0 وما فوق. العناصر الصالحة في مثالنا هي something[0] إلى something[4]، و something[5] **غير موجود** في المصفوفة وهو عنصر غير صالح.

يقرأ البرنامج التالي المحارف من الدخل، ويرتبها وفقًا لقيمتها العددية، ويطبعها في الخرج مرةً أخرى. ألقِ نظرةً على البرنامج وحاول فهم ما يحصل، إذ لن نتكلم عن الخوارزمية مفصلاً في شرحنا.

```
#include <stdio.h>
#include <stdlib.h>
#define ARSIZE 10
main(){
    int ch_arr[ARSIZE],count1;
    int count2, stop, lastchar;
    lastchar = 0;
    stop = 0;
    /*
     * Read characters into array.
     * Stop if end of line, or array full.
     */
    while(stop != 1){
```



```
    ch_arr[lastchar] = getchar();
    if(ch_arr[lastchar] == '\n')
        stop = 1;
    else
        lastchar = lastchar + 1;
    if(lastchar == ARSIZE)
        stop = 1;
}
lastchar = lastchar-1;

/*
 * Now the traditional bubble sort.
 */
count1 = 0;
while(count1 < lastchar){
    count2 = count1 + 1;
    while(count2 <= lastchar){
        if(ch_arr[count1] > ch_arr[count2]){
            /* swap */
            int temp;
            temp = ch_arr[count1];
            ch_arr[count1] = ch_arr[count2];
            ch_arr[count2] = temp;
        }
        count2 = count2 + 1;
    }
    count1 = count1 + 1;
}
count1 = 0;
while(count1 <= lastchar){
    printf("%c\n", ch_arr[count1]);
    count1 = count1 + 1;
}
exit(EXIT_SUCCESS);
}
```

[مثال 4.1]

ستلاحظ استخدام الثابت المُعرّف ARSIZE في كل مكان ضمن المثال السابق بدلاً من كتابة حجم المصفوفة الفعلي بصورة صريحة، ويمكننا بفضل ذلك تغيير العدد الأقصى من المحارف الممكن ترتيبها ضمن هذا البرنامج بتغيير سطرٍ واحدٍ منه وإعادة تصريفه. الانتباه إلى امتلاء المصفوفة هي نقطة غير مشدّد عليها ولكنها هامة لأمان برنامجنا؛ فإذا نظرت بتمعّن للمثال، ستجد أن البرنامج يتوقف عند تمرير العنصر ذو الدليل ARSIZE-1 للمصفوفة، وذلك لأن أي مصفوفة بحجم N عنصر تحتوي العناصر من 0 إلى N-1 فقط أي ما مجموعه N عنصر.

على عكس بعض اللغات، لا تُعلمك لغة سي أنك وصلت إلى نهاية المصفوفة، بل تُنتج ذلك بما يعرف باسم **التصرف غير المحدد undefined behaviour** في البرنامج، وهذا ما يتسبب ببعض الأخطاء الغامضة في برنامجك. يتجنّب المبرمجون الخبراء هذا الخطأ عن طريق اختبار البرنامج المتكرّر للتأكد من عدم حصول ذلك عند تطبيق الخوارزمية المستخدمة، أو عن طريق فحص القيمة قبل محاولة الحصول عليها من المصفوفة. وتُعد هذه المشكلة من أبرز مصادر أخطاء وقت التشغيل run-time في لغة سي، لقد حذرتك!

خلاصة القول:

- تبدأ المصفوفات بالدليل 0 دائماً، ولا يمكنك تجنّب هذا الاصطلاح.
- تحتوي المصفوفة من الحجم "N" على عناصر من الدليل "0" إلى الدليل "N-1"، والعنصر "N" غير موجود داخل المصفوفة هذه، ومحاولة الوصول إليه هو خطأ فادح.

1.5 مصطلحات

هناك نوعان من الأشياء المختلفة في البرامج المكتوبة بلغة سي، أشياء تُستخدم لتخزين القيم، وأشياء تدعى بالدوال، وبدلاً عن الإشارة للشئيين بصورة منفصلة بعبارة طويلة، نعتقد أنه من الأفضل أن نرمز إليهما بتسميةٍ واحدةٍ عامةٍ ألا وهي "الكائنات objects"، وسنستخدم هذه التسمية كثيراً في الفصول القادمة، إذ يتبع الشئان نفس القواعد إلى حدٍّ ما؛ ولكن يجدر الذكر هنا إلى أن معنى المصطلح هذا مختلف عما يقصده المعيار، إذ يُستخدم مصطلح "كائن" في المعيار على نحوٍ خاصٍ لوصف منطقة من الذاكرة المحجوزة لتمثيل قيمة، والدالة مختلفة عن هذا التعريف تماماً. يؤدي هذا لاستخدام المعيار المصطلحين على نحوٍ منفصلٍ وغالبًا ما ستجد العبارة "... الدوال والكائنات ..."، ونظراً لأن هذا الاختلاف لا يؤدي لكثيرٍ من الخلط ويحسن قراءة النص في كثيرٍ من الحالات، فسنتمّر في استخدام المصطلح العام "كائن" للدلالة على الدوال والقيم، وسنستخدم المصطلح "دالة" و"كائن بيانات Data object" عندما نريد التمييز بين الاثنين وفقاً للحالة. لذا، قد تجد اختلافًا بسيطًا في المعنى إن كنت تقرأ المعيار.

1.6 خاتمة

قدّم هذا الفصل العديد من مبادئ اللغة وأساسياتها -بصورة غير رسمية، إذ تشكّل الدوال اللبنة الأساسيّة لبناء البرامج في لغة سي C وسنستعرض لاحقًا شرحًا مفصّلًا عن هذه الكائنات الأساسية، لكنك غالبًا تفهم الأمور الأساسية المتعلقة بها على نحوٍ كافٍ يسمح لك بمتابعة استخدامها وفهمها في الفصول القادمة.

بالرغم من أنّنا تكلمنا عن فكرة الدوال ضمن المكتبة، إلا أننا لم نوضح أهميتها الشديدة في برمجة التطبيقات بلغة سي، إذ تُعد المكتبة القياسية **مهمة جدًا** لتحسين عمل البرامج المحمولة portable للاستخدام العام، إضافةً للمساعدة في عملية إنتاج البرامج بواسطة الدوال التي تحتويها.

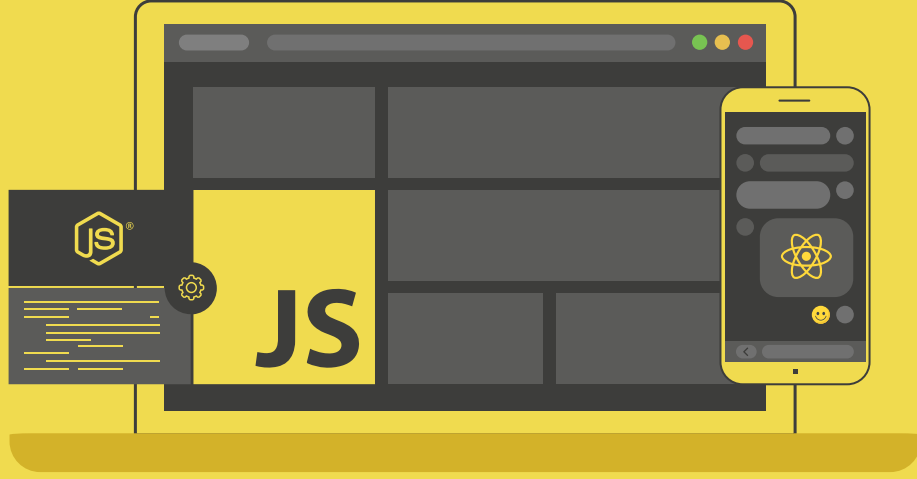
سنشرح قريبًا استخدام المتغيرات والتعبيرات والعمليات الحسابية بتفصيل أكبر، إذ استعرض هذا الفصل بصورة بسيطة أن لغة سي مختلفةٌ بعض الشيء عن لغات البرمجة الحديثة الأخرى. بقي لنا استعراض البيانات المنظّمة، مع أن التكلّم عن المصفوفات كان وجيزًا.

1.7 تمارين

1. اكتب المثال 1.1 على حاسوبك وشغله وافحصه.
2. بالاستعانة بالمثال 2.1، اكتب برنامجًا يطبع زوجين من الأعداد الأولية المختلفة بحيث يكون الفرق بينهما 2، على سبيل المثال 11 و13، 29 و31 (إن استطعت إيجاد نمط بالنظر لهذه الأرقام مبارك عليك! أنت عبقرى، أو مخطئ).
3. اكتب دالة تعيد قيمة عدديّة `int` تمثل القيمة العشرية للمحارف الموجودة في سلسلة نصية ما باستخدام الدالة `getchar`. على سبيل المثال، إذا أُدخل الرقم 1 متبوعًا بالرقم 4 متبوعًا بالرقم 6، يجب أن تُعيد الدالة العدد 146. لك أن تفترض أن دخل المستخدم سيكون محصورًا بين الأرقام 0 و9 وأن هذه الأرقام متتالية (ومرتبة كما يقول المعيار) وأنه يتعين على الدالة التعامل مع الأعداد الصالحة والأسطر الجديدة، لذا تستطيع غض النظر عن تشخيص الأخطاء الناتجة عن الدخل غير المناسب.
4. استخدم الدالة التي كتبتها في التمرين السابق لقراءة سلسلة من الأرقام، وإسناد كلٍّ من الأرقام إلى مصفوفة مصرّح عنها في الدالة `main` عن طريق استدعاء الدالة عدة مرّات، ورتب الأرقام ترتيبًا تنازليًا، واطبع مصفوفة الأرقام الناتجة.
5. مجددًا، وباستخدام الدالة من التمرين 3.1، اكتب برنامجًا يقرأ الأرقام من الدخل، واطبع هذه الأرقام بالتمثيل الثنائي والعشري والسادسي عشري `hexadecimal`، دون استخدام أي من ميزات الدالة `printf` المذكور في الفصل هذا، بالأخص مِيزة الطباعة بالنظام السداسي عشري. من المفترض

أن تحل هذا التمرين بتحديد كل عدد ستطبعه في وقته وأن تتأكد أن هذه الأعداد مطبوعة بالترتيب الصحيح، هذا التمرين ليس صعب، ولكنه ليس بسيطًا بالضرورة أيضًا.

دورة تطوير التطبيقات باستخدام لغة JavaScript



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



2. المتغيرات والعمليات الحسابية

2.1 بعض الأساسيات

سنلقي في هذا الفصل نظرةً على الأجزاء التي لم نلق لها بالأ في الفصل السابق، الذي كان بمثابة مقدمة سريعة عن لغة سي، التحدي هنا هو التكلم عن أساسيات اللغة بصورةٍ موسَّعة وكافية للسماح لك بفهم المزيد عن اللغة دون إغراق المبتدئين بالمعلومات والتفاصيل غير الضرورية في هذه المرحلة.

بالنظر إلى أن هذا الفصل فهو طويل ويغطي بعض المفاهيم والمشاكل الدقيقة التي لا تقرأ عنها في النصوص التقديمية للغة، فيجب عليك أن تقرأه بعقليةٍ منفتحة وبمزاجٍ جيّد.

قد يجد دماغك المرهق التمارين الموجودة بين الفقرات استراحةً مفيدة، إذ ننصحك بشدّة أن تحاول حلّ التمارين هذه بينما تقرأ الفصل، إذ أن ذلك من شأنه أن يساعدك في موازنة الكفة بين تعلّم المفاهيم الجديدة -التي قد تشعر في بعض المراحل بغزارتها- والتمارين.

حان الوقت لتقديم بعض الأساسيات في لغة سي.

2.2 المحارف المستخدمة في لغة سي

هذه الفقرة مثيرة للاهتمام، وسندعو المحارف المُستخدمة في لغة سي بأبجدية سي C، وهذه الأبجدية مهمة جدًّا، وربما يكون هذا الجزء الوحيد من هذا الفصل الذي يمكنك قراءته بصورةٍ سطحية وفهم جميع محتوياته من المرة الأولى. لذلك، اقرأه لتضمن أنك تعرف محتوياته والمعلومات الواردة فيه جيّدًا وتذكر أن تعود إليه في حال أردت مرجعًا بهذا الخصوص.

2.2.1 الأبجدية الاعتيادية

تعرف قلة من لغات البرمجة أبجديتها أو تلقي بالأمر، إذ أن هناك افتراضًا مسبقًا بأن أحرف الأبجدية الإنجليزية وخليطًا من علامات الترقيم والرموز ستكون متاحة في أي بيئة داعمة للغة، ولكن هذا الافتراض غير محقق دائمًا. تعاني لغات البرمجة القديمة من هذه المشكلة بدرجة أقل حدة، ولكن تخيل إرسال برنامج مكتوب بلغة سي عبر جهاز تللكس Telex أو عن طريق نظام بريد إلكتروني يحتوي على بعض القيود، أتعني أهمية الأمر الآن؟

يوصف المعيار مجموعتين مختلفتين من المحارف: واحدة تُكتب بها البرامج وأخرى تُنفذ بها، وذلك للسماح للأنظمة المختلفة بتصريف البرنامج وتنفيذه بغض النظر عن اختلاف طرق ترميز المحارف لكل نظام. في الحقيقة، الأمر مهم فقط في حال استخدامك محارفًا ثابتة constant في المعالج المُسبق preprocessor، إذ من الممكن أن تختلف قيم هذه المحارف عند التنفيذ، وهذا السلوك معرّف عند التنفيذ implementation-defined، فهو موثوق بالتأكيد، ولكن لا تقلق بخصوص هذا الأمر الآن.

يملي المعيار وجود أبجدية مؤلفة من 96 رمزًا للغة سي، وهي:

الجدول 1: أبجدية لغة سي

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9																
!	"	#	%	&	'	()	*	+	,	-	.	/												
:	;	<	=	>	?	[\]	^	_	{														

المسافات spaces ومسافات الجدولة الأفقية والعمودية horizontal and vertical tab ومحرف السطر الجديد newline وفاصل الصفحة form feed

أتضح أن معظم أبجديات الحاسوب الشائعة تحتوي على جميع الرموز اللازمة للغة سي، عدا بعض الحالات الشاذة النادرة مثل المحارف الموجودة في الأسفل، والتي تُعد مثالًا عن محارف أبجدية لغة سي المفقودة من مجموعة المحارف ذات 7 بت لمعيار منظمة المعايير العالمية International Standards Organization المدعوّ ISO 646، وهي مجموعة جزئية من المحارف المستخدمة في أبجديات الحاسوب على نطاق واسع.

[\] ^ { | } ~

لتضمين هذه الأنظمة التي لا تحتوي على مجموعة المحارف البالغ عددها 96 والمطلوبة لكتابة برامج بلغة سي، حدّد المعيار طريقةً لاستخدام معيار ISO 646 لتمثيل المحارف المفقودة ألا وهي تقنية **ثلاثيات المحارف trigraphs**.

2.2.2 ثلاثيات المحارف

ثلاثيات المحارف Trigraphs هي سلسلة من ثلاثة محارف ضمن المعيار ISO 646، وتُعامل معاملة محرفٍ واحدٍ ضمن أبجدية لغة سي. تبدأ جميع ثلاثيات المحارف بعلامتي استفهام "??"، ويساعد هذا في الدلالة على أن هناك شيء "خارج عن المألوف" ضمن البرنامج. يوضح الجدول 2 أدناه جميع ثلاثيات المحارف المُعرّفة ضمن المعيار.

الجدول 2: ثلاثيات المحارف

ثلاثي المحرف	محرف أبجدية سي C
??=	#
??([
??)]
??<	{
??>	}
??/	\
??!	
??-	~
??'	^

دعنا نفترض مثلًا، أن طرفيتك terminal لا تحتوي على الرمز "#" لكتابة سطر المعالج المُسبق التالي:

```
#define MAX 32767
```

عندها، نستطيع استخدام طريقة ثلاثيات المحارف على النحو التالي:

```
??=define MAX 32767
```

ستعمل ثلاثيات المحارف حتى لو كان المحرف "#" موجودًا، ولكن هذه التقنية موجودة لمساعدتك في الحالات الحرجة ولا يُحبَّذ استخدامها بديلًا عن محارف أبجدية سي دومًا.

ترتبط إشارة الاستفهام "?" بالمحرف الواقع على يمينها، لذا في أي سلسلة مؤلفة من عدّة إشارات استفهام، تشكّل إشارتان فقط ضمن السلسلة ثلاثي محارف، ويعتمد المحرف الذي تمثّله على المحرف الذي يقع بعد الإشارتين مباشرةً، وهذا من شأنه أن يجتّب كثيرًا من الالتباس.

من الخطأ الاعتقاد أن البرامج المكتوبة بلغة سي، والتي تلقي بالًا كبيرًا لإمكانية تشغيلها على نحوٍ محمول portable وعلى جميع الأنظمة مكتوبةً باستخدام ثلاثيات المحارف "إلا في حال ضرورة نقلها إلى نظام يدعم معيار ISO 646 فقط"؛ فإذا كان نظامك يدعم جميع المحارف البالغ عددها 96 محرفًا، واللازمة لكتابة البرامج

بلغة سي، فيجب استخدامها دون الاستعانة بثلاثيات المحارف، إذ وُجدت هذه التقنية لتنفيذها ضمن بيئات محدودة، ومن السهل جدًا كتابة مفسر يحوّل برنامجك بين التمثيلين عن طريق فحص كل محرف بمحرفه. ستميّز جميع المصروفات المتوافقة مع المعيار ثلاثيات المحارف عندما تجدها، إذ أن تبديل ثلاثيات الأحرف هو من أولى العمليات التي يجريها المصروف على البرنامج.

2.2.3 المحارف متعددة البايت Multibyte Characters

أضيف دعم المحارف متعددة البايت multibyte characters إلى المعيار، ولكن ما هو السبب؟ تتضمن نسبة كبيرة من الحوسبة الاعتيادية اليومية بياناتٍ ممثلة بنص بشكلٍ أو بآخر، وساد الاعتقاد في مجال الحوسبة أنه من الكافي دعم ما يقارب مئة محرف مطبوع فقط، ومن هنا كان عدد المحارف الممثلة لأبجدية سي 96 محرفًا، وذلك بناءً على متطلبات اللغة الإنجليزية وهذا الأمر ليس مفاجئًا بالنظر إلى أن معظم سوق تطوير البرمجيات والحوسبة التجاري كان في الولايات المتحدة الأمريكية. تُعرف مجموعة المحارف هذه باسم **المخزون repertoire** وتتناسب مع 7 أو 8 بتات من الذاكرة، وهذا السبب في أهمية استخدام 8-بت واحدةً لتخزين وقياس للبيانات بصورةٍ أساسية في معيار US-ASCII ومعمارية الحواسيب المصغرة minicomputers والحواسيب الدقيقة microcomputers.

تتبع لغة سي أيضًا توجّهًا في تخزين البيانات باستخدام البايت byte، إذ أن أصغر وحدات التخزين الممكن استخدامها مباشرةً في لغة سي هي البايت، والمعروفة بكونها تتألف من 8 بت. قد تشكو الأنظمة والمعماريات القديمة التي لم تُصمم مباشرةً لدعم ذلك من بطء بسيط في الأداء عند تشغيل لغة سي، لكن لا ينظر معظم الناس لذلك الأمر بكونه مشكلةً كبيرة.

لربما كانت الأبجدية الإنجليزية مقبولةً لتطبيقات معالجة البيانات حول العالم في وقتٍ مضى، إذ استُخدمت الحواسيب في أماكن يتوقّع من مستخدميها الاعتياد على هذا الأمر، لكن هذا لا يصلح في زمننا الحالي؛ إذ أصبح من الضروري في وقتنا المعاصر تزويد الأنظمة بوسائل معالجة البيانات وتخزينها باللغة الأم لمستخدميها. قد نستطيع حشر جميع المحارف المستخدمة في لغات كلٍّ من الولايات المتحدة الأمريكية وأوروبا الغربية ضمن مجموعة محارف لا يتجاوز حجمها 8 بت لكل محرف، ولكن الأمر مستحيلٌ بالنسبة للغات الآسيوية.

هناك طريقتان لتوسعة مجموعة المحارف، الأولى عن طريق إضافة عددٍ معين من البايتات (عادةً اثنين) لكل محرف، وهذه هي الطريقة المصممة لدعم محارف أكبر حجمًا في لغة سي؛ أما الطريقة الأخرى فهي باستخدام مخطط ترميز الإدخال بالإزاحة shift-in والخرج بالإزاحة shift-out؛ وهو ترميزٌ شائع في قنوات الاتصال ذات سعة 8-بت. ألقى نظرةً على سلسلة المحارف التالية:

```
a b c <SI> a b g <SO> x y
```

إذ يعني المحرف <SI> "بَدَل إلى اليونانية" والمحرف <SO> "بَدَل مجدداً إلى الإنجليزية"، ويعرض جهاز العرض المُهيئاً للعمل وفق هذا الترميز النتيجة على النحو التالي: a, b, c, alpha, beta, gamma, x, y. هذه هي الطريقة المُتبعة تقريباً في معيار shift-JIS الياباني، والاختلاف في ذلك المعيار هو أن المحارف الموجودة ضمن الإدخال بالإزاحة تتألف من محرفين يُستخدمان في تمثيل محرفٍ واحدٍ باللغة اليابانية، وهناك العديد من الطرق البديلة التي تستخدم عدة محارف مدخلة بالإزاحة، ولكنها أقل شيوعاً.

يسمح المعيار الآن باستخدام مجموعات المحارف الموسَّعة extended character sets، إذ تُستخدم المحارف المُعرفة مُسبقاً وبالبالغ عددها 96 في كتابة برنامج بلغة سي، ولكن مجموعة المحارف الموسَّعة مسموحة في كتابة التعليقات والسلاسل النصية والمحارف الثابتة وأسماء ملفات الترويسة (جميع ما ذُكر يمثل بيانات وليس جزءاً من كتابة البرنامج). يضع المعيار بعض القواعد البديهية لوصف طريقة استخدام هذه المجموعات، ولن نكرها هنا، لكن أبرزها هو أن البايث ذو القيمة الصفرية يُفسَّر محرفاً فارغاً **null** بغض النظر عن أي حالة إزاحة. هذا المحرف مهم لأن لغة سي ترمز لنهاية السلسلة النصية به وتعتمد عليه العديد من دوال المكتبات. هناك متطلبٌ آخر ألا وهو أن سلاسل المحارف متعددة البايث يجب أن تبدأ وتنتهي ضمن حالة الإزاحة المبدئية.

يصف المعيار النوع char بكونه مناسباً لتخزين قيمة جميع المحارف ضمن "مجموعة محارف التنفيذ execution character set" التي ستكون مُعرفةً في توثيق نظامك؛ وهذا يعني (في المثال السابق) أن نوع char يمكنه تخزين 'a' أو 'b' أو محرف الإزاحة إلى اللغة اليونانية بنفسه <SI>، إذ لا يوجد أي فرق في القيم المخزنة بداخل المتغير من نوع char بسبب تقنية الإدخال والإخراج بالإزاحة؛ وهذا يعني أنه لا يمكننا تمثيل 'a' على أنه محرف "ألفا" في اللغة اليونانية، وحتى نستطيع تحقيق ذلك يلزمنا أكثر من 8 بتات، وهو أكبر من حجم char في معظم الأنظمة، وهنا يأتي دور نوع المتغير wchar_t الذي قدّمه المعيار، ولكن يجب تضمين ملف الترويسة <stddef.h> قبل استخدامه، لأن wchar_t معرّفٌ على أنه اسم بديل عن نوعٍ موجودٍ في لغة سي. سنناقش هذا الأمر بتوسع أكبر لاحقاً.

ختاماً، نستطيع تلخيص ما سبق:

- تتطلب لغة سي مجموعة محارف عددها 96 محرف على الأقل لاستخدامها في محارف الشيفرة المصدرية للبرنامج.
- لا تحتوي جميع مجموعات المحارف على 96 محرف، بالتالي تسمح ثلاثيات المحارف للمعيار ISO 646 الأساسي كتابة برامج سي في حال الضرورة.
- أُضيفت المحارف متعددة البايث حديثاً مع المعيار، وتدعم:

- المحارف متعددة البايت المُرمّزة بالإزاحة Shift-encoded multibyte characters، التي تسمح بحشر المحارف الإضافية ضمن سلاسل محارف "اعتيادية"، بحيث يمكننا استخدام النوع char معها.
- المحارف الموسّعة wide characters التي تتسع لمساحة أكبر من المحارف الاعتيادية، ولها نوع بيانات مختلف عن النوع char.

2.3 البنية النصية للبرامج

2.3.1 تخطيط البرنامج

اعتمدت أمثلتنا حتى اللحظة في تنسيقها على المسافات البادئة indentation والأسطر الجديدة newlines، وهذا الأسلوب في التنسيق شائع في لغات البرمجة التي تنتمي إلى عائلة لغة سي، إذ تُعد هذه اللغات لغات "حرة التنسيق free format" وتُستخدم هذه الحرية في كتابة وتنسيق السطور البرمجية بحيث يحسّن قراءتها ويبرز تسلسل منطقها. تُستخدم محارف المسافات الفارغة space بما فيها مسافات الجدولة tab الأفقية لإنشاء المسافات البادئة في أي مكان دون أن تؤثر على عمل البرنامج عدا ضمن المعرّفات identifiers والكلمات المفتاحية. تعمل الأسطر الجديدة على نحوٍ مماثل لعمل المسافات الفارغة ومسافات الجدولة **باستثناء** أسطر أوامر المعالج المسبق، الذي يمتلك بنية سطريّة line-by-line.

هناك حلان يمكنك اللجوء إليهما في حال كان أحد السطور طويلًا جدًا وغير مريحًا للقراءة، إذ يمكنك استبدال محرف المسافة space بمحرف سطر جديد، بحيث يصبح لديك سطرين بدلًا من سطرٍ واحد. ألق نظرةً على المثال التالي للتوضيح:

```
/* a long line */
a = fred + bill * ((this / that) * sqrt(3.14159));
/* the same line */
a = fred + bill *
    ((this / that) *
    sqrt(3.14159));
```

لن نستطيع في بعض الحالات استبدال المحارف بالطريقة السابقة، وذلك بسبب اعتماد المعالج المسبق على "تعليمات" السطر الواحد، ولحلّ هذه المشكلة يمكننا استخدام السلسلة "n" التي تعني الانتقال لسطرٍ جديد، إذ يصبح هذا المحرف غير مرئي لنظام تصريف لغة سي، ويمكن نتيجةً لذلك استخدام هذه السلسلة في أماكن لا نستطيع استخدام الفراغات فيها في الحالات الاعتيادية، أي ضمن المعرّفات مثلًا أو الكلمات المفتاحية أو السلاسل النصية أو غيرها، وتسبق مرحلة معالجة هذه المحارف مرحلة معالجة ثلاثيات المحارف Trigraphs فقط.

```

/*
 * Example of the use of line joining
 */
#define IMPORTANT_BUT_LONG_PREPROCESSOR_TEXT \
printf("this is effectively all ");\
printf("on a single line ");\
printf("because of line-joining\n");

```

ينبغي أن تُستخدم هذه الطريقة في تقسيم الأسطر (بعيدًا عن أسطر تحكم المعالج المُسبق) في حالة واحدة فقط، ألا وهي لمنع السلاسل النصية الطويلة من أن تختفي إلى اليمين عند النظر لسطور البرنامج. بما أن السطور الجديدة غير مسموحة داخل السلاسل النصية والمحارف الثابتة، قد تعتقد أن هذه الفكرة جيّدة:

```

/* not a good way of folding a string */
printf("This is a very very very\
long string\n");

```

سيعمل المثال السابق بالتأكيد، ولكن من المحبّب استخدام ميزة ضمّ السلاسل النصية string-joining التي قُدّمت ضمن المعيار عند التعامل مع السلاسل النصية:

```

/* This string joining will not work in Old C */
printf("This is a very very very"
       "long string\n");

```

يسمح لك المثال الثاني بإضافة الفراغات دون تغيير مضمون السلسلة النصية، إذ إنّ المثال الأول يضيف الفراغات إلى مضمون السلسلة النصية.

لكن هل انتهت أن المثالين يحتويان على خطأ؟ لا يوجد هناك أي مسافة فارغة تسبق الكلمة "long"، مما يعني أن خرج البرنامج سيكون "verylong" دون مسافة بين الكلمتين.

2.3.2 التعليق

تكلّمنا عن التعليق سابقًا وقلنا أن التعليق يبدأ بالمحرفين `/*` وينتهي بالمحرفين `*/`، ويُترجم التعليق إلى مسافة فارغة واحدة أينما وجد وهو يتبع القوانين ذاتها الخاصة بالمسافة الفارغة. من المهم هنا معرفة أن هذا التعليق لا يختفي -كما كان الحال في لغة سي القديمة- ومن غير الممكن وضع التعليق بداخل سلسلة نصية أو محرف ثابت وإلا أصبح جزءًا منهما:

```

/*"This is comment"*/
/*The quotes mean that this is a string*/

```

لم تكن لغة C القديمة واضحةً بشأن تفاصيل حذف التعليق، إذ كان من الممكن أن يكون الناتج هنا:

```
int/**/egral();
```

هو حذف التعليق، أي أن المصرف سينظر للتعليمة السابقة بكونها استدعاءً لدالة اسمها `integral`، لكنه سيُبدّل التعليق بالاعتماد على معيار سي بمسافة فارغة وستكون التعليمة السابقة مساويةً للتعليمة التالية:

```
int egral();
```

التي تصرح عن دالة باسم `egral` تُعيد قيمةً من نوع `int`.

2.3.3 مراحل الترجمة

تُجرى ترجمة المحارف المختلفة وضمّ الأسطر والتعرف على التعليقات ومراحل أخرى من الترجمة المبكرة وفق ترتيبٍ معيّن، إذ يقول المعيار أن هذه المراحل تحدث بالترتيب التالي:

1. ترجمة المحارف الثلاثية.
2. ضمّ الأسطر.
3. ترجمة التعليقات إلى مسافات فارغة (عدا التعليقات الموجودة ضمن السلاسل النصية والمحارف الثابتة)، إذ من الممكن في هذه المرحلة جمع عدّة مسافات فارغة إلى مسافة فارغة وحيدة.
4. ترجمة البرنامج.

تُنهي كل مرحلة قبل أن تبدأ المرحلة التي تليها.

هناك مزيدٌ من المراحل ولكننا ذكرنا أكثر المراحل أهمية.

2.4 الكلمات المفتاحية والمعرفات

بعد أن تكلمنا عن أبجدية لغة سي، سنلقي نظرةً على مزيدٍ من عناصر اللغة المثيرة للاهتمام، إذ تُعد **الكلمات المفتاحية** `keywords` و**المعرفات** `identifiers` المكونات الأكثر وضوحًا، وعلى الرغم من تشابه تركيبها إلا أنها مختلفة.

2.4.1 الكلمات المفتاحية `Keywords`

تحجز لغة سي مجموعةً صغيرةً من **الكلمات المفتاحية** لاستخدامها الخاص، ولا يمكن استعمال هذه الكلمات المفتاحية على أنها معرفّات داخل البرنامج، وهذا أمرٌ شائعٌ في معظم لغات البرمجة الحديثة. قد يتفاجئ بعض مستخدمو لغة سي القديمة بوجود بعض الكلمات المفتاحية الجديدة، وإن كانت هذه الكلمات المفتاحية مُستخدمةً مثل معرفّات في برامج سابقة فيجب عليك تغييرها، ولحسن الحظ الانتباه لهذا النوع من

الأخطاء والعثور عليها سهل وبسيط، إذ سيخبرك المصنّف أن هناك بعض الأسماء غير الصالحة. يضمّ الجدول التالي الكلمات المفتاحية المُستخدمة في معيار سي، إذ ستلاحظ أن جميع الكلمات لا تبدأ بأحرف كبيرة.

الجدول 3: كلمات مفتاحية

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

الكلمات المفتاحية المُضافة جديدًا التي قد تفاجئ المبرمجين السابقين للغة سي هي: `signed` و `const` و `void` و `volatile` (على الرغم من وجود `void` منذ فترة). سيلاحظ بعض القراء شديدي الانتباه أن الكلمات المفتاحية `asm` و `entry` و `fortran` غير موجودة في الجدول، إذ أنها ليست من ضمن المعيار، وسيفتقدتها القليل منهم فقط.

2.4.2 المعرفات Identifier

يُعد **المعرف Identifier** مرادفًا لما نطلق عليه "اسم name"، وتُستخدم المعرفات في لغة سي للدلالة على العديد من الأشياء، فقد لاحظنا استخدامها حتى الآن في تسمية المتغيرات والدوال، ولكنها تُستخدم أيضًا في تسمية مزيدٍ من الأشياء التي لم نراها بعد، مثل:

- العناوين **labels**
- "وسوم" الهياكل **tags of structures**
- الاتحادات **unions**
- المعدّات **enums**

قواعد إنشاء معرف بسيطة جدًا، إذ يمكنك استخدام الأحرف البالغ عددها 52 حرفًا من الأبجدية الإنجليزية (أحرف كبيرة أو صغيرة)، والأرقام العشرة من 0 حتى 9 وأخيرًا الشرطة السفلية "_"، التي يمكن عدّها حرفًا من الأبجدية في حالتنا هذه، لكن هناك قيدٌ واحدٌ ألا وهو أن المعرف يجب أن يبدأ بحرف أبجدي.

على الرغم من أن المعيار لا ينص صراحةً على حد أقصى لطول اسم المعرّف، إلا أننا نحتاج أن نتكلم عن هذه النقطة، إذ لا يوجد هناك أي حد في لغة سي القديمة ومعياري سي لطول اسم المعرف، ولكن المشكلة هي أنه لا يوجد هناك أي ضمانات أن جميع محارف اسم المعرف تُفحص أثناء موازنة المساواة، فقد كان حد الموازنة في لغة سي القديمة 8 محارف، أما في المعيار فهو 31 محرف.

وبذلك يكون عملياً الحد الجديد للمعرف في المعيار هو 31 محرف، ومع ذلك يمكن للمعرفات تجاوز هذا الطول ولكنها يجب أن تختلف في أول 31 حرفاً إذا كنت تريد التأكد من أن برامجك محمولة portable. يسمح المعيار بالأسماء الطويلة لبعض التطبيقات، لذا إذا استخدمت الأسماء الطويلة وكان لا بدّ منها، فتأكد من أنها فريدة قبل أن يتوقف التحقق من اسمها في المحرف ذو الرقم 31.

يُعد طول **المعرفات الخارجية external identifiers** واحداً من أكثر الأشياء المثيرة للجدل في الإصدار الجديد؛ إذ تُعرف المعرفات الخارجية بأنها المعرفات التي يجب أن تكون مرئية خارج نطاق الشيفرة المصدرية المستخدمة فيها، وتُعد برامج المكتبة أو الدوال التي يجب أن تُستدعى من عدّة ملفات مصدرية مثلاً جيداً عليها.

اختار المعيار الحفاظ على القيود القديمة التي تخص هذه المعرفات، إذ لا تعدّ المعرفات الخارجية مختلفةً عن بعضها إلا في حالة اختلافها مع بعضها في المحارف الست الأولى، ويزداد الأمر سوءاً فقد تُعامل الأحرف الكبيرة والصغيرة بنفس الطريقة؛ والسبب وراء هذا عملي، إذ أن معظم أنظمة تصريف لغة سي تعمل بمساعدة أدوات نظام معينة للربط بين دوال المكتبات والبرنامج المكتوب بلغة سي C، وهذه الأدوات هي خارج تحكم مصرّف لغة C، ولذا على المعيار أن يضع بعض الحدود العملية التي ستتوافق مع شروط هذه الأدوات.

لا يوجد أي قيود إجبارية على عدد الأحرف، ولكن الالتزام بهذا القيد (الأحرف الست الأولى متكافئة الحالة بين الأحرف الكبيرة والصغيرة) يزيد من فرصة عمل البرنامج على مختلف الأجهزة دون مشاكل (برنامج محمول). يذكرنا المعيار دائماً بأنه ينظر إلى استخدام تكافؤ الحالة بين الأحرف الصغيرة والكبيرة إضافةً لحدّ المحارف في تسمية المعرفات على كونها ميزاتٍ قديمة، ومن الممكن أن يلغي المعيار القادم استخدام هذه القيود. لنأمل أن يحصل ذلك قريباً.

2.5 التصريح عن المتغيرات

ذكرنا في الفصول السابقة أنه يجب التصريح عن أسماء الأشياء قبل استخدامها، والاستثناء الوحيد هنا لهذه القاعدة هو أسماء الدوال التي تُعيد قيمةً من النوع int، لأنه مصرّح عنها افتراضياً بالإضافة لأسماء **العناوين labels**. بإمكانك إما **التصريح declaration** عن الأشياء، وهي العملية التي تصف اسم ونوع الشيء ولكنها لا تحجز أيّ مكان على الذاكرة، أو **التعريف definition**، الذي يحجز مكاناً في الذاكرة للشيء المُصرّح عنه.

الفرق بين التصريح والتعريف مهم، وللأسف فإنّ الكلمتين متشابهتان مما يسبب الخلط لدى الكثير، ومن هذه النقطة فصاعدًا سنستخدم هاتين الكلمتين في سياقهما الصحيح، لذلك إذا نسيت الفرق بين المصطلحين وأردت التأكد مرةً أخرى فارجع لهذه الفقرة.

القواعد المتعلقة بجعل التصريح ضمن التعريف معقّدةٌ بعض الشيء، لذا سنؤجلها ونكتفي حاليًا ببعض الأمثلة والقواعد التي ستؤدي الغرض لأمثلتنا القادمة.

```

/*
 * A function is only defined if its body is given
 * so this is a declaration but not a definition
 */

int func_dec(void);

/*
 * Because this function has a body, it is also
 * a definition.
 * Any variables declared inside will be definitions,
 * unless the keyword 'extern' is used.
 * Don't use 'extern' until you understand it!
 */

int def_func(void){
    float f_var;          /* a definition */
    int counter;         /* another definition */
    int rand_num(void);  /* declare (but not define) another
function */

    return(0);
}

```

سنستمرّ قُدّمًا في القسم التالي ونتكلم عن **نوع المتغيرات والتعابير**.

2.5.1 تمارين

1. ما هو استخدام ثلاثيات المحارف؟
2. متى تتوقع أن تستخدمها ومتى تستبعد استخدامها؟
3. متى لا يتساوى السطر الجديد مع المسافة أو مسافة الجدولة؟
4. متى تتوقع استخدام سلسلة "السطر الجديد باستخدام الشرطة المائلة الخلفية" النصية؟
5. ما الذي يحصل عندما توضع سلسلتان نصيتان بجوار بعضهما؟
6. لماذا لا يمكنك وضع تعليق بداخل تعليق آخر؟ (هذا القيد يمنع "تعليق" بعض أجزاء الشيفرة البرمجية إن لم تكن حذرًا).
7. ما هو أطول اسم يمكنك تسميته بصورة آمنة للمتغيرات؟
8. ما معنى التصريح؟
9. ما معنى التعريف؟

2.6 الأنواع الحقيقية Real Types

سيكون من الأسهل التعامل مع الأنواع الحقيقية real أولاً، لأن هناك تفاصيل وتعقيدات أقل بخصوصها موازنةً بنوع الأعداد الطبيعية Integers. يقدّم المعيار تفاصيلاً جديدةً بخصوص دقة ونطاق الأعداد الحقيقية، ويمكن أن تجدهم في ملف الترويسة "float.h" الذي سنناقشه بالتفصيل لاحقاً. هذه التفاصيل مهمة جداً ولكنها ذات طبيعة تقنية للغاية، ولن تُفهم بالكامل غالباً إلا من قبل مختصي التحليل العددي.

أنواع الأعداد الحقيقية هي:

- float: العدد العشري
- double: العدد العشري مضاعف الدقة
- long double: العدد العشري الأدق

يسمح لنا كل واحد من هذه الأنواع بتمثيل الأعداد الحقيقية بطريقة معينة باستخدام الحاسوب؛ فإذا كان هناك نوعٌ واحدٌ لتمثيل الأعداد الحقيقية، فهذا يعني أن تمثيل الأعداد سيكون متماثلاً بغض النظر عن الاستخدام؛ أما إذا كان العدد يتجاوز الثلاثة أنواع، فهذا يعني أن لغة سي لن تستطيع تصنيف أي من الأنواع الإضافية. يُستخدم النوع float للتمثيل السريع والبسيط للأرقام الصغيرة وهو مشابهٌ للنوع REAL في لغة فورتران؛ أما double فيستخدم للدقة الإضافية، و long double لدقة أكبر من سابقتها.

التركيز الأساسي هنا هو أنّ الزيادة في "دقة" كلٍ من float و double و long double تعطي لكل نوع نطاقاً ودقةً مساويةً للنوع الذي يسبقها على الأقل، فأخذ القيمة من متغير نوع double مثلاً، وتخزينها في متغير من نوع long double، يجب أن يمثّل القيمة ذاتها.

لا توجد هناك أي متطلبات للأشكال الثلاثة من متغيرات الأعداد "الحقيقية" لتختلف في خصائصها، وبالتالي إن لم توفر الآلة سوى نوع واحدٍ من أنواع متغيرات الأعداد الحقيقية، فيمكن عندئذٍ تمثيل جميع أنواع الأعداد الحقيقية الثلاثة في لغة سي بهذا النوع المتوقّر. لكن بغض النظر، يجب أن يُنظر إلى هذه الأنواع الثلاثة بأنها مختلفة، وكأنّ هناك فرقٌ بينها حقاً، وهذا يساعد في نقل البرنامج إلى نظام تختلف فيه هذه الأنواع حقاً، بحيث لن يظهر لك مجموعةٌ من التحذيرات من المصنّف بخصوص عدم توافق الأنواع التي لم تكن موجودةً على النظام الأول.

تسمح لغة سي بمزج جميع أنواع البيانات العددية في التعابير بعكس كثيرٍ من لغات البرمجة الصارمة بخصوص قواعد كتابتها، وذلك يضم مختلف أنواع الأعداد الصحيحة إضافةً إلى الأعداد الحقيقية وأنواع المؤشرات؛ وعندما يتضمن التعبير مزيجاً من أنواع الأعداد الحقيقية والصحيحة، سيُستدعى تحويلٌ ضمني يعمل بدوره على معرفة نوع المزيج الكلي الناتج. هذه القواعد مهمةٌ جداً وتدعى **التحويلات الحسابية الاعتيادية usual arithmetic conversions**، ومن المفيد أن تتذكرها، إذ سنشرح كامل هذه القواعد لاحقاً، إلا أننا سننظر في الوقت الحالي إلى الحالات التي تتضمن مزيجاً من float و double و long double ونحاول فهمها.

الحالة الوحيدة التي نحتاج فيها إجراء التحويلات المذكورة هي عندما يُمزج نوعان من البيانات في تعبير، كما في هو موضح في المثال التالي:

```
int f(void){
    float f_var;
    double d_var;
    long double l_d_var;

    f_var = 1; d_var = 1; l_d_var = 1;
    d_var = d_var + f_var;
    l_d_var = d_var + f_var;
    return(l_d_var);
}
```

[مثال 1.2]

نلاحظ في المثال السابق وجود كثيرٍ من التحويلات القسرية، لنبدأ بأسهلها أولاً، ولننظر إلى تعيين القيمة الثابتة 1 لكلٍ من المتغيرات الثلاثة. لا بُد من التنويه (كما سيشير القسم الذي يتكلم عن القيم الثابتة

constants لاحقًا) إلى أن القيمة 1 هي من نوع int، أي تمثل عددًا صحيحًا وليس قيمةً ثابتةً حقيقيةً، ويحوّل الإسناد قيمة العدد الصحيح إلى نوع العدد الحقيقي المناسب والأسهل للتعامل معه.

التحويلات المثيرة للاهتمام تأتي بعدها، وأولها ضمن السطر التالي:

```
d_var = d_var + f_var;
```

ما هو نوع التعبير الذي يتضمن العامل +؟ الإجابة عن هذا السؤال سهلة ما دمت ملماً ببعض القواعد؛ إذ يُحوّل النوع الأقل دقةً ضمناً إلى النوع الأكثر دقةً وتُنجز العملية الحسابية باستخدام هذه الدقة، وذلك عندما يجتمع نوعان من الأعداد الحقيقية في التعبير ذاته. يتضمن المثال السابق استخدام كلٍّ من double و float، لذلك تُحوّل قيمة المتغير f_var إلى النوع double وتُضاف فيما بعد إلى قيمة النوع double أي المتغير d_var، وتكون نتيجة هذا التعبير هي من نوع double أيضاً، لذا من الواضح أن عملية الإسناد إلى المتغير d_var صائبة.

عملية الجمع الثانية أكثر تعقيداً، ولكنها ما زالت سهلة الفهم، إذ تُحوّل قيمة المتغير f_var وتُجرى العملية الحسابية باستخدام دقة النوع double، ألا وهي عملية جمع المتغيرين، لكن هناك مشكلة، وهي أن نتيجة عملية الجمع من نوع double، لكن عملية الإسناد من نوع long double، ويكون مجدداً الحل البديهي في هذه الحالة هو تحويل القيمة الأقل دقة إلى الأكبر دقة، وهو ما يُجرى ضمناً قبل عملية الإسناد.

الآن بعد أن أخذنا نظرةً سريعةً على الأمثلة السهلة، حان وقت الأمثلة الأكثر صعوبة وهي الحالة التي يتسبب فيها التحويل القسري بتحويل نتيجةً بدقة عالية إلى دقة أقل منها، ففي مثل هذه الحالات قد يكون من الضروري خسارة الدقة بطريقةً محدّدة من تنفيذ التحويل. ببساطة، يجب أن يحدد التنفيذ طريقة تقريب أو اقتطاع للقيمة، وفي أسوأ الحالات قد يكون نوع الهدف غير قادرٍ على تخزين تلك القيمة الضرورية (على سبيل المثال محاولة جمع أكبر قيمة لعدد إلى نفسه)، وتُعد نتيجة التنفيذ في هذه الحالة غير محددة، والبرنامج يشكو من خطأ ولا يمكنك التنبؤ بتصرفه.

لا ضرر من تكرار فكرتنا السابقة، إذ يقصد المعيار بحالة **السلوك غير المحدد undefined behaviour** معنى اسمه حرفياً، وحالما يدخل البرنامج منطقة السلوك غير المحدد، يمكن لأي شيء أن يحدث؛ فمن الممكن إيقاف البرنامج من طرف نظام التشغيل مصحوباً برسالة معيّنة؛ أو قد يحدث شيء غير مُلاحظ ويستمر البرنامج للعمل باستخدام قيم خاطئة مُخزّنة في المتغير. منع البرنامج من إبداء أي سلوك غير محدّد تعد من مسؤولياتك، فتوخّ الحذر.

لتلخيص ما سبق:

- تُجرى العمليات الحسابية التي تتضمن نوعين باستخدام النوع الأعلى دقةً منهما.

- قد يتضمن الإسناد خسارة لدقة القيمة في حال كان نوع المتغير الهدف ذو دقة أقل من دقة القيمة التي تُسند لهذا المتغير.
- هناك مزيداً من التحويلات التي تُجرى عند مزج الأنواع ضمن تعبير واحد، إذ لم نصف جميعها بعد.

2.6.1 طباعة الأعداد الحقيقية

يمكن استخدام دالة الخرج التقليدي `printf` لتنسيق الأعداد الحقيقية وطباعتها، كما يوجد العديد من الطرق لتنسيق هذه الأعداد، ولكننا سنتطرق إلى طريقة واحدة في الوقت الحالي. يوضح الجدول 4 التنسيق الموافق لكل نوع من أنواع الأعداد الحقيقية.

الجدول 4: رموز التنسيق للأعداد الحقيقية

النوع	التنسيق
float	f%
double	f%
long double	Lf%

ألق نظرةً على المثال التالي لتجربة المعلومة السابقة:

```
#include <stdio.h>
#include <stdlib.h>

#define BOILING 212      /* degrees Fahrenheit */

main(){
    float f_var; double d_var; long double l_d_var;
    int i;

    i = 0;
    printf("Fahrenheit to Centigrade\n");
    while(i <= BOILING){
        l_d_var = 5*(i-32);
        l_d_var = l_d_var/9;
        d_var = l_d_var;
        f_var = l_d_var;
        printf("%d %f %f %Lf\n", i,
                f_var, d_var, l_d_var);
        i = i+1;
    }
}
```

```

    }
    exit(EXIT_SUCCESS);
}

```

[مثال 2.2]

جرب المثال السابق على حاسوبك الشخصي، ولاحظ النتائج.

2.6.2 تمارين

1. أي نوع من المتغيرات يخزن أكبر نطاق من القيم؟
2. أي نوع من المتغيرات يخزن أعلى دقة من القيم؟
3. هل هناك أي مشاكل ممكنة الحدوث عند إسناد float أو double إلى double أو long double؟
4. ما المشاكل التي قد تحدث عند إسناد long double إلى double؟
5. ما هي التوقعات التي يمكنك توقعها من برنامج له "سلوك غير محدد"؟

2.7 الأنواع الصحيحة Integral types

كانت الأنواع الحقيقية أسهل الأنواع، إذ تتصف القوانين الخاصة بالأنواع الصحيحة بتعقيد أكبر، ولكنها ما زالت مفهومة وينبغي تعلّمها. لحسن الحظ، الأنواع الوحيدة المستخدمة في لغة سي لتخزين البيانات هي الأنواع الحقيقية والصحيحة، إضافةً إلى الهياكل **structures** والمصفوفات **arrays** المبنية عليهما، إذ لا تحتوي لغة سي على أنواع مميزة للتلاعب بالمحارف، أو التعامل مع القيم البوليانية **boolean**، وإنما تستخدم الأنواع الصحيحة بدلاً من ذلك، وهذا يعني أنه حالما تفهم الأنواع الصحيحة والحقيقية فأنت تعرف جميع الأنواع. سنبدأ بالنظر إلى الأنواع المختلفة للأعداد الصحيحة وقوانين التحويل فيما بينها.

2.7.1 الأعداد الصحيحة البسيطة

هناك نوعان من متغيرات الأعداد الصحيحة يطلق عليهما "نكهات **flavours**"، ويمكن بناء أنواع أخرى انطلاقاً من هذين النوعين كما سنرى لاحقاً، لكن تبقى الأنواع البسيطة **int** هي الأساس. النوع الأكثر شهرةً هو العدد الصحيح ذو الإشارة **signed** أو **int**، أما النوع الأقل شهرةً هو العدد الصحيح عديم الإشارة أو **unsigned int**، ومن المفترض أن تُخزن القيم في المتغيرات ذات النوع المناسب حسب الآلة التي تشغل البرنامج.

عندما تبحث عن نوع بيانات بسيط لتمثيل عدد صحيح، فإن النوع **int** هو الاختيار البديهي لأي استخدام مُتساهل، مثل عدّاد ضمن حلقة تكرارية قصيرة، إذ لا توجد هناك أي قاعدة تحدّد عدد البتات التي يخزنها نوع

int لقيمة ما، لكنه سيكون **دائمًا** مساويًا إلى 16 بت أو أكثر، ويفضّل ملف الترويسة القياسي `<limits.h>` العدد الفعلي للبتات المُتاحة في تنفيذ معين.

لم تحتو لغة سي القديمة على أية معلومات بخصوص طول متغيّر من نوع int، ولكن الجميع كان يفترض اصطلاحًا أنها على الأقل 16 بت. في الحقيقة، لا يحدّد ملف الترويسة `<limits.h>` العدد الدقيق للبتات، ولكنه يقدّم تقديرًا لأعظم عدد وأقل عدد بتات لقيمة في متغيّر من نوع int، والقيم التي يحددها هي ما بين 32767 و-32767 أي 16 بت فما فوق، سواءً كانت عملية المتمم الأحادي أو الثنائي الحسابية مستخدمة أم لا، وبالطبع لا يوجد هناك أي قيود من توفير نطاق أكبر في أي الطرفين في حال توفرت الطريقة المناسبة.

يتراوح النطاق المحدد وفق المعيار للمتغير من نوع unsigned int من 0 إلى 65535، مما يعني أن القيمة لا يمكن أن تكون سالبة، وستكلم بإسهاب عن هذه النقطة لاحقًا.

إذا لم تعدّ التفكير بعدد البتات لمتغيّر ما، وبدأت بالقلق عمّا إذا سيؤثر ذلك على قابلية نقل البرنامج كون هذه المشكلة مرتبطةً بوضوح بالآلة (أي الحاسوب الذي يشغّل البرنامج)، فقلقك في محلّه. تأخذ لغة سي قابلية نقل البرنامج على محمل الجدّ كما تدلّك على القيم والمجالات الآمنة، وتشجّعك أيضًا عوامل العوامل الثنائية bitwise operators على التفكير بعدد البتات في متغيّر ما، لأنها تمنحك الوصول المباشر إلى بتات المتغير التي تعالجها بصورة منفردة (كل بت على حدة) أو في مجموعات. ونتيجة لذلك تكوّن لدى مبرمجي لغة سي المعرفة الكافية بخصوص مشكلات قابلية نقل البرنامج، ممّا يتسبب ببرمجة برامج قابلة للنقل، لكننا **لا ننفي** هنا إمكانية كتابة برامج غير قابلة للنقل إطلاقًا.

2.7.2 متغيرات المحارف

المتغير char هو النوع الثاني من أنواع الأعداد الصحيحة البسيطة، فهو نوع آخر من int ولكن بتطبيق مختلف، إذ تُعدّ فكرة تخصيص نوع خاص للتعامل مع المحارف فكرة جيّدة خاصةً وأن كثيرًا من برامج سي تتعامل بالمحارف، لأن تمثيل القيم باستخدام النوع int يأخذ كثيرًا من المساحة غير الضرورية لتمثيل المحرف.

يصف ملف ترويسة الحدود "limits" ثلاثة أشياء عن النوع char ألا وهي:

- عدد البتات 8 على الأقل.
 - يمكنها تخزين قيمة +127 على الأقل.
 - القيمة الدنيا للنوع char هي صفر أو أقل، مما يعني أن المجال يتراوح ما بين 0 إلى 127.
- يحدّد تنفيذ المتحول char فيما إذا كان سيتصرف تصرّف المتحولات ذات الإشارة signed أو عديمة الإشارة unsigned.

باختصار، تحتل متغيرات المحارف مساحةً أقل من المتغيرات الصحيحة `int` التقليدية، ويمكن استخدامها لمعالجة المحارف، لكنها تدرج تحت تصنيف الأعداد الصحيحة، ويمكن استخدامها لإجراء العمليات الحسابية كما هو موضح في المثال التالي:

```
include <limits.h>
include <stdio.h>
include <stdlib.h>

main(){
    char c;

    c = CHAR_MIN;
    while(c != CHAR_MAX){
        printf("%d\n", c);
        c = c+1;
    }

    exit(EXIT_SUCCESS);
}
```

[مثال 3.2]

تشغيل البرنامج في المثال السابق تمرينٌ لك، وربما ستثير النتائج إعجابك. إذا كنت تتساءل عن قيمة `CHAR_MIN` و `CHAR_MAX`، فاطّلع على ملف الترويسة `limits.h` واقراه.

إليك مثالٌ آخر مثيرٌ للإعجاب حقًا، إذ سنستخدم فيه محارف ثابتة `constants`، والتي يمكن كتابتها بين إشارتين تنصيص أحادية على النحو التالي:

```
'x'
```

لأن القواعد الحسابية تُطبّق هنا، فسيحوّل المحرف الثابت السابق ليكون من النوع `int`، ولكن هذا لا يهم حقًا لأن قيمة المحرف صغيرة دائمًا ويمكن تخزينها في متغير من نوع `char` دون فقدان أي دقة (لسوء الحظ هناك بعض الحالات التي لا ينطبق فيها هذا الكلام، تجاهلها في الوقت الحالي). عندما يُطبع محرف باستخدام الرمز `%c` ضمن دالة `printf`، يُطبع المحرف كما هو، لكن يمكنك استخدام الرمز `%d` إذا أردت طباعة قيمة العدد الصحيح الموافقة لهذا المحرف. لماذا استخدم الرمز `%d`؟ كما ذكرنا سابقًا، النوع `char` هو في الحقيقة نوع من أنواع الأعداد الصحيحة.

من المهم أيضًا وجود طريقة لقراءة المحارف إلى البرنامج، وتتكفل الدالة `getchar` بهذه المهمة، إذ تقرأ المحارف من **الدخل القياسي standard input** للبرنامج وتُعيد قيمةً صحيحةً `int` موافقةً لتخزين هذا المحرف في متغير من نوع `char`، تخدم هذه القيمة المُمرّرة من نوع `int` غرضين، هما: تمثيل جميع قيم المحارف الممكنة بواسطتها، إضافةً إلى تمرير قيمة **إضافية** للدلالة على نهاية الدخل. لا يتسع مجال قيم متغير من نوع `char` في جميع الحالات لهذه القيمة الإضافية، لذلك يُستخدم النوع `int`.

يقرأ البرنامج التالي الدخل ويعدّ الفواصل والنقاط المُدخلة، وعند وصوله لنهاية الدخل يطبع النتيجة.

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int this_char, comma_count, stop_count;

    comma_count = stop_count = 0;
    this_char = getchar();
    while(this_char != EOF){
        if(this_char == '.')
            stop_count = stop_count+1;
        if(this_char == ',')
            comma_count = comma_count+1;
        this_char = getchar();
    }
    printf("%d commas, %d stops\n", comma_count,
           stop_count);
    exit(EXIT_SUCCESS);
}
```

[مثال 4.2]

هناك ميزتان نستطيع ملاحظتهما من المثال السابق، الأولى هي الإسناد المتعدّد للعدادين، والثانية هي استخدام الثابت المعرّف `EOF`؛ وهي قيمة تُمرّر من الدالة `getchar` في نهاية الدخل وتمثّل اختصارًا لكلمة نهاية الملف `End Of File`، وتكون معرفةً ضمن ملف الترويسة `<stdio.h>`؛ أما الإسناد المتعدد فهي ميزة شائعة الاستخدام في برامج لغة سي.

لنأخذ مثالًا آخر، وليكن برنامجًا لطباعة جميع الأحرف الأبجدية بأحرف صغيرة إذا كان تنفيذك يحتوي على محارف مخزنة بصورة متتالية، أو طباعة نتيجة مثيرة للاهتمام إذا لم يكن كذلك. لا تقدّم لغة سي العديد من الضمانات بترتيب المحارف داخليًا، لذلك قد يتسبب هذا البرنامج بنتائج مختلفة ويكون **غير محمول**.


```
#include <stdio.h>
#include <stdlib.h>
main(){
    char c;

    c = 'a';
    while(c <= 'z'){
        printf("value %d char %c\n", c, c);
        c = c+1;
    }

    exit(EXIT_SUCCESS);
}
```

[مثال 5.2]

يذكرنا هذا المثال مرةً أخرى بأن `char` شكلٌ مختلفٌ من أشكال متغيرات الأعداد الصحيحة ويمكن استخدامه مثل أي عدد صحيح آخر، فهو **ليس** نوع مميّز بقواعد مختلفة.

تصبح المساحة التي يوفرها `char` موازنةً مع `int` ملحوظةً ومهمةً عندما يُستخدم الكثير من المتغيرات. تستخدم معظم عمليات معالجة المحارف مصفوفات كبيرة منها وليس محرفًا واحدًا أو اثنين فقط، وفي هذه الحالة يصبح الفرق واضحًا بين الاثنين. لتخيل سويًا مصفوفةً مؤلفةً من 1024 متغيرًا من نوع `int`، تحجز هذه المصفوفة مساحة 4098 بايت (كل بايت 8-بت) من التخزين على معظم الآلات، على افتراض أن طول كل `int` هو 4 بايت؛ فإذا كانت معمارية الحاسوب تسمح بتخزين هذه المعلومات بطريقة فعّالة، قد تطبّق لغة سي هذا عن طريق متغيرات من نوع `char` بحيث يأخذ كل متغير بايتًا واحدًا، وبذلك ستأخذ المصفوفة مساحة 1024 بايت، مما سيوفّر مساحة 3072 بايت.

لا يهمننا في بعض الحالات إن كان سيوفّر البرنامج مساحةً أم لا، ولكنه يوفّرنا بغض النظر، ومن الجيد أن تعطينا لغة سي فرصة اختيار نوع المتغير المناسب لاستخدامنا.

2.7.3 المزيد من الأنواع المعقدة

النوعان السابقان الذين تكلمنا عنهما سابقًا بسيطان، سواءً بخصوص تصريحهما أو استخدامهما، ولكن دقتهما في التحكم بالتخزين وسلوكهما غير كافيين في استخدامات نظم البرمجة المعقدة. تقدّم لغة سي أنواعًا إضافية من أنواع الأعداد الصحيحة للتعلّب على هذه المشكلة وتُقسم إلى تصنيفين، الأنواع ذات الإشارة `signed` والأنواع عديمة الإشارة `unsigned` (بالرغم من هذه المصطلحات كلمات محجوز في لغة سي إلا أن معناها يدلّ على غرضها أيضًا)، والفرق بين النوعين واضح؛ إذ يمكن للأنواع ذات الإشارة أن تكون قيمتها سالبة؛

بينما يكون من المستحيل أن تخزن الأنواع عديمة الإشارة قيمةً سالبة، وتستخدم الأنواع عديمة الإشارة في معظم الأحيان لحالتين، هما: إعطاء القيمة دقةً أكبر، أو عندما نضمن أن المتغير لن يخزن أي قيم سالبة في استخدامه، والحالة الثانية هي الحالة الأكثر شيوعًا.

تملك الأنواع عديمة الإشارة خاصيةً مميزةً ألا وهي أنها ليست عرضةً للطفحان الحسابي overflowing عند إجراء العمليات الحسابية، إذ سيتسبب إضافة 1 إلى متغيرٍ من نوعٍ ذي إشارة يخزن أكبر قيمة يمكن تخزينها بحدوث طفحان، ويصبح سلوك البرنامج نتيجةً لذلك غير محدد، ولا يحصل هذا الأمر مع المتغيرات من نوعٍ عديم الإشارة، لأنها تعمل وفق "باقي قسمة واحد زائد القيمة العظمى التي يمكن للمتغير تخزينها على هذه القيمة"، أي باقي قسمة "(max+1)/max"، والمثال التالي يوضح ما نقصده:

```
#include <stdio.h>
#include <stdlib.h>
main(){
    unsigned int x;
    x = 0;
    while(x >= 0){
        printf("%u\n", x);
        x = x+1;
    }

    exit(EXIT_SUCCESS);
}
```

[مثال 6.2]

بفرض أن المتغير x يحتل مساحة 16 بت، فهذا يعني أن مجال قيمته يتراوح بين 0 و 65535، وأن الحلقة التكرارية في المثال ستتكرر لأجل غير مسمى، إذ أن الشرط التالي محقق دائمًا:

```
x >= 0
```

وذلك بالنسبة لأي متغير عديم الإشارة.

يوجد ثلاثة أنواع فرعية لكلٍ من الأعداد الصحيحة ذات الإشارة وعديمة الإشارة، هي: short والنوع الاعتيادي و long، ونستطيع بعد أخذ هذه المعلومة بالحسبان كتابة لائحة بجميع أنواع متغيرات الأعداد الصحيحة في لغة سي باستثناء نوع تخزين المحرف char، على النحو التالي:

```
unsigned short int
unsigned int
```

```
unsigned long int
signed short int
signed int
signed long int
```

ليس مهمًا استخدام الكلمة المفتاحية `signed` ويمكن الاستغناء عنها في الأنواع الثلاث الأخيرة، إذ أن نوع `int` ذو إشارة افتراضيًا، ولكن **ينبغي عليك** استخدام الكلمة المفتاحية `unsigned` إذا أردت الحصول على نتيجة مغايرة لذلك. من الممكن أيضًا التخلي عن الكلمة المفتاحية `int` من أي تعليمة تصريح شرط أن تحتوي على كلمة مفتاحية أخرى، مثل `long` أو `short`، وسيفهم المتغير على أنه `int` ضمنيًا ولكنه أمر غير محدد، على سبيل المثال الكلمة المفتاحية `long` مساوية للكلمات `signed long int`.

يمنحك النوع `long` و `short` تحكمًا أكبر بمقدار المساحة التي تريد حجزها للمتغير، ولكلّ منهما مجال أدنى محدد في ملف الترويسة `<limits.h>`، وهو 16 بت على الأقل لكل من `short` و `int`، و32 بتًا على الأقل للنوع `long`، سواءً كان ذو إشارة `signed` أو دون إشارة `unsigned`. وكما ذكرنا آنفًا من الممكن للتنفيذ أن يحجز مقدارًا يزيد على المقدار الأدنى من البتات إذا أراد ذلك، والقيد الوحيد هنا هو أن حدود المجال يجب أن تكون متساوية أو محسنة، وألا تحصل على عدد أكبر من البتات في متغير من نوع أصغر موازنةً بنوع أكبر منه، وهي قاعدة منطقية.

أنواع متغيرات المحارف الوحيدة هي `signed char` و `unsigned char`، ويتمثل الفرق بين متغيرات من نوع `int` و `char` في أن جميع متغيرات `int` ذات إشارة إن لم يُذكر عكس ذلك، وهذا لا ينطبق على أنواع المحارف `char` التي قد تكون ذات إشارة أو عديمة الإشارة اعتمادًا على اختيار المُنفذ، وعادةً ما يُتخذ القرار بناءً على أسس الكفاءة. يمكنك طبعًا اختيار نوع المتغير قسريًا إذا أردت باستخدام الكلمة المفتاحية الموافقة، ولكن هذه النقطة لا تهمك إلا في حالة استخدامك لمتغيرات المحارف بنوعها القصير `short` لتوفير مساحة التخزين.

لتلخيص ما سبق:

- تتضمن أنواع الأعداد الصحيحة `short` و `long` و `signed` و `unsigned` والنوع الاعتيادي `int`.
- النوع الأكثر استخدامًا وشيوعًا هو النوع الاعتيادي `int` وهو ذو إشارة إلا في حالة تحديد عكس ذلك.
- **يمكن** للمتغيرات من نوع `char` أن تكون ذات إشارة أو عديمة الإشارة حسب تفضيلك، ولكن في حال غياب تخصيص لها ستُخصص الحالة الأفضل افتراضيًا.

2.7.4 طباعة أنواع الأعداد الصحيحة

يمكننا طباعة هذه الأنواع المختلفة أيضًا باستخدام الدالة `printf`، إذ تعمل متغيرات المحارف بنفس الطريقة التي تعمل بها الأعداد الصحيحة الأخرى، ويمكنك استخدام الترميز القياسي لطباعة محتوياتها (أي العدد الذي يمثل المحرف)، على الرغم من كون القيم الخاصة بها غير مثيرة للاهتمام أو مفيدة في معظم

الاستخدامات. نستخدم الرمز %c لطباعة محتويات متغيرات المحارف كما أشرنا سابقًا، كما يمكن طباعة جميع قيم الأعداد الصحيحة بالنظام العشري باستخدام الرمز %d أو %ld لأنواع long، ويوضح الجدول 5 المزيد من الرموز المفيدة لطباعة القيم بتنسيقٍ مختلف. لاحظ أنه في كل حالة تبدأ بالحرف l تُطبع قيمةً من نوع long، وهذا التخصيص ليس موجودًا فقط لعرض القيمة الصحيحة بل لتجنُّب السلوك غير المحدد لدالة printf إذا أُدخل الترميز الخاطئ.

الجدول 5: المزيد من رموز التنسيق

التنسيق	يُستخدم مع
%c	char (طباعة المحرف)
%d	القيمة العشرية لأنواع short و signed int و char
%u	القيمة العشرية لأنواع unsigned int و unsigned short و unsigned char
%x	القيمة الست عشرية لأنواع char و short و int
%o	القيمة الثمانية لأنواع char و short و int
%ld	القيمة العشرية للنوع signed long
%lu% lx% lo	كما ذُكر في الأعلى ولكن للنوع long

سنتكلم على نحوٍ مفصّل حول رموز التنسيق المستخدمة مع الدالة printf لاحقًا.

2.8 التعابير والعمليات الحسابية

من الممكن أن تكون التعابير في لغة سي معقدةً بعض الشيء نظرًا لاستخدام عدد من الأنواع المختلفة والعوامل operators في التعبير الواحد. سيشرح هذا القسم من الكتاب كيفية عمل التعابير هذه، وقد نتطرق للتفاصيل الصغيرة في بعض الأحيان، لذلك سيتوجب عليك قراءتها عدّة مرات حتى تتحقق من فهمك للفكرة.

دعنا نبدأ أولاً ببعض المصطلحات، إذ تُبنى التعابير في لغة سي من مزيج يتكون من **العوامل والمُعاملات**

operands. لنأخذ على سبيل المثال التعبير التالي:

$$x = a + b * (-c)$$

لدينا العوامل = و + و * و -، والمُعاملات التي هي المتغيرات x و a و b و c، كما يمكنك ملاحظة القوسين أيضًا اللذين يمكن استخدامهما في جميع التعبيرات الجزئية مثل -c. تنقسم معظم مجموعة عوامل لغة سي الواسعة إلى **عوامل ثنائية binary operators** تأخذ مُعاملين، أو **عوامل أحادية unary operators** تأخذ مُعاملًا واحدًا؛ ففي مثالنا كان - مُستخدمًا مثل عاملٍ أحادي، ويؤدي دورًا مختلفًا عن عامل الطرح الثنائي الذي يُمثّل بالرمز ذاته. قد تنظر إلى الفرق بأنه لا يستحق الذكر وأن وظيفة العامل ذاتها أو متشابهة في الحالتين،

ولكنه على النقيض تمامًا فإنه يستحق الذكر، لأن لبعض العوامل -كما ستجد لاحقًا- شكل ثنائي وآخر أحادي وكلّ وظيفة مختلفة تمامًا عن الأخرى، ويُعد عامل الضرب الثنائي * الذي يعمل عمل الموجّه باستخدام المؤشرات في حالته الأحادية مثالًا جيدًا على ذلك.

تتميز لغة سي بأن العوامل قد تظهر بصورة متتالية دون الحاجة للأقواس للفصل فيما بينهما في تعبيرٍ ما، إذ يمكننا كتابة المثال السابق على النحو التالي وسيظلّ تعبيرًا صالحًا.

$$x = a+b*-c;$$

بالنظر إلى عدد العوامل في لغة سي والطريقة الغريبة التي تعمل بها عملية الإسناد، تُعد **أسبقية precedence** العامل و**ارتباطه associativity** مسألةً هامةً جدًّا بالنسبة لمبرمج بلغة سي موازنًا باللغات الأخرى، وستناقش هذه النقطة بالتفصيل بعد التكلم عن أهمية عوامل العمليات الحسابية. لكن علينا قبل ذلك أن ننظر إلى عملية تحويل النوع التي قد تحصل.

2.8.1 التحويلات

تسمح لغة سي بمزج عدّة أنواع ضمن التعبير الواحد، وتسمح أيضًا بالعوامل التي يؤدي استخدامها إلى تحويلات للأنواع ضمنياً، يصف هذا القسم الطريقة التي تحدث بها هذه التحويلات. ينبغي على مبرمجي لغة سي القديمة (التي سبقت المعيار) قراءة هذا القسم بانتباه، إذ تغيّرت العديد من القواعد وبالأخص التحويل من float إلى double والتحويل من أنواع الأعداد الصحيحة short، كما أن القواعد الأساسية في **حفظ القيم value preserving** قد تغيّرت جدًّا في لغة سي المعيارية.

على الرغم من عدم ارتباط هذه المعلومة مباشرةً في هذا السياق، تجدر الإشارة هنا إلى أن أنواع الأعداد العشرية floating والصحيحة تُعرف باسم **الأنواع الحسابية arithmetic types** وتدعم لغة سي عدّة أنواع أخرى، أبرزها أنواع المؤشر pointer. تنطبق القوانين التي سنناقشها هنا على التعابير التي تحتوي الأنواع الحسابية فقط، إذ أن هناك بعض القواعد الإضافية عند إضافة أنواع مؤشر مع الأنواع الحسابية إلى هذا المزيج وسنناقشها لاحقًا.

إليك أنواع التحويلات المتنوعة في التعابير الحسابية:

- الترقيات العددية الصحيحة integral promotions.
- التحويلات بين الأنواع العددية الصحيحة.
- التحويلات بين الأنواع العددية العشرية.
- التحويلات ما بين الأنواع العددية الصحيحة والعشرية.

سبق وأن ناقشنا التحويلات بين الأنواع العددية الصحيحة في فصل الأنواع الحقيقية والصحيحة فقرة الأعداد الصحيحة، وما سنفعله في الوقت الحالي هو تحديد طريقة عمل التحويلات الأخرى، ومن ثم سننظر متى يجب استخدامها، عليك أن تحفظ هذه التحويلات عن ظهر قلب إذا أردت أن تصبح مبرمجًا بارعًا بلغة سي.

من الأشياء المختلف عليها التي قدمها المعيار، هي قواعد **الحفاظ على القيمة value preserving**، إذ تتطلب معرفةً معينةً من الحاسوب الهدف الذي ينفذ البرنامج من أجل معرفة نوع القيمة الناتجة من التعبير. عندما كنا نصادف في السابق نوعًا عديم الإشارة ضمن تعبير ما، كان هذا يعني ضمناً بأن القيمة الناتجة من نوع unsigned أيضًا، ولكن في الوقت الحالي النتيجة ستكون من نوع unsigned فقط إذا كان التحويل يتطلب ذلك، وهذا يعني أنه في معظم الحالات ستكون النتيجة من نوع signed.

السبب في هذا التغيير هو تقليل القيم التي قد تفاجئك عند مزج قيم من نوع ذو إشارة مع آخر عديم الإشارة، ففي معظم الحالات لا تعرف سبب هذا التغيير، وكان الدافع هنا التحويل إلى نتيجة "أكثر استخدامًا وطلبًا".

أ. الترقية العددية الصحيحة

أقل العمليات الحسابية دقةً في لغة سي هي باستخدام نوع الأعداد الصحيحة int، لذلك تحصل هذه التحويلات ضمناً في كل مرة تُستخدم الكائنات المذكورة في الأسفل ضمن تعبير ما. التحويل مُعرّف كما يلي:

- عند تطبيق الترقية العددية الصحيحة إلى نوع short أو char (أو **حقل البت bitfield** أو نوع **المعدّد enumeration type** الذين لم نتطرق إليهما بعد):

- ستُحوّل القيمة إلى int، إذا كان من الممكن للمتغير تخزين جميع قيم النوع الأصل.
- عدا ذلك، ستُحوّل إلى unsigned int.

يحفظ هذا التحويل كلاً من القيمة والإشارة الخاصة بالقيمة الأصلية، تذكر أن موضوع معاملة نوع char بإشارة أو دون إشارة يعود إلى التنفيذ.

تُطبّق هذه الترقّيات على نحوٍ متكرر بمثابة جزءٍ من **التحويلات الحسابية الاعتيادية** ومُعاملات عوامل الإزاحة الأحادية، مثل + و - و ~، كما تُطبّق عندما يكون التعبير مُستخدمًا مثل وسيط لدالة ما دون أي معلومات عن النوع ضمن نموذج الدالة الأولي function prototype، كما سنشرح لاحقاً.

ب. الأعداد الصحيحة ذات الإشارة وعديمة الإشارة

هناك الكثير من التحويلات الناتجة بين عدد من أنواع الأعداد الصحيحة المختلفة ومزج نكهاتها (أنواعها) المختلفة ضمن تعبير ما، وعند حدوث ذلك، ستحدث الترقية العددية الصحيحة. يمكن للنوع الجديد الناتج في جميع الحالات أن يخزن جميع القيم التي يستطيع النوع القديم تخزينها، وبذلك يمكن الحفاظ على القيم دون تغييرها.

في حال التحويل من عدد صحيح ذو إشارة إلى عدد صحيح عديم الإشارة وكان طول هذا العدد مساويًا لطول (أو أطول من) النوع الأصلي، فلن تتغير القيمة بعد التحويل إذا كان العدد ذو الإشارة موجبًا؛ أما إذا كانت القيمة سالبة فهذا يعني تحويلها إلى صيغة ذات إشارة للنوع الأطول وجعلها عديمة الإشارة عن طريق إضافة قيمتها إلى القيمة العظمى التي يستطيع النوع عديم الإشارة تخزينها زائد واحد. تُحافظ هذه العملية على نمط البتات الأصلي للأرقام الموجبة وتضمن "خانة الإشارة الموسّعة" للأرقام السالبة وذلك في نظام المتمم الثنائي.

لا يوجد هناك أي حالات "طفحان overflow" في جميع حالات تحويل عدد صحيح إلى نوع عديم إشارة قصير، فالنتيجة معرّفة وفق "الباقى غير السالب مقسومًا على القيمة العظمى للرقم عديم الإشارة الذي يمكن تمثيله باستخدام النوع القصير زائد واحد". يعني هذا ببساطة أنه في بيئة تعمل بنظام المتمم الثنائي، تُنسخ البتات منخفضة الترتيب low-order إلى الهدف ويكون التخلّص من البتات مرتفعة الترتيب high-order.

قد تحصل بعض المشاكل عند تحويل العدد الصحيح إلى نوع ذي إشارة قصير إن لم يكن هناك مساحة كافية لتخزين القيمة، وفي هذه الحالة تكون النتيجة حسب التنفيذ implementation defined، كما قد يتوقع معظم من اعتاد على سي القديمة أن يُنسخ نمط البتات منخفضة الترتيب.

من الممكن أن يكون البند الأخير مثيرًا للقلق بعض الشيء إذا كنت تتذكر الترقية العددية الصحيحة، لأنك قد تنظر إلى الأمر على النحو التالي: إذا أسندت متغيرًا من نوع char إلى متحولٍ من نوع char، فسيُرقى المتغير على اليمين إلى نوع من أنواع int. إذًا، هل من الممكن أن يؤدي الإسناد إلى تحويل int إلى char (مثلًا) وتفعيل البند "التعريف حسب التنفيذ"؟ الإجابة هي لا، لأن عملية الإسناد لا تضمّ ترقية الأعداد الصحيحة، لذا لا تقلق.

ج. الأعداد العشرية والصحيحة

يتخلّص تحويل نوع عدد عشري floating إلى نوع عدد صحيح بسيط من جميع الأجزاء العشرية للقيمة، فإذا كان نوع العدد الصحيح غير قابل لتخزين القيمة المتبقية، فسيصبح لدينا سلوك غير محدد، أي حالة شبيهة بالطفحان overflow.

كما ذكرنا سابقًا، لا توجد هناك أي مشكلة إذا حدث التحويل بصورة تصاعدية من float إلى double إلى long double، إذ من الممكن لجميع الأنواع السابقة تخزين جميع القيم التي تتسع في الأنواع الأصغر منها، وبذلك تحصل عملية التحويل دون أي فقدان للمعلومات؛ بينما سينتج في التحويل في الاتجاه المعاكس سلوك غير محدد في حال كانت القيمة خارج مجال القيم التي يمكن للنوع تخزينها، وفي حال كانت القيمة ضمن المجال ولكن لا يمكن تخزينها بدقتها بالضبط النتيجة، ستكون واحدة من القيمتين المجاورتين الممكن تخزينها، ويجري اختيارها حسب التنفيذ، وهذا يعني أن القيمة ستفقد جزءًا من دقتها.

د. التحويلات الحسابية الاعتيادية

هناك العديد من التعبيرات التي تتضمن استخدام تعابير فرعية subexpressions تحتوي على خليط من الأنواع مع عوامل، مثل "+" و "*" وما شابه. إذا كانت للمعاملات ضمن التعبير عدّة أنواع، فهذا يعني أن هناك بعض التحويلات الواجب إجراؤها حتى تكون النتيجة النهائية من نوع معين، والتحويلات هي:

- إذا كان أي من المُعاملين من نوع long double يُحوّل المُعامل الآخر إلى long double ويكون هذا هو نوع النتيجة.
- ماعدا ذلك، إذا كان أي من المُعاملين من نوع double، يُحوّل المُعامل الآخر إلى double ويكون هذا هو نوع النتيجة.
- ماعدا ذلك، إن كان أي من المُعاملين من نوع float، يُحوّل المُعامل الآخر إلى float ويكون هذا هو نوع النتيجة.
- ماعدا ذلك، تُطبّق الترقية العددية الصحيحة لكلا المُعاملين حسب التحويلات التالية:
 - إذا كان أي من المُعاملين من نوع unsigned long int، يُحوّل المُعامل الآخر إلى unsigned long int ويكون هذا نوع النتيجة.
 - ماعدا ذلك، إذا كان أي من المُعاملين من نوع long int، يُحوّل المُعامل الآخر إلى long int ويكون هذا هو نوع النتيجة.
 - ماعدا ذلك، إذا كان أي من المُعاملين من نوع unsigned int، يُحوّل المُعامل الآخر إلى unsigned int ويكون هذا نوع النتيجة.
 - ماعدا ذلك، يجب أن يكون كلا المُعاملين من نوع int وعلى هذا نوع النتيجة أيضًا.

يتضمن المعيار جملة غريبة: "يمكن تمثيل قيم المُعاملات من نوع الأعداد العشرية ونتائج تعابيرها بدقة ومجال أكبر من المطلوبة بالنسبة لنوعها، بالتالي لا يحدث تغيير للأنواع". السبب في هذا هو الحفاظ على معاملة لغة سي القديمة للمتغيرات من أنواع الأعداد العشرية، إذ كانت تُرقى المتغيرات من نوع float في سي القديمة تلقائيًا إلى double بالطريقة ذاتها التي تُرقى متغيرات من نوع char إلى int، لذلك يمكن إنجاز التعبير الذي يحوي متغيرات من نوع float فقط كما لو كانت المتغيرات من نوع double، ولكن نوع النتيجة سيكون دائمًا float.

التأثير الوحيد لهذه العملية هو على حساب الأداء، وهو غير مهم لمعظم المستخدمين؛ ويُحدّد ما إذا كانت التحويلات ستُطبّق أم لا، وأي نوع منها سيُطبّق، عند الوصول إلى العامل operator.

لا تسبّب التحويلات بين الأنواع ومزجها أي مشكلات عمومًا، ولكن هناك بعض النقاط التي يجب الانتباه إليها؛ إذ يُعد المزج بين الأنواع ذات الإشارة وعديمة الإشارة بسيطًا إلى أن يحتوي النوع ذو الإشارة قيمة سالبة،

إذ لا يمكن تمثيل قيمته باستخدام متغير عديم الإشارة، وعلينا إيجاد حل لهذه المشكلة. ينص المعيار على أن نتيجة تحويل عدد سالب إلى نوع عديم الإشارة هي أكبر قيمة يمكن تخزينها في النوع عديم الإشارة زائد واحد مضافةً إلى العدد السالب، ولأن الطفحان غير ممكن الحدوث في الأنواع عديمة الإشارة فالنتيجة دائماً معرّفة على المجال. ولتأخذ `int` بطول 16 بت مثلاً، إذ أن مجال النوع عديم الإشارة هو 0 إلى 65535، وبتحويل قيمة سالبة (ولتكن "-7") للنوع هذا يجب إضافة 7- إلى 65536 الذي يعطينا الناتج 65529.

يحتفظ المعيار بالطريقة القديمة في لغة سي، إذ يُسند نمط البتات في الرقم ذي الإشارة إلى الرقم عديم الإشارة، والطريقة التي يصفها المعيار هي الطريقة ذاتها التي تنتج عن إسناد نمط بتات على حاسوب يعمل بنظام المتمم الثنائي، وعلى أنظمة المتمم الأحادي أن تبذل مزيداً من المجهود لتصل للنتيجة المرجوة.

لتوضيح الأمر أكثر، سينتج عن رقم صغير سالب رقم كبير موجب عند تحويله إلى نوع عديم الإشارة، وإذا لم تُعجبك هذه الطريقة فحاول التفكير بطريقة أفضل من هذه. يُعد إسناد رقم سالب إلى متغير عديم الإشارة خطأً فادحاً، وستكون عواقب هذا الخطأ على عاتقك.

من السهل القول "لا تفعل هذا"، ولكن الأمر قد يحدث عن طريق الخطأ وفي هذه الحالة ستكون النتائج مفاجئة جداً. ألق نظرةً على المثال التالي:

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int i;
    unsigned int stop_val;

    stop_val = 0;
    i = -10;

    while(i <= stop_val){
        printf("%d\n", i);
        i = i + 1;
    }
    exit(EXIT_SUCCESS);
}
```

[مثال 7.2]

ربما تتوقع أن يطبع البرنامج لأحة قيم من "-10" إلى "0"، لكن هذا خاطئ، إذ تكمن المشكلة هنا في الموازنة؛ أي يُوازَن المتغير `i` الذي يخزن القيمة -10 مع متغير عديم الإشارة يخزن القيمة 0، ووفقاً لقواعد

الحساب (استذكرها إن أردت) يجب أن نحول كلا النوعين إلى `unsigned int` أولاً ومن ثم نجري الموازنة، ونصح القيمة 10- مساوية 65526 على الأقل (تفقد ملف الترويسة `<limits.h>`) بعد تحويلها، وتوازن فيما بعد مع 0 وهي أكبر من القيمة كما هو واضح، وبذلك لا تُنفذ الحلقة التكرارية إطلاقاً. العبرة هنا هو أنه عليك تجنّب استخدام الأعداد عديمة الإشارة إلا في حالة استخدامك المقصود لها، وعندما تستخدمها انتبه جيّداً بخصوص مزجها مع الأعداد ذات الإشارة.

ه. المحارف العريضة

كما ذكرنا سابقاً، يسمح المعيار بمجموعات المحارف الموسّعة، إذ يمكنك استخدام ترميز الإدخال بالإزاحة `shift-in` والإخراج بالإزاحة `shift-out`، التي تسمح بتخزين المحارف متعددة البايتات في سلاسل نصية اعتيادية في لغة سي، والتي في حقيقة الأمر مصفوفات من نوع `char` كما سنتعرف لاحقاً؛ أو يمكنك استخدام التمثيل الذي يستخدم أكثر من بايت واحد لتخزين كل محرف من المحارف. يمكننا استخدام سلاسل الإزاحة فقط في حالة معالجة المحارف بترتيب محدد، إذ إن الطريقة عديمة الفائدة في حال أردت إنشاء مصفوفة محارف والوصول إليهم بغض النظر عن ترتيبهم. إليك مثلاً استخدمناه سابقاً مضافاً إليه أدلة `indexes` مصفوفة منطقية وفعلية:

```
1 2 3 4 5 6 7 8 9 (actual array index)
a b c <SI> a b g <SO> x y
1 2 3 4 5 6 7 (logical index)
```

حتى لو استطعنا الوصول إلى المُدخلة "الصالحة `correct`" ذات الدليل "5" في المصفوفة، فلن تنتهِ المشكلة، إذ لا يمكن تمييز النتيجة التي حصلنا عليها إن كانت مرّزة أو هي "g" حرفياً. الحل الواضح لهذه المشكلة هو استخدام قيم مميّزة لجميع المحارف في مجموعة المحارف التي نستخدمها، لكن هذا الأمر يتطلب مزيداً من البتات الموجودة في `char` اعتيادي، وأن نكون قادرين على تخزين كل قيمة على نحوٍ منفصل دون استخدام تقنية الإزاحة أو أي تقنية تعتمد على موضع القيم، وهذا هو الغرض من استخدام النوع `wchar_t`؛ إذ يُعدّ هذا النوع مرادفاً لأنواع الأعداد الصحيحة الأخرى (يمكنك الاطلاع على تعريفه في ملف الترويسة `<stddef.h>`)، وهو نوعٌ معرّف حسب التنفيذ ويُستخدم في تخزين المحارف الموسّعة عندما تريد إنشاء مصفوفة منها. يضمن المعيار التفاصيل التالية المتعلقة بقيمة المحارف العريضة:

- يستطيع المتغير من نوع `wchar_t` تخزين قيم فريدة لكل محرف من أكبر مجموعة محارف يدعمها التنفيذ.
- المحرف الفارغ `null` قيمته الصفر.
- تماثل قيمة ترميز كل محرف من مجموعة المحارف الأساسية (ألقى نظرةً على فصل المحارف المُستخدمة في لغة C فقرة الأبجدية الاعتيادية) في النوع `wchar_t` القيمة المُخزنة في `char`.

هناك دعم أكبر لطريقة ترميز المحارف هذه، مثل السلاسل النصية strings التي تكلمنا عنها سابقًا، إذ تُنقَد على أنها مصفوفة من المحارف char، مع أن قيمتها تبدو على النحو التالي:

```
"a string"
```

للحصول على سلاسل نصية من نوع wchar_t، اكتب السلسلة النصية كما هي مسبوقة بالحرف L، على سبيل المثال:

```
L"a string"
```

علينا أن نفهم الفرق بين المثالين السابقين، إذ أن السلاسل النصية هي في حقيقة الأمر مصفوفات وعلى الرغم من غرابة الأمر إلا أننا نستطيع استخدام دليل المصفوفة عليها:

```
"a string"[4]
L"a string"[4]
```

كلا التعبيرين السابقين صالح، إذ أن التعبير الأول من نوع char وقيمه ممثلةً بالحرف r (تذكر أن دليل المصفوفات يبدأ من صفر وليس واحد)، والتعبير الثاني من نوع wchar_t وقيمه ممثلةً أيضًا بالحرف r. يصبح الأمر مثيرًا للاهتمام عند استخدامنا للمحارف الموسعة، إذ تظهر لنا بعض المشكلات إذا استخدمنا الترميز <a> و للدلالة على محارف "إضافية" عن مجموعة المحارف الاعتيادية، أي ترميز هذه المحارف باستخدام تقنية إزاحة ما، لاحظ المثالين:

```
"abc<a><b>"[3]
L"abc<a><b>"[3]
```

الحالة الثانية سهلة الفهم، فهي مصفوفة من نوع wchar_t والترميز الموافق لها يبدأ بالمحرف <a> أيًا كان هذا الترميز (لنفترض أنه ترميز إلى الحرف اليوناني الموافق)؛ أما الحالة الأولى فهي غير ممكنة التنبؤ، إذ أن النوع هو char بلا شك لكن قيمته هي وسم الإدخال بالإزاحة غالبًا.

كما هو الحال مع السلاسل النصية، هناك ثوابت محارف عريضة، مثل 'a' التي لها نوع char وقيمة الترميز متجاوبة مع قيمة المحرف a، أما المحرف التالي:

```
L'a'
```

فهو ثابت من نوع wchar_t، وعند استخدام المحارف متعددة البايتات في المثال الذي سبقه، فهذا يعني أن قيمته تساوي محارف متعددة في محرف ثابت واحد على سبيل المثال:

```
'xy'
```

في الحقيقة، يُعد هذا التعبير صحيحًا ولكنه يعني شيئًا طريفًا. وسيُحوَّل المحرف متعدد البايتات في المثال الثاني إلى قيمة `wchar_t` الموافقة.

إذا لم تفهم جميع التفاصيل المتعلقة بالمحارف العريضة، فكل ما هنالك قوله هو أننا حاولنا أفضل ما لدينا لشرحها، عُد مرةً أخرى لاحقًا وقرأها من جديد، لعل التفاصيل تصبح مفهومة عندها. تدعم المحارف العريضة عمليًا استخدام مجموعات المحارف الموسعة في لغة سي وستفهمها حالما تعتاد عليها.

تمرين 15.2: بفرض أن أحجام الأنواع `char` و `int` و `long` هي 8 بت و 16 بت و 32 بت على الترتيب، وأن `char` يُحوَّل افتراضيًا إلى `unsigned char` في نظام ما، ما هو النوع الناتج من التعابير التي تتضمن خليطًا من المتغيرات التالية، بعد تطبيق عمليات التحويل الحسابية؟

1. متغيرات من نوع `signed char`.

2. متغيرات من نوع `unsigned char`.

3. نوع `int` و `unsigned int`.

4. نوع `int` و `unsigned long`.

5. نوع `char` و `long`.

6. نوع `char` و `float`.

7. نوع `float` و `float`.

8. نوع `float` و `long double`.

و. التحويل بين الأنواع Cast

في بعض الأحيان، ينتج نوع بيانات من تعبير ما ولكنك لا تريد استخدام هذا النوع، وتريد تحويله قسرًا إلى نوع مختلف، وهذا هو الغرض من **التحويل بين الأنواع casts**. عند وضع اسم النوع بين قوسين على النحو التالي:

```
(int)
```

فأنت تنشئ هنا عاملًا أحاديًا unary operator يُسمَّى بالتحويل بين الأنواع `cast`، إذ يغير التحويل بين الأنواع قيمة التعبير الواقع على يمينه إلى النوع المحدد بداخل الأقواس. على سبيل المثال، إذا كنت تجري عملية القسمة بين عددين صحيحين `a/b` فسيستخدم التعبير الناتج قسمة الأعداد الصحيحة ويتخلص من أي باقي، ويمكنك استخدام متغيرات وسيطة من نوع أعداد عشرية للحفاظ على الجزء العشري من القيمة الناتجة أو استخدام التحويل بين الأنواع. يوضح المثال التالي الطريقتين:

```
#include <stdio.h>
#include <stdlib.h>

/*
 * Illustrates casts.
 * For each of the numbers between 2 and 20,
 * print the percentage difference between it and the one
 * before
 */
main(){
    int curr_val;
    float temp, pcnt_diff;

    curr_val = 2;
    while(curr_val <= 20){
        /*
         * % difference is
         * 1/(curr_val)*100
         */
        temp = curr_val;
        pcnt_diff = 100/temp;
        printf("Percent difference at %d is %f\n",
            curr_val, pcnt_diff);
        /*
         * Or, using a cast:
         */
        pcnt_diff = 100/(float)curr_val;
        printf("Percent difference at %d is %f\n",
            curr_val, pcnt_diff);
        curr_val = curr_val + 1;
    }
    exit(EXIT_SUCCESS);
}
```

[مثال 8.2]

الطريقة الأسهل لتذكر الاستخدام الصحيح للتحويل بين الأنواع هو كتابته وكأنك تُصرِّح عن متحول من نوع تريده، ومن ثم ضع الأقواس حول التصريح بالكامل واحذف اسم المتغير، مما سيعطيك التحويل بين الأنواع. يوضح الجدول 6 بعض الأمثلة البسيطة، قد تلاحظ أن بعض الأنواع لم تُقدِّم بعد، لكن سيتوضَّح التحويل بين الأنواع أكثر عند استخدام الأنواع المعقدة. تجاهل الأمثلة التي لا تفهمها بعد، لأنك ستكون قادرًا على استخدام هذا الجدول مثل مرجع لاحقًا.

الجدول 6: التحويل بين الأنواع

النوع	التحويل بين الأنواع	التصريح
int	(int)	int x;
float	(float)	float f;
مصفوفة من char	(char [30])	char x[30];
مؤشر إلى int	(int *)	int *ip;
مؤشر لدالة تُعيد النوع int	(int (*) ())	int (*f)();

2.8.2 العوامل Operators

يعرض هذا القسم مفهوم العوامل في لغة C وكيفية استخدامها في العمليات والتعليمات.

1. عوامل المضاعفة

تتضمن عوامل المضاعفة multiplicative operators عامل الضرب "*" والقسمة "/" وباقي القسمة "%"، ويعمل عاملا الضرب والقسمة بالطريقة التي تتوقع أن تعملان بها، لكل من الأنواع الحقيقية والصحيحة، إذ تنتج قيمة مقتطعة دون فواصل عشرية عند تقسيم الأعداد الصحيحة ويكون الاقتطاع باتجاه الصفر. يعمل عامل باقي القسمة وفق تعريفه فقط مع الأنواع الصحيحة، لأن القسمة الناتجة عن الأعداد الحقيقية لن تعطينا باقيًا.

إذا كانت القسمة غير صحيحة، وكان أي من المُعاملان غير سالب، فنتيجة العامل "/" هي موجبة ومقرّبة باتجاه الصفر، ونستخدم العامل "%" للحصول على الباقي من هذه العملية، على سبيل المثال:

```
9/2 == 4
```

```
9%2 == 1
```

إذا كان أحد المُعاملات سالب، فنتيجة العامل "/" قد تكون أقرب عدد صحيح لنتيجة القسمة على أي من الاتجاهين (باتجاه الأكبر أو الأصغر)، وقد تكون إشارة نتيجة العامل "%" موجبة أو سالبة، وتعتمد النتيجة السابقتين حسب تعريف التنفيذ. التعبير الآتي مساوٍ الصفر في جميع الحالات عدا حالة b مساوية للصفر.

```
(a/b)*b + a%b - a
```

تُطبَّق التحويلات الحسابية الاعتيادية على كلا المُعاملين.

ب. عوامل الجمع

تتضمن عوامل الجمع additive operators عامل الجمع "+" والطرح "-", وتتبع طريقة عمل هذه الدوال قواعدها المعتادة التي تعرفها. للعامل الثنائي والأحادي نفس الرمز، ولكن لكل واحد منهما معنى مختلف؛ فعلى سبيل المثال، يُستخدم التعبيران $a+b$ و $a-b$ عاملاً ثنائياً (العامل - للجمع و + للطرح).

إذا أردنا استخدام العوامل الأحادية بذات الرموز، فسنكتب $+b$ أو $-b$ ، ولعامل الطرح الأحادي وظيفة واضحة ألا وهي أخذ القيمة السالبة لقيمة المُعامل، ولكن ما وظيفة عامل الجمع الأحادي؟ في الحقيقة لا يؤدي أي دور؛ ويُعد عامل الجمع الأحادي إضافةً جديدةً إلى اللغة، إذ يعادل وجوده وجود عامل الطرح الأحادي ولكنه لا يؤدي أي دور لتغيير قيمة التعبير. القلة من مستخدمي لغة سي القديمة لاحظ عدم وجوده.

تُطبَّق التحويلات الحسابية الاعتيادية على كلٍّ من مُعاملات العوامل الثنائية (للجمع والطرح)، وتُطبَّق الترقية العددية الصحيحة على مُعاملات العوامل الأحادية فقط.

ج. عوامل العمليات الثنائية

واحدة من مزايا لغة سي هي الطريقة التي تسمح بها لمبرمجي النظم بالتعامل مع الشيفرة البرمجية وكأنها شيفرة تجميعية Assembly code، وهو نوع شيفرة برمجية كان شائعاً قبل مجيء لغة سي، وكان هذا النوع من الشيفرات صعب التشغيل على عدة منصات (غير محمول non-portable). كما وضحت لنا لغة سي أن هذا الأمر لا يتطلب سحراً لجعل الشيفرة محمولة، لكن ما هو هذا الشيء؟ هو ما يُعرف أحياناً باسم "العيب بالبتات bit-twiddling" وهي عملية التلاعب ببتات متغيرات الأعداد الصحيحة على نحوٍ منفرد. لا يمكن استخدام عوامل العمليات الثنائية bitwise operators على مُعاملات من نوع أعداد حقيقية إذ لا تُعد البتات الخاصة بها منفردة individual أو يمكن الوصول إليها.

هناك ستة عوامل للعمليات الثنائية موضحة في الجدول 7، الذي يوضح أيضاً نوع التحويلات الحسابية المُطبَّقة.

الجدول 7: عوامل العمليات الثنائية

العامل	التأثير	التحويل
&	العملية الثنائية AND	التحويلات الحسابية الاعتيادية
\	العملية الثنائية OR	التحويلات الحسابية الاعتيادية
^	العملية الثنائية XOR	التحويلات الحسابية الاعتيادية
<<	إزاحة إلى اليسار	الترقية العددية الصحيحة
>>	إزاحة إلى اليمين	الترقية العددية الصحيحة

العامل	التأثير	التحويل
~	المتمم الأحادي one's complement	الترقية العددية الصحيحة

العامل الوحيد الأحادي هو الأخير (المتمم الأحادي)، إذ يعكس حالة كل بت في قيمة المُعامل وله نفس تأثير عامل الطرح الأحادي في حاسوب يعمل بنظام المتمم الأحادي، لكن معظم الحواسيب الآن تعمل بنظام المتمم الثنائي، لذلك وجوده مهم.

سيسهّل استخدام النظام الست عشري عن استخدام النظام العشري فهم طريقة هذه العوامل، لذا حان الوقت الآن لأن نعرفك على الثوابت الست عشرية. أي رقم يُكتب في بدايته "0x" يفسّر على أنه رقم ست عشري، على سبيل المثال القيمة "15" و"0xf"، أو "0xF" متكافئتان، جرّب تشغيل المثال التالي على حاسوبك، أو الأفضل من ذلك تنبأً بوظيفة البرنامج قبل تشغيله.

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int x,y;
    x = 0; y = ~0;

    while(x != y){
        printf("%x & %x = %x\n", x, 0xff, x&0xff);
        printf("%x | %x = %x\n", x, 0x10f, x|0x10f);
        printf("%x ^ %x = %x\n", x, 0xf00f, x^0xf00f);
        printf("%x >> 2 = %x\n", x, x >> 2);
        printf("%x << 2 = %x\n", x, x << 2);
        x = (x << 1) | 1;
    }
    exit(EXIT_SUCCESS);
}
```

[مثال 2.9]

لنتكلم أولاً عن طريقة عمل الحلقة التكرارية في مثالنا، إذ أن المتغير الذي يتحكم بالحلقة هو x ومُهيّأ بالقيمة صفر، وفي كل دورة تُوازن قيمته مع قيمة y الذي صُيِّط بنمطٍ مستقل بنفس طول الكلمة ومكون من الواحدات، وذلك بأخذ المتمم الأحادي للصفر، وفي أسفل الحلقة يُزاح المتغير x إلى اليسار مرةً واحدةً وتُجرى العملية الثنائية OR عليه، مما ينتج سلسلةً تبدأ على النحو التالي: "0، 1، 11، 111، ..." بالنظام الثنائي.

تُجرى العمليات الثنائية على x باستخدام كل من عوامل AND و OR و XOR (أي OR الحصرية أو دارة عدم التماثل) إضافةً إلى مُعاملات أخرى جديرة بالاهتمام، ومن ثم تُطبع النتيجة.

نجد أيضًا عوامل الإزاحة إلى اليمين واليسار، التي تعطينا نتيجةً بنوع وقيمة المُعامل الموجود على الجهة اليسرى مزاحةً إلى الجهة المحددة عددًا من المراتب حسب المُعامل الموجود على جهتها اليمنى، ويجب أن يكون المُعاملان أعدادًا صحيحة. تختفي البتات المُزاحة إلى أي من طرفي المُعامل الأيسر، وينتج عن إزاحة مقدار من البتات أكبر من البتات الموجودة في الكلمة نتيجةً معتمدةً على التنفيذ.

تضمن الإزاحة إلى اليسار إزاحة الأصفار إلى البتات منخفضة الترتيب، بينما تكون الإزاحة إلى اليمين أكثر تعقيدًا، إذ إن الأمر متروك لتنفيذك للاختيار بين إجراء إزاحة منطقية أو حسابية إلى اليمين عند إزاحة المُعاملات ذات الإشارة. هذا يعني أن الإزاحة المنطقية تُزيح الأصفار باتجاه البت ذو الأهمية، بينما تسخ الإزاحة الحسابية محتوى البت الأكثر أهمية الحالي إلى البت نفسه، ويصبح الخيار أوضح إذا أُزيح مُعامل عديم الإشارة إلى اليمين، ولا يوجد أي خيار هنا، إذ يجب أن تكون الإزاحة منطقية. ولهذا السبب يجب أن تتوقع أن تكون القيمة المُزاحة عند استخدام الإزاحة إلى اليمين مصرحًا عنها مثل قيمةٍ عديمة الإشارة، أو أن يُحوّل نوعها `cast` إلى عديمة الإشارة لإجراء عملية الإزاحة كما يصف المثال التالي ذلك:

```
int i, j;
i = (unsigned)j >> 4;
```

لا ينبغي على المُعامل الثاني (على الطرف الأيمن) لعامل الإزاحة أن يكون ثابتًا، إذ من الممكن استخدام أي دالة ذات نوع عدد صحيح؛ ومن المهم هنا الإشارة إلى أن قوانين مزج أنواع المُعاملات لا تنطبق على عوامل الإزاحة، إذ أن نوع نتيجة الإزاحة هي النوع المُزاح ذاته (بعد الترقية العددية الصحيحة) ولا تعتمد على أي شيءٍ آخر.

لنتكلم عن شيءٍ مختلف قليلًا، وهي إحدى الحيل المفيدة التي يستخدمها مبرمجو لغة سي لكتابة برامجهم على نحوٍ أفضل. إذا أردت تشكيل قيمةٍ تحتوي على واحدات "1" في جميع خاناتها عدا البت الأقل أهمية، بهدف تخزين نمطٍ آخر فيها، فمن غير المطلوب معرفة طول الكلمة في النظام الذي تستخدمه. على سبيل المثال، تستطيع استخدام الطريقة التالية لضبط البتات الأقل ترتيبًا لمتغيرٍ من نوع `int` إلى `0x0f0` وجميع البتات الأخرى إلى 1:

```
int some_variable;
some_variable = ~0xf0f;
```

أُجري المتمم الأحادي على المتمم الأحادي لنمط البتات منخفضة الترتيب المرغوب، وهذا يُعطي بدوره القيمة المطلوبة والمستقلة تمامًا عن طول الكلمة، وهو شيء متكرر الحدوث في شيفرة لغة سي.

لا يوجد هناك مزيداً من الأشياء لقولها عن عوامل التلاعب بالبتات، وتجربتنا في تعليم لغة سي تدلنا على أنها سهلة الفهم والتعلم من معظم الناس، لذا دعنا ننتقل للموضوع التالي.

د. عوامل الإسناد

العنوان ليس خطأً مطبعياً بل قصدنا "عوامل" بالجمع، إذ أن لدى لغة سي عدة عوامل إسناد على الرغم من رؤيتنا للعامل "=" فقط حتى الآن. المثير للاهتمام بخصوص هذه العوامل هو أنها تعمل مثل العوامل الثنائية الأخرى، إذ تأخذ مُعاملين وتعطينا نتيجة. وتستخدم النتيجة لتكون جزءاً من التعبير. مثلاً في هذا التعبير:

```
x = 4;
```

تُسند القيمة 4 إلى المتغير x، والنتيجة عن إسناد القيمة إلى المتغير x هو استخدامها على النحو التالي:

```
a = (x = 4);
```

إذ ستخزن a القيمة 4 المُسندة إليها بعد أن تُسند إلى x. تجاهلت جميع عمليات الإسناد السابقة التي نظرنا إليها لحد اللحظة (عدا مثال واحد) ببساطة القيمة الناتجة عن عملية الإسناد بالرغم من وجودها.

بفضل هذه القيمة، يمكننا كتابة تعابير مشابهة لهذه:

```
a = b = c = d;
```

إذ تُسند قيمة المتغير d إلى المتغير c وتُسند هذه النتيجة إلى b وهكذا دواليك، ونلاحظ هنا معالجة تعابير الإسناد من الجهة اليمنى إلى الجهة اليسرى، ولكن ما عدا ذلك فهي تعابير اعتيادية (القوانين التي تصف اتجاه المعالجة من اليمين أو من اليسار موجودة في الجدول 9.2).

هناك وصف موجود في القسم الذي يتكلم عن "التحويلات"، يصف ما الذي سيحدث في حالة التحويل من أنواع طويلة إلى أنواع قصيرة، وهذا ما يحدث عندما يكون المُعامل الذي يقع على يسار عامل الإسناد البسيط أقصر من المُعامل الذي يقع على يمينه.

عوامل الإسناد المتبقية هي عوامل إسناد مركّبة، وتعد اختصارات مفيدة يمكن اختصارها عندما يكون المُعامل ذاته على يمين ويسار عامل الإسناد، على سبيل المثال، يمكن اختصار التعبير التالي:

```
x = x + 1;
```

إلى التعبير:

```
x += 1;
```

وذلك باستخدام إحدى عوامل الإسناد المركّبة، وتكون النتيجة للتعبير الأول هي ذاتها للتعبير الثاني المختصر في أي حالة. يصبح هذا الأمر مفيداً عندما يكون الجانب الأيسر من العامل تعبيراً معقداً، وليس متغيراً

وحيثاً، وذلك في حالة استخدام المصفوفات والمؤشرات. يميل معظم مبرمجي لغة سي لاستخدام التعبير في المثال الثاني لأنه يبدو "أكثر منطقية"، وهذا ما يختلف عنده الكثير من المبتدئين عند تعلّم هذه اللغة. يوضح الجدول 8 عوامل الإسناد المركبة، وستلاحظ استخدامنا المكثف لها من الآن فصاعداً.

الجدول 8: عوامل الإسناد المركبة

*=

+=

&=

>>=

تُطبّق التحويلات الحسابية في كل حالة وكأنها تُطبق على كامل التعبير، أي كأن التعبير $a+=b$ مثلاً مكتوباً على النحو $a=a+b$.

للتذكير، تحمل نتيجة عامل الإسناد نوع وقيمة الكائن الذي أُسند إليه.

ه. عوامل الزيادة والنقصان

قدّمت لغة سي عاملين أحاديين مميزين لإضافة واحد أو طرحه من تعبير ما نظراً لشيوع هذه العملية البسيطة؛ إذ يضيف عامل الزيادة "++" واحداً، وي طرح عامل النقصان "--" واحداً، وتُستخدم هذه العوامل على النحو التالي:

```
x++;
++x;
x--;
--x;
```

إذ من الممكن أن يقع العامل قبل أو بعد المُعامل، ولا يختلف عمل العامل في الحالات الموضحة سابقاً حتى لو اختلف موقعه، ولكن الحالة تصبح أكثر تعقيداً في بعض الأحيان ويصبح الفرق وفهمه مهمّاً للاستخدام الصحيح.

إليك الفرق موضحاً في المثال التالي:

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int a,b;
    a = b = 5;
    printf("%d\n", ++a+5);
```

```
printf("%d\n", a);
printf("%d\n", b++ +5);
printf("%d\n", b);
exit(EXIT_SUCCESS);
}
```

[مثال 10.2]

خرج المثال السابق هو:

```
11
6
10
6
```

يعود السبب في الفرق في النتائج إلى تغيير مواضع العوامل؛ فإذا ظهر عامل الزيادة أو النقصان قبل المتغير، فستتغير القيمة بمقدار واحد و تُستخدم القيمة **الجديدة** ضمن التعبير؛ أما إذا ظهر العامل بعد المتغير فستُستخدم القيمة **القديمة** في التعبير، ثم تُغير قيمة المتغير.

لا يستخدم مبرمجو سي عادةً التعليمة التالية لطرح أو جمع واحد:

```
x += 1;
```

بل يستخدمون التعليمة التالية:

```
x++; /* or */ ++x;
```

ينبغي تجنُّب استخدام المتغير ذاته أكثر من مرة في ذات التعبير إذا كان هذا النوع من العوامل مرتبطًا به، إذ لا يوجد هناك أي قاعدة واضحة تدلك على الجزء المحدد من التعبير الذي ستتغير فيه قيمة المتغير. قد يختار المصنّف "حفظ" جميع التغييرات وتطبيقها دفعةً واحدة، فعلى سبيل المثال لا يضمن التعبير التالي إسناد قيمة x الأصلية مرتين إلى y :

```
y = x++ + --x;
```

وقد يُقيّم كما لو كان قد جرى توسعته إلى التعبير التالي:

```
y = x + (x-1);
```

لأن المصنّف يلاحظ عدم وجود تأثير أبدًا على قيمة x .

تُجرى العمليات الحسابية في هذه الحالة تمامًا كما في حالة تعبير جمع، مثلًا $x=x+1$ ، وتُطبَّق التحويلات الحسابية الاعتيادية.

تمرين 16.2: بالنظر إلى تعريف الدالتين التاليتين:

```
int i1, i2;
float f1, f2;
```

1. كيف يمكنك إيجاد باقي القسمة عند تقسيم $i1$ على $i2$ ؟
 2. كيف يمكنك إيجاد باقي القسمة عند تقسيم $i1$ على $f1$ ، إذ أن $f1$ عدد صحيح؟
 3. ما الذي يمكنك تنبؤه بخصوص إشارة بواقي القسمة الناتجة من العمليتين السابقتين؟
 4. ما هي المعاني التي قد يحملها العامل -؟
 5. كيف يمكنك تحييد جميع البتات عدا البتات الأربع الأقل ترتيبًا في $i1$ ؟
 6. كيف يمكنك تشغيل جميع البتات الأربع الأقل ترتيبًا في $i1$ ؟
 7. كيف يمكنك تحييد البتات الأربع الأقل ترتيبًا فقط في $i1$ ؟
 8. كيف يمكنك وضع قيمة الثماني بتات الأقل ترتيبًا للمتغير $i2$ في $i1$ ، مع قلب أهمية البتات الأربع الأقل ترتيبًا من البتات التي تليها؟
- ما الخطأ في التعبير التالي؟

```
f2 = ++f1 + ++f1;
```

و. الأسبقية والتجميع

علينا النظر إلى الطريقة التي تعمل بها هذه العوامل بعد التكلم عنها. قد تعتقد أن عملية الجمع ليست بتلك الأهمية، فنتيجة التعبير

```
a + b + c
```

مساوية للتعبير:

```
(a + b) + c
```

أو التعبير:

```
a + (b + c)
```

أليس كذلك؟ في الحقيقة لا، فهناك فرقٌ بين التعابير السابقة؛ فإذا تسبَّب التعبير $a+b$ بحالة طفحان، وكانت قيمة المتغير c قريبة من قيمة $-b$ ، فسيعطي التعبير الثاني الإجابة الصحيحة، بينما سيتسبب الأول بسلوك غير محدد. يمكننا ملاحظة هذه المشكلة بوضوح أكبر باستخدام قسمة الأعداد الصحيحة، إذ يعطي التعبير التالي:

$$a/b/c$$

نتائج مختلفة تمامًا عند تجميعه بالطريقة:

$$a/(b/c)$$

أو بالطريقة:

$$(a/b)/c$$

يمكنك تجربة استخدام القيم $a=10$ و $b=2$ و $c=3$ للتأكد، إذ سيكون التعبير الأول: $10/(2/3)$ ، ونتيجة $2/3$ في قسمة الأعداد الصحيحة هي 0 ، لذلك سنحصل على $10/0$ ، مما سيسبب طفحاً `overflow`؛ بينما سيعطينا التعبير الثاني القيمة $(10/2)$ ، وهي 5 ، وبتقسيمها على 3 تعطينا 1 .

يُعرّف تجميع العوامل على هذا النحو بمصطلح **الارتباط associativity**، والمكون الثاني لتحديد طريقة عمل العوامل هو **الأسبقية precedence**، إذ لبعض العوامل أسبقية عن عوامل أخرى، وتُحسب قيم العوامل هذه في التعابير الفرعية أولاً قبل الانتقال إلى العوامل الأقل أهمية. تُستخدم قوانين الأسبقية في معظم لغات البرمجة عالية المستوى. "نعلم" أن التعبير:

$$a + b * c + d$$

يُجمَع على النحو التالي:

$$a + (b * c) + d$$

إذ أن عملية الضرب ذات أسبقية أعلى موازنةً بعملية الجمع.

تحظى لغة سي بوجود 15 مستوى أسبقية بفضل مجموعة العوامل الكبيرة التي تحتويها، يحاول القلة من الناس تذكرها جميعاً. يوضح الجدول 9 جميع المستويات، ويصف كلاً من الأسبقية والارتباط. لم نتكلم عن جميع العوامل المذكورة في الجدول بعد. كن حذراً من استخدام نفس الرمز لبعض العوامل الأحادية والثنائية، والموضحة في الجدول أيضاً.

الجدول 9: أسبقية العوامل وترابطها

ملاحظات	الاتجاه	العامل
1	من اليسار إلى اليمين	() [] -> .
جميع العوامل أحادية	من اليمين إلى اليسار	sizeof & * (cast) + - -- ++ ~ !
عوامل ثنائية	من اليسار إلى اليمين	% / *
عوامل ثنائية	من اليسار إلى اليمين	- +
عوامل ثنائية	من اليسار إلى اليمين	<< >>
عوامل ثنائية	من اليسار إلى اليمين	<= < >= >
عوامل ثنائية	من اليسار إلى اليمين	!= ==
عامل ثنائي	من اليسار إلى اليمين	&
عامل ثنائي	من اليسار إلى اليمين	^
عامل ثنائي	من اليسار إلى اليمين	\
عامل ثنائي	من اليسار إلى اليمين	&&
عامل ثنائي	من اليسار إلى اليمين	\ \
2	من اليمين إلى اليسار	?:
عوامل ثنائية	من اليمين إلى اليسار	+ = وجميع عوامل الإسناد المركبة
عامل ثنائي	من اليسار إلى اليمين	,

1. الأقواس بهدف تجميع التعابير، وليس استدعاء الدوال.

2. هذا العامل غير مألوف، راجع القسم 1.4.3. مثال عن تنفيذ عملية الدخل.

السؤال هنا هو، كيف أستطيع الاستفادة من هذه المعلومات؟ من المهم طبعًا أن تكون قادرًا على كتابة تعابير تعطي قيمًا صحيحة بمعرفة ترتيب تنفيذ العمليات، إضافةً إلى فهم وقراءة تعابير مكتوبة من مبرمجين آخرين. تبدأ خطوات كتابة تعبير أو قراءة تعبير مكتوب بالعثور على العامل الأحادي والمعاملات المرتبطة معه، وهذه ليست بالمهمة الصعبة ولكنها تحتاج إلى بعض التمرين، بالأخص عندما تعرف أنه يمكن استخدام العوامل عددًا من المرات الاعتبارية بجانب مُعاملاتها، مثل التعبير التالي باستخدام العامل * الأحادي:

```
a*****b
```

يعني التعبير السابق أن المتغير a مضروبٌ بقيمة ما، وهذه القيمة هي تعبير يتضمن b وعددًا من عوامل * الأحادية.

تحديد العوامل الأحادية في التعبير ليست بالمهمة الصعبة، إليك بعض القواعد التي يجب أن تلجأ إليها:

1. العاملان "++" و "-" أحاديان في جميع الحالات.

2. العامل الذي يقع على يمين المُعامل مباشرةً هو عامل ثنائي (في حالة لم تتحقق القاعدة 1)، إذا كان العامل الذي يقع على يمين المُعامل ذاك ثنائيًا.

3. جميع العوامل الواقعة على يسار المُعامل أحادية (في حالة لم تتحقق القاعدة 2).

يمكنك دائمًا التفكير بعمل العوامل الأحادية أولاً قبل العوامل الأخرى بسبب أسبقيتها المرتفعة، إذ من الأشياء التي يجب عليك الانتباه عليها هي مواضع العاملين "++" و "--" إذا كانا قبل أو بعد المُعامل، فعلى سبيل المثال يحتوي التعبير التالي:

```
a + -b++ + c
```

على عاملين أحاديين مُطبَّقان على b. ترتبط العوامل الأحادية من اليمين إلى اليسار، إذًا على الرغم من قدوم - أولاً، يُكتب التعبير على النحو التالي (باستخدام الأقواس للتوضيح):

```
a + -(b++) + c
```

تصبح الحالة أكثر وضوحًا إذا استُخدم العامل في البداية prefix بدلاً من النهاية postfix مثل عامل زيادة أو نقصان، ويكون الترتيب من اليمين إلى اليسار في هذه الحالة أيضًا، ولكن العوامل ستظهر متتاليةً بجانب بعضها بعضًا.

بعد تعلُّمنا لطريقة فهم العوامل الأحادية، أصبح من السهل قراءة التعبير من اليسار إلى اليمين، وعندما تجد عاملًا ثنائيًا أبقيه في بالك، وانظر إلى يمينه؛ فإذا كان العامل الثنائي التالي ذو أسبقية أقل فسيكون العامل الذي تنظر إليه (الذي تبقى في بالك) هو جزء من تعبير جزئي عليك تقييم قيمته قبل أي خطوة أخرى؛ أما إذا كان العامل الثنائي التالي من نفس الأسبقية فأعد تنفيذ العملية حتى تصل إلى عامل ذو أسبقية مختلفة؛ وعندما تجد عاملًا ذا أسبقية منخفضة، قيّم قيمة التعبير الجزئي الواقع على يسار العامل وفقًا لقوانين الارتباط؛ أما إذا وجدت عاملًا ذا أسبقية عالية فانس جميع ما سبقه، إذ أن المُعامل الواقع على يسار العامل عالي الأسبقية هو جزء من تعبير جزئي آخر يقع على الطرف الأيسر وينتمي إلى العامل الجديد.

لا تقلق إذا لم تُصح لديك الصورة بعد، إذ يواجه العديد من مبرمجي لغة سي مشكلات تتعلق بهذه النقطة، ويعتادون فيما بعد على تجميع التعابير "بنظرة عين"، دون الحاجة لتطبيق القوانين مباشرةً.

لكن الأمر **المهم** هنا هو الخطوة التي تلي تجميعك لهذه التعابير، أتذكر "التحويلات الحسابية الاعتيادية"؟ إذ فسّرت هذه التحويلات كيف يمكنك التنبؤ بنوع التعبير عن طريق النظر إلى المُعاملات الموجودة. والآن إذا مزجت أنواعًا مختلفة في تعبير معقد، ستُحدّد أنواع التعابير الجزئية فقط من خلال أنواع المُعاملات الموجودة في التعبير الجزئي، ألقي نظرةً على المثال التالي:

```
#include <stdio.h>
#include <stdlib.h>
```



```
main(){
    int i,j;
    float f;

    i = 5; j = 2;
    f = 3.0;

    f = f + j / i;
    printf("value of f is %f\n", f);
    exit(EXIT_SUCCESS);
}
```

[مثال 11.2]

قيمة الخرج هي 3.0000 وليس 5.0000 مما يفاجئ البعض الذي اعتقد أن القسمة ستكون قسمة أعداد حقيقية فقط لأن متغير من نوع float كان موجودًا في التعليمة.

كان عامل القسمة بالطبع يحتوي على متغيرين من نوع int فقط على الجانبين، لذا أُجريت العملية الحسابية على أنها قسمة أعداد صحيحة وأنتجت صفرًا، واحتوى عامل الجمع على float و int على طرفيه، وبذلك -وحسب قوانين التحويلات الحسابية الاعتيادية- يُحوّل النوع int إلى float، وهو النوع الصحيح لإجراء عملية الإسناد أيضًا، مما أعفانا من تحويلات أخرى.

استعرض القسم السابق عن تحويل الأنواع casts طريقةً لتغيير نوع التعبير من نوعه الطبيعي إلى النوع الذي تريده، ولكن كُن حذرًا، إذ سيستخدم التحويل التالي قسمة صحيحة:

```
(float)(j/i)
```

ثم يحوّل النتيجة إلى float، وللمحافظة على باقي القسمة، يجب عليك كتابة التحويل بالطريقة:

```
(float)j/i
```

مما سيغير استخدام القسمة الحقيقية.

2.8.3 الأقسام

كما وضح المثال السابق، يمكنك تجاوز أسبقية وارتباط لغة سي الاعتيادية عن طريق استخدام الأقسام. لم يكن للأقسام أي معنى آخر في لغة سي القديمة، ولم تضمن أيضًا ترتيب تقييم القيمة ضمن التعابير، مثل:

```
int a, b, c;
```

```
a+b+c;
(a+b)+c;
a+(b+c);
```

إذا كان عليك استخدام متغيرات مؤقتة للحصول على ترتيب التقييم كما أردته، وهو أمر مهم إذا كنت تعرف أن هناك بعض التعبيرات التي تكون عرضةً للطفحان overflow، ولتجنّب هذا الأمر عليك أن تعدّل ترتيب تقييم التعبير.

ينص معيار سي على أن تقييم التعبير **يجب** أن يحدث بناءً على الترتيب المحدد وفق الأسبقية وتجميع التعبيرات، ويمكن للمصرّف أن يعيد تجميع التعبيرات إن لم يؤثر ذلك على النتيجة النهائية بهدف زيادة الكفاءة. على سبيل المثال، لا يمكن للمصرّف أن يعيد كتابة التعبير التالي:

```
a = 10+a+b+5;
```

إلى:

```
a = 15+a+b;
```

إلا في حالة تأكده من أن القيمة النهائية لن تكون مختلفةً عن التعبير الأولي الذي يتضمن قيم a و b ، وهذه الحالة محققةً إذا كان المتغيران من نوع عدد صحيح عديم الإشارة، أو عدد صحيح ذو إشارة وكانت العملية لا تسبب في بإطلاق استثناء عند التشغيل run-time exception والناتج عن طفحان.

2.8.4 الآثار الجانبية Side Effects

نعيد ونكرر التحذير الوارد بشأن عوامل الزيادة: ليس من الآمن استخدام المتغير ذاته أكثر من مرة في نفس التعبير، وذلك في حالة كان تقييم التعبير يغيّر من قيمة المتغير ويؤثر على القيمة النهائية للتعبير، وذلك بسبب التغيير (أو التغييرات) "المحفوظة" والمطبّقة فقط عند الوصول لنهاية التعليمة. على سبيل المثال التعبير $f = f + 1$ آمن على الرغم من ظهور المتغير f مرتين في تعبير يغير من قيمتها، والتعبير $f++$ آمن أيضًا، ولكن $f = ++f$ غير آمن.

تنبع المشكلة من استخدام عامل الإسناد أو عوامل الزيادة والنقصان أو استدعاء دالة تغيّر من قيمة متغير خارجي External مُستخدم في تعبير ما، وتُعرف هذه المشاكل باسم "الآثار الجانبية Side Effects"، ولا تحدد سي ترتيب حدوث هذه الآثار الجانبية ضمن تعبير ما (سنتوسّع لاحقًا بخصوص هذه المشكلة، ونناقش "نقاط التسلسل Sequence points").

2.9 الثوابت

يشرح هذا القسم الأعداد الحقيقية والصحيحة الثابتة وسلاسل التهريب في لغة C.

2.9.1 الأعداد الصحيحة الثابتة

مبدأ الأعداد الصحيحة الثابتة بسيط، فهي تمثل أي عدد صحيح مثل 1 أو 1034 وغيرها، ويمكنك إضافة الحرف `l` أو `L` في نهاية عدد صحيح ثابت لتحويله قسرياً إلى نوع `long`، وإضافة `u` أو `U` لتحويلها إلى `unsigned`، ويمكن كتابة الأعداد الصحيحة الثابتة بالنظام الست عشري عن طريق استخدام `0x` أو `0X` قبل كتابة الثابت، والأحرف `a` و `b` و `c` و `d` و `e` و `f`.

وتُكتب الأعداد الصحيحة الثابتة بالنظام الثماني أيضاً باستخدام `0` في بداية الرقم، وتستخدم الأرقام `0-1`، `2-3`، `4-5`، `6-7` فقط. عليك الحذر هنا بهذا الشأن، إذ من السهل النظر إلى `015` ومعاملته على أنه رقم صحيح بالنظام العشري، يقع المبتدئون في هذا الخطأ أغلب الأحيان، ولكنك ستبدأ بالاعتیاد على الأمر بعد اقتراف بعض الأخطاء.

قدّم معيار سي طريقة جديدة لمعرفة نوع العدد الصحيح الثابت، إذ تحدث ترقية `promoted` للثابت في لغة سي القديمة إلى النوع `long` في حال كان كبيراً ولا يتسع في النوع `int` دون أي تحذيرات، وتنص القاعدة على أن التحويل يجري بهذا الترتيب إلى أن تتسع قيمة الثابت بالنظام العشري:

```
int long unsigned long
```

بينما تستخدم الأعداد الست عشرية والثمانية الصحيحة هذه القائمة:

```
int unsigned int long unsigned long
```

إذا كان الثابت مسبوقةً بالحرف `"u"` أو `"U"`:

```
unsigned int unsigned long
```

إذا كان مسبوقةً بالحرف `"l"` أو `"L"`:

```
long unsigned long
```

وأخيراً، إذا كان مسبوقةً بكلٍ من `u` أو `U` و `l` أو `L` فالنوع هو `unsigned long` حصراً.

تُجرى جميع هذه التحويلات لإعطائك النوع "الذي قصدته"، وهذا يعني أن معرفة نوع الثابت ضمن تعبير أمرٌ صعب بعض الشيء إن لم تكن تعرف أي شيء بخصوص عتاد الجهاز. لحسن الحظ هناك بعض المصرفات التي تحدّرك عند ترقية ثابت ما إلى طول آخر ولم يُحدّد النوع باستخدام `U` أو `L` أو غيرها.

تحوي هذه التعليمة على خطأ مُخْبَأً:

```
printf("value of 32768 is %d\n", 32768);
```

سيكون العدد 32768 طويلًا بالنسبة لآلة تعمل بنظام المتمم الثنائي ذي 16 بتًا وفقًا للقواعد المذكورة أعلاه، ولكن تتوقع الدالة printf عددًا صحيحًا فقط على أنه وسيط، ويشير %d إلى ذلك، إلا أن نوع هذا الوسيط خاطئ وينبغي عليك توخي الحذر وتحويل مثل هذه الحالات إلى النوع الصحيح:

```
printf("value of 32768 is %d\n", (int)32768);
```

من الجدير بالذكر أنه لا وجود للثوابت السالبة، فكتابة -23 هو تعبيرٌ مكون من ثابت موجب وعامل. تمتلك ثوابت المحارف نوع int لأسباب تاريخية، وتُكتب عن طريق وضعها بين علامتي تنصيص أحادية على النحو التالي:

```
'a'  
'b'  
'like this'
```

تُكتب المحارف الموسعة الثابتة بالطريقة ذاتها ولكن يسبقها وجود L:

```
L'a'  
L'b'  
L'like this'
```

للأسف، يمكن أن تحتوي المحارف الثابتة على أكثر من محرف واحد، ولكن تنفيذها يعطي نتيجة مرتبطةً بالجهاز الذي تعمل عليه. تُعد المحارف الوحيدة من أفضل الحلول للبرامج المحمولة Portable، إذ تعطي قيمة عدد صحيح ثابت اعتيادي حسب تمثيل الجهاز لهذا الحرف. صادفت في تعريفنا عن المحارف الموسعة هذا المحرف <a> الذي يمثل محرفًا متعدد البايتات مُرمزًا لعمليات الإدخال بالإزاحة والإخراج بالإزاحة)، ويُعد <a> هنا محرفًا ثابتًا، مثل المحرف abcde. سيتسبب هذا النوع من المحارف بالعديد من المشاكل في المستقبل، نأمل أن يحذرك المصرف بشأنهم.

هناك ما يُدعى باسم **سلسلة التهريب Escape sequence**، والتي تهدف إلى تسهيل عملية تمثيل بعض المحارف الخاصة التي سيكون من الصعب استخدامها ضمن محرف ثابت (هل المحرف ' ' هو محرف مسافة space أم مسافة جدولة \tab?). يوضح الجدول 10 **سلاسل التهريب** المُعرفة في المعيار.

الجدول 10: سلاسل التهريب في لغة C

السلسلة	الغرض منها
\a	تحذير صوتي
\b	فراغ للخلف Backspace
\f	فاصل صفحة
\n	سطر جديد
\r	إرجاع المؤشر
\t	مسافة جدول
\v	مسافة جدول شاقولية
\\	شرطة مائلة للخلف
\'	علامة تنصيص فردية
\"	علامة تنصيص مزدوجة
\?	إشارة استفهام

من الممكن أيضًا استخدام سلاسل تهريب عددية لتحديد محرف باستخدام القيمة الداخلية التي تمثله، مثل السلسلة \000 أو \xhhh، إذ أن 000 هي ثلاث خانات ثمانية و hhh هو أي عدد ممثل بالنظام الست عشري. أكثر السلاسل شيوعًا هي \033، التي تُستخدم لتمثل زرّ ESC (الهروب Escape) على لوحة المفاتيح في الحواسيب التي تعمل بترميز ASCII. انتبه إلى أن المحارف الثابتة الممثلة بالقيمة الست عشرية تشمل جميع المحارف الموجودة ضمنها، فعلى سبيل المثال إذا أردت سلسلة نصية تحتوي على القيمة الست عشرية ff متبوعةً بالحرف f، فالطريقة الآمنة لكتابة ذلك هو استخدام خاصية ضمّ السلاسل النصية:

```
"\xff" "f"
```

إذ تمثل السلسلة النصية:

```
"\xffff"
```

محرفًا واحدًا، مكوّنًا من ثلاثة حروف f تمثل قيمة السلسلة الست عشرية.

تتطلب بعض محارف التهريب تفسيرًا إذ أن بعضها غير واضح الوظيفة. للحصول على علامة تنصيص فردية على أنها محرف ثابت نستخدم \\'، وللحصول على علامة استفهام نستخدم \?، للحصول على علامتي استفهام لا يمكنك استخدام ??، لأن السلسلة ?? تعد ثلاثية محارف Trigraph، وبالتالي، عليك استخدام \??. محرف التهريب \" مهم فقط في حالة السلاسل النصية، وستتكلم عن ذلك لاحقًا.

هناك هدفان مختلفان وراء سلاسل التهريب، إذ من المهم طبعًا تمثيل بعض المحارف مثل علامة التنصيص الفردية والشرطة المائلة للخلف بوضوح، وهذا الهدف الأول، أما الهدف الثاني فهو مرتبطٌ بسلاسل التهريب التالية التي تتحكم في حركة أجهزة الطباعة، على النحو التالي:

`\a`

اقرع الجرس في حال وجود شيء ما للطباعة، ولا تتحرك.

`\b`

فراغ للخلف.

`\f`

إذهب إلى أول موضع في "الصفحة التالية"، وقد يعني هذا أشياءً مختلفة لأجهزة الخرج المختلفة.

`\n`

إذهب إلى بداية السطر التالي.

`\r`

عُد إلى بداية السطر الحالي.

`\t`

إذهب إلى مسافة الجدولة الأفقية التالية.

`\v`

اذهب إلى بداية السطر الواقع في موضع مسافة الجدولة الأفقية التالية.

بالنسبة للمحارف `\b` و `\t` و `\v` إن لم يكن هناك موضع موافق فسيكون التصرف غير محدد. يتجنب المعيار ذكر الوجهات الفيزيائية للحركة بالنسبة لأجهزة الخرج، لأنها لا تعمل من الأعلى إلى الأسفل ومن اليسار إلى اليمين بالضرورة كما في بيئات العمل الموجودة في الثقافة الغربية في جميع الحالات.

من المضمون أن لكل محرف مُهَرَّب قيمة عدد صحيح فريدة، وتُخزَّن في النوع `char`.

2.9.2 الأعداد الحقيقية الثابتة

تتبع هذه الأعداد التنسيق الاعتيادي للأعداد الحقيقية:

```
1.0
2.
.1
2.634
.125
2.e5
2.e+5
.125e-3
2.5e5
3.1E-6
```

وإلى آخره. حتى لو كان هناك جزء من الرقم الحقيقي ذو قيمة صفرية يجب إظهاره بهدف تسهيل القراءة:

```
1.0
0.1
```

يدل الجزء الأسّي exponent على مرتبة قوة العدد، مثلًا:

```
3.0e3
```

يكافئ قيمة العدد الصحيح الثابت:

```
3000
```

وكما ترى، يمكن استبدال e بالحرف E أيضًا لنفس الغرض، وهذه الثوابت من نوع double إلا في حال سبق القيمة المحرف f أو F وفي هذه الحالة هي من نوع float؛ وإذا سبقها l أو L فهي في هذه الحالة من نوع long double.

يهدف الوصف الكامل، إليك وصف رسمي يصف طبيعة الأعداد الحقيقية الثابتة:

العدد الحقيقي الثابت يحقق واحدةً من الحالات التالية:

- عدد كسري ثابت متبوعٌ بأُس اختياري.
- سلسلة أرقام متبوعة بأُس.

في الحالتين السابقتين، يمكن أن يُتبع العدد الحقيقي بالأحرف الاختيارية f و l و F و L، بحيث يتحقق:

- الثابت الكسري واحدٌ من الحالات التالية:
 - سلسلة اختيارية من **الخانات** متبوعةً بفاصلة عشرية متبوعةً **بسلسلة من الخانات**.
 - **سلسلة من الخانات** متبوعةً بفاصلة عشرية.
- الأس واحدٌ من الحالات التالية:
 - الحرف **e** أو **E** متبوعٌ برمز **+** أو **-** اختياري متبوعٌ **بسلسلة من الخانات**.
 - **سلسلة من الخانات** هي تركيب اعتباطي من خانةٍ واحدة أو أكثر.

2.10 خاتمة

كان هذا الفصل طويلًا ومقلقًا بعض الشيء.

ليست أبجدية لغة C - على الرغم من أهميتها- مصدرًا يوميًا لتفكير المبرمجين الممارسين، ولهذا ناقشناها، لكننا سنتجاهلها.

يمكننا قول الشيء ذاته على الكلمات المفتاحية والمعرّفات، فالفكرة منها ليست معقدة كثيرًا ومن السهل تذكرها.

نادرًا ما يسبب التصريح عن المتغيرات أي مشكلات، ولكن يجب التشديد على الفرق بين التصريح والتعريف، وإذا كان الفرق غير واضحٍ بالنسبة لك بعد، جرّب قراءة وصف كلٍّ منهما مرةً أخرى.

بعيدًا عن أي سؤال، يكمن التعقيد بلا أي شك في الحالات التي تُجرى فيها الترقية العددية الصحيحة أو التحويلات الحسابية، إذ أن فهم هذه النقطة وإتقانها تمامًا أمرٌ صعبٌ بعض الشيء، ولعل قولنا ذلك يطمئن المبتدئين. لا يوجد أي شيء آخر أكثر أهمية في اللغة يستدعي اهتمامك من هذا الأمر، أو سيساعدك في كتابة برامج صحيحة يمكن الاعتماد عليها. لا نطلب من المبتدئين هنا تذكر جميع التفاصيل، بل نشجعهم بتعلم المزيد عن تفاصيل اللغة لبناء قدر أكبر من الثقة. بعد شهرين أو ثلاثة من تعلم وممارسة الأجزاء الأبسط من اللغة، سيحين الوقت الذي ستحتاج فيه بشدة لفهم التحويلات ما بين الأنواع.

لا يلتفت العديد من المبرمجين المتمرسين إلى أسبقية العوامل المختلفة، ما عدا بعض الحالات، وسيساعدك وجود جدول مطبوع بجانب مكتبك يحتوي مختلف مراحل الأسبقية أو مرجع سهل الفهم.

أثر المعيار بشدة في الأجزاء التي ذكرناها في هذا الفصل، وبالأخص التغييرات الحاصلة على التحويلات والتغيير من "المحافظة على عدمية الإشارة" إلى "المحافظة على القيمة"، وقد تُسبب القواعد الحسابية بعض المفاجأة لمبرمجي سي المتمرسين، إذ يتطلب الأمر منهم إعادة التعلم أيضًا.

2.11 تمارين

2.11.1 تمرين 17.2

أولاً، ضع التعابير التالية ضمن أقواس حسب قوانين الأسبقية والارتباط، ومن ثم استبدل المتغيرات والثوابت بأسماء أنواعها المناسبة، ووضح كيف ينتج كل نوع من التعبير باستبدال التعبير ذو الأسبقية الأعلى بنوع نتيجته.

المتغيرات هي:

```
char c;
int i;
unsigned u;
float f;
```

على سبيل المثال: يصبح التعبير $i = u+1$ بعد وضع الأقواس على النحو التالي:

```
(i = (u + 1));
```

والأنواع هي:

```
(int = (unsigned + int));
```

ومن ثم:

```
(int = (unsigned)); /* usual arithmetic conversions */
```

ومن ثم:

```
(int); /* assignment */
```

.1

```
c = u * f + 2.6L;
```

.2

```
u += --f / u % 3;
```

.3

```
i <<= u * - ++f;
```

.4

```
u = i + 3 + 4 + 3.1;
```

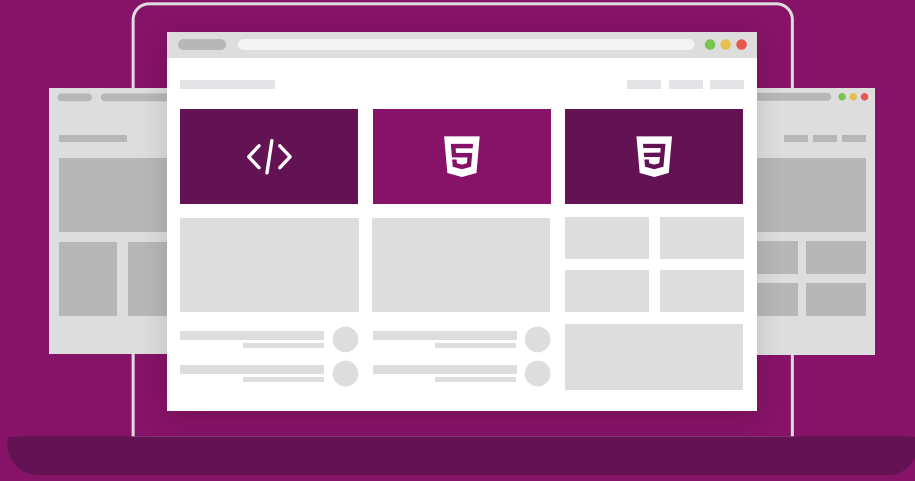
.5

```
u = 3.1 + i + 3 + 4;
```

.6

```
c = (i << - --f) & 0xf;
```

دورة تطوير واجهات المستخدم



ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية فور انتهاءك من الدورة

التحق بالدورة الآن



3. التحكم بالتدفق والتعابير المنطقية

سننظر في هذا الفصل إلى الطرق المختلفة التي تُستخدم بها تعليمات التحكم بتدفق التنفيذ Flow control statements في برنامج سي C، بما فيها بعض التعليمات التي لم نتطرق إليها بعد، والتي تُستخدم في معظم الحالات مع تعبيرات منطقية تحدد الخطوة القادمة، وتُعد تعابير if و while البسيطة المُستخدمة سابقًا مثالًا على هذه **التعابير المنطقية Logical expressions**، ويمكنك استخدام تعابير أكثر تعقيدًا من الموازنات البسيطة، مثل < و <= و == وغيرها، لكن ما قد يفاجئك هو نوع النتيجة.

3.1 التعابير المنطقية والعوامل العلاقية

تجنبنا عمدًا التعقيد باستخدام تعابير منطقية في تعليمات التحكم بالتدفق في جميع الأمثلة المُستخدمة سابقًا، إذ صادفنا مثلًا تعابير تشابه:

```
if(a != 100){...
```

ومن المفترض الآن أنك تعرف دعم لغة C لمفهوم "صواب True" و"خطأ False" لهذه العلاقات، لكنها تدعمها بطريقةٍ مختلفة عن المتوقع.

تُستخدم العوامل العلاقية Relational operators المذكورة في الجدول 11 للموازنة بين مُعاملين بالطريقة المذكورة، وعندما تكون المُعاملات من أنواع عديدة، تطبق التحويلات الحسابية الموافقة إليهما.

الجدول 11: العوامل العلائقية

العامل	العملية
<	أصغر من
<=	أصغر من أو يساوي
>	أكبر من
>=	أكبر من أو يساوي
==	يساوي
!=	لا يساوي

كما أشرنا سابقاً، لا بُدّ من الانتباه لعامل اختبار المساواة == وإمكانية خلطه مع عامل الإسناد = إذ لا تحذرك لغة سي C عند حدوث ذلك كون التعبير صالح، ولكن ستكون نتائج التعبيرين مختلفة، ويستغرق المبتدئون وقتاً طويلاً للاعتياد على استخدام == و =.

لعلّك تطرح على نفسك السؤال "لماذا؟"، أي "لماذا التعبيران صالحان؟" الإجابة بسيطة، إذ تفسر لغة سي مفهوم "صواب True" و"خطأ False" بقيمة "غير صفرية" و"صفرية"، وعلى الرغم من استخدامنا للعوامل العلائقية في التعابير المُستخدمة للتحكم بتعليمات if و do، إلا أننا نستخدم في الحقيقة القيمة العددية الناتجة عن هذا التعبير؛ فإذا كانت قيمة التعبير غير صفرية، فهذا يعني أن النتيجة صحيحة؛ أما إذا تحققت الحالة المعاكسة فالنتيجة خاطئ، وينطبق هذا على جميع التعابير والعوامل العلائقية.

توازن العوامل العلائقية ما بين المُعاملات وتعطي نتيجة صفر للنتيجة الخاطئة (موازنة غير محققة) وواحد للنتيجة الصحيحة، وتكون النتيجة من نوع int، ويوضح المثال التالي كيفية عملها:

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int i;

    i = -10;
    while(i <= 5){
        printf("value of i is %d, ", i);
        printf("i == 0 = %d, ", i==0 );
        printf("i > -5 = %d\n", i > -5);
        i++;
    }
}
```

```

exit(EXIT_SUCCESS);
}

```

يعطينا المثال السابق الخرج القياسي التالي:

```

value of i is -10, i == 0 = 0, i > -5 = 0
value of i is -9, i == 0 = 0, i > -5 = 0
value of i is -8, i == 0 = 0, i > -5 = 0
value of i is -7, i == 0 = 0, i > -5 = 0
value of i is -6, i == 0 = 0, i > -5 = 0
value of i is -5, i == 0 = 0, i > -5 = 0
value of i is -4, i == 0 = 0, i > -5 = 1
value of i is -3, i == 0 = 0, i > -5 = 1
value of i is -2, i == 0 = 0, i > -5 = 1
value of i is -1, i == 0 = 0, i > -5 = 1
value of i is 0, i == 0 = 1, i > -5 = 1
value of i is 1, i == 0 = 0, i > -5 = 1
value of i is 2, i == 0 = 0, i > -5 = 1
value of i is 3, i == 0 = 0, i > -5 = 1
value of i is 4, i == 0 = 0, i > -5 = 1
value of i is 5, i == 0 = 0, i > -5 = 1

```

ما الذي تعتقد حدوثه عند تنفيذ هذه التعليمة المحتمل أنها تكون خاطئة؟

```
if(a = b)...
```

تُسند قيمة b إلى قيمة a ، وكما نعلم تعطي عملية الإسناد نتيجةً من نوع a مهما كانت القيمة المُسندة إلى a ، وبالتالي ستنفَّذ تعليمة `if` الشرطية التعليمة التالية لها إذا كانت القيمة المُسندة لا تساوي الصفر؛ أما إذا كانت القيمة تساوي الصفر، فستجاهل التعليمة التالية. لذا يجب أن تفهم الآن ما الذي يحدث إن أخطأت استخدام عامل الإسناد بدلاً عن عامل المساواة!

سيُفحص التعبير في تعليمات `if` و `while` و `do` فيما إذا كان مساوياً للصفر أم لا، وسننظر إلى كلٍّ من هذه التعليمات عن كثب.

3.2 التحكم بالتدفق

3.2.1 تعليمة إذا if الشرطية

تُكتب تعليمة if بطريقتين:

```
if(expression) statement
```

```
if(expression) statement1
```

```
else statement2
```

تُنفذ **تعليلة إذا الشرطية** في الطريقة الأولى، إذا (و فقط إذا) كان **التعبير expression** لا يساوي إلى الصفر؛ أما إذا كان **التعبير** مساويًا للصفر، ستُهمل **التعليلة statement**. تذكر بأن **التعليلة** قد تكون مركبة، وذلك بوضع عدة تعليمات تابعة لتعليلة if واحدة.

تُشابه الطريقة الثانية سابقتها، باختلاف أن التعليلة statement1 تُفحص قبل statement2 وتنفذ واحدة منهما.

تصنّف الطريقتان بكونهما تعليمة واحدة حسب قواعد صياغة لغة سي، وبالتالي يكون المثال التالي صالح تمامًا.

```
if(expression)
```

```
if(expression) statement
```

إذ تُتبع تعليمة (expression) if بتعليلة if متكاملة أخرى، وبما أنها تعليلة صالحة، يمكننا قراءة التعليلة الشرطية الأولى على النحو التالي:

```
if(expression) statement
```

وبذلك فهي مكتوبة بصورة صحيحة، كما يمكن إضافة مزيدٍ من الوسائط arguments كما تريد، ولكنها عادةً برمجة سيئة، ومن الأفضل أن تحاول جعل التعليلة مختصرة قدر الإمكان حتى لو لم يكن الأمر ضروريًا في حالة استخدامها، لأن ذلك سيسهل إضافة مزيدٍ من التعابير إن احتجت إلى الأمر لاحقًا ويحسن من سهولة القراءة.

ينطبق ما سبق على طريقة كتابة تعليمة else، ويمكنك كتابتها على النحو التالي:

```
if(expression)
```

```
if(expression)
```

```
statement
```

```
else
    statement
```

كما ذكرنا في [الفصل الأول](#)، ما تزال طريقة الكتابة هذه باستخدام المسافات فقط غامضة وغير واضحة، فأي تعليمات `else` تتبع لتعليمات `if`؟ إذا اتبعنا المسافات في مثالنا فسيوحي هذا لنا بأن التعليمة `if` الثانية متبوعةً بالتعليمة `statement` مما يجعل كل منها تعليمةً مستقلة، و `else` إذاً تنتمي للتعليمة الشرطية الأولى.

هذه **ليست** الطريقة التي تنظر بها لغة C إلى المثال، إذ تنص القاعدة على أن `else` تتبع التعليمة الشرطية `if` التي تسبقها إن لم تحتوي هذه التعليمة على `else`؛ وبالتالي تتبع `else` إلى التعليمة الشرطية الثانية في المثال الذي ناقشناه.

لتجنب أي مشكلة بخصوص `if` و `else` كما لاحظنا في المثال السابق، يمكن إهمال التعليمة الشرطية باستخدام تعليمة مركبة. بالعودة إلى مثالنا السابق في [الفصل الأول](#):

```
if(expression){
    if(expression)
        statement
}else
    statement
```

والذي يصبح باستخدام أقواس التعليمات المركبة على النحو التالي:

```
if(expression){
    if(expression){
        statement
    }
}else{
    statement
}
```

إن لم يرق لك مكان الأقواس في المثال السابق، فتغيير مكانها هو تفضيل شخصي ويمكنك تعديله لما تراه مناسباً، لكن فقط كن متسقاً حيال ذلك، كما تجدر الإشارة إلى أن هناك كثيراً من التعصّب بين المبرمجين بخصوص المكان الصحيح.

3.2.2 تعليمة do و while التكرارية

تعليمة while بسيطة:

```
while(expression)
    statement
```

تُنفذ **التعليمة** في حال كانت قيمة **التعبير** لا تساوي الصفر، وتُفحص قيمة **التعبير** مجددًا بعد كل تكرار وتُعاد التعليمة إذا لم تكن قيمة التعبير مساويةً لصفر. هل هناك أي شيء أكثر وضوحًا من ذلك؟ النقطة الوحيدة الواجب الانتباه بشأنها هي إمكانية عدم تنفيذ **التعليمة** على الإطلاق، أو تنفيذها بدون توقُّف إذا لم تتضمن **التعليمة** أي شيء يؤثر على قيمة **التعبير** المبدئية.

نحتاج في بعض الأحيان تنفيذ التعليمة مرةً واحدةً على الأقل، ونستطيع في هذه الحالة استخدام الطريقة المعروفة بتعليمة do على النحو التالي:

```
do
    statement
while(expression);
```

ينبغي الانتباه على الفاصلة المنقوطة، فهي غير اختيارية. تضمن بهذه الطريقة تنفيذ التعليمة مرةً واحدةً على الأقل قبل تقييم التعبير. كان استخدام الكلمة المفتاحية `while` للاستخدامين السابقين خيارًا غير موفِّقًا، لكن لا يبدو أن هناك الكثير من الأخطاء الناتجة عن ذلك.

فكّر طويلًا قبل استخدام التعليمة `do`، فعلى الرغم من أهميتها في بعض الحالات إلا أن استخدامها المفرط ينتج شيفرةً برمجيةً سيئة التصميم. هذا الأمر غير محقق في معظم الحالات، لكن يجب أن تتوقف وتساءل نفسك عن أهمية استخدام تعليمة `do` في كل حالة قبل استخدامها للتأكد من أن هذا هو الاستخدام الصحيح لها، إذ يدل استخدامها على تفكير غير مخطَّط له وتوظيف وسائل تُستخدم في لغات أخرى، أو تصميمًا سيئًا ببساطة. عندما **تقتنع** بأهمية استخدامها دونًا عن أي وسيلة أخرى، فاستخدمها بحرص.

١. اختصار عملية الإسناد والتحقق في تعبير واحد

يُعد استخدام نتيجة عملية الإسناد للتحكم بعمل حلقات `while` و `do` التكرارية حيلةً مستخدمةً كثيرًا في برامج سي C، وهي شائعة الاستخدام كثيرًا إذ سترغب بتعلُّمها إن صادفتها. تقع هذه الحيلة تحت تصنيف لغة سي "الاصطلاحية" واستخدامها طبيعيٌ وتلقائي لكل من يستخدم اللغة. إليك المثال الأكثر شيوعًا لاستخدامها:

```
#include <stdio.h>
#include <stdlib.h>
```

```

main(){
    int input_c;

    /* The Classic Bit */
    while( (input_c = getchar()) != EOF){
        printf("%c was read\n", input_c);
    }
    exit(EXIT_SUCCESS);
}

```

[مثال 2.3]

تكمّن الحيلة في تعبير إسناد `input_c`، إذ يُستخدم هذا التعبير في إسناد القيمة إلى المتغير وموازنته مع EOF (نهاية الملف End of File) والتحكم بعمل الحلقة التكرارية في آنٍ واحد. يُعد تضمين عملية الإسناد على هذا النحو تحسینًا عمليًا سهلًا، وعلى الرغم من كونها تختصر سطرًا واحدًا فقط إلا أن فائدتها في تسهيل عملية القراءة (بعد الاعتياد على استخدامها) كبيرة. لا بُد أيضًا من تعلّم مكان استخدام الأقواس أيضًا، فهي مهمة في تحديد الأسبقية.

لاحظ أن `input_c` من النوع `int`، وذلك لتمكين الدالة `getchar` من إعادة أي قيمة ممكنة لأي `char` إضافةً لقيمة `EOF`، ولهذا احتجنا لنوع أطول من `char`.

تُعدّ التعليمتان `while` و `do` وفق قواعد صياغة لغة سي تعليمةً واحدةً مثل تعليمة `if`، ويمكن استخدامهما في أي مكان يُمكن استخدام تعليمةٍ واحدةٍ فيه. ينبغي عليك استخدام تعليمة مركّبة إذا أردت التحكم بعدّة تعليمات، كما وُصّحت أمثلة فقرة تعليمة `if`.

3.2.3 تعليمة for التكرارية

يعدّ استخدام الحلقات التكرارية والمتغيرات عدّاداتٍ لها ميزةً شائعةً في لغات البرمجة، إذ لا ينبغي للعداد أن يكون ذا قيم متعاقبة حصراً، والاستخدام الشائع له هو تهيئته خارج الحلقة وتفقد قيمته عند كل تكرار للتأكد من انتهاء الحلقة التكرارية وتحديث قيمته كل دورة. هناك ثلاث خصائص مهمة مرتبطة بتحكم الحلقة، هي: التهيئة `initialize` والتحقق `check` والتحديث `update`، كما هو موضح في المثال التالي:

```

#include <stdio.h>
#include <stdlib.h>
main(){
    int i;

```

```

/* initialise التهيئة*/
i = 0;
/* check التحقق*/
while(i <= 10){
    printf("%d\n", i);
    /* update التحديث*/
    i++;
}
exit(EXIT_SUCCESS);
}

```

[مثال 3.3]

الأجزاء المرتبطة بعملية التهيئة والتحقق متقاربين كما تلاحظ، وموقعهما واضح بسبب استخدام الكلمة المفتاحية `while`، أما الجزء الأكثر صعوبةً لتحديده هو التحديث وبالأخص إذا استُخدمت قيمة المتغير الذي يتحكم بالحلقة داخلها؛ وفي هذه الحالة -الأكثر شيوعًا- يجب أن يكون التحديث في نهاية الحلقة، بعيدًا عن التهيئة والتحقق. يؤثر ذلك في سهولة القراءة بصورة سلبية، إذ من الصعب فهم وظيفة الحلقة إلا بعد قراءة محتواها كاملًا بحرص. إدًا، نحن بحاجة إلى طريقة لجمع أجزاء التهيئة والتحقق والتحديث في مكان واحد لنستطيع قراءتها بسرعة وسهولة، وهذا الغرض من تصميم تعليمة `for`، إذ تُكتب على النحو التالي:

```
for (initialize; check; update) statement
```

جزء **التهيئة** هو تعبير إسناد معظم الحالات، ويُستخدم لتهيئة المتحول الذي يتحكم بالحلقة. يأتي تعبير **التحقق** بعد التهيئة، إذ تُنفَّذ التعليمة داخل الحلقة إذا كان هذا التعبير ذو قيمة غير صفرية، ويُتبع ذلك بتعبير **التحديث** الذي يزيد من قيمة المتغير المُتحكم (في معظم الحالات)، وتُعاد هذه العملية عند كل دورة، وتنتهي الحلقة التكرارية في حال كانت قيمة تعبير التحقق مساويةً للصفر.

هناك أمران مهمان يجب معرفتهما بخصوص الوصف السابق: الأول وهو أن كل جزء من أجزاء تعليمة حلقة `for` التكرارية الثلاث بين القوسين هو تعبير، الثاني وهو أن الشرح وَصَفَ بحرص وظيفة الحلقة التكرارية `for` الأساسية دون ذكر أي استخدامات بديلة. يمكنك استخدام التعابير على النحو الذي تراه مناسبًا، لكن ذلك سيكون على حساب قابلية القراءة إذا لم تُستخدم للغرض المقصود منها.

إليك البرنامج التالي الذي ينجز المهمة ذاتها بطريقتين، الطريقة الأولى باستخدام حلقة `while` والطريقة الثانية باستخدام حلقة `for`، واستُخدم عامل الزيادة بالطريقة المعتادة التي يُستخدم بها في هذه الحلقات.

```

#include <stdio.h>
#include <stdlib.h>

```

```

main(){
    int i;

    i = 0;
    while(i <= 10){
        printf("%d\n", i);
        i++;
    }

    /* the same done using ``for'' */
    for(i = 0; i <= 10; i++){
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}

```

[مثال 4.3]

ليس هناك أي اختلاف بين الطريقتين، عدا أن استخدام الحلقة `for` مناسب وقابل للتعديل بصورة أفضل من تعليمة `while`. حاول استخدام التعليمة `for` في معظم الحالات المناسبة، أي عندما تحتاج إلى حلقة يتحكم فيها عدّاد ما مثلاً؛ بينما يُعد استخدام حلقة `while` أنسب عندما يكون العدد الذي يتحكم في عدد الدورات جزءاً من عمل البرنامج. يتطلب الأمر تحكيماً من كاتب البرنامج وفهماً لشكل وتنسيق البرنامج المكتوب بطريقة جيّدة، إذ لا يوجد أي دليل يقول أن زيادة هذه الخصائص ينعكس سلبيًا على شركات كتابة البرمجيات، فتمزّن على هذه الأمور قدر الإمكان.

يمكن حذف وإهمال أيّ من أجزاء التهيئة والتحقق والتحديث في تعليمة `for` التكرارية، إلا أن الفواصل المنقوطة يجب أن تبقى، ويمكن اتباع هذا الأسلوب عندما يكون العداد مهيباً مسبقاً أو يُحدّث بداخل متن الحلقة. إذا حُذف جزء تعبير التحقق، فهذا يعطينا نتيجةً افتراضيةً تتمثل بالقيمة "صحيح `true`" وهذا سيجعلها حلقةً لا نهائية. الطريقة الشائعة في كتابة حلقات تكرارية لا نهائية، هي:

```
for(;;)
```

أو

```
while(1)
```

يمكنك ملاحظة هذه التعليمات في بعض البرامج المكتوبة بلغة سي C.

3.2.4 أهمية تعليمات التحكم بالتدفق

يمكننا كتابة برامج بدرجات تعقيد متفاوتة باستخدام تعليمات التحكم بالتدفق، إذ تعد هذه التعليمات من صلب لغة C وستوضح قراءتك لبعض البرامج الأساسية في لغة C أهمية هذه التعليمات بما يخص تقديم الأدوات الأساسية وهيكل البرامج. ستعطي التعليمات المتبقية التي سنذكرها للمبرمج تحكماً أكبر بهذه الحلقات أو ربما ستساعده في بعض الحالات الاستثنائية. لا تحتاج تعليمة `switch` إلى أي شرح بخصوص أهمية استخدامها؛ فمن الممكن استبدالها بالعديد من تعليمات `if`، ولكنها تسهّل قراءة البرنامج كثيراً. يمكنك النظر إلى `break` و `continue` و `goto` على أنها بهارات لصلصة حساسة المقادير، إذ يمكن لاستخدامها الحريص أن يجعل من هذه الصلصة أكلةً لذيذة، وبالمقابل سيجعل الاستخدام المفرط لها طعم الصلصة مشتتاً وضائعاً.

3.2.5 تعليمة `switch`

يمكنك الاستغناء عن هذه التعليمة، فهي ليست جزءاً أساسياً من لغة سي C، لكن استخدامها سيجعل اللغة أقلّ تعبيراً ومتعةً للاستخدام؛ إذ تُستخدم هذه التعليمة لاختيار عددٍ من الإجراءات المختلفة حسب قيمة تعبيرٍ ما، وتجعل من تعليمة `break` تعليمةً مستخدمةً كثيراً ضمنها، إذ تُكتب على النحو التالي:

```
switch (expression){
  case const1:    statements
  case const2:    statements
  default:        statements
}
```

تُقدّر قيمة **التعبير expression** وتوازن قيمته مع جميع تعابير `const1` و `const2` إلى آخره، والتي تكون قيمها مختلفة (من نوع **تعابير أعداد صحيحة ثابتة** حصراً، راجع **الفصل السادس** وما يليه لمزيدٍ من الشرح)؛ فإذا تساوت القيمة مع قيمة **التعبير**، تُنفَّذ التعليمة التي تتبع الكلمة المفتاحية `case`؛ وتُنفَّذ الحالة الافتراضية التابعة للكلمة المفتاحية `default` -إن وجدت- في حال عدم وجود أي قيمة مكافئة؛ وإن لم تتواجد هذه الكلمة المفتاحية (ولم تتواجد أي قيمة مكافئة)، فلن تُنفَّذ التعليمة `switch` أي شيء وسيتابع البرنامج تنفيذ التعليمة التالية.

من الميزات المثيرة للاهتمام هي أن الحالات **ليست** استثنائية، كما هو موضح في المثال التالي:

```
#include <stdio.h>
#include <stdlib.h>

main(){
  int i;
  for(i = 0; i <= 10; i++){
```

```

switch(i){
    case 1:
    case 2:
        printf("1 or 2\n");
    case 7:
        printf("7\n");
    default:
        printf("default\n");
}
}
exit(EXIT_SUCCESS);
}

```

[مثال 5.3]

تتكرر الحلقة بحسب قيمة i ، إذ تأخذ قيمًا من 0 إلى 10، تتسبب قيمة 1 أو 2 بطباعة الرسالة "1 or 2" عن طريق تنفيذ تعليمة `printf` الأولى. وما قد يفاجئك هنا هو طباعة الرسائل التي تليها أيضًا، لأن تعليمة `switch` تختار نقطة دخول واحدة إلى متن التعليمة، فبعد البدء في نقطة معينة، تُنفذ جميع التعليمات التي تليها. تُستخدم عناوين `case` و `default` ببساطة لتحديد أي من التعليمات التي سيقع عليها الاختيار. عندما تكون قيمة i مساويةً للقيمة 7، ستُطبع الرسالتان الأخيرتان فقط، وأي قيمة لا تساوي إلى 1 أو 2 أو 7 ستتسبب بطباعة الرسالة الأخيرة فقط.

يمكن أن تُكتب العناوين `labels` بأي ترتيب كان، لكن لا يجب أن تتكرر أي قيمة ويمكنك استخدام عنوان `default` واحد فقط أو الاستغناء عنه، وليس من الضروري أن يكون العنوان الأخير، كما يمكن وضع تعليمة واحدة لعدة عناوين أو عدة تعليمات لعنوان واحد.

يمكن أن يكون التعبير الذي يتحكم بتعليمة `switch` من أي نوع عدد صحيح. كانت لغة سي C القديمة تقبل نوع `int` فقط، واقتطعت المصرفات قسرًا الأنواع الأطول مما تسبب ببعض الأخطاء الغامضة بعض الأحيان.

1. أكبر قيود تعليمة Switch

تكمّن المشكلة الأكبر بخصوص تعليمة `switch` في عدم إمكانية تحديد أجزاء معينة من التعليمات بصورة استثنائية، إذ تُنفذ جميع التعليمات الموجودة بداخل تعليمة `switch` التي تلي التعليمة المُنفذة على نحو متعاقب، والحل هنا هو استخدام تعليمة `break`. ألقِ نظرةً على المثال السابق المُعدّل بحيث لا تُطبع الرسائل تبعًا بعد طباعة حالة ما، إذ تتسبب التعليمة `break` بمغادرة تنفيذ تعليمة `switch` مباشرةً وتمنع تنفيذ أي تعليمات أخرى ضمنها.

```

#include <stdio.h>
#include <stdlib.h>
main(){
    int i;
    for(i = 0; i <= 10; i++){
        switch(i){
            case 1:
            case 2:
                printf("1 or 2\n");
                break;
            case 7:
                printf("7\n");
                break;
            default:
                printf("default\n");
        }
    }
    exit(EXIT_SUCCESS);
}

```

[مثال 6.3]

يوجد مزيدٌ من الاستخدامات لتعليمة `break` سنتكلم عنها في فقرتها الخاصة.

ب. تعبير العدد الصحيح الثابت

سنتكلم لاحقاً عن التعابير الثابتة، ولكن من المهم التطرق إلى طبيعة تعبير العدد الصحيح الثابت كونه التعبير الذي يجب أن تُلحقه بعنوان `case` في تعليمة `switch`. عمومًا، لا يحتوي تعبير العدد الصحيح الثابت أي عوامل تغيّر من قيمته، مثل عامل الزيادة أو الإسناد، أو استدعاءٍ للدوال أو عوامل الفاصلة. ويجب أن تكون جميع المُعاملات في التعبير ثوابت صحيحة وثوابت محرفية وثوابت تعداد `constants` و `enumeration` وتعابير `sizeof` وثوابت الفاصلة العائمة وهي المُعاملات المباشرة لمحوّلات النوع `casts`، كما يجب أن تكون أنواع العدد الصحيح هي الأنواع الناتجة عن أي مُعامل محوّل للنوع، وجميع هذه القيم يمكن أن تتوقعها بكونها تحت تصنيف تعبير العدد الصحيح الثابت.

3.2.6 تعليمة `break`

هذه التعليمة بسيطة، ويمكن فهمها في سياق استخدامها ضمن تعليمة `switch`، أو `do`، أو `while`، أو `for`، إذ يقفز تدفق البرنامج عند استخدامها إلى التعليمة التالية خارج متن التعليمة الحالية التي تتضمن تعليمة

`break`؛ وتُستخدم كثيرًا في تعليمات `switch`، إذ إنها ضرورية بطريقةٍ أو بأخرى للحصول على التحكم الذي يحتاجه معظم الناس.

استخدام تعليمة `break` بداخل الحلقات التكرارية استخدامٌ غير محبّب بحسب الحالة، إذ إنه مبرّر عند حدوث ظروف استثنائية بداخل الحلقة مما يتطلب مغادرتها. من الجيد أن نستطيع مغادرة عدة حلقات دفعةً واحدة باستخدام تعليمة `break` واحدة فقط، لكن هذا غير محقّق. ألق نظرةً على المثال التالي:

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int i;

    for(i = 0; i < 10000; i++){
        if(getchar() == 's')
            break;
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

[مثال 7.3]

يقرأ البرنامج السابق محرفًا وحيدًا من الدخل قبل طباعة قيمة `i` وفق سلسلةٍ من الأعداد، وتتسبب تعليمة `break` بالخروج من الحلقة في حال إدخال المحرف `s`.

يُعد استخدام `break` خيارًا خاطئًا إذا أردت الخروج من عدة مستويات من الحلقات، إذ أن استخدام `goto` هو الطريقة السهلة الوحيدة لكن سنتركها إلى النهاية بما أن ذكرها يتطلب وجود ذكر أشياء أخرى قبلها.

3.2.7 تعليمة `continue`

يوجد لهذه التعليمة عددٌ محدودٌ من حالات الاستخدام، وقواعد استخدامها مطابقةٌ لقواعد استخدام `break` عدا أنها لا تُطبّق في تعليمات `switch`. يبدأ التكرار التالي لأصغر تعليمة (أقلها مستوى) سواء كانت `do`، أو `while`، أو `for` فور تنفيذ تعليمة `continue`، ويقتصر استخدامها في بداية الحلقات حيث يجب اتخاذ قرار بشأن تنفيذ بقية متن الحلقة أم لا. تَصْمَن تعليمة `continue` في المثال التالي عدم تنفيذ القسمة على صفر، والتي تتسبب بسلوك غير محدد.

```
#include <stdio.h>
#include <stdlib.h>
```



```

main(){
    int i;

    for(i = -10; i < 10; i++){
        if(i == 0)
            continue;
        printf("%f\n", 15.0/i);
        /*
         * Lots of other statements .....
         */
    }
    exit(EXIT_SUCCESS);
}

```

[مثال 8.3]

قد تنظر إلى التعليمة `continue` بكونها غير ضرورية، وأنه ينبغي أن يكون تنفيذ متن الحلقة شرطياً بدلاً من ذلك، لكنك لن تجد الكثير من المؤيدين لرأيك، إذ يفضل معظم مبرمجي لغة سي C استخدام `continue` بدلاً من استخدام مستوي إضافي من المسافات لتضمين جزء معين من الحلقة وبالأخص إن كان كبيراً. يمكنك طبعاً استخدام `continue` في أجزاء أخرى من الحلقة، بهدف تبسيط منطق الشيفرة وتحسين قابلية قراءتها، ولكن لا بُد من استخدامها باعتدال.

أبق في بالك أن التعليمة `continue` ليس لها أي معنى داخل تعليمة `switch` على عكس `break`، إذ أن استخدام `continue` بداخل `switch` ذو قيمة فقط في حالة وجود حلقة تكرارية تحتوي `switch`، وفي هذه الحالة يبدأ التكرار التالي من الحلقة عند تنفيذ `continue`.

هناك فرق مهم بين الحلقات التكرارية المكتوبة باستخدام `while` و `for`؛ ففي حلقات `while` وعند استخدام `continue` يقفز التنفيذ إلى فحص قيمة التعبير التي تتحكم بالحلقة؛ بينما تُنفذ تعليمة التحديث وتعبير التحكم في حالة `for`.

3.2.8 تعليمة goto والعناوين labels

يعرف الجميع أن استخدام تعليمة `goto` هو "تصرف سيء"، إذ أنها تجعل برنامجك صعب المتابعة والقراءة، وتشئت هيكله وتدفعه إذا استخدمت من دون عناية. كتبت مجموعة دكسترا Dijkstra ورقة شهيرة في 1968 باسم "تعليمة Goto مُضرة Goto Statement Considered Harmful" التي يذكرها الجميع ولكن لم يقرأها أي أحد.

الشيء الأكثر إزعاجًا هو عندما يكون استخدامها في بعض الحالات ضروريًا للغاية، إذ تُستخدم في لغة سي C للخروج من عدة حلقات تكرارية متداخلة أو للانتقال إلى مخرج للتعامل مع الأخطاء في نهاية الدالة، وستحتاج لاستخدام **عنوان label** عندما تقرر استخدام goto، ويوضح المثال التالي استخدامهما:

```
goto L1;
/* whatever you like here */
L1: /* anything else */
```

العنوان هو معرفٌ متبوعٌ بنقطتين، ويوجد للعناوين "مجال أسماء namespace" خاص بهم لتجنب الخلط بينهم وبين أسماء المتغيرات والدوال. مجال الأسماء هذا متواجدٌ فقط بداخل الدالة التي تحتويه، لذلك يمكنك إعادة استخدام أسماء العناوين في دوال مختلفة، كما يُمكن للعنوان أن يُستخدم قبل التصريح عنه أيضًا عن طريق ذكره ضمن تعليمة goto ببساطة.

يجب أن تكون العناوين جزءًا من تعليمة كاملة، حتى لو كانت فارغة، وعادة ما يكون هذا مهمًا فقط عندما تحاول وضع عنوان في نهاية التعليمة المركبة على النحو التالي:

```
label_at_end: ; /* empty statement */
}
```

تعمل goto بطريقة واضحة، إذ تنتقل إلى التعليمة المعنونة، ولأن اسم العنوان مرئي فقط من داخل الدالة التي تحتويه فمن غير الممكن الانتقال من دالةٍ إلى دالةٍ أخرى.

من الصعب إعطاء قوانين صارمة حول استخدام تعليمة goto ولكن على نحوٍ مشابه لتعليمات do و continue و break (عدا وجودها في تعليمة switch)، فإن الاستخدام المفرط لها غير محبذ. فكّر طويلًا قبل استخدامها وليكن استخدامها بالنسبة لهيكلية البرنامج مقنعًا، إذ استخدامك لتعليمة goto كل 3 أو 5 دوال يدل على مشكلة ويجب أن تجد طريقةً مختلفةً لكتابة برنامجك.

3.2.9 خلاصة

الآن وبعد استعراضنا لتعليمات التحكم بتدفق البرنامج المختلفة ورأينا أمثلةً عن استخدامها، يجب أن تستخدم بعضها في أي فرصة تُتاح لك، بينما يُستخدم بعضها الآخر لأغراض خاصة ولا يجب الإفراط في استخدامها. يجعل الانتباه إلى استخدام التعليمات برامجك المكتوبة بلغة سي C أنيقة، إذ تعطيك تعليمات التحكم بالتدفق المخصصة الفرصة لإضافة خصائص غير موجودة في بعض اللغات الأخرى.

وبذلك يبقى التكلّم عن العوامل المنطقية كل ما تبقى للانتهاء من جانب التحكم بتدفق البرنامج في لغة سي C.

3.3 عوامل منطقية أخرى

وضحنا سابقاً كيف أن لغة سي C لا تميّز بين القيم "المنطقية" والقيم الأخرى، إذ تعطي المُعاملات العلائقية relational operators نتيجة متمثلةً بالقيمة 0 للخطأ، أو 1 للصواب. وتُقيّم قيمة تعبير منطقي لتحديد ما إذا كانت تعليمة تحكم بتدفق البرنامج ستنفَّذ أم لا، إذ تعني القيمة 0 "لا تنفَّذ" وأي قيمة أخرى تعني "نفَّذ". تُعد جميع الأمثلة التالية تعابير منطقية صالحة:

```
while (a<b)...
while (a)....
if ( (c=getchar()) != EOF )...
```

لن يتفاجأ أي مبرمج لغة سي متمرس بأي تعبير من التعابير السابقة، إذ يمثّل التعبير الثاني (a) while اختصاراً شائعاً للتعبير (a != 0) while (على الأغلب استنتجت معناه ضمناً).

نحتاج الآن إلى طريقة تمكّننا من كتابة تعابير أكثر تعقيداً باستخدام هذه القيم المنطقية "خطأ وصواب"، إذ استخدمنا حتى الآن الطريقة التالية لكتابة التعبير الذي يعطينا (if(a<b AND c<d)، إلا أن هناك طريقة أخرى لكتابة التعليمة.

```
if (a < b){
    if (c < d)...
}
```

هناك ثلاثة عوامل مشاركة في هذا النوع من العمليات، هي: العامل المنطقي AND أو "&&" والعامل المنطقي OR أو "||" والعامل NOT أو "!", إذ يُعد العامل الأخير عاملاً أحاديًا، أما العاملان الآخران فهما ثنائيان. تستخدم هذه العوامل تعابيرًا مثل مُعاملات وتعطي نتيجةً مساويةً إلى "1" أو "0"؛ إذ يعطي العامل "&&" القيمة "1" فقط في حال كانت قيمة المُعاملان غير صفرية؛ ويعطي "||" القيمة "0" فقط في حال كانت قيمة المُعاملان صفرًا؛ بينما يعطي "!" القيمة صفر إذا كانت قيمة المُعامل غير صفرية وبالعكس، فالأمر بسيطٌ للغاية، وتكون نتيجة العوامل الثلاث من النوع "int".

لا تخلط عوامل العمليات الثنائية bitwise operators "|" و"&" مع عواملها المنطقية المشابهة، فهي مختلفةٌ عن بعضها البعض؛ إذ أن للعوامل المنطقية ميزةً لا نجدها في العوامل الأخرى، ألا وهي تأثيرها على عملية تقييم التعبير، إذ تُقيّم من اليسار إلى اليمين بعد أخذ الأسبقية في الحسبان، ويتوقف أي تعبير منطقي عن عملية التقييم عند التوصل إلى قيمة التعبير النهائية. على سبيل المثال، تتوقف سلسلة من عوامل "||" عن التقييم حالما تجد مُعاملًا ذا قيمةً غير صفرية. يوضّح المثالين التاليين تعبيرين منطقيين يمنعان القسمة على صفر.

```

if (a!=0 && b/a > 5)...
/* alternative */
if (a && b/a > 5)

```

سُتَقِيَم قيمة b/a في كلا الحالتين فقط في حالة كون قيمة a غير صفرية؛ وفي حال كانت a مساوية الصفر، سَتَقِيَم قيمة التعبير فورًا، وبذلك ستتوقف عملية تقييم التعبير وفق قوانين لغة سي C للعوامل المنطقية.

عامل النفي الأحادي NOT بسيط، لكن استخدامه غير شائع جدًا نظرًا لإمكانية كتابة معظم التعبيرات دون استخدامه، كما توضح الأمثلة التالية:

```

if (!a)...
/* alternative */
if (a==0)...

if(!(a>b))
/* alternative */
if(a <= b)

if (!(a>b && c<d))...
/* alternative */
if (a<=b || c>=d)...

```

توضح الأمثلة السابقة مع بدائلها طريقةً لتجنُّب أو على الأقل الاستغناء عن استخدام العامل !. في الحقيقة، يُستخدم هذا العامل بهدف تسهيل قراءة التعبير، فإذا كانت المشكلة التي تحلها تستخدم علاقة منطقية مثل العلاقة " $(b*b-4*a*c) > 0$ " الموجودة في حل المعادلات التربيعية، فمن الأفضل كتابتها بالطريقة:

```
( !((b*b-4*a*c) > 0))
```

بدلاً من كتابتها بالطريقة:

```
if( (b*b-4*a*c) <= 0)
```

لكن التعبيران يؤديان الغرض ذاته، فاختر الطريقة التي تناسبك.

تعمل معظم التعابير التي تستخدم العوامل المنطقية وفق قوانين الأسبقية الاعتيادية، لكن الأمر لا يخلو من بعض المفاجآت، فإذا أخذت نظرةً أخرى على جدول الأسبقية، فستجد أن هناك بعض العوامل في مستوى أسبقية أقل موازنةً بالعوامل المنطقية، ويسبب ذلك خطأ شائعًا:

```
if(a&b == c){...
```

إذ تُوازَن b بالمساواة مع c أولاً في هذه الحالة، ومن ثم تُضاف القيمة إلى a سواءً كانت 0 أو 1، مما يتسبب بسلوك غير متوقع للبرنامج بسبب هذا الخطأ.

3.4 عوامل غريبة

تبقى عاملان لم نتكلم عنهما بعد، ويتميزان بشكلهما الغريب، إذ لا تُعد هذه العوامل "أساسية"، لكنها تُستخدم من حينٍ إلى آخر، فلا تتجاهلهما كلياً، وستكون هذه الفقرة الوحيدة التي سنتكلم فيها عنهما، إذ سيتضمن الشرح سلوكهما عند مزجهما مع أنواع المؤشر، مما يوحي أنهما معقدَيْن أكثر من اللازم.

3.4.1 عامل الشرط?:

كما هو الحال في عزف آلة الأكورديون، سيكون من الأسهل النظر إلى كيفية عمل هذا العامل بدلاً من وصفه.

```
expression1?expression2:expression3
```

إذا كان التعبير $expression1$ صحيحاً، فهذا يعني أن قيمة التعبير بالكامل هي من قيمة التعبير $expression2$ ، وإلا فهي قيمة التعبير $expression3$ ، إذ ستُقيَّم قيمة واحدٍ من التعبيرين حسب قيمة التعبير $expression1$.

تُعد أنواع البيانات المختلفة المسموح استخدامها في التعبير $expression2$ و $expression3$ والأنواع الناتجة عن التعبير بالكامل معقدة، والسبب في هذا التعقيد هو بعض الأنواع والمفاهيم التي لم نشرحها بعد. سنشرح هذه النقاط في الأسفل، لكن كن صبوراً بخصوص بعض المفاهيم التي سنشرحها لاحقاً.

أبسط حالة ممكنة هي حالة تعبير بأنواع حسابية (أعداد صحيحة أو حقيقية)، إذ تُطبَّق في هذه الحالة التحويلات الحسابية لإيجاد نوع مشترك لكلا التعبيرين، وهو نوع النتيجة. لنأخذ المثال التالي:

```
a>b?1:3.5
```

يحتوي المثال الثابت 1 من النوع int والثابت 3.5 من النوع $double$ ، وبتطبيق قوانين التحويل الحسابية نحصل على نتيجة النوع $double$.

هناك بعض الحالات المسموحة أيضاً، هي:

- إذا كان المُعاملان من نوع $structure$ أو اتحاد $union$ متوافق، فالنتيجة هي من هذا النوع.
- إذا كان المُعاملان من نوع $void$ ، فالنتيجة هي من هذا النوع.

ويمكن مزج عدة أنواع مؤشرات على النحو التالي:

- يمكن للمُعاملين أن يكونا من أنواع مؤشرات متوافقة (من المحتمل أن تكون **مؤهلة (qualified)**).
- يمكن لمُعامل أن يكون مؤشرًا والمُعامل الآخر **مؤشر ثابت فارغ (Null pointer constant)**.
- يمكن لمُعامل أن يكون **مؤشرًا لكائن (pointer to an object)** أو **نوع غير مكتمل (incomplete type)** والمُعامل الآخر مؤشر إلى نوع "void" (من المحتمل أن يكون **مُوهل**).

لمعرفة النوع الناتج عن التعبير عند استخدام المؤشرات تتبع الخطوتين:

1. إذا كان أيٌّ من المُعاملين مؤشرًا إلى نوع قيمة مؤهلة، فستكون النتيجة مؤشرًا إلى نوع مؤهل من جميع المؤهلات في كلا المُعاملين.

2. إذا كان أحد المُعاملات مؤشر ثابت فارغ، فستكون النتيجة هي نوع المُعامل الآخر؛ فإذا كان أحد المُعاملات يُشير إلى الفراغ، سيُحوّل المُعامل الآخر إلى مؤشر إلى "void" وسيكون هذا نوع النتيجة؛ أما إذا كان كلا المُعاملات مؤشرات من أنواع متوافقة (بتجاهل أي مؤهلات) فالنوع الناتج هو من **نوع مركب (composite type)**.

سنناقش كلاً من المؤهلات والأنواع المركبة والمتوافقة في الفصول القادمة.

نلاحظ استخدام هذا العامل في المثال البسيط أدناه، الذي يختار السلسلة النصية لطباعتها باستخدام الدالة `printf`:

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int i;

    for(i=0; i <= 10; i++){
        printf((i&1) ? "odd\n" : "even\n");
    }
    exit(EXIT_SUCCESS);
}
```

[مثال 10.3]

يُعد هذا العامل جيدًا عندما تحتاجه، مع أن بعض الناس يشعرون بالغرابة عندما يرونه للمرة الأولى إلا أنهم سرعان ما يبدؤون باستخدامه.

بعد الانتهاء من تقييم قيمة المُعامل الأول، هناك مرحلة **نقاط التسلسل sequence points** التي سنشرحها لاحقًا.

3.4.2 عامل الفاصلة

يربح هذا العامل جائزة "أكثر العوامل غرابة"، إذ يسمح لك بإنشاء قائمة طويلة من التعبيرات المفصول بينها بالفاصلة:

```
expression-1,expression-2,expression-3,...,expression-n
```

يمكنك استخدام أي عدد من التعبيرات، وتُقيّم **التعبيرات** من اليسار إلى اليمين حصريًا وتُهمل قيمها، عدا التعبير الأخير الذي يحدد قيمة ونوع التعبير كاملًا. لا تخلط هذه الفاصلة مع الفواصل المستخدمة في لغة سي C لأغراض أخرى، بالأخص الفواصل المُستخدمة للفصل بين وسطاء الدالة. إليك عدة أمثلة لاستخدام هذا العامل:

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int i, j;

    /* comma used - this loop has two counters */
    for(i=0, j=0; i <= 10; i++, j = i*i){
        printf("i %d j %d\n", i, j);
    }

    /*
    * In this futile example, all but the last
    * constant value is discarded.
    * Note use of parentheses to force a comma
    * expression in a function call.
    */
    printf("Overall: %d\n", ("abc", 1.2e6, 4*3+2));
    exit(EXIT_SUCCESS);
}
```

[مثال 11.3]

عامل الفاصلة مُتجاهل أيضًا، إلا إذا أردت تجربة هذه الميزة واستخدامها شخصيًا، لذلك لن تراها إلا في بعض المناسبات الخاصة.

بعد تقييم كل مُعامل، تأتي مرحلة **نقاط التسلسل sequence points** التي سنشرحها لاحقًا.

3.5 خاتمة

استعرضنا في هذا الفصل جميع أدوات التحكم بتدفق البرنامج المتاحة في لغة سي C، وكانت النقاط المفاجئة في هذا الفصل، هي كيفية عمل حالات تعليمة `switch` بصورة منفردة وغير معتمدة على بعضها بعضًا، وعدم إمكانية تعليمة `goto` الانتقال إلى أي نقطة في دالةٍ أخرى عدا الدالة الموجودة فيه، وهذه النقاط غير معقدة للغاية ولا تسبب عادةً أي مشاكل سواءً للمبتدئين أو مبرمجي لغات أخرى.

تعطي جميع التعبيرات المنطقية قيمة عدد صحيح دائمًا، وربما يكون هذا خارج عن العادة بعض الشيء لكنه لن يستغرق الكثير من الوقت للاعتياد عليه.

لعل الأجزاء الأكثر غرابةً في هذا الفصل هي طريقة العامل الشرطي "?:" وعامل الفاصلة، إذ قد يعتقد البعض أن التخلص من العامل الشرطي أمرٌ ممكن لأنه غير مناسب ومتوافق لشيفرة برمجية مكتوبة مسبقًا، لكن لعامل الفاصلة استخدامات مهمة، وبالأخص المولدات التلقائية لبرامج لغة سي C.

لم يؤثر المعيار كثيرًا على النقاط المذكورة في هذا الفصل. وينبغي أن يتأكد جميع مستخدمو لغة سي C المستقبليين من فهمهم لجميع المواضيع المذكورة (باستثناء العامل الشرطي وعامل الفاصلة)، إذ إنها ضروريةٌ للاستخدام العملي للغة وغير صعبة.

3.6 تمارين

1. ما هو النوع والقيمة الناتجة عن العوامل العلائقية؟
2. ما هو النوع والقيمة الناتجة عن العوامل المنطقية ("`&&`" و"`||`" و"`!`")؟
3. ما هو الشيء غير الاعتيادي بشأن العوامل المنطقية؟
4. ما هي فائدة `break` في تعليمات `switch`؟
5. لماذا تُعد تعليمة `continue` غير مفيدة في تعليمات `switch`؟
6. ما هي المشاكل المحتملة الناتجة عن استخدام `continue` في تعليمة `while`؟
7. كيف يمكنك الانتقال من دالةٍ إلى أخرى؟

خُدُسات

لبيع وشراء الخدمات المصغرة

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من \$5 فقط

تصفح الخدمات

4. الدوال Functions

تكلّمنا سابقًا عن أهمية الدوال في لغة سي C وكيف أنها تشكّل اللبنة الأساسية لبرامج سي. إذًا، ليس من المُستغرب أن نخصص هذا الفصل بالكامل للتعريف عنها وعن استخدامها، إذ سننظرُ إلى كيفية التصريح عنها واستخدامها بالإضافة إلى أنواع وسطائها وبعض الأمثلة العملية.

4.1 ما التغييرات التي طرأت على لغة سي المعيارية بخصوص الدوال؟

كانت أسوأ ميزة في لغة سي القديمة هي عدم إمكانية التصريح عن عدد وأنواع وسطاء الدالة، إذ لم يكن ممكنًا للمصرّف compiler حينها أن يتحقق من صحة استخدام الدالة ضمن البرنامج وفق تصريحها، وعلى الرغم من أنها لم تؤثر على نجاح لغة سي، إلا إنها تسببت بمشاكل تخص قابلية نقل البرنامج وصيانتها التي كان من الممكن تفاديها جميعًا.

غيّرت لغة سي المعيارية من ذلك بقدمها، إذ أصبح من الممكن الآن التصريح عن الدوال بطريقة تسمح للمصرّف بالتحقق من صحة استخدامها، وهذه الطريقة الجديدة متوافقة كثيرًا مع الطريقة القديمة، لذا ستعمل البرامج المكتوبة بلغة سي القديمة دون أي أخطاء، شرط ألا يحتوي البرنامج طبعًا على أخطاء. يُعد استخدام عدة متغيرات على أنها وسطاء عند استخدام الدالة ميزةً أخرى مفيدة مثل استخدام الدالة `printf` التي اعتاد استخدامها أن يكون غير قابل للنقل، وكانت الطريقة الوحيدة التي يمكن استخدامها لتصبح قابلة للنقل هي بالاعتماد على معرفة عميقة بالعتاد الصلب.

أخذت لغة سي المعيارية الحل لهذه المشاكل من لغة ++C، إذ سبق لها وطبّقت هذه الأفكار بنجاح، كما تبنت العديد من مصرّفات لغة سي القديمة هذه الحلول من C المعيارية نظرًا لنجاحها.

ستظل لغة سي المعيارية متوافقةً مع طريقة تصريح الدوال في سي القديمة بهدف الإبقاء على صحة عمل البرامج السابقة فحسب، ويجب على أي برنامج يُكتب من جديد استخدام هذه الطريقة الجديدة الأكثر صرامةً التي قدمتها سي المعيارية وتقادي طريقة لغة سي القديمة بشدة، والتي ستندثر في المستقبل على الأرجح.

4.2 أنواع الدوال

تمتلك جميع الدوال نوعًا ما، والذي يكون هو نوع القيمة التي تُعيدها عند استخدامها. السبب في عدم احتواء لغة سي على "إجراءات procedures"، والتي هي في معظم اللغات دوال بدون قيمة، هو أنها تسمح بصورة إجبارية في بعض الأحيان بتجاهل القيمة النهائية لمعظم التعابير، وإن فاجأك ذلك تذكر تعبير الإسناد التالي:

```
a = 1;
```

الإسناد السابق صالح ويعيد قيمةً ما، لكن القيمة تُهمل. إذا أردت مفاجأة أكبر من السابقة، جرّب التعبير التالي:

```
1;
```

إذ إنه تعبير متبوعٌ بفاصلةٍ منقوطة، وهذه تعليمةٌ صالحةٌ وفق قواعد اللغة ولا يوجد أي خطأ فيها، ولكنها عديمة الفائدة. يمكنك التفكير بخصوص الدالة التي تُستخدم مثل إجراء بنفس الطريقة، إذ إنها تُعيد قيمةً **دائمًا** لكنها لا تُستخدم:

```
f(argument);
```

التعبير السابق هو تعبير ذو قيمةٍ مُهملة.

من السهل فهم نقطة أن القيمة المُعادة من الدالة يمكن إهمالها، لكن هذا يعني أن عدم استخدام القيمة المُعادة هو خطأ برمجي، وعلى العكس تمامًا إن لم يكن هناك أي قيمة مفيدة مُعادة من الدالة، إذًا من الأفضل أن يكون لدينا القدرة على مراقبة فيما إذا كانت القيمة مُستخدمةً عن طريق الخطأ، وللسببين السابقين، يجب التصريح عن أي دالة بكونها لا تعيد أي قيمة مفيدة بالنوع void.

يمكن أن تُعيد الدوال أي نوع مدعوم من لغة سي C عدا المصفوفات arrays والدوال بحد ذاتها، وهذا يتضمن المؤشرات pointers والبُنى structures والاتحادات unions، وستتكم عنهم لاحقًا. يمكننا التحايل على الأنواع التي لا يُمكن إعادتها من الدوال باستخدام المؤشرات بدلًا منها، كما يمكن استدعاء جميع الدوال تعاوديًا recursively.

4.2.1 التصريح عن الدوال

علينا الآن للأسف تقديم بعض المصطلحات بهدف التقليل من النص الوصفي المتكرر بعد ذكر كل مصطلح برمجي للوصول إلى نتيجة أقصر وأكثر دقة دون أي تشويش للقارئ، إليك المصطلحات:

- **التصريح declaration**: نذكر فيه النوع type الذي يرتبط باسم ما.
- **التعريف definition**: يماثل التصريح، لكنه يحجز أيضًا مساحة تخزينية للكائن المذكور، وقد تكون القواعد التي تفصل بين التصريح والتعريف معقدة، لكن الأمر بسيط بالنسبة للدوال؛ إذ يصبح التصريح تعريفًا عندما يُضاف محتوى الدالة على أنه تعليمة مُركّبة compound statement.
- **المُعاملات parameters والمعاملات الصوريّة formal parameters**: الأسماء التي تُشير إلى الوسطاء بداخل الدالة.
- **الوسطاء arguments والوسطاء الفعلية actual arguments**: القيم المُستخدمة مثل وسطاء في دالة ما، أي قيم المُعاملات الصوريّة عند تنفيذ الدالة.

يُستخدم المصطلحان "مُعامل" و"وسيط" على نحوٍ تبادلي، لذا لا تتساءل عن سبب استخدامنا لمصطلح عن الآخر في الفقرات القادمة.

تُصرّح الدالة ضمنيًا على أنها "تُعيد قيمة من نوع int"، إذا استخدمتها قبل التصريح عنها، ولكن تعد هذه من الممارسات الخاطئة في لغة سي المعيارية، على الرغم من استخدام هذه الطريقة على نحوٍ واسع في لغة سي القديمة؛ إذ يؤدي استخدام الدوال دون التصريح عنها إلى مشاكل معقدة مرتبطة بعدد ونوع الوسطاء المُتوقّعة، ويجب أن يُصرّح عن الدوال بصورةٍ كاملة قبل استخدامها. على سبيل المثال، إذا أردت استخدام دالة موجودة في مكتبة خاصة، لا تأخذ أي وسطاء، وتعيد القيمة double، وتسمى aax1، فعليك التصريح عنها كما يلي:

```
double aax1(void);
```

وإليك مثالاً عن استخدامها الخاطئ:

```
main(){
    double return_v, aax1(void);
    return_v = aax1();
    exit(EXIT_SUCCESS);
}
```

[مثال 1]

التصريح في المثال السابق مثير للاهتمام، إذ عرّفنا `return_v` مما تسبب بإنشاء متغير جديد، كما صرّحنا عن `ax1` دون تعريفها، إذ أن الدوال تُعرّف فقط في حالة وجود متن الدالة، كما ذكرنا سابقاً، وفي هذه الحالة يُفترض أن تُعيد الدالة `ax1` النوع `int` ضمناً مع أنها تعيد النوع `double`، مما يعني أن ذلك سيتسبب بتصرف غير محدد، وهو أمر كارثي دائماً.

يوضّح وجود النوع `void` ضمن لائحة الوسائط عند التصريح بأن الدالة لا تقبل أي وسيط، وإن كانت مفقودةً من لائحة الوسائط فلن يترك التصريح أي معلومات عن وسائط الدالة، وبهذه الطريقة نحافظ على التوافقية مع لغة سي C القديمة على حساب قدرة المصرّف على التحقق.

ينبغي كتابة متن للدالة مثل تعليمة مركّبة حتى **تعرفها**، إذ لا يمكن لتعريف دالةٍ ما أن يكون محتوياً بتعريف دالةٍ أخرى. ونتيجةً لذلك، جميع الدوال مستقلةً عن بعضها بعضاً وموجودةً في المستوى الخارجي لهيكل البرنامج. يوضح التعريف التالي طريقةً ممكنةً للتعريف عن الدالة `ax1`:

```
double
ax1(void) {
    /*متن الدالة هنا*/
    return (1.0);
}
```

من غير الاعتيادي أن تمنعك لغةٌ تعتمد على الهيكلية الكُتلية عن تعريف الدوال داخل دوالٍ أخرى، ولكن هذه سمةٌ من سمات لغة سي C، ويساعد ذلك على تحسين الأداء وقت التنفيذ run-time للغة سي، لأنه يقلل المهام المطلوبة المتعلقة بتنظيم استدعاء الدوال.

4.2.2 تعليمة الإعادة `return`

تُعد تعليمة `return` مهمةً للغاية، إذ تستخدمها جميع الدوال -عدا التي تُعيد `void`- مرةً واحدةً على الأقل، وتوضّح التعليمة `return` عند ذكرها القيمة التي يجب أن تُعيدها. من الممكن أن نعيد قيمةً من دالةٍ ما عن طريق وضع `return` في نهاية الدالة قبل القوس المعقوص الأخير "}"؛ إلا أن هذا الأمر سيتسبب بتصرف غير محدد عند استخدامها في دالة تُعيد `void`، إذ ستُعاد قيمةً غير معروفة.

إليك مثلاً عن دالة أخرى تستخدم `getchar` لقراءة المحارف من دخل البرنامج ومن ثمّ تعيدها باستثناء المسافة `space` ومسافة الجدولة `tab` والأسطر الجديدة `newline`.

```
#include <stdio.h>

int
non_space(void){
```

```
int c;
while ( (c=getchar ())=='\t' || c== '\n' || c== ' ')
    ; /*تعليلة فارغة*/
return (c);
}
```

لاحظ عملية التحقق من المحارف عن طريق تعليمة `while`، والتي لا تحتوي على أي تعليمة في متنها، إذ أن وجود فاصلة منقوطة لوحدها ليست نادرة الحدوث وإنما هي شائعة الاستخدام، وعادةً ما تُصحّب بتعليق بسيط لمواساة وحدتها، لكن رجاءً ثم رجاءً لا تكتبها على النحو التالي:

```
while (something);
```

فمن السهل ألا تلاحظ الفاصلة المنقوطة في نهاية السطر عند قراءة البرنامج، وتفترض أن التعليمات أسفلها تتبع لتعليمة `while`.

يجب أن يكافئ نوع التعبير المُعاد نوع الدالة ضمن تعليمة `return`، أو على الأقل أن يكون بالإمكان تحويله ضمن تعليمة إسناد. على سبيل المثال، يمكن أن تحتوي دالة مُصرح عنها أنها تُعيد النوع `double` التعليمة:

```
return (1);
```

سيحوّل بذلك العدد الصحيح إلى نوع `double`، ومن الممكن أيضًا كتابة `return` دون أي تعبير مصاحب لها، لكن ذلك سيتسبب بخطأ برمجي إن استخدمت هذه الطريقة ما لم تُعيد الدالة النوع `void`. من غير المسموح إلحاق تعبير بتعليمة `return` إذا كانت الدالة تعيد النوع `void`.

4.2.3 وسطاء الدوال

لم يكن بالإمكان قبل مجيء لغة سي C المعيارية إضافة أي معلومات عن وسطاء الدالة عدا داخل تعريف الدالة نفسها، وكانت هذه المعلومات تُستخدم داخل متن الدالة فقط وتُنسى في نهاية المطاف. كان من الممكن -في تلك الأيام القديمة البائسة- تعريف دالة بثلاث وسطاء من نوع `double` وتميرر وسيط من نوع `int` لها عند استدعائها، وسيُصرّف البرنامج بصورة طبيعية دون إظهار أي أخطاء ولكنه لن يعمل بالنحو الصحيح، إذ كان من واجب المبرمج التحقق من صحة عدد وأنواع الوسطاء للدالة. كما ستوقع، كان هذا المسبب الأساسي لكثيرٍ من الأخطاء الأولية في البرنامج ومشكلات قابلية التنقل. إليك مثالاً عن تعريف دالة واستخدامها مع وسطائها لكن دون التصريح عن الدالة كاملاً.

```
#include <stdio.h>
#include <stdlib.h>
main(){
```

```

void pmax(); /* التصريح */
int i,j;
for(i = -10; i <= 10; i++){
    for(j = -10; j <= 10; j++){
        pmax(i,j);
    }
}
exit(EXIT_SUCCESS);
}
/*
* Function pmax.
* Returns: void
* Prints larger of its two arguments.
*/
void
pmax(int a1, int a2){ /*التعريف*/
    int biggest;

    if(a1 > a2){
        biggest = a1;
    }else{
        biggest = a2;
    }

    printf("larger of %d and %d is %d\n",
        a1, a2, biggest);
}

```

[مثال 2]

ما الذي يمكنك تعلمه من المثال السابق؟ بدايةً، لاحظ بحذر أن التصريح هو للدالة pmax التي تُعيد void، إذ يقع النوع void الموافق للدالة عند تعريفها السطر الذي يسبق اسمها، وهذه الطريقة في الكتابة أسلوبٌ وتحييدٌ شخصي، إذ من السهل إيجاد التصريح عن الدالة إذا كان اسم الدالة يقع في بداية السطر.

لا يشير التصريح عن الدالة في main إلى وجود أي وسطاء لها، إلا أن استخدام الدالة بعد عدة أسطر يدل على استخدام وسيطين، وهذا مسموح في كلِّ من لغة سي المعيارية وسي C القديمة ولكنه يُعد **ممارسةً برمجيةً سيئةً**، فمن الأفضل تضمين معلومات عن الوسطاء في التصريح عن الدالة كما سنرى لاحقًا. هذا

الأسلوب القديم في الكتابة هو ميزةٌ عفا عليها الزمن ومن الممكن أن تختفي في إصدارات سي المعيارية القادمة.

دعنا ننتقل الآن إلى تعريف الدالة حيث يقع متنها، ولاحظ أنه يدل على أن الدالة تأخذ وسيطين باسم a1 و a2، كما أن نوع الوسطاء محدد بالنوع int.

لا يتوجب عليك تحديد نوع كلِّ الوسطاء في هذه الحالة لأن النوع سيكون int افتراضيًا، ولكن هذه ممارسة سيئة، إذ يجب عليك الاعتياد على تحديد نوع الوسطاء في كل مرة حتى لو كانت من نوع int، لأن ذلك يسهل عملية قراءة شيفرة البرنامج ويشير إلى أنك تعتمد استخدام النوع int ولم يحصل ذلك عن طريق الخطأ بنسيانك لتحديد نوع الوسيط. **يمكن** تعريف الدالة pmax بهذه الطريقة ولكنها ممارسة سيئة كما ذكرنا سابقًا:

```
/* مثال سيء على التصريح عن الدالة */
```

```
void
pmax(a1, a2){
    /* and so on */
```

الطريقة المثالية للتصريح والتعريف عن الدوال هي باستخدام ما يُدعى **النماذج الأولية prototypes**.

4.2.4 نماذج الدوال الأولية function prototypes

كان تقديم **نماذج الدوال الأولية function prototypes** من أكبر التغييرات في لغة سي C المعيارية؛ إذ أن نموذج الدالة الأولية هو تصريح أو تعريف يتضمن معلومات عن عدد وأنواع الوسطاء التي تستخدمها الدالة. على الرغم من إمكانية إهمال تحديد أي معلومات بخصوص وسطاء الدالة عند التصريح عنها، إلا أن ذلك يحدث بهدف التوافقية مع لغة سي القديمة فقط، وينبغي تجنُّبه، ولا يعدّ التصريح دون أي معلومات عن وسطاء الدالة نموذجًا أوليًا. إليك المثال السابق مكتوبًا "بطريقة صحيحة":

```
#include <stdio.h>
#include <stdlib.h>

main(){
    void pmax(int first, int second);    /*التصريح*/
    int i,j;
    for(i = -10; i <= 10; i++){
        for(j = -10; j <= 10; j++){
            pmax(i,j);
        }
    }
}
```



```

    }
    exit(EXIT_SUCCESS);
}

void
pmax(int a1, int a2){ /*التعريف*/
    int biggest;

    if(a1 > a2){
        biggest = a1;
    }
    else{
        biggest = a2;
    }

    printf("largest of %d and %d is %d\n",
        a1, a2, biggest);
}

```

[مثال 3]

احتوى التصريح هذه المرة على معلومات بخصوص وسطاء الدالة، لذا يصنّف على أنه نموذج أولي. لا يُعد الاسمان `first` و `second` جزءاً ضرورياً من التصريح، لكنهما موجودان لتصبح عملية الإشارة إلى الوسطاء عند توثيق عمل الدالة عمليّةً أسهل، إذ نستطيع وصف عمل الدالة ببساطة عن طريق استخدامهما عن طريق التصريح التالي:

```
void pmax (int xx, int yy );
```

وهنا نستطيع القول أن الدالة `pmax` تطبع القيمة الأكبر ضمن الوسيطين `xx` و `yy` بدلاً عن الإشارة إلى الوسطاء بتموضعها، أي الوسيط الثاني والأول وهكذا، لأنها طريقة معرّضة لسوء الفهم أو الخطأ في عدّ الترتيب. عدا عن ذلك، تستطيع التخلص من أسماء الوسطاء في تصريح الدالة ويكون التصريح التالي مساوياً للتصريح السابق:

```
void pmax (int,int);
```

الفرق بين التصريحين هو فقط أسماء الوسطاء.

يكون التصريح عن دالة لا تأخذ أي وسطاء على النحو التالي:

```
void f_name (void);
```

وللتصريح عن دالة تأخذ وسيطًا من نوع `int` وآخرًا من نوع `double` وعددًا غير محدد من الوسطاء الآخرين، نكتب:

```
void f_name (int,double,...);
```

توضّح هنا النقاط الثلاث . . . أن هناك المزيد من الوسطاء، وهذا مفيدٌ في حال كانت الدالة تسمح بوجود عددٍ غير محدد من الوسطاء، مثل دالة `printf`، إذ أن تصريح الدالة هذه يكون على النحو التالي:

```
int printf (const char *format_string,...)
```

الوسيط الأول هو "مؤشر `pointer` للقيمة من النوع `const char`"، وسنناقش معنى ذلك لاحقًا. يتحقق المصرّف من استخدام الدالة بما يتوافق مع تصريحها وذلك حالما يُعلم بأنواع وسطاء الدالة أثناء قراءتهم من النموذج الأولي للدالة، وتحوّل قيمة الوسيط إلى النوع الصحيح حسب النموذج الأولي في حال استُديت الدالة باستخدام نوع وسيطٍ مغاير "بصورةٍ مشابهة للتحويل الحاصل عند عملية الإسناد". إليك مثالًا توضيحيًا: دالةٌ تحسب الجذر التربيعي لقيمةٍ ما باستخدام طريقة نيوتن للتقريبات المتعاقبة `Newton's method of successive approximations`.

```
#include <stdio.h>
#include <stdlib.h>
#define DELTA 0.0001
main(){
    double sq_root(double); /*النموذج الأولي*/
    int i;

    for(i = 1; i < 100; i++){
        printf("root of %d is %f\n", i, sq_root(i));
    }
    exit(EXIT_SUCCESS);
}

double
sq_root(double x){ /*التعريف*/
    double curr_appx, last_appx, diff;
```

```

last_appx = x;
diff = DELTA+1;

while(diff > DELTA){
    curr_appx = 0.5*(last_appx
        + x/last_appx);
    diff = curr_appx - last_appx;
    if(diff < 0)
        diff = -diff;
    last_appx = curr_appx;
}
return(curr_appx);
}

```

[مثال 4]

يوضح النموذج الأولي للدالة أن `sqrt` تأخذ وسيطاً واحداً من نوع `double`، وفي حقيقة الأمر، تُمرّر قيمة الوسيط ضمن الدالة `main` بنوع `int` لذا يجب تحويلها إلى `double` أولاً؛ لكن يجدر الانتباه هنا إلى أن سي ستفترض أن المبرمج تقصد تمرير قيمة `int`، إذا لم يكن هناك أي نموذج أولي وفي هذه الحالة ستُعامل القيمة على أنها `int` دون تحويل.

يشير المعيار ببساطة إلى أن هذا سيتسبب بسلوك غير محدد، إلا أن ذلك الوصف يقلل من خطورة الخطأ تماماً مثل القول أن الإصابة بمرض قاتل أمرٌ مؤسفٌ فقط، إذ أن هذا الخطأ **خطير جداً** وتسبب سابقاً في برامج سي القديمة بالكثير والكثير من المشكلات.

السبب في التحويل من `int` إلى `double` في هذه الحالة هو بسبب رؤية المصنّف للنموذج الأولي مما دلّه على ما يجب فعله، وكما توقعنا، هناك العديد من القواعد المستخدمة لتحديد التحويل المناسب في كل حالة، وستنكلم عنها.

4.2.5 تحويلات الوسطاء

تُجرى عدة تحويلات عند استدعاء دالةٍ ما حسب كل حالة وفقاً لقيم الوسطاء وحسب وجود أو عدم وجود النموذج الأولي، ولا بُدّ أن نوضح قبل أن نبدأ أنه **من الممكن** لك استخدام هذه القوانين لمعرفة قيم الوسطاء الناتجة دون استخدام نموذج أولي، ولكن هذه وصفتها لكارثة تُطهى على نار هادئة، ولا يوجد أي عذر لعدم استخدام النماذج الأولية فهي سهلة الاستخدام؛ لذا استخدم هذه القواعد فقط في الدوال ذات عدد الوسطاء المتغيّر باستخدام علامة الاختصار Ellipsis "..." كما سنشرح لاحقاً.

تتضمن هذه القواعد **ترقيات الوسطاء الافتراضية default argument promotions والأنواع المتوافقة compatible types**. وفيما يخص ترقية الوسطاء الافتراضية، فهي:

- تُطبَّق الترقية العددية الصحيحة على كل قيمة وسيط.
- يُحوَّل الوسيط إلى نوع `double` إذا كان نوعه `float`.

أبرزَ ظهور النماذج الأولية -إضافةً إلى أشياء أخرى- الحاجة للدقة بخصوص "الأنواع المتوافقة"، التي لم تكن مشكلةً في سي القديمة. سنستعرض لأحة قوانين الأنواع المتوافقة كاملةً لاحقاً، لأننا نعتقد أن معظم مبرمجي لغة سي لن يكونوا محتاجين لتعلمها كاملةً، وخصوصاً عن ذلك سنكتفي في الوقت الحالي بمعرفة أن الأنواع المتماثلة متوافقة فيما بينها.

تُطبَّق التحويلات بالاعتماد على القوانين التالية (ليست اقتباساً مباشراً من المعيار، بل ستدلك على كيفية تطبيق قوانين سي المعيارية):

أولاً، تُرقى وسطاء الدالة وفق ترقية الوسطاء الاعتيادية، إذا لم يكن هناك أي نموذج أولي يسبق نقطة استدعاء الدالة، وذلك وفق التفاصيل:

- نحصل على سلوكٍ غير محدد إذا كان عدد الوسطاء المزوّد لا يكافئ عدد الوسطاء الفعلي للدالة.
- يجب أن تكون أنواع الوسطاء المزوّد للدالة **متوافقة** مع أنواع الوسطاء الفعلية وفق تعريف الدالة بعد تطبيق الترقية عليهم وذلك إذا لم يكن لتعريف الدالة نموذج أولي، ونحصل على سلوك غير محدد فيما عدا ذلك.
- نحصل على سلوك غير محدد إذا لم تحتوي الدالة على نموذج أولي في تعريفها وكانت أنواع الوسطاء المزوّد غير متوافقة مع أنواع الوسطاء الفعلية، كما يكون السلوك غير محدد أيضاً إذا تضمّن النموذج الأولي للدالة علامة الاختصار "...".

ثانياً، تُحوَّل الوسطاء إلى الأنواع المُسندة إلى كلٍّ منها وفقاً للنموذج الأولي، وبصورةٍ مشابهة للتحويل الذي يحصل عند الإسناد، وذلك إذا كان النموذج الأولي **ضمن** النطاق `scope` عند استدعاء الدالة، ويشمل ذلك جميع المتغيرات في لأحة الوسطاء بما فيها المتغيرات المُشار إليها بعلامة الاختصار "...".

من الممكن كتابة برنامج يحتوي على نموذج أولي ضمن النطاق عند استدعاء الدالة لكن دون وجود نموذج أولي داخل تعريف الدالة، وهذا طبقاً أسلوب سيء جداً، وفي هذه الحالة يجب على نوع الدالة المُستدعاة أن يكون **متوافقاً** مع النوع المُستخدم عند استدعاء هذه الدالة.

لا يحدّد المعيار صراحةً ترتيب تقييم وسطاء الدالة عند استدعائها.

4.2.6 تعريف الدوال

تسمح النماذج الأولية للدوال باستخدام النص ذاته للتصريح عن الدالة والتعريف عنها.

لتحويل تصريح الدالة التالي إلى تعريف:

```
double
some_func(int a1, float a2, long double a3);
```

نضيف متناً إلى الدالة:

```
double
some_func(int a1, float a2, long double a3){
    /* متن الدالة */
    return(1.0);
}
```

وذلك باستبدال الفاصلة المنقوطة في نهاية التصريح بتعليمة مركبة.

يعمل تعريف الدالة أو التصريح عنها مثل نموذج أولي شرط تحديد أنواع المُعاملات parameters، ويُعدّ المثالين السابقين نموذجين أوليين.

ما تزال لغة C المعيارية تدعم طريقة سي القديمة في التصريح عن دالة باستخدام وسائطها، ولكن يجب تجنّب استخدامها. يمكننا التصريح عن الدالة بالطريقة المذكورة على النحو التالي:

```
double
some_func(a1, a2, a3)
    int a1;
    float a2;
    long double a3;
{
    /* متن الدالة */
    return(1.0);
}
```

لا يمثّل التعريف السابق نموذجاً أولياً، لعدم وجود أي معلومات بخصوص المُعاملات عند تسميتها، ويقدم التعريف السابق معلومات حول النوع الذي تُعيده الدالة، وبهذا لا يتذكر المصنّف Compiler أي معلومات تخص أنواع الوسائط بنهاية التعريف.

يحدّر المعيار بخصوص هذه الطريقة بقوله أنها ستختفي غالباً في إصدارات قادمة، ولذلك لن نذكرها مرةً أخرى.

دعنا نلخص ما تكلمنا عنه سابقاً بإيجاز:

- يمكن استدعاء الدوال على نحوٍ تعاودي.
- يمكن للدوال إعادة أي قيمة تصرّح عنها عدا المصفوفات والدوال (إلا أنه يمكنك التحايل على هذا القيد باستخدام المؤشرات)، ويجب أن تكون الدوال من النوع `void` إذا لم تُعد أي قيمة.
- استخدم نماذج الدوال الأولية دائماً.
- نحصل على سلوك غير محدد، إذا استُديت دالة أو تعريفها إلا إذا:
 - كان النموذج الأولي دائماً ضمن النطاق في كل مرة تُستدعى فيها الدالة أو تُعرّف.
 - كنت حريصاً جداً بهذا الخصوص.
- تُحوّل قيم الوسطاء عند استدعاء الدالة إلى أنواع المُعاملات الفعلية للدالة (المعرّفة وفقها)، بصورةٍ مشابهة للتحويل عند عملية الإسناد باستخدام العامل `operator =`، وذلك **بفرض** أنك تستخدم نموذجاً أولياً.
- يجب أن يشير النموذج الأولي إلى النوع `void`، إذا لم تأخذ الدالة أي وسطاء.
- يجب تحديد اسم وسيط واحد على الأقل في دالة تقبل عدداً متغيراً من الوسطاء، ومن ثم الإشارة إلى العدد المتغير من الوسطاء بالعلامة "..."، كما هو موضح:

```
int
vfunc(int x, float y, ...);
```

سنناقش لاحقاً استخدام هذا النوع من الدوال.

4.2.7 التعليمات المركبة والتصريحات

يتألف متن الدالة من تعليمة مركبة `Compound statement`، ومن الممكن التصريح عن متغيرات جديدة داخل هذه التعليمة المركبة. تغطّي أسماء المتغيرات الجديدة على أسماء المتغيرات الموجودة مسبقاً، إذا تشابهت أسماءهم ضمن التعليمة المركبة، وهذا مماثلٌ لأي لغة تعتمد تنسيقاً كتلياً مشابهاً للغة سي. تقيّد لغة سي `C` التصريحات لتكون ضمن بداية التعليمة المركبة أو "الكتلة البرمجية"، ويُمنع استخدام التصريحات داخل الكتلة حالما يُكتب أي نوع من التعليمات `statements` داخلها.

كيف يمكن التغطية على أسماء المتغيرات؟ يوضح المثال التالي ما نقصد بذلك:

```

int a; /* يمكن رؤيته من هذه النقطة فصاعدًا */

void func(void){
    float a; /* يمثل متغير a مختلف */
    {
        char a; /* متغير a أيضًا مختلف */
    }

    /* يمكن رؤية المتغير ذو النوع قيمة حقيقية */
}

/* يمكن رؤية المتغير ذو النوع قيمة صحيحة */

```

[مثال 5]

عند التصريح عن اسم ما داخل كتلة فهذا يتسبب بإهمال أي اسم مشابه خارج الكتلة حتى الوصول لنهايتها، كما يمكنك التصريح عن الاسم ذاته في كتلة داخلية، وتكرار هذه العملية إلى ما لانهاية.

نطاق Scope الاسم هو المجال الذي يكون فيه لهذا الاسم معني، ويبدأ نطاق الاسم من نقطة ذكر الاسم إلى نهاية الكتلة التي دُكر فيها، ويستمر إلى نهاية الملف إذا كان الاسم خارجي External (خارج أي دالة)، ويختفي في نهاية الدالة إذا كان الاسم داخلي Internal (داخل دالة)، ويمكن إعادة تعيين النطاق عند إعادة التصريح عن الاسم داخل الكتلة ذاتها.

يمكنك تنفيذ حيل طريقة مثل المثال التالي باستخدام قوانين النطاق:

```

main () {}
int i;
f () {}
f2 () {}

```

يمكن للدالتين f و f2 استخدام المتغير i، لكن main لا تستطيع لأن التصريح عن الدالة أتى بعد الدالة main، ولا تُستخدم كثيرًا هذه الطريقة بالضرورة ولكنها تستفيد من طريقة لغة سي C الضمنية في معالجة التصريحات. قد تتسبب هذه الطريقة ببعض من الحيرة لمن يقرأ ملف الشيفرة البرمجية، ويجب تجنُّبها عن طريق التصريح عن المتغيرات الخارجية قبل تعريف أي دالة في الملف.

غيّرت لغة سي C المعيارية بعض الأمور بخصوص معاملات الدالة الفعلية، إذ افترض تصريحها داخل التعليمة المركبة الأولى حتى لو لم تكن هذه الحالة محققة فعلاً كتابيًا، وهذا ينطبق على الطريقة القديمة والجديدة في التعريف عن الدالة. بناءً على ما سبق، نحصل على خطأ إذا كان اسم معاملات الدالة الفعلية مطابقًا لاسم قد صُرح عنه في التعليمة المركبة الخارجية.

كان خطأ إعادة التعريف غير المقصود في لغة سي C القديمة خطأً صعباً للتتبع والحل، إليك ما قد يبدو عليه الخطأ:

```
/* إعادة تصريح خاطئة للوسطاء */

func(a, b, c){
    int a; /* AAAAgh! */
}
```

الجزء المسبب للمتاعب هنا هو التصريح الجديد للمتغير a في متن الدالة، الذي سيغطي على المعامل a، ولن نتكلم بالمزيد عن هذه المشكلة بما أنها غير ممكنة الحدوث بعد الآن.

4.3 مفهوم التعاود Recursion وتميرير الوسطاء إلى الدوال

نظرنا سابقاً إلى كيفية تعيين نوع للدالة Function type (كيفية التصريح عن القيمة المُعادَة ونوع أيّ وسيط argument تأخذ الدالة)، وأن تعريف definition الدالة يمثّل متنها أو جسمها body، ولننظر الآن إلى استخدامات الوسطاء.

4.3.1 استدعاء الوسيط بقيمته call by value

تُعامل لغة سي C وسطاء الدالة بطريقة بسيطة وثابتة دون أي استثناءات فردية؛ إذ تُعامل وسطاء الدالة عندما تُستدعى الدالة مثل أي تعبير اعتيادي، وتُحوّل قيم هذه التعبيرات وتُستخدم فيما بعد لتهيئة القيمة الأولية لمعاملات الدالة المُستدعاة الموافقة، التي تتصرف بدورها مثل أيّ متغير محلي داخل الدالة، كما هو موضح في المثال:

```
void called_func(int, float);

main(){
    called_func(1, 2*3.5);
    exit(EXIT_SUCCESS);
}

void
called_func(int iarg, float farg){
    float tmp;

    tmp = iarg * farg;
```


}

[مثال 6]

يوجد للدالة `called_func` تعبيران يحلّان محل الوسطاء في دالة `main`، وتُقيّم قيمتهما ويُستخدمان في إسناد قيمة مبدئية للمعاملين `irag` و `frag` في الدالة `called_func`، ويملك المعاملان الخصائص ذاتها التي يملكها أي متغير داخلي مصرّح عنه في الدالة `called_func` دون أي تفريق، مثل `tmp`. تُعد عملية إسناد القيمة المبدئية للمعاملات الفعلية التواصل الأخير بين مستدعي الدالة والدالة المُستدعاة، إذا استثنينا القيمة المُعادة.

يجب أن ينسى من اعتاد البرمجة باستخدام فورتران FORTRAN وباسكال Pascal طريقة استخدام وسطاء من نوع `var`، والتي **يمكن** للدالة أن تغير من قيم وسطائها؛ إذ لا يمكنك في لغة سي أن تؤثر على قيم وسطاء الدالة بأي شكل من الأشكال، إليك مثالاً نوضّح فيه المقصود.

```
#include <stdio.h>
#include <stdlib.h>
main(){
    void changer(int);
    int i;

    i = 5;
    printf("before i=%d\n", i);
    changer(i);
    printf("after i=%d\n", i);
    exit(EXIT_SUCCESS);
}

void
changer(int x){
    while(x){
        printf("changer: x=%d\n", x);
        x--;
    }
}
```

[مثال 7]

ستكون نتيجة المثال السابق على النحو التالي:

```
before i=5
changer: x=5
changer: x=4
changer: x=3
changer: x=2
changer: x=1
after i=5
```

تستخدم الدالة `changer` معاملها الفعلي `x` بمثابة متغير اعتيادي (وهو فعلاً متغير اعتيادي)، ورغم أن قيمة `x` تغيرت إلا أن المتغير `i` في الدالة `main` لم يتأثر بالتغيير، وهذه هي النقطة التي نريد توضيحها لك بمثالنا، إذ تُمرّر الوسطاء في سي `C` إلى الدالة باستخدام قيمها فقط ولا تُمرر أي تغييرات من الدالة بالمقابل.

4.3.2 استدعاء الوسيط بمرجعه `call by reference`

من الممكن كتابة دوال تأخذ **المؤشرات** `pointers` على أنها وسطاء، مما يعطينا شكلاً من أشكال الاستدعاء بالمرجع. سنناقش هذا لاحقاً، إذ ستسمح هذه الطريقة للدالة بتغيير قيم المتغيرات التي استدعتها.

4.3.3 التعاود `Recursion`

بعد أن تكلمنا عن كيفية تمرير الوسطاء بأمان، حان الوقت للتكلم على **التعاود** `Recursion`، إذ يثير هذا الموضوع جدلاً طويلاً غير مثمر بين المبرمجين، فالبعض يعدّه رائعاً ويستخدمه متى ما أتيحت له الفرصة، بينما يتجنّب الطرف الآخر استخدامه بأي ثمن، لكن دعنا نوضح أنك **ستضطر** لاستخدامه في بعض الحالات دون أي مفرّ. لا يتطلب دعم التعاود أيّ جهد إضافي لتضمينه في أي لغة برمجة، وبذلك -وكما توقّعت- تدعم لغة سي `C` التعاود.

يمكن لأي دالة أن تستدعي نفسها من داخلها أو من داخل أي دالة أخرى في لغة سي، ويتسبب كل استدعاء للدالة بحجز متغيرات جديدة مصرّح عنها داخل الدالة، وفي الحقيقة، كانت تفتقر التصريحات التي استخدمناها حتى اللحظة إلى شيء ما، ألا وهو الكلمة المفتاحية `auto` التي تعني "الحجز التلقائي".

```
/* مثال عن الحجز التلقائي */
main(){
    auto int var_name;
    .
    .
    .
}
```

تُحجز وتُحرَّر مساحة التخزين للمتغيرات التلقائية تلقائيًا عند البدء بتنفيذ الدالة وعند إعادتها للقيمة، أي الخروج منها، وبذلك سيحتاج البرنامج فقط لحجز مساحة لمصفوفتين مثلًا، في حال صرّحت دالتين عن مصفوفتين تلقائيتين automatic ونُفذت الدالتان في الوقت نفسه.

على الرغم من كون "auto" كلمة مفتاحية في لغة سي، لكنها لا تُستخدم عمليًا لأنها الحالة الافتراضية لعمليات التصريح عن المتغيرات الداخلية وغير صالحة في حال استخدامها مع عمليات التصريح عن المتغيرات الخارجية. تكون قيمة المتغير التلقائي غير معروفة عند التصريح عنه إذا لم تُسند أي قيمة ابتدائية إليه، وسيتسبب استخدام قيمة المتغير في هذه الحالة في ظهور سلوكٍ غير محدد.

يجب علينا انتقاء الأمثلة التي سنشرح عن طريقها مفهوم التعاود، إذ لا توضّح الأمثلة البسيطة مفهوم التعاود على النحو المناسب، والأمثلة التي توضح المفهوم كاملًا صعبة الفهم على المبتدئين، الذين يواجهون بعض الصعوبة في التمييز بين التصريح والتعريف على سبيل المثال، وستنكلم لاحقًا عن مفهوم التعاود وفائدته باستخدام بعض الأمثلة عندما نتكلم عن هياكل البيانات.

يوضح المثال التالي برنامجًا يحتوي دالةً تعاوديةً تتحقق من التعابير المُدخلة إليها بما فيها الأرقام (0-9) والعوامل "*" و "%" و "/" و "+" و "-"، إضافةً إلى الأقواس، بالطريقة نفسها التي تستخدمها لغة سي، كما استخدم ستروستروب Stroustrup في كتابه عن لغة C++ المثال ذاته لتوضيح مفهوم التعاود، وهذا من قبيل الصدفة لا غير.

يُقيّم التعبير في المثال التالي، ثم تُطبع قيمته إن صادف محرفًا غير موجودًا في لغته (المحارف التي ذكرناها سابقًا)، ولغرض البساطة لن يكون في المثال أي طريقة للتحقق من الأخطاء. يعتمد المثال كثيرًا على الدالة ungetc التي تسمح للمحرف الأخير الذي قُرأ بواسطة الدالة getchar أن يُعيّن على أنه "غير مقروء" للسماح بقراءته مرةً أخرى، والمُعامل الثاني المُستخدم في المثال مُصرّح عنه في `stdio.h`.

سيرغب من يفهم صيغة باكوس نور BNF بمعرفة أن التعبير سيُفهم عن طريق استخدام الصيغة التالية:

```
<primary> ::= digit | (<exp>)
<unary>   ::= <primary> | -<unary> | +<unary>
<mult>    ::= <unary> | <mult> * <unary> |
              <mult> / <unary> | <mult> % <unary>
<exp>     ::= <exp> + <mult> | <exp> - <mult> | <mult>
```

يُمكن التعاود في مثالنا ضمن مكانين أساسيين، هما: الدالة `unary_exp` التي تستدعي نفسها، والدالة `primary` التي تستدعي الدالة الموجودة على المستوى العلوي للبرنامج (نقصد دالة `expr`) لتقييم التعابير المكتوبة بين قوسين.

حاول تشغيل البرنامج باستخدام كل من الأمثلة التالية إذا لم تفهم عمله، وتتبع عمله يدويًا على المُدخلات،

كما يلي:

```
1
1+2
1+2 * 3+4
1+--4
1+(2*3)+4
```

سيستغرق هذا بعض الوقت منك.

```
/*
 * برنامج يتحقق من تعابير لغة سي على نحوٍ تعاودي
 * لم يُشَدّد على حالات الإدخال الخاطئة من المستخدم
 */

#include <stdio.h>
#include <stdlib.h>

int expr(void);
int mul_exp(void);
int unary_exp(void);
int primary(void);

main(){
    int val;

    for(;;){
        printf("expression: ");
        val = expr();
        if(getchar() != '\n'){
            printf("error\n");
            while(getchar() != '\n')
                /* فارغ */;
        } else{
            printf("result is %d\n", val);
        }
    }
}
```

```
    }
    exit(EXIT_SUCCESS);
}

int
expr(void){
    int val, ch_in;

    val = mul_exp();
    for(;;){
        switch(ch_in = getchar()){
            default:
                ungetc(ch_in,stdin);
                return(val);
            case '+':
                val = val + mul_exp();
                break;
            case '-':
                val = val - mul_exp();
                break;
        }
    }
}

int
mul_exp(void){
    int val, ch_in;

    val = unary_exp();
    for(;;){
        switch(ch_in = getchar()){
            default:
                ungetc(ch_in, stdin);
                return(val);
            case '*':
                val = val * unary_exp();
```

```
        break;
    case '/':
        val = val / unary_exp();
        break;
    case '%':
        val = val % unary_exp();
        break;
    }
}

int
unary_exp(void){
    int val, ch_in;

    switch(ch_in = getchar()){
    default:
        ungetc(ch_in, stdin);
        val = primary();
        break;
    case '+':
        val = unary_exp();
        break;
    case '-':
        val = -unary_exp();
        break;
    }
    return(val);
}

int
primary(void){
    int val, ch_in;

    ch_in = getchar();
    if(ch_in >= '0' && ch_in <= '9'){
```

```

        val = ch_in - '0';
        goto out;
    }
    if(ch_in == '('){
        val = expr();
        getchar();        /* تخطي قوس الإغلاق "()" */
        goto out;
    }
    printf("error: primary read %d\n", ch_in);
    exit(EXIT_FAILURE);

out:
    return(val);
}

```

[مثال 8]

4.4 مفهوم النطاق Scope والربط Linkage على مستوى الدوال

على الرغم من تفادينا لموضوعي النطاق Scope والربط Linkage في أمثلتنا البسيطة سابقاً، إلا أن الوقت قد حان لشرح هذين المفهومين وأثرهما على قابلية الوصول للكائنات المختلفة في برنامج سي C، ولكن لم علينا الاكتراث بذلك على أي حال؟ لأن البرامج العملية التي نستخدمها تُبنى من عدّة ملفات ومكتبات، وبالتالي من المهم للدوال في ملف ما أن تكون قادرةً على الإشارة إلى دوال، أو كائنات في ملفات أو مكتبات أخرى، وهناك عدّة قوانين ومفاهيم تجعل من ذلك ممكناً.

عُد لاحقاً إلى هذه الجزئية إن كنت جديداً على لغة سي، لأن هناك بعض المفاهيم الأهم التي يجب عليك معرفتها أولاً.

4.4.1 الربط Linkage

هناك نوعان أساسيان من الكائنات في لغة سي C، هما: الكائنات الخارجية والكائنات الداخلية، والفرق بين الاثنين مُعتمداً على الدوال؛ إذ أن أيّ شيء يُصرّح عنه خارج الدالة فهو خارجي؛ وأي شيء يُصرّح عنه داخل الدالة بما فيه مُعاملات الدالة فهو داخلي. بما أننا لا نستطيع تعريف دالة ما داخل دالة أخرى، فهذا يعني أن الدوال هي كائنات خارجية دائماً، وإذا نظرنا إلى بنية برنامج سي على المستوى الخارجي، سنلاحظ أنه يمثّل مجموعةً من الكائنات الخارجية **External objects**.

يمكن للكائنات الخارجية فقط أن تُشارك في هذا الاتصال عبر الملفات والمكتبات، وتُعرف قابلية الوصول للكائنات هذه من ملف إلى آخر أو ضمن الملف نفسه وفقاً للمعيار باسم **الربط Linkage**، وهناك ثلاثة أنواع

للربط، هي: **الربط الخارجي External linkage** و**الربط الداخلي Internal Linkage** و**عديم الربط No linkage**. يكون أي شيء داخلي في الدالة سواءً كان وسطاء الدالة أو متغيراتها عديم الربط **دائمًا** ويمكن الوصول إليه من داخل الدالة فقط، ويمكنك التصريح عن الشيء الذي تريده داخل الدالة مسبقًا بالكلمة المفتاحية `extern` لتجاوز هذا القيد، وهذا سيدل على أن الكائن ليس داخليًا، وليس عليك القلق بهذا الشأن في الوقت الحالي.

تكون الكائنات ذات الربط الخارجي موجودةً على المستوى الخارجي لبنية البرنامج، وهذا هو نوع الربط الافتراضي للدوال ولأي شيء آخر يُصرَّح عنه خارج الدوال، و**تُشير جميع الأسماء المماثلة لاسم الكائن ذي الربط الخارجي إلى الكائن نفسه**. ستحصل على سلوك غير محدد من البرنامج، إذا صرحت عن كائن بنفس الاسم مرتين أو أكثر بربط خارجي وبأنواع غير متوافقة. المثال الذي يأتي إلى بالنا مباشرةً بخصوص الربط الخارجي هو الدالة `printf` والمُصرَّح عنها في ملف الترويسة `<stdio.h>` على النحو التالي:

```
int printf(const char *, ...);
```

يمكننا فورًا بالنظر إلى التصريح السابق معرفة أن الدالة `printf` تُعيد قيمةً من نوع `int` باستخدام النموذج الأولي الموضح، كما نعرف أن للدالة ربطًا خارجيًا لأنها كائن خارجي (تذكّر، كل دالة هي كائن خارجي افتراضيًا)، وبالتالي فنحن نقصد هذه الدالة تحديدًا عندما نكتب الاسم `printf` في أي مكان ضمن البرنامج، إذا استخدمنا الربط الخارجي.

ستحتاج في بعض الأحيان لطريقة تمكّنك من التصريح عن الدوال والكائنات الأخرى في ملفٍ واحد بحيث تسمح لهم بالإشارة إلى بعضهم البعض **دون** القدرة على الوصول إلى تلك الكائنات والدوال من خارج الملف. نحتاج هذه الطريقة غالبًا في الوحدات `modules` التي تدعم دوال المكتبات، إذ من الأفضل في هذه الحالة إخفاء بعض الكائنات التي تجعل استخدام هذه الدالة ضمن المكتبة ممكنًا، فهي ليست ضرورية المعرفة لمستخدم المكتبة وستكون سببًا للإزعاج لا غير، ونستطيع تحقيق ذلك الأمر عن طريق استخدام **الربط الداخلي**.

تُشير الأسماء ذات الربط الداخلي للكائن ذاته ضمن ملف الشيفرة المصدرية الواحد، ويمكنك التصريح عن كائن ذي ربط داخلي عن طريق بدء التصريح بالكلمة المفتاحية `"static"` التي ستغيّر ربط الكائن من ربط خارجي (افتراضي) إلى ربط داخلي، كما يمكنك التصريح عن كائنات داخلية باستخدام `"static"` بهدف استخدام آخر ولكننا لن نتطرق لهذا الاستخدام حاليًا.

من المُربك استخدام المصطلحين "داخلي" و"خارجي" لوصف نوع الربط ونوع الكائن، ولكن يعود ذلك لأسباب تاريخية، إذ سيتذكر مبرمجو لغة سي القدماء كون الاستخدامين متساويين، وأن استخدام أحدهما يتضمن تحقّق الآخر، ولكن تغيّر ذلك الأمر للأسف حديثًا وأصبح معنى الاستخدامين مختلف، ولنلخّص الفرق فيما يلي:

الجدول 12: الربط وقابلية الوصول

نوع الربط	نوع الكائن	قابلية الوصول
خارجي	خارجي	يمكن الوصول إليه من أي مكان ضمن البرنامج
داخلي	خارجي	يمكن الوصول إليه عبر ملف واحد فقط
لا يوجد ربط	داخلي	محلي لدالة واحدة

أخيرًا وقبل أن ننتقل إلى مثالٍ آخر، علينا أن نعرف أن لجميع الكائنات ذات الربط الخارجي تعريفٌ واحدٌ فقط، مع أنه بالإمكان أن يوجد عدة تصريحات متوافقة حسب حاجتك. إليك المثال:

```

/* الملف الأول */

int i; /* تعريف*/
main () {
    void f_in_other_place (void); /* تصريح*/
    i = 0
}
/* نهاية الملف الأول */

/* بداية الملف الثاني */

extern int i; /* تصريح*/
void f_in_other_place (void){ /* تعريف*/
    i++;
}
/* نهاية الملف الثاني */

```

[مثال 9]

على الرغم من تعقيد القوانين الكاملة التي تحدد الفرق بين التعريف والتصريح إلا أن هناك طريقةً بسيطةً وسهلة، هي:

- التصريح عن الدالة دون تضمين متن الدالة هو تصريحٌ لا غير.
- التصريح عن الدالة مرفقًا بمتن الدالة هو تعريف.
- عمومًا، التصريح عن كائن على المستوى الخارجي للبرنامج (مثل المتغير `i` في المثال السابق) هو تعريف، إلا إذا سبق بالكلمة المفتاحية `extern` وعندها يصبح تصريحًا فقط.

وستتكمّل لاحقاً عن التعريف والتصريح بصورةٍ أعمق لا تبقي مجالاً للشك.

من الواضح في مثالنا السابق أنه من السهل الوصول من أي ملف للكائنات المعرفة في ملفات أخرى باستخدام أسمائها فقط، وسيدلك المثال على كيفية بناء برامج ذات ملفات ودوال ومتغيرات متعددة سواءً كانت مُعرّفةً أو مصرّحٌ عنها وفق ما يناسب كل حالة.

إليك مثالاً آخر يوضح استخدام "static" لتقييد قابلية الوصول إلى الدوال والأشياء الأخرى.

```

/* مثال عن وحدة في مكتبة */
/* هي الدالة الوحيدة المرئية على المستوى الخارجي callable الدالة */
static buf [100];
static length;
static void fillup(void);

int
callable (){
    if (length ==0){
        fillup ();
    }
    return (buf [length--]);
}

static void
fillup (void){
    while (length <100){
        buf [length++] = 0;
    }
}

```

[مثال 10]

يمكن للمستخدم -بفرض استخدام المثال السابق وحدةً مستقلة- إعادة استخدام الأسماء length و buf و fillup بأمان دون أي تأثيرات جانبية أو أخطاء غير متوقعة، ونستثنى من ذلك الاسم callable، إذ إنه قابل الوصول خارج الملف (الوحدة المستقلة).

تكون قيمة الكائن الخارجي الذي يمتلك مُهيئاً initializer واحدًا مساوية للصفر قبل بدء البرنامج (لم نتكلم عن أي مُهيئات عدا الدوال حتى الآن)، وتعتمد الكثير من البرامج على ذلك، بما فيها المثال السابق لقيمة length الابتدائية.

4.4.2 تأثير النطاق

لا تقتصر عملية مشاركة الأسماء وقابلية الوصول إليها على الربط ببساطة، فالربط يسمح لك باستخدام عدة أسماء والوصول إليها سويًا ضمن البرنامج أو ضمن الملف، ولكن النطاق Scope يحدد رؤية الأسماء، وقواعد النطاق لحسن الحظ مستقلة عن مبدأ الربط، لذا ليس عليك حفظ أي قواعد مركبة بين المفهومين.

تزيد الكلمة المفتاحية `extern` من تعقيد البرنامج، فعلى الرغم من وضوح استخدامها وتأثيرها إلا أنها تغيّر من بنية برنامج لغة سي الكُتلية التي اعتدنا عليها، وسنناقش المشاكل الناتجة عن استخدامها الخاطئ وغير المسؤول لاحقًا، وقد نظرنا إلى استخدامها مُسبقًا للتأكد من أن التصريح لشيء ما ضمن المستوى الخارجي للبرنامج هو تصريحٌ وليس تعريف.

يمكنك تجاوز الكلمة المفتاحية "extern" عن طريق مُهَيِّئ للكائن.

التصريح عن أي كائن بيانات (ليس بدالة) ضمن المستوى الخارجي للبرنامج هو تعريفٌ، إلا إذا سبق التصريح الكلمة المفتاحية "extern"، راجع المثال 9.4 من أجل ملاحظة هذه النقطة عمليًا.

تحتوي تصريحات الدوال الكلمة المفتاحية "extern" ضمنيًا سواءً كانت مكتوبةً أم لا، والطريقتان التاليتان للتصريح عن الدالة `some_function` متكافئتان، وتُعدان تصريحًا وليس تعريفًا:

```
void some_function(void);

extern void some_function(void);
```

الشيء الوحيد الذي يفصل التصريحات السابقة عن كونها تعريفات هو متن الدالة، الذي يُعد مُهَيِّئًا للدالة، لذلك عند إضافة المُهَيِّئ يتحول التصريح إلى تعريف، لا توجد أي مشكلة بخصوص ذلك.

لكن ما الذي يحدث في المثال التالي؟

```
void some_function(void){
    int i_var;
    extern float e_f_var;
}

void another_func(void){
    int i;
    i = e_f_var;    /* مشكلة تتعلق بالنطاق */
}
```

ما الهدف من المثال السابق؟ من المفيد في بعض الأحيان أن تستخدم كائنًا خارجيًا ضمن دالة ما، وإن اتبعت الطريقة الاعتيادية بالتصريح عن الكائن في بداية ملف الشيفرة المصدرية، فسيكون صعبًا على القارئ معرفة أي من الدوال تستخدم ذلك الكائن؛ بدلًا من ذلك، يمكنك تقييد نطاق الكائن وقابلية الوصول إليه في المكان الذي تريد الوصول إليه، مما سيسهل على القارئ معرفة أن الاسم سيستخدم فقط في هذا المكان المحدود وليس على كامل نطاق ملف الشيفرة المصدرية. يجدر الانتباه إلى أن معظم طرق إدارة الوسطاء في هذه الحالة عملية صعبةٌ بعض الشيء.

سنناقش المزيد من القواعد عن الطريقة الأمثل لإنشاء برنامج ذو ملفات متعددة، وما الممكن حدوثه عند المزج بين التصريحات الداخلية والخارجية والكلمات المفتاحية "extern" و "static". لن تكون عملية قراءة هذه القواعد ممتعةً، لكنها ستكون إجابةً لأسئلة من نوع "ماذا لو؟".

4.4.3 الكائنات الداخلية الساكنة

يمكنك التصريح عن كائنات داخلية على أنها كائنات داخلية ساكنة باستخدام الكلمة المفتاحية "static"، وتكتسب المتغيرات الداخلية بعضًا من الخصائص باستخدامها هذه الكلمة المفتاحية ألا وهي:

- تُهيأ قيمتها إلى الصفر عند بداية البرنامج.
- تحافظ على قيمتها من بداية التعليمة التي تضم تصريحها إلى نهايتها.
- يوجد نسخة واحدة من كل متغير داخلي ساكن تشاركه الاستدعاءات التعاودية للدوال التي تحوي هذه المتغيرات.

يمكن أن تُستخدم المتغيرات الداخلية الساكنة لعدة أمور، أحدها هو عدّ مرات استدعاء دالة ما، إذ تحافظ المتغيرات الداخلية الساكنة على قيمتها بعد الخروج من الدالة بعكس المتغيرات الداخلية الاعتيادية. إليك دالة تُعيد عددًا بين 0 و15، ولكنها تُبقي عدد المرات التي استُدعيت بها:

```
int
small_val (void) {
    static unsigned count;
    count ++;
    return (count % 16);
}
```

[مثال 11]

يمكن أن نستخدم هذه الطريقة للكشف عن الاستدعاءات التعاودية مفردة الحدوث:

```

void
r_func (void){
    static int depth;
    depth++;
    if (depth > 200) {
        printf ("excessive recursion\n");
        exit (1);
    }
    else {
        /* نفذ بعض الأوامر الاعتيادية. قد تتسبب النتائج باستدعاء آخر .r_func()
        لدالة التعاود */
        x_func();
    }
    depth--;
}

```

[مثال 12]

4.5 الخاتمة

يمكن تحديد قابلية الوصول إلى المتغيرات ورؤيتها عبر برنامج سي C باستخدام التصريح المناسب، سواءً كان الوصول لكافة ملفات البرنامج أو لملفٍ واحد أو دالةٍ واحدة.

إليك احتمالات استخدام الكلمات المفتاحية مع أنواع التصريح والربط الناتج عن كل حالة:

الجدول 13: ملخص الربط

التصريح	الكلمة المفتاحية	نوع الربط الناتج	قابلية الوصول	ملاحظة
خارجي	لا يوجد	خارجي	ضمن كامل البرنامج	2
خارجي	"extern"	خارجي	ضمن كامل البرنامج	2
خارجي	"static"	داخلي	ضمن ملف واحد	2
داخلي	لا يوجد	لا يوجد	ضمن دالة واحدة	
داخلي	"extern"	خارجي	ضمن كامل البرنامج	1
داخلي	"static"	لا يوجد	ضمن دالة واحدة	2

1. لا تنفي قابلية الوصول للتصريحات الداخلية المُسبقة بالكلمة المفتاحية "extern" ضمن كامل البرنامج أهمية الانتباه إلى نطاق الاسم المُصرَّح عنه.

2. تُهيأ الكائنات الخارجية (أو الداخلية باستخدام "static") مرةً واحدةً فقط عند بداية البرنامج، وتُهيأ إلى قيمة الصفر ضمنياً إذا لم يكن هناك أي مُهيئ لها.

هناك بعض القواعد الذهبية التي تخص استخدام الدوال، ويجب التشديد على ذكرها:

- يجب وجود تصريح أو تعريف ضمن النطاق لاستخدام دالة تُعيد قيمةً مختلفةً عن النوع int.
- لا تُعاد أي قيمة من دالة ما خارج متنها، إلا إذا كانت من نوع void.
- التصريح عن أنواع وسطاء الدالة ليس إلزامياً، ولكنه محببٌ جداً.

يمكن كتابة الدوال التي تأخذ عدداً متغيراً من الوسطاء بطرق قابلة للنقل، وسنناقش هذه الطرق لاحقاً.

الدوال هي الحجر الأساس للغة سي، وقد كانت النماذج الأولية أكثر التغييرات الواضحة بقدوم سي المعيارية، ونال هذا التغيير إعجاباً واسعاً من مجتمع مستخدمي اللغة وساعد في تحسين قابلية قراءة برامج لغة سي C، كما فتح المجال لزيادة كفاءة البرنامج بواسطة المصروفات وهو ما لم يكن متاحاً في سي القديمة.

قد يُفاجئ استخدام استدعاء الوسيط بقيمته بعض الناس الذين اعتادوا على لغات أخرى تستخدم طُرُقاً مغايرة، ولكن لغة سي تفضل أن تسلك الطريق "الأكثر أماناً" في معظم الحالات.

حاولت لغة سي المعيارية التخلُّص من بعض النقاط الغامضة بخصوص النطاق ومعاني التصريحات، مما ولد بدوره منطقةً غامضة، ولكنها لم تسبب الكثير من المشكلات بالتطبيق عملياً.

من المهم على المبتدئ أن يفهم ويتعلم كل ما ذُكر في هذا الفصل، ويمكن استثناء قواعد الربط، إذ من الممكن تأجيلها والرجوع إليها لاحقاً.

4.6 تمارين

ستواجه بعض المصاعب بخصوص تمرين 2 و3 و4 إذا تخطيت القسم الذي يتكلم عن الربط، ولك أن تختار العودة إلى هذه التمارين لاحقاً بعد قراءة القسم المذكور.

اكتب دالةً مع التصريحات المناسبة وفق المهام التالية:

1. دالة تدعى "abs_val" تُعيد قيمةً من النوع int وتأخذ وسيطاً من نوع int، وتكون القيمة المُعادَة موافقة للقيمة المطلقة للوسيط، وذلك بنفي القيمة إذا كانت سالبة.

2. دالة تدعى "output" تأخذ محرّفاً واحداً مثل وسيط لها وتعيده خرجاً للبرنامج باستخدام الدالة "putchar"، وستتذكر هذه الدالة رقم السطر والعمود الحاليين في جهاز الإخراج، ويُضمّن للقيم أن تكون أحرف أو أرقام أو مسافات فارغة أو محارف أسطر جديدة فقط.

3. اكتب برنامجًا لفحص الدالة "output" في حالة كانت في ملف مستقل عن ملف الدوال التي تستخدمها، وسيكون هنا دالتين، هما: "current_column" و "current_line" في ملف دالة "output" للحصول على قيمة السطر والعمود الحالي. تأكد أن قيم الدالتين (عدّاد السطر والعمود) ممكنة الوصول من داخل الملف الذي يحويهما فقط.

4. اكتب وافحص دالةً تعاوديةً لطباعة لائحة من الأرقام من 100 إلى 1، بحيث تكون الزيادة بمقدار واحد على قيمة متغير ساكن عند الدخول للدالة، وتستدعي الدالة نفسها مجددًا إذا كانت قيمة المتغير أقل من 100، ومن ثم تطبع قيمة المتغير وتنقصه بمقدار واحد وتعيد قيمته. تحقق من عمل هذه الدالة.

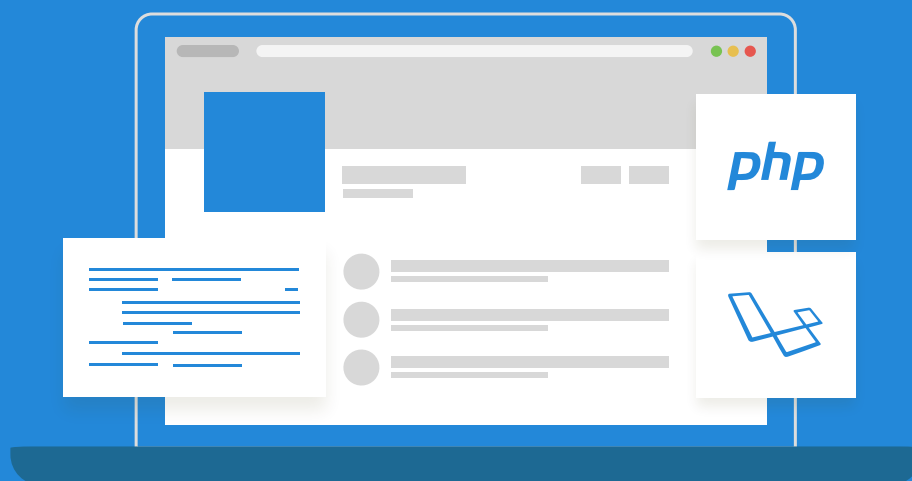
5. اكتب دالتين لحساب جيب Sin وجيب التمام Cosin لقيمة ما، واختر أنواعًا مناسبة للقيم (كل من قيم الوسطاء والقيمة المُعادَة)، وتوضح السلسلة أدناه طريقةً لتقريب الإجابة. يجب أن تعيد الدالة النتيجة عندما يكون الفرق بين قيمة الحد النهائي والقيمة الحالية للدالة هو 0.000001.

```
sin x = x - pow(x,3)/fact(3) + pow(x,5)/fact(5)...
cos x = 1 - pow(x,2)/fact(2) + pow(x,4)/fact(4)...
```

لاحظ أن الإشارة قبل كل حد متناوبة على النحو ... - + - + - + ...، وتُعيد الدالة $\text{pow}(x, n)$ قيمة x بأس n ، وتُعيد الدالة $\text{fact}(n)$ قيمة n عاملي (أي $1 \times 2 \times 3 \times \dots \times n$).

عليك كتابة هذه الدوال بنفسك، ومن ثم التأكد من عملها بفحص النتيجة النهائية.

دورة تطوير تطبيقات الويب باستخدام لغة PHP



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



5. المصفوفات Arrays والمؤشرات Pointers

5.1 تمهيد الفصل

5.1.1 ما أهمية هذا الفصل؟

استعرضنا سابقًا أنواع البيانات الحسابية والعوامل في لغة سي C، وقد كانت لافتةً للانتباه ولكنها غير مثيرة للاهتمام، إذ وضح لنا الجزء السابق رونق لغة سي المميّز وما الذي يجب علينا توقعه بالمضي قدمًا من هذه اللغة، ولكن تعدّ هذه الأشياء بمثابة التوابل للطبق الرئيسي وليس الطبق بحد ذاته. ينظر بعض مستخدمي اللغة للدوال functions والأجزاء الأخرى التي سنغطّيها في هذا الفصل بكونها أساسًا للغة سي.

سيتسبب هذا الجزء للقارئ الجديد بالمشكلات الأكبر، فبينما يألّف المبتدئون في لغة سي استخدام العمليات الحسابية والدوال والمصفوفات، تظهر لهم المشكلات عند استخدام الأنواع المهيكلة Structured types، مثل الهياكل Structures والاتحادات Unions، إضافةً إلى المؤشرات التي تتميز بها لغة سي.

لا يمكنك تجاهل استخدام المؤشرات ببساطة، إذ إنها مستخدمةٌ في كل مكان، وتؤثر على اللغة ككل وهي الميزة الوحيدة الملاحظة في جميع برامج سي مهما بلغت بساطتها. إذا اعتقدت أن هذا الجزء من الكتاب يمكن تخطيه لأنه صعب ولا يبدو مهمًا للغاية فأنت مخطئ، إذ أن معظم الأمثلة التي استعرضناها ضمن الكتاب تستخدم المؤشرات (بصورة غير واضحة في بعض الأحيان)، لذا فإن فهم عمل المؤشرات ضروريّ لإتقان اللغة، تقبّل هذا الأمر وابدأ بتعلمه.

علينا النظر أولًا إلى المصفوفات إذا أردنا تقديم مفهوم المؤشرات على نحوٍ سهل فهمه، إذ يتداخل المفهومين السابقين عميقًا ببعضهما في لغة C ومن الصعب الفصل بينهما. سنفترض معرفتك المسبقة بمفهوم المصفوفات وسنتطرق إليها بصورة بسيطة بقصد توضيح استخدام المؤشرات كما سنرى لاحقًا.

5.1.2 تأثير لغة سي المعيارية

كان للمعيار الجديد تأثير ضئيل على محتويات هذا الفصل، ومعظم محتوياته ستكون مماثلةً فيما لو كنا نتكلم عن لغة سي القديمة، وذلك لعدم وجود أي اختلاف عن الإصدار القديم من اللغة بهذا الشأن، ولم يكن هناك أي جدوى من التعديل على ما يعمل على نحو جيد، ولعلّ معرفة هذا ستطمئن مستخدمي لغة سي القديمة ولجنة معيار سي، الذين لم يجدوا أي خطأ بخصوص هذا الجزء من اللغة.

بالرغم من ذلك، تضيف **Qualified types** الأنواع المؤهلة بعضًا من التعقيد على هذا الفصل، إذ كنا قد وضحنا سابقًا القوانين المختلفة التي تصف سلوك العوامل العلائقية relational operators والعوامل الحسابية arithmetic operators عند استخدامها مع المؤشرات بفقرة نصية طويلة، ولكن هذه القوانين لم تتغير كثيرًا في المعيار الجديد، ولم نولّ الكثير من الاهتمام لهذه القوانين في أمثلتنا السابقة، إذ حاولنا تقديمها بصورة بسيطة في بادئ الأمر وشرحها عندما تستدعي الحاجة فقط.

5.2 المصفوفات Arrays

تستخدم لغة سي المصفوفات Arrays مثل سائر اللغات الأخرى لتمثيل مجموعة من متغيرات ذات خصائص متماثلة، إذ يكون لهذه المجموعة اسمًا واحدًا وتُحدّد عناصرها عن طريق دليل Index. إليك مثالًا للتصريح عن مصفوفةٍ ما:

```
double ar[100];
```

في هذا المثال اسم المصفوفة هو ar ويمكن الوصول لعناصر المصفوفة عن طريق دليل كلٍّ منها كما يلي: ar[0] وصولًا إلى ar[99] لا غير، كما يوضح الشكل 2:

ar[0]	ar[1]	...	ar[99]
-------	-------	-----	--------

الشكل 2: مصفوفة ذات 100 عنصر

يمثل كلّ عنصر من عناصر المصفوفة المئة متغيّرًا منفصلًا من نوع double، ويُرمز لكل عنصر من أي مصفوفة في لغة سي بدءًا من الدليل 0 وصولًا إلى الدليل الذي يساوي حجم المصفوفة المُصرّح عنه ناقص واحد، ويعدّ البدء بالترقيم من 0 مفاجئًا لبعض المبتدئين فركّز على هذه النقطة.

عليك أن تنتبه أيضًا إلى أن المصفوفات لا تقبل حجمًا متغيّرًا عند التصريح عنها، إذ يجب أن يكون الرقم تعبيرًا ثابتًا يمكن معرفة قيمته الثابتة وقت تصريف البرنامج compile time وليس وقت التشغيل run time. يوضح المثال التالي طريقة خاطئة للتصريح عن مصفوفة باستخدام الوسيط x:

```
f(int x){
```

```
char var_sized_array[x];          /* هذه الطريقة ممنوعة */
}
```

وهذا ممنوع لأن قيمة x غير معروفة عند تعريف البرنامج، فهي قيمة تُعرف عند تشغيل البرنامج وليس عند تعريفه.

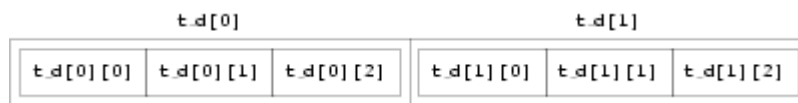
من الأسهل لو كان مسموحًا استخدام المتغيرات لتحديد حجم المصفوفات وبالأخص البعد الأول لها، ولكن هذا الأمر لم تسمح به سي القديمة أو سي المعيارية، إلا أن هناك مصرّف قديم جدًا للغة سي اعتاد السماح بهذا.

5.2.1 المصفوفات متعددة الأبعاد

يمكن التصريح عن المصفوفات متعددة الأبعاد Multidimensional arrays على النحو التالي:

```
int three_dee[5][4][2];
int t_d[2][3]
```

تُستخدم الأقواس المعقوفة بعد كلٍّ من المصفوفات السابقة، وإذا نظرت إلى جدول الأسبقية في فصل [العوامل في لغة سي C](#)، فستلاحظ أن قراءة القوسين [] تكون من اليسار إلى اليمين وبذلك تكون نتيجة التصريح مصفوفةً تحتوي على خمس عناصر باسم `three_dee`، ويحتوي كل عنصر من عناصر هذه المصفوفة بدوره على مصفوفة بحجم أربعة عناصر وكل عنصر من هذه المصفوفة الأخيرة يحتوي على مصفوفة من عنصرين، وجميع العناصر من نوع `int`، وبهذه الحالة فنحن صرحنا عن مصفوفة مصفوفات، ويوضح الشكل 3 مثالاً على مصفوفة ثنائية البعد باسم `t_d` في مثال التصريح.



الشكل 3: هيكل مصفوفة ثنائية البعد

ستلاحظ في الشكل السابق أن `t_d[0]` عنصرٌ واحدٌ متبوعٌ بعنصر `t_d[1]` دون أي فواصل، وكلا العنصرين يمثلان مصفوفةً بحدّ ذاتهما بسعة ثلاث أعداد صحيحة. يأتي العنصر `t_d[1][0]` مباشرةً بعد العنصر `t_d[0][2]`، ومن الممكن الوصول إلى `t_d[1][0]` بالاستفادة من عدم وجود أي طريقة للتحقق من حدود المصفوفة باستخدام التعبير `t_d[0][3]` إلا أن هذا غير محبّب أبداً، لأن النتائج ستكون غير متوقعة إذا تغيّرت تفاصيل التصريح عن المصفوفة `t_d`.

حسناً، لكن هل هذا الأمر يؤثر على سلوك البرنامج عملياً؟ في الحقيقة لا، إلا أنه من الجدير بالذكر أن موقع تخزين العنصر الواقع على أقصى اليمين ضمن المصفوفة "يتغير بسرعة"، ويؤثر ذلك على المصفوفات عند استخدام المؤشرات معها، ولكن يمكن استخدامها بشكلها الطبيعي عدا عن تلك الحالة، مثل التعابير التالية:

```
three_dee[1][3][1] = 0;
three_dee[4][3][1] += 2;
```

التعبير الأخير مثير للاهتمام لسببين، أولهما أنه يصل إلى قيمة العنصر الأخير من المصفوفة والمصرّح أنها بحجم [2][4][5]، والدليل الذي نستطيع استخدامه هو أقل بواحد دائماً من العدد الذي صرّحنا عنه، أما ثانيًا، فنلاحظ أهمية وسهولة استخدام عامل الإسناد المُركّب في هذه الحالة. يفضّل مبرمجو لغة سي المتمرسون هذه الطريقة المختصرة، إليك كيف سيبدو الأمر لو كان التعبير مكتوبًا بلغة أخرى لا تسمح باستخدام هذا العامل:

```
three_dee[4][3][1] = three_dee[4][3][1] + 2;
```

ففي هذه الحالة يجب أن يتحقق القارئ أن العنصر على اليمين عامل الإسناد هو ذات العنصر على يسار عامل الإسناد، كما أن الطريقة المُختصرة أفضل عند تصريفها، إذ يُحسب دليل العنصر وقيّمته مرةً واحدة، مما ينتج شيفرةً برمجيّةً أقصر وأسرع. قد ينتبه بعض المصرّفات طبعًا إلى أن العنصرين على طرفي عامل الإسناد متساويين ولن تلجأ للوصول للقيمة مرتين، ولكن هذه الحالة لا تنطبق على جميع المصرّفات، وهناك العديد من الحالات أيضًا التي لا تستطيع فيها المصرّفات الذكية هذه باختصار الخطوات.

على الرغم من تقديم لغة سي دعمًا للمصفوفات متعددة الأبعاد، إلا أنه من النادر أن تجدها مُستخدمةً عمليًا، إذ تُستخدم المصفوفات أحادية البعد أكثر في معظم البرامج، وأبسط هذه الأسباب هو أن السلسلة النصية String تُمثّل بمصفوفة أحادية البعد، وقد تلاحظ استخدام المصفوفات ثنائية البعد في بعض الحالات، ولكن استخدام المصفوفات ذات أبعاد أكثر من ذلك نادرة الحدوث، وذلك لكون المصفوفة هيكل بيانات غير مرّن بالتعامل، كما أن سهولة عملية إنشاء ومعالجة هياكل البيانات وأنواعها في لغة سي تعني إمكانية استبدال المصفوفات في معظم البرامج متقدمة المستوى، وسننظر إلى هذه الطرق عندما ننظر إلى المؤشرات.

5.3 المؤشرات Pointers

يشابه استخدام المؤشرات Pointers في لغة سي عملية تعلّم قيادة الدراجة الهوائية، فعندما تصل إلى النقطة التي تعتقد أنك لن تتعلمها أبدًا، تبدأ بإتقانها، وبعد أن تتعلمها سيكون من الصعب نسيانها. لا يوجد هناك أي شيء مميّز بخصوص المؤشرات، ونعتقد أن معظم القراء يعرفون عنها مسبقًا، وفي الحقيقة، واحدة من ميزات لغة سي هي اعتمادها الكبير على استخدام المؤشرات مقارنةً باللغات الأخرى، إضافةً إلى الأشياء الأخرى الممكن إنجازها بواسطة المؤشرات بسهولة ودون قيود إلى حدٍّ ما.

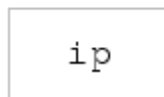
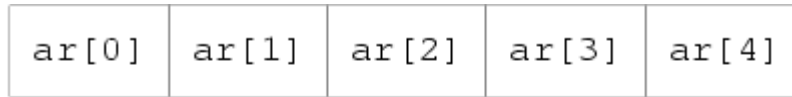
5.3.1 التصريح عن المؤشرات

ينبغي التصريح عن المؤشرات قبل استخدامها بصورةٍ مماثلة لأي متغير آخر تعاملنا معه مسبقًا. يتشابه التصريح عن المؤشر مع أي تصريح آخر، ولكن هناك نقطة مهمّة، إذ تدلّ الكلمة المفتاحية عند التصريح عن المؤشر في البداية، مثل `int` أو `char` وغيرها عن نوع المتغير الذي سيشير المؤشر إليه، وليس نوع المؤشر

بذات نفسه، ويشير المؤشر على قيمة واحدة كل مرة من ذلك النوع وليس جميع القيم من النوع ذاته. إليك مثالاً يوضح التصريح عن مصفوفة ومؤشر:

```
int ar[5], *ip;
```

يصح لدينا بعد التصريح مصفوفة ومؤشر، كما يوضح الشكل 4:



الشكل 4: مصفوفة ومؤشر

يوضح الرمز * الموجود أمام ip أن هذا مؤشر، وليس متغيراً اعتيادياً، وهو مؤشر من النوع pointer to int، أي يشير إلى قيمة من نوع int فقط، لكن لم تُسند له قيمة أولية بعد، ولا يمكننا استخدامه في هذه الحالة قبل أن نجعله يُشير على قيمة ما. لاحظ أنه لا يمكنك تعيين قيمة من نوع int فوراً، لأن القيم الصحيحة لها النوع int ونحن نريد هنا قيمة من نوع "مؤشر إلى نوع صحيح pointer to int". لكن، ما الذي سيشير إليه ip في هذه الحالة إذا كانت التعليمة التالية صحيحة؟

```
ip = 6;
```

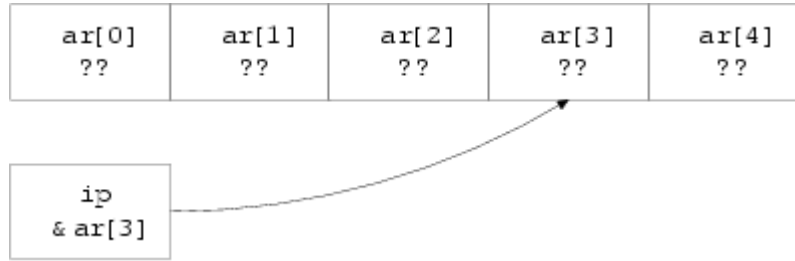
قد تختلف الإجابة هنا، ولا يوجد إجابة واحدة عما قد يشير إليه ip، ولكن خلاصة الأمر أن هذا النوع من الإسناد غير صحيح في لغة سي.

إليك الطريقة الصحيحة لإسناد قيمة أولية لمؤشر ما:

```
int ar[5], *ip;
ip = &ar[3];
```

يُشير المؤشر في هذا المثال إلى عنصر من المصفوفة ar بدليل 3، أي العنصر الرابع من المصفوفة.

يمكنك إسناد القيم إلى المؤشرات كما في أي متغير اعتدت عليه، ولكن تكمن الأهمية في نوع هذه القيمة وما الذي تعنيه. يدل الشكل 5 على قيم المتغيرات الموجودة بعد التصريح، ويدل ?? على كون المتغير غير مُسند لقيمة أولية أي غير مهياً.



الشكل 5: مصفوفة ومؤشر مهياً

نلاحظ أن قيمة المتغير `ip` مساوية لقيمة التعبير `&ar[3]`. ويشير السهم إلى أن المؤشر `ip` يشير إلى المتغير `ar[3]`.

لكن ما الذي يعنيه العامل الأحادي `&`؟ يُشار إلى هذا العامل بكونه عامل "عنوان المتغير"، إذ أن المؤشرات تخزن عنوان المتغير الذي تُؤسّر عليه في معظم الأنظمة. ربّما ستواجه صعوبةً بخصوص هذا الأمر إذا كنت تفهم ما نعني هنا بالعنوان مقارنةً بالأشخاص الذين لا يفهمون هذا الأمر، إذ أن التفكير بالمؤشرات كونها عناويناً يؤدي إلى كثيرٍ من المشاكل في الفهم.

قد تكون عملية التلاعب بعناوين معالج "أ" مستحيلاً على متحكّم آلة غسيل من نوع "ب" يستخدم عناويناً بسعة 17-بت عندما تكون في طور الغسيل، ويقلب ترتيب البتات الزوجية والفردية عندما ينفذ من مسحوق الغسيل. من المستبعد لأي أحد أن يستخدم لغة سي بمعمارية مشابهة لمثالنا السابق، ولكن هناك بعض الحالات الأخرى والأقل شدةً التي قد **يمكن** تشغيل لغة سي على معماريتها.

لكننا سنتابع استخدام الكلمة "عنوان المتغير"، لأن استخدام مصطلح أو كلمة مغايرة لذلك سيتسبب بمزيدٍ من المشكلات.

يعيد تطبيق العامل `&` لمعامل ما مؤسّراً لهذا المعامل:

```
int i;
float f;
/* مؤشر إلى عدد صحيح */
/* مؤشر إلى عدد حقيقي */
```

وسيشير المؤشر في كل حالة إلى الكائن الذي يوافق اسمه اسم المستخدم في التعبير.

المؤشر مفيدٌ فقط في حال وجود طريقة للوصول إلى الشيء الذي يشير إليه، وتستخدم لغة سي العامل الأحادي `*` لتحقيق ذلك؛ فإذا كان `p` من نوع "مؤشر إلى نوع ما `pointer to something`"، فيشير التعبير `*p` إلى الشيء الذي يشير إليه ذلك المؤشر. على سبيل المثال، تتبع مايلي للوصول إلى المتغير `x` باستخدام المؤشر `p`:

```

#include <stdio.h>
#include <stdlib.h>
main(){
    int x, *p;

    p = &x;          // تهيئة المؤشر
    *p = 0;          // إسناد القيمة 0 إلى المتغير x
    printf("x is %d\n", x);
    printf("**p is %d\n", *p);

    *p += 1;         /* زيادة القيمة التي يشير إليها المؤشر */
    printf("x is %d\n", x);

    (*p)++;         /* زيادة القيمة التي يشير إليها المؤشر */
    printf("x is %d\n", x);

    exit(EXIT_SUCCESS);
}

```

[مثال 1]

من الجدير بالذكر معرفة أن استخدام التركيب المؤلف من `&` و `*` بالشكل `*&` يلغي تأثير كلٍّ منهما، لأن `&` تعيد عنوان الكائن أي قيمة مؤشّره، و `*` تعني "القيمة التي يشير إليها المؤشر". لكن انتبه، فليس لبعض الأشياء مثل الثوابت أي عنوان، وبذلك لا يمكن تطبيق العامل `&` عليها، والتعبير `&1.5` ليس مؤشراً بل تعبيراً خاطئاً. من المثير للانتباه أيضاً إلى أن لغة سي من اللغات القليلة التي تسمح بوجود تعبير على الجانب الأيسر من عامل الإسناد. انظر مجدداً إلى المثال؛ إذ نصادف التعبير `*p` مرتين، ومن ثم التعليمة `++(*p)` المثيرة للاهتمام، وستثير هذه التساؤلات من معظم المبتدئين حتى لو استطعت فهم أن التعليمة `*p = 0` تسند القيمة 0 إلى المتغير المُشار إليه بواسطة المؤشر `p`، وأن التعليمة `*p += 1` تضيف واحداً إلى المتغير المُشار إليه بالمؤشر `p`، فاستخدام العامل `++` مع `*p` يبدو صعب الفهم قليلاً.

تستحق أسبقية التعبير `++(*p)` النظر إليها بدقة، وسناقش مزيداً من التفاصيل بهذا الخصوص، ولكن دعونا نركّز عمّا يحدث في هذا المثال تحديداً. تُستخدم الأقواس للتأكد بأن `*` تُطبّق على `p` فقط، ومن ثم تحدث زيادةً بمقدار واحد على الشيء المُشار إليه بالمؤشر `p`، وبالنظر إلى جدول الأسبقية في فصل العوامل في لغة سي `C` نلاحظ أن للعاملين `++` و `*` الأسبقية ذاتها، ولكن العاملين يرتبطان من اليمين إلى اليسار، وبمعنى آخر تصبح العملية بالتخلص من الأقواس مكافئةً للعملية `++(p)`، وبغض النظر عن معنى هذه العملية (التي سنتكلم عن معناها لاحقاً)، لا بُدّ من الحفاظ على الأقواس في هذه الحالة والانتباه إلى مواضعها الصحيحة.

لذا، وبما أن المؤشر يعطي عنوان الشيء الذي يشير إليه، فاستخدام *pointer (إذ أن pointer هو أيضًا مؤشر) يعيد الشيء بذاته مباشرةً، ولكن ما الذي نستفيد من ذلك؟ أول الأمور التي نستفيد منها هي تجاوز قيد الاستدعاء بالقيمة call-by-value عند استخدام الدوال؛ فعلى سبيل المثال تخيل دالةً تعيد قيمتين ممثلتين بأعداد صحيحة تمثل الشهر واليوم لهذا الشهر، وأن لهذه الدالة طريقةً (غير محددة) لتحديد هذه القيم، والتحدي هنا هو إعادة قيمتين منفصلتين بنفس الوقت. إليك طريقةً لتجاوز هذه العقبة بالمثال:

```
#include <stdio.h>
#include <stdlib.h>
void
date(int *, int *);    /* التصريح عن الدالة */

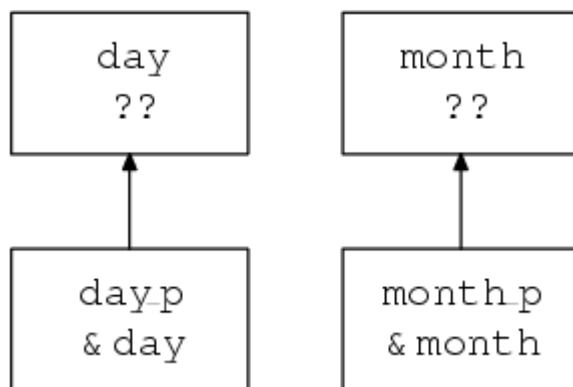
main(){
    int month, day;
    date (&day, &month);
    printf("day is %d, month is %d\n", day, month);
    exit(EXIT_SUCCESS);
}

void
date(int *day_p, int *month_p){
    int day_ret, month_ret;
    /* حساب قيمة day و month في day_ret و month_ret*/
    *day_p = day_ret;
    *month_p = month_ret;
}
```

[مثال 2]

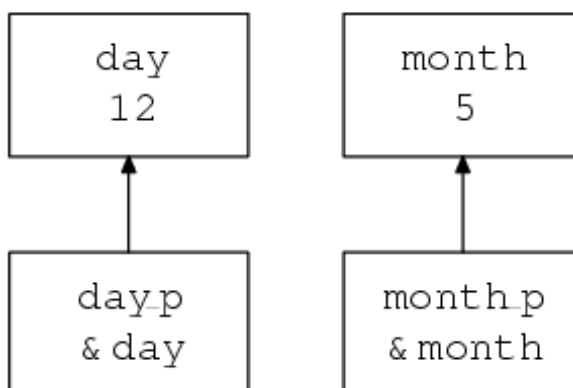
لاحظ طريقة التصريح عن date المتقدمة، التي توضح أنها دالةٌ تأخذ وسيطين من نوع "مؤشر إلى قيمة من نوع int"، وتعيد void لأن القيم التي تُمرّر بواسطة المؤشرات ليست من نوع قيمة اعتيادية. تمرّر الدالة main المؤشرات إلى الدالة date على أنها وسطاء باستخدام المتغيرات الداخلية day_ret و month_ret، ومن ثم تأخذ قيمتهما وتسندهما إلى الأماكن التي تشير إليها وسطاء الدالة (المؤشرات).

يوضح الشكل 6 ما الذي يحدث عند استدعاء الدالة date:



الشكل 6: عند استدعاء الدالة date

تُمرّر الوسطاء إلى date، ولكن المتغيرين day و month غير مهَيَّأين بقيمةٍ أولية ضمن الدالة main. يوضح الشكل 7 ما الذي يحدث عندما تصل الدالة date إلى تعليمة return، بفرض أن قيمة day هي "12" وقيمة month هي "5".



الشكل 7: عندما تصل الدالة date إلى تعليمة return

واحدةً من المزايا الرائعة التي قدّمتها لغة سي المعيارية هي السماح بالتصريح عن أنواع وسطاء دالة date مسبقًا، إذ كان نسيان أن الدالة تقبل مؤشرات ووسطاء لها وتمرير نوع آخر أمرًا شائع الحدوث. تخيل استدعاء الدالة date دون أي تصريح واضح مُسبق لها على النحو التالي:

```
date(day, month);
```

لن يعرف المصرّف هنا أن الدالة تقبل المؤشرات ووسطاء لها وستُمرر قيمًا من نوع int افتراضيًا لكل من day و month، وبما أن تمرير المؤشرات والأعداد الصحيحة يجري على معظم الحواسيب بالطريقة نفسها، لذا ستُنقذ الدالة في هذه الحالة، ثم تعيد القيم في النهاية وتسندّها إلى المكان الذي يشير إليه كلاً من day و month، إذا كانا مؤشرين. لن يعطي ذلك أي نتيجة بل وربما يتسبب بضرر مفاجئ للبيانات في مكان آخر بذاكرة الحاسوب، وتعبُّب هذا النوع من الأخطاء صعبٌ جدًّا.

لحسن الحظ، يمكن أن يعرف المصنّف المعلومات الكافية عن `date` بالتصريح عن الدالة مسبقًا، وذلك من شأنه أن يحذره بخصوص أي أخطاء مرتكبة.

قد يفاجئك سماع أن المؤشرات لا تستخدم كثيرًا لتمكين طريقة الاستدعاء بالإشارة `call-by-reference`، إذ يُعدّ استخدام الاستدعاء بالقيمة `call-by-value` وإعادة قيمة واحدة باستخدام `return` كافٍ في معظم الحالات، والاستخدام الأكثر شيوعًا للمؤشرات هو الانتقال ما بين المصفوفات.

5.3.2 المصفوفات والمؤشرات

تملك عناصر المصفوفة عناوينًا مثل أي متغير اعتيادي آخر.

```
int ar[20], *ip;

ip = &ar[5];
*ip = 0;           // يكافئ التعبير ar[5] = 0;
```

في المثال السابق، يُخزّن عنوان العنصر `ar[5]` في المؤشر `ip`، ثم تُسند القيمة صفر إلى موضع المؤشر في السطر الذي يليه. هذا الشيء غير مثير للإعجاب بحد ذاته، بل إن طريقة عمل العمليات الحسابية والمؤشر سويًا هي التي تستدعي الاهتمام، فعلى الرغم من بساطة هذا الأمر، إلا أنه يُعدّ واحدًا من أساسات لغة سي المميزة.

نحصل على مؤشر إذا أضفنا قيمة عدد صحيح إلى مؤشر ما، ويكون للمؤشر الذي حصلنا عليه نفس نوع المؤشر الأصلي؛ فإضافة العدد "n" إلى مؤشر ما يشير إلى عنصر في مصفوفة، سنحصل على عنصر يشير إلى العنصر الذي يلي العنصر السابق بمقدار "n" داخل المصفوفة ذاتها، ويمكن تطبيق حالة الطرح بما أن العدد "n" يمكن أن يكون سالبًا. بالرجوع إلى المثال السابق، ينتج عن التعبير التالي:

```
*(ip+1) = 0;
```

تغيير قيمة `ar[6]` إلى صفر، وهكذا. لا تعدّ هذه الطريقة تحسينًا على طرق الوصول إلى عناصر المصفوفة الاعتيادية، إليك مثالًا عن ذلك بدلًا من السابق:

```
int ar[20], *ip;

for(ip = &ar[0]; ip < &ar[20]; ip++)
    *ip = 0;
```

يوضّح المثال السابق ميزةً كلاسيكيةً للغة سي، إذ يشير المؤشر إلى بداية المصفوفة، وبينما يشير المؤشر إلى المصفوفة يمكن الوصول إلى عناصر المصفوفة واحدًا تلو الآخر بزيادة المؤشر بمقدار واحد كلّ مرة. يدعم

معياري سي بعض الممارسات الموجودة وذلك بالسماح لاستخدام **عنوان** العنصر `ar[20]` على الرغم من عدم وجود هذا العنصر، ويسمح لك هذا باستخدام المؤشرات لاختبار حدود المصفوفة ضمن الحلقات التكرارية كما هو الحال في المثال السابق، إذ أن ضمان عمله يمتد لعنصر واحد فقط خارج نهاية المصفوفة وليس أكثر من ذلك.

لكن ما المميز في هذه الطريقة مقارنةً باستخدام دليل المصفوفة للوصول إلى العنصر بالطريقة الاعتيادية؟ يمكن الوصول إلى عناصر المصفوفات في معظم الحالات بالتعاقب، واستعرضت القليل من الأمثلة البرمجية السابقة خيار الوصول إلى العناصر "عشوائياً". إذا أردت الوصول إلى عناصر المصفوفة بالتعاقب فسيقدم استخدام المؤشرات تنفيذاً أسرع، إذ يتطلب الأمر على معظم معماريات الحاسوب عملية ضرب وجمع واحدة للوصول إلى عنصر ضمن مصفوفة أحادية البعد باستخدام دليله، بينما لا يتطلب الأمر باستخدام المؤشرات إجراء أي عمليات حسابية إطلاقاً، إذ يخزن المؤشر العنوان الدقيق للكائن (عنصر المصفوفة في هذه الحالة). العملية الحسابية الوحيدة المُجرّاة في المثال السابق هي ضمن حلقة `for` التكرارية، إذ تحدث عملية إضافة ومقارنة كل دورة داخل الحلقة. إذا أردنا استخدام طريقة الوصول لعناصر المصفوفة باستخدام الأدلة، نكتب:

```
int ar[20], i;
for(i = 0; i < 20; i++)
    ar[i] = 0;
```

تحدث العمليات الحسابية ذاتها في الحلقة التكرارية كما في المثال السابق، لكن بإضافة حسابات العنوان المُجرّاة كل دورة في الحلقة.

لا تعدّ نقطة توفير الوقت والفاعلية مشكلةً كبيرةً في معظم الأحيان، إلا أن الأمر مهمٌ هنا في حالة الحلقات التكرارية، إذ تتكرر الحلقات عدداً كبيراً من المرات وكل جزء من الثانية يمكن توفيره في كل دورة له تأثير، ولا يستطيع المصنّف مهما كان ذكياً التعرّف على جميع الحالات التي يمكن له استخدام المؤشرات بدلاً من طريقة دليل المصفوفة ضمناً.

إذا استوعبت جميع ما قرأته حتى الآن فتابع معنا، وإلا فتجاوز هذا القسم واذهب للفصل التالي، فعلى الرغم من المعلومات المثيرة للاهتمام في الأقسام التالية إلا أنها غير ضرورية وتُعرّف برهبتها لمبرمجي لغة سي المتمرسين حتى.

في حقيقة الأمر، لا "تفهم" لغة سي مبدأ الوصول لعناصر المصفوفة باستخدام أدلتها (باستثناء نقطة التصريح عن المصفوفة)، فالتعبير `x[n]` يُترجم بالنسبة للمصنّف على النحو التالي: `*(x+n)`، إذ يُحوّل اسم المصفوفة إلى مؤشر يشير إلى العنصر الأول للمصفوفة وذلك أينما وجد اسم المصفوفة ضمن تعبير ما. يُعد هذا سبباً من ضمن أسباب أخرى لبدء عناصر المصفوفة بالرقم صفر؛ فإذا كان `x` اسم المصفوفة، سيكون التعبير `&x[0]` مساوياً للمصفوفة `x`، أي مؤشر إلى العنصر الأول من المصفوفة.

نستطيع الوصول إلى $x[0]$ باستخدام المؤشرات باستخدام التعبير $(&x[0])$ ، مما يعني أن $(&x[0] + 5)$ مساوٍ للتعبير $(x + 5)$ وهو ذات التعبير $x[5]$. يفتح ذلك الأمر سيلاً من التساؤلات عن الإمكانيات الناتجة، فإذا كان التعبير $x[5]$ يُترجم إلى $(x + 5)$ والتعبير $x + 5$ يعطي النتيجة ذاتها للتعبير:

+ x

فالتعبير $x[5]$ مساوٍ للتعبير $x[5]$. إليك برنامجاً يُصَرِّف وينفذ دون أي أخطاء إن لم تصدق هذا الأمر:

```
#include <stdio.h>
#include <stdlib.h>
#define ARSZ 20
main(){
    int ar[ARSZ], i;
    for(i = 0; i < ARSZ; i++){
        ar[i] = i;
        i[ar]++;
        printf("ar[%d] now = %d\n", i, ar[i]);
    }

    printf("15[ar] = %d\n", 15[ar]);
    exit(EXIT_SUCCESS);
}
```

[مثال 3]

لتلخيص ما سبق:

- تبدأ العناصر في أي مصفوفة من الدليل ذي الرقم صفر.
- ليس هناك أي وجود للمصفوفات متعددة الأبعاد، بل هي في حقيقة الأمر مصفوفاتٌ تحتوي على مصفوفات.
- تُشير المؤشرات إلى أشياء، والمؤشرات التي تشير إلى أشياء من أنواع مختلفة هي بدورها من أنواع مختلفة أيضاً ولا يوجد أي تشابه بين الأنواع المختلفة في لغة سي وأي تحويل ضمني تلقائي بينهما.
- يمكن استخدام المؤشرات لمحاكاة استخدام الاستدعاء بالإشارة ضمن الدوال، ولكن الأمر يستغرق بعضاً من الجهد لتحقيقه.
- تُستخدم زيادة أو نقصان قيمة مؤشر ما للتنقل بين عناصر المصفوفة.

- يضمن المعيار أن محاولة الوصول إلى العنصر ذي الدليل "n" في مصفوفة ذات حجم "n" محاولةً صالحة على الرغم من عدم وجود هذا العنصر وذلك لتسهيل أمر التنقل داخل المصفوفة بزيادة قيمة المؤشر، ويكون مجال قيم مصفوفة مصرّح عنها على النحو `int ar[N]` هو `ar[0]` وصولاً إلى `ar[N]` ولكن يجب عليك تفادي الوصول إلى قيمة العنصر الأخير الزائف.

5.3.3 الأنواع المؤهلة Qualified

تابع قراءة الفقرات التالية إن كنت واثقاً من فهمك لأساسيات عملية التصريح عن المؤشرات واستخدامها، وإلا فمن المهم العودة إلى الفقرات السابقة ومحاولة فهمها جيّداً قبل قراءة الفقرات التالية، إذ أن المعلومات المذكورة في الفقرات التالية أكثر تعقيداً، ولا داعٍ لجعل الأمر أسوأ بعدم تحضيرك وفهمك لما سبق.

قدم المعيار شيئين باسم **مؤهلات النوع type qualifiers**، إذ لم يكونا في لغة سي القديمة مسبقاً، ويمكن تطبيقهما لأي نوع مصرّح عنه للتعديل من تصرفه، ومن هنا أتى اسمهما. سنتجاهل إحداهما (المدعو باسم "volatile") إلا أنه لا يمكننا تجاهل الآخر "const".

إذا سبقت الكلمة المفتاحية "const" أي تصريح، فهذا يعني أن الشيء الذي صُرح عنه هو ثابت، ويجب عليك تجنب محاولة التغيير على قيمة الكائنات الثابتة "const" وإلا فستحصل على سلوك غير محدد `undefined behaviour`، وسيحدّرك المصرّف عادةً بمنعك من هذه المحاولة إلا إذا تجاوزت هذا القيد بحيلةٍ ما.

هناك فائدتان مرجوتان من تصريح كائن ما بكونه ثابتاً "const":

1. يشير استخدام الكلمة المفتاحية "const" إلى أن القيمة غير قابلة للتغيير، مما يجبر المصرّف على التحقق من أي تغييرات طرأت على هذه القيمة ضمن الشيفرة البرمجية، وهذا الأمر باعث للطمأنينة في حال أردت استخدام قيم ثابتة، مثل المؤشرات التي تُستخدم وسطاء في بعض الدوال. إذا احتوى التصريح عن الدالة مؤشرات تشير إلى كائنات ثابتة على أنها وسطاء، فهذا يعني أن الدالة لن تشير إلى أي كائن آخر.

2. عندما يعلم المصرّف بأن الأشياء هي كائنات ثابتة، ينعكس ذلك إيجابياً على قدرته برفع فاعلية الشيفرة البرمجية وسرعتها.

الثوابت عديمة الفائدة في حال لم تسند إليها أي قيمة، لن نتطرق بالتفصيل بخصوص تهيئة الثوابت (سنناقش الأمر لاحقاً)، كل ما عليك تذكره الآن هو أنه من الممكن لأي تصريح إسناد القيمة لتعبيرٍ ثابت. إليك بعض الأمثلة عن التصريح عن ثوابت:

```
const int x = 1;          /* ثابت x */
const float f = 3.5;    /* ثابت f*/
```

```
const char y[10];      /* مصفوفة من 10 عناصر ذات قيم صحيحة ثابتة y */
                      /* لا تفكر بخصوص تهيئة قيمها بعد */
```

ما يثير الاهتمام هو كون هذا المؤهل ممكن التطبيق على المؤشرات بطريقتين: إما بجعل الشيء الذي يشير إليه المؤشر ثابتًا، بحيث يصبح نوع المؤشر "مؤشر إلى ثابت"، أو بجعل المؤشر بذات نفسه ثابتًا (مؤشرًا ثابتًا)، إليك مثالًا عن ذلك:

```
int i;                  /* عدد صحيح اعتيادي */
const int ci = 1;      /* عدد صحيح ثابت */
int *pi;               /* مؤشر إلى عدد صحيح */
const int *pci;        /* مؤشر إلى عدد صحيح ثابت */
                      /* بالانتقال إلى الأمثلة الأكثر تعقيدًا */

// مؤشر ثابت يشير إلى قيمة عدد صحيح
int *const cpi = &i;

// مؤشر ثابت يشير إلى قيمة عدد صحيح ثابتة
const int *const cpci = &ci;
```

التصريح الأول (للمتغير *i*) اعتيادي، ولكن تصريح *ci* يوضح أنه عدد صحيح ثابت وبذلك لا يمكن التعديل على قيمته، وسيكون بلا فائدة إن لم يُهيأ (إسناد قيمة أولية له).

ليس من الصعب فهم الغرض من مؤشر لعدد صحيح، ومؤشر لعدد صحيح ثابت، ولكن يجب الانتباه إلى أنواع المؤشرات المختلفة، إذ لا يجوز الخلط بينهم. يمكنك تغيير قيمة كل من *pi* و *pci* بجعلهما يشيران إلى أشياء أخرى، كما يمكنك تغيير قيمة الشيء الذي يشير إليه المؤشر *pi* بالنظر إلى كونه عدد صحيح غير ثابت، ولكن لا يمكنك إلا الوصول إلى قيمة الشيء الذي يشير إليه المؤشر *pci* دون التعديل عليه نظرًا لكونه ثابتًا.

يُعد التصريحان الأخيران أكثر التصريحات تعقيدًا ضمن المثال؛ فإذا كانت المؤشرات بذات نفسها ثابتة، فمن غير المسموح تغيير المكان الذي تشير إليه، وبالتالي يجب تهيئتهما بقيمة أولية مثل *ci*. يمكن للشيء المُشار إليه بالمؤشر أن يكون ثابتًا أو غير ثابت بغض النظر عن كون المؤشر ثابتًا بدوره أم لا، وهذا يملي بعض القيود على استخدام الكائن المُشار إليه.

أخيرًا للتوضيح: ما الذي يملي وجود نوع مؤهل؟ كان *ci* في المثال السابق نوعًا مؤهلًا بكل وضوح، ولكن *pci* لم تنطبق عليه هذه الحالة بما أن المؤشر ليس من نوع مؤهل بل الشيء الذي يشير إليه. الأشياء الوحيدة الذي كانت ذات أنواع مؤهلة في المثال هي: *ci* و *cpi* و *cpci*.

سيطلب منك الأمر بعض الوقت للاعتياد على هذا النوع من التصريحات، ولكن استخدامها سيكون تلقائيًا وطبيعيًا بعد فترة فلا تقلق، إلا أن التعقيدات تبدأ بالظهور لاحقًا عندما يتوجب عليك الإجابة على السؤال: "هل من المسموح لي -على سبيل المثال- مقارنة مؤشر اعتيادي مع مؤشر ثابت؟ وإذا كان الجواب نعم، ما الذي تعنيه المقارنة؟". معظم القوانين واضحة بهذا الخصوص، ولكن يجب ذكرها وتوضيحها بغض النظر عن سهولتها.

سنتكلم عن أنواع البيانات المؤهلة بتوسيع أكبر لاحقًا.

5.3.4 عمليات المؤشرات الحسابية

سنتكلم بالتفصيل لاحقًا عن عمليات المؤشرات الحسابية، ولكننا سنكتفي الآن بإصدار مختصر يفي بالغرض.

يمكنك طرح أو المقارنة بين مؤشرين من النوع نفسه بالإضافة إلى إضافة قيمة عددية صحيحة إلى المؤشر، ويجب أن يشير كلا المؤشرين إلى المصفوفة ذاتها وإلا حصلنا على سلوك غير محدد. نتيجة طرح مؤشرين هي قيمة العناصر التي تفصل بينهما في المصفوفة، ونوع النتيجة معرفٌ بحسب التطبيق وسيكون إما "short" أو "int" أو "long"، ويوضح المثال القادم مثالاً على حساب الفرق بين مؤشرين واستخدام النتيجة، ولكن قبل أن ننتقل إلى المثال يجب أن تعرف معلومة مهمة.

يُحوّل اسم المصفوفة في أي تعبير إلى مؤشر يشير إلى العنصر الأول ضمن هذه المصفوفة. والحالة الوحيدة الاستثنائية هي عند استخدام اسم المصفوفة مع الكلمة المفتاحية sizeof، أو عند استخدام سلسلة نصية لتهيئة قيمة مصفوفة ما، أو عندما يكون اسم المصفوفة مرتببًا بعامل "عنوان الكائن" (العامل الأحادي &). لكننا لم نتطرق إلى أي من الحالات السابقة بعد، وسنناقشها لاحقًا. إليك المثال:

```
#include <stdio.h>
#include <stdlib.h>
#define ARSZ 10

main(){
    float fa[ARSZ], *fp1, *fp2;

    fp1 = fp2 = fa; /* عنوان العنصر الأول */
    while(fp2 != &fa[ARSZ]){
        printf("Difference: %d\n", (int)(fp2-fp1));
        fp2++;
    }
    exit(EXIT_SUCCESS);
}
```

}

[مثال 4]

يشير المؤشر fp2 إلى عناصر المصفوفة، ويُطبع الفرق بين قيمته الحالية وقيمه الأصلية، وللتأكد من عدم تمرير النوع الخاطئ للوسيط للدالة printf تُحوّل القيمة الناتجة عن فرق المؤشرين قسريًا إلى int باستخدام تحويل الأنواع (int)، وهذا يجنبنا من الأخطاء على الحواسيب التي تعيد قيمة long لهذا النوع من العمليات. قد يعود إلينا المثال السابق بإجابات خاطئة إذا كان الفرق بين القيمتين من نوع long وكانت المصفوفة كبيرة الحجم، ونلاحظ في المثال التالي إصدارًا آمنًا من المثال السابق، إذ يُستخدم تحويل الأنواع قسريًا للسماح بوجود قيم long:

```
#include <stdio.h>
#define ARSZ 10

main(){
    float fa[ARSZ], *fp1, *fp2;

    fp1 = fp2 = fa; /* عنوان العنصر الأول */
    while(fp2 != &fa[ARSZ]){
        printf("Difference: %ld\n", (long)(fp2-fp1));
        fp2++;
    }
    return(0);
}
```

[مثال 5]

5.3.5 مؤشرات void و null والمؤشرات الإشكالية

لغة سي حريصة بشأن أنواع المؤشرات، ولن تسمح لك عمومًا باستخدام مؤشرات ذات أنواع مختلفة ضمن التعبير ذاته. يختلف مؤشر إلى نوع "char" عن مؤشر إلى نوع "int" ولا يمكنك -على سبيل المثال- إسناد قيمة أحدهما إلى الآخر أو المقارنة فيما بينهما أو طرحهما من بعضهما واستخدام النتيجة مثل وسيط في دالة ما، كما يمكن أيضًا تخزين النوعين في الذاكرة بصورة مختلفة وأن يكونا بأطوال مختلفة.

لا تتماثل المؤشرات من أنواع مختلفة فيما بينها، وليس هناك أي تحويلات ضمنية بين الأنواع كما شاهدنا في الأنواع الحسابية سابقًا.

لكن نريد في بعض الحالات تجاوز هذه القيود، فكيف نفعل ذلك؟

يكمن الحل هنا باستخدام أنواع خاصة، وقد قدمنا واحدًا من هذه الأنواع سابقًا ألا وهو "مؤشر إلى void"، وقُدِّمت هذه الميزة مع سي المعيارية إذ افترض سابقًا أن المؤشر من نوع "مؤشر إلى char" كافٍ لهذه المهمة، وكان الافتراض صحيحًا إلى حدٍّ ما إلا أنه كان حلاً غير منظمًا، بينما قدّم الحل الجديد طريقةً أكثر أمانًا وأقل تشويشًا. لا يوجد أي استخدام للمؤشر هذا، لأن "void *" لا يشير إلى أي قيمة، لذا يحسّن هذا الأمر من سهولة قراءة الشيفرة البرمجية. يمكن أن يخزن المؤشر من النوع "void *" أي قيمة من أي مؤشر آخر، ويمكن إسناده إلى مؤشر آخر من أي نوع أيضًا، إلا أنه يجب استخدام هذا النوع من المؤشرات بحذر لأنه قد ينتهي الأمر بك ببعض الأخطاء الوخيمة، وسنناقش استخدامه الآمن مع دالة "malloc" لاحقًا.

قد تحتاج أيضًا في بعض الحالات إلى مؤشر مضمون أنه لا يشير إلى أي كائن والذي يُدعى مؤشر من نوع "null" أو **مؤشر الفراغ null pointer**. من الشائع في لغة سي كتابة بعض الإجراءات routines التي تعيد مؤشرات، بحيث يمكن الدلالة على فشل ذلك الإجراء بعدم قدرته على إعادة مؤشر صالح عن طريق إعادة مؤشر الفراغ. ومن الأمثلة على ذلك إجراء فحص لقيم موجودة في جدول ما بحثًا عن قيمة معينة ويعيد مؤشرًا على الكائن المُطابق إن وُجدت النتيجة أو مؤشر الفراغ إن لم تُوجد أي نتيجة مطابقة.

لكن كيف يمكن كتابة مؤشر فراغ؟ هناك طريقتان لفعل ذلك وكلا الطريقتين متماثلتين في النتيجة، إما باستخدام رقم صحيح ثابت بقيمة "0" أو تحويل القيمة إلى نوع "void *" باستخدام التحويل بين الأنواع، وتدعى نتيجة الطريقتين **بمؤشر الفراغ الثابت null pointer constant**. إذا أسندت مؤشر فراغ إلى أي مؤشر آخر أو قارنت بين مؤشر الفراغ ومؤشر آخر فسيُحوّل مؤشر الفراغ إلى نوع المؤشر الآخر تلقائيًا، مما سيحلّ أي مشكلة بخصوص توافقية الأنواع، ولن تُساوي القيمة التي يشير إليها ذلك المؤشر -مؤشر الفراغ- أي قيمة كائن آخر يشير إليها أي مؤشر داخل البرنامج (أي سيشير إلى قيمة فريدة).

القيم الوحيدة الممكن إسنادها للمؤشرات باستثناء القيمة "0" هي قيم المؤشرات الأخرى من نفس النوع، إلا أن الأمر الذي يجعل من لغة سي مميزة وبدليًا مفيدًا للغة التجميعية هو سماحها لك بفعل بعض الأشياء التي لا تسمح لك بها معظم لغات البرمجة الأخرى، جرّب التالي:

```
int *ip;
ip = (int *)6;
*ip = 0xFF;
```

ما نتيجة السابق؟ نهيًا قيمة المؤشر إلى 6 (لاحظ تحويل نوع 6 من int إلى مؤشر)، وهذه عملية تُجرى على مستوى الآلة غالبًا ويكون تمثيل قيمة المؤشر بالبتات غير مشابه لما قد يكون تمثيل الرقم 6، كما تُسند القيمة الست عشرية FF بعد التهيئة إلى الكائن الذي يشير إليه المؤشر. تُكتب القيمة 0xFF على الرقم الصحيح ذو الموضع 6، إذ يعتمد الموضع 6 على تفسير الآلة له على الذاكرة.

قد تحتاج لهذا النوع من الأشياء وقد لا تحتاج إليها إطلاقًا، ولكن لغة سي تعطيك الخيار، ومن الواجب معرفتها إذ أنه من الممكن كتابتها على نحو خاطئ غير مقصود مما سيتسبب بنتائج مفاجئة جدًا.

5.4 التعامل مع المحارف والسلاسل النصية

تُستخدم لغة سي على نطاق واسع في تطبيقات المعالجة والتعامل بالمحارف characters والسلاسل النصية strings، وهذا الأمر غريب بعض الشيء لأن اللغة لا تحتوي على مزايا موجهة لهذا الغرض بالتحديد؛ وإذا كنت معتادًا على لغات البرمجة التي تحتوي على مزايا موجهة للتعامل مع المحارف والسلاسل النصية، فستجد التعامل مع لغة سي بهذا الخصوص مضجرًا على أقل تقدير.

تحتوي المكتبة القياسية على العديد من الدوال التي تساعدك في التعامل مع السلاسل النصية، إلا أن الأمر يبقى صعبًا بعض الشيء مقارنةً بلغاتٍ أخرى. على سبيل المثال، عليك استدعاء دالة مخصصة للمقارنة ما بين سلسلتين نصيتين بدلًا من استخدام عامل المساواة "=", ولكن هناك جانبٌ مشرقٌ لهذا الأمر، إذ يعني ذلك أن اللغة غير مُثقلة بطرق دعم معالجة السلاسل النصية مما يساعد بالمحافظة على برامج أصغر وأقل تشعبًا، وحالما تكتب برنامجًا لمعالجة السلاسل النصية بلغة سي بنجاح أخيرًا، ستكون قادرًا على تشغيله بسرعة أكبر مقارنةً باللغات الأخرى.

5.4.1 التعامل مع المحارف

يجري التعامل مع محارف السلسلة النصية في لغة سي عن طريق التصريح عن مصفوفات، أو حجزهم ديناميكيًا، والتعامل مع المحارف وتحريكها "يدويًا". إليك مثالًا عن برنامج يقرأ نصًا سطرًا بسطر من دخل البرنامج القياسي، ويتوقف البرنامج عن القراءة إذا كان السطر مكوّنًا من السلسلة النصية "stop"، ويُطبع طول السطر فيما عدا ذلك. يعتمد البرنامج على تقنية تُستخدم في معظم برامج سي ألا وهي: يقرأ البرنامج المحارف ويحوّلها إلى مصفوفة ويحدّد نهاية المصفوفة بمحرفٍ إضافي له القيمة صفر؛ كما يستخدم هذا المثال دالة strcmp للمقارنة بين سلسلتين نصيتين من خلال تضمين المكتبة string.h.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LINELNG 100 /* الطول الأعظمي لسطر الدخل الواحد */

main(){
    char in_line[LINELNG];
    char *cp;
    int c;
```

```

cp = in_line;
while((c = getc(stdin)) != EOF){
    if(cp == &in_line[LINELNG-1] || c == '\n'){
        /*إدخال ما يدل على نهاية السطر*/
        *cp = 0;
        if(strcmp(in_line, "stop") == 0 )
            exit(EXIT_SUCCESS);
        else
            printf("line was %d characters long\n",
                (int)(cp-in_line));
        cp = in_line;
    }
    else
        *cp++ = c;
}
exit(EXIT_SUCCESS);
}

```

[مثال 6]

يوضح لنا هذا المثال مزيدًا من مزايا وطرق لغة سي المُستخدمة في برامجها، وأهمها هو طريقة تمثيل السلاسل النصية ومعالجتها.

إليك تطبيقًا عمليًا عن الدالة `strcmp` التي تقارن بين سلسلتين نصيتين وتعيد القيمة صفر إن تساوت قيمتهما، وللدالة هذه في الحقيقة تطبيقات أكثر ولكننا سنتجاهل المعلومات التي قد تزيد من تعقيد شرحنا. لاحظ استخدام الكلمة المفتاحية `const` في التصريح عن الوسطاء، والذي يوضّح أن الدالة لن تعدل على محتويات السلسلة النصية بل ستفحص قيم محتوياتها فقط، ونلاحظ استخدام هذه الطريقة في التعريف عن العديد من دوال المكتبة القياسية.

```

/*
 * برنامج يختبر مساواة سلسلتين نصيتين
 * يُعيد القيمة "خطأ" إذا تساوت السلسلتين
 */
int
str_eq(const char *s1, const char *s2){
    while(*s1 == *s2){
        /*

```

```

        إعادة 0 عند نهاية السلسلة النصية *
    */
    if(*s1 == 0)
        return(0);
    s1++; s2++;
}
/* عُثر على فرق بين السلسلتين */
return(1);
}

```

[مثال 7]

5.4.2 السلاسل النصية Strings

يعرف كل مبرمج لغة سي معنى السلسلة النصية، فهي مصفوفة من متغيرات من نوع "char"، ويكون المحرف الأخير لهذه السلسلة النصية متبوعاً بمحرف فراغ null. ربما تصرخ الآن وتقول "ولكنني اعتقدت أن السلسلة النصية هي نصٌ محتوى داخل إشارتي تنصيص!" أنت محقٌّ، إذ تُعد السلسلة التالية في لغة سي مصفوفةً من المحارف:

```
"a string"
```

وهذا الشيء الوحيد الذي يمكنك التصريح عنه لحظة استخدامه، أي دون التصريح عن مصفوفة المحارف وتحديد حجمها.

كانت السلاسل النصية في لغة سي القديمة تُخزن مثل أي سلسلة محارف اعتيادية، وكانت قابلةً للتعديل. إلا أن المعيار ينص على أن محاولة التعديل على سلسلة نصية سيتسبب بسلوك غير محدد على الرغم من كون السلاسل النصية مصفوفةً من نوع "char" وليس "const char".

لاستخدام السلسلة النصية داخل علامتي تنصيص أتران: أولهما أن السلسلة النصية تحل محل تصريح وبديل عن الاسم، كما أنها تمثل تصريحاً خفياً لمصفوفة من المحارف، وتُهيئ قيمة هذه المصفوفة إلى قيم المحارف الموجودة في السلسلة النصية متبوعاً بمحرف قيمته صفر، ولا يوجد لهذه المصفوفة أي اسم، لذا باستثناء الاسم، يكون الوضع مشابهاً للتالي:

```

char secret[9];
secret[0] = 'a';
secret[1] = ' ';
secret[2] = 's';

```

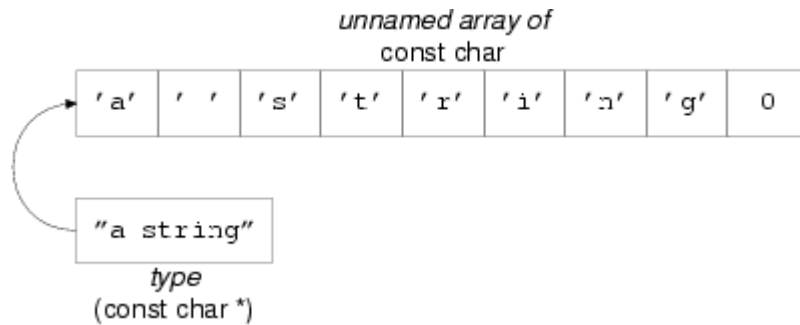
```
secret[3] = 't';
secret[4] = 'r';
secret[5] = 'i';
secret[6] = 'n';
secret[7] = 'g';
secret[8] = 0;
```

وهي مصفوفةٌ من المحارف متبوعةً بقيمة صفر، وتحتوي جميع قيم المحارف بداخلها، لكنها عديمة الاسم إذا صُرِّحَ عنها باستخدام طريقة السلسلة النصية المحاطة بعلامتي تنصيص، فكيف نستطيع استخدامها؟

يكون وجود السلسلة النصية بمثابة اسمٍ مخفي لها حالما تراها لغة سي محاطةً بعلامتي تنصيص، إذ أن وجود السلسلة النصية بهذا الشكل لا يؤدي إلى تصريح ضمني فقط، بل يؤدي إلى إعطاء اسمٍ لمصفوفة موافقة. نتذكر جميعنا أن اسم المصفوفة مساوٍ لعنوان عنصرها الأول (تفقد فصل المؤشرات) فما نوع السلسلة التالية؟

"a string"

النوع مؤشرٍ طبعًا، إذ أن السلسلة النصية المكتوبة بالشكل السابق تكافئ مؤشرًا إلى أول عناصر المصفوفة الخفية عديمة الاسم، وهي مصفوفةٌ من نوع char، فالمؤشر من نوع "مؤشر إلى char"، ويوضح الشكل 8 هذه الحالة.



الشكل 8: أثر استخدام السلسلة النصية

للبرهان على السابق، ألق نظرةً على البرنامج التالي:

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int i;
    char *cp;

    cp = "a string";
```

```

while(*cp != 0){
    putchar(*cp);
    cp++;
}
putchar('\n');

for(i = 0; i < 8; i++)
    putchar("a string"[i]);
putchar('\n');
exit(EXIT_SUCCESS);
}

```

[مثال 8]

تضبط الحلقة الأولى مؤشرًا يشير إلى بداية المصفوفة، ويُمرّر المؤشر على المصفوفة إلى أن يصل المؤشر إلى القيمة صفر في النهاية، بينما "تُعرف" الحلقة الثانية طول السلسلة النصية وهي أقل فائدة مقارنةً بسابقتها. لاحظ أن الحلقة الأولى غير معتمدة على طول السلسلة النصية بل على وصولها لنهايتها وهو الشيء الأهم الذي ينبغي عليك تذكره من هذا المثال.

يجري التعامل مع السلاسل النصية في لغة سي بطريقة ثابتة دون أي استثناءات وهي الطريقة التي تتوقعها جميع دوال التعامل مع السلاسل النصية، فيسمح الصفر في نهاية السلسلة النصية للدوال بمعرفة نهاية السلسلة النصية. عُد للمثال السابق الخاص بدالة `str_eq`، إذ تأخذ الدالة مؤشرين يشيران لمحرفين مثل وسطاء (تُقبل السلسلة النصية ذاتها بكونها واحدًا من الوسيطين أو كلاهما) وتُقارن السلسلة النصية بالتحقق من كل محرف واحدًا تلو الآخر، وإذا تساوى المحرفين فتتحقق من أن المؤشر لم يصل إلى نهاية السلسلة عن طريق الجملة الشرطية التالية:

```
if(*s1 == 0):
```

وإذا وصل المؤشر للنهاية فإنها تُعيد 0 للدلالة على أن السلسلتين متساويتين، يمكن إجراء الاختبار ذاته باستخدام المؤشر `*s2` دون أي فارق؛ أما في حالة الفرق بين المحرفين تُعاد القيمة 1 للدلالة على فشل المساواة.

تُستدعى الدالة `strcmp` في المثال باستخدام وسيطين مختلفين عن بعضهما، إذ أن الوسيط الأول هو مصفوفة محارف والثاني سلسلة نصية، لكنهما في الحقيقة يمثلان الشيء ذاته؛ فمصفوفة المحارف التي تنتهي بالعنصر صفر (يسند برنامجنا القيمة صفر إلى أول عنصر فارغ في مصفوفة `in_line`) والسلسلة

النصية بين قوسين (التي تُمَثَّل بدورها أيضًا مصفوفة محارف تنتهي بصفر) من نفس الطبيعة، واستخدامهما مثل وسيطين للدالة strcmp ينتج بتمرير مؤشري محارف (وقد شرحنا السبب بذلك بإفاضة سابقًا).

5.4.3 المؤشرات وعامل الزيادة

ذكرنا سابقًا التعبير التالي، وقلنا أننا ستعيد النظر فيه لاحقًا:

```
(*p)++;
```

حان الوقت الآن للكلام عن ذلك؛ إذ تُستخدم المؤشرات بكثرة مع المصفوفات والتمرير بينها، لذلك من الطبيعي استخدام العاملين -- و ++ معها. يوضح المثال التالي إسناد القيمة صفر إلى مصفوفة باستخدام المؤشر وعامل الزيادة:

```
#define ARLEN 10

int ar[ARLEN], *ip;

ip = ar;
while(ip < &ar[ARLEN])
    *(ip++) = 0;
```

[مثال 9]

يُضبط المؤشر ip ليبدأ من بداية المصفوفة، وعلى الرغم من إشارة المؤشر إلى داخل المصفوفة إلا أن القيمة التي يشير إليها تساوي الصفر، وعند وصولنا للحركة التكرارية ينجز عامل الزيادة ما هو متوقع ويُنقل المؤشر للعنصر الذي يليه داخل المصفوفة، وبالتالي استخدام العامل ++ عامل إلحاق postfix مفيد في هذه الحالة.

معظم الأشياء التي ناقشناها شائعة الوجود، وستجدها في معظم البرامج (استخدام عامل الزيادة والمؤشرات بالشكل الموضح في المثال السابق ليس مرة واحدة أو مرتين بل تقريبًا كل عدّة أسطر ضمن الشيفرة البرمجية، وستعتقد أنك تراها بصورة أكثر إذا كنت تجد استخدامها صعب الفهم. لكن ما هي التشكيلات التي يمكننا الحصول عليها؟ بالنظر إلى أن * تعني التأشير و ++ تعني الزيادة و -- تعني النقصان، كما أنه لدينا خيار وضع العاملين السابقين مثل عامل إلحاق postfix أو عامل إسباق prefix، نحصل على الاحتمالات التالية (بغض النظر عن عامل الزيادة أو النقصان) مع التركيز على موضع الأقواس:

الجدول 14: معاني المؤشرات

زيادة سابقة للشئ الذي يشير إليه المؤشر	++(*p)
زيادة لاحقة للشئ الذي يشير إليه المؤشر	(*p)++
زيادة لاحقة على المؤشر	*(p++)
زيادة سابقة على المؤشر	*(++p)*

اقرأ الجدول السابق بحرص وتأكد أنك تفهم جميع التركيبات التي ذُكرت.

يمكن فهم محتوى الجدول السابق بعد تفكير بسيط، ولكن هل يمكنك توقع ما الذي سيحدث عند إزالة الأقواس بالنظر إلى أن الأسبقية للعوامل الثلاث * و - - و ++ متساوية؟ تتوقع حدوث أخطاء كارثية، أليس كذلك؟ يوضح الجدول 14 أن هناك حالة واحدة يجب أن تحافظ فيها على الأقواس.

الجدول 15: المزيد من معاني المؤشرات

مع أقواس	دون أقواس إن أمكن
++(*p)	++*p
(*p)++	(*p)++
*(p++)	*p++
*(++p)	*++p

لعلّ الأشكال المثيرة للتشويش في الجدول السابق ستدفعك لاستخدام الأقواس بغض النظر عن أهمية استعمالها في أي حالة سعيًا منك لتحسين قابلية قراءة الشيفرة البرمجية وفهمها، لكن يتخلى معظم مبرمجي لغة سي عن استخدام الأقواس بعد تعلّم قوانين الأسبقية ونادرًا ما يستخدمون الأقواس في تعابيرهم، لذا عليك الاعتياد على قراءة الأمثلة التالية سواءً كانت مع أقواس أو بدونها، فالاعتياد على هذا الأمر سيساعدك بحقّ في تعلمك.

5.4.4 المؤشرات عديمة النوع

من المهم في بعض الأحيان تحويل نوع من المؤشرات إلى نوع آخر بمساعدة التحويل بين الأنواع مثل التعبير التالي:

```
(type *) expression
```

يُحوّل التعبير expression في المثال السابق إلى مؤشر من نوع "مؤشر إلى نوع type" بغض النظر عن نوع التعبير السابق، إلا أنه يجب تفادي هذه الطريقة إلا في حال كنت مدركًا تمامًا لما تفعله، فمن غير المحبذ استخدامها إلا إذا كنت مبرمجًا خبيرًا. لا تفترض أن التحويل بين الأنواع يلغي أي حسابات أخرى بخصوص

"الأنواع غير المتوافقة مع بعضها" التي تقع على عاتق المبرمج، إذ من المهم إعادة حساب القيم الجديدة للمؤشر بعد تغيير نوعه على العديد من معماريات الحاسوب.

هناك بعض الحالات التي ستحتاج فيها لاستخدام مؤشر "معمم generic"، وأبرز مثال على ذلك هو تطبيق لدالة المكتبة القياسية malloc التي تُستخدم لحجز المساحة على الذاكرة للكائن الذي لم يصرّح عنه بعد، ويجري تزويد الحجم المراد حجزه عن طريق تزويد مثل وسيط سواء كان الكائن متغيراً من نوع float أو مصفوفة من نوع int أو أي شيء آخر. تعيد الدالة مؤشراً إلى عنوان التخزين المحجوز التي تختاره بطريقتها الخاصة (والتي لن نتطرق إليها) من مجموعة من عناوين الذاكرة الفارغة، ومن ثم يُحوّل المؤشر إلى النوع المناسب. على سبيل المثال، تحتاج القيمة من نوع float إلى 4 بايتات من الذاكرة. وبالتالي نكتب ما يلي لحجز مساحة للقيمة:

```
float *fp;

fp = (float *)malloc(4);
```

تعتبر الدالة malloc على 4 بايتات من الذاكرة الفارغة، ويُحوّل عنوان الذاكرة إلى مؤشر من نوع "مؤشر إلى float"، ثم تُسند القيمة إلى المؤشر (fp في حالة مثالنا السابق).

لكن ما هو نوع المؤشر الذي ستُسند قيمة malloc إليه؟ نحن بحاجة نوع يمكن أن يحتوي جميع أنواع المؤشرات فنحن لا نعلم نوع المؤشر الذي ستعيده الدالة malloc.

الحل هو باستخدام نوع المؤشر * void الذي تكلمنا عنه سابقاً، إليك المثال السابق مع إضافة تصريح للدالة malloc:

```
void *malloc();

float *fp;

fp = (float *)malloc(4);
```

لا حاجة لاستخدام تحويل الأنواع على القيمة المُعادَة من الدالة malloc حسب قوانين الإسناد للمؤشرات، ولكن استُخدم تحويل الأنواع لممارسة الأمر لا أكثر.

لا بد من طريقة لمعرفة قيمة وسيط malloc الدقيقة في نهاية المطاف، ولكن القيمة ستكون مختلفة على أجهزة بمعماريات مختلفة، لذا لا يمكنك الاكتفاء باستخدام القيمة الثابتة 4 فقط، بل يجب علينا استخدام عامل sizeof.

5.5 عامل sizeof وحجز مساحات التخزين

يُعيد العامل "sizeof" حجم المُعامل operator بالبايتات، وتعتمد نتيجة العامل "sizeof" بكونها عددًا صحيحًا عديم الإشارة "unsigned int" أو عددًا كبيرًا عديم الإشارة "unsigned long" على التطبيق implementation، وهذا هو السبب في تفادينا لأي مشكلات في المثال السابق (الفصل السابق) عند التصريح عن دالة malloc على الرغم من عدم تزويد التصريح بأي تفاصيل عن معاملاتها؛ إذ يجب استخدام ملف الترويسة stdlib.h عوضًا عن ذلك عادةً للتصريح عن malloc على النحو الصحيح. إليك المثال ذاته ولكن بتركيز على جعله قابلًا للتنقل portable عبر مختلف الأجهزة:

```
#include <stdlib.h> /* يتضمن ملف الترويسة تصريحًا عن malloc() */
float *fp;

fp = (float *)malloc(sizeof(float));
```

يجب أن يُكتب معامل sizeof داخل قوسين إذا كان فقط اسمًا لنوع بيانات (وهي الحالة في مثالنا السابق)، بينما يمكنك التخلي عن القوسين إذا كنت تستخدم اسم كائن بيانات عوضًا عن ذلك، ولكن هذه الحالة نادرة الحدوث.

```
#include <stdlib.h>

int *ip, ar[100];
ip = (int *)malloc(sizeof ar);
```

لدينا في المثال السابق مصفوفة باسم ar مكونة من 100 عنصر من نوع عدد صحيح int، ويشير ip إلى مساحة التخزين الخاصة بهذه المصفوفة (مساحة لمئة قيمة من نوع int) بعد استدعاء malloc (بفرض أن الاستدعاء كان ناجحًا).

تعدّ char (محرف وهي اختصارٌ إلى character) وحدة القياس الأساسية للتخزين في لغة سي، وتساوي بايتًا واحدًا، جرّب نتيجة التعليمة الآتية:

```
sizeof(char)
```

وبناءً على ذلك، يمكنك حجز مساحة لعشرة قيم من نوع char على النحو التالي:

```
malloc(10)
```

ولحجز مساحة لمصفوفة بحجم عشرة قيم من نوع int، نكتب:

```
malloc(sizeof(int[10]))
```

تُعيد الدالة malloc مؤشرًا إلى الفراغ null pointer في حال لم تتوفر المساحة الكافية للإشارة إلى خطأ ما. يحتوي ملف الترويسة `stdio.h` ثابتًا معرفًا باسم `NULL`، والذي يُستخدم عادةً للتحقق من القيمة المُعادة من الدالة `malloc` ودوال أخرى من المكتبة القياسية، وتُعد القيمة 0 أو `(void *)0` مساويةً لهذا الثابت ويمكن استخدامها.

إليك المثال التالي لتوضيح استخدام الدالة `malloc`، إذ يقرأ البرنامج في المثال سلاسلًا نصيةً بعدد `MAXSTRING` من الدخل، ثم يرتب السلاسل النصية أوجدًا باستخدام الدالة `strcmp`. يُشار إلى نهاية السلسلة النصية بمحرف التهريب `escape character` التالي `\n`، وتُرتَّب السلاسل باستخدام مصفوفة من المؤشرات تُشير إلى السلسلة النصية وتبديل مواضع المؤشرات حتى الوصول إلى الترتيب الصحيح، مما يجتنبنا عناء نسخ السلاسل النصية ويحسن من سرعة تنفيذ البرنامج ويحد من هدر الموارد إلى حدٍّ ما.

استخدمنا في الإصدار الأول من المثال مصفوفةً ثابتة الحجم، ثم استخدمنا في الإصدار الثاني حجز المساحة باستخدام `malloc` لكل سلسلة نصية عند وقت التشغيل `run-time`، بينما بقيت مصفوفة المؤشرات -لسوء الحظ- ثابتة الحجم، إلا أنه يمكننا تطبيق حلٍّ أفضل باستخدام قائمة مترابطة `Linked list`، أو أي هيكل بيانات مشابه لتخزين المؤشرات دون الحاجة لاستخدام المصفوفات ثابتة الحجم إطلاقًا، ولكننا لم نتكلم عن هياكل البيانات بعد.

إليك ما يبدو عليه هيكل برنامجنا:

```
while(number of strings read < MAXSTRING
    && input still remains){

        read next string;

    }
    sort array of pointers;
    print array of pointers;
    exit;
```

سنستخدم بعض الدوال في برنامجنا أيضًا:

```
char *next_string(char *destination)
```

تقرأ الدالة السابقة سطرًا من المحارف بحيث ينتهي السطر بالمحرف `\n` من دخل البرنامج، وتُسند المحارف البالغ عددها `MAXLEN-1` إلى المصفوفة المُشار إليها بالمصفوفة الهدف `destination`، إذ يمثّل `MAXLEN` قيمةً ثابتةً لطول السلسلة النصية العظمى.

إذا كان المحرف الأول المقروء هو EOF (أي نهاية الملف)، أعد مؤشرًا إلى الفراغ، وفيما عدا ذلك أعد عنوان بداية السلسلة النصية (الهدف destination)، بحيث تحتوي السلسلة النصية الهدف دائمًا على المحرف \n، الذي يشير إلى نهاية السلسلة.

```
void sort_arr(const char *p_array[])
```

تمثل المصفوفة p_array[] مصفوفة المؤشرات التي تشير للمحارف، ويمكن أن تكون المصفوفة كبيرة الحجم ولكن يُشار إلى نهايتها بأول عنصر يحتوي على مؤشر فراغ null pointer.

ترتب الدالة sort_arr المؤشرات بحيث تُشير إلى السلاسل النصية المرتبة أجددًا عند اجتياز مصفوفة المؤشرات بناءً على دليل index المؤشر.

```
void print_arr(const char *p_array[])
```

تُشابه دالة print_arr الدالة sort_arr ولكنها تطبع السلاسل النصية حسب ترتيبها الأبجدي.

تذكر أنه يجري تحويل اسم المصفوفة إلى عنوانها وعنصرها الأول في أي تعبير يحتوي على اسمها، ومن شأن ذلك أن يساعدك في فهم الأمثلة على نحو أفضل؛ والأمر مماثلٌ بالنسبة لمصفوفة ثنائية البعد، مثل مصفوفة strings في المثال التالي، فنوع التعبير strings[1][2] هو char، ولكن للعنصر strings[1] نوع "مصفوفة من char"، ولذلك يُحوّل اسم المصفوفة إلى عنوان العنصر الأول ونحصل على strings[1][0].

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSTRING    50    /* العدد الأعظمي للسلاسل النصية */
#define MAXLEN       80    /* الطول الأعظمي لكل سلسلة نصية */

void print_arr(const char *p_array[]);
void sort_arr(const char *p_array[]);
char *next_string(char *destination);

main(){
    /* نصح عن المصفوفة مع إضافة عنصر فارغ في نهايتها */
    char *p_array[MAXSTRING+1];

    /* مصفوفة تخزين السلاسل النصية */
```

```
char strings[MAXSTRING][MAXLEN];

/* عدد السلاسل النصية المقروءة */
int nstrings;

nstrings = 0;
while(nstrings < MAXSTRING &&
      next_string(strings[nstrings]) != 0){

    p_array[nstrings] = strings[nstrings];
    nstrings++;
}

/* إعدام قيمة المصفوفة */
p_array[nstrings] = 0;

sort_arr(p_array);
print_arr(p_array);
exit(EXIT_SUCCESS);
}

void print_arr(const char *p_array[]){
    int index;
    for(index = 0; p_array[index] != 0; index++){
        printf("%s\n", p_array[index]);
    }
}

void sort_arr(const char *p_array[]){
    int comp_val, low_index, hi_index;
    const char *tmp;

    for(low_index = 0;
        p_array[low_index] != 0 &&
        p_array[low_index+1] != 0;
        low_index++){
```

```

        for(hi_index = low_index+1;
            p_array[hi_index] != 0;
            hi_index++){

            comp_val=strcmp(p_array[hi_index],
                p_array[low_index]);
            if(comp_val >= 0)
                continue;
            /* التبدل بين السلسلتين النصيتين */
            tmp = p_array[hi_index];
            p_array[hi_index] = p_array[low_index];
            p_array[low_index] = tmp;

        }
    }
}

char *next_string(char *destination){
    char *cp;
    int c;

    cp = destination;
    while((c = getchar()) != '\n' && c != EOF){
        if(cp-destination < MAXLEN-1)
            *cp++ = c;
    }
    *cp = 0;
    if(c == EOF && cp == destination)
        return(0);
    return(destination);
}

```

[مثال 10]

إعادة الدالة next_string لمؤشر ليس من قبيل المصادفة، إذ أصبح بإمكاننا الآن الاستغناء عن استخدام مصفوفة السلاسل النصية واستخدام next_string لحجز مساحة التخزين الموافقة لها.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSTRING      50      /* العدد الأعظمي للسلاسل النصية */
#define MAXLEN        80      /* الطول الأعظمي لكل سلسلة نصية */

void print_arr(const char *p_array[]);
void sort_arr(const char *p_array[]);
char *next_string(void);

main(){
    char *p_array[MAXSTRING+1];
    int nstrings;

    nstrings = 0;
    while(nstrings < MAXSTRING &&
          (p_array[nstrings] = next_string()) != 0){

        nstrings++;
    }
    /* إعدام قيمة المصفوفة */
    p_array[nstrings] = 0;

    sort_arr(p_array);
    print_arr(p_array);
    exit(EXIT_SUCCESS);
}

void print_arr(const char *p_array[]){
    int index;
    for(index = 0; p_array[index] != 0; index++){
        printf("%s\n", p_array[index]);
    }
}
```

```
void sort_arr(const char *p_array[]){
    int comp_val, low_index, hi_index;
    const char *tmp;

    for(low_index = 0;
        p_array[low_index] != 0 &&
        p_array[low_index+1] != 0;
        low_index++){

        for(hi_index = low_index+1;
            p_array[hi_index] != 0;
            hi_index++){

            comp_val=strcmp(p_array[hi_index],
                p_array[low_index]);
            if(comp_val >= 0)
                continue;
            /* التبدیل بین السلسلتین النصیتین */
            tmp = p_array[hi_index];
            p_array[hi_index] = p_array[low_index];
            p_array[low_index] = tmp;
        }
    }
}

char *next_string(void){
    char *cp, *destination;
    int c;

    destination = (char *)malloc(MAXLEN);
    if(destination != 0){
        cp = destination;
        while((c = getchar()) != '\n' && c != EOF){
            if(cp-destination < MAXLEN-1)
                *cp++ = c;
        }
    }
}
```



```

        *cp = 0;
        if(c == EOF && cp == destination)
            return(0);
    }
    return(destination);
}

```

[مثال 11]

وأخيراً إليك المثال كاملاً مع استخدام مصفوفة `p_array` للدالة `malloc`، ولاحظ إعادة كتابة معظم أدلة المصفوفة لتستخدم ترميز المؤشرات. إذا كنت تشعر بالإرهاق من جميع المعلومات التي قرأتها فتجاوز المثال التالي، فهو صعبٌ بعض الشيء.

شرح المثال: تعني `char **p` مؤشرًا يشير إلى المؤشر الذي يشير إلى محرف، ويوجد معظم مبرمجو لغة سي هذه الطريقة في استخدام المؤشرات صعبة الفهم.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSTRING    50    /* العدد الأعظمي للسلاسل النصية */
#define MAXLEN      80    /* الطول الأعظمي لكل سلسلة نصية */

void print_arr(const char **p_array);
void sort_arr(const char **p_array);
char *next_string(void);

main(){
    char **p_array;
    int nstrings;    /* عدد السلاسل النصية المقروءة */

    p_array = (char **)malloc(
                sizeof(char *[MAXSTRING+1]));
    if(p_array == 0){
        printf("No memory\n");
        exit(EXIT_FAILURE);
    }
}

```

```
nstrings = 0;
while(nstrings < MAXSTRING &&
      (p_array[nstrings] = next_string()) != 0){

    nstrings++;
}
/* إعدام قيمة المصفوفة */
p_array[nstrings] = 0;

sort_arr(p_array);
print_arr(p_array);
exit(EXIT_SUCCESS);
}

void print_arr(const char **p_array){
    while(*p_array)
        printf("%s\n", *p_array++);
}

void sort_arr(const char **p_array){
    const char **lo_p, **hi_p, *tmp;

    for(lo_p = p_array;
        *lo_p != 0 && *(lo_p+1) != 0;
        lo_p++){
        for(hi_p = lo_p+1; *hi_p != 0; hi_p++){

            if(strcmp(*hi_p, *lo_p) >= 0)
                continue;
            /* التبدل بين السلسلتين النصيتين */
            tmp = *hi_p;
            *hi_p = *lo_p;
            *lo_p = tmp;
        }
    }
}
```

```

    }
}

char *next_string(void){
    char *cp, *destination;
    int c;

    destination = (char *)malloc(MAXLEN);
    if(destination != 0){
        cp = destination;
        while((c = getchar()) != '\n' && c != EOF){
            if(cp-destination < MAXLEN-1)
                *cp++ = c;
        }
        *cp = 0;
        if(c == EOF && cp == destination)
            return(0);
    }
    return(destination);
}

```

[مثال 12]

سنستعرض مثالاً آخر لتوضيح استخدام دالة `malloc` وإمكاناتها في التعامل مع السلاسل النصية الطويلة؛ إذ يقرأ المثال السلاسل النصية من الدخل ويبحث عن محرف سطر جديد لتحديد نهاية السلسلة النصية (أي `\n`)، ثم يطبع السلسلة النصية إلى الخرج، ويتوقف البرنامج عن العمل عندما يصادف محرف نهاية الملف `EOF`. تُسند المحارف إلى مصفوفة، ويُدلّ على نهاية المصفوفة -كما هو معتاد- بالقيمة صفر، مع ملاحظة أن محرف السطر الجديد لا يُخزّن بالمصفوفة بل يُستخدم فقط لتحديد سطر الدخل الواجب طباعته للخرج. لا يعلم البرنامج طول السلسلة النصية تحديداً، ولذلك يبدأ بفحص كل عشرة محارف وحجز المساحة الخاصة بهم (الثابت `GROW_BY`).

تُستدعى الدالة `malloc` في حال كانت السلسلة النصية أطول من عشرة محارف لحجز المساحة للسلسلة النصية وإضافة عشرة محارف أخرى، ثم تُنسخ المحارف الحالية للمساحة الجديدة وتُستخدم من البرنامج وتُحرّر المساحة القديمة.

تُستخدم الدالة `free` لتحرير المساحة القديمة المحجوزة من `malloc` مسبقًا، إذ يجب عليك تحرير المساحة غير المُستخدمة بعد الآن دوريًا قبل أن تتراكم، واستخدام `free` يحزّر المساحة ويسمح بإعادة استخدامها لاحقًا.

يستخدم البرنامج الدالة `fprintf` لعرض أي أخطاء، وهي دالةٌ مشابهة للدالة `printf` التي اعتدنا على رؤيتها، والفرق الوحيد بينهما هو أن الدالة `fprintf` تأخذ وسيطًا إضافيًا يدل على وسيط الخرج الذي سيُطبع إليه، وهناك ثابتان لهذا الغرض معرّفان في ملف الترويسة `stdio.h`؛ إذ أن استخدام الثابت الأول `stdout` يعني استخدام خرج البرنامج القياسي، بينما يشير استخدام الثابت الثاني `stderr` إلى مجرى أخطاء البرنامج القياسي `standard error stream`، وقد يكون وسيط الخرج متماثلين في بعض الأنظمة إلا أن بعض الأنظمة الأخرى تفصل بين الاثنين.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define GROW_BY 10      /* يزداد حجم السلسلة النصية كل مرة بمقدار 10 */

main(){
    char *str_p, *next_p, *tmp_p;
    int ch, need, chars_read;

    if(GROW_BY < 2){
        fprintf(stderr,
                "Growth constant too small\n");
        exit(EXIT_FAILURE);
    }

    str_p = (char *)malloc(GROW_BY);
    if(str_p == NULL){
        fprintf(stderr, "No initial store\n");
        exit(EXIT_FAILURE);
    }

    next_p = str_p;
    chars_read = 0;
    while((ch = getchar()) != EOF){
```

```

/* (*) */

if(ch == '\n'){
    /* الإشارة إلى نهاية السطر */
    *next_p = 0;
    printf("%s\n", str_p);
    free(str_p);
    chars_read = 0;
    str_p = (char *)malloc(GROW_BY);
    if(str_p == NULL){
        fprintf(stderr, "No initial store\n");
        exit(EXIT_FAILURE);
    }
    next_p = str_p;
    continue;
}
/*
 * التحقق من وصولنا إلى نهاية المساحة المحجوزة
 */
if(chars_read == GROW_BY-1){
    *next_p = 0;    /* للدلالة على نهاية السلسلة النصية */
    /* نستخدم الطرح بين المؤشرات لإيجاد طول السلسلة النصية الحالية */
    need = next_p - str_p + 1;
    tmp_p = (char *)malloc(need+GROW_BY);
    if(tmp_p == NULL){
        fprintf(stderr, "No more store\n");
        exit(EXIT_FAILURE);
    }
    /*
     * نسخ السلسلة النصية باستخدام دالة المكتبة
     */
    strcpy(tmp_p, str_p);
    free(str_p);
    str_p = tmp_p;
    /*
     * إعادة ضبط next_p
     */
}

```

```

        next_p = str_p + need-1;
        chars_read = 0;
    }
    /*
     * إسناد المحرف إلى نهاية السلسلة النصية
     */
    *next_p++ = ch;
    chars_read++;
}
/*
 * عند وصولنا إلى نهاية الملف
 * هل توجد محارف غير مطبوعة؟
 */
if(str_p - next_p){
    *next_p = 0;
    fprintf(stderr, "Incomplete last line\n");
    printf("%s\n", str_p);
}
exit(EXIT_SUCCESS);
}

```

[مثال 13]

(*) تُعاد الحلقة في الموضع المذكور عند كل سطر، وهناك مساحةٌ للعنصر صفر في نهاية السلسلة النصية دائماً، لأننا نتحقق من أصغر من 2 وهو ما تحققنا منه سابقاً GROW_BY ذلك في الشرط التالي إلا في حال كان.

قد لا يكون برنامجنا السابق مثلاً واقعياً عن التعامل مع السلاسل النصية الطويلة، إذ يتطلب حجم التخزين الأعظمي **ضعف** الحجم المطلوب لأطول سلسلة نصية، ولكنه برنامج يعمل صحيحاً بغض النظر، إلا أنه يكلفنا الكثير بخصوص الموارد بنسخ السلاسل النصية ويمكن حل المشكلتين عن طريق استخدام دالة `realloc`.

نستطيع استخدام القوائم المترابطة لطريقة أكثر تعقيداً، مع استخدام **الهياكل Structures** التي سنتكلم عنها لاحقاً، إلا أن هذه الطريقة تأتي أيضاً ببعض المشكلات لأن دوال المكتبة القياسية لن تعمل عند استخدام طريقة مغايرة لتخزين السلاسل النصية.

5.5.1 ما الأشياء التي لا يستطيع العامل sizeof فعلها؟

يرتكب المبتدئون غالبًا الخطأ التالي عند استخدام العامل sizeof:

```
#include <stdio.h>
#include <stdlib.h>

const char arr[] = "hello";
const char *cp = arr;
main(){

    printf("Size of arr %lu\n", (unsigned long)
           sizeof(arr));
    printf("Size of *cp %lu\n", (unsigned long)
           sizeof(*cp));
    exit(EXIT_SUCCESS);
}
```

[مثال 14]

لن تكون الأرقام ذاتها عند الطباعة، إذ سيعرف أولاً حجم arr بكونها 6 بصورة صحيحة (خمسة محارف متبوعاً بمحرف الفراغ null)، بينما ستطبع التعليمة الثانية -على جميع الأنظمة- القيمة 1، لأن المؤشر *cp من نوع const char ذو الحجم 1 بايت، بينما arr مختلفة فهي مصفوفة من نوع const char. تسبب هذه المشكلة مصدرًا للحيرة، إذ أن هذه الحالة الوحيدة التي لا يجري فيها تحويل المصفوفة إلى مؤشر أولاً، فمن المستحيل استخدام sizeof لإيجاد طول مصفوفة باستخدام مؤشر يشير إليها، ويجب عليك استخدام اسم المصفوفة حصراً.

5.5.2 نوع قيمة sizeof

لعلك تتساءل الآن عن نتيجة التالي:

```
sizeof ( sizeof (anything legal) )
```

فما هو نوع نتيجة عامل sizeof؟ الإجابة على هذا السؤال معروفة بحسب التطبيق، وقد تكون unsigned long أو unsigned int بحسب تطبيقك، إلا أن هناك شيئين يمكن فعلهما للتأكد من أنك تستخدم القيمة بصورة صحيحة، وهما:

- يمكنك استخدام تحويل الأنواع cast وتحويل القيمة إلى unsigned long قسرًا (كما فعلنا في المثال السابق).
- يمكنك استخدام النوع المُعرّف size_t الموجود في ملف الترويسة stddef.h كما يوضح المثال التالي:

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

main(){
    size_t sz;
    sz = sizeof(sz);
    printf("size of sizeof is %lu\n",
        (unsigned long)sz);
    exit(EXIT_SUCCESS);
}
```

[مثال 15]

5.6 مؤشرات الدوال

من المفيد أن يكون لدينا إمكانية استخدام المؤشرات على الدوال، كما أن التصريح عن هذا النوع من المؤشرات سهل عن طريق كتابته وكأنك تصرّح عن دالة على النحو التالي:

```
int func(int a, float b);
```

ومن ثم إضافة قوسين حول اسم الدالة والرمز * أمامه، مما يدل على أن هذا التصريح يعود لمؤشر. لاحظ أن التخلي عن القوسين يتسبب بالتصريح عن دالة تُعيد مؤشرًا حسب قوانين الأسبقية:

```
/* دالة تعيد مؤشرًا إلى قيمة صحيحة int */
int *func(int a, float b);

/* int* مؤشر إلى دالة تعيد قيمة صحيحة */
int (*func)(int a, float b);
```

حالما تحصل على المؤشر تستطيع إسناد العنوان إلى النوع المحدد للدالة باستخدام اسمها، إذ يُحوّل اسم الدالة إلى عنوان في أي تعبير تحتويه بصورة مشابهة لاسم المصفوفة، ويمكنك استدعاء الدالة في هذه الحالة باستخدام إحدى الطريقتين:


```
(*func)(1,2);
/* or */
func(1,2);
```

تفضّل لغة سي المعيارية الطريقة الثانية، إليك مثالاً بسيطاً عنها:

```
#include <stdio.h>
#include <stdlib.h>

void func(int);

main(){
    void (*fp)(int);

    fp = func;

    (*fp)(1);
    fp(2);

    exit(EXIT_SUCCESS);
}

void
func(int arg){
    printf("%d\n", arg);
}
```

[مثال 16]

يمكنك توظيف مصفوفة من المؤشرات التي تشير إلى مصفوفات مختلفة إذا أردت كتابة آلة محدودة الحالات finite state machine، وسيبدو التصريح عنها مماثلاً لما يلي:

```
void (*fparr[])(int, float) = {
    /* المهيئات initializers*/
};

/* استدعاء أحد القيم */
fparr[5](1, 3.4);
```

[مثال 17]

ولكننا لن نتكلم عن هذه الطريقة.

5.7 المؤشرات في التعابير

بعد إدخال الأنواع المؤهلة `qualified types` ومفهوم الأنواع غير المُكتملة `incomplete types` مع استخدام مؤشر الفراغ `* void`، أصبح هناك بعض القواعد المعقدة عن مزج المؤشرات وما هو مسموح لك فعليًا في العمليات الحسابية معها. قد تستطيع تجاوز هذه القواعد دون أي مشكلات، لأن معظمها "بديهي" ولكننا سنتكلم عنها بغض النظر عن ذلك، ولا شك أنك سترغب بقراءة معيار لغة سي لتحريّ الدقة، لأن ما سيأتي هو تفسير بلغة بسيطة لما ورد في المعيار.

لعلك لا تعلم بدقة ما الذي يقصده المعيار عندما يذكر مصطلحي **الكائنات objects** و**الأنواع غير المُكتملة incomplete types**، فقد استخدمنا هذه المصطلحات حتى الآن بتهاون. يُعد الكائن جزءًا من البيانات المخزنة التي يمكن تفسير محتوياتها إلى قيمة، وبناءً على ذلك فالدالة ليست كائنًا؛ بينما يُعرف النوع غير المكتمل بكونه نوعًا معروفًا ذا اسم معروف لكن دون حجم محدد بعد، ويمكنك الحصول على هذا النوع عن طريق وسيلتين، هما:

1. التصريح عن مصفوفة دون تحديد حجمها: `int x[]`؛ ويجب توفير المزيد من المعلومات بخصوص هذه المصفوفة في التعريف لاحقًا، ويبقى النوع غير مُكتملاً حتى الوصول لنقطة التعريف.
 2. التصريح عن هيكل **Structure** أو **اتحاد Union** دون التعريف عن محتوياته، ويجب التعريف عن محتوياته لاحقًا في هذه الحالة، ويبقى النوع غير مُكتملاً حتى الوصول لنقطة التعريف.
- سنناقش المزيد عن الأنواع غير المكتملة لاحقًا.

5.7.1 التحويلات

يمكن تحويل المؤشرات التي تشير إلى `void` إلى مؤشرات تشير إلى أي كائن أو نوع غير مكتمل، وتحصل على قيمةٍ مساوية لقيمة المؤشر الأصل بعد تحويل مؤشر يشير إلى كائن أو نوع غير مكتمل إلى مؤشر من نوع `* void`:

```
int i;
int *ip;
void *vp;

ip = &i;
vp = ip;
ip = vp;
if(ip != &i)
```

```
printf("Compiler error\n");
```

يمكن تحويل مؤشر من نوع غير مؤهل unqualified إلى مؤشر من نوع مؤهل، ولكن العكس غير ممكن، وستكون قيمة المؤشرين متكافئتين:

```
int i;
int *ip;
const int *cpi;

ip = &i;
cpi = ip;      /* مسموح */
if(cpi != ip)
    printf("Compiler error\n");
ip = cpi;      /* ممنوع */
```

لا يساوي مؤشر ثابت فارغ null pointer constant (سنتكلم عن هذا النوع لاحقاً) أيّ مؤشر يشير لأي كائن أو دالة.

5.7.2 العمليات الحسابية

يمكن للتعبير Expressions أن تجمع (أو تطرح، وهو ما يكافئ جمع قيم سالبة) أعداداً صحيحةً إلى قيمة المؤشرات بغض النظر عن نوع الكائن الذي تشير إليه، وتكون النتيجة مماثلةً لنوع المؤشر؛ وفي حالة إضافة القيمة n ، فسيشير المؤشر إلى العنصر الذي يلي العنصر السابق ضمن المصفوفة بمقدار n ، والاستخدام الأكثر شيوعاً لهذه الميزة هي إضافة 1 إلى المؤشر لتمريره على المصفوفة من بدايتها إلى نهايتها، إلا أن استخدام قيم مغايرة للقيمة 1 والطرح بدلاً من الجمع ممكن.

نحصل على حالة طفحان overflow أو طفحان تجاوز الحد الأدنى underflow إذا كان المؤشر الناتج عن عملية الجمع يشير إلى ما يسبق المصفوفة أو ما يلي العنصر المعلوم الأخير للمصفوفة، وهذا يعني أن النتيجة غير مُعرّفة.

يملك العنصر الأخير الزائد في المصفوفة عنواناً صالحاً، ويؤكد المعيار لنا ذلك، إلا أنه ليس من المفترض أن تحاول الوصول إلى ذلك العنصر، وعنوانه موجودٌ للتأكد من وجوده لتجنّب الوقوع في حالة طفحان.

تعتمدنا استخدام الكلمة "تعبير" عوضاً عن قولنا "إضافة قيمة إلى المؤشر بنفسه"، إلا أنه يمكنك فعل ذلك شرط ألا يكون المؤشر مؤهلاً بالكلمة المفتاحية "const"، ويكافئ طبعاً استخدام عامل الزيادة "++" وعامل النقصان "--" جمع أو طرح واحد.

يمكن طرح مؤشرين من أنواع متوافقة **compatible types** أو غير مؤهلة من بعضهما بعضًا، وتكون النتيجة من النوع "ptrdiff_t"، المعرّف في ملف الترويسة `stddef.h`، إلا أنه يجب أن يشير كلا المؤشرين إلى المصفوفة ذاتها، أو على الأقل أن يشير واحدًا منها إلى ما بعد أو قبل المصفوفة، وإلا سنحصل على سلوك غير محدد، وتكون نتيجة عملية الطرح هي عدد العناصر التي تفصل المؤشرين ضمن المصفوفة. إليك المثال التالي:

```
int x[100];
int *pi, *cpi = &x[99]; /* يشير cpi إلى العنصر الأخير من ال x */

pi = x;
if((cpi - pi) != 99)
    printf("Error\n");

pi = cpi;
pi++; /* increment past end of x */
if((pi - cpi) != 1)
    printf("Error\n");
```

5.7.3 التعابير العلاقية

تسمح لنا التعابير العلاقية بالمقارنة بين المؤشرات، لكن يمكنك فقط مقارنة:

- المؤشرات التي تشير لكائنات ذات أنواع متوافقة مع بعضها الآخر.
- المؤشرات التي تشير لأنواع غير مكتملة متوافقة مع بعضها الآخر.

ولا يهم إذا كانت الأنواع المُشارَة إليها مؤهلة أو غير مؤهلة.

إذا تساوت قيم مؤشرين، فهذا يعني أن المؤشرين يشيران إلى الشيء ذاته، سواءً كان هذا الشيء كائنًا أو عنصرًا غير موجودًا خارج مصفوفة ما (راجع فقرة العمليات الحسابية أعلاه). تقدّم العوامل العلاقية "<" و "<=" وغيرها النتيجة التي تتوقعها عند استخدامها مع المؤشرات ضمن نفس المصفوفة، وإذا كانت قيمة أحد المؤشرات أقل مقارنةً مع الآخر، فهذا يعني أنه يشير لقيمةٍ أقرب لمقدمة المصفوفة (العنصر ذو الدليل index الأقل).

يمكن إسناد مؤشر فارغ ثابت إلى مؤشر آخر، وسيكون متساوي مع مؤشر فارغ ثابت آخر إذا فحصناهما باستخدام عامل المقارنة، بينما لن يتساوى مؤشر فارغ ثابت أو غير ثابت عند مقارنتهما مع أي مؤشر آخر يشير لشيء ما.

5.7.4 الإسناد

يمكنك استخدام المؤشرات مع عوامل الإسناد، شرط أن يستوفي الاستخدام الشروط التالية:

- يجب أن يكون الجانب الأيسر من عامل الإسناد مؤشراً، وأن يكون الجانب الأيمن منه مؤشراً فارغاً ثابتاً.
- يجب أن يكون مُعاملٌ من المعاملات مؤشراً يشير إلى كائن أو نوع غير مُكتمل، والمعامل الآخر مؤشراً إلى الفراغ "void"، سواءً كان مؤهلاً أو لا.
- يُعدّ المُعاملان مؤشرين لأنواع متوافقة سواءً كانت مؤهلاً أم لا.

يجب أن يكون للنوع المُشار إليه في الحالتين الأخيرتين على الجانب الأيسر من عامل الإسناد النوع ذاته من المؤهلات على الأقل، والموافق لمؤهل النوع الواقع على الجانب الأيمن من عامل الإسناد، أو أكثر من مؤهل مماثل.

يمكنك إذاً إسناد مؤشر يشير إلى قيمة صحيحة "int" إلى مؤشر يشير إلى قيمة من نوع عدد صحيح ثابت "const int" (مؤهلات النوع الأيسر تزيد عن مؤهلات النوع الأيمن) ولكن لا يمكنك إسناد مؤشر يشير إلى "const int" إلى مؤشر يشير إلى "int"، والأمر منطقي جداً إذا أخذت لحظةً للتفكير به.

يمكن استخدام العاملين "+=" و "-=" مع المؤشرات طالما أن الجانب الأيسر من العامل مؤشر يشير إلى كائن، والجانب الأيمن من العامل تعبير ينتج قيمةً صحيحةً "integral"، وتوضح قوانين العمليات الحسابية في الفقرات السابقة ما سيحصل في هذه الحالة.

5.7.5 العامل الشرطي

وَصَّحنا سابقاً سلوك العامل الشرطي conditional operator عند استخدامه مع المؤشرات.

5.8 المصفوفات وعامل & والدوال

ذكرنا عدّة مرات أنه يجري تحويل اسم المصفوفة إلى عنوانها وعنصرها الأول، وقلنا أن الاستثناء الوحيد هو عند استخدام اسم المصفوفة مع عامل "sizeof"، وهو عاملٌ مهمٌ إذا أردت استخدام الدالة "malloc"، إلا أن هناك استثناءً آخر، ألا وهو عندما يكون اسم المصفوفة مُعاملاً لعامل "&" (عنوان العامل)، إذ يُحوّل اسم المصفوفة هنا إلى عنوان **كامل** المصفوفة بدلاً من عنوان عنصرها الأول عادةً، لكن ما الفرق؟ لعلك تعتقد أن العنوانين متماثلان، إلا أن الفرق هو نوعهما، فبالنسبة لمصفوفةٍ تحتوي "n" عنصر بنوع "T"، يكون عنوان عنصرها الأول من نوع "مؤشر إلى T"، بينما يكون عنوان كامل المصفوفة من نوع "مؤشر إلى مصفوفة من n عنصر من نوع T"، وهو مختلف جداً، إليك مثالاً عن ذلك:

```
int ar[10];
int *ip;
int (*ar10i)[10]; /* مؤشر لمصفوفة من 10 عناصر صحيحة */

ip = ar; /* عنوان العنصر الأول */
ip = &ar[0]; /* عنوان العنصر الأول */
ar10i = &ar; /* عنوان كامل المصفوفة */
```

أين تُستخدم المؤشرات إلى المصفوفات؟ في الحقيقة ليس غالبًا، إلا أننا نعلم أن التصريح عن مصفوفة متعددة الأبعاد هو في الحقيقة تصريح عن مصفوفة مصفوفات. إليك مثالًا يستخدم هذا المفهوم (إلا أن فهم ما يفعل يقع على عاتقك)، وليس من الشائع استخدام هذه الطريقة:

```
int ar2d[5][4];
int (*ar4i)[4]; /* مؤشر إلى مصفوفة من 4 أعداد صحيحة */

for(ar4i= ar2d; ar4i < &(ar2d[5]); ar4i++)
    (*ar4i)[2] = 0; /* ar2d[n][2] = 0 */
```

ما قد يثير اهتمامك أكثر من عناوين المصفوفات هو ما الذي قد يحدث عندما نصرِّح عن دالة تأخذ مصفوفةً في أحد وسطائها. بالنظر إلى أن المصفوفة تحوّل إلى عنوان عنصرها الأول فحتى لو حاولت تمرير مصفوفة إلى دالة باستخدام اسم المصفوفة وسيطًا، فسينتهي بك الأمر بتمرير مؤشر إلى عنصر المصفوفة الأول. لكن ماذا لو صرّحت عن الدالة بكونها تأخذ وسيطًا من نوع "مصفوفة من نوع ما" على النحو التالي:

```
void f(int ar[10]);
```

ما الذي يحدث في هذه الحالة؟ قد تفاجئك الإجابة هنا، إذ أن المصرّف ينظر إلى السطر السابق ويقول لنفسه "سيكون هذا مؤشرًا لهذه المصفوفة" ويعيد كتابة الوسيط على أنه من نوع مؤشر، ووفقًا لذلك نجد أن التصريحات الثلاثة التالية متكافئة:

```
void f(int ar[10]);
void f(int *ar);
void f(int ar[]); /* حجم المصفوفة هنا لا علاقة له! */
```

قد تضع يدك على رأسك بعد هذه المعلومة، لكن تمهّل! إليك بعض الأسئلة للتهديئة من غضبك وإحباطك:

- لم كانت المعلومة السابقة منطقية؟
- لماذا تعمل التعابير بالصياغة `ar[5]` أو أي صياغةٍ أخرى ضمن التصريح عن دالة، ثم داخل الدالة كما هو متوقَّع منها؟

فكّر في الأسئلة السابقة، وستفهم استخدام المؤشرات مع المصفوفات بصورة ممتازة عندما تتوصل لإجابة ترضيك.

5.9 خاتمة

قدّم هذا الفصل شرحًا عن المصفوفات والمؤشرات ومخصّصات المساحة، وسنجد أن المفهوم الأخير عظيم الفائدة في الفصول القادمة، بينما ستبرز أهمية المفهومين الآخرين بالنسبة للغة سي C ككل.

لا يمكنك استخدام لغة سي بصورة صحيحة دون فهم استخدام المؤشرات. تتصف المصفوفات بالسهولة باستثناء حالة استخدامها ضمن تعبير ما، وعندها سيحوّل اسم المصفوفة إلى مؤشر يشير إلى عنصرها الأول (نُعيد هذا الأمر لضرورته، فتذكره جيّدًا).

قد يتفاجأ بعض الناس من السلوك الذي تسلكه لغة سي للتعامل مع السلاسل النصية، إلا أنه مرّنٌ وجيّدٌ، خصوصًا محرف الفراغ الزائد الذي يُنهي المصفوفة، وقد يدفع هذا الاعتقاد بعض الناس بالنظر إلى لغة سي بكونها لغةً تركز على المحارف دونًا عن غيرها لافتقادها أدوات التعامل مع السلاسل النصية والتلاعب فيها وسيكون هذا الاعتقاد صائبًا، وتُعد هذه النقطة في صالح لغة سي مقارنةً بالحلول التي تتبناها لغات البرمجة الأخرى بخصوص سرعة التنفيذ، إلا أن الأمر يصعب من مهمة المبرمج لا شك.

العمليات الحسابية على المؤشرات سهلةٌ ومنطقية، ويعدّ تعلمها تحدّيًا لمبرمج اعتاد العمل مع لغة تجميعيّة، نظرًا لاعتياده على ترجمة التعبير إلى ما تفعله الآلة (بحكم ممارسته)، إلا أن تعلمها أصعب بكثير على الناس الذين ليس لهم أي خبرة أو علم بأنواع المؤشرات المختلفة غير المتكافئة. حاول التخلص من فكرة أن المؤشرات تحتوي على عناوين (بمفهوم العناد الصلب) وسيسهل عليك فهم الأمر.

تُعد طريقة الحصول على مساحات تخزينية ذات سعة محددة باستخدام "malloc" وما يترتب على هذه الطريقة مهمًّا جدًّا، وربما تختار تفادي هذه الطريقة للوقت الحالي، لكن تجنّب إهمال هذا الجانب من اللغة، فالسمة الواضحة لمبرمجي لغة سي الهواة هي استخدامهم مصفوفات ذات سعة محدّدة مسبقًا، إذ تعطي الدالة "malloc" وسيلةً مرنةً لحجز المساحة، ويستحق الأمر تعلمها وإتقانها بلا شك.

ستساعدك الأمثلة المذكورة في هذا الفصل على فهم استخدام "sizeof" وتصحيح بعض المفاهيم الخاطئة عن هذا العامل وما الذي يفعله، ولعلك لن تستخدمه على نحوٍ متكرّر إلا أنه الخيار الوحيد عندما تحتاج إليه، ولا بديل عنه.

5.10 تمارين

1. ما النطاق الصالح لأدلة مصفوفة تحتوي على عشر كائنات (عناصر)؟
2. ما الذي سيحدث إذا حصلت على قيمة عنوان العنصر الحادي عشر للمصفوفة السابقة؟
3. ما هي الحالات الصالحة التي يمكننا مقارنة مؤشرين فيها؟
4. ما استخدام المؤشر من نوع void؟
5. اكتب دوالاً تحقق التالي:
 1. دالة تقارن سلسلتين نصيتين وتبحث عن المساواة، وتعيد القيمة "0" إن تساوت السلسلتين، أو تُعيد فرق القيمة بين أول محرفين غير متماثلين عدا ذلك.
 2. دالة تجد أول ظهور لمحرف في سلسلة نصية ما، وتعيد مؤشرًا إلى المحرف ضمن السلسلة، أو القيمة صفر إن لم يوجد المحرف المطلوب ضمن السلسلة.
 3. دالة تأخذ سلسلتين نصيتين وسيطين لها، وتعيد مؤشرًا في حال كانت السلسلة النصية الأولى مُحتواة داخل السلسلة النصية الثانية مثل سلسلة فرعية substring، بحيث يشير المؤشر إلى أول ظهور للسلسلة، أو إعادة القيمة صفر في حال عدم إيجاد نتيجة.
6. اشرح الأمثلة التي وردت سابقًا باستخدام الدالة "malloc" لشخص آخر.

دورة إدارة تطوير المنتجات



تعلم تحويل أفكارك لمنتجات ومشاريع حقيقية بدءًا من دراسة السوق وتحليل المنافسين وحتى إطلاق منتج مميز وناجح

التحق بالدورة الآن



6. هياكل البيانات

6.1 لمحة تاريخية

اتجه تطوير لغات الحاسوب سابقًا في اتجاهٍ من اتجاهين، إذ سلكت كوبول COBOL سلوكًا ركز على استخدام هياكل البيانات بعيدًا عن العمليات الحسابية والخوارزميات، بينما سلكت لغات مثل فورتران FORTRAN وألغول Algol سلوكًا معاكسًا. أراد العلماء وقتها إجراء العمليات الحسابية باستخدام بيانات غير مُهيكلَة نسبيًا، إلا أنه سرعان ما لاحظ الجميع أن استخدام المصفوفات لا غنى عنه؛ بينما أراد المستخدمون الاعتياديون طريقةً لإجراء العمليات الحسابية البسيطة فقط، إلا أن طريقة هيكلة البيانات كانت عائقًا أمام تحقيق ذلك.

أثر كلا السلوكين في تصميم لغة سي، إذ أنها تحتوي تحكّمًا هيكليًا لتدفق البرنامج مناسب للغة من هذا العمر، كما أنها جعلت من مفهوم هياكل البيانات شائعًا. ركزنا على جانب الخوارزميات من اللغة حتى اللحظة، ولم نولي الكثير من الانتباه بخصوص تخزين البيانات، ومع أننا تطرقنا إلى المصفوفات التي تُعدّ هيكل بيانات إلا أنها شائعة الاستخدام وبسيطة ولا تستحق فصلًا مخصصًا لها، واكتفينا إلى الآن بالنظر إلى اللغة انطلاقًا من بيئة هيكليّة شبيهة بلغة فورتران.

كان استخدام كلٍ من البيانات والخوارزميات هو التوجه الأكثر رواجًا في أواخر ثمانينيات وبداية تسعينيات القرن الماضي، وفق ما يُدعى بالبرمجة كائنية التوجه Object-Oriented Programming. لا يوجد أي دعم لهذه الطريقة في لغة سي، إلا أن لغة ++C قدمت دعمًا لها (وهي لغةٌ مبنيةٌ على لغة سي)، ولكن هذا النقاش خارج موضوعنا حاليًا.

تأخذ البيانات الانتباه الأكبر لمعظم مشاكل الحوسبة المتقدمة وليس الخوارزميات، فستكون مهمتك بسيطةً ببرمجة البرنامج إن استطعت تصميم هياكل بيانات صحيحة ومناسبة، إلا أنك تحتاج إلى دعمٍ من اللغة

في هذه الحالة، فمهمتك ستصبح أقل سهولة ومعرضةً أكثر للأخطاء إن لم يكن هناك أي دعم لأنواع هياكل البيانات المختلفة عن المصفوفات. تقع هذه المهمة على كاهل لغة البرمجة، فليس كافيًا أن تسمح لك اللغة بفعل ما تريد، بل يجب أن **تساعدك** في فعل ما تريد.

تقدم لك لغة سي سعيًا منها بتقديم هياكل بيانات مناسبة كلاً من المصفوفات Arrays والهياكل Structures والاتحادات Unions، وقد برهنت على أنها كافيةٌ لمعظم المستخدمين الاعتياديين وبالتالي لم يُضف المعيار أي جديد بشأنها.

6.2 الهياكل Structures

تسمح لك المصفوفات بتخزين مجموعة من الكائنات المتماثلة تحت اسم معين، وهذا مفيدٌ لعدد من المهام، ولكنه ليس مرناً التعامل، إذ تحتوي معظم كائنات البيانات ذات التطبيقات الواقعية على هيكل معيّن معقد لا يمكن استخدامه مع طريقة تخزين المصفوفة للبيانات.

لنوضح ما سبق بالمثل التالي: لنفرض أننا نريد تمثيل سلسلة نصية ذات خصائص معينة، بجانب محتواها. هناك نوع الخط وحجمه، وهما سمتان لا تؤثران في محتوى السلسلة، لكنهما تحددان الطريقة التي تُعرض فيها السلسلة على الشاشة سواءً كان النص **مكتوبًا بخط غامق** أو **مائل**، والأمر ذاته ينطبق على حجم الخط. كيف نستطيع تمثيل السلسلة النصية بكائن واحد ضمن مصفوفة إذا كان يحتوي على ثلاث سمات مختلفة؟

يمكننا تحقيق ذلك في لغة سي C بسهولة، حاول أولاً تمثيل السمات الثلاث باستخدام الأنواع الأساسية، فعلى فرض أنه يمكننا تخزين كل محرف باستخدام النوع char، يمكننا الإشارة إلى نوع الخط المستخدم باستخدام النوع short (نستخدم "1" للإشارة إلى الخط الاعتيادي و"2" للخط المائل و"3" للخط الغامق، وهكذا)، كما يمكننا تخزين حجم الخط باستخدام النوع short، ونُعد جميع الفرضيات السابقة معقولةً عمليًا، إذ تدعم معظم الأنظمة عددًا قليلاً من الخطوط مهما كانت هذه الأنظمة معقدة، ويتراوح حجم الخط بين 6 ومرتبة المئات القليلة، فأى خط أصغر من 6 هو صعب القراءة، والخط الأكبر من 50 هو خط أكبر من خطوط عناوين الجرائد. إذًا، لدينا الآن محرف وعددين صغيرين ونُعامل هذه البيانات معاملة كائن واحد، إليك كيف نصرّح عن ذلك في لغة سي:

```
struct wp_char{
    char wp_cval;
    short wp_font;
    short wp_psize;
};
```

يصرح ما سبق عن نوع جديد من الكائنات يمكنك استخدامه ضمن البرنامج، ويعتمد الأمر بصورة رئيسية على ذكر الكلمة المفتاحية `struct`، المتبوعة بمعرّف `identifier` اختياري هو الوسم `wp_char` في هذه الحالة، ويسمح لنا هذا الوسم بتسمية النوع للإشارة إليه فيما بعد. يمكننا أيضًا استخدام الوسم بالطريقة التالية بعد التصريح عنه:

```
struct wp_char x, y;
```

يُعرّف ما سبق متغيرين باسم `x` و `y`، بالطريقة ذاتها للتعريف التالي:

```
int x, y;
```

لكن المتغيرات في المثال الأول من نوع `struct wp_char` عوضًا عن `int` في المثال الثاني، ويمثّل الوسم اسمًا للنوع الذي صرحنا عنه سابقًا.

نذكر هنا أنه من الممكن استخدام اسم وسم الهيكل مثل أي معرّف اعتيادي بصورة آمنة، ويدل الاسم على معنى مختلف عندما يُسبق بالكلمة المفتاحية `struct` فقط، ومن الشائع أن يُعرّف كائن مُهيكل باسم مماثل لوسم الهيكل الخاص به.

```
struct wp_char wp_char;
```

يُعرّف السطر السابق متغيرًا باسم `wp_char` من النوع `struct wp_char`، ويمكننا فعل ذلك لأن لوسوم الهياكل "فضاء أسماء `name space`" خاصة بها ولا تتعارض مع الأسماء الأخرى، وسنتكلم أكثر عن الوسوم عندما نناقش "الأنواع غير المكتملة `incomplete types`".

يمكن التعريف عن المتغيرات على الفور عقب التصريح عن هيكل ما:

```
struct wp_char{
    char wp_cval;
    short wp_font;
    short wp_psize;
}v1;

struct wp_char v2;
```

لدينا في هذه الحالة متغيرين، أحدهما باسم `v1` والآخر باسم `v2`، وإذا استخدمنا الطريقة السابقة في التعريف عن `v1`، يصبح الوسم غير ضروري ويُتخلّى عنه غالبًا إلا في حال احتجنا إلى الوسم لاستخدامه مع عامل `sizeof` والتحويل بين الأنواع `casts`.

يُعد المتغيران السابقان كائنات مُهيكلّة، ويحتوي كل منهما على ثلاثة أعضاء **members** مستقلين عن بعضهم باسم `wp_cval` و `wp_font` و `wp_psize`، ونستخدم عامل النقطة . للوصول إلى كلٍّ من الأعضاء السابقة على النحو التالي:

```
v1.wp_cval = 'x';
v1.wp_font = 1;
v1.wp_psize = 10;

v2 = v1;
```

تُضبط أعضاء المتغير `v1` في المثال السابق إلى قيمها المناسبة، ومن ثم تُنسخ قيم `v1` إلى `v2` باستخدام عملية الإسناد.

في الحقيقة، العملية الوحيدة المسموحة في الهياكل بكاملها هي الإسناد؛ إذ يمكن إسناد الهياكل إلى بعضها بعضًا أو تمريرها مثل وسطاء للدوال أو قيمة تُعيدها دالة ما، إلا أن نسخ الهياكل عملية غير فعالة وتتفادها معظم البرامج عن طريق التلاعب بالمؤشرات التي تشير إلى الهياكل عوضًا عن ذلك، إذ من الأسرع عمومًا نسخ المؤشرات عوضًا عن الهياكل. تسمح اللغة بمقارنة الهياكل بحثًا عن المساواة فيما بينها، وهي سهوة مفاجئة ولا يوجد سبب مقنع لسماحها بذلك كما سنذكر قريبًا.

إليك مثالًا يستخدم مصفوفةً من الهياكل، وهو الهيكل ذاته الذي تكلمنا عنه سابقًا، إذ تُستخدم دالة لقراءة المحارف من دخل البرنامج القياسي وتُعيد هيكلًا مهنيًا بقيم مناسبة مقابله، ومن ثم تُرتب الهياكل بحسب قيمة كل محرف وتُطبع وذلك عندما يُقرأ محرف سطر جديد، أو عندما تمتلئ المصفوفة.

```
#include <stdio.h>
#include <stdlib.h>

#define ARSIZE 10

struct wp_char{
    char wp_cval;
    short wp_font;
    short wp_psize;
}ar[ARSIZE];

/* نوع دخل الدالة الذي كان من الممكن التصريح عنه سابقًا، وتعيد الدالة هيكلًا ولا تأخذ أي وسطاء */
struct wp_char infun(void);
```

```

main(){
    int icount, lo_indx, hi_indx;

    for(icount = 0; icount < ARSIZE; icount++){
        ar[icount] = infun();
        if(ar[icount].wp_cval == '\n'){
            /*
             * غادر الحلقة التكرارية
             * دون زيادة قيمة icount
             * مع تجاهل \n
             */
            break;
        }
    }

    /* نجري الآن عملية الترتيب */

    for(lo_indx = 0; lo_indx <= icount-2; lo_indx++){
        for(hi_indx = lo_indx+1; hi_indx <= icount-1; hi_indx++){
            if(ar[lo_indx].wp_cval > ar[hi_indx].wp_cval){
                /*
                 * التبدل بين الهيكلين
                 */
                struct wp_char wp_tmp = ar[lo_indx];
                ar[lo_indx] = ar[hi_indx];
                ar[hi_indx] = wp_tmp;
            }
        }
    }

    /* طباعة القيم */
    for(lo_indx = 0; lo_indx < icount; lo_indx++){
        printf("%c %d %d\n", ar[lo_indx].wp_cval,
            ar[lo_indx].wp_font,
            ar[lo_indx].wp_psize);
    }
    exit(EXIT_SUCCESS);
}

```

```

}

struct wp_char
infun(void){
    struct wp_char wp_char;

    wp_char.wp_cval = getchar();
    wp_char.wp_font = 2;
    wp_char.wp_psize = 10;

    return(wp_char);
}

```

[مثال 1]

من الطبيعي أن نلجأ إلى التصريح عن مصفوفات من الهياكل حالما نستطيع ونتعلم كيفية التصريح عنها، وأن نستخدم هذه الهياكل عناصر لهياكل أخرى وما إلى ذلك، والقيود الوحيدة هنا هو أنه لا يمكن للهيكل أن يحتوي مثلاً لنفسه على أنه عضو داخله (يصبح حينها حجمها موضع جدل مثير للفلاسفة، ولكنه غير مفيد لأي مبرمج سي C).

6.2.1 المؤشرات والهياكل

ذكرنا سابقاً أنه من الشائع استخدام المؤشرات في الهياكل بدلاً من استخدام الهياكل مباشرةً، لتتبع كيفية تحقيق ذلك إذاً. يُعد التصريح عن المؤشرات سهلاً، ونعتقد أنك أتقنته:

```
struct wp_char *wp_p;
```

يمكننا التصريح السابق مؤشراً مباشرةً، ولكن كيف يمكننا الوصول إلى أعضاء الهيكل؟ تتمثل إحدى الطرق باستخدام المؤشر الذي يشير إلى الهيكل، ثم اختيار العضو على النحو التالي:

```
/* نحصل على الهيكل ومن ثم نحدد العضو */
(*wp_p).wp_cval
```

نستخدم الأقواس لأن أسبقية عامل النقطة . أعلى من *، إلا أن الطريقة السابقة غير سهلة التعامل وقدمت لغة سي نتيجة لذلك عاملاً جديداً لجعل التعليمة أنيقة ويُعرف باسم العامل "المُشير إلى pointing-to"، إليك مثلاً عن استخدامه:

```
// wp_cval في الهيكل wp_p الذي يشير إليه المؤشر
wp_p->wp_cval = 'x';
```

ومع أن مظهره غير مثالي، إلا أنه مفيد جدًا في حال احتواء هيكل ما على المؤشرات، مثل القوائم المترابطة Linked list، إذ أن استخدام الطريقة السابقة أسهل بكثير إن أردت تتبع مرحلة أو مرحلتين من الروابط ضمن قائمة مترابطة. إذا لم تصادف القوائم المترابطة بعد، فلا تقلق، إذ سنتطرق إليها لاحقًا.

إذا كان الكائن على يسار العامل . أو -> نوعًا مؤهلاً qualified type (باستخدام الكلمة المفتاحية const أو volatile)، فستكون النتيجة أيضًا معرفةً حسب هذه المؤهلات qualifiers. إليك مثالًا عن ذلك باستخدام المؤشرات؛ فعندما يشير المؤشر إلى نوع مؤهل، تكون النتيجة من نوع مؤهل أيضًا.

```
#include <stdio.h>
#include <stdlib.h>

struct somestruct{
    int i;
};

main(){
    struct somestruct *ssp, s_item;
    const struct somestruct *cssp;

    s_item.i = 1;    /* مسموح */
    ssp = &s_item;
    ssp->i += 2;    /* مسموح */
    cssp = &s_item;
    cssp->i = 0;    /* غير مسموح لأن المؤشر cssp يشير إلى كائن ثابت */

    exit(EXIT_SUCCESS);
}
```

يبدو أن بعض مبرمجي المصرفات نسوا هذا المتطلب، إذ استخدمنا مصرّفًا لتجربة المثال السابق ولم يحذروا بخصوص الإسناد الأخير الذي خرق القيد.

إليك المثال 1 مكتوبًا باستخدام المؤشرات، وبتغيير دالة الدخل in fun بحيث تقبل مؤشرًا يشير إلى هيكل بدلًا من إعادة مؤشر، وهذا ما ستراه على الأغلب عندما تنظر إلى بعض البرامج العملية.

نتخلى عن نسخ الهياكل في البرامج إن أردنا زيادة فاعلية تنفيذها ونستخدم بدلاً من ذلك مصفوفات تحتوي على مؤشرات؛ إذ تُستخدم هذه المؤشرات للحصول على البيانات المخزنة، إلا أن هذه الطريقة ستزيد من تعقيد الأمور، ولا تستحق العناء في التطبيقات البسيطة.

```
#include <stdio.h>
#include <stdlib.h>

#define ARSIZE 10

struct wp_char{
    char wp_cval;
    short wp_font;
    short wp_psize;
}ar[ARSIZE];

void infun(struct wp_char *);

main(){
    struct wp_char wp_tmp, *lo_idx, *hi_idx, *in_p;

    for(in_p = ar; in_p < &ar[ARSIZE]; in_p++){
        infun(in_p);
        if(in_p->wp_cval == '\n'){
            /*
             * غادر الحلقة التكرارية
             * in_p زيادة قيمة
             * \n مع تجاهل
             */
            break;
        }
    }

    /*
     * نبدأ بترتيب القيم
     * علينا الحرص هنا وتجنب حالة طفحان
     * لذا نتفقد دائماً وجود قيمتين لترتيبهما
    */
}
```

```

    */

    if(in_p-ar > 1) for(lo_indx = ar; lo_indx <= in_p-2; lo_indx++){
        for(hi_indx = lo_indx+1; hi_indx <= in_p-1; hi_indx++){
            if(lo_indx->wp_cval > hi_indx->wp_cval){
                /*
                 * التبدل بين الهيكلين
                 */
                struct wp_char wp_tmp = *lo_indx;
                *lo_indx = *hi_indx;
                *hi_indx = wp_tmp;
            }
        }
    }

    /* طباعة القيم */
    for(lo_indx = ar; lo_indx < in_p; lo_indx++){
        printf("%c %d %d\n", lo_indx->wp_cval,
                lo_indx->wp_font,
                lo_indx->wp_psize);
    }
    exit(EXIT_SUCCESS);
}

void
infun( struct wp_char *inp){

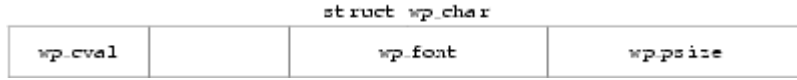
    inp->wp_cval = getchar();
    inp->wp_font = 2;
    inp->wp_psize = 10;

    return;
}

```

[مثال 2]

هناك مشكلة أخرى يجب النظر إليها، ألا وهي كيف سيبدو الهيكل عند تخزينه في الذاكرة؟ إلا أننا لن نقلق بهذا الخصوص كثيرًا في الوقت الحالي، ولكن من المفيد أن نستخدم في بعض الأحيان هياكل لغة سي C مكتوبة بواسطة برامج أخرى. تُحجز المساحة للهيكل `wp_char` كما هو موضح على النحو التالي:



الشكل 9: مخطط تخزين الهيكل

يفترض الشكل بعض الأشياء مسبقًا: يأخذ المتغير من نوع `char` بايتًا واحدًا من الذاكرة، بينما يأخذ `short` 2 بايت من الذاكرة، وأن لجميع المتغيرات من نوع `short` عنوانًا زوجيًا على هذه المعمارية، ونتيجةً لما سبق يبقى عضو واحد بحجم 1 بايت ضمن الهيكل دون تسمية مُدخل من المصرف وذلك لأغراض تتعلق بمعمارية الذاكرة. القيود السابقة شائعة الوجود وتتسبب غالبًا بما يسمى هياكل ذات "ثقوب holes".

تضمن لغة سي المعيارية بعض الأمور بخصوص تنسيق الهياكل والاتحادات:

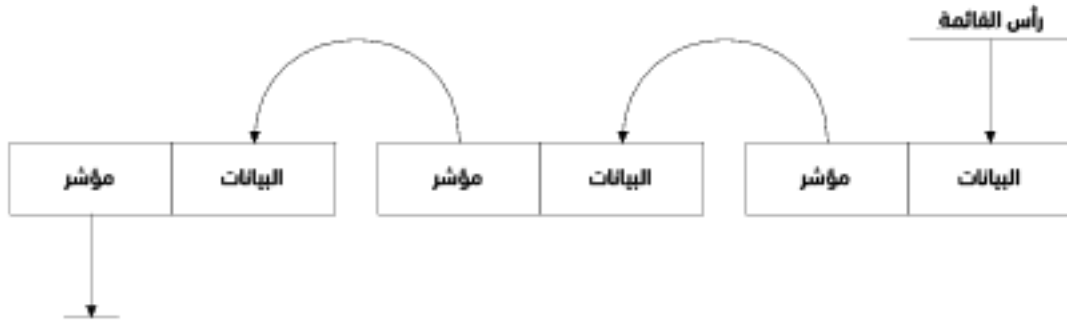
- تُحجز الذاكرة لكلٍ من أعضاء الهياكل بحسب الترتيب التي ظهرت بها هذه الأعضاء ضمن التصريح عن الهيكل وبترتيبٍ تصاعدي للعناوين.
- لا يجب أن يكون هناك أي حشو `padding` في الذاكرة أمام العضو الأول.
- يماثل عنوان الهيكل عنوان العضو الأول له، وذلك بفرض استخدام تحويل الأنواع `casting` المناسب، وبالنظر إلى التصريح السابق للهيكل `wp_char` فإن التالي محقق:

```
(char *)item == &item.wp_cval
```

- ليس لحقول البتات `bit fields` (سنذكرها لاحقًا) أي عناوين، فهي محزّمة تقنيًا إلى وحدات `units` وتطبق عليها القوانين السابقة.

6.2.2 القوائم المترابطة وهياكل أخرى

يفتح استخدام الهياكل مع المؤشرات الباب لكثيرٍ من الإمكانيات. لسنا بصدد تقديم شرح مفصل ومعقد عن هياكل البيانات المترابطة هنا، ولكننا سنشرح مثالين شائعين جدًا من هذه الطبيعة، ألا وهما القوائم المترابطة `Linked lists` والأشجار `Trees`، ويجمع بين الهيكلين السابقين استخدام المؤشرات بداخلهما تشير إلى هياكل أخرى، وتكون الهياكل الأخرى عادةً من النوع ذاته. يوضح الشكل 2 طبيعة القائمة المترابطة.



الشكل 10: قائمة مترابطة باستخدام المؤشرات

نحتاج للحصول على ما سبق إلى لتصريح عنه بما يوافق التالي:

```
struct list_ele{
    int data;        /*تستطيع تسمية العضو بأي اسم */
    struct list_ele *ele_p;
};
```

يبدو للوهلة الأولى أن الهيكل يحتوي نفسه (وهو ممنوع) ولكن يحتوي الهيكل في حقيقة الأمر مؤشرًا يشير إلى نفسه فقط، لكن لم يُعد التصريح عن المؤشر بالشكل السابق مسموحًا؟ يعلم المصنف بحلول وصوله إلى تلك النقطة بوجود `struct list_ele`، ولهذا السبب يكون التصريح مسموح، ومن الممكن أيضًا كتابة تصريح غير مكتمل للهيكل على النحو التالي قبل التصريح الكامل:

```
struct list_ele;
```

يصرح التصريح السابق عن نوع غير مكتمل **incomplete type**، سيسمح بالتصريح عن المؤشرات قبل التصريح الكامل، يفيد ذلك أيضًا في حال وجود حالة للإشارة إلى هياكل فيما بينها التي يجب أن تحتوي مؤشرًا لكل منها كما هو موضح في المثال.

```
struct s_1;        /* نوع غير مكتمل */
struct s_2{
    int something;
    struct s_1 *sp;
};
struct s_1{        /* التصريح الكامل */
    float something;
    struct s_2 *sp;
};
```

[مثال 3]

يوضح المثال السابق حاجتنا للأنواع غير المكتملة، كما يوضح خاصيةً مهمةً لأسماء أعضاء الهيكل إذ يشكّل كل هيكل فضاء أسماء name space خاص به، ويمكن بذلك أن تتماثل أسماء عناصر من هياكل مختلفة دون أي مشاكل.

تُستخدم الأنواع غير المكتملة فقط في حال لم نكن بحاجة استخدام حجم الهيكل، وإلا فيجب التصريح كاملاً عن الهيكل قبل استخدام حجمه، ولا يجب أن يكون هذا التصريح بداخل كتلة برمجية داخلية وإلا سيصبح تصريحًا جديدًا لهيكل مختلف.

```

struct x;          /* نوع غير مكتمل */

/* استخدام مسموح للوسوم */
struct x *p, func(void);

void f1(void){
    struct x{int i;};          /* إعادة تصريح */
}

/* التصريح الكامل */
struct x{
    float f;
}s_x;

void f2(void){
    /* تعليمات صالحة */
    p = &s_x;
    *p = func();
    s_x = func();
}

struct x
func(void){
    struct x tmp;
    tmp.f = 0;
    return (tmp);
}

```

[مثال 4]

يجدر الانتباه إلى أنك تحصل على هيكل من نوع غير مكتمل فقط عن طريق ذكر اسمه، وبناءً على ما سبق، تعمل الشيفرة التالية دون مشاكل:

```
struct abc{ struct xyz *p;};
/* struct xyz غير المكتمل */
struct xyz{ struct abc *p;};
/* أصبح النوع غير المكتمل مكتملاً */
```

هناك خطرٌ كبير في المثال السابق، كما هو موضح هنا:

```
struct xyz{float x;} var1;

main(){
    struct abc{ struct xyz *p;} var2;

    /* إعادة تصريح للهيكـل xyz */
    struct xyz{ struct abc *p;} var3;
}
```

نتيجةً لما سبق، يمكن للمتغير `var2.p` أن يخزن عنوان `var1`، وليس عنوان `var3` قطعاً الذي هو من نوع مختلف. يمكن تصحيح ما سبق (بفرض أنك لم تتعمد فعله) على النحو التالي:

```
struct xyz{float x;} var1;

main(){
    struct xyz; /* نوع جديد غير مكتمل */
    struct abc{ struct xyz *p;} var2;
    struct xyz{ struct abc *p;} var3;
}
```

يُستكمل نوع الهيكل أو الاتحاد عند الوصول إلى قوس الإغلاق `}`، ويجب أن يحتوي عضوًا واحدًا على الأقل أو نحصل على سلوك غير محدد.

نستطيع الحصول على أنواع غير مكتملة عن طريق تصريح مصفوفة دون تحديد حجمها، ويصنف النوع على أنه غير مكتمل حتى يقدم تصريحًا آخرًا حجمها:

```
int ar[]; /* نوع غير مكتمل */
int ar[5]; /* نكمل النوع هنا */
```

سيعمل المثال السابق إن جربته فقط في حال كانت التصريحات خارج أي كتلة برمجية (تصريحات خارجية)، إلا أن السبب في ذلك ليس متعلقًا بموضوعنا.

بالعودة إلى مثال القوائم المترابطة، كان لدينا ثلاث عناصر مترابطة في القائمة، التي يمكن بناؤها على

النحو التالي:

```

struct list_ele{
    int data;
    struct list_ele *pointer;
}ar[3];

main(){

    ar[0].data = 5;
    ar[0].pointer = &ar[1];
    ar[1].data = 99;
    ar[1].pointer = &ar[2];
    ar[2].data = -7;
    ar[2].pointer = 0;        /* ترميز نهاية المصفوفة */
    return(0);
}

```

[مثال 5]

يمكننا طباعة محتويات القائمة بطريقتين، إما بالمرور على المصفوفة بحسب دليلها index، أو باستخدام

المؤشرات كما سنوضح في المثال التالي:

```

#include <stdio.h>
#include <stdlib.h>

struct list_ele{
    int data;
    struct list_ele *pointer;
}ar[3];

main(){

    struct list_ele *lp;

```

```

ar[0].data = 5;
ar[0].pointer = &ar[1];
ar[1].data = 99;
ar[1].pointer = &ar[2];
ar[2].data = -7;
ar[2].pointer = 0;          /* ترميز نهاية المصفوفة */

/* اتباع المؤشرات */
lp = ar;
while(lp){
    printf("contents %d\n", lp->data);
    lp = lp->pointer;
}
exit(EXIT_SUCCESS);
}

```

[مثال 6]

الطريقة التي تُستخدم فيها المؤشرات في المثال السابق مثيرة للاهتمام، لاحظ كيف أن المؤشر الذي يشير إلى عنصر ما يُستخدم للإشارة إلى العنصر الذي يليه حتى إيجاد المؤشر ذو القيمة 0، مما يتسبب بتوقف حلقة while التكرارية. يمكن ترتيب المؤشرات بأي طريقة وهذا ما يجعل القائمة هيكلًا من التعامل. إليك دالة يمكن تضمينها مثل جزء من برنامجنا السابق بهدف ترتيب القائمة المترابطة بحسب قيمة بياناتها العددية، وذلك عن طريق إعادة ترتيب المؤشرات حتى الوصول إلى عناصر القائمة عند المرور عليها بالترتيب. من المهم أن نشير هنا إلى أن البيانات لا تُنسخ، إذ تعيد الدالة مؤشرًا إلى بداية القائمة لأن بدايتها لا تساوي إلى التعبير ar[0] بالضرورة.

```

struct list_ele *
sortfun( struct list_ele *list )
{

    int exchange;
    struct list_ele *nextp, *thisp, dummy;

    /*
    الخوارزمية على النحو التالي: *

```



```

* البحث عبر القائمة بصورة متكررة *
* إذا وجد عنصرين خارج الترتيب *
* إربطهما بصورة معاكسة *
* توقف عند المرور بجميع عناصر القائمة *
* دون أي تبديل مطلوب *
* يحدث الخلط عند العمل على العنصر خلف العنصر الأول المثير للاهتمام *
* وهذا بسبب الآليات البسيطة المتعلقة بربط العناصر وإلغاء ربطها *
*/

dummy.pointer = list;
do{
    exchange = 0;
    thisp = &dummy;
    while( (nextp = thisp->pointer)
        && nextp->pointer){
        if(nextp->data < nextp->pointer->data){
            /* exchange */
            exchange = 1;
            thisp->pointer = nextp->pointer;
            nextp->pointer =
                thisp->pointer->pointer;
            thisp->pointer->pointer = nextp;
        }
        thisp = thisp->pointer;
    }
}while(exchange);

return(dummy.pointer);
}

```

[مثال 7]

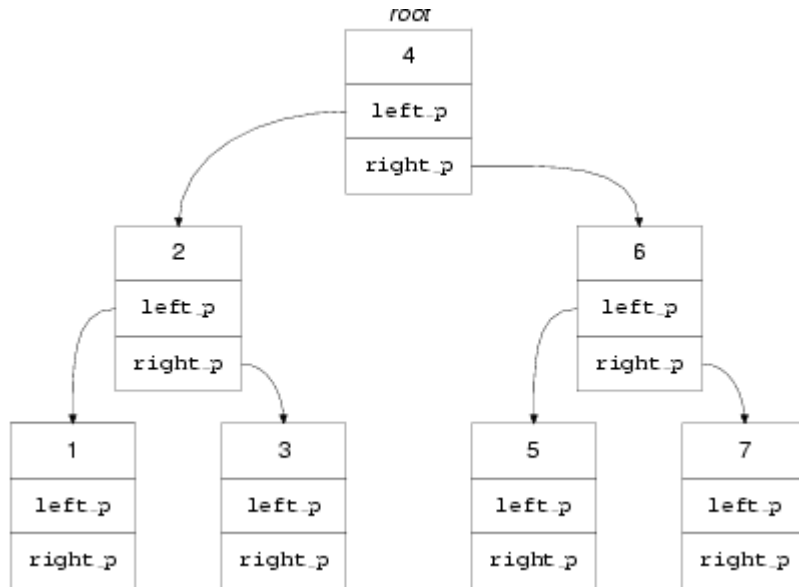
ستلاحظ استخدام تعابير مشابهة للتعبير `thisp->pointer->pointer` عند التعامل مع القوائم، وبالتالي يجب أن تفهم هذه التعابير، وهي بسيطة إذ يدل شكلها على الروابط المتبعة.

6.2.3 الأشجار

تُعد الأشجار هيكل بيانات شائع أيضًا، وهي في حقيقة الأمر قائمةً مترابطةً ذات فروع، والنوع الأكثر شيوعًا هو **الشجرة الثنائية binary tree**، التي تحتوي على عناصر تُدعى العقد "nodes" كما يلي:

```
struct tree_node{
    int data;
    struct tree_node *left_p, *right_p;
};
```

تعمل الأشجار في علوم الحاسوب من الأعلى إلى الأسفل (لأسباب تاريخية لن نناقشها)، إذ توجد عقدة **الجذر root** أعلى الشجرة وتتفرع فروع هذه الشجرة في الأسفل. تُستبدل بيانات أعضاء الهيكل الخاصة بالعقد بقيمتها في الشكل التالي والتي سنستخدمها لاحقًا.



الشكل 11: شجرة

لن تجذب الأشجار انتباهك إذا كان اهتمامك الرئيس هو التعامل مع المحارف والتلاعب بها، ولكنها مهمة جدًا بالنسبة لمصممي كل من قواعد البيانات والمصنّفات والأدوات المعقدة الأخرى.

تتميز الأشجار بميزة خاصة جدًا ألا وهي أنها مرتبة، فيمكن أن تدعم بكل سهولة خوارزميات البحث الثنائي، ومن الممكن دائمًا إضافة مزيدٍ من العقد الجديدة إلى الشجرة في الأماكن المناسبة، فالشجرة هيكل بيانات مفيدي ومرن.

بالنظر إلى الشكل السابق، نلاحظ أن الشجرة مبنيةٌ بحرص حتى تكون مهمة البحث عن قيمة ما في حقول البيانات من العقد مهمةً سهلةً، وإن فرضنا أننا نريد أن نعرف فيما إذا كانت القيمة x موجودةً في الشجرة عبر البحث عنها، تتبع الخوارزمية التالية:

- نبدأ بعقدة جذر الشجرة:
 - إذا كانت الشجرة فارغة (لا تحتوي على عقد)
 - إعادة القيمة "فشل البحث"
 - إذا كانت القيمة التي نبحث عنها مساويةً إلى قيمة العقدة الحالية
 - إعادة القيمة "نجاح البحث"
 - إذا كانت القيمة في العقدة الحالية أكبر من القيمة التي نبحث عنها
 - ابحث عن القيمة في الشجرة المُشار إليها بواسطة المؤشر الأيسر
 - فيما عدا ذلك، ابحث عن القيمة في الشجرة المُشار إليها بواسطة المؤشر الأيمن
- إليك الخوارزمية بلغة سي:

```
#include <stdio.h>
#include <stdlib.h>
struct tree_node{
    int data;
    struct tree_node *left_p, *right_p;
}tree[7];
/*
خوارزمية البحث ضمن الشجرة
* تبحث عن القيمة v في الشجرة
* تُعيد مؤشر يشير إلى أول عقدة تحوي النتيجة
* أو تُعيد القيمة 0
*/
struct tree_node *
t_search(struct tree_node *root, int v){

    while(root){
        if(root->data == v)
            return(root);
        if(root->data > v)
            root = root->left_p;
        else
            root = root->right_p;
```

```

    }
    /* لا يوجد أي شجرة متبقية ولم يُعثر على القيمة */
    return(0);
}

main(){
    /* بناء الشجرة يدويًا */
    struct tree_node *tp, *root_p;
    int i;
    for(i = 0; i < 7; i++){
        int j;
        j = i+1;

        tree[i].data = j;
        if(j == 2 || j == 6){
            tree[i].left_p = &tree[i-1];
            tree[i].right_p = &tree[i+1];
        }
    }
    /* الجذر */
    root_p = &tree[3];
    root_p->left_p = &tree[1];
    root_p->right_p = &tree[5];

    /* حاول أن تبحث */
    tp = t_search(root_p, 9);
    if(tp)
        printf("found at position %d\n", tp-tree);
    else
        printf("value not found\n");
    exit(EXIT_SUCCESS);
}

```

[مثال 8]

يعمل المثال السابق بنجاح دون أخطاء، ومن الجدير بالذكر أنه يمكننا استخدام ذلك في جعل أي قيمة مدخلة إلى الشجرة تُخزن في مكانها الصحيح باستخدام خوارزمية البحث ذاتها، أي بإضافة شيفرة برمجية إضافية

تحتج مساحة لقيمة جديدة باستخدام دالة `malloc` عندما لا تجد الخوارزمية القيمة، وتُضاف العقدة الجديدة في مكان مؤشر الفراغ `null pointer` الأول. من المعقد تحقيق ما سبق، وذلك بسبب مشكلة التعامل مع مؤشر العقدة الجذر، ونلجأ في هذه الحالة إلى مؤشر يشير إلى مؤشر آخر. اقرأ المثال التالي بانتباه، إذ أنه أحد أكثر الأمثلة تعقيدًا حتى اللحظة، وإذا استطعت فهمه فهذا يعني أنك تستطيع فهم الأغلبية الساحقة من برامج سي.

```
#include <stdio.h>
#include <stdlib.h>

struct tree_node{
    int data;
    struct tree_node *left_p, *right_p;
};

/*
خوارزمية البحث ضمن شجرة
ابحث عن القيمة v ضمن الشجرة
أعد مؤشرًا إلى أول عقدة تحتوي على القيمة هذه
أعد القيمة 0 إن لم تجد نتيجة
*/
struct tree_node *
t_search(struct tree_node *root, int v){

    while(root){
        printf("looking for %d, looking at %d\n",
            v, root->data);
        if(root->data == v)
            return(root);
        if(root->data > v)
            root = root->left_p;
        else
            root = root->right_p;
    }
    /* value not found, no tree left */
    return(0);
}
/*
```

```

* أدخل عقدة ضمن شجرة *
* أعد 0 عند نجاح العملية أو *
* أعد 1 إن كانت القيمة موجودة في الشجرة *
* أعد 2 إن حصل خطأ في عملية حجز الذاكرة malloc error *
*/
int
t_insert(struct tree_node **root, int v){

    while(*root){
        if((*root)->data == v)
            return(1);
        if((*root)->data > v)
            root = &((*root)->left_p);
        else
            root = &((*root)->right_p);
    }
    /* value not found, no tree left */
    if((*root = (struct tree_node *)
        malloc(sizeof (struct tree_node)))
        == 0)
        return(2);
    (*root)->data = v;
    (*root)->left_p = 0;
    (*root)->right_p = 0;
    return(0);
}

main(){
    /* construct tree by hand */
    struct tree_node *tp, *root_p = 0;
    int i;

    /* we ignore the return value of t_insert */
    t_insert(&root_p, 4);
    t_insert(&root_p, 2);
    t_insert(&root_p, 6);
}

```

```

t_insert(&root_p, 1);
t_insert(&root_p, 3);
t_insert(&root_p, 5);
t_insert(&root_p, 7);

/* try the search */
for(i = 1; i < 9; i++){
    tp = t_search(root_p, i);
    if(tp)
        printf("%d found\n", i);
    else
        printf("%d not found\n", i);
}
exit(EXIT_SUCCESS);
}

```

[مثال 9]

تسمح لك الخوارزمية التالية بالمرور على الشجرة وزيارة جميع العُقد بالترتيب باستخدام التعاودية recursion، وهي أحد أكثر الأمثلة أناقةً، انظر إليها وحاول فهمها.

```

void
t_walk(struct tree_node *root_p){

    if(root_p == 0)
        return;
    t_walk(root_p->left_p);
    printf("%d\n", root_p->data);
    t_walk(root_p->right_p);
}

```

[مثال 10]

6.3 الاتحادات Unions

لن تستغرق الاتحادات Unions وقتًا طويلاً لشرحها، فهي تشابه الهياكل بفرق أنك لا تستخدم الكلمة المفتاحية struct بل تستخدم union، وتعمل الاتحادات بالطريقة ذاتها التي تعمل بها الهياكل structures بفرق أن أعضائها مُخزنون على كتلة تخزينية واحدة بعكس أعضاء الهياكل التي تُخزن على كتل تخزينية متفرقة

متعاقبة، ولكن ما الذي يفيدنا هذا الأمر؟ تدفعنا الحاجة في بعض الأحيان إلى استخدام الهياكل بهدف تخزين قيم مختلفة بأنواع مختلفة وبأوقات مختلفة مع المحافظة قدر الإمكان على مساحة التخزين وعدم هدر الموارد؛ في حين يمكننا باستخدام الاتحادات تحديد النوع الذي ندخله إليها والتأكد من استرجاع القيمة بنوعها المناسب فيما بعد. إليك مثالاً عن ذلك:

```
#include <stdio.h>
#include <stdlib.h>

main(){
    union {
        float u_f;
        int u_i;
    }var;

    var.u_f = 23.5;
    printf("value is %f\n", var.u_f);
    var.u_i = 5;
    printf("value is %d\n", var.u_i);
    exit(EXIT_SUCCESS);
}
```

[مثال 11]

إذا أضفنا قيمةً من نوع `float` إلى الاتحاد في مثالنا السابق، ثم استعدناه على أنه قيمةً من نوع `int`، فسنحصل على قيمة غير معروفة، لأن النوعان يُخزنان على نحوٍ مختلف وأضف على ذلك أنهما من أطوالٍ مختلفة؛ فالقيمة من نوع `int` ستكون غالبًا تمثيل الآلة (الحاسوب) لبتات `float` منخفضة الترتيب، ولربما ستشكل جزءًا من قيمة `float` العشرية (ما بعد الفاصلة). ينص المعيار على اعتماد النتيجة في هذه الحالة على تعريف التطبيق (وليست سلوكًا غير معرفًا)، والنتيجة معرفةً من المعيار في حالة واحدة، ألا وهي أن يكون لبعض أعضاء الاتحاد هياكل ذات "سلسلة مبدئية مشتركة `common initial sequence`"، أي أن لأول عضو من كل هيكل نوع متوافق `compatible type`، أو من الطول ذاته في حالة حقول البتات `bitfields`، ويوافق اتحادنا الشروط التي ذكرناها، وبالتالي يمكننا استخدام السلسلة المبدئية المشتركة على نحوٍ تبادلي، يا لحظنا الرائع.

يعمل مصرّف لغة سي على حجز المساحة اللازمة لأكبر عضو ضمن الاتحاد لا أكثر (بعنوان مناسب إن أمكن)، أي لا يوجد هناك أي تفقد للتأكد من أن استخدام الأعضاء صائب فهذه مهمتك، وستكتشف عاجلاً أم

أجلًا إذا فشلت في تحقيق هذه المهمة. تبدأ أعضاء الاتحاد من عنوان التخزين ذاته (من المضمون أنه لا يوجد هناك أي فراغات بين أي من الأعضاء).

يُعد تضمين الاتحاد في هيكل من أكثر الطرق شيوعًا لتذكر طريقة عمل الاتحاد، وذلك باستخدام عضو آخر من الهيكل ذاته ليذل على نوع الشيء الموجود في الاتحاد. إليك مثالًا عمّا سيبدو ذلك:

```
#include <stdio.h>
#include <stdlib.h>

/* شيفرة للأنواع في الاتحاد */
#define FLOAT_TYPE      1
#define CHAR_TYPE      2
#define INT_TYPE       3

struct var_type{
    int type_in_union;
    union{
        float    un_float;
        char     un_char;
        int      un_int;
    }vt_un;
}var_type;

void
print_vt(void){

    switch(var_type.type_in_union){
        default:
            printf("Unknown type in union\n");
            break;
        case FLOAT_TYPE:
            printf("%f\n", var_type.vt_un.un_float);
            break;
        case CHAR_TYPE:
            printf("%c\n", var_type.vt_un.un_char);
            break;
        case INT_TYPE:
```

```

        printf("%d\n", var_type.vt_un.un_int);
        break;
    }
}

main(){

    var_type.type_in_union = FLOAT_TYPE;
    var_type.vt_un.un_float = 3.5;

    print_vt();

    var_type.type_in_union = CHAR_TYPE;
    var_type.vt_un.un_char = 'a';

    print_vt();
    exit(EXIT_SUCCESS);
}

```

[مثال 12]

يوضح المثال السابق أيضًا استخدام عامل النقطة للوصول إلى ما داخل الهياكل أو الاتحادات التي تحتوي على هياكل أو اتحادات أخرى بدورها، تسمح لك بعض مصرفات لغة سي الحالية بإهمال بعض الأجزاء من أسماء الكائنات المُدمجة شرط ألا يتسبب ذلك بجعل الاسم غامض، فعلى سبيل المثال يسمح استخدام الاسم الواضح `var_type.un_int` للمصرف بمعرفة ما تقصده، إلا أن هذا غير مسموح في المعيار.

لا يمكن مقارنة الهياكل بحثًا عن المساواة فيما بينها ويقع اللوم على الاتحادات، إذ أن احتمالية احتواء هيكل ما على اتحاد يجعل من مهمة المقارنة مهمةً صعبة، إذ لا يمكن للمصرف أن يعرف ما الذي يحويه الاتحاد في الوقت الحالي مما لا يسمح له بإجراء عملية المقارنة. قد يبدو الكلام السابق صعب الفهم وغير دقيق بنسبة 100%، إذ أن معظم الهياكل لا تحتوي على اتحادات، ولكن هناك مشكلة فلسفية بخصوص القصد من كلمة "مساواة" عندما تُسقطها على الهياكل. بغض النظر، تمنح الاتحادات عذرًا مناسبًا للمعيار بتجنبه لأي مشاكل بواسطة عدم دعمه لمقارنة الهياكل.

6.4 حقول البتات Bitfields

دعنا نلقي نظرةً على حقول البتات بما أننا نتكلم عن موضوع هياكل البيانات، إذ يمكن تعريفها فقط بداخل هيكل أو اتحاد، وتسمح لك حقول البتات بتحديد بعض الكائنات الصغيرة بحسب طول بتات محدد، إلا أن

فأدائها محدودةً ولا تُستخدم إلا في حالات نادرة، ولكننا سنتطرق إلى الموضوع بغض النظر عن ذلك. يوضح لك المثال استخدام حقول البتات:

```
struct {
    /* كل حقل بسعة 4 بتات */
    unsigned field1 :4;
    /*
     * حقل بسعة 3 بتات دون اسم
     * تسمح الحقول عديمة الاسم بالفراغات بين عناوين الذاكرة
     */
    unsigned      :3;
    /*
     * حقل بسعة بت واحد
     * تكون قيمته 0 أو -1 في نظام المتمم الثنائي
     */
    signed field2  :1;
    /* محاذاة الحقل التالي مع وحدة التخزين */
    unsigned      :0;
    unsigned field3 :6;
}full_of_fields;
```

[مثال 13]

يمكن التلاعب والوصول إلى كل حقل بصورة منفردة وكأنه عضو اعتيادي من هيكل ما، وتعني الكلمتان المفتاحيتان `signed` و `unsigned` ما هو متوقع، إلا أنه يجدر بالذكر أن حقلًا بحجم 1 بت ذا إشارة سيأخذ واحدةً من القيمتين 0 أو -1 وذلك في آلة تعمل بنظام المتمم الثنائي، ويُسمح للتصريحات بأن تحتوي المؤهلين `volatile` أو `const`.

تُستخدم حقول البتات بشكل رئيس إما للسماح بتحزيم مجموعة من البيانات بأقل مساحة، أو لتحديد الحقول ضمن ملفات بيانات خارجية. لا تقدم لغة سي أي ضمانات بخصوص ترتيب الحقول بكلمات الآلة التي تعمل عليها، لذا إذا كنت تريد استخدام حقول البتات للهدف الثاني، فسيصبح برنامجك غير قابل للتنقل ومعتمدًا على المصنّف الذي يصرف البرنامج أيضًا. ينص المعيار على أن الحقول مُحزّمة بما يدعى "وحدات تخزين"، التي تكون عادةً كلمات آلة. يُحدّد ترتيب التحزيم وفيما إذا كان سيتجاوز حقل البتات حاجز التخزين أم لا بحسب تعريف التطبيق، ونستخدم حقلًا بعرض صفر قبل الحقل الذي تريد تطبيق الحد عنده لإجبار الحقل على البقاء ضمن حدود وحدة التخزين.

كن حذرًا عند استخدام حقول البتات، إذ يتطلب الأمر شيفرة وقت تشغيل run-time طويلة للتلاعب بهذه الأشياء، وقد ينتج ذلك بتوفير الكثير من المساحة (أكثر من حاجتك).

ليس لحقول البتات أي عناوين، وبالتالي لا يمكنك استخدام المؤشرات أو المصفوفات معها.

6.5 المعدّات enums

تقع المُعدّات enums تحت تصنيف "منجزة جزئيًا"، إذ ليست بأنواع مُعددة بصورٍ كاملة مثل لغة باسكال، ومهمتها الوحيدة هي مساعدتك في التخفيف من عدد تعليمات #define في برنامجك، إليك ما تبدو عليه:

```
enum e_tag{
    a, b, c, d=20, e, f, g=20, h
}var;
```

يمثل e_tag الوسم بصورةٍ مشابهة لما تكلمنا عنه في الهياكل والاتحادات، ويمثل var تعريفًا للمتغير.

الأسماء المُعلنة بداخل المُعدّد ثوابت من نوع int، إليك قيمها:

```
a == 0
b == 1
c == 2
d == 20
e == 21
f == 22
g == 20
h == 21
```

تلاحظ أنه بغياب أي قيمة مُسندة للمتغيرات، تبدأ القيم من الصفر تصاعديًا، ويمكنك إسناد قيمة مخصصة إن أردت في البداية، إلا أن القيم التي ستتزايد بعدها ستكون من نوع عدد صحيح ثابت integral constant (كما سنرى لاحقًا)، وتُمثّل هذه القيمة بنوع int ومن الممكن أن تحمل عدة أسماء القيمة ذاتها.

تُستخدم المُعدّات للحصول على إصدار ملائم للنطاق Scope بدلاً من استخدام #define على النحو التالي:

```
#define a 0
#define b 1
/* وهكذا دواليك */
```

إذ يتبع استخدام المعدّات لقوانين نطاق لغة سي C، بينما تشمل تعليمات `#define` كامل نطاق الملف. قد لا تهتمك هذه المعلومة، ولكن المعيار ينص على أن أنواع المعدّات من نوع متوافق مع أنواع الأعداد الصحيحة بحسب تعريف التطبيق، لكن ما الذي يعنيه ذلك؟ لاحظ المثال التالي:

```
enum ee{a,b,c}e_var, *ep;
```

تسلك الأسماء `a` و `b` و `c` سلوك الأعداد الصحيحة الثابتة `int` عندما تستخدمها، و `e_var` من نوع `enum ee` و `ep` مؤشر يشير إلى المعدد `ee`. تعني متطلبات التوافقية بين الأنواع (بالإضافة لمشكلات أخرى) أن هناك نوع عدد صحيح ذو عنوان يمكن إسناده إلى `ep` من غير خرق أي من متطلبات التوافقية بين الأنواع للمؤشرات.

6.6 المؤهلات والأنواع المشتقة

تعد المصفوفات والهياكل والاتحادات "مشتقة من `derived from`" (أي تحتوي) أنواع أخرى، ولا يمكن لأي مّا سبق أن تُشتق من أنواع غير مكتملة `incomplete types`. وهذا يعني أنه من غير الممكن للهيكل أو الاتحاد أن يحتوي مثلاً من نفسه، لأن نوعه غير مكتمل حتى ينتهي التصريح عنه، وبما أن المؤشر الذي يشير إلى نوع غير مكتمل ليس بنوع غير مكتمل بذات نفسه فمن **الممكن** استخدامه باشتقاق المصفوفات والهياكل والاتحادات.

لا ينتقل التأهيل إلى النوع المُشتق إن كان أي من الأنواع التي اشتق منها تحتوي على مؤهلات مثل `const` أو `volatile`، وهذا يعني أن الهيكل الذي يحتوي على كائن ذو مؤهل ثابت `const` لا يجعل من الهيكل بنفسه مؤهلاً بهذا المؤهل، ويمكن لأي عضو غير ثابت أن يُعدّل عليه بداخل الهيكل، وهذا ما هو متوقع، إلا أن المعيار ينص على أن أي نوع مشتق يحتوي على نوع مؤهل باستخدام `const` (أو أي نوع داخلي تعاودي) لا يمكن التعديل عليه، فالهيكل الذي يحتوي الثابت لا يمكن وضعه على الطرف الأيسر من عامل الإسناد.

6.7 التهيئة Initialization

حان الوقت للتكلم عن التهيئة `Initialization` في لغة سي بعد أن تكلمنا عن جميع أنواع البيانات المدعومة في اللغة، إذ تسمح لغة سي للمتغيرات الاعتيادية والهياكل والاتحادات والمصفوفات بأن تحمل قيمةً أوليةً عند التعريف عنها، وكان للغة سي القديمة بعض القوانين الغريبة بهذا الخصوص التي تعكس تقاعس مبرمجي مصرّفات سي عن إنجاز بعض العمل الإضافي، وأتت لغة C المعيارية لحل هذه المشكلات وأصبح من الممكن الآن تهيئة الأشياء عندما تريد وكيفما تريد.

6.7.1 أنواع التهيئة

هناك نوعان من التهيئة؛ تهيئة عند وقت التصريف compile time وتهيئة عند وقت التشغيل run time. ويعتمد النوع الذي ستحصل عليه على **مدة التخزين storage duration** للشيء الذي يُهيأ.

يُصرح عن الكائنات ذات **المدة الساكنة static duration** إما خارج الدوال، أو داخلها باستخدام الكلمة المفتاحية extern أو static على أنها جزء من التصريح، ويُهيأ هذا النوع عند وقت التصريف **فقط**.

لجميع الكائنات الأخرى **مدة تلقائية automatic duration**، يمكن تهيئتها **فقط** عند وقت التشغيل، إذ أن التصنيفين حصريان فيما بينهما. على الرغم من ارتباط مدة التخزين بالربط (انظر فصل الربط) إلا أنهما مختلفان ويجب عدم الخلط فيما بينهما.

يمكن استخدام التهيئة عند وقت التصريف فقط في حال استخدام التعابير الثابتة constant expressions، بينما يمكن استخدام التهيئة عند وقت التشغيل لأي تعبير كان، وقد أُلغي قيد لغة سي القديمة الذي ينص على إمكانية تهيئة المتغيرات البسيطة وحدها عند وقت التشغيل، وليس المصفوفات، أو الهياكل، أو الاتحادات.

6.7.2 التعابير الثابتة

هناك بعض الاستخدامات الضرورية للتعابير الثابتة، ويُعد تعريف التعبير الثابت بسيط الفهم، إذ يُقِيم **التعبير الثابت constant expression** عند وقت التصريف وليس عند وقت التشغيل، ويمكن استخدام هذا النوع من التعابير في أي مكان يسمح باستخدام قيمة ثابتة. يُشترط على التعبير الثابت ألا يحتوي على أي عامل إسناد أو زيادة أو نقصان أو استدعاء لدالة ما أو عامل الفاصلة، إلا إذا كان التعبير جزءاً من معامل sizeof، وقد يبدو هذا الشرط غريباً بعض الشيء، إلا أنه بسبب أن العامل sizeof يُقِيم نوع التعبير وليس قيمته.

يؤكد المعيار على وجوب تقييم الأعداد الحقيقية عند وقت التصريف بدقة ونطاق مماثلين لحالة تقييمهم في وقت التشغيل. يوجد هناك طريقةٌ محدودة أكثر تدعى **تعابير الأعداد الصحيحة الثابتة integral constant expressions**، ولهذه التعابير نوع عدد صحيح وتحتوي على معاملات operands من نوع عدد صحيح ثابت أو معدّات ثابتة enumeration constants، أو محارف ثابتة، بالإضافة إلى تعابير sizeof والأعداد الحقيقية الثابتة التي تكون معاملات لتحويل الأنواع casts، ويسمح لعوامل تحويل الأنواع بتحويل الأنواع الحسابية إلى أنواع صحيحة فقط. لا تُطبّق أي قيود على محتويات تعابير sizeof طبقاً لما سبق قوله (يُقِيم نوع التعبير وليس قيمته).

يُشابه التعبير الحسابي الثابت arithmetic constant expression التعبير الصحيح الثابت، ولكنه يسمح باستخدام الأعداد الصحيحة الثابتة، ويحدّ من استخدام تحويل الأنواع بالتحويل من نوع حسابي إلى آخر.

العنوان الثابت address constant هو مؤشر يشير إلى كائن ذي مدة تخزين ساكنة أو إلى مؤشر يشير إلى دالة ما، ويمكنك الحصول على هذه العناوين باستخدام العامل "&" أو باستخدام التحويلات الاعتيادية للمصفوفات وأسماء الدوال إلى مؤشرات عندما تُستخدم ضمن تعبير، ويمكن استخدام كلٍ من العوامل "[]" و "." و ">" و "&" و "*" ضمن التعبير طالما لا يتضمن ذلك الاستخدام محاولة للوصول لقيمة أي كائن.

6.7.3 استكمال عن التهيئة

يُسمح لأنواع الثوابت المختلفة بالتواجد في العديد من الأماكن، إلا أن تعابير الأعداد الصحيحة الثابتة شديدة الأهمية على وجه الخصوص لأنها من النوع الوحيد الذي قد يُستخدم لتحديد حجم المصفوفة وقيم ما يسبق تعليمة case. تتميز أنواع الثوابت المسموح باستخدامها مثل قيمة أولية لتهيئة التعابير بأنها أقل محدودية، إذ من المسموح لك باستخدام تعابير حسابية ثابتة، أو مؤشر فراغ، أو عنوان ثابت، أو عنوان ثابت لكائن زائد أو ناقص تعبير صحيح ثابت، ويعتمد الأمر طبعًا على نوع الشيء الذي يجري تهيئته سواءً كان نوع تعبير ثابت محدد مناسبًا أم لا.

إليك مثالًا يحتوي على تهيئة لعدة متغيرات:

```
#include <stdio.h>
#include <stdlib.h>

#define NMONTHS 12

int month = 0;

short month_days[] =
    {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

char *mnames[] = {
    "January", "February",
    "March", "April",
    "May", "June",
    "July", "August",
    "September", "October",
    "November", "December"
};

main(){
```

```

int day_count = month;
for(day_count = month; day_count < NMONTHS;
    day_count++){
    printf("%d days in %s\n",
        month_days[day_count],
        mnames[day_count]);
}
exit(EXIT_SUCCESS);
}

```

[مثال 14]

تهيئة المتغيرات الاعتيادية بسيطة، فعليك فقط إضافة `expression` = بعد اسم المتغير في التصريح، والتي تدل `expression` على التعبير الذي ستُستند قيمته إلى المتغير، وسيُهيأ المتغير بهذه القيمة، ويعتمد استخدام مجمل التعابير أو استخدام التعابير الثابتة حصراً على مدة التخزين، وهذا ينطبق على جميع الكائنات.

تهيئة المصفوفات أحادية البعد بسيطة أيضاً، إذ عليك فقط كتابة قائمة بالقيم التي تريدها وتفصل كل قيمة بفاصلة داخل أقواس معقوفة؛ ويوضح المثال التالي طريقة تهيئة المصفوفة، ويُحدّد حجم المصفوفة طبقاً للقيم الأولية الموجودة إن لم تحدد حجم المصفوفة على نحو صريح، أما إذا حددت الحجم صراحةً، فيجب أن يكون عدد القيم الأولية التي تزودها يساوي أو أصغر من الحجم، إذ يسبب إضافة قيم أكبر من حجم المصفوفة خطأً، بينما ستهيئ القيم الموجودة -وإن كانت قليلة- العناصر الأولى من المصفوفة.

يمكنك بناء سلسلة نصية يدوياً بالطريقة:

```
char str[] = {'h', 'e', 'l', 'l', 'o', 0};
```

يمكن أيضاً استخدام سلسلة نصية ضمن علامتي تنصيص لتهيئة مصفوفة من المحارف:

```
char str[] = "hello";
```

سيُضمّن المحرف الفارغ في نهاية المصفوفة في حالتنا السابقة تلقائياً إذا كانت هناك مساحة كافية، أو إذا لم يُحدد حجم المصفوفة، إليك المثال التالي:

```

/* لا يوجد مكان للمحرف الفارغ */
char str[5] = "hello";

/* يوجد مكان للمحرف الفارغ */
char str[6] = "hello";

```


استخدم البرنامج في مثال 14 السلاسل النصية لأهداف مختلفة، إذ كان استخدامهم بهدف تهيئة مصفوفة من مؤشرات تشير إلى محارف، وهذا استخدام مختلف تمامًا. يمكن استخدام تعبير من نوع مناسب لتهيئة هياكل من نوع مدة تلقائية، وإلا فيجب استخدام قائمة تحتوي على تعابير ثابتة بين قوسين على النحو التالي:

```
#include <stdio.h>
#include <stdlib.h>

struct s{
    int a;
    char b;
    char *cp;
}ex_s = {
    'a', "hello"
};

main(){
    struct s first = ex_s;
    struct s second = {
        'b', "byebye"
    };

    exit(EXIT_SUCCESS);
}
```

[مثال 15]

يمكن تهيئة العنصر الأول فقط من الاتحاد.

يحدث تجاهلًا للأعضاء عديمة الاسم ضمن الهيكل أو الاتحاد خلال عملية التهيئة، سواءً كانت حقول بتات bitfields، أو مسافات فارغة لمحاذاة عنوان التخزين، فمن غير المطلوب أخذهم بالحسبان عندما تختار القيم الأولية لأعضاء الهيكل الحقيقية (ذات الاسم).

هناك طريقتان لكتابة القيم الأولية لكائنات تحتوي على كائنات فرعية بداخلها، فيمكن كتابة قيمة أولية لكل عضو بالطريقة:

```
struct s{
```

```

int a;
struct ss{
    int c;
    char d;
}e;
}x[] = {
2, 'a',
4, 'b'
};

```

[مثال 16]

سيُسنَد ما سبق القيمة 1 إلى `x[0].a` و 2 إلى `x[0].e.c` و `x[0].e.d` إلى `a` و 3 إلى `x[1].a` وهكذا. استخدام الأقواس الداخلية **أكثر أمانًا** للتعبير عن القيم التي تقصدها، إذ تسبب قيمة خاطئة واحدة فوضى عارمة.

```

struct s{
    int a;
    struct ss{
        int c;
        char d;
    }e;
}x[] = {
    {1, {2, 'a'}},
    {3, {4, 'b'}}
};

```

[مثال 17]

استخدم الأقواس دائمًا، لأن هذه الطريقة آمنة، والأمر مماثل بالنسبة للمصفوفات بكونها هياكل:

```

float y[4][3] = {
    {1, 3, 5}, /* y[0][0], y[0][1], y[0][2] */
    {2, 4, 6}, /* y[1][0], y[1][1], y[1][2] */
    {3, 5, 7} /* y[2][0], y[2][1], y[2][2] */
};

```

[مثال 18]

تُهيأ قيم الأُسُطر الثلاثة الأولى كاملةً من y ، ويبقى السطر الرابع $y[3]$ غير مُهيأ.

تُسند لجميع الكائنات ذات المدة الساكنة قيمٌ أولية ضمنية تلقائية إذا لم يوجد أي قيمة أولية لها، ويكون أثر القيمة الأولية التلقائية الضمنية مماثلاً لأثر إسناد القيمة 0 الثابتة، وهذا الأمر شائع الاستخدام، وتفترض معظم برامج لغة سي نتيجةً لذلك أن الكائنات الخارجية والكائنات الساكنة الداخلية تبدأ بالقيمة صفر.

تُضمن عملية التهيئة للكائنات ذات المدة التلقائية فقط في حالة وجود تعليمتها المركبة "في الأعلى"، إذ قد يتسبب الانتقال إلى أحد الكائنات فوراً بعدم حصول عملية التهيئة وهذا الأمر غير مستحب ويجب تجنبه. يشير المعيار بصراحة إلى أن وجود القيم الأولية في التصريح ضمن تعليمات `switch` لن يكون مفيداً، لأن التصريح لا يُصنف بكونه تعليمة، ويمكن عنونة `label` التعليمة فقط، ونتيجةً لذلك فلا يمكن للقيم الأولية في تعليمات `switch` أن تُنفذ لأن كتلة الشيفرة البرمجية التي تحتويها يجب أن تلي ذكر التصريح.

يمكن أن يُستخدم التصريح داخل دالة ما (نطاق مرتبط بكتلة الدالة) للإشارة إلى كائن ذي ربطٍ خارجي `External linkage` أو ربط داخلي `Internal linkage` باستخدام عدّة طرق تطرقنا إليها في فصل [الربط والنطاق](#) وهناك مزيدٌ من الطرق التي سنتطرق إليها لاحقاً. إذا اتبعت الطرق السابقة (التي من المستبعد أن تتحقق من قبيل الصدفة)، فلا يمكنك تهيئة الكائن على أنه جزء من التصريح، وإليك الطريقة الوحيدة التي تستطيع تحقيق ذلك بها:

```
int x;                /* ربط خارجي */
main(){
    extern int x = 5; /* استخدام ممنوع */
}
```

لم يكشف مصرّفنا التهيئة الممنوعة في هذا المثال أيضاً.

6.8 خاتمة

تفهم الآن بالوصول لهذه النقطة ماهية الهياكل والاتحادات، إضافةً إلى أنواع حقول البتات والمعدّات إلا أنهما ليسا مهمتان بقدر كبير ويمكنك الاستغناء عنهما دون أي مشاكل.

لا يمكننا أن ننصف مقدار أهمية استخدام الهياكل والمؤشرات ودالة "malloc" في البرامج المتقدمة بالقدر الكافي، فإن لم تستوعب مفهوم الهياكل واستخدامها في القوائم والأشجار وغيرها، احصل على كتاب جيد يشرحها، بل الأفضل حاول أن تلتحق بدورة تدريبية ما. يمنحك استخدام هياكل البيانات جيداً القدرة على إنشاء برمجيات صغيرة سهلة الصيانة بسيطة وليس القدرة على كتابة الخوارزميات المعقدة -باستثناء بعض التطبيقات المتخصصة- وعادةً ما يقول مصمموا البرمجيات الخبراء أن المهمة الصعبة تنتهي باختيار هيكل البيانات المناسب للتطبيق وما تبقى بعدها بسيط.

الحرية التي تمنحها لغة سي C بخصوص تشكيل هياكل البيانات دون التضحية بسرعة التنفيذ هي من أبرز الأسباب التي ساهمت بزيادة شعبيتها بين مطوري البرمجيات الخبراء دون أي شك.

لا يجب صرف النظر عن جانب التهيئة إطلاقاً، فعلى الرغم من بساطة المفهوم إلا أن معظم اللغات الأخرى تجعل من تلك العملية غير ملائمة بصورة مفاجئة، وذلك بإجبارك على استخدام تعليمات الإسناد، بينما للغة سي طريقة عملية وملائمة. إذا كانت طريقة تضمين القيم الأولية داخل أقواس بالكامل غير سارة لك فلا تقلق، من النادر أن تستخدمها خلال ممارستك، إذ كل ما عليك معرفته هو كيفية إنجاز عمليات التهيئة البسيطة واختيار مرجع جيد يصف لك عمليات التهيئة الأكثر تعقيداً، اقرأ المعيار إذا أردت الحصول على شرح كامل لهذه النقطة إذ يقدم شرحاً واضحاً ودقيقاً حول الموضوع.

6.9 التمارين

1. كيف نصح عن هيكل عديم الوسم يحتوي قيمتين من نوع `int` باسم `a` و `b`؟
2. ما السبب في كون التصريح السابق محدود الاستخدام؟
3. كيف ستبدو عملية التعريف عن هيكل بوسم `int_struct` ومتغيرين باسم `x` و `y` داخله؟
4. كيف تستطيع التصريح عن متغير ثالث لاحقاً، من نوع مماثل للمتغيرين `x` و `y` باسم `z`؟
5. بفرض أن للمؤشر `p` النوع المناسب، كيف يمكنك جعل المؤشر يشير إلى `z` ومن ثم يسند قيمة `a.z` إلى صفر باستخدامه؟
6. ما الطريقتان في التصريح عن هيكل يحتوي على نوع غير مكتمل؟
7. ما الأمر غير الاعتيادي بخصوص استخدام السلسلة نصية `"like this"` مقل قيمة أولية لتهيئة سلسلة محارف؟
8. ماذا لو أسندنا القيمة الأولية السابقة باستخدام `*char`؟
9. ابحث عن القائمة مزدوجة الارتباط `Doubly linked list`، وأعد كتابة مثال القائمة المترابطة باستخدامها، هل عملية إدخال وحذف العناصر باستخدامها أسهل؟

مستقل
mostaql.com

ادخل سوق العمل و نفذ المشاريع باحترافية
عبر أكبر منصة عمل حر بالعالم العربي

ابدأ الآن كمستقل

7. المعالج المسبق Preprocessor

يتناول الفصل مرحلة المعالجة المسبقة للشفيرة المصدرية بما فيها مراحل استبدال الماكرو ومختلف موجهات المعالج المسبق الأخرى.

7.1 أثر المعيار

ستشعر أن المعالج المُسبق Preprocessor لا ينتمي إلى لغة سي عمومًا، إذ لا يسمح لك وجوده بالتعامل بصورة متكاملة مع اللغة كما أنك لا تستطيع الاستغناء عنه في ذات الوقت، وفي الحقيقة، كان استخدام المعالج المُسبق في أيام سي الأولى اختياريًا واعتاد الناس على كتابة برامج لغة سي C بدونه، ويمكننا أن ننظر إلى كونه جزءًا من لغة سي حاليًا صدفةً إلى حدٍ ما، إذ كان يعالج بعضًا من أوجه القصور في اللغة، مثل تعريف الثوابت وتضمين التعريفات القياسية، وأصبح نتيجةً لذلك جزءًا ضمن حزمة لغة سي ككل.

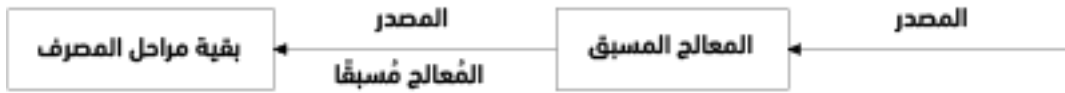
لم يكن هناك في تلك الفترة معيارٌ رسميٌ متفقٌ عليه يوحد ما يفعله المعالج المسبق، وكانت إصدارات مختلفة منه مُطبَّقة بصورةٍ مختلفٍ على عدة أنظمة، وأصبحت عملية نقل البرنامج وتصديره إلى أنظمة أخرى مشكلةً كبيرةً إذا استخدم ما يزيد عن الخصائص الأساسية للمعالج.

كانت وظيفة المعيار الأساسية هنا هي تعريف سلوك المعالج المُسبق بما يتوافق مع الممارسات الشائعة، وقد سبق حصول ذلك مع لغة سي القديمة، إلا أن المعيار اتخذ إجراءات إضافية وسط الخلاف وحدد مجموعةً من الخصائص الإضافية التي قُدمت مع إصدارات المعالج المُسبق الأكثر شعبيةً، وعلى الرغم من فائدة هذه الخصائص إلا أن الخلاف كان بخصوص الاتفاق على طريقة تطبيقها. لم يكتثر المعيار لمشكلة القابلية مع البرامج القديمة بالنظر إلى أن هذه البرامج تستخدم طرقًا غير قابلة للنقل في المقام الأول، وسيحسن تواجد هذه الخصائص المتقدمة ضمن المعيار سهولة نقل برامج لغة سي مستقبلاً بصورة ملحوظة.

يعدّ استخدام المعالج المسبق سهلاً إذا استُخدم لمهمته الأساسية البسيطة في جعل البرامج سهلة القراءة والصيانة، ولكن يُفضّل ترك خصائصه المتقدمة لاستخدام الخبراء. بحكم تجربتنا، يُعد استخدام `#define` ومجموعة تعليمات التصريف الشرطي `conditional compilation` (أوامر `#if`) مناسباً للمبتدئين، وإذا ما زلت مبتدئ في لغة سي، فاقراً هذا الفصل مرةً واحدة لمعرفة إمكانيات المعالج المُسبق واستخدم التمارين للتأكد من فهمك، وإلا فنحن ننصح بخبرة لا تقل عن ستة أشهر في لغة سي حتى تستطيع فهم إمكانيات المعالج المسبق كاملةً، لذا لن نركز على منحك مقدمة سهلة هنا بل سنركز على التفاصيل الدقيقة فوراً.

7.2 كيف يعمل المعالج المسبق؟

بالرغم من أن المعالج المسبق الموضح في الشكل التالي سينتهي به المطاف غالباً بكونه جزءاً مهمّاً من مصرف لغة سي المعيارية، إلا أنه يمكننا التفكير به على أنه برنامجٌ منفصلٌ يحول شيفرة سي المصدرية التي تحتوي على موجّهات المعالج المسبق إلى شيفرة مصدرية لا تحتوي على هذه الموجّهات.



الشكل 12: المعالج المسبق في لغة سي

من المهم هنا أن نتذكر أن المعالج المسبق لا يعمل متبعاً القوانين الخاصة بشيفرة لغة سي ذاتها، وإنما يعمل على أساس كل سطرٍ بسطره، وهذا يعني أن نهاية السطر حدثٌ مميز وليس كما تنظر لغة سي إلى نهاية السطر بكونه مشابهاً لمحرف مسافة أو مسافة جدولة.

لا يعي المعالج المسبق قوانين لغة سي الخاصة بالنطاق `Scope`، إذ تأخذ موجّهات المعالج المسبق (مثل `#define`) تأثيرها فور رؤيتها ويبقى تأثيرها موجوداً حتى الوصول إلى نهاية الملف الذي يحتوي هذه الموجّهات، ولا ينطبق هنا هيكل البرنامج المتعلق بالكتل البرمجية. من المحبذ إذًا استخدام موجّهات المعالج المسبق بأقل ما أمكن، فكلما قلّ عدد الأجزاء التي لا تتبع قوانين النطاق "الاعتيادية" كلما قلت إمكانية ارتكاب الأخطاء، وهذا ما نقصده عندما نقول أن تكامل المعالج المسبق ولغة سي C محدودٌ فيما بينهما.

يصف المعيار بعض القوانين المعقدة بخصوص كتابة موجّهات المعالج المسبق، وبالأخص بالنسبة للمفاتيح `Tokens`، ويجب عليك معرفة القوانين كلها إذا أردت فهم موجّهات المعالج المسبق، فالنص الذي يُعالج لا يعدّ سلسلةً من المحارف بل هو مُجزأٌ إلى مفاتيح ومن ثم معلومات معالجة مُجزأة.

من الأفضل اللجوء إلى المعيار إذا أردت تعريفاً كاملاً بالعملية، إلا أننا سنتطرق إلى شرح بسيط؛ إذ سنشرح كل جزء موجود في القائمة التالية لاحقاً.

1. اسم ملف الترويسة

◦ < يمكن استخدام أي محرف هنا (باستثناء) >.

2. مفتاح المعالج المسبق

- اسم ملف الترويسة كما دُكر سابقًا لكن فقط في حالة ذكره ضمن `#include`
 - أو معرف `identifier` مثل معرف لغة C أو كلمة مفتاحية
 - أو ثابت وهو أي عدد صحيح أو طبيعي ثابت
 - أو سلسلة نصية وهو سلسلة نصية سي اعتيادية
 - أو عامل وهو من أحد عوامل لغة سي
 - أو واحد من علامات الترقيم `[] () { } * , : = ; ... #`
 - أو أي محرف غير فارغ (محرف فارغ مثل محرف المسافة) غير مذكور في اللائحة أعلاه
- نقصد أي محرف (باستثناء) أي باستثناء المحرفين > أو محرف السطر الجديد.

7.3 الموجهات Directives

تبدأ موجهات المعالج المسبق بالمحرف "#" دائمًا، وتُتبع بمحرف مسافة فارغة اختياريًا إلا أن هذا الاستخدام غير شائع، ويوضح الجدول التالي الموجهات المعروفة في المعيار.

الجدول 16: موجهات المعالج المُسبق

المعنى	الموجه
تضمين ملف مصدري	<code># include</code>
تعريف ماكرو	<code># define</code>
التراجع عن تعريف ماكرو	<code># undef</code>
تصريف شرطي	<code># if</code>
تصريف شرطي	<code># ifdef</code>
تصريف شرطي	<code># ifndef</code>
تصريف شرطي	<code># elif</code>
تصريف شرطي	<code># else</code>
تصريف شرطي	<code># endif</code>
التحكم بتقارير الأخطاء	<code># line</code>
عرض رسالة خطأ قسرية	<code># error</code>
تُستخدم للتحكم المعتمد على التنفيذ	<code># pragma</code>
موجه فارغ؛ دون تأثير	<code>#</code>

المعنى	الموجّه
--------	---------

سنشرح كل من معنى واستخدام الموجّهات بالتفصيل في الفقرات التالية. لاحظ أن المحرف # والكلمة المفتاحية التي تليه عنصرين مستقلين منفصلين، ويمكن إضافة مسافة بيضاء بينهما.

7.3.1 الموجّه الفارغ

هذا الموجّه بسيط، إذ ليس لإشارة # بمفردها على السطر أي تأثير.

7.3.2 موجّه تعريف الماكرو define

هناك طريقتان لتعريف الماكرو، أولهما تبدو مثل تابع والأخرى على النقيض، إليك مثالاً باستخدام الطريقتين:

```
#define FMAC(a,b) a here, then b
```

```
#define NONFMAC some text here
```

يعرّف التصريحان السابقان ماكرو ونصاً بديلاً له، إذ سيستخدم ليُستبدل بالماكرو المذكور ضمن كامل البرنامج، ويمكن استخدامهما بعد التصريح عنهما على النحو التالي مع ملاحظة أثر استبدال الماكرو الموضح في التعليقات:

```
NONFMAC
```

```
/* النص هنا */
```

```
FMAC(first text, some more)
```

```
/* النص الأول، ومزيد من النص */
```

يُستبدل اسم الماكرو في الحالة التي لا يبدو فيها مثل دالة بالنص البديل ببساطة، وكذلك الأمر بالنسبة لماكرو من نوع دالة، وفي حال كان النص البديل يحتوي على معرّف يطابق اسم معامل من معاملات الماكرو، يُستخدم النص الموجود وسيطاً بدلاً من المعرّف في النص البديل. يُحدّد نطاق الأسماء المذكورة في وسطاء الماكرو بالكتلة التي تحتوي الموجّه #define.

تُهمل أي مسافات فارغة بعد أو قبل النص البديل ضمن تعريف الماكرو، وذلك لكلا الطريقتين على حد سواء.

يتبادر إلى البعض السؤال الفضولي التالي: كيف يمكنني تعريف ماكرو بسيط بحيث يكون النص البديل الخاص به ينتهي بالقوس المفتوح ("")؟ الإجابة بسيطة، إذا احتوى تعريف الماكرو على مسافة أمام القوس ("")،

فلن يحتسب الماكرو من نوع ماكرو دالة، بل نص بديل لماكرو بسيط فحسب، إلا أنه لا يوجد قيد مماثل عندما تستخدم الماكرو الشبيه بالدالة.

يسمح المعيار للماكرو بغض النظر عن نوع أن يُعاد تعريفه في أي لحظة باستخدام موجّه `define` # آخر، وذلك بفرض عدم تغيير نوع الماكرو وأن تكون المفاتيح التي تشكل التعريف الأساسي وإعادة التعريف متماثلة بالعدد والترتيب والكتابة بما فيها استخدام المسافة الفارغة، وتُعد المسافات الفارغة في هذا السياق متساوية، وطبقاً لذلك فالتالي صحيح:

```
# define XXX abc /*تعليق*/def hij
# define XXX abc def hij
```

وذلك لأن التعليق شكّل من أشكال المسافات الفارغة، وسلسلة المفاتيح للحالتين السابقتين هي:

```
# w-s define w-s XXX w-s abc w-s def w-s hij w-s
```

إذ تعني `w-s` مفتاح مسافة بيضاء.

1. استبدال الماكرو

أين سيتسبب اسم الماكرو باستبدال النص بالنص البديل؟ يحدث الاستبدال عملياً في أي مكان يحدث التعرف فيه على المعرّف `identifier` مثل مفتاح منفصل ضمن البرنامج، عدا المعرف المتبوع بالمحرف `"#"` الخاص بموجّه المعالج المُسبق. يمكنك كتابة التالي:

```
#define define XXX
#define YYY ZZZ
```

ومن المتوقع أن يتسبب استبدال سطر `#define` الثاني بالسطر `#xxx` بخطأ.

يُستبدل المعرف المرتبط بماكرو بسيط عندما يُرى بمفتاح الماكرو البديل، ومن ثم يُعاد مسحه `rescanned` (سننكلم عن ذلك لاحقاً) للعثور على أي استبدالات أخرى.

يمكن استخدام ماكرو الدالة مثل أي دالة أخرى اعتيادية، وذلك بوضع مساحات فارغة حول اسم الماكرو ولائحة الوسطاء وغيره، كما قد يحتوي على محرف سطر جديد:

```
#define FMAC(a, b) printf("%s %s\n", a, b)

FMAC ("hello",
      "sailor"
    );
```

```
/* ينتج ما سبق بالتالي */
printf("%s %s\n", "hello", "sailor")
```

يمكن أن تأخذ وسطاء الماكرو من نوع دالة أي تسلسل عشوائي للمفتاح، وتستخدم الفاصلة ", " لفصل الوسطاء عن بعضهم بعضًا، ولكن يمكن إخفاؤها بوضعها داخل أقواس " () ". توازن الأزواج المتطابقة من الأقواس داخل الوسيط بعضها بعضًا، وبالتالي ينهي القوس " () " استدعاء الماكرو إذا كان القوس " () " هو الذي بدأ باستدعائه.

```
#define CALL(a, b) a b

CALL(printf, ("%d %d %s\n", 1, 24, "urgh"));
/* results in */
printf ("%d %d %s\n", 1, 24, "urgh");
```

لاحظ كيف حافظنا على الأقواس حول الوسيط الثاني للدالة CALL عند الاستبدال، ولم تُزال من النص. إذا أردت استخدام ماكرو مثل printf، لن يساعدك المعيار بهذا الخصوص عندما تختار عدد متغير من الوسطاء، فذلك غير مدعوم.

نحصل على سلوك غير معرف إذا لم يحتوي أحد الوسطاء على مفتاح معالج مسبق، والأمر مماثل إذا احتوت سلسلة مفاتيح المعالج المسبق التي تشكل الوسيط على موجه معالج مسبق مغاير:

```
#define CALL(a, b) a b

/* كل حالة تنتج عن سلوك غير محدد */
CALL(,hello)
CALL(xyz,
#define abc def)
```

إلا أننا نعتقد برأينا أن الاستخدام الثاني الخاطئ للدالة CALL يجب أن ينتج بسلوك معرف، إذ أن أي أحد قادر على كتابة ذلك سيستفيد من انتباه المصرف.

تتبع معالجة ماكرو الدالة الخطوات التالية:

1. جميع وسطائها معرفة.
2. إن كان أي من المفاتيح ضمن الوسيط مرشح لاستبدال بواسطة ماكرو، فسيُستبدل حتى الوصول للنقطة التي لا يمكن فيها إجراء المزيد من الاستبدالات، باستثناء الحالات المذكورة في البند الثالث

التالي. لا يوجد هناك أي خطر بخصوص امتلاك الماكرو لعدد مختلف من الوسطاء بعد إضافة فاصلة إلى قائمة الوسطاء الأساسية، إذ يُحدد الوسطاء في الخطوة السابقة **فقط**.

3. تُستبدل المعرفات التي تسمي وسيط الماكرو في نص الاستبدال بسلسلة مفاتيح مثل وسيط فعلي، ويُهمل الاستبدال إذا كان المعرف مسبقاً بإشارة "#" أو اثنتين "##" أو متبوعاً بالإشارتين "##".

ب. التنصيص

هناك طريقة خاصة لمعالجة الأماكن التي يسبق فيها وسيط الماكرو الإشارة "#" في نص الماكرو البديل، إذ تهمل أي مسافة فارغة تسبق أو تلي قائمة الوسطاء الفعلية للمفتاح، ومن ثم تُحوّل قائمة المفاتيح والإشارة # إلى سلسلة نصية واحدة، وتُعامل المسافات بين المفاتيح كأنها محارف مسافة في سلسلة نصية؛ ولمنع حدوث أي نتائج مفاجئة، يُسبق أي محرف " \ أو \ في السلسلة النصية الجديدة بالمحرف \.

إليك المثال التالي الذي يوضح الخاصية المذكورة:

```
#define MESSAGE(x) printf("Message: %s\n", #x)

MESSAGE (Text with "quotes");

/*
 * النتيجة هي
 * printf("Message: %s\n", "Text with \"quotes\"");
 */
```

ج. لصق المفاتيح Token pasting

قد نجد العامل ## في أي مكان ضمن النص البديل للماكرو باستثناء نهايته أو بدايته، وتُستخدم سلسلة مفاتيح وسيط الماكرو لاستبدال النص البديل إذا ورد في اسم الوسيط لماكرو دالة مسبقاً أو متبوعاً بأحد هذه العوامل، وتُدمج المفاتيح المحيطة بالعامل ## سويّاً سواءً كانت ضمن ماکرو دالة أو ماکرو بسيط، ونحصل على سلوك غير معرف إذا شكّل ذلك مفتاح غير صالح، ويُجرى إعادة مسح بعدها.

إليك عملية تحصل على عدة مراحل يُستخدم فيها إعادة المسح لتوضيح لصق المفاتيح:

```
#define REPLACE some replacement text
#define JOIN(a, b) a ## b

JOIN(REP, LACE)
becomes, after token pasting,
REPLACE
becomes, after rescanning
```

```
some replacement text
```

د. إعادة المسح

يُسمح النص البديل مضافاً إلى مفاتيح الملف المصدري مجدداً حالما تحصل العملية الموضحة في الفقرة السابقة، وذلك للبحث عن أسماء ماكرو أخرى لاستبدالها، مع استثناء أن اسم الماكرو داخل النص البديل لا يُستبدل. من الممكن أن نضيف ماكرو متداخلة Nested macros وبالتالي يمكن لعدد من الماكرو أن تُعالج لاستبدالها في أي نقطة دفعةً واحدة، ولا يوجد في هذه الحالة أي اسم مرشح لاستبداله ضمن المستوى الداخلي لهذا التداخل، وهذا يسمح لنا بإعادة تعريف الدوال الموجودة سابقاً مثل ماكرو:

```
#define exit(x) exit((x)+1)
```

تصبح أسماء الماكرو التي لم تُستبدل مفاتيحٍ محمية من الاستبدال مستقبلاً، حتى لو وردت أي عمليات تالية لتبديلها، وهذا يدرأ الخطر عن حدوث التعادوية recursion اللانهائية في المعالج المسبق، وتنطبق هذه الحماية فقط في حالة نتج اسم الماكرو عن النص البديل، وليس النص المصدري ضمن البرنامج، إليك ما الذي نقصده:

```
#define m(x) m((x)+1)
/* هذا */
m(abc);
/* ينتج عن هذا بعد الاستبدال */
m((abc)+1);
/*
على الرغم من أن النتيجة السابقة تبدو مثل ماكرو
إلا أن القواعد تنص على لزوم عدم استبداله
*/

m(m(abc));
/*
تبدأ m (الخارجية باستدعاء الماكرو،
لكن تُستبدل الداخلية أولاً
* لتصبح بالشكل m((abc)+1)
وُستخدم مثل وسيط، مما يعطينا
*/
m(m((abc+1)));
/*
ويصبح بعد الاستبدال على النحو التالي
*/
```

```
*/
m((m((abc+1))+1));
```

إذا لم يؤلمك دماغك بقراءة ما سبق، فإذهب واقرأ ما الذي يقوله المعيار عن هذا ونضمن لك أنه سيؤلمك.

ه. ملاحظات

هناك مشكلة غير واضحة تحدث عند استخدام وسطاء ماكرو الدالة.

```
/* تحذير: هناك مشكلة في هذا البرنامج */
#define SQR(x) ( x * x )
/*
* عند ورود المعاملات الصورية في النص البديل، تُستبدل بالمعاملات الفعلية للماكرو
*/
printf("sqr of %d is %d\n", 2, SQR(2));
```

المعامل الصوري formal parameter للماكرو SQR هو x ، والمعامل الفعلي actual argument هو 2، وبالتالي سينتج النص البديل عن:

```
printf("sqr of %d is %d\n", 2, ( 2 * 2 ));
```

لاحظ استخدام الأقواس، فالمثال التالي قد يتسبب بمشكلة:

```
/* مثال سيء */
#define DOUBLE(y) y+y

printf("twice %d is %d\n", 2, DOUBLE(2));
printf("six times %d is %d\n", 2, 3*DOUBLE(2));
```

تكمّن المشكلة في أن التعبير الأخير في استدعاء الدالة printf الثاني يُستبدل بالتالي:

```
3*2+2
```

وهذا ينتج عن 8 وليس 12. تنص القاعدة على أنه يجب عليك الحرص بخصوص الأقواس فهي ضرورية في حالة استخدام الماكرو لبناء تعابير. إليك مثلاً آخر:

```
SQR(3+4)

/* تصبح بالشكل التالي بعد الاستبدال */
( 3+4 * 3+4 )
```

```
/* للأسف، ما زالت خاطئة! */
```

لذا، يجب عليك النظر بحرص إلى الوسطاء الصورية عندما ترد ضمن نصل بديل. إليك الأمثلة الصحيحة عن الدالتين SQR و DOUBLE:

```
#define SQR(x) ((x)*(x))
#define DOUBLE(x) ((x)+(x))
```

في جعبة الماكرو حيلة صغيرة بعد لمفاجئتك، كما سيوضح لك المثال التالي:

```
#include <stdio.h>
#include <stdlib.h>
#define DOUBLE(x) ((x)+(x))

main(){
    int a[20], *ip;

    ip = a;
    a[0] = 1;
    a[1] = 2;
    printf("%d\n", DOUBLE(*ip++));
    exit(EXIT_SUCCESS);
}
```

[مثال 1]

لم يتسبب المثال السابق بمشاكل؟ لأن نص ماکرو البديل يشير إلى `*ip++` مرتين، مما يتسبب بزيادة `ip` مرتين، لا يجب للماکرو أن يُستخدم مع التعابير التي لها آثار جانبية، إلا إذا تحققت بحرص من أمانها. بغض النظر عن هذه التحذيرات التي تخص الماکرو، إلا أنها تقدم خصائص مفيدة، وستستخدم هذه الخصائص كثيرًا من الآن فصاعدًا.

7.3.3 موجّه التراجع عن تعريف ماکرو `undef`

يُمكن أن يُهمل (يُنسى) أي معرف يعود لموجه `#define` بكتابة:

```
#undef NAME
```

إذ لا يولد `#undef` خطأً إن لم يكن الاسم `NAME` معرفًا مسبقًا.

سنستفيد من هذا الموجه كثيرًا عمّا قريب، وستتكمّل لاحقًا عن بعض دوال المكتبات التي هي في الحقيقة ماكرو وليس دالة، وستصبح قادرًا على الوصول إلى الدالة الفعلية عن طريق التراجع عن تعريفها.

7.3.4 موجه تضمين ملف مصدري include

يمكن كتابة هذا الموجه بشكلين:

```
#include <filename>
#include "filename"
```

ينجم عن استخدام أحد الطريقتين قراءة ملف جديد عند نقطة ذكر الموجه، وكأننا استبدلنا سطر الموجه بمحتويات الملف المذكور، وإذا احتوى هذا الملف على بعض التعليمات الخاطئة ستظهر لك الأخطاء مع إشارتها إلى الملف التي نجمت عنه مصحوبةً برقم السطر، وهذه مهمة مطوّر المصرف، وينص المعيار على أنه يجب للمصرف دعم ثمانية طبقات من موجّهات `include` المتداخلة على الأقل.

يمكن الاختلاف بين استخدام `<>` و `" "` حول اسم الملف بالمكان الذي سيُبحث فيه عن الملف؛ إذ يتسبب استخدام الأقواس في البحث في عددٍ من الأماكن المعرّفة بحسب التطبيق؛ بينما يتسبب استخدام علامتي التنصيص بحثًا في المكان المرتبط بمكان ملف الشيفرة المصدرية، وستُعلمك ملاحظات تطبيقك بما هو المقصود بكلمة "المكان" والتفاصيل المرتبطة بها، إذا لم تعود عملية البحث عن الملف باستخدام علامتي التنصيص بأي نتيجة، تُعاود عملية البحث من جديد وكأنك استخدمت القوسين.

تُستخدم الأقواس عمومًا عندما تريد تحديد ملفات ترويسة لمكتبة قياسية `Standard library`، بينما تُستخدم علامتي التنصيص لملفات الترويسة الخاصة بك، التي تكون مخصصة غالبًا لبرنامج واحد.

لا يحدد المعيار كيفية تسمية ملف بصورةٍ صالحة، إلا أنه يحدد وجوب وجود طريقة فريدة معرفة بحسب التطبيق لترجمة اسم الملفات من الشكل `xxx.x` (يمثل كل `x` حرفًا)، إلى أسماء ملفات الشيفرة المصدرية، ويمكن تجاهل الفرق بين الأحرف الكبيرة والصغيرة من قبل التطبيق، ويمكن أن يختار التطبيق أيضًا ستة محارف ذات أهمية فقط بعد محرف النقطة ..

يمكنك أيضًا الكتابة بالشكل التالي:

```
# define NAME <stdio.h>
# include NAME
```

للحصول على نتيجة مماثلة لهذا:

```
# include <stdio.h>
```


إلا أن هذه الطريقة تعقيدٌ لا داعي له، وهي معرضةٌ لبعض الأخطاء طبقًا للقواعد المعرفة بحسب التطبيق إذ تحدد هذه القواعد كيف سيُعالج النص بين القوسين < > .

من الأبسط أن يكون النص البديل للماكرو NAME سلسلة نصية، على سبيل المثال:

```
#define NAME "stdio.h"
```

```
#include NAME
```

لا يوجد في حالتنا السابقة أي فرصة للأخطاء الناتجة عن التصرف المعرف بحسب التطبيق، إلا أن مسارات البحث مختلفة كما وضعنا سابقًا. سلسلة المفتاح في حالتنا الأولى التي تستبدل NAME هي على النحو التالي (بحسب القوانين التي ناقشناها سابقًا):

```
<
stdio
.
h
>
```

أما في الحالة الثانية فهي من الشكل:

```
"stdio.h"
```

الحالة الثانية سهلة الفهم، لأنه يوجد لدينا فقط سلسلة نصية وهي مفتاح تقليدي لموجه #include، بينما الحالة الثانية معرفة بحسب التطبيق، وبالتالي يعتمد تشكيل سلسلة المفاتيح لاسم ترويسة صالح على التطبيق.

أخيرًا، المحرف الأخير من الملف المُضمّن داخل موجه include يجب أن يكون سطرًا جديدًا، وإلا سنحصل على خطأ.

7.3.5 الأسماء مسبقة التعريف

الأسماء التالية هي أسماء مسبقة التعريف predefined names داخل المعالج المُسبق:

- الاسم __LINE__: ثابت عدد صحيح بالنظام العشري، ويشير إلى السطر الحالي ضمن ملف الشيفرة المصدرية.
- الاسم __FILE__: اسم ملف الشيفرة المصدرية الحالي، وهو سلسلة نصية.
- الاسم __DATE__: التاريخ الحالي، وهو سلسلة نصية من الشكل:

Apr 21 1990

إذ يظهر اسم الشهر كما هو معرّف في الدالة المكتبية `asctime` وأول خانة من التاريخ مسافة فارغة إذ أن كان التاريخ أقل من 10.

- الاسم `__TIME__`: وقت ترجمة الملف، وهو سلسلة نصية موافقة للشكل السابق باستخدام الدالة `asctime`، أي من الشكل "hh:mm:ss".
- الاسم `__STDC__`: عدد صحيح ثابت بقيمة 1، ويُستخدم لاختبار اتباع المصنّف لضوابط المعيار، إذ يمتلك هذا العدد قيمًا مختلفة لإصدارات مختلفة من المعيار.

الطريقة الشائعة في استخدام هذه الأسماء المعرفة مسبقًا هي على النحو التالي:

```
#define TEST(x) if(!(x))\
    printf("test failed, line %d file %s\n",\
        __LINE__, __FILE__)\

/**/

TEST(a != 23);

/**/
```

[مثال 2]

إذا كانت نتيجة `TEST` في المثال السابق خطأ، فستُطبع الرسالة متضمنةً اسم الملف ورقم السطر في الرسالة. إلا أن استخدام تعليمة `if` في هذه الحالات قد يتسبب ببعض من اللبس، كما يوضح المثال التالي:

```
if(expression)
    TEST(expr2);
else
    statement_n;
```

إذ سترتبط تعليمة `else` بتعليمة `if` المخفية التي ستُستبدل باستخدام الماكرو `TEST`، وعلى الرغم من أن حدوث هذا الشيء عند الممارسة مستبعد إلا أنه سيكون خطأً لعيّنًا صعب الحل والتشخيص إذا حدث لك، ولتفادي ذلك، يُحبذ استخدام الأقواس وجعل محتوى كل تعليمة تحكم بتدفق البرنامج مثل تعليمة مركّبة بغض النظر عن طولها.

لا يمكننا استعمال أي من الأسماء `__LINE__` أو `__FILE__` أو `__DATE__` أو `__TIME__` أو `__STDC__` أو أي من الأسماء المعرفة الأخرى ضمن موجه `#define` أو `#undef`.

ينص المعيار على أن أي اسم محجوز يجب أن يبدأ بشرطةٍ سفلية `underscore` وحرفٍ كبير، أو شرطتان، وبالتالي يمكنك استخدام أي اسم لاستخدامك الخاص، لكن انتبه من استخدام الأسماء المحجوزة في ملفات الترويسة التي ضمّنتها في برنامجك.

7.3.6 موجه التحكم بتقارير الأخطاء line

يُستخدم هذا الموجه في ضبط القيمة التي يحملها كل من `__LINE__` و `__FILE__`، لكن ما المُستفاد من ذلك؟ تولّد العديد من الأدوات في الوقت الحالي شيفرةً بلغة سي C خرجًا لها، ويسمح هذا الموجه لهذه الأدوات بالتحكم برقم السطر الحالي إلا أن استخدامه محدودٌ لمبرمج لغة سي الاعتيادية.

يأتي الموجه بالشكل التالي:

```
# line number optional-string-literal newline
```

يُضبط الرقم `number` قيمة `__LINE__` وتُضبط السلسلة النصية الاختيارية إن وُجدت قيمة `__FILE__`. في الحقيقة، سنُوسّع سلسلة المفاتيح التي تتبع الموجه `#line` باستخدام ماكرو، ومن المفترض أن تشكّل موجهًا صالحًا بعد التوسعة.

7.3.7 التصريف الشرطي

يتحكم بالتصريف الشرطي عدد من الموجهات، إذ تسمح هذه الموجهات بتصريف أجزاء معينة من البرنامج اختياريًا أو تجاهلها حسب الشروط، وهذه الشروط هي: `#if` و `#ifdef` و `#ifndef` و `#elif` و `#else` و `#endif` إضافةً إلى عامل المُعالج المسبق الأحادي.

يمكننا استخدام الموجهات على النحو التالي:

```
#ifdef NAME
/* صرّف هذا القسم إذا كان الاسم معرفًا */
#endif
#ifndef NAME
/* صرّف هذا القسم إذا كان الاسم غير معرفًا */
#else
/* صرّف هذا القسم إذا كان الاسم معرفًا */
#endif
```

يُستخدم كل من `#ifdef` و `#endif` لاختبار تعريف اسم الماكرو، ويمكن استخدام `#else` طبقاً مع `#ifdef` (و `#if` أو `#elif` أيضاً)، لا يوجد التباس حول استخدام `#else`، لأن استخدام `#endif` يحدّد نطاق الموجّه مما يبعد أي مجال للشبهات. ينص المعيار على وجوب دعم ثمان طبقات من الموجّهات الشرطية المتداخلة، إلا أنه من المستبعد وجود أي حدّ عملياً.

تُستخدم هذه الموجّهات لتحديد فقرات صغيرة من برامج سي التي تعتمد على الآلة التي تعمل عليها (عندما لا يكون بالإمكان جعل كامل البرنامج ذا أداء مستقل غير مرتبط بطبيعة الآلة التي يعمل عليها)، أو لتحديد خوارزميات مختلفة تعتمد على بعض المقايضات لتنفيذها.

تأخذ بنية `#if` و `#elif` تعبيراً ثابتاً وحيداً ذا قيمة صحيحة، وقيم المعالج المسبق هذه مماثلة للقيم الاعتيادية باستثناء أنها يجب أن تخلو من عوامل تحويل الأنواع `casts`. تخضع سلسلة المفاتيح التي تشكّل التعبير الثابت لعملية استبدال بالماكرو، عدا الأسماء المسبوقة بموجه التعريف فلا تُستبدل. بالاعتماد على ما سبق ذكره، فالتعبير `defined NAME` أو `(NAME) defined` يُقيّمان إلى القيمة 1 إذا كان `NAME` معرفاً، وإلى القيمة 0 إن لم يكن معرفاً، وتُستبدل جميع المعرفات ضمن التعبير -بما فيها كلمات لغة سي المفتاحية- بالقيمة 0، ومن ثم يُقيّم التعبير. يُقصد بالاستبدال (بما فيه استبدال الكلمات المفتاحية) أن `sizeof` لا يمكن استخدامه في هذه التعابير للحصول على القيمة التي تحصل عليها في الحالة الاعتيادية.

تُستخدم القيمة الصفر -كما هو الحال في تعليمات سي الشرطية- للدلالة على القيمة "خطأ `false`"، وتدل أي قيمة أخرى على القيمة "صواب `true`".

يجب استخدام المعالج المسبق العمليات الحسابية وفق النطاقات المحددة في ملف الترويسة `<limits.h>`، وأن تُعامل التعابير ذات قيمة العدد الصحيح مثل عدد صحيح طويل والعدد الصحيح عديم الإشارة مثل عدد صحيح طويل عديم الإشارة، بينما لا يتوجب على المحارف أن تكون مساوية إلى القيمة ذاتها عند وقت التنفيذ، لذا من الأفضل في البرامج القابلة للنقل أن نتجنب استخدامها في تعابير المعالج المُسبق. تعني القوانين السابقة عموماً أنه بإمكاننا أن نحصل على نتائج حسابية من المعالج المسبق التي قد تكون مختلفة عن النتائج التي نحصل عليها عند وقت التنفيذ، وذلك بفرض إجراء الترجمة والتنفيذ على آلات مختلفة. إليك مثلاً:

```
#include <limits.h>

#if ULONG_MAX+1 != 0
    printf("Preprocessor: ULONG_MAX+1 != 0\n");
#endif

if(ULONG_MAX+1 != 0)
```

```
printf("Runtime: ULONG_MAX+1 != 0\n");
```

[مثال 3]

من الممكن أن يُجري المعالج المسبق بعض العمليات الحسابية بنطاق أكبر من النطاق المُستخدم في البيئة المُستهدفة، ويمكن في هذه الحالة ألا يطفح تعبير المعالج المسبق `ULONG_MAX+1` ليعطي النتيجة 0، بينما يجب أن يحدث لك في بيئة التنفيذ.

يوضح المثال التالي استخدام الثوابت المذكور آنفًا، وتعليلة "وإلا `else`" الشرطية `#elif`.

```
#define NAME    100

#if    ((NAME > 50) && (defined __STDC__))
/* افعل شيئًا */
#elif  NAME > 25
/* افعل شيئًا آخر */
#elif  NAME > 10
/* افعل شيئًا آخر */
#else
/* الاحتمال الأخير */
#endif
```

يتوجب التنويه هنا على أن موجّهات التصريف الشرطية لا تتبع لقوانين النطاق الخاصة بلغة سي، ولهذا فيجب استخدامها بحرص، إلا إذا أردت أن يصبح برنامجك صعب القراءة، إذ من الصعب أن تقرأ برنامج سي C مع وجود هذه الأشياء كل بضعة أسطر، وسيتملك الغضب تجاه كاتب البرنامج إذا صادفت شيفرة مشابهة لهذه دون أي موجّه `#if` واضح بالقرب منها:

```
#else
}
#endif
```

لذلك يجب أن تعامل هذه الموجّهات معاملة الصلصة الحارّة، استخدامها ضروري في بعض الأحيان، إلا أن الاستخدام الزائد لها سيعقّد الأمور.

7.3.8 موجه التحكم المعتمد على التنفيذ pragma

كان هذا الموجه بمثابة محاولة للجنة المعيار بإضافة طريقة للولوج للباب الخلفي، إذ يسمح هذا الموجه بتنفيذ بعض الأشياء المعرفة بحسب التطبيق. إذا لم يعرف التطبيق ما الذي سيفعله بهذا الموجه (أي لم يتعرف على وظيفته) فسيتجاهله ببساطة، إليك مثالاً عن استخدامه:

```
#pragma byte_align
```

يُستخدم الموجه السابق لإعلام التطبيق بضرورة محاذاة جميع أعضاء الهياكل بالنسبة لعنوان البايث الخاص بهم، إذ يمكن لبعض معماريات المعالجات أن تتعامل مع أعضاء الهياكل بطول الكلمة بمحاذاة عنوان البايث، ولكن مع خسارة السرعة في الوصول إليها.

يمكن أن يُفسّر هذا الأمر بحسب تفسير التطبيق له طبعاً، وإن لم يكن للتطبيق أي تفسير خاص به، فلن يأخذ الموجه أي تأثير، ولن يُعد خطأً، ومن المثير للاهتمام رؤية بعض الاستخدامات الخاصة لهذا النوع من الموجهات.

7.3.9 موجه عرض رسالة خطأ قسرية error

يُتبع هذا الموجه بمفتاح أو أكثر في نهاية السطر الخاص به، وتُشكّل رسالة تشخيصية من قبل المصرف تحتوي على هذه المفاتيح، ولا توجد مزيداً من التفاصيل عن ذلك في المعيار. يُمكن استخدام هذا الموجه لإيقاف عملية التصريف على آلة ذات عتاد صلب غير مناسب لتنفيذ البرنامج:

```
#include <limits.h>
#if CHAR_MIN > -128
#error character range smaller than required
#endif
```

سيتسبب ذلك ببعض الأخطاء المتوقعة عند مرحلة التصريف إذا نُقِّد الموجه وعُرضت الرسالة.

7.4 الخاتمة

على الرغم من الخصائص القوية والمرنة التي يوفرها المعالج المسبق، إلا أنها شديدة التعقيد، وهناك عددٌ قليل من الجوانب المتعلقة به ضرورة الفهم، مثل القدرة على تعريف الماكرو ودوال الماكرو، والمستخدم بكثرة في أي برنامج سي تقريباً عدا البرامج البسيطة، كما أن تضمين ملفات الترويسة مهمٌ أيضاً طبعاً.

للتصريف الشرطي استخدامان مهمان، أولهما هو القدرة على تصريف البرنامج بوجود أو بعدم وجود تعليمات الاكتشاف عن الأخطاء Debugging ضمن البرنامج، وثانيهما هو القدرة على تحديد تعليمات معينة يعتمد تنفيذها على طبيعة التطبيق أو الآلة التي يعمل عليها البرنامج.

باستثناء ما ذكرنا سابقاً، من الممكن نسيان المزايا الأخرى التي ذُكرت في الفصل، إذ لن يتسبب التخلي عنها بفقدان كثيرٍ من الخصائص، ولعلّ الحل الأمثل هو تواجد فرد واحد متخصص في المعالج المسبق ضمن الفريق البرمجي لتطوير الماكرو التي تفيد مشروعيًا ما بعينه باستخدام بعض المزايا الغريبة، مثل التنصيب ولصق المفاتيح. سيستفيد معظم مستخدمي لغة سي C أكثر إذا وجّهوا جهدهم بدلاً من ذلك نحو تعلم بعض الأجزاء الأخرى من لغة سي، أو تقنيات التحكم بجودة البرمجيات إذا أتقنوا اللغة.

7.5 التمارين

تهدف هذه التمارين لفحص فهمك لأساسيات المعالج المسبق، وهي مناسبة للمبتدئين، ولن يحتاج معظم المستخدمين لفهم ما هو خارج عن هذه التمارين بخصوص المعالج المسبق.

1. كيف يمكنك استبدال المعرف MAXLEN بالقيمة 100 ضمن برنامج ما؟
2. ما المسبب للمشاكل في تعريف من الشكل `MAXLEN+100 VALUE #define`؟
3. اكتب ماكرو باسم REM يأخذ وسيطين من نوع عدد صحيح ويُعيد الباقي الناتج عن قسمة الوسيط الأول بالوسيط الثاني.
4. أعد كتابة المثال السابق، ولكن استخدم تحويلات الأنواع casts بحيث يمكن استخدام أي نوع حسابي مثل وسيط، وذلك بفرض عدم وجود أي مشاكل طفحان.
5. ما الذي تعنيه الأقواس <> حول اسم ملف موجه `#include`؟
6. ما الذي يحصل عندما نستبدل الأقواس <> بعلامات التنصيب " "؟
7. كيف يمكنك استخدام المعالج المسبق في تحديد أجزاء معينة من النص يعتمد تنفيذها على مواصفات معينة للتطبيق؟
8. ما نوع العمليات الحسابية التي يستخدمها المعالج المسبق؟

دورة تطوير التطبيقات باستخدام لغة بايثون



احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن



8. مواضيع مخصصة عن لغة سي

8.1 مقدمة

استعرضنا في الفصول السابقة مبادئ اللغة وغطينا معظم الجوانب المعرّفة من المعيار تقريبًا، إلا أن هناك بعض التفاصيل المتفرقة التي لا تنطوي تحت أي عنوان محدد، ويأتي هذا الفصل لجمع هذه التفاصيل المتفرقة العويصة من لغة سي C. استعدّ للفصل القادم ولا تتردد بتدوين الملاحظات التي تعتقد أنها ستهمّك، وقرأها من وقتٍ لآخر، فالأمر الذي تجده للمرة الأولى غير مثير للاهتمام وصعب الفهم سيصبح ضروريًا ومفيدًا لك بعد أن تكتسب الخبرة الكافية لتوظيفه. سنناقش في هذا الفصل بعض المشاكل شائعة الحدوث وبعض التفاصيل الأخرى الاستثنائية.

8.2 التصاريح declarations والتعاريف definitions وإمكانية الوصول

accessibility

قدمنا سابقًا مفهوم النطاق `scope` والربط `linkage`، ووضحنا كيف يمكن استخدامهما سوياً للتحكم بقابلية الوصول لأجزاء معينة ضمن البرنامج، وعمدنا إلى إعطاء وصف غامض لما يحدّد التعريف `Definition` لأن شرح ذلك سيتسبب بتشويشك في تلك المرحلة ولن يكون مثيرًا لرحلة تعلمك، إلا أنه علينا تحديد هذا الأمر في مرحلة من المراحل وهذا ما سنفعله في هذا الفصل، كما سنتكلم عن صنف التخزين `Storage class` لجعل الأمر أكثر إثارة للاهتمام.

لعلك ستجد مزج هذه المفاهيم واستخدامها سويًا معقدًا ومربكًا، وهذا مبررٌ، إلا أننا سنحاول إزالة الغموض بالتكلم عن بعض القوانين المفيدة لاحقًا، لكنك بحاجة لقراءة بعض التفاصيل قبل ذلك على الأقل مرةً واحدةً لتفهم هذه القوانين.

حتى تفهم القوانين جيدًا، عليك أن تفهم ثلاثة مفاهيم مختلفة -ولكن مرتبطة- وهي كما يدعوها المعيار:

- المدة الزمنية duration.

- النطاق scope.

- الربط linkage.

ويشرح المعيار هذه المصطلحات، كما أننا ناقشنا النطاق والربط في الفصل سابق الذكر لكننا سنعيد ذكرهما بصورةٍ مقتضبة.

8.2.1 محددات صف التخزين

تدرج خمس كلمات مفتاحية تحت تصنيف محددات صف التخزين storage class specifiers، أحدها هو typedef، وبالرغم من أنه أشبه باختصار عن شبهه بخاصية، إلا أننا سنناقش هذه الكلمة المفتاحية بالتفصيل لاحقًا. يتبقى لدينا الكلمات المفتاحية auto و extern و register و static.

تساعدك محددات صف التخزين على تحديد نوع التخزين المستخدم لكائنات البيانات، وهناك صف تخزين واحد مسموح به عند التصريح، وهذا الأمر منطقي لأن هناك طريقةً واحدةً لتخزين الأشياء. يُطبق المحدد الافتراضي على عملية التصريح إذا أهمل محدد صف التخزين، ويعتمد اختيار المحدد الافتراضي على نوع التصريح إذا كان خارج دالة (تصريح خارجي) أو داخل الدالة (تصريح داخلي)، إذ أن محدد التخزين الافتراضي للتصريح الخارجي هو extern بينما auto هو محدد التخزين الافتراضي للتصريح الداخلي، والاستثناء الوحيد لهذه القاعدة هو تصريح الدوال، إذ إن قيمة المحدد الافتراضي لها هو extern دائمًا.

يمكن أن يؤثر مكان التصريح ومحددات صف التخزين المستخدمة (أو الافتراضية في حال عدم وجودها) على ربط الاسم، بالإضافة إلى تصريحات الاسم ذاته التي تلي التصريح الأولي، ولحسن الحظ فإن ذلك لا يؤثر على النطاق أو المدة الزمنية. سنستعرض المفاهيم الأسهل أولًا.

1. المدة الزمنية

تصف المدة الزمنية لكائن ما طبيعة تخزينه، وذلك إذا كان حجز المساحة يصبح مرةً واحدةً عند تشغيل البرنامج أو أنه من طبيعة عابرة بمعنى أن مساحته تُحجز وتحرر عند الضرورة. هناك نوعان فقط من المدة الزمنية، هما: **المدة الساكنة static duration** و**المدة التلقائية automatic duration**؛ إذ تعني المدة الساكنة أن مساحة التخزين المحجوزة للكائن دائمة؛ بينما تعني المدة التلقائية أن مساحة التخزين المحجوزة للكائن تُحرر

وتُحجز عند الضرورة، ومن السهل معرفة أي من المديتين ستحصل عليهما، لأنك ستحصل على المدة التلقائية فقط في حالة:

- إذا كان التصريح داخل دالة،
- ولم يكن التصريح يحتوي على أي من الكلمتين المفتاحيتين `static` أو `extern`.
- وليس التصريح تصريحًا لدالة.

ستجد أن معاملات الدالة الصورية تطابق هذه القوانين الثلاث دائمًا، وبذلك فهي ذات مدة تلقائية. مع أن تواجد الكلمة المفتاحية `static` ضمن التصريح يعني أن الكائن ذو مدة ساكنة دون أي شك، إلا أنها ليست الطريقة الوحيدة لإنجاز ذلك، ويسبب هذا بعض اللبس لدى الكثير، وعلينا تقبل هذا الأمر.

تُمنح كائنات البيانات المصرحة داخل الدوال محدد صنف التخزين الافتراضي `auto`، إلا إذا استُخدم محدد آخر لصنف التخزين. لن نحتاج الوصول إلى هذه الكائنات من خارج الدالة في معظم الحالات، لذا ستريد أن تكون **عديمة الربط linkage no**، وفي هذه الحالة نستخدم الحالة الافتراضية `auto` أو محدد صنف التخزين `register storage`، إذ سيعطينا هذا كائنًا عديم الربط وذا مدة تلقائية. لا يمكن تطبيق الكلمة المفتاحية `auto` أو `register` ضمن تصريح يقع خارج دالة ما.

يُعد صنف التخزين `register` مثيرًا للاهتمام، فعلى الرغم من قلة استخدامه هذه الأيام إلا أنه يقترح على المصرف تخزين الكائن في مسجل `register` واحد أو أكثر في الذاكرة والذي ينعكس بالإيجاب على سرعة التنفيذ. لا ينفذ المصرف الأوامر بناءً على هذا الأمر إلا أن متغيرات `register` لا تمتلك أي عنوان (يُمنع استخدام العامل & معها) لتسهيل الأمر وذلك لأن بعض الحواسيب لا تدعم المسجلات ذات العناوين. قد يتسبب التصريح عن عدة كائنات `register` بتأثير عكسي فيبطئ البرنامج بدلاً من تسريعه، وذلك لأن المصرف سوف يضطر إلى حجز مزيدٍ من المسجلات عند الدخول إلى (تنفيذ) الدالة وهي عملية بطيئة، أو لن يكون هناك العدد الكافي من المسجلات المتبقية لتُستخدم في العمليات الحسابية الوسيطة.

يعود الاختيار في استخدام المسجلات إلى الآلة المُستخدمة، ويجب استخدام هذه الطريقة فقط عندما توضح الحسابات أن هذا النوع ضروري لتسريع تنفيذ دالة ما بعينها، وعندها يتوجب عليك فحص البرنامج وتجربته. يجب ألا تصرح عن متغيرات المسجلات أبدًا خلال عملية تطوير البرنامج برأينا، بل تأكد من أن البرنامج يعمل ومن ثم أجر بعض القياسات، وعندها قرّر استخدام هذا النوع من المتغيرات حسب النتائج فيما إذا كان يتسبب استخدامها بتحسين ملحوظ في الأداء، ولكن عليك تكرار العملية ذاتها إذا اتبعت هذه الطريقة في كل نوع من المعالجات المسبقة التي تنقل برنامجك إليها، إذ يحتوي كل نوع من المعالجات المسبقة على خصائص مختلفة.

ملاحظة أخيرة بخصوص متغيرات `register`: يُعد محدد صنف التخزين هذا المحدد الوحيد الممكن استخدامه ضمن نموذج دالة أولي `function prototype` أو تعريف دالة، ويجري تجاهل محدد صنف التخزين

في حالة نموذج الدالة الأولي، بينما يشير تعريف الدالة على أن المعامل الفعلي مخزن في مسجل إذا كان الأمر ممكناً. إليك مثالاً يوضح كيفية يمكن توظيف ذلك:

```
#include <stdio.h>
#include <stdlib.h>

void func(register int arg1, double arg2);

main(){
    func(5, 2);
    exit(EXIT_SUCCESS);
}

/*
توضح الدالة أنه يمكن التصريح عن المعاملات الصورية باستخدام صنف تخزين مسجل
*/

void func(register int arg1, double arg2){

    /*
هذا الاستخدام لأهداف توضيحية، فلا أحد يكتب ذلك في هذا السياق
لا يمكن أخذ عنوان arg1 حتى لو أردت ذلك
*/
    double *fp = &arg2;

    while(arg1){
        printf("res = %f\n", arg1 * (*fp));
        arg1--;
    }
}
```

[مثال 1]

تعتمد إذا المدة الزمنية لكائن على محدد صنف التخزين المستخدم -بغض النظر عن كون الكائن كائن بيانات أو دالة-، كما تعتمد على مكان التصريح (في كتلة داخلية أو على كامل نطاق الملف؟). يعتمد الربط أيضاً على محدد صنف التخزين، إضافةً إلى نوع الكائن ونطاق التصريح عنه. يوضح الجدول 17 والجدول 18 التاليين مدة التخزين الناتجة والربط لكل من الحالات الممكنة عند استخدام محددات صنف التخزين الممكنة، وموضع

التصريح. يُعد ربط الكائنات باستخدام المدة الساكنة أكثر تعقيدًا، لذا ننصحك باستخدام هذه الجداول لتوجيهك في الحالات البسيطة وانتظر قليلاً حتى نصل إلى الجزء الذي نتكلم فيه عن التعريفات.

الجدول 17: التصريحات الخارجية (خارج الدوال)

محدد صنف التخزين	دالة أو كائن بيانات	الربط	المدة الزمنية
static	أحدهما	داخلي	ساكنة
extern	أحدهما	خارجي غالبًا	ساكنة
لا يوجد	دالة	خارجي غالبًا	ساكنة
لا يوجد	كائن بيانات	خارجي	ساكنة

أهمنا في الجدول السابق المحددين `register` و `auto` لأنه من غير المسموح استخدامهما على نطاق التصريحات الخارجية (على نطاق الملف).

الجدول 18: التصريحات الداخلية

محدد صنف التخزين	دالة أو كائن بيانات	الربط	المدة الزمنية
register	كائن بيانات فقط	لا يوجد	تلقائية
auto	كائن بيانات فقط	لا يوجد	تلقائية
static	كائن بيانات فقط	لا يوجد	ساكنة
extern	كلاهما	خارجي غالبًا	ساكنة
لا يوجد none	كائن بيانات	لا يوجد	تلقائية
لا يوجد none	دالة	خارجي غالبًا	ساكنة

تحتفظ المتغيرات الساكنة `static` الداخلية بقيمتها بين استدعاءات الدوال التي تحتويها، وهذا شيء مفيد جدًا في بعض الحالات (راجع الفصل الرابع).

8.2.2 النطاق Scope

علينا الآن إلقاء نظرة على نطاق أسماء الكائنات مجددًا، والذي يعرف أين ومتى يكون لاسم ما معنى محدد. هناك أنواع مختلفة للنطاق هي:

- نطاق دالة.
- نطاق ملف.
- نطاق كتلة.
- نطاق نموذج دالة أولي.

النطاق الأسهل هو نطاق الدالة، إذ ينطبق ذلك على عناوين labels الأسماء المرئية ضمن دالة ما صُرح عنها بداخلها، ولا يمكن لعنوانين داخل نفس الدالة أن يمتلكان الاسم ذاته، لكن يمكن للعنوان استخدام الاسم ذاته إذا كان ضمن أي دالة أخرى بحكم أن النطاق هو نطاق دالة. ليست العناوين كائنات فهي لا تمتلك أي مساحة تخزينية ولا يعني مفهوم الربط والمدة الزمنية أي معنًى في عالمها.

يملك أي اسم مُصرّح عنه خارج الدالة نطاق ملف، وهذا يعني أن الاسم قابلٌ للاستخدام ضمن أي نقطة من البرنامج من لحظة التصريح عنه إلى نهاية ملف الشيفرة المصدرية الذي يحتوي ذلك التصريح، ومن الممكن طبعًا لهذه الأسماء أن تصبح مخفية مؤقتًا باستخدام تصريحات ضمن تعليمات مركبة، لأنه كما نعلم، ينبغي على تعريفات الدالة أن تكون خارج الدوال الأخرى حتى يكون اسم الدالة في التعريف ذو نطاق ملف.

يملك الاسم المُصرّح عنه بداخل تعليمة مركبة أو معامل دالة صوري نطاق كتلة ومن الممكن استخدامه ضمن الكتلة حتى الوصول للقوس } الذي يغلّق التعليمة المركبة، ويُخفي أي تصريح لاسم ضمن تعليمة مركبة أي تصريح خارجي آخر للاسم ذاته حتى نهاية التعليمة المركبة.

يعدّ نطاق النموذج الأولي للدالة مثالاً مميّزاً وبسيطاً من النطاقات، إذ يمتد تصريح الاسم حتى نهاية النموذج الأولي للدالة فقط، وهذا يعني أن ما يلي خاطئ (باستخدام نفس الاسم مرتين):

```
void func(int i, int i);
```

وهذا هو الاستعمال الصحيح:

```
void func(int i, int j);
```

وتختفي الأسماء المُصرّح عنها بداخل الأقواس خارجها. نطاق الاسم مستقل تمامًا عن أي محدد صنف تخزين مُستخدم في التصريح.

8.2.3 الربط

سنذكر بمصطلح الربط Linkage بصورة مقتضبة هنا أيضًا، إذ يُستخدم الربط لتحديد ما الذي يجعل الاسم المُعلن عنه في نطاقات مختلفة يشير إلى الشيء ذاته، إذ يملك الكائن اسمًا واحدًا فقط، ولكننا في بعض الحالات نحتاج للإشارة إلى هذا الكائن على مستوى نطاقات مختلفة، ويُعدّ استدعاء الدالة printf من أماكن متعددة ضمن البرنامج أبسط الأمثلة، حتى لو كانت هذه الأماكن المذكورة لا تنتمي إلى ملف الشيفرة المصدرية ذاته.

يحدّر المعيار أنه يجب على التصاريح التي تشير إلى الشيء ذاته أن تكون من نوع متوافق، وإلا فسنحصل على برنامج ذي سلوك غير محدد، وسنتكلم عن الأنواع المتوافقة بالتفصيل لاحقًا، وسنكتفي حاليًا بقول أن التصاريح يجب أن تكون متطابقةً باستثناء استخدام محددات صنف محدد التخزين، وتحقيق هذا من مسؤوليات المبرمج، إلا أن هناك بعض الأدوات لتساعدك في تحقيق هذا غالبًا.

هناك ثلاثة أنواع مختلفة من الربط:

- الربط الخارجي.
- الربط الداخلي.
- عديم الربط.

إذا كان اسم الكائن ذو ربط خارجي فهذا يعني أن جميع النسخ instances الموجودة في البرنامج -الذي قد يكون مؤلفًا من عدد من ملفات الشيفرة المصدرية والمكتبات- تعود إلى الكائن ذاته، ويعني الربط الداخلي لكائن ما أن نُسخ هذا الكائن الموجودة في ملف الشيفرة المصدرية نفسه فقط تُشير إلى الشيء ذاته، بينما يعني الكائن عديم الربط أن كل كائن ذي اسم مماثل له هو كائنٌ منفصلٌ عنه.

8.2.4 الربط والتعاريف

يجب لكل كائن بيانات أو دالة مُستخدمة في برنامج (عدا معاملات عامل sizeof) أن تمتلك تعريفًا واحدًا فقط، ومع أننا لم نتطرق إلى هذا بعد، إلا أن هذا الأمر مهمٌ جدًّا؛ ويعود السبب بعدم تطرقنا لهذا الأمر إلى استخدام جميع أمثلتنا كائنات بيانات ذات مدة تلقائية فقط وكون تصاريحها تعاريفًا، أو دوالًا كنا قد عرّفناها من خلال كتابة متنها.

تعني القاعدة السابقة أنه يجب للكائنات ذات الربط الخارجي أن تحتوي على تعريفٍ واحد فقط ضمن كامل البرنامج، ويجب للكائنات ذات الربط الداخلي (المقيدة داخل ملف شيفرة مصدرية واحد) أن يُعرّف عنها لمرة واحدة فقط ضمن الملف المُصرّح فيه عنها، كما أن للكائنات عديمة الربط تعريفٌ واحد فقط، إذ أن التصريح عنها هو تعريفٌ أيضًا.

لجمع النقاط الآتف ذكرها، نسأل الأسئلة التالية:

1. كيف أستطيع الحصول على نوع الربط الذي أريده؟

2. ما الشيء الذي يحدد التعريف؟

علينا أن ننظر إلى الربط أولاً ومن ثم التعريف.

إدًا، كيف نحصل على التعريف المناسب لاسم ما؟ القوانين معقدة بعض الشيء.

1. ينتج التصريح خارج دالة (نطاق ملف) تحتوي على محدد صنف تخزين ساكن ربطًا داخليًا لهذا الاسم، ويحدد المعيار وجوب وجود تصاريح الدالة التي تحتوي على الكلمة المفتاحية static على مستوى الملف وخارج أي كتلة برمجية.

2. إذا احتوى التصريح على محدد صنف التخزين extern، أو إذا كان تصريح الدالة لا يحتوي على محدد صنف التخزين، أو كلا الحالتين، فهذا يعني:

1. إذا وجد تصريح للمعرف ذاته بنطاق ملف، فهذا يعني أن الربط الناتج مماثلٌ لهذا التصريح السابق المرئي.
 2. وإلا، فالنتيجة هي ربط خارجي.
 3. إذا لم يكن التصريح ذو نطاق الملف تصريحًا لدالة أو لم يحتوي على محدد صنف تخزين واضح، فالنتيجة هي ربط خارجي.
 4. أي شكل آخر من التصاريح سيكون عديم الربط.
 5. إذا وجد معرف ذو ربط داخلي وخارجي في ذات الوقت ضمن ملف شيفرة مصدرية ما، فالنتيجة غير محددة.
- استُخدمت القوانين السابقة لكتابة جدول الربط السابق (جدول 2) دون تطبيق كامل للقاعدة 2، وهذا السبب في استخدامنا الكلمة "خارجي غالبًا"، وتسمح لك القاعدة 2 بتحديد الربط بدقة في هذه الحالات. ما الذي يجعل التصريح تعريفًا؟

- التصاريح التي تعطينا كائنات عديمة الربط هي تعاريف أيضًا.
 - التصاريح التي تتضمن قيمةً أولية هي تعاريف دائمًا، وهذا يتضمن تهيئة دالة ما بكتابة متن الدالة، ويمكن للتصاريح ذات نطاق الكتلة أن تحتوي على قيم أولية فقط في حال كانت عديمة الربط.
 - وإلا، فالتصريح عن الاسم على نطاق ملف بدون محدد صنف التخزين أو مع محدد صنف التخزين **static** هو **تعريف مبدئي tentative definition**، وإذا احتوى ملف شيفرة مصدرية على تعريف مبدئي واحد أو أكثر لكائن ما وكان الملف لا يحتوي على أي تعاريف فعلية، يصبح للكائن تعريف افتراضي وهو مشابه لحالة إسناد القيمة الأولية "0" إليه (تُهيأ عناصر المصفوفات والهياكل جميعها إلى قيمة "0")، ولا يوجد للدوال تعريف مبدئي.
- طبقًا لما سبق، لا تتسبب التصريحات التي تحتوي على محدد صنف تخزين خارجي **extern** بتعريف، إلا إذا ضمنت قيمةً أوليةً للتصريح.

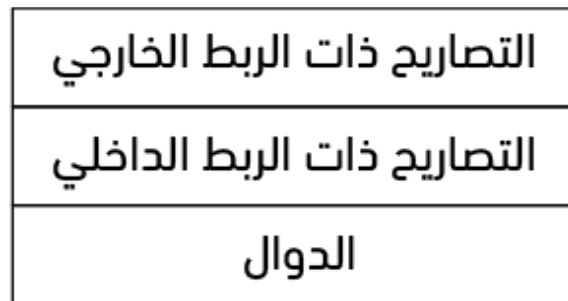
8.2.5 الاستخدام العملي لكل من الربط والتعاريف

- تبدو القوانين التي تحدد كل من الربط والتعريف المرتبطة بالتصاريح معقدةً بعض الشيء، إلا أن استخدام هذه القوانين عمليًا ليس بالأمر الصعب، دعونا نناقش بعض الحالات الاعتيادية.
- أنواع إمكانية أو قابلية الوصول الثلاث التي ستريدها لكائنات البيانات أو الدوال هي:
- على كامل نطاق البرنامج.

- مقيّد بنطاق ملف شيفرة مصدريّة واحد.

- مقيّد بنطاق دالة واحدة، أو تعليمة مركبة واحدة.

ستحتاج إلى ربط خارجي وربط داخلي وربط عديم لكل من الحالات الثلاث السابقة بالترتيب، ومن الممارسات المُحبذة بالنسبة للحالة الأولى والثانية هي التصريح عن الأسماء ضمن ملف الشيفرة المصدريّة الموافق لها قبل أن تعرّف أي دالة، ويوضح الشكل 12 هيكل ملف الشيفرة المصدريّة بهذا الخصوص.



الشكل 13: هيكل ملف الشيفرة المصدريّة

يمكن أن تُسبق تصاريح الربط الخارجية بالكلمة المفتاحية `extern` وتصاريح الربط الداخلية بالكلمة المفتاحية `static`. إليك مثالاً عن ذلك:

```

/* مثال عن هيكل ملف الشيفرة المصدريّة */
#include <stdio.h>

/*
يمكن الوصول للأشياء ذات الربط الخارجي عبر البرنامج
ما يلي هو تصاريح وليس تعاريف، لذا نفترض أن التعاريف في مكان ما آخر
*/

extern int important_variable;
extern int library_func(double, int);

/*
تعاريف ذات ربط خارجي
*/

extern int ext_int_def = 0;      /* تعريف صريح */
int tent_ext_int_def;          /* تعريف مبدئي */

/*

```

```
* يمكن الوصول للأشياء ذات الربط الداخلي فقط من داخل الملف *
* يعني استخدام المحدد الساكن أن التعريفات هي تعريفات مبدئية *
*/

static int less_important_variable;
static struct{
    int member_1;
    int member_2;
}local_struct;

/*
    ذات ربط داخلي لكنها ليست بتعريف مبدئي لأنها دالة
*/
static void lf(void);

/*
    التعريف مع الربط الداخلي
*/
static float int_link_f_def = 5.3;

/*
    وأحيانًا إليك تعريف الدوال ضمن هذا الملف
*/

/*
    للدالة التالية ربط خارجي ويمكن استدعاؤها من أي مكان ضمن البرنامج
*/
void f1(int a){}

/*
    يمكن استخدام الدالتين التاليتين باسمهما ضمن هذا الملف
*/
static int local_function(int a1, int a2){
    return(a1 * a2);
}
```

```

static void lf(void){
    /*
    متغير ساكن عديم الربط، لذا يمكن استخدامه فقط ضمن هذه الدالة،
    وهو تعريف بحكم أنه عديم الربط
    */
    static int count;
    /*
    متغير تلقائي عديم الربط ولكنه ذو مُهيأ بقيمة أولية
    */
    int i = 1;

    printf("lf called for time no %d\n", ++count);
}
/*
صُمّنت التعاريف الفعلية لجميع التعاريف المبدئية المتبقية في نهاية الملف
*/

```

[مثال 2]

نقترح عليك قراءة الفقرات السابقة مجددًا لملاحظة القوانين التي طُبِّقت في المثال 2.

8.3 معرف النوع typedef

يسود الاعتقاد بأن معرّف النوع هو صنف تخزين، إلا أن هذا الاعتقاد خاطئ، إذ يسمح لك هذا المعرّف باختيار مرادفات لأنواع أخرى، والتي من الممكن أن يُصرّح عنها بطريقة مختلفة، ويصبح الاسم الجديد المرادف مكافئاً للنوع الذي تريده، كما سيوضح المثال التالي:

```

typedef int aaa, bbb, ccc;
typedef int ar[15], arr[9][6];
typedef char c, *cp, carr[100];

/* صرّح عن بعض الكائنات */

/* جميع الأعداد الصحيحة */
aaa    int1;
bbb    int2;
ccc    int3;

```

```

ar    yyy;    /* مصفوفة من 15 عدد صحيح */
arr   xxx;    /* مصفوفة من 6×9 عدد صحيح */

c     ch;     /* محرف */
cp    pnt;    /* مؤشر يشير لمحرف */
carr  chry;   /* مصفوفة من 100 محرف */

```

تنص القاعدة العامة في استخدام معرف النوع على كتابة التصريح وكأنك تصرح عن متغيرات من الأنواع التي تريدها، فعندما يتضمن التصريح الاسم مع نوعه المحدد، فإن إسباق ذلك بمعرّف النوع يعني أنك تصرح عن اسم جديد لنوع ما بدلاً من التصريح عن متغيرات، ويمكن بعدها استخدام أسماء النوع الجديد مثل سابقة prefix لتصريح متغير من النوع الجديد.

لا يُعد استخدام الكلمة المفتاحية typedef شائعاً في معظم البرامج، ويُستخدم غالباً في ملفات الترويسة ومن النادر أن تجده في الممارسة اليومية الاعتيادية.

تُعرّف الأنواع الجديدة للمتغيرات الأساسية في البرامج التي تتطلب قابلية نقل كبيرة غالباً، وتُستخدم تعليمات typedef المناسبة لكتابة البرنامج بصورة مُخصصة للآلة الهدف، إلا أن استخدامها الزائد قد يتسبب ببعض اللبس لمبرمجي اللغة إذا كانوا يستخدمون بيئة مغايرة، يوضح المثال التالي ما نقصده:

```

// 'mytype.h' ملف
typedef short    SMALLINT    /* 30000***** النطاق */
typedef int      BIGINT      // 2E9 ***** النطاق

/* البرنامج */
#include "mytype.h"

SMALLINT        i;
BIGINT          loop_count;

```

لا يتسع نطاق العدد الصحيح في BIGINT في بعض الآلات، ونتيجةً لذلك يجب إعادة تعريف النوع ليصبح long.

لإعادة استخدام الاسم المُصرّح مثل تعريف نوع typedef، يجب أن يحتوي تصريحه محدد نوع واحد على الأقل، وهذا من شأنه أن يبعد أي لبس:

```

typedef int new_thing;
func(new_thing x){
    float new_thing;
}

```

```

    new_thing = x;
}

```

نحذّر هنا أن الكلمة المفتاحية `typedef` يمكن استخدامها فقط للتصريح عن نوع القيمة المُعادَة من دالة وليس نوع الدالة الكامل، ونقصد بنوع الدالة الكامل معلومات حول معاملات الدالة ونوع القيمة المُعادَة أيضًا.

```

/*
   صرّح عن func باستخدام typedef لتكون من النوع الذي يأخذ وسيطين من نوع عدد صحيح ويعيد
   قيمة عدد صحيح
*/
typedef int func(int, int);

/* خطأ */
func func_name{ /*...*/ }

// مثال صالح، يعيد مؤشر إلى النوع func
func *func_name(){ /*...*/ }

/*
   مثال صالح إذا كانت الدوال تعيد دوالًا، لكن هذا غير ممكن في لغة سي
*/
func func_name(){ /*...*/ }

```

لا يمكن استخدام معرّف مثل معامل صوري في دالة إذا كان ذلك المعرّف مرتبطًا بمعرّف نوع `typedef` معين ضمن النطاق، إذ سيسبب التصريح من الشكل التالي بمشكلة:

```

typedef int i1_t, i2_t, i3_t, i4_t;

int f(i1_t, i2_t, i3_t, i4_t) //X هنا النقطة

```

يصل المصرف إلى النقطة "X" عند قراءة تصريح الدالة، ولا يعرف فيما إذا كان يقرأ تصريحًا عن دالة، وهذه الحالة مشابهة إلى:

```

int f(int, int, int, int) /* نموذج أولي */

```

أو

```

int f(a, b, c, d) /* ليس نموذجًا أوليًا */

```

يمكن حل المشكلة السابقة (في أسوأ الحالات) بالنظر إلى ما يتبع النقطة "X"؛ فإذا كانت فاصلة منقوطة فهذا تصريح؛ أما إذا كان { فهذا تعريف. تعني القاعدة التي تمنع أسماء تعريف النوع من أن تكون لمعامل صوري أن المصرف يمكنه دائماً أن يخبر فيما إذا كان يعالج تصريحاً أو تعريفاً بالنظر إلى المعرف الأول الذي يتبع اسم الدالة.

استخدام معرف النوع مفيد عندما تريد التصريح عن أشياء ذات صياغة معقدة، مثل "مصفوفة مؤلفة من عشرة مؤشرات تشير إلى مصفوفة تتألف من خمسة أعداد صحيحة"، وهي صيغة معقدة للكتابة حتى لأهمهر المبرمجين. يمكنك كتابتها لمرة واحدة فقط باستخدام معرف النوع أو تجزئتها إلى قطع مقبولة التعقيد:

```
typedef int (*a10ptoa5i[10])[5];
/* أو */
typedef int a5i[5];
typedef a5i *atenptoa5i[10];
```

جربها بنفسك!

8.4 المؤهلان volatile و const

تُعد المؤهلات qualifiers من الأشياء الجديدة التي أتت مع لغة سي C المعيارية، على الرغم من أن فكرة const كانت من لغة ++C أصلاً. دعنا أولاً نوضح لك شيئاً، وهو أن مفهومي const و volatile مستقلان كلياً عن بعضهما بعضاً، إذ يسود الاعتقاد الخاطئ أن const تؤدي عكس غرض volatile، إلا أن المفهومين غير مرتبطين ويجب أن تبقى ذلك ببالك.

بما أن تصاريح const هي الأبسط فسنبدأ بها، إلا أننا سنستعرض الحالات التي تُستخدم فيها كلا النوعين من المؤهلات. إليك لائحة بالكلمات المفتاحية المرتبطة بهذا الشأن:

char	long	float	volatile
short	signed	double	void
int	unsigned	const	

يمثل كل من const و volatile في اللائحة السابقة أنواع مؤهلات، والكلمات المفتاحية المتبقية هي محددات نوع type specifiers، ويُسمح باستخدام عدة محددات أنواع بالشكل التالي:

```
char, signed char, unsigned char
int, signed int, unsigned int
short int, signed short int, unsigned short int
long int, signed long int, unsigned long int
float
```

```
double
long double
```

هناك بعض النقاط التي يجب أن ننوه إليها، وهي أن جميع التصاريح التي تحتوي على `int` ستكون ذات إشارة `signed` افتراضيًا، وبذلك فالكلمة المفتاحية `signed` هي تكرار لا لزوم له في هذا السياق، ويمكن التخلص من `int` إذا وُجد أي محدد نوع أو مؤهل لأن `int` هو النوع الافتراضي.

يمكن تطبيق الكلمتين المفتاحيتين `const` و `volatile` لأي تصريح، متضمّنًا تصريح الهياكل والاتحادات وأنواع المعدّات أو أسماء `typedef`، ونقول عن تصريح يحتوي هذه الكلمتين المفتاحيتين بأنه مؤهل، وهذا السبب في تسمية `const` و `volatile` بالمؤهلات عوضًا عن تسميتهما بمحددات النوع، إليك بعض الأمثلة:

```
volatile i;
volatile int j;
const long q;
const volatile unsigned long int rt_clk;
struct{
    const long int li;
    signed char sc;
}volatile vs;
```

لا تشعر بالضيق من الأمثلة السابقة، فبعضها معقّد وسنشرح معناها لاحقًا، لكن تذكر أنه من الممكن زيادة التعقيد باستخدام محددات صنف التخزين أيضًا، ويوضح المثال التالي استخدامًا واقعيًا ضمن بعض أنوية نظام تشغيل في الوقت الحقيقي:

```
extern const volatile unsigned long int rt_clk;
```

8.4.1 المؤهل `const`

لنلقي نظرة على كيفية استخدام المؤهل `const`، والأمر بسيط جدًا، إذ تعني `const` أن الشيء ليس قابلاً للتعديل، فإذا صرّح عن كائن بيانات باستخدام الكلمة المفتاحية `const` مثل جزء من توصيف نوعه فهذا يعني أنه من غير الممكن إسناد أي قيمة إليه خلال تشغيل البرنامج، ويحتوي التصريح عن الكائن غالبًا على قيمة أولية (وإلا فمتى سيحصل على قيمة إن لم يكن الإسناد إليه مسموحًا؟) إلا أنها ليست الحالة دائمًا؛ فعلى سبيل المثال، إذا أردت الوصول إلى منفذ `port` للعتاد الصلب ضمن عنوان ذاكرة محدد واحتجت للقراءة منه فقط، فسيُصرّح عنه أنه مؤهل `const` لكنه لن يُهيأ.

يُعدُّ أخذُ عنوان كائن بيانات ذي نوع ليس `const` ووضعه في مؤشر يشير إلى إصدار مؤهل باستخدام `const` للنوع نفسه طريقةً آمنةً ومسموحة، إذ يمكنك ذلك من استخدام المؤشر للنظر إلى الكائن دون إمكانية التعديل عليه، بينما يُعدُّ وضع عنوان نوع ثابت إلى مؤشر يشير إلى النوع غير المؤهل طريقةً خطيرةً وبالتالي فهي محظورة، إلا أنه يمكنك تجاوز ذلك باستخدام تحويل الأنواع `cast`. إليك مثالاً عن ذلك:

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int i;
    const int ci = 123;

    /* تصريح عن مؤشر يشير إلى ثابت */
    const int *cpi;

    /* مؤشر اعتيادي يشير إلى كائن غير ثابت */
    int *ncpi;

    cpi = &ci;
    ncpi = &i;

    /* التعليمة التالية صالحة */
    cpi = ncpi;

    /*
    يتطلب الأمر في هذه الحالة تحويل للأنواع لأنه خطأ كبير، انظر لما يلي لمعرفة السماحيات
    */
    ncpi = (int *)cpi;

    /*
    عدّل على ثابت من خلال المؤشر للحصول على سلوك غير محدد
    */
    *ncpi = 0;
    exit(EXIT_SUCCESS);
}
```

[مثال 3]

كما يوضح المثال السابق، فمن الممكن أخذ عنوان كائن ثابت وإنشاء مؤشر يشير إلى كائن غير ثابت ومن ثم استخدام المؤشر، وسيولد ذلك خطأً في برنامجك ويؤدي إلى سلوك غير محدد.

الهدف الرئيسي من استخدام الكائنات الثابتة هو وضعها في حالة قراءة فقط، والسماح للمصرف بإجراء بعض التفقد الإضافي لها ضمن البرنامج، وسيكون المصرف قادرًا على التحقق من أن كائنات `const` لم تُعدّل قسريًا من قبل المستخدم إلا إذا كنت قادرًا على تجاوز ذلك باستخدام المؤشرات.

إليك ميزة إضافية. ما الذي يعنيه التالي؟

```
char c;
char *const cp = &c;
```

الأمر بسيط جدًا، إذ أن `cp` مؤشر يشير إلى `char` وهي الحالة الاعتيادية إن لم تتواجد الكلمة المفتاحية `const`، وتعني الكلمة المفتاحية `const` أن `cp` لا يمكن التعديل عليه، إلا أنه من الممكن تعديل الشيء المُشار إليه بواسطة المؤشر، فالمؤشر هو الثابت وليس الشيء الذي يشير إليه. المثال المُعاكس لما سبق هو:

```
const char *cp;
```

الذي يعني أن `cp` هو مؤشر اعتيادي يمكن التعديل عليه، إلا أن الشيء الذي يشير إليه يجب عدم تعديله، إذًا من الممكن اختيار كون المؤشر أو الشيء الذي يشير إليه قابلًا للتعديل أو لا باستخدام التصريح المناسب بحسب التطبيق الذي تحتاجه.

8.4.2 الموهل volatile

نتقل إلى `volatile` بعد تحدثنا عن `const`. يعود السبب في استخدامنا لهذا النوع من المؤهلات إلى معالجة المشاكل الناتجة عند وقت التشغيل أو الأنظمة المُدمجة المُبرمجة باستخدام لغة سي.

تخيّل كتابة شيفرة برمجية تتحكم بجهاز عتاد صلب بوضع قيم مناسبة في مسجّلات الجهاز في عناوين معروفة، ودعنا نتخيل أيضًا أن للجهاز مسجّلين، كل مسجل بطول 16 بت، بعنوان ذاكرة تصاعدي، والمسجل الأول هو مسجل التحكم والحالة `control and status register` -أو اختصارًا `csr`-، والمسجل الثاني هو منفذ البيانات، ونستطيع الوصول إلى جهاز مماثل بالطريقة التالية:

```
// volatile أو const دون استخدام
// مثال بلغة C المعيارية دون استخدام const أو volatile

/* صرّح عن مسجلات الجهاز، واستخدام العدد الصحيح أو العدد الصحيح القصير معرّف بحسب التطبيق */
*/

struct devregs{
    unsigned short csr; /* مسجل التحكم والحالة */
```

```

        unsigned short data; /* منفذ البيانات */
};

// أنماط البتات في CSR
#define ERROR 0x1
#define READY 0x2
#define RESET 0x4

/* العنوان المطلق للجهاز */
#define DEVADDR ((struct devregs *)0xffff0004)

/* عدد الأجهزة في النظام */
#define NDEVS 4

/*
انتظر الدالة حتى تقرأ بت من الجهاز n، ثم تحقق من نطاق رقم الجهاز
انتظر حتى READY أو ERROR، وقرأ البايت إن لم يحدث أي خطأ وأعد قيمته
0xffff وإلا فأعد ضبط الخطأ وأعد القيمة 0xffff
*/
unsigned int read_dev(unsigned devno){

    struct devregs *dvp = DEVADDR + devno;

    if(devno >= NDEVS)
        return(0xffff);

    while((dvp->csr & (READY | ERROR)) == 0)
        ; /* فراغ. انتظر حتى الانتهاء من الحلقة */

    if(dvp->csr & ERROR){
        dvp->csr = RESET;
        return(0xffff);
    }
    return((dvp->data) & 0xff);
}

```

[مثال 4]

تُعد الطريقة المتبعة في استخدام تصريح الهيكل لوصف تخطيط مسجل الجهاز واسمه ممارسة شائعة، لاحظ أنه لا يوجد أي كائنات معرفة من هذا النوع، إذ يحدّد التصريح ببساطة الهيكل دون استخدام أي مساحة. نستخدم تحويل أنواع ثابت للوصول إلى مسجلات الجهاز وكأنها تشير إلى هيكل، إلا أنها تشير في هذه الحالة إلى عنوان الذاكرة بدلاً من ذلك.

إلا أن هناك مشكلة كبيرة في مصرّفات لغة سي السابقة، وهي متعلقة بحلقة while التكرارية التي تختبر المسجل الأول (مسجل الحالة) وتنتظر البت ERROR أو READY ليظهر، وسيلاحظ أي مصرّف جيد أن الحلقة تفحص عنوان ذاكرة مماثل بصورة متكررة، وسيحاول المصرف أن يشير إلى الذاكرة مرة واحدة وأن ينسخ القيمة إلى المسجل لتسريع العملية؛ وهذا ما لا نريده، إذ أن هذه الحالة من الحالات التي يجب علينا التحقق من المكان الذي يشير إليه المؤشر كل دورة ضمن الحلقة.

نتيجة للمشكلة السابقة، لم تكن معظم مصرّفات لغة سي قادرة سابقاً على تحسين أداء البرنامج، وللتخلص من هذه المشكلة (ومشاكل أخرى مشابهة مرتبطة بمتى يمكن الكتابة على شيء يشير إليه المؤشر) قُدّمت الكلمة المفتاحية volatile، التي تخبر المصرف أن الكائن عرضة للتغير المفاجئ بسبب أسباب لا يمكن التنبؤ بها من خلال النظر إلى البرنامج ذات نفسه، وتُجبر كل مرجع refernece للكائن بأن يصبح مرجعاً فعلياً (وليس عن طريق مؤشر).

إليك البرنامج السابق (مثال 4) مكتوباً باستخدام volatile و const.

```

/*
صرّح عن مسجلات الجهاز، واستخدام العدد الصحيح أو العدد الصحيح القصير معرّف بحسب التطبيق
*/

struct devregs{
    unsigned short volatile csr;
    unsigned short const volatile data;
};

// أنماط البتات في csr
#define ERROR    0x1
#define READY   0x2
#define RESET    0x4

/* العنوان المطلق للجهاز */
#define DEVADDR ((struct devregs *)0xffff0004)

/* عدد الأجهزة في النظام */

```

```

#define NDEVS 4

/*
انتظر الدالة حتى تقرأ بت من الجهاز n، ثم تحقق من نطاق رقم الجهاز
انتظر حتى READY أو ERROR، واقراء البايت إن لم يحدث أي خطأ وأعد قيمته
وإلا فأعد ضبط الخطأ وأعد القيمة 0xffff
*/
unsigned int read_dev(unsigned devno){

    struct devregs * const dvp = DEVADDR + devno;

    if(devno >= NDEVS)
        return(0xffff);

    while((dvp->csr & (READY | ERROR)) == 0)
        ; /* فراغ. انتظر حتى الانتهاء من الحلقة من الحلقة */

    if(dvp->csr & ERROR){
        dvp->csr = RESET;
        return(0xffff);
    }

    return((dvp->data) & 0xff);
}

```

[مثال 5]

تطابق القوانين الخاصة بمزج `volatile` والأنواع الاعتيادية تلك الخاصة بالمؤهل `const`، إذ من الممكن إسناد مؤشر يشير إلى كائن مؤهل بالمؤهل `volatile` إلى عنوان كائن اعتيادي بأمان دون أي مشاكل، إلا أنه من الخطر (ويجب استخدام تحويل الأنواع في هذه الحالة) أخذ عنوان الكائن المؤهل بالمؤهل `volatile` ووضعه في مؤشر يشير إلى كائن اعتيادي، واستخدام مؤشر من هذا النوع سيتسبب بسلوك غير معرّف.

إذا صرّح عن مصفوفة أو هيكل باستخدام إحدى المؤهلين `const` أو `volatile`، فمن الممكن لجميع الأعضاء أن تمتلك هذا المؤهل أيضًا، وهذا الأمر المنطقي إذا فكرت به للحظة، فكيف لعضو من هيكل مؤهل بالمؤهل `const` أن يكون قابلاً للتعديل؟

هذا يعني أن أي محاولة لإعادة كتابة المثال السابق ممكنة، فعوضًا عن تصريح مسجلات الجهاز بكونها `volatile` في الهيكل، من الممكن للمؤشر أي يُصرَّح بأن يشير إلى هيكل `volatile` عوضًا عن ذلك، على النحو التالي:

```
struct devregs{
    unsigned short  csr;    /* مسجل التحكم والحالة */
    unsigned short  data;  /* منفذ البيانات */
};
volatile struct devregs *const dvp=DEVADDR+devno;
```

بما أن `dvp` يشير إلى كائن `volatile`، فمن غير المسموح تحسين المراجع عن طريق المؤشرات، وعلى الرغم من أن ما سبق يعمل، إلا أنه ممارسة سيئة، إذ ينتمي تصريح `volatile` إلى الهيكل، ومسجلات الجهاز هي `volatile` وهذا المكان الذي يجب أن يبقى فيه التصريح، لتسهيل قراءة الشيفرة البرمجية.

إذًا، مؤهل النوع `volatile` مهم جدًا لأي كائن سيتعرض لتغييرات، سواءً كانت التغييرات بواسطة العتاد الصلب أو برامج خدمات المقاطعة غير المتزامنة `asynchronous interrupt service routines`.

وما إن اعتقدت أنك فهمت كل ما سبق بصورة مثالية، حتى يأتي التصريح التالي الذي سيغير من رأيك:

```
volatile struct devregs{
    /* محتوى */
}v_decl;
```

الذي يصرح عن النوع `struct devregs` إضافةً إلى كائن مؤهل باستخدام `volatile` من ذلك النوع باسم `v_decl`. ويتبعه التصريح التالي:

```
struct devregs nv_decl;
```

الذي يصرح عن `nv_decl` وهو **ليس** مؤهلًا باستخدام `volatile`؛ فالتأهيل ليس جزءًا من النوع `struct devregs` وإنما ينطبق فقط على تصريح `v_decl`. لننظر للأمر من زاوية أخرى لعل الأمر يتضح لك بعض الشيء (التصريحان متماثلان في نتيجهما):

```
struct devregs{
    /* محتوى */
}volatile v_decl;
```

إذا أردت الطريقة المختصرة لإرفاق مؤهل إلى نوع آخر، فيمكنك استخدام `typedef` لتحقيق الآتي:

```
struct x{
```

```

int a;
};
typedef const struct x csx;

csx const_sx;
struct x non_const_sx = {1};

const_sx = non_const_sx;          /* التعديل على ثابت يتسبب خطأ */

```

1. العمليات غير القابلة للتجزئة

سيفهم الذي يتعاملون مع تقنيات مقاطعات العتاد الصلب والجوانب الأخرى للوقت الحقيقي في البرمجة أهمية أنواع `volatile`، وهناك ضرورة في هذا المجال للتأكد من أن الوصول إلى كائنات البيانات متواصل، إلا أن مناقشة هذا الأمر سيأخذنا في رحلة بعيدة عن موضوعنا هنا، لكن دعنا نتكلم عن بعض المشكلات بخصوص هذا الأمر على الأقل.

لا نخطئ الاعتقاد وتفترض أن جميع العمليات المكتوبة في لغة سي متواصلة، فعلى سبيل المثال قد يكون التصريح التالي عدداً يُحدّث عن طريق مقاطعة برنامج ساعة:

```
extern const volatile unsigned long realtimeclock;
```

من المهم هنا أن يتضمن التصريح على المؤهل `volatile` بسبب التغييرات اللامتزامنة التي تحصل له، ويحتوي على المؤهل `const` لأنه من غير الممكن التعديل على قيمته سوى عن طريق برنامج المقاطعة، وإن سُمح للبرنامج الوصول إليه بالطريقة التالية سنحصل على مشكلة:

```

unsigned long int time_of_day;

time_of_day = real_time_clock;

```

ماذا لو استغرقت عملية نسخ `long` إلى `long` آخر عدّة تعليمات آلة لنسخ الكلمتين `real_time_clock` و `time_of_day`؟ من الممكن حدوث مقاطعة خلال عملية الإسناد وستكون أسوأ حالة لهذه المقاطعة هي عندما تكون الكلمة الأقل ترتيباً لـ `real_time_clock` تساوي `0xffff` والكلمة مرتفعة الترتيب تساوي `0x0000`، وبهذا ستكون قيمة الكلمة منخفضة الترتيب مساوية إلى `0xffff`. تعمل المقاطعة عملها وتزيد الكلمة منخفضة الترتيب لـ `real_time_clock` إلى `0x0` والكلمة مرتفعة الترتيب إلى `0x1` ومن ثم تعيد القيمة، ويُستكمل ما تبقى من الإسناد فيما بعد وينتهي الأمر باحتواء `time_of_day` على القيمة `0x0001ffff` و `real_time_clock` على القيمة الصائبة `0x00010000`.

تعدّ المشكلات المسابقة لما سبق منطقةً خطيرة، ويعلم جميع من يعمل في البيئات غير المتزامنة هذا الأمر جيّدًا، ولا تأخذ لغة سي المعيارية أي إجراءات احترازية لتفادي هذا النوع من المشكلات، ويجب تطبيق الطريقة الاعتيادية.

يُصرّح ملف الترويسة `signal.h` عن نوع باسم `sig_atomic_t` ومن المضمون إمكانية تعديل هذا النوع بأمان عند التعامل مع الأحداث غير المتزامنة، وهذا يعني أنه من الممكن تعديله عن طريق إسناد قيمة إليه أو زيادة قيمته أو إنقاصها أو أي شيء آخر يعطي قيمة جديدة بحسب القيمة السابقة، وهو ليس آمنًا.

8.5 نقاط التسلسل Sequence points

ترتبط نقاط التسلسل `sequence points` بمشكلات برمجة الوقت الحقيقي، إلا أنها مختلفة عن المشكلات التي ناقشناها، وتعدّ بمثابة محاولة للمعيار في تعريف الحالات التي تسمح -أو لا تسمح- بها طرق معينة من التحسين، على سبيل المثال، ألق نظرةً على البرنامج التالي:

```
#include <stdio.h>
#include <stdlib.h>

int i_var;
void func(void);

main(){
    while(i_var != 10000){
        func();
        i_var++;
    }
    exit(EXIT_SUCCESS);
}

void
func(void){
    printf("in func, i_var is %d\n", i_var);
}
```

[مثال 6]

قد يحاول المصرّف تحسين الأداء في الحلقة التكرارية بحيث يخزّن `i_var` في مسجّل الآلة لزيادة السرعة، إلا أن الدالة تحتاج وصولاً إلى القيمة الصحيحة من `i_var` حتى يمكنها طباعة القيمة الصحيحة، وهذا يعني أن

المسجل يجب أن يعيد تخزين قيمة `i_var` عند كل استدعاءٍ للدالة على الأقل، ويصف المعيار الشروط التي تحدد متى وأين يحصل ذلك. تُستكمل التأثيرات الجانبية لكل تعبير في نقطة التسلسل التي سبقتها، وهذا السبب في عدم قدرتنا على الاعتماد على تعابير مشابهة لهذه:

```
a[i] = i++;
```

وذلك بسبب عدم وجود أي نقطة تسلسلية مُحدّدة ضمن الإسناد أو عوامل الزيادة والنقصان، ولا يمكننا معرفة متى سيؤثر عامل الزيادة على `i` تحديداً.

يعرّف المعيار نقاط التسلسل على النحو التالي:

- نقطة استدعاء دالة، بعد تقييم وسطائها.
- نهاية المعامل الأول للعامل `&&`.
- نهاية المعامل الأول للعامل `||`.
- نهاية المعامل الأول للعامل الشرطي `?:`.
- نهاية كل من معاملات عامل الفاصلة `,`.
- نهاية تقييم تعبير كامل، على النحو التالي:
 - تقييم القيمة الأولية لكائن `auto`.
 - تعبير اعتيادي، أي متبوع بفاصلة منقوطة.
 - تعابير التحكم في تعليمات `do` أو `while` أو `if` أو `switch` أو `for`.
 - التعبيران الآخزان في تعليمة حلقة `for`.
 - التعبير في تعليمة `return`.

8.6 الخاتمة

قدّم هذا الفصل شرحاً عن النقاط التخصصية من اللغة. تُعدّ المشكلات المتعلقة بالنطاق والربط والمدة الزمنية مهمةً دون أي شك، وإن وجدت هذا الموضوع صعب الفهم بعض الشيء، حاول تعلم القوانين الأساسية فقط. يضع المعيار كثيراً من القواعد سعياً منه للوضوح والبعد عن الغموض، ومن الأسهل الالتزام فقط بالطريقة البسيطة لإنجاز المهام دون محاولة الطرق المعقدة، واستخدم المثال 2 (الفصل الأول) مرجعاً إذا أردت.

يعتمد استخدام `typedef` على مستوى خبرتك، فمن الشائع استخدامها لتجنب بعض الجوانب غير السارة بخصوص التصاريح عن الأنواع المعقدة.

ستشهد استخدام `const` في برامج كثيرة، وتستخدم النماذج الأولية لدوال بعض المكتبات مبدأ استخدام مؤشر يشير إلى شيء لا يمكن التعديل عليه بكثرة.

تُستخدم `volatile` من قبل التطبيقات الاختصاصية فقط، وسيهمك هذا الاستخدام إذا كنت تعمل في مجال برمجة الوقت الحقيقي أو الأنظمة المُدمجة، وإلا فأهميتها لك محدودة، وينطبق الأمر أيضاً على نقاط التسلسل، إذ ستكون درجة دعم المصنفات القادمة لهذه الميزتين سؤالاً مثيراً للاهتمام.

دورة تطوير تطبيقات الويب باستخدام لغة Ruby



دورة تدريبية متكاملة من الصفر وحتى الاحتراف
تمكنك من التخصص في هندسة الويب ودخول سوق العمل

[التحق بالدورة الآن](#)



9. المكتبات Libraries

9.1 مقدمة

سيساهم قرار لجنة لغة سي المعيارية بتعريف إجراءات Routines عدد من المكتبات بما يعود بالنفع الكبير لجميع مستخدمي لغة سي دون أي شك، إذ لم يكن هناك أي معيار مُتَّفَق عليه يعرف إجراءات المكتبات ويقدم دعمًا للغة، وانعكس ذلك سلبيًا على قابلية نقل البرامج Portability كثيرًا.

ليس من المطلوب أن تتواجد إجراءات المكتبة داخل البرنامج، إذ تتواجد فقط في البيئات المُستضافة Hosted environment وتنطبق هذه الحالة غالبًا على مبرمجي التطبيقات، بينما لن تكون المكتبات موجودةً في حالة مبرمجي النظم المُدمجة ومبرمجي البيئات المُستضافة؛ إذ يستخدم هذا النوع من المبرمجين لغة سي لوحدها ضمن بيئة مستقلة Freestanding environment، وبالتالي لن يكون هذا الفصل مهمًا لهم.

لن تكون المواضيع التي ستبغ هذه المقدمة مكتوبةً بهدف قراءتها بالتسلسل، ويمكنك قراءتها أجزاء منفصلة، إذ نهدف هنا إلى توفير محتوى يُستخدم كمرجع بسيط للمعلومات وليس درس تعليمي شامل، وإلا فسيتطلب الأمر كتابًا مخصصًا لنستطيع تغطية جميع المكتبات.

9.1.1 ملفات الترويسات والأنواع القياسية

تُستخدم عدّة أنواع types وماكرو macro على نحوٍ واسعٍ في دوال المكتبات، وتُعرّف في ملف `#include` الموافق للدالة. كما سيصرّح ملف الترويسة Header عن الأنواع والنماذج الأولية المناسبة لدوال المكتبة، وعلينا أن نذكر عدّة نقاط مهمة بهذا الخصوص:

- تُحجز جميع المعرّفات الخارجية External identifiers وأسماء الماكرو المُصرّح عنها في أي ملف ترويسة لمكتبة، بحيث لا يُمكن استخدامها أو إعادة تعريفها لأي استعمال آخر. قد تحمل الأسماء في

بعض الأحيان أترًا "سحريًا" عندما تكون معروفةً للمصرف ويتسبب ذلك باستخدام بعض الأساليب الخاصة لتطبيقها.

- جميع المعرفات التي تبدأ بحرف الشرطة السفلية underscore _ محجوزة.
- يمكن تضمين ملفات الترويسة بأي ترتيب كان ولأكثر من مرة، إلا أن تضمينها يجب أن يحدث خارج أي تصريح داخلي أو تعريف وقبل أي استخدام للدوال والماكرو المعرّفة بداخلها.
- نحصل على سلوك غير معرّف إن مرّرنا قيمة غير صالحة لدالة، مثل مؤشر فارغ، أو قيمة خارج نطاق القيم التي تقبلها الدالة.

لا يُحدّد المعيار النوع ذاته من القيود الموضحة أعلاه بخصوص المعرّفات، وقد يتبادر إلى ذهنك المغامرة والاستفادة من هذه الثغرات، إلا أننا ننصحك بالالتزام بالطرق الآمنة.

ملفات الترويسة المعيارية هي:

```
<assert.h>   <locale.h>   <stddef.h>
<ctype.h>    <math.h>     <stdio.h>
<errno.h>    <setjmp.h>   <stdlib.h>
<float.h>    <signal.h>  <string.h>
<limits.h>   <stdarg.h>  <time.h>
```

معلومة عامّة أخيرة: تُنفذ العديد من إجراءات المكتبات على أنها ماكرو في عملها، شرط ألا يتسبب ذلك في أي مشاكل ناتجة عن الآثار الجانبية لهذا الاستخدام (كما وضح الفصل السابع). يضمن المعيار وجود دالة اعتيادية إذا كان هناك دالة تُستخدم عادةً مثل ماكرو موافقة لها، بحيث تُنجز الدالتان العمل ذاته، وحتى تستخدم الدالة الاعتيادية عليك أن تُلغي تعريف الماكرو باستخدام التوجيه #undef، أو أن تكتب اسم الماكرو داخل قوسين، مما يضمن أنه لن يُعامل معاملة الماكرو:

```
some function("Might be a macro\n"); // قد يمثل هذا ماكرو
(some function)("Can't be a macro\n"); // من غير الممكن أن يكون هذا ماكرو
```

9.1.2 مجموعات المحارف والاختلافات اللغوية

قدّمت لجنة المعيار بعض المزايا الموجهة لاستخدام سي في البيئات التي لا تستخدم مجموعة محارف معيار US ASCII، والاختلافات اللغوية الأخرى التي تستخدم الفاصلة أو النقطة للدلالة على الفاصلة العشرية. قدّمت التسهيلات (ألق نظرةً على القسم) بفكرة برنامج يتحكم بسلوك دوال المكتبات ليوافق الاختلافات اللغوية.

تُعد مهمة تقديم دعم متكامل لمختلف اللغات والتقاليد مهمّة صعبة، وغالبًا ما يُساء فهمها، والتسهيلات المُزوّدة بمكتبات لغة سي هي الخطوة الأولى في هذا المشوار الطويل للوصول إلى الحل الكامل.

الحل الوحيد المُعرّف من المعيار هو ما يدعى بلغة C المحلية locale، ويقدم هذا دعمًا فعليًا على نحوٍ مشابه لعمل لغة سي القديمة، بينما تقدم الإعدادات المحلية الأخرى سلوكًا مختلفًا بحسب تعريف التطبيق.

9.1.3 ملف ترويسة <stddef.h>

هناك عددٌ صغير من الأنواع والماكرو الموجودة في <stddef.h> والمُستخدمة كثيرًا في ملفات الترويسة الأخرى، الذين سنتكلم عنها لاحقًا.

تعطينا عملية طرح مؤشر من آخر نتيجةً من نوع مختلف بحسب التطبيق، وللسماع باستخدام الآمن في حال الاختلاف، يعرّف ملف الترويسة <stddef.h> النوع ptrdiff_t، كما يمكنك استخدام النوع size_t لتخزين نتيجة العامل sizeof بصورةٍ مشابهة.

لأسباب لا تزال مخفية عنّا للوقت الحالي، هناك "مؤشر ثابت فارغ معرّف بحسب التنفيذ" معرّف في <stddef.h> باسم NULL، قد يبدو ذلك غير ضروريًا بالنظر إلى أن لغة سي تعرّف ثابت الرقم الصحيح 0 إلى القيمة التي يمكن إسنادها إلى مؤشر فارغ ومقارنتها معه، إلا أن الممارسة التالية **شائعة جدًا** وسط مبرمجي لغة سي المتمرسين:

```
#include <stdio.h>
#include <stddef.h>
FILE *fp;

if((fp = fopen("somefile", "r")) != NULL){
    /* وهلمّ جرًا */
}
```

هناك ماكرو باسم offsetof مهمته إيجاد مقدار الإزاحة offset بالبايت لعضو هيكل ما؛ إذ أن مقدار الإزاحة هو المسافة بين العضو وبداية الهيكل، إليك مثالًا عن ذلك:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

main(){
    size_t distance;
    struct x{
        int a, b, c;
```

```

    }s_tr;

    distance = offsetof(s_tr, c);
    printf("Offset of x.c is %lu bytes\n",
          (unsigned long)distance);

    exit(EXIT_SUCCESS);
}

```

[مثال 1]

يجب أن يكون التعبير `s_tr.c` قادرًا على التقييم مثل عنوانٍ لثابت (انظر الفصل السادس)، فإذا كان العضو الذي تبحث عن مقدار إزاحته هو حقل بتات `bitfield` فستحصل على سلوك غير معرّف في هذه الحالة. لاحظ طريقة تحويل الأنواع في `size_t` التي نحوّل فيها لأطول نوع ممكن عديم الإشارة للتأكد من أن وسيط `printf` هو من النوع المناسب (`%ul` هو رمز التنسيق الخاص بطباعة النوع `unsigned long`) مع المحافظة على دقة القيمة، وهذا بسبب أن نوع `size_t` مجهول للمبرمج. العنصر الأخير المُصرّح عنه في `<stddef.h>` هو `wchar_t` وهو قيمة عدد صحيح كبيرة يُمكن تخزين محرف عريض `wide character` فيها ينتمي إلى أي مجموعة محارف موسّعة `extended character set`.

9.1.4 ملف الترويسة `<error.h>`

يعرّف ملف الترويسة ما يُدعى `errno` الذي يُستبدل بتعبير ثابت ذو قيمة صحيحة لا تساوي الصفر، ويُضمن أن يكون هذا التعبير مقبولاً في موجّهات `#if`، ويعرّف أيضاً الماكرو `EDOM` والماكرو `ERANGE` اللذان يُستخدمان في الدوال الرياضية للدلالة على نوع الخطأ الحاصل وسنشرحهما بتوسع أكبر لاحقاً.

يُستخدم `errno` للدلالة على خطأ مُكتشف من دوال المكتبات، وهو ليس متغير خارجي بالضرورة -كما كان سابقاً- بل هو قيمة متغيرة من نوع `int`، إذ تُسند القيمة صفر إليه عند بداية تشغيل البرنامج، ولا يُعاد ضبط قيمته من تلك النقطة فصاعداً إلا إذا جرى ذلك مباشرةً؛ أي بكلمات أخرى، لا تحاول إجراءات المكتبات إعادة ضبطه إطلاقاً، وإذا حدث أي خطأ في إجراء المكتبة فإن قيمة `errno` تتغير إلى قيمة معينة تشير إلى نوع الخطأ الحاصل ويُعيد الإجراء هذه القيمة (غالبًا -1) للدلالة على الخطأ، إليك تطبيقاً عملياً عن ذلك:

```

#include <stdio.h>
#include <stddef.h>
#include <errno.h>

errno = 0;

```

```
if(some_library_function(arguments) < 0){
    // خطأ في معالجة الشيفرة المصدرية
    // قد يستخدم قيمة errno مباشرة
```

تطبيق `errno` غير معروف بالنسبة للمبرمج، فلا تحاول فعل أي شيء على هذه القيمة عدا إعادة ضبطها أو فحصها، فعلى سبيل المثال، من غير المضمون أن يكون لهذه القيمة عنوانًا على الذاكرة. يجب أن تتفقد قيمة `errno` فقط في حال كانت دالة المكتبة المُستخدمة توثق تأثيرها على `errno`، إذ يمكن لدوال المكتبات الأخرى أن تضبطها إلى قيمة عشوائية بعد استدعائها إلا إذا كان وصف الدالة يحدد ما الذي تفعله الدالة بالقيمة بصورة صريحة.

9.2 تشخيص الأخطاء

من المفيد عندما تبحث عن الأخطاء في برنامجك أن تكون قادرًا على فحص قيمة تعبير ما والتأكد من أن قيمته هي ما تتوقعها فعليًا، وهذا ما تقدمه لك دالة `assert`.

يجب عليك أن تُضمّن ملف الترويسة `<assert.h>` أولاً حتى تتمكن من استخدام الدالة `assert`، وهذه الدالة معرفة على النحو التالي:

```
#include <assert.h>

void assert(int expression)
```

إذا كانت قيمة التعبير صفر (أي "خطأ false")، فستطبع الدالة `assert` رسالةً تدل على التعبير الفاشل، وتتضمن الرسالة اسم ملف الشيفرة المصدرية والسطر الذي يحتوي على التوكيد `assertion` والتعبير، ومن ثم تُستدعى دالة `abort` التي تقطع عمل البرنامج.

```
assert(1 == 2);

/* قد يتسبب ما سبق بالتالي */

Assertion failed: 1 == 2, file silly.c, line 15
```

في حقيقة الأمر الكلمة `Assert` معرفة مثل ماكرو، وليس مثل دالة حقيقية. لتعطيل التوكيدات في برنامج يستوفي شروط عمله دون مشاكل، نعرّف الاسم `NDEBUG` **قبل** تضمين `<assert.h>`، وسيُعطل هذا جميع التوكيدات الموجودة في كامل البرنامج. عليك أن تعرف الآثار الجانبية التي يتسبب بها هذا للتعبير، فلن يُقيم التعبير تعطيل التوكيدات باستخدام `NDEBUG`، وبذلك سيسلك المثال التالي سلوكًا غير مُتوقع عند إلغاء التوكيدات باستخدام `#define NDEBUG`.

```
#define NDEBUG
#include <assert.h>

void
func(void)
{
    int c;
    assert((c = getchar()) != EOF);
    putchar(c);
}
```

[مثال 2]

لاحظ أن الدالة `assert` لا تُعيد أي قيمة.

9.3 التعامل مع المحارف

هناك مجموعة متنوعة من الدوال تهدف لفحص وربط mapping المحارف، إذ تسمح لك دوال الفحص `test functions` التي سنناقشها أولاً- بفحص فيما إذا كان المحرف من نوع معين، مثل حرف أبجدي، أو حرف صغير أم كبير، أو محرف رقمي، أو محرف تحكّم `control character`، أو إشارة ترقيم، أو محرف قابل للطباعة أو لا، وهكذا. تُعيد دوال فحص المحرف قيمة عدد صحيح `integer` تساوي الصفر إذا لم يكن المحرف المُحدّد منتمياً إلى التصنيف المذكور، أو قيمة غير صفرية عدا ذلك، ويأخذ هذا النوع من الدوال وسيطاً ذا قيمة عدد صحيح تُمثّل قيمته من نوع `"unsigned char"`، أو عدد صحيح ثابت قيمته `"EOF"` مثل تلك القيمة المُعادة من دوال مشابهة، مثل `getchar()`، ونحصل على سلوك غير معرّف خارج هذه الحالات.

تعتمد هذه الدوال على إعدادات البرنامج المحلية:

محرّف الطباعة `printing character` هو عضو من مجموعة المحارف المعرّفة بحسب التطبيق، ويشغل كل محرف طباعة موقع طباعة واحد، و**محرّف التحكم `control character`** هو عضو من مجموعة المحارف المعرّفة بحسب التطبيق أيضاً إلا أن كل محرف منها ليس بمحرّف طباعة. إذا استخدمنا مجموعة محارف معيار `ASCII 7-bit`، ستكون محارف الطباعة بين الفراغ `(0x20)` وتيلدا `(0x7e)`، بينما تكون محارف التحكم بين `(0x0)` `NUL` و `(0x1f)` `US` والمحرّف `(0x7f)` `DEL`.

تجد أدناه ملخصاً يحتوي على جميع دوال فحص المحرف، ويجب تضمين ملف الترويسة `<ctype.h>` قبل استخدام أيّ منها.

- دالة `isalnum(int c)`: تُعيد القيمة "True" إذا كان `c` حرفًا أبجديًا أو رقمًا؛ أي `(isalpha(c) || isdigit(c))`.
 - دالة `isalpha(int c)`: تُعيد القيمة "True" إذا كان هذا الشرط `(isupper(c) || islower(c))` محققًا، كما أنها تُعيد القيمة True لمجموعة المحارف المُعرفة بحسب التطبيق التي لا تعيد القيمة True عند تمريرها على الدالة `isdigit` أو `isctrl` أو `ispunct` أو `isspace` وتكون مجموعة المحارف الإضافية هذه فارغة في لغة سي المحلية.
 - دالة `isctrl(int c)`: تُعيد القيمة True إذا كان المحرف محرف تحكم.
 - دالة `isdigit(int c)`: تُعيد القيمة True إذا كان المحرف رقمًا عشريًا `decimal`.
 - دالة `isgraph(int c)`: تُعيد القيمة True إذا كان المحرف هو محرف طباعة عدا محرف المسافة الفارغة.
 - دالة `islower(int c)`: تُعيد القيمة True إذا كان المحرف حرفًا أبجديًا صغيرًا `lower case`، كما أنها محققة لمجموعة محارف معرفة حسب التطبيق لا تُعيد القيمة True لأي من الدالة `isctrl` أو `isdigit` أو `ispunct` أو `isspace`، وتكون مجموعة المحارف الإضافية هذه فارغة في C المحلية.
 - دالة `isprint(int c)`: تُعيد القيمة True إذا كان المحرف محرف طباعة (متضمّنًا محرف المسافة الفارغة).
 - دالة `ispunct(int c)`: تُعيد القيمة True إذا كان المحرف محرف طباعة عدا محرف المسافة الفارغة أو المحارف التي تُعيد القيمة True في دالة `isalnum`.
 - دالة `isspace(int c)`: تُعيد القيمة True إذا كان المحرف محرف مسافة بيضاء (المحرف ' ' أو '\n أو \f أو \r أو \t أو \v')
 - دالة `isupper(int c)`: تُعيد القيمة True إذا كان المحرف حرفًا أبجديًا كبيرًا `upper case`، كما أنها محققة لمجموعة محارف معرفة حسب التطبيق لا تُعيد القيمة True لأي من الدالة `isctrl` أو `isdigit` أو `ispunct` أو `isspace`، وتكون مجموعة المحارف الإضافية هذه فارغة في لغة سي المحلية.
 - دالة `isxdigit(int c)`: تُعيد القيمة True إذا كان المحرف رقم ستّ عشري صالح.
- هناك دالتان إضافيتان تربطان المحارف من مجموعةٍ إلى أخرى، إذ تُعيد الدالة `tolower` حرفًا صغيرًا موافقًا لمحرف كبير مُرّر لها، على سبيل المثال:

```
tolower('A') == 'a'
```

تُعيد الدالة `tolower` المحرف ذاته، إذا تلقت أي محرف مُغاير للمحارف الأبجدية الكبيرة. تربط الدالة `toupper` المعاكسة للدالة السابقة في عملها المحرف المُمرّر لها إلى مكافئه الكبير. تُجرى عملية الربط في الدالتين السابقتين فقط في حال وجود محرف موافق للمحرف المُمرّر لها، إذ لا تمتلك بعض اللغات محرّفًا كبيرًا موافقًا لمحرف صغير والعكس صحيح.

9.4 التوطين Localization

نستطيع التحكم بالإعدادات المحليّة للبرنامج من هنا، ويصرح ملف الترويسة `<locale.h>` دوال `setlocale` و `localeconv` وعددًا من الماكرو:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

تُستبدل جميع الماكرو بتعبير ثابت ذي قيمة عدد صحيح وتُستخدم القيمة الناتجة عن التعبير مكان الوسيط `category` في الدالة `setlocale` (يمكن تعريف أسماء أخرى أيضًا، ويجب أن يبدأ كل منها بـ `LC_X`، إذ يمثّل X المحرف الأبجدي الكبير)، ويُستخدم النوع `struct lconv` لتخزين المعلومات المتعلقة بتنسيق القيم الرقمية، ويُستخدَم `CHAR_MAX` للأعضاء من النوع `char` للدلالة على أن القيمة غير متوافرة في الإعدادات المحلية الحالية.

يحتوي `lconv` على عضو واحد على الأقل من الأعضاء التالية:

العضو	الاستخدام	تمثيله في إصدارات سي المحلية	ملاحظات إضافية
<code>char *decimal_point</code>	يُستخدم المحرف للفاصلة العشرية في القيم المنسقة غير المالية.	"."	---
<code>char *thousands_sep</code>	يُستخدم المحرف لفصل مجموعات من الخانات الواقعة على يسار الفاصلة العشرية في القيم المنسقة غير المالية.	""	---
<code>char *grouping</code>	يعرّف عدد الخانات في كل	""	يحدد "3" أن الخانات

العضو	الاستخدام	تمثيله في إصدارات سي المحلية	ملاحظات إضافية
	مجموعة في القيم المنسقة غير المالية، وتحدد القيمة CHAR_MAX أنه لا يوجد أي تجميع إضافي مطلوب، بينما تحدد القيمة 0 أنه يجب تكرار العنصر السابق للخانات الرقمية المتبقية، وإذا استُخدمت أي قيمة أخرى فهي تمثل قيمة العدد الصحيح المُمثل لعدد الخانات التي تتألف منها المجموعة الحالية (المحرف اللاحق في السلسلة النصية يُفسَّر قبل التجميع.		يجب أن تجمع كل ثلاثة في مجموعة ويشير محرف الإنهاء الفارغ terminating null في السلسلة النصية إلى تكرار ١3.
char *int_curr_symbol	تُستخدم المحارف الأولى الثلاث لتخزين رمز العملة العالمي الأبجدي لإصدار سي المحلي، بينما يُستخدم المحرف الرابع للفصل بين رمز العملة العالمي والكمية النقدية.	""	---
char *currency_symbol	يمثل رمز العملة للإصدار المحلي الحالي.	""	---
char *mon_decimal_point	المحرف المُستخدم مثل فاصلة عشرية عند تنسيق القيم النقدية.	""	---
char *mon_thousands_sep	يمثل فاصل مجموعات خانات الأرقام ذات القيم المنسقة بتنسيق نقدي.	""	---
char *mon_grouping	يعرف عدد الخانات في كل مجموعة عند تنسيق قيم نقدية، وتُفسَّر عناصره على أنها جزء من التجميع	""	---
char *positive_sign	السلسلة النصية	""	---

العضو	الاستخدام	تمثيله في إصدارات سي المحلية	ملاحظات إضافية
	المُستخدمة للدلالة على قيمة نقدية غير سالبة.		
char *negative_sign	السلسلة النصية المُستخدمة للدلالة على قيمة نقدية سالبة.	""	---
char int_frac_digits	عدد الخانات التي تُعرض بعد الفاصلة العشرية في قيمة نقدية منسقة عالميًا.	CHAR_MAX	---
char frac_digits	عدد الخانات التي تُعرض بعد الفاصلة العشرية في قيمة نقدية غير منسقة عالميًا.	CHAR_MAX	---
char p_cs_precedes	قيمة 1 تدل على وجوب إتباع currency_symbol بالقيمة عند تنسيق قيمة غير سالبة نقدية، بينما تدل القيمة 0 على إسباق currency_symbol بالقيمة.	CHAR_MAX	---
char p_sep_by_space	قيمة 1 تدل على تفريق رمز العملة من القيمة بمسافة فارغة عند تنسيق قيمة غير سالبة نقدية، بينما تدل قيمة 0 على عدم وجود أي مسافة فارغة.	CHAR_MAX	---
char n_cs_precedes	تشابه p_cs_precedes ولكن للقيم النقدية السالبة.	CHAR_MAX	---
char n_sep_by_space	تشابه p_sep_by_space ولكن للقيم النقدية السالبة.	CHAR_MAX	---
char n_sign_posn	يشابه p_sign_posn	CHAR_MAX	---

العضو	الاستخدام	تمثيله في إصدارات سي المحلية	ملاحظات إضافية
	ولكن للقيم النقدية السالبة.		
			يتبع الشروط التالية: تُحيط الأقواس القيمة النقدية و .currency_symbol تسبق السلسلة النصية كل من القيمة النقدية و .currency_symbol تتبع السلسلة النصية القيمة النقدية و .currency_symbol تسبق السلسلة النصية القيمة .currency_symbol تتبع السلسلة النصية القيمة currency_symbol
char p_sign_posn	يمثل موقع positive_sign للقيم النقدية المنسقة غير السالبة.	CHAR_MAX	

9.4.1 دالة setlocale لضبط الإعدادات المحلية

يكون تعريف دالة setlocale على النحو التالي:

```
#include <locale.h>
```

```
char *setlocale(int category, const char *locale);
```

تسمح هذه الدالة بضبط إعدادات البرنامج المحلية، ويمكن ضبط جميع أجزاء الإصدار المحلي باختيار القيم المناسبة لوسيط التصنيف category كما يلي:

- القيمة LC_ALL: تضبط كامل الإصدار المحلي.
- القيمة LC_COLLATE: تعديل سلوك strcoll و strxfrm.
- القيمة LC_CTYPE: تعديل سلوك دوال التعامل مع المحارف character-handling.
- القيمة LC_MONETARY: تعديل تنسيق القيم النقدية المُعادة من دالة localeconv.
- القيمة LC_NUMERIC: تعديل محرف الفاصلة لتنسيق الدخل والخرج وبرامج تحويل السلاسل النصية.

• القيمة LC_TIME: تعديل سلوك strftime.

يمكن ضبط قيم الإعدادات المحلية إلى:

تحديد البيئة ذات المتطلبات الدنيا لترجمة سي C	"C"
تحديد البيئة الأصيلة المعرفة حسب التطبيق	""
تحديد البيئة الموافقة لهذه القيمة	قيمة معرفة بحسب التنفيذ

البيئة الافتراضية عند بداية البرنامج موافقة للبيئة التي نحصل عليها عند تنفيذ التعليمة التالية:

```
setlocale(LC_ALL, "C");
```

يمكن فحص السلسلة النصية الحالية المترافقة مع تصنيف ما بتمرير مؤشر فارغ null قيمةً للوسيط locale؛ نحصل على السلسلة النصية المترافقة مع التصنيف category المحدد للتوطين الجديد إذا كان من الممكن حصول التصنيف المحدد، وتُستخدم هذه السلسلة النصية في استدعاء لاحق للدالة setlocale مع تصنيفها المترافق لاستعادة الجزء الموافق من إعدادات البرنامج المحلية، وإذا كان التحديد غير ممكن الحصول نحصل على مؤشر فراغ دون تغيير الإعدادات المحلية.

9.4.2 دالة localeconv

يكون تصريح الدالة على النحو التالي:

```
#include <locale.h>

struct lconv *localeconv(void);
```

تُعيد هذه الدالة مؤشرًا يشير إلى هيكل من النوع struct lconv، ويُضبط هذا المؤشر طبقًا للإعدادات المحلية الحالية ويمكن تغييره باستدعاء لاحق للدالة localconv أو setlocale، ويجب ألا يكون الهيكل قابلاً للتعديل بأي طريقة أخرى.

على سبيل المثال، إذا كانت إعدادات القيم النقدية المحلية الحالية مُمثَّلةً حسب الإعدادات التالية:

تنسيق القيم الموجبة	IR£1,234.56
تنسيق القيم السالبة	(IR£1,234.56)
التنسيق العالمي	IRP 1,234.56

يجب أن تحمل الأعضاء التي تمثّل القيم النقدية في lconv القيم التالية:

" IRP"	int_curr_symbol
"£IR"	currency_symbol

mon_decimal_point	","
mon_thousands_sep	","
mon_grouping	"3\"
postive_sign	""
negative_sign	""
int_frac_digits	2
frac_digits	2
p_cs_precedes	1
p_sep_by_space	0
n_cs_precedes	`1
n_sep_by_space	0
p_sign_posn	CHAR_MAX
n_sign_posn	0

9.5 القيم الحدية

يُعرّف ملفا الترويسة `<float.h>` و `<limits.h>` عدة قيم حدية معرفة حسب التطبيق.

9.5.1 ملف الترويسة `<limits.h>`

الترويسة `<limits.h>` يوضح الجدول 1 الأسماء المُصرّح عنها في هذا الملف وقيمها المسموحة، إضافةً إلى وصف موجز عن وظيفتها، إذ يوضح وصف `SHRT_MIN` مثلاً أن قيمة الاسم في بعض التطبيقات يجب أن تكون أقل من أو تساوي القيمة `-32767`، وهذا يعني أن البرنامج لا يستطيع الاعتماد على متغيرات صغيرة `short` لتخزين قيم سالبة تتعدى `-32767` إذا أردنا قابلية نقل أكبر للبرنامج. قد يدعم التطبيق في بعض الأحيان القيم السالبة الأكبر إلا أن الحد الأدنى الذي يجب أن يدعمه التطبيق هو `-32767`.

الجدول 19: أسماء ملف الترويسة `<limits.h>`

الاسم	القيم المسموحة	الوصف
CHAR_BIT	($8 \leq$)	بتات في قيمة من نوع char
CHAR_MAX	اقرأ الملاحظة	القيمة العظمى لنوع char
CHAR_MIN	اقرأ الملاحظة	القيمة الدنيا لنوع char
INT_MAX	($+32767 \leq$)	القيمة العظمى لنوع int
INT_MIN	($-32767 \geq$)	القيمة الدنيا لنوع int
LONG_MAX	($+2147483647 \leq$)	القيمة العظمى لنوع long
LONG_MIN	($-2147483647 \geq$)	القيمة الدنيا لنوع long

الوصف	القيم المسموحة	الاسم
عدد البتات الأعظمي في محرف متعدد البتات multibyte character	(1≤)	MB_LEN_MAX
القيمة العظمى لنوع signed char	(+127≤)	SCHAR_MAX
القيمة الدنيا لنوع signed char	(-127≥)	SCHAR_MIN
القيمة العظمى لنوع short	(+32767≤)	SHRT_MAX
القيمة الدنيا لنوع short	(-32767≥)	SHRT_MIN
القيمة العظمى لنوع unsigned char	(255U≤)	UCHAR_MAX
القيمة الدنيا لنوع unsigned int	(65535U≤)	UINT_MAX
القيمة العظمى لنوع unsigned long	(4294967295U≤)	ULONG_MAX
القيمة الدنيا لنوع unsigned short	(65535U≤)	USHRT_MAX

ملاحظة: إذا كان التطبيق يعامل char على أنه من نوع ذو إشارة فقيمة CHAR_MAX و CHAR_MIN مماثلة لقيمة SCHAR الموافق لها، وإلا فقيمة CHAR_MIN هي صفر وقيمة CHAR_MAX هي مساوية لقيمة UCHAR_MAX.

9.5.2 ملف الترويسة <float.h>

يتضمن ملف الترويسة <float.h> قيمًا دنيا للأرقام ذات الفاصلة العائمة floating point بصورة مشابهة لما سبق، ويمكن الافتراض عند عدم وجود قيمة دنيا لنوع ما أن هذا النوع لا يمتلك قيمة دنيا أو أن القيمة مرتبطة بقيمة أخرى.

الجدول 20: أسماء ملف الترويسة <float.h>

الوصف	القيم المسموحة	الاسم
تمثيل أساس الأس	(2≤)	FLT_RADIX
عدد خانات الدقة في نوع double	(10≤)	DBL_DIG
العدد الموجب الأدنى الذي يحقق $1.0 + x \neq 1.0$	(1E-9≥)	DBL_EPSILON
عدد خانات أساس FLT_RADIX في الجزء العشري من النوع double	(-)	DBL_MANT_DIG
القيمة العظمى لنوع double	(1E+37≤)	DBL_MAX
القيمة العظمى لأس أساسه 10 من نوع double	(+37≤)	DBL_MAX_10_EXP
القيمة العظمى لأس أساسه FLT_RADIX من نوع double	(-)	DBL_MAX_EXP
القيمة الدنيا للنوع double	(1E-37≥)	DBL_MIN
القيمة الدنيا لأس أساسه 10 من نوع double	(37≥)	DBL_MIN_10_EXP

الوصف	القيم المسموحة	الاسم
القيمة الدنيا لأس أساسه FLT_RADIX من نوع double	(-)	DBL_MIN_EXP
عدد خانات الدقة في نوع float	(6≤)	FLT_DIG
العدد الموجب الأدنى الذي يحقق $1.0 + x \neq 1.0$	(1E-5≥)	FLT_EPSILON
عدد خانات أساس FLT_RADIX في الجزء العشري من النوع float	(-)	FLT_MANT_DIG
القيمة العظمى للنوع float	(1E+37≤)	FLT_MAX
القيمة العظمى لأس أساسه 10 من نوع float	(+37≤)	FLT_MAX_10_EXP
القيمة العظيمة لأس أساسه FLT_RADIX من نوع float	(-)	FLT_MAX_EXP
القيمة الدنيا للنوع float	(1E-37≥)	FLT_MIN
القيمة الدنيا لأس (أساسه 10) من نوع float	(-37≥)	FLT_MIN_10_EXP
القيمة الدنيا لأس أساسه FLT_RADIX من نوع float	(-)	FLT_MIN_EXP
يحدد التقريب للفاصلة العائمة، غير مُحدّد لقيمة -1، تقريب باتجاه الصفر لقيمة 0، تقريب للقيمة الأقرب لقيمة 1، تقريب إلى الا نهاية الموجبة لقيمة 2، تقريب إلى الا نهاية السالبة لقيمة 3. أي قيمة أخرى تكون محددة بحسب التطبيق	(0)	FLT_ROUNDS
عدد خانات الدقة في نوع long double	(10≤)	LDBL_DIG
العدد الموجب الأدنى الذي يحقق $1.0 + x \neq 1.0$	(1E-9≥)	LDBL_EPSILON
عدد خانات أساس FLT_RADIX في الجزء العشري من النوع long double	(-)	LDBL_MANT_DIG
القيمة العظمى للنوع long double	(1E+37≤)	LDBL_MAX
القيمة العظمى لأس أساسه 10 من نوع long double	(+37≤)	LDBL_MAX_10_EXP
القيمة العظمى لأس أساسه FLT_RADIX من نوع long double	(-)	LDBL_MAX_EXP
القيمة الدنيا للنوع long double	(1E-37≥)	LDBL_MIN
القيمة الدنيا لأس -أساسه -10 من نوع long double	(-37≥)	LDBL_MIN_10_EXP
القيمة الدنيا لأس -أساسه -FLT_RADIX من نوع long double	(-)	LDBL_MIN_EXP

9.6 الدوال الرياضية

إذا كنت تكتب برامجاً رياضيةً تجري عمليات على الفاصلة العائمة وما شابه، فهذا يعني أنك تحتاج الوصول إلى مكتبات الدوال الرياضية دون أدنى شك، ويأخذ هذا النوع من الدوال وسطاءً من النوع `double` ويعيد نتيجةً من النوع ذاته أيضاً. تُعرّف الدوال والماكرو المرتبطة بها في ملف الترويسة `<math.h>`.

يُستبدل الماكرو `HUGE_VAL` المُعرّف إلى تعبير ذي قيمة موجبة من نوع عدد عشري مضاعف الدقة "double"، ولا يمكن تمثيله بالضرورة باستخدام النوع `float`.

نحصل على **خطأ نطاق domain error** في جميع الدوال إذا كانت قيمة الوسيط المُدخل خارج النطاق المُعرّف للدالة، مثل محاولة الحصول على جذر تربيعي لعدد سالب، وإذا حصل هذا الخطأ يُضبط `errno` إلى الثابت `EDOM`، وتُعيد الدالة قيمة معرفّة بحسب التطبيق.

نحصل على **خطأ مجال range error** إذا لم يكن من الممكن تمثيل نتيجة الدالة بقيمة عدد عشري مضاعف الدقة، تُعيد الدالة القيمة `±HUGE_VAL` إذا كانت قيمة النتيجة كبيرة جداً (الإشارة موافقة للقيمة) وتُضبط `errno` إلى `ERANGE` إذا كانت القيمة صغيرة جداً وتُعاد القيمة `0.0` وتُعتمد قيمة `errno` على تعريف التطبيق.

تصف اللائحة التالية كلاً من الدوال المتاحة باختصار:

- الدالة `double acos(double x)`: تُعيد القيمة الرئيسة `Principal value` لقوس جيب التمام `Arc cosine` للوسيط `x` في النطاق من `0` إلى `π` راديان، ونحصل على الخطأ `EDOM` إذا كان `x` خارج النطاق `-1` إلى `1`.
- الدالة `double asin(double x)`: تُعيد القيمة الرئيسة لقوس الجيب `Arc sin` للوسيط `x` في النطاق من `-π/2` إلى `+π/2` راديان، ونحصل على الخطأ `EDOM` إذا كان `x` خارج النطاق `-1` إلى `1`.
- الدالة `double atan(double x)`: تُعيد القيمة الرئيسة لقوس الظل `Arc tan` للوسيط `x` في النطاق من `-π/2` إلى `+π/2` راديان.
- الدالة `double atan2(double y, double x)`: تُعيد القيمة الرئيسة لقوس الظل للقيمة `y/x` في النطاق من `-π` إلى `+π` راديان، وتستخدم إشارتي الوسيطين `x` و `y` لتحديد الربع الذي تقع فيه قيمة الإجابة، ونحصل على الخطأ `EDOM` في حال كان `x` و `y` مساويين إلى الصفر.
- الدالة `double cos(double x)`: تُعيد جيب تمام قيمة الوسيط `x` (تُقاس `x` بالراديان).
- الدالة `double sin(double x)`: تُعيد جيب قيمة الوسيط `x` (تُقاس `x` بالراديان).

- الدالة `double tan(double x);` تُعيد ظل قيمة الوسيط x (تُقاس x بالراديان)، وتكون إشارة HUGE_VAL غير مضمونة الصحة إذا حدث خطأ مجال.
- الدالة `double cosh(double x);` تُعيد جيب التمام القطعي Hyperbolic للقيمة x ، ونحصل على الخطأ ERANGE إذا كان مقدار x كبيرًا جدًا.
- الدالة `double sinh(double x);` تُعيد الجيب القطعي للقيمة x ، ونحصل على الخطأ ERANGE إذا كان مقدار x كبيرًا للغاية.
- الدالة `double tanh(double x);` تُعيد الظل القطعي للقيمة x .
- الدالة `double exp(double x);` دالة أسية للقيمة x ، ونحصل على الخطأ ERANGE إذا كان مقدار x كبيرًا جدًا.
- الدالة `double frexp(double value, int *exp);` تجزئة عدد ذو فاصلة عائمة إلى كسر طبيعي وأُس عدد صحيح من الأساس 2، ويخزن هذا العدد الصحيح في الغرض المُشار إليه بواسطة المؤشر `exp`.
- الدالة `double ldexp(double x, int exp);` ضرب x بمقدار 2 إلى الأس `exp`، وقد نحصل على الخطأ ERANGE.
- الدالة `double log(double x);` اللوغاريتم الطبيعي للقيمة x ، وقد نحصل على الخطأ EDOM إذا كانت القيمة x سالبة، و ERANGE إذا كانت x تساوي إلى الصفر.
- الدالة `double log10(double x);` اللوغاريتم ذو الأساس 10 للقيمة x ، ونحصل على الخطأ EDOM إذا كانت x سالبة، و ERANGE إذا كانت x تساوي إلى الصفر.
- الدالة `double modf(double value, double *iptr);` تجزئة قيمة الوسيط `value` إلى جزء عدد صحيح وجزء كسري، ويحمل كل جزء إشارة الوسيط ذاتها، وتُخزن قيمة العدد الصحيح على أنها قيمة من نوع `double` في الكائن المُشار إليه بواسطة المؤشر `iptr` وتُعيد الدالة الجزء الكسري.
- الدالة `double pow(double x, double y);` تحسب x إلى الأس y ، ونحصل على الخطأ EDOM إذا كانت القيمة x سالبة و y عدد غير صحيح، أو ERANGE إذا لم يكن من الممكن تمثيل النتيجة في حال كانت x تساوي إلى الصفر و y موجبة أو تساوي الصفر.
- الدالة `double sqrt(double x);` تحسب مربع القيمة x ، ونحصل على الخطأ EDOM إذا كانت x سالبة.
- الدالة `double ceil(double x);` أصغر عدد صحيح لا يكون أصغر من x .

- الدالة `double fabs(double x);`: القيمة المطلقة للوسيط `x`.
- الدالة `double floor(double x);`: أكبر عدد صحيح لا يكون أكبر من `x`.
- الدالة `double fmod(double x, double y);`: الباقي العشري من عملية القسمة `x/y`، ويعتمد الأمر على تعريف التطبيق فيما إذا كانت `fmod` تُعيد صفرًا أو خطأ نطاق في حال كانت `y` تساوي إلى الصفر.

9.7 القفزات اللامحلية Non-local jumps

تقدم القفزات اللامحلية `non-local jumps` طريقةً مشابهة لطريقة `goto` بالانتقال من دالة إلى أخرى. نلجأ إلى استخدام الماكرو `setjmp` والدالة `longjmp` لأن الأمر غير ممكن الحدوث باستخدام `goto` والعناوين `labels` إذ أن للعناوين نطاق `scope` داخل الدالة فقط، وتُعرف هذه الطريقة باسم `goto` اللامحلية أو القفزة اللامحلية.

يصرح ملف الترويسة `<setjmp.h>` شيئاً يدعى `jmp_buf`، وهو اسم مستخدم في الماكرو والدالة لتخزين المعلومات الضرورية لإجراء القفزة، وتُكتب التصاريح على النحو التالي:

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

يُستخدم الماكرو `setjmp` لتهيئة قيمة `jmp_buf` ويُعيد القيمة صفر عند استدعائه الأولي، إلا أن الأمر غير الاعتيادي هنا، هو أنه يُعيد **مجددًا** قيمة غير صفرية لاحقًا عند استدعاء الدالة `longjmp`، وتكون القيمة غير الصفرية هذه مساويةً للقيمة المُمرّرة للدالة `longjmp`. لعل الأمر سيتضح لك بوضوح بعد المثال التالي:

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

void func(void);
jmp_buf place;

main(){
    int retval;

    /*
```

```

    يُعيد الاستدعاء الأول القيمة 0، ويعيد استدعاء آخر للدالة longjmp قيمة غير صفرية
    */
    if(setjmp(place) != 0){
        printf("Returned using longjmp\n");
        exit(EXIT_SUCCESS);
    }

    /*
    لن يُعيد الاستدعاء التالي أي قيمة لأنه يقفز مجددًا إلى الأعلى
    */
    func();
    printf("What! func returned!\n");
}

void
func(void){
    /*
    العودة إلى main، ويبدو أن الاستدعاء الثاني لدالة setjmp يعيد القيمة 4
    */
    longjmp(place, 4);
    printf("What! longjmp returned!\n");
}

```

[مثال 3]

يمثل وسيط الدالة longjmp المسمى val القيمة المُعادة من الاستدعاء الثاني اللاحق لتعليمة الإعادة return ضمن الدالة setjmp، ويجب أن تكون هذه القيمة قيمة غير صفرية عادةً، وستغيّر القيمة إلى 1 إذا حاولت إعادة القيمة صفر باستخدام longjmp، وبذلك يمكننا معرفة فيما إذا كان استدعاء الدالة setjmp مباشرةً أو عن طريق استدعاء الدالة longjmp.

تأثير الدالة setjmp غير محدد إذا لم يكن هناك أي استدعاء لها قبل استدعاء longjmp، وسيتسبب ذلك غالبًا بتوقف البرنامج. لا يُتوقّع من الدالة longjmp أن تُعيد قيمة بعد استدعائها مباشرةً. يكون لجميع الكائنات الممكن الوصول إليها من تعليمة الإعادة return داخل الدالة setjmp القيم السابقة المخزنة عند استدعاء longjmp عدا الكائنات ذات صنف التخزين التلقائي automatic storage class التي لا تحتوي على نوع "volatile"، وتكون قيمها غير محددة إذا تغيرت هذه الكائنات بين استدعاء setjmp واستدعاء longjmp.

تُنفَّذ الدالة `longjmp` على نحوٍ صحيح بخصوص المقاطعات `interrupts` والإشارات وأي دوال أخرى مرتبطة، ونحصل على سلوك غير معرف إذا حصل استدعاء `longjmp` باستخدام دالة نتج استدعاؤها عن إشارة وصلت بينما تُعالج إشارة أخرى.

يُعدُّ القفز إلى دالة غير فعالة باستخدام `longjmp` خطأً فادحًا (ويُقصد بدالة غير فعالة أنها أعادت قيمة للتو، أو أن استدعاء `longjmp` آخر تحوّل إلى `setjmp` ضمن مجموعة من الاستدعاءات المترابطة (nested calls).

يصرّ المعيار على أن `setjmp` يجب أن تستخدم فقط مثل تعبير للتحكم في تعليمات `if` و `switch` و `do` و `while` و `for` (إضافةً إلى كونها التعليمة الوحيدة الموجودة في تعليمة تعبير)، وامتدادًا لهذه القاعدة، يمكن لاستدعاء `setjmp` (طالما يشكّل تعبير التحكم بأكمله كما ذكرنا سابقًا) أن يخضع للعامل `!`، أو أن يُقارن مباشرةً مع تعبير ثابت ذي قيمة عدد صحيح باستخدام إحدى العوامل العلائقية أو عوامل المساواة، ولا يجب استخدام أي تعابير معقدة أكثر من ذلك. إليك الأمثلة التالية:

```
setjmp(place);                /* تعليمة تعبير */
if(setjmp(place)) ...        /* تعبير تحكم كامل */
if(!setjmp(place)) ...      /* تعبير تحكم كامل */
if(setjmp(place) < 4) ...    /* تعبير تحكم كامل */
if(setjmp(place)<;4 && 1!=2) ... /* ممنوع */
```

9.8 التعامل مع الإشارة

تقدّم لنا دالتان إمكانية التعامل مع الأحداث غير المتزامنة؛ وتُعرف الإشارة `signal` بأنها شرط قد يحدث خلال تنفيذ البرنامج ويمكن تجاهله أو التعامل معه بصورة خاصة أو استخدامه لإنهاء البرنامج كما هي الحالة الاعتيادية. تُرسل إحدى الدوال الإشارة بينما تُستخدم الأخرى لتحديد كيفية معالجة الإشارة، وقد تولّد الكثير من الإشارات من العتاد الصلب أو نظام التشغيل إضافةً إلى دوال إرسال الإشارات `raise`.

الإشارات المُعرّفة في ملف الترويسة `<signal.h>`، هي:

- الإشارة `SIGABRT`: إنهاء غير اعتيادي للبرنامج، مثل الإنهاء الحاصل باستخدام الدالة `abort` (إبطال).
- الإشارة `SIGFPE`: عملية حسابية خاطئة، مثل التقسيم على الصفر أو الطفحان `overflow` (استثناء الفاصلة والأرقام العشرية `Floating point exception`).
- الإشارة `SIGILL`: العثور على "كائن برنامج غير صالح"، وهذا يعني غالبًا أن هناك تعليمات غير صالحة في البرنامج. (تعليمة غير صالحة `Illegal instruction`).

- الإشارة SIGINT: إشارة تفاعلية للفت الانتباه، وتولد هذه الإشارة على الأنظمة التفاعلية عادةً بكتابة مفتاح الهروب break-in في الطرفية terminal (مقاطعة Interrupt).
- الإشارة SIGSEGV: محاولة غير صالحة للوصول إلى مساحة تخزين، وتُسبب غالبًا بمحاولة تخزين قيمة في كائن مُشار إليه بمؤشر خاطئ. (انتهاك جزء segment violation).
- الإشارة SIGTERM: طلب إنهاء للبرنامج. (إنهاء Terminate).

قد تمتلك بعض التنفيذات implementations بعض الإشارات الإضافية الزائدة عن الإشارات السابقة المعرفة في المعيار، وستبدأ أسماء الإشارات بالأحرف SIG وستملك قيمًا مميزة مختلفة عن القيم السابقة. تسمح لك الدالة signal بتحديد الفعل الذي تريد اتخاذه عند تلقي إشارة، وتُصطحب بحالة إشارة من الإشارات المذكورة سابقًا ومؤشر يشير إلى دالة تُنقذ للتعامل مع الإشارة، وذلك بتغيير المؤشر وإعادة القيمة الأصلية، إذًا، نعرف الدالة كما يلي:

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);
```

يدل ما سبق على أن الدالة signal تُعيد مؤشرًا يشير إلى دالة أخرى وتأخذ الدالة الثانية وسيطًا واحدًا من نوع عدد صحيح وتُعيد void؛ بينما يكون الوسيط الثاني للدالة signal مؤشرًا يشير إلى دالة تعيد void بصورة مشابهة، وتأخذ int وسيطًا لها.

يمكن استخدام قيمتين مميزتين لوسيط لدالة func (دالة التعامل مع الإشارة)، ألا وهما SIG_DFL وهي معالج الإشارة الافتراضي الأولي و SIG_IGN الذي يُستخدم لتجاهل الإشارة، ويضبط التنفيذ حالة جميع الإشارات إلى واحدة من هذه القيمتين في بداية البرنامج.

تُعاد قيمة func السابقة للإشارة إذ استدعيت signal بنجاح، وإلا فتُعاد SIG_ERR ويُضبط errno إلى قيمة.

عند حصول حدث إشارة غير مُتجاهل، يُنقذ أول signal(sig, SIG_DFL) يطابق الحالة وذلك إذا كانت الدالة func المترافقة تمثل مؤشرًا يشير إلى دالة، وتتسبب تلك العملية بإعادة تشغيل معالج الإشارة إلى الإجراء الافتراضي ألا وهو إنهاء البرنامج، وإذا كانت الإشارة هي SIGILL فسيكون إعادة التشغيل معرفًا حسب التنفيذ، إذ قد تختار بعض التنفيذات حجب أي حالات أخرى من الإشارة عوضًا عن إعادة التشغيل.

بعد ذلك، يُجرى استدعاء لدالة معالجة الإشارة، وسيعاود البرنامج عمله من نقطة حصول الحدث في معظم الحالات وذلك إذا أعادت الدالة قيمة بنجاح، إلا أننا نحصل على سلوك غير معرف إذا كانت قيمة sig مساويةً إلى SIGFPE (استثناء الفاصلة العائمة) أو أي استثناء حسابي معرف بحسب التنفيذ، والحل الأكثر استخدامًا لمعالج SIGFPE هو استدعاء إحدى الدوال: abort، أو exit، أو longjmp.

يستعرض الجزء التالي استخدام الإشارة لتحقيق خروج أنيق من البرنامج عند تلقي مقاطعة أو إشارة "الانتباه

التفاعلي "interactive attention".

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

FILE *temp_file;
void leave(int sig);

main() {
    (void) signal(SIGINT,leave);
    temp_file = fopen("tmp","w");
    for(;;) {
        /*
         * اعمل بعض الأشياء هنا
         */
        printf("Ready...\n");
        (void) getchar();
    }
    /* لا يمكننا الوصول إلى هذه النقطة */
    exit(EXIT_SUCCESS);
}

/*
 * أغلق الملف tmp عند الحصول على SIGINT، لكن انتبه
 * لأن استدعاء دوال المكتبات من معالج الإشارة غير مضمون العمل في جميع التنفيذات
 * وهذا ليس ببرنامج متجاوب مع جميع التنفيذات بالضرورة
 */

void
leave(int sig) {
    fprintf(temp_file, "\nInterrupted..\n");
    fclose(temp_file);
    exit(sig);
}
```


[مثال 4]

من الممكن للبرنامج أن يرسل إشارات إلى نفسه باستخدام دالة `raise` وهذا معرّف على النحو التالي:

```
include <signal.h>
int raise (int sig);
```

تُرسل الإشارة `sig` في هذه الحالة إلى البرنامج.

تُعيد التعليلة `raise` القيمة صفر في حال النجاح، وقيمة غير صفرية عدا ذلك، تُنفذ دالة `abort` على النحو التالي:

```
#include <signal.h>

void
abort(void) {
    raise(SIGABRT);
}
```

إذا حصلنا على إشارة لأي سبب كان -باستثناء استدعاء `abort` أو `raise`- فمن الممكن للدالة أن تستدعي فقط الإشارة أو أن تُسند قيمةً إلى كائن ساكن `static volatile` متطابق `volatile` (مؤهل باستخدام `volatile`) من النوع `sig_atomic_t`، وهذا النوع مصرّح في ملف الترويسة `<signal.h>`، وهو النوع الوحيد من الكائنات الممكن تعديله بأمان مثل كيان ذري `atomic entity` حتى مع وجود المقاطعات الا متزامنة، وهذا قيدٌ مرهق مفروض من المعيار، الذي على سبيل المثال، يُبطل الدالة `leave` في مثالنا أعلاه، وعلى الرغم من أن الدالة ستعمل بصورةٍ صحيحة في بعض البيئات إلى أنها لا تتبع القوانين الصارمة الخاصة بالمعيار.

9.9 أعداد متغيرة من الوسطاء

غالبًا ما يكون تنفيذ دالة تأخذ عددًا غير معروفًا أو غير ثابت من الوسطاء محببًا عند كتابة الدالة، نذكر دالة `printf` على سبيل المثال التي سنتكلم عنها لاحقًا. يوضح المثال التالي تصريح دالة مشابهة.

```
int f(int, ... );

int f(int, ... ) {
    .
    .
    .
}
```

```
int g() {
    f(1,2,3);
}
```

[مثال 5]

علينا تضمين الدوال المصرح عنها ضمن ملف الترويسة <stdarg.h> لكي نستطيع الوصول إلى الوسطاء الموجودة بداخل الدالة المُستدعاة، ونحصل نتيجةً لذلك على نوع جديد يدعى `va_list` وثلاثة دوال تتعامل مع كائنات من هذا النوع وتدعى `va_start` و `va_arg` و `va_end`.

علينا استدعاء `va_start` قبل محاولة الوصول إلى لائحة الوسطاء المتغيرة، وهي معرفة على النحو التالي:

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

يُهيئ الماكرو `va_start` الوسيط `ap` بهدف الاستخدام اللاحق من قبل الدالتين `va_arg` و `va_end`، بينما يكون الوسيط الثاني للدالة `va_start` المسمّى `parmN` المعرّف `identifier` الذي يسمّي المعامل الذي يقع أقصى اليمين في لائحة المعاملات المتغيرة (أي المعامل الذي يقع قبل "...")، ولا يجب التصريح عن المعرف `parmN` باستخدام صنف التخزين `class` `storage` من النوع `register` أو على أنه دالة أو نوع مصفوفة.

يمكن الوصول إلى الوسطاء على نحوٍ متتابعي بعد التهيئة وذلك باستخدام الماكرو `va_arg`، وهذا غير مألوف لأن النوع المُعاد يحدّد باستخدام وسيط للماكرو. لاحظ أن ذلك مستحيل التنفيذ في دالة فعلية، ويمكن تنفيذه فقط باستخدام الماكرو، وهو معرّف على النحو التالي:

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

سيتسبب كل استدعاء للماكرو السابق بالحصول على الوسيط التالي من لائحة الوسطاء بقيمة من النوع المُحدّد، ويجب للوسيط `va_list` أن يُهيئاً باستخدام `va_start`، ونحصل على سلوك غير معرّف إذا لم يكن الوسيط التالي من النوع المُحدّد. احذر من المشاكل التي قد تنتج من التحويلات الحسابية وتفادها، إذ أن استخدام النوع `char` أو عدد صغير `short` وسيطاً ثانيًا للدالة `va_arg` خطأ واضح؛ لأن هذه الأنواع تُرقى دائماً إلى `signed int` أو `unsigned int` ويُحوّل `float` إلى `double`.

لاحظ أن ترقية الكائنات المصرّحة عنها من الأنواع `char` و `unsigned char` و `unsigned short` وحقول البت عديمة الإشارة `unsigned bitfields` إلى النوع `unsigned int` الذي سيعقد أكثر استخدام

الدالة `va_arg` هو معرّف بحسب التنفيذ، وقد يكون ذلك هو السبب في الحصول على بعض المشاكل الخفية غير المتوقعة.

نحصل على سلوك غير معرف أيضًا إذا استُديت الدالة `va_arg` ولم يكن هناك مزيدًا من الوسطاء.

يجب أن يكون الوسيط `type` - في تعريفنا السابق لدالة `va_arg` - ممثلًا لاسم نوع يمكن تحويله إلى مؤشر يشير إلى كائن بإضافة المحرف `*` ببساطة (حتى يعمل الماكرو)، وذلك محقق لأنواع البسيطة مثل `char` (لأن `* char` يمثل نوع مؤشر يشير إلى محرف)، لكن لن تعمل مصفوفة المحارف (لا يتحول النوع `[] char` إلى مؤشر يشير إلى مصفوفة محارف بإضافة `*` إليه). يمكن لحسن الحظ معالجة المصفوفات إذا ما تذكرنا أن اسم المصفوفة الذي يُستخدم وسيطًا فعليًا لاستدعاء الدالة يُحوّل إلى مؤشر، وبذلك فإن النوع الصحيح لوسيط من النوع "مصفوفة من المحارف" هو `* char`.

تُستدعى الدالة `va_end` بعد معالجة جميع الوسطاء، وهذا سيمنع اللائحة `va_list` من أن تُستخدم بعد ذلك، ونحصل على سلوك غير معرف إذا لم تُستخدم الدالة `va_end`.

يمكن إعادة قراءة كامل لائحة الوسطاء باستدعاء الدالة `va_start` مجددًا بعد استدعاء `va_end`، وتُصرّح الدالة `va_end` كما يلي:

```
#include <stdarg.h>
void va_end(va list ap);
```

يوضح المثال التالي كيفية استخدام كل من `va_start` و `va_arg` و `va_end` ضمن دالة تُعيد أكبر قيم وسطائها التي تكون من نوع عدد صحيح.

```
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>

int maxof(int, ...);
void f(void);

main(){
    f();
    exit(EXIT_SUCCESS);
}

int maxof(int n_args, ...){
    register int i;
```

```

int max, a;
va_list ap;

va_start(ap, n_args);
max = va_arg(ap, int);
for(i = 2; i <= n_args; i++) {
    if((a = va_arg(ap, int)) > max)
        max = a;
}

va_end(ap);
return max;
}

void f(void) {
    int i = 5;
    int j[256];
    j[42] = 24;
    printf("%d\n", maxof(3, i, j[42], 0));
}

```

[مثال 6]

9.10 الدخل والخرج I/O

9.10.1 مقدمة

يعدّ افتقار لغات البرمجة لدعمها للدخل والخرج إحدى أبرز الأسباب التي منعت التبني واسع النطاق واستخدامها في البرمجة العملية، وهو الموضوع الذي لم يرد أي مصمّم لغة أن يتعامل معه، إلا أن لغة سي تفادت هذه المشكلة، بعدم تضمينها لأي دعم للدخل والخرج، إذ كان سلوك سي هو أن تترك التعامل مع الدخل والخرج لدوال المكتبات، مما عني أنه بالإمكان لمصممي الأنظمة استخدام طرق دخل وخرج مخصصة بدلاً من إجبارهم على تغيير اللغة بذات نفسها.

تطوّرت حزمة مكتبات عُرفت باسم "مكتبة الدخل والخرج القياسي Standard I/O Library" -أو اختصارًا `stdio`- في الوقت ذاته الذي كانت لغة سي تتطوّر، وقد أثبتت هذه المكتبة مرونتها وقابلية نقلها وأصبحت الآن جزءاً من المعيار.

اعتمدت حزمة الدخل والخرج القياسية كثيرًا على نظام يونيكس UNIX للوصول إلى الملفات وبالأخص الافتراض أنه لا يوجد أي فرق بين ملفات ثنائية غير مُهيكله وملفات أخرى تحتوي على نص مقروء، إلا أن العديد من أنظمة التشغيل تفصل ما بين الاثنين وعُدلت الحزمة فيما بعد لضمان قابلية نقل برامج لغة سي بين نوعي نظام الملفات. هناك بعض التغييرات في هذا المجال التي تتسبب بالضرر لكثير من البرامج المكتوبة مسبقًا على الرغم من الجهود التي تحاول أن تحدّ من هذا الضرر.

من المفترض أن تعمل برامج لغة سي القديمة بنجاح دون تعديل في بيئة يونيكس.

9.10.2 نموذج الدخل والخرج

لا يُميّز نموذج الدخل والخرج بين أنواع الأجهزة المادية التي تدعم الدخل والخرج، إذ يُعامل كل مصدر أو حوض من البيانات بالطريقة ذاتها ويُنظر إليه على أنه مجرّي من البايتات stream of bytes. بما أن الكائن الأصغر الذي يمكن تمثيله في لغة سي هو المحرف، فالوصول إلى الملف مسموح باستخدام حدود أي محرف، وبالتالي يمكن قراءة أو كتابة أي عدد من المحارف انطلاقًا من نقطة متحركة تُعرف باسم مؤشر الموضع position indicator، وتُكتب أو تُقرأ المحارف تبعًا بدءًا من هذه النقطة ويُحرّك مؤشر الموضع خلال ذلك. يُضبط مؤشر الموضع مبدئيًا إلى بداية الملف عند فتحه، لكن من الممكن تحريكه باستخدام طلبات تحديد الموقع، ويُتجاهل مؤشر موضع الملف في حال كان الوصول العشوائي إلى الملف غير ممكن. لفتح الملف في نمط الإضافة append تأثيرات على مجرى موضع المؤشر في الملف معرفة بحسب التنفيذ.

الفكرة العامة هي تقديم إمكانية القراءة أو الكتابة بصورةٍ تتابعية، باستثناء حالة فتح المجرى باستخدام نمط الإضافة، أو إذا حُرّك مؤشر موضع الملف مباشرةً.

هناك نوعان من أنواع الملفات، هما: **الملفات النصية text files والملفات الثنائية binary files** التي يمكن التعامل معها داخل البرنامج على أنها **مجاري نصية text streams** أو **مجاري ثنائية binary streams** بعد فتحها لعمليات الإدخال والإخراج. لا تسمح حزمة stdio بالعمليات على محتوى الملف مباشرةً، بل بالتعديل على المجرى الذي يحتوي على بيانات الملف.

1. المجاري النصية

يحدّد المعيار المجرى النصي text stream، الذي يمثّل ملفًا يحتوي على أسطر نصية ويتألف السطر من صفر محرف أو أكثر ينتهي بمحرف نهاية السطر، ومن الممكن أن يكون تمثيل الأسطر الفعلي في البيئة الخارجية مختلفًا عن تمثيله هنا، كما من الممكن إجراء تحويلات على مجرى البيانات عند دخولها إلى أو خروجها من البرنامج، وأكثر المتطلبات شيوعًا هو ترجمة المحرف الذي ينهي السطر "\n" إلى السلسلة "\r\n" عند الخرج وإجراء عكس العملية عند الدخل، ومن الممكن تواجد بعض الترجمات الضرورية الأخرى.

يُضمن للبيانات التي تُقرأ من المجرى النصي أن تكون مساويةً إلى البيانات المكتوبة سابقًا إلى الملف، وذلك إذا كانت هذه البيانات مؤلفةً من أسطر مكتملة تحتوي على محارف يمكن طباعتها، وكانت محارف

التحكم control characters ومحارف مسافة الجدولة الأفقية horizontal-tab ومحارف الأسطر الجديدة newline فقط، ولم يُتبع أي محرف سطر جديد بمحرف مسافة فارغة space مباشرةً، وكان المحرف الأخير في المجرى هو محرف سطر جديد.

كما أن هناك ضمان بأن المحرف الأخير المكتوب إلى الملف النصي هو محرف سطر جديد، ومن الممكن قراءة الملف مجددًا بمحتوياته المماثلة التي كُتبت إليه سابقًا.

إلحاق المحرف الأخير المكتوب إلى الملف بمحرف سطر جديد معرفٌ بحسب التنفيذ، وذلك لأن الملفات النصية والملفات الثنائية تُعامل نفس المعاملة في بعض التنفيذات.

قد تُجرّد بعض التنفيذات المسافة الفارغة البادئة من الأسطر التي تتألف من مسافات فارغة فقط متبوعةً بسطر جديد، أو تُجرّد المسافة الفارغة في نهاية السطر.

يجب أن يدعم التنفيذ الملفات النصية التي تحتوي سطورها على 254 محرفًا على الأقل، ويتضمن ذلك محرف السطر الجديد الذي يُنهي السطر.

قد نحصل على مجرى ثنائي عند فتح مجرى نصي بنمط التحديث update mode في بعض التنفيذات.

قد تتسبب الكتابة على مجرى نصي باقتطاع الملف عند نقطة الكتابة في بعض التنفيذات، أي ستُهمل جميع البيانات التي تتبع البايت الأخير المكتوب.

ب. المجاري الثنائية

يمثل المجرى الثنائي سلسلةً من المحارف التي يمكن استخدامها لتسجيل البيانات الداخلية لبرنامج ما، مثل محتويات الهياكل structures، أو المصفوفات وذلك بالشكل الثنائي، إذ تكون البيانات المقروءة من المجاري الثنائية مساويةً للبيانات المكتوبة إلى المجرى ذاته سابقًا ضمن نفس التنفيذ، وقد يُضاف عددٌ من المحارف الفارغة "NULL" في بعض الظروف إلى نهاية المجرى الثنائي، ويكون عدد المحارف معرفًا بحسب التنفيذ.

تعتمد بيانات الملفات الثنائية على الآلة التي تعمل عليها لأبعد حد، وهي غير قابلة للنقل عمومًا.

ج. المجاري الأخرى

قد تتوفر بعض أنواع المجاري الأخرى، إلا أنها معرفة بحسب التنفيذ.

9.10.3 ملف الترويسة <stdio.h>

هناك عدد من الدوال والماكرو الموجودة لتقديم الدعم لمختلف أنواع المجاري، ويحتوي ملف الترويسة <stdio.h> العديد من التصريحات المهمة لهذه الدوال، إضافةً إلى الماكرو التالية وتصاريح الأنواع:

- النوع FILE: نوع الكائن المُستخدم لاحتواء معلومات التحكم بالمجرى، ولا يحتاج مستخدمو مكتبة "stdio" لمعرفة محتويات هذه الكائنات، إذ يكفي التعامل مع المؤشرات التي تشير إليهم. لا يُعد نسخ الكائنات هذه ضمن البرنامج آمنًا، إذ أن عناوينهم قد تكون في بعض الأحيان معقدة.
- النوع fpos_t: نوع الكائن الذي يُستخدم لتسجيل القيم الفريدة من نوعها التي تنتمي إلى مجرى مؤشر موضع الملف.
- القيم _IOFBF و _IOLBF و _IONBF: وهب قيم تُستخدم للتحكم بالتخزين المؤقت buffering للمجرى بالاستعانة بالدالة setvbuf.
- القيمة BUFSIZ: حجم التخزين المؤقت المُستخدم بواسطة الدالة setbuf، وهو تعبير رقم صحيح integral ثابت constant تكون قيمته 256 على الأقل.
- القيمة EOF: تعبير رقم صحيح سالب ثابت يحدد نهاية الملف end-of-file ضمن مجرى، أي عند الوصول إلى نهاية الدخل.
- القيمة FILENAME_MAX: الطول الأعظمي الذي يمكن لاسم ملف أن يكون إذا كان هناك قيد على ذلك، وإلا فهو الحجم الذي يُنصح به لمصفوفة تحمل اسم ملف.
- القيمة FOPEN_MAX: العدد الأدنى من الملفات التي يضمن التنفيذ فتحها في وقت آني، وهو ثمانية ملفات. لاحظ أنه من الممكن إغلاق ثلاث مجاري مُعرفة مسبقًا إذا احتاج البرنامج فتح أكثر من خمسة ملفات مباشرةً.
- القيمة L_tmpnam: الطول الأعظمي المضمون لسلسلة نصية في tmpnam، وهو تعبير رقم صحيح ثابت.
- القيم SEEK_CUR و SEEK_END و SEEK_SET: تعابير رقم صحيح ثابتة تُستخدم للتحكم بأفعال fseek.
- القيمة TMP_MAX: العدد الأدنى من أسماء الملفات الفريدة من نوعها المولدة من قبل tmpnam، وهو تعبير رقم صحيح ثابت بقيمة لا تقل عن 25.
- الكائنات stdin و stdout و stderr: وهي كائنات معرفة مسبقًا من النوع "FILE" وتشير إلى مجرى الدخل القياسي ومجرى الخرج القياسي ومجرى الخطأ بالترتيب، وتُفتح هذه المجاري تلقائيًا عند بداية تنفيذ البرنامج.

9.10.4 العمليات على المجاري

بعد أن تعرفنا على أنواع المجاري وطبيعتها والقيم المرتبطة بها، نستعرض الآن العمليات الأساسية عليها ألا وهي فتح المجرى وإغلاقه، إضافةً إلى التخزين المؤقت.

أ. فتح المجرى

يتصل المجرى بالملف عن طريق دالة `fopen`، أو `freopen`، أو `tmpfile`، إذ تعيد هذه الدوال -إذا كان استدعاؤها ناجحًا- مؤشرًا يشير إلى كائن من نوع `FILE`.

هناك ثلاث أنواع من المجاري المتاحة افتراضيًا دون أي جهد إضافي مطلوب منك، وتتصل هذه المجاري عادةً بالجهاز المادي المرتبط بالبرنامج المُنفَّذ ألا وهو الطرفية Terminal عادةً، ويشار إلى هذه المجاري بالأسماء:

- `stdin`: وهو مجرى الدخل القياسي `standard input`.
- `stdout`: وهو مجرى الخرج القياسي `standard output`.
- `stderr`: وهو مجرى الخطأ القياسي `standard error`.

ويكون دخل لوحة المفاتيح في الحالة الطبيعية من المجرى `stdin` وخرج الطرفية هو `stdout`، بينما تُوجّه رسائل الأخطاء إلى المجرى `stderr`. الهدف من فصل رسائل الأخطاء عن رسائل الخرج العادية هو السماح بربط مجرى `stdout` إلى شيءٍ آخر مغاير لجهاز للطرفية مثل ملف ما والحصول على رسائل الخطأ بنفس الوقت على الشاشة أمامك عوضًا عن توجيه الأخطاء إلى الملف، وتخزين كامل الملفات مؤقتًا إذا لم توجّه إلى أجهزة تفاعلية.

كما ذكرنا سابقًا، قد يكون مؤشر موضع الملف قابلاً للتحريك أو غير قابل للتحريك بحسب الجهاز المُستخدم، إذ يكون مؤشر موضع الملف غير قابل للتحريك ضمن مجرى `stdin` على سبيل المثال إذا كان متصلًا إلى الطرفية (الحالة الاعتيادية له).

جميع الملفات غير المؤقتة تمتلك اسمًا `filename` وهو سلسلة نصية، والقوانين التي تحدد اسم الملف الصالح معرفةً حسب التنفيذ، وينطبق الأمر ذاته على إمكانية فتح الملف لعدة مرات بصورةٍ آتية. قد يتسبب فتح ملف جديد بإنشاء هذا الملف، وتتسبب إعادة إنشاء ملف موجود مسبقًا بإهمال محتوياته السابقة.

ب. إغلاق المجرى

تُغلق المجاري عند استدعاء `fclose` أو `exit` بصورةٍ صريحة، أو عندما يعود البرنامج إلى الدالة `main`، وتُمسح جميع البيانات المخزنة مؤقتًا عند إغلاق المجرى. تصبح حالة الملفات المفتوحة غير معروفة إذا توقف البرنامج لسببٍ ما دون استخدام الطرق السابقة لإغلاقه.

ج. التخزين المؤقت للمجرى

هناك ثلاث أنواع للتخزين المؤقت:

1. دون تخزين مؤقت `unbuffered`: تُستخدم مساحة التخزين بأقل ما يمكن من قبل `stdio` بهدف إرسال أو تلقي البيانات أسرع ما يمكن.
 2. تخزين مؤقت خطي `line buffered`: تُعالج المحارف سطرًا تلو سطر، ويُستخدم هذا النوع من التخزين المؤقت كثيرًا في البيئات التفاعلية، وتُمسح محتويات الذاكرة المؤقتة الداخلية `internal buffers` فقط عندما تمتلئ أو عندما يُعالج سطر جديد.
 3. التخزين المؤقت الكامل `fully buffered`: تُسمح الذاكرة المؤقتة الداخلية فقط عندما تمتلئ.
- يُمكن مسح محتوى الذاكرة الداخلية المرتبطة بمجرى ما عن طريق استخدام `fflush` مباشرة. يُعرّف الدعم لأنواع التخزين المؤقت المختلفة بحسب التنفيذ، ويمكن التحكم به ضمن الحدود المعرفة باستخدام `setbuf` و `setvbuf`.

د. التلاعب بمحتويات الملف مباشرة

هناك عدة دوال تسمح لنا بالتعامل مع الملف مباشرة.

```
#include <stdio.h>

int remove(const char *filename);
int rename(const char *old, const char *new);
char *tmpnam(char *s);
FILE *tmpfile(void);
```

- الدالة `remove`: تتسبب بإزالة الملف، وستفشل محاولات فتح هذا الملف لاحقًا إلا في حال إنشاء الملف مجددًا. يكون سلوك الدالة `remove` عندما يكون الملف مفتوحًا معرفًا بحسب التنفيذ، وتعيد الدالة القيمة صفر للدلالة على النجاح، بينما تدل أي قيمة أخرى على فشل عملها.
- الدالة `rename`: تُغيّر اسم الملف المعرف بالكلمة `old` في مثالنا السابق إلى `new`، وستفشل محاولات فتح الملف باستخدام اسمه القديم، إلا إذا أنشئ ملف جديد يحمل الاسم القديم ذاته، وكما هو الحال في `remove` فإن الدالة `rename` تُعيد القيمة صفر للدلالة على نجاح العملية وأي قيمة مغايرة لذلك تدل على حصول خطأ. السلوك معرف حسب التنفيذ إذا حاولنا تسمية الملف باسم جديد باستخدام `rename` وكان هناك ملف بالاسم ذاته مسبقًا. لن يُعدّل على الملف إذا فشلت الدالة `rename` لأي سببٍ كان.

- الدالة `tmpnam`: تولّد سلسلة نصية لتُستخدم اسمًا لملف، ويضمن لهذه السلسلة النصية أن تكون فريدةً من نوعها بالنسبة لأي اسم ملف آخر موجود، ويمكن أن تُستدعى بصورة متتالية للحصول على اسم جديد كل مرة. يُستخدم الثابت `TMP_MAX` لتحديد عدد مرات استدعاء الدالة `tmpnam` قبل أن يتعذر عليه العثور على اسماء فريدة، وقيمته 25 على الأقل، ونحصل على سلوك غير معرّف من قبل المعيار في حال استدعاء الدالة `tmpnam` عدد مرات يتجاوز هذا الثابت إلا أن الكثير من التنفيذات تقدم حدًا لا نهائيًا. تستخدم `tmpnam` ذاكرة مؤقتة داخلية لبناء الاسم وتُعيد مؤشرًا يشير إليه وذلك إذا صُبط الوسيط `s` إلى القيمة `NULL`، وقد تغيّر الاستدعاءات اللاحقة للدالة الذاكرة المؤقتة الداخلية ذاتها. يمكن استخدام مؤشر يشير إلى مصفوفة مثل وسيط بدلاً من السابق، بحيث تحتوي المصفوفة على `L_tmpnam` محرف على الأقل، وفي هذه الحالة سيملاً الاسم إلى الذاكرة المؤقتة المزوّدة (المصفوفة). ويمكن فيما بعد إنشاء ملف بهذا الاسم واستخدامه ملفًا مؤقتًا. لن يكون اسم الملف مفيدًا ضمن سياقات أخرى غالبًا، بالنظر إلى توليده من قبل الدالة. لا تُزال الملفات المؤقتة من هذا النوع إلا إن استدعيت دالة الحذف، وغالبًا ما تُستخدم هذه الملفات لتميرير البيانات المؤقتة بين برنامجين منفصلين.

- الدالة `tmpfile`: تُنشئ ملف ثنائي مؤقت يُمكن التعديل على محتوياته، وتعيد الدالة مؤشرًا يشير إلى مجرى الملف، ويُزال هذا الملف فيما بعد عند إغلاق مجراه، وتُعيد الدالة `tmpfile` مؤشرًا فارغًا `null` إذا لم ينجح فتح الملف.

9.10.5 فتح الملفات بالاسم

يمكن فتح الملفات الموجودة بالاسم عن طريق استدعاء الدالة `fopen` المصرّح عنها على النحو التالي:

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *mode);
```

يمثل الوسيط `pathname` اسم الملف الذي تريد فتحه، مثل الاسم الذي تعيده الدالة `tmpnam` أو أي اسم ملف معين آخر.

يمكن فتح الملفات باستخدام عدة أنماط `modes`، مثل نمط القراءة `read` لقراءة البيانات، ونمط الكتابة `write` لكتابة البيانات وهكذا.

لاحظ أن الدالة `fopen` ستُنشئ ملفًا إذا أردت كتابة البيانات على ملف، أو أنها ستتخلص من محتويات الملف إذا وُجد ليصبح طوله صفر (أي أنك ستخسر محتويات الملف السابقة).

يوضح الجدول التالي جميع الأنماط الموجودة في المعيار، إلا أن التنفيذ قد يسمح بأنماط أخرى بإضافة محارف إضافية في نهاية كل من الأنماط.

الجدول 21: أنماط فتح الملف

النمط	نوع الملف	القراءة	الكتابة	إنشاء جديد	حذف القيمة السابقة
"r"	نصي	نعم	لا	لا	لا
"rb"	ثنائي	نعم	لا	لا	لا
"r+"	نصي	نعم	نعم	لا	لا
"r+b"	ثنائي	نعم	نعم	لا	لا
"rb+"	ثنائي	نعم	نعم	لا	لا
"w"	نصي	لا	نعم	نعم	نعم
"wb"	ثنائي	لا	نعم	نعم	نعم
"w+"	نصي	نعم	نعم	نعم	نعم
"w+b"	ثنائي	نعم	نعم	نعم	نعم
"wb+"	ثنائي	نعم	نعم	نعم	نعم
"a"	نصي	لا	نعم	نعم	لا
"ab"	ثنائي	لا	نعم	نعم	لا
"a+"	نصي	نعم	نعم	نعم	لا
"a+b"	ثنائي	لا	نعم	نعم	لا
"ab+"	ثنائي	لا	نعم	نعم	لا

انتبه من بعض التنفيذات التي تضيف إلى النمط الأخير محارف NULL في حالة الملفات الثنائية، إذ قد يتسبب فتح هذه الملفات بالنمط ab أو ab+ أو a+b بوضع مؤشر الملف خارج نطاق آخر البيانات المكتوبة.

تُكتب جميع البيانات في نهاية الملف إذا فُتح باستخدام نمط الإضافة append، بغض النظر عن محاولة تغيير موضع المؤشر باستخدام الدالة fseek، ويكون موضع مؤشر الملف المبدئي معرف بحسب التنفيذ.

تفشل محاولات فتح الملف بنمط القراءة (النمط 'r')، إذا لم يكن الملف موجوداً أو لم يمكن قراءته.

يمكن القراءة من والكتابة إلى الملفات المفتوحة بنمط التحديث update (باستخدام '+' مثل المحرف الثاني أو الثالث ضمن النمط) إلا أنه من غير الممكن إلحاق القراءة بالكتابة مباشرةً أو الكتابة بالقراءة دون استدعاء بينهما لدالة واحدة (أو أكثر) من الدوال: fflush أو fseek أو fsetpos أو rewind، والاستثناء الوحيد هنا هو جواز إلحاق الكتابة مباشرةً بعد القراءة إذا قرأ المحرف EOF (نهاية الملف).

من الممكن أيضاً في بعض التنفيذات أن يُتخلى عن b في أنماط فتح الملفات الثنائية واستخدام الأنماط ذاتها الخاصة بالملفات النصية.

تُخزّن المجاري المفتوحة باستخدام `fopen` تخزينًا مؤقتًا بالكامل إذا لم تكن متصلة إلى جهاز تفاعلي، ويضمن ذلك التعامل مع الأسئلة `prompts` والطلبات `responses` على النحو الصحيح.

تعيد الدالة `fopen` مؤشرًا فارغًا `null` إذا فشلت بفتح الملف، وإلا فتعيد مؤشرًا يشير إلى الكائن الذي يتحكم بالمجرى. كائنات المجاري `stdin` و `stdout` و `stderr` غير قابلة للتعديل بالضرورة ومن الممكن عدم وجود إمكانية استخدام القيمة المُعادَة من الدالة `fopen` لإسنادها إلى واحدة من هذه الكائنات، بدلاً من ذلك نستخدم `freopen` لهذا الغرض.

9.10.6 الدالة `freopen`

تُستخدم الدالة `freopen` لأخذ مؤشر يشير إلى مجرى وربطه مع اسم ملف آخر، وتصرّح الدالة على النحو التالي:

```
#include <stdio.h>
FILE *freopen(const char *pathname,
              const char *mode, FILE *stream);
```

وسيط `mode` مشابه لمثيله في دالة `fopen`. يُغلق المجرى `stream` أولاً ويحدث تجاهل أي أخطاء متولدة عن ذلك، ونحصل على قيمة `NULL` في حالة حدوث خطأ عند تنفيذ الدالة، وإلا فإننا نحصل على القيمة الجديدة للمجرى `stream`.

9.10.7 إغلاق الملفات

يمكننا إغلاق ملف مفتوح باستخدام الدالة `close` والمصرح عنها كما يلي:

```
#include <stdio.h>
int fclose(FILE *stream);
```

يُتخلّص من أي بيانات موجودة على الذاكرة المؤقتة لم تُكتب على الملف الخاص بالمجرى `stream` إضافةً إلى أي بيانات أخرى لم تُقرأ، وتُحرّر الذاكرة المؤقتة المرتبطة بالمجرى إذا رُبطت به تلقائيًا، وأخيرًا يُغلق الملف. نحصل على القيمة صفر للدلالة على نجاح العملية، وإلا فالقيمة `EOF` للدلالة على الخطأ.

9.10.8 الدالتان `setbuf` و `setvbuf`

تُستخدم الدالتان للتعديل على استراتيجية التخزين المؤقتة لمجرى معين مفتوح، ويُصرّح عن الدالتين كما يلي:

```
#include <stdio.h>

int setvbuf(FILE *stream, char *buf,
            int type, size_t size);
void setbuf(FILE *stream, char *buf);
```

يجب استخدام الدالتين **قبل** قراءة الملف أو الكتابة إليه، ويعرف الوسيط `type` نوع التخزين المؤقت للمجرى `stream`، ويوضح الجدول التالي أنواع التخزين المؤقت.

الجدول 22: أنواع التخزين المؤقت

التأثير	القيمة
لا تخزن الدخل والخرج مؤقتًا	<code>_IONBF</code>
خزن الدخل والخرج مؤقتًا	<code>_IOFBF</code>
تخزين مؤقت خطي: تخلص من محتويات الذاكرة المؤقتة عندما تمتلئ، أو عند كتابة سطر جديد، أو عند طلب القراءة	<code>_IOLBF</code>

يمكن للوسيط `buf` أن يكون مؤشرًا فارغًا، وفي هذه الحالة تنشأ مصفوفة تلقائيًا لتخزين البيانات مؤقتًا، ويمكن بخلاف ذلك للمستخدم توفير ذاكرة مؤقتة لكن يجب التأكد من استمرارية الذاكرة المؤقتة بقدر مساوٍ (أو أكثر) لاستمرارية التدفق `stream`. يُعد استخدام مساحة التخزين المحجوزة تلقائيًا ضمن تعليمة مركبة `compound statement` من الأخطاء الشائعة، إذ أن الحصول على المساحة التخزينية على النحو الصحيح في هذه الحالة يجري عن طريق الدالة `malloc` عوضًا عن ذلك. يُحدد حجم الذاكرة المؤقتة باستخدام الوسيط `size`. يشابه استدعاء الدالة `setbuf` استدعاء الدالة `setvbuf` إذا استخدمنا `_IOFBF` قيمةً للوسيط `type` والقيمة `BUFSIZ` للوسيط `size`، وتُستخدم القيمة `_IONBF` للوسيط `type` إذا كان `buf` مؤشرًا فارغًا.

لا تُعاد أي قيمة بواسطة الدالة `setbuf`، بينما تُعيد الدالة `setvbuf` القيمة صفر للدلالة على نجاح الاستدعاء، وإلا فقيمة غير صفرية إذا كانت قيم `type`، أو `size` غير صالحة، أو كان الطلب غير ممكن التنفيذ.

9.10.9 دالة `fflush`

يُصرّح عن الدالة `fflush` كما يلي:

```
#include <stdio.h>

int fflush(FILE *stream);
```

إذا أشار المجرى stream إلى ملف مفتوح للخروج أو بنمط التحديث، وكان هناك أي بيانات غير مكتوبة فإنها تُكتب خارجًا، وهذا يعني أنه لا يمكن لدالة داخل بيئة مستضافة hosted environment، أو ضمن لغة سي أن تضمن -على سبيل المثال- أن البيانات تصل مباشرةً إلى سطح قرص يدعم الملف. تُهمل أي عملية ungetc سابقة إذا كان المجرى مرتبطًا بالملف المفتوح بهدف الخروج أو التحديث.

يجب أن تكون آخر عملية على المجرى عملية خرج، وإلا فسنحصل على سلوك غير معرّف.

يتخلص استدعاء fflush الذي يحتوي على وسيط قيمته صفر من جميع مجاري الدخل والخروج، ويجب هنا الانتباه إلى المجاري التي لم تكن عملياتها الأخيرة عملية خرج، أي تفادي حصول السلوك غير المعرّف الذي ذكرناه سابقًا.

تُعيد الدالة القيمة EOF للدلالة على الخطأ، وإلا فالقيمة صفر للدلالة على النجاح.

9.11 الدخل والخروج المنسق Formatted I/O

هناك عدد من الدوال المُستخدمة لتنسيق الدخل والخروج، وتحدد كلاً من هذه الدوال التنسيق المتبع للدخل والخروج باستخدام **سلسلة التنسيق النصية format string**، وتتألف سلسلة التنسيق النصية في حالة الخروج من نص عادي plain text يمثل الخروج كما هو، إضافةً إلى **مواصفات التنسيق format specifications** التي تتطلب معالجة خاصة لواحد من الوسائط المتبقية في الدالة، بينما تتألف سلسلة التنسيق النصية في حالة الخروج من نص عادي يطابق مجرى الدخل وتحدد هنا مواصفات التنسيق معنى الوسائط المتبقية.

يُشار إلى كل واحدة من مواصفات التنسيق باستخدام المحرف "%" متبوعًا ببقية التوصيف.

9.11.1 الخرج: دوال printf

تتخذ مواصفات التنسيق في دوال الخرج الشكل التالي، ونشير إلى الأجزاء الاختيارية بوضعها بين قوسين:

```
%<flags><field width><precision><length>conversion
```

نشرح معنى كل من الراية flag وعرض الحقل field width والدقة precision والطول length والتحويل conversion أدناه، إلا أنه من الأفضل النظر إلى وصف المعيار إذا أردت وصفًا مطولاً ودقيقًا.

9.11.2 الرايات

يمكن ألا تأخذ الرايات أي قيمة أو أن تأخذ أحد القيم التالية:

قيمة الراية	الشرح
-	ملئ سطر التحويل من اليسار ضمن حقوله.
+	يبدأ التحويل ذو الإشارة بإشارة زائد أو ناقص دائماً.
مسافة فارغة space	إذا كان المحرف الأول من تحويل ذو إشارة ليس بإشارة، أضف مسافةً فارغةً، ويمكن تجاوز الراية باستخدام "+" إذا وجدت.
0	يُجبر استخدام تنسيق مختلف للخروج، مثل: الخانة الأولى لأي تحويل ثماني لها القيمة "0"، وإضافة "0x" أمام أي تحويل ست عشري لا تساوي قيمته الصفر، ويُجبر الفاصلة العشرية في جميع تحويلات الفاصلة العائمة حتى إن لم تكن ضرورية، ولا يُزيل أي صفر لاحق من تحويلات g و G.
عرض الحقل field width	يُضيف إلى تحويلات d و i و o و u و X و x و E و e و f و F و G أصفاً إلى يسارها بملء عرض الحقل، ويمكن تجاوزه باستخدام الراية "-", وتُتجاهل الراية إذا كان هناك أي دقة محددة لتحويلات d، أو i، أو o، أو u، أو x، أو X، ونحصل على سلوك غير معرف للتعريفات الأخرى.
الدقة precision	عدد صحيح عشري يحدد عرض حقل الخروج الأدنى ويمكن تجاوزه إن لزم الأمر، يُحوّل الوسيط التالي إلى عدد صحيح ويُستخدم مثل قيمة لعرض الحقل إن استُخدمت علامة النجمة "*", وتُعامل هذه القيمة إذا كانت سالبة كأنها راية "-". متبوعة بعرض حقل ذي قيمة موجبة. يُملأ الخرج ذو الطول الأقصر من عرض الحقل بالمسافات الفارغة (أو بأصفار إذا كان العدد الصحيح المعبر عن عرض الحقل يبدأ بالصفري)، ويُملأ الخرج من الجهة اليسرى إلا إذا حُدّت راية تعديل اليسار left-adjustment.
الطول length	تبدأ قيمة الدقة بالنقطة '.', وهي تحدد عدد الخانات الدنيا لتحويلات d، أو i، أو o، أو u، أو x، أو X، أو عدد الخانات التي تلي الفاصلة العشرية في تحويلات e، أو E، أو f، أو العدد الأعظمي لخانات تحويلات g و G، أو عدد المحارف المطبوعة من سلسلة نصية في تحويلات s. يتسبب تحديد كمية حشو الحقل padding بتجاهل قيمة field width. يُحوّل الوسيط التالي في حال استخدامنا لعلامة النجمة "*" إلى عدد صحيح ويُستخدم بمثابة قيمة لعرض الحقل، وتعامل القيمة كأنها مفقودة إذا كانت سالبة، وتكون الدقة صفر إذا وجدت النقطة فقط.
	وهي h تسبق محدد specifier لطباعة نوع عدد صحيح integral ويتسبب ذلك في معاملتها وكأنها من النوع "short" (لاحظ أن الأنواع المختلفة القصيرة shorts تُرقى إلى واحدة من أنواع القيم الصحيحة int عندما تُمرّر مثل وسيط). تعمل l مثل عمل h إلا أنها تُطبّق على وسيط عدد صحيح من نوع "long"، وتُستخدم L للدلالة على أنه يجب طباعة وسيط من نوع "long double"، ويطبّق ذلك فقط على محددات الفاصلة العائمة. يتسبب استخدام هذا في سلوك غير معرف إذا كانت باستخدام النوع الخاطئ من التحويلات.

يوضح الجدول التالي أنواع التحويلات:

الجدول 23: التحويلات

المحدد	التأثير	الدقة الافتراضية
d	عدد عشري ذو إشارة	1
i	عدد عشري ذو إشارة	1
u	عدد عشري عديم الإشارة	1
o	عدد ثماني عديم الإشارة	1
x	عدد ست عشري عديم الإشارة من 0 إلى f	1
X	عدد ست عشري عديم الإشارة من 0 إلى F	1
	تحدد الدقة عدد خانات الأدنى المُستبدل بأصفار إن لزم الأمر، ونحصل على خرج دون أي محارف عند استخدام الدقة صفر لطباعة القيمة صفر	
f	يطبع قيمة من النوع double بعدد خانات الدقة (المقربة) بعد الفاصلة العشرية. استخدم دقة بقيمة صفر للحد من الفاصلة العشرية، وإلا فستظهر خانة واحدة على الأقل بعد الفاصلة العشرية	6
e, E	يطبع قيمة من نوع double بالتنسيق الأسّي مُقَرَّبًا بخانة واحدة قبل الفاصلة العشرية، وعدد من الخانات يبلغ الدقة المحددة بعده، وتُلغى الفاصلة العشرية عند استخدام الدقة صفر، وللأس خانتان على الأقل تطبع بالشكل 1.23e15 في تنسيق e أو 1.23E15 في حالة التنسيق E	6
g, G	تستخدم أسلوب التنسيق f، أو e مع E- بحسب الأس، ولا يُستخدم التنسيق f إذا كان الأس أصغر من "-4" أو أكبر أو يساوي الدقة. تُحدّد الأصفار التي تتبع القيمة وتُطبع الفاصلة العشرية فقط في حال وجود خانات تابعة.	غير محدد
c	يُحوّل الوسيط من نوع عدد صحيح إلى محرف عديم الإشارة ويُطبع المحرف الناتج عن التحويل	
s	تُطبع سلسلة نصية بطول خانات الدقة، ويجب إنهاء السلسلة النصية باستخدام NUL إذا لم تُحدّد الدقة أو كانت أكبر من طول السلسلة النصية	لا نهائي
p	إظهار قيمة مؤشر من نوع (void *) بطريقة تعتمد على النظام	
n	يجب أن يكون الوسيط مؤشرًا يشير إلى عدد صحيح، ويكون عدد محارف الخرج باستخدام هذا الاستدعاء مُسنَدًا إلى العدد الصحيح	
%	علامة "%"	-

تجد وصف الدوال التي تستخدم هذه التنسيقات في الجدول التالي، وجميع الدوال المُستخدمة مُضمّنة في المكتبة `<stdio.h>`، إليك تصاريح هذه الدوال:

```
#include <stdio.h>

int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);

#include <stdarg.h> // بالإضافة إلى stdio.h
int fprintf(FILE *stream, const char *format, va list arg);
int vprintf(const char *format, va list arg);
int vsprintf(char *s, const char *format, va list arg);
```

الجدول 24: الدوال التي تطبع خرجًا مُنسقًا

الاسم	الاستخدام
fprintf	نحصل على الخرج المنسق العام بواسطتها كما وصفنا سابقًا، ويكتب الخرج إلى الملف المُحدد باستخدام المجرى <code>stream</code> .
printf	دالة مُطابقة لعمل الدالة <code>fprintf</code> إلا أن وسيطها الأول هو <code>stdout</code> .
sprintf	دالة مُطابقة لعمل الدالة <code>fprintf</code> باستثناء أن خرجها لا يُكتب إلى ملف، بل يُكتب إلى مصفوفة محارف يُشار إليها باستخدام المؤشر <code>s</code> .
fprintf	خرج مُنسق مشابه لخرج الدالة <code>fprintf</code> إلا أن لائحة الوسيط المتغيرة تُستبدل بالوسيط <code>arg</code> الذي يجب أن يُهيأ باستخدام <code>va_start</code> ، ولا تُستدعى <code>va_end</code> باستخدام هذه الدالة.
vprintf	مطابقة للدالة <code>vfprintf</code> باستثناء أن الوسيط الأول يساوي إلى <code>stdout</code> .
vsprintf	خرج مُنسق مشابه لخرج الدالة <code>sprintf</code> إلا أن لائحة الوسيط المتغيرة تُستبدل بالوسيط <code>arg</code> الذي يجب أن يُهيأ باستخدام <code>va_start</code> ، ولا تُستدعى <code>va_end</code> باستخدام هذه الدالة.

تُعيد كل الدوال السابقة عدد المحارف المطبوعة أو قيمة سالبة للدلالة على حصول خطأ، ولا يُحسب المحرف الفارغ الزائد بواسطة دالة `sprintf` و `vsprintf`.

يجب أن تسمح التنفيذات بالحصول على 509 محارف على الأقل عند استخدام أي تحويل.

9.11.3 الدخول: دوال scanf

هناك عدة دوال مشابهة لمجموعة دوال printf بهدف الحصول على الدخول، والفارق الواضح بين مجموعتي الدوال هذه هو أن مجموعة دوال scanf تأخذ وسيطاً متمثلاً بمؤشر حتى تُسند القيم المقروءة إلى وجهتها المناسبة، ويُعد نسيان تمرير المؤشر خطأ شائع الحدوث ولا يمكن للمصنّف أن يكتشف حدوثه، إذ يمنع استخدام لائحة وسطاء متغيرة من ذلك.

تُستخدم سلسلة التنسيق النصية للتحكم بتفسير المجري للبيانات المُدخلة التي تحتوي غالباً على قيم تُسند إلى كائنات يُشار إليها باستخدام وسطاء دالة scanf المتبقية، وقد تتألف سلسلة التنسيق النصية من:

- مساحة فارغة white space: تتسبب بقراءة مجرى الدخول إلى المحرف التالي الذي لا يمثّل محرف مسافة فارغة.

- محرف اعتيادي ordinary character: ويمثل المحرف أي محرف عدا محارف السلسلة الفارغة أو "%"، ويجب أن يطابق المحرف التالي في مجرى الدخول هذا المحرف المُحدّد.

- توصيف التحويل conversion specification: وهو محرف "%" متبوع بمحرف "*" اختياري (الذي يكبح التحويل)، ويُتبع بعدد عشري صحيح لا يساوي الصفر يحدد عرض الحقل الأعظمي، ومحرف "h"، أو "l"، أو "L" اختياري للتحكم بطول التحويل، وأخيراً محدد تحويل إجباري. لاحظ أن استخدام "h"، أو "l"، أو "L" سيؤثر على نوع المؤشر الواجب استخدامه.

حقل الدخول -باستثناء المحددات "c" و "n" و "]" - هو سلسلة من المحارف التي لا تمثل مسافة فارغة وتبدأ من أول محرف في الدخول (بشرط ألا يكون المحرف مسافة فارغة)، وتُنتهي السلسلة عند أول محرف متعارض أو عند الوصول إلى عرض الحقل المُحدّد.

تُسند النتيجة إلى الشيء الذي يُشير إليه الوسيط إلا إذا كان الإسناد مكبوّحاً باستخدام "*" المذكورة سابقاً، ويمكن استخدام محددات التحويل التالية:

- المحددات d i o u x: تُحوّل d عدد صحيح ذو إشارة، وتحوّل i عدد صحيح ذو إشارة وتنسيق ملائم strtold، وتحوّل o عدد صحيح ثماني، وتحوّل u عدد صحيح عديم الإشارة، وتحوّل x عدد صحيح ست عشري.

- المحددات e f g: تحوّل قيمة من نوع float (وليس double).

- المحدد s: يقرأ سلسلة نصية ويُضيف محرف فارغ في نهايته، وتُنتهي السلسلة النصية باستخدام مسافة فارغة عند الدخول (ولا تُقرأ هذه المسافة الفارغة على أنها جزء من الدخول).

- المحدد [: يقرأ سلسلة نصية، وتتبع] لائحة من المحارف تُدعى **مجموعة المسح scan set**، ويُنتهي المحرف] هذه اللائحة. تُقرأ المحارف إلى (غير متضمنةً) المحرف الأول غير الموجود ضمن مجموعة

المسح؛ فإذا كان المحرف الأول في اللائحة هو "^" فهذا يعني أن مجموعة القراءة تحتوي على أي محرف غير موجود في هذه القائمة، وإذا كانت السلسلة الأولية هي "[^]" أو "[]" فهذا يعني أن [] ليس محدّدًا بل جزءًا من السلسلة ويجب إضافة محرف "]" آخر لإنهاء اللائحة. إذا وجدت علامة ناقص "-" في اللائحة، يجب أن يكون موقعها المحرف الأول أو الأخير، وإلا فإن معناها معرف بحسب التنفيذ.

- المحدد c: يقرأ محرفًا واحدًا متضمنًا محارف المسافات الفارغة، ولقراءة المحرف الأول باستثناء محارف المسافات الفارغة استخدم %s، ويحدد عرض الحقل مصفوفة المحارف التي يجب قراءتها.
- المحدد p: يقرأ مؤشرًا من النوع (void *) والمكتوب سابقًا باستخدام المحدد %p ضمن استدعاء سابق لمجموعة دوال printf.
- المحدد %: المحرف "%" متوقّع في الدخل ولا يُجرى أي إسناد.
- المحدد n: يُعاد عددًا صحيح يمثل عدد المحارف المقروءة باستخدام هذا الاستدعاء.

يوضح الجدول التالي تأثير محددات الحجم size specifiers:

الجدول 25: محددات الحجم

المحدد	يُحدّد	يُحوّل
l	d i o u x	عدد صحيح كبير long int
h	d i o u x	عدد صحيح صغير short int
l	e f	عدد عشري مضاعف double
L	e f	عدد عشري مضاعف كبير long double

إليك وصف دوال مجموعة scanf مع تصاريحها:

```
#include <stdio.h>

int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
int scanf(const char *format, ...);
```

تأخذ الدالة fscanf دخلها من المجرى المُحدد، وتُطابق الدالة scanf الدالة fscanf مع اختلاف أن الوسيط الأول هو المجرى stdin، بينما تأخذ sscanf دخلها من مصفوفة محارف مُحدّدة.

نحصل على القيمة EOF المُعادة في حال حدوث خطأ دخل قبل أي تحويلات، وإلا فنحصل على عدد التحويلات الناجحة الحاصلة وقد يكون هذا العدد صفر إن لم تُجرى أي تحويلات، ونحصل على خطأ دخل إذا قرأنا

EOF، أو بوصولنا إلى نهاية سلسلة الدخل النصية، بينما نحصل على خطأ تحويل إذا فشل العثور على نمط مناسب يوافق التحويل المحدد.

9.12 عمليات الإدخال والإخراج على المحارف

هناك عدد من الدوال التي تسمح لنا بإجراء عمليات الدخل والخروج على المحارف بصورة منفردة، إليك تصاريحها:

```
#include <stdio.h>
/* دخل المحرف */
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
/* خرج المحرف */
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);

/* دخل السلسلة النصية */
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);

/* خرج السلسلة النصية */
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

لنستعرض سوياً كلاً منها.

9.12.1 دخل المحرف

تقرأ مجموعة الدوال التي تنفذ هذه المهمة المحرف مثل قيمة من نوع "unsigned char" من مجرى الدخل المحدد أو من stdin، ونحصل على المحرف الذي يليه في كل حالة من مجرى الدخل. يُعامل المحرف مثل قيمة "unsigned char" ويحوّل إلى "int" وهي القيمة المُعاداة من الدالة. نحصل على الثابت EOF عند الوصول إلى نهاية الملف، ويُضبط مؤشر نهاية الملف end-of-file indicator إلى المجرى المحدد، كما نحصل على EOF في حالة الخطأ ويُضبط مؤشر الخطأ إلى المجرى المحدد. نستطيع الحصول على المحارف بصورة

تتابعية باستدعاء الدالة تبعاً. قد نحصل على وسيط المجرى stream أكثر من مرة في حال استخدام هذه الدوال على أنها ماكرو، لذا لا تستخدم الآثار الجانبية هنا.

هناك برنامج "ungetc" الداعم أيضاً، الذي يُستخدم لإعادة محرف إلى المجرى مما يجعله المحرف التالي الذي سيُقرأ، إلا أن هذه ليست بعملية خرج ولن تتسبب بتغيير محتوى الملف، ولذا تتسبب عمليات fflush و fseek و rewind على المجرى بين عملية إعادة المحرف وقراءته بتجاهل هذا المحرف، ويمكن إعادة محرف واحد فقط وأي محاولات لإعادة EOF تُتجاهل، ولا يُعدّل على موضع مؤشر الملف في جميع حالات إعادة قراءة مجموعة من المحارف وإعادة قراءتها أو تجاهلها. يتسبب استدعاء ungetc الناجح على مجرى ثنائي بتناقص موضع مؤشر الملف إلا أن ذلك غير محدد عند استخدام مجرى نصي، أو مجرى ثنائي موجود في بداية الملف.

9.12.2 خرج المحرف

هذه الدوال مطابقة لدوال الدخل الموصوفة سابقاً إلا أنها تجري عمليات الخرج، إذ تعيد المحرف المكتوب أو EOF عند حدوث خطأ ما، ولا يوجد ما يعادل نهاية الملف End Of File في ملف الخرج.

9.12.3 خرج السلسلة النصية

تكتب هذه الدوال سلاسل نصية إلى ملف الخرج باستخدام المجرى stream إن دُكر وإلا فإلى المجرى stdout، ولا يُكتب محرف الإنهاء الفارغ. نحصل على قيمة لا تساوي الصفر عند حدوث خطأ وإلا فالقيمة صفر.

تضيف puts سطرًا جديدًا إلى سلسلة الخرج النصية بينما لا تفعل fputs ذلك.

9.12.4 دخل السلسلة النصية

تقرأ الدالة fgets السلسلة النصية إلى مصفوفة يُشار إليها باستخدام المؤشر s من المجرى stream، وتتوقف عن القراءة في حال الوصول إلى EOF أو عند أول سطر جديد (وتقرأ محرف السطر الجديد)، وتضيف محرفًا فارغًا null في النهاية. يُقرأ n-1 محرف على الأكثر (لترك حيز للمحرف الفارغ).

تعمل الدالة gets على نحوٍ مشابه لمجرى stdin إلا أنها تتجاهل محرف السطر الجديد.

تعيد كلا الدالتين s في حال نجاحهما وإلا فمؤشر فارغ، إذ نحصل على مؤشر فارغ عندما نصادف EOF قبل قراءة أي محرف ولا يطرأ أي تغيير على المصفوفة، بينما تصبح محتويات المصفوفة غير معرفة إذا ما واجهنا خطأ قراءة وسط السلسلة النصية بالإضافة إلى إعادة مؤشر فارغ.

9.13 الدخل والخرج غير المنسق Unformatted I/O

هذا الجزء بسيط، إذا هناك فقط دالتان تقدمان هذه الخاصية، واحدة منهما للقراءة والأخرى للكتابة ويصرّح عنهما على النحو التالي:

```
#include <stdio.h>
```

```
size_t fread(void *ptr, size_t size, size_t nelem, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nelem, FILE
*stream);
```

تُجرى عملية القراءة أو الكتابة المناسبة للبيانات المُشار إليها بواسطة المؤشر `ptr` وذلك على `nelem` عنصر، وبحجم `size`، ويفشل نقل ذلك المقدار الكامل من العناصر فقط عند الكتابة، إذ قد تعيق نهاية الملف دخل العناصر بأكملها، وتُعيد الدالة عدد العناصر التي نُقلت فعلياً. نستخدم `feof` أو `ferror` للتمييز بين نهاية الملف عند الدخل أو للإشارة على خطأ.

تُعيد الدالة القيمة صفر دون أي فعل إذا كانت قيمة `size` أو `nelem` تساوي إلى الصفر.

قد يساعدنا المثال الآتي في توضيح عمل الدالتين المذكورتين:

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
struct xx{
```

```
    int xx_int;
```

```
    float xx_float;
```

```
}ar[20];
```

```
main(){
```

```
    FILE *fp = fopen("testfile", "w");
```

```
    if(fwrite((const void *)ar,
        sizeof(ar[0]), 5, fp) != 5){
```

```
        fprintf(stderr, "Error writing\n");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    rewind(fp);
```

```
    if(fread((void *)&ar[10],
```

```
        sizeof(ar[0]), 5, fp) != 5){
```

```

        if(ferror(fp)){
            fprintf(stderr,"Error reading\n");
            exit(EXIT_FAILURE);
        }
        if(feof(fp)){
            fprintf(stderr,"End of File\n");
            exit(EXIT_FAILURE);
        }
    }
    exit(EXIT_SUCCESS);
}

```

[مثال 7]

9.14 الدوال عشوائية الوصول Random access functions

تعمل جميع دوال دخل وخرج الملفات بصورةٍ مشابهة بين بعضها، إذ أن الملفات ستُقرأ أو يُكتب إليها بصورةٍ متتابعةٍ إلا إذا اتخذ المستخدم خطوات مقصودة لتغيير موضع مؤشر الملف. ستتسبب عملية قراءة متبوعة بكتابة متبوعة بقراءة ببدء عملية القراءة الثانية بعد نهاية عملية كتابة البيانات فورًا، وذلك بفرض أن الملف مفتوح باستخدام نمط يسمح بهذا النوع من العمليات، كما يجب أن تتذكر أن المجرى `stdio` يُصّرّ على إدخال المستخدم لعملية تحرير ذاكرة مؤقتة بين كل عنصر من عناصر دورة قراءة-كتابة-قراءة، وللتحكم بذلك، تسمح دالة الوصول العشوائي `random access function` بالتحكم بموضع الكتابة والقراءة ضمن الملف، إذ يُحرّك موضع مؤشر الملف دون الحاجة لقراءة أو كتابة ويشير إلى البايت الذي سيخضع لعملية القراءة أو الكتابة التالية.

هناك ثلاثة أنواع من الدوال التي تسمح بفحص موضع مؤشر الملف أو تغييره، إليك تصاريح كل منهم:

```

#include <stdio.h>

/* إعادة موضع مؤشر الملف */
long ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);

/* ضبط موضع مؤشر الملف إلى الصفر */
void rewind(FILE *stream);

```

```

/* ضبط موضع مؤشر الملف */
int fseek(FILE *stream, long offset, int ptrname);
int fsetpos(FILE *stream, const fpos_t *pos);

```

تُعيد الدالة `ftell` القيمة الحالية لموضع مؤشر الملف (المُقاسة بعدد المحارف)، إذا كان المجري `stream` يشير إلى ملف ثنائي، وإلا فإنها تعيد رقمًا مميزًا في حالة الملف النصي، ويمكن استخدام هذه القيمة فقط عند استدعاءات لاحقة لدالة `fseek` لإعادة ضبط موضع مؤشر الملف الحالية. نحصل على القيمة `-1L` في حالة الخطأ ويُضبط `errno`.

تضبط الدالة `rewind` موضع مؤشر الملف الحالي إلى بداية الملف المُشار إليه بالمجري `stream`، ويُعاد ضبط مؤشر خطأ الملف باستدعاء الدالة `rewind` ولا تُعيد الدالة أي قيمة.

تسمح الدالة `fseek` لموضع مؤشر الملف ضمن المجري أن يُضبط لقيمة عشوائية (للملفات الثنائية)، أو إلى الموضع الذي نحصل عليه من `ftell` فقط بالنسبة للملفات النصية، وتتبع الدالة القوانين التالية:

- يُضبط موضع مؤشر الملف في الحالة الاعتيادية بفارق معين من البايتات (المحارف) عن نقطة الملف المُحددة بالقيمة `ptrname`، وقد يكون الفارق `offset` سالبًا. قد يأخذ `ptrname` القيمة `SEEK_SET` التي تضبط موضع مؤشر الملف نسبيًا إلى بداية الملف، أو القيمة `SEEK_CUR` التي تضبط موضع مؤشر الملف نسبيًا إلى قيمتها الحالية، أو القيمة `SEEK_END` التي تضبط موضع مؤشر الملف نسبيًا إلى نهاية الملف، إلا أنه من غير المضمون أن تعمل القيمة الأخيرة بنجاح في المجاري الثنائية.
- يجب أن تكون قيمة `offset` في الملفات النصية إما صفر أو قيمة مُعادة بواسطة استدعاء سابق للدالة `ftell` على المجري ذاته، ويجب أن تكون قيمة `ptrname` مساويةً إلى `SEEK_SET`.
- يُفرغ `fseek` مؤشر نهاية الملف للمجري المُحدد ويحذف بيانات أي استدعاء لعملية `ungetc`، ويعمل ذلك لكلٍّ من الدخل والخرج.
- تُعاد القيمة صفر للدلالة على النجاح وأي قيمة غير صفرية تدل على طلب ممنوع للدالة.

لاحظ أنه يمكن لكلٍ من `ftell` و `fseek` ترميز قيمة موضع مؤشر الملف إلى قيمة من نوع `long`، وقد لا يحدث هذا بنجاح في حالة استخدامه على الملفات الطويلة جدًا؛ لذلك، يقدم المعيار كلاً من `fgetpos` و `fsetpos` للتغلب على هذه المشكلة.

تخزّن الدالة `fgetpos` موضع مؤشر الملف الحالي ضمن المجري للكائن المُشار إليه باستخدام المؤشر `pos`، والقيمة المخزنة هي قيمة مميزة تُستخدم فقط للعودة إلى الموضع المحدد ضمن المجري ذاته باستخدام الدالة `fsetpos`. تعمل الدالة `fsetpos` كما وضحنا سابقًا، كما أنها تُفرغ مؤشر نهاية الملف للمجري وتُزيل أي تأثير لعمليات `ungetc` سابقة.

نحصل على القيمة صفر في حالة النجاح لكلا الدالتين، بينما نحصل على قيمة غير صفرية في حالة الخطأ
وَيُضَبَط `.errno`.

9.14.1 التعامل مع الأخطاء

تحافظ دوال الدخل والخرج القياسية على مؤشرين لكل مجرى مفتوح للدلالة على نهاية الملف وحالة الخطأ
ضمنه، ويمكن الحصول على قيم هذه المؤشرات وضبطها عن طريق الدوال التالية:

```
#include <stdio.h>

void clearerr(FILE *stream);

int feof(FILE *stream);

int ferror(FILE *stream);

void perror(const char *s);
```

- تُفَرِّغ الدالة `clearerr` كلاً من مؤشري الخطأ ونهاية الملف EOF للمجری `stream`.
- تُعيد الدالة `feof` قيمةً غير صفرية إذا كان لمؤشر نهاية الملف الخاص بالمجری `stream` قيمة، وإلا فإنها تعيد القيمة صفر.
- تُعيد الدالة `ferror` قيمة غير صفرية إذا كان لمؤشر الخطأ الخاص بالمجری `stream` قيمة، وإلا فإنها تعيد القيمة صفر.
- تطبع الدالة `perror` سطرًا واحدًا يحتوي على رسالة خطأ على خرج البرنامج القياسي مسبقًا بالسلسلة النصية المُشار إليها بواسطة المؤشر `s` مع إضافة مسافة فارغة ونقطتين ":". تُحدد رسالة الخطأ بحسب قيمة `errno` وتُعطي شرحًا بسيطًا عن سبب الخطأ، على سبيل المثال يتسبب البرنامج التالي برسالة خطأ:

```
#include <stdio.h>
#include <stdlib.h>

main(){

    fclose(stdout);
    if(fgetc(stdout) >= 0){
```

```

        fprintf(stderr, "What - no error!\n");
        exit(EXIT_FAILURE);
    }
    perror("fgetc");
    exit(EXIT_SUCCESS);
}

/* رسالة الخطأ */
fgetc: Bad file number

```

[مثال 8]

لم نُقل أن الرسالة التي سنحصل عليها ستكون واضحة.

9.15 أدوات مكتبة `stdlib`

نستعرض هنا الأدوات الموجودة في ملف الترويسة `<stdlib.h>` الذي يصرح عن عدد من الأنواع والماكرو وعدة دوال للاستخدام العام، تتضمن الأنواع والماكرو التالي:

- النوع `size_t`: تكلمنا عنه سابقًا.
- النوع `div_t`: نوع من الهياكل التي تعيدها الدالة `div`.
- النوع `ldiv_t`: نوع من الهياكل التي تعيدها الدالة `ldiv`.
- النوع `NULL`: تكلمنا عنه سابقًا.
- القيمة `EXIT_FAILURE` و `EXIT_SUCCESS`: يمكن استخدامهما مثل وسيط للدالة `exit`.
- القيمة `MB_CUR_MAX`: العدد الأعظمي للبايتات في محرف متعدد البايتات `multibyte character` من مجموعة المحارف الإضافية والمحددة حسب إعدادات اللغة المحلية `locale`.
- القيمة `RAND_MAX`: القيمة العظمى المُعادة من استدعاء دالة `rand`.

9.15.1 دوال تحويل السلسلة النصية

هناك ثلاث دوال تقبل سلسلة نصية وسيطًا لها وتحولها إلى عدد من نوع معين كما هو موضح هنا:

```

#include <stdlib.h>

double atof(const char *nptr);

```

```
long atol(const char *nptr);
int atoi(const char *nptr);
```

نحصل على عدد مُحوّل مُعاد لكل من الدوال الثلاث السابقة، ولا تضمن لك أيّ من الدوال أن تضبط القيمة `errno` (إلا أن الأمر محقق في بعض التنفيذات)، وتكون النتائج التي نحصل عليها من تحويلات تتسبب بحدوث طفحان `overflow` ولا يمكننا تمثيلها غير معرّفة.

هناك بعض الدوال أكثر تعقيدًا:

```
#include <stdlib.h>

double strtod(const char *nptr, char **endptr);
long strtol(const char *nptr, char **endptr, int base);
unsigned long strtoul(const char *nptr, char **endptr, int base);
```

تعمل الدوال الثلاث السابقة بطريقة مشابهة، إذ يجري تجاهل أي مسافات فارغة بادئة ومن ثم يُعثر على الثابت المناسب `subject sequence` متبوعًا بسلسلة محارف غير معترف عليها، ويكون المحرف الفارغ في نهاية السلسلة النصية غير مُعترف عليه دائمًا. تُحدد السلسلة المذكورة حسب التالي:

- في الدالة `strtod`: إشارة "+" أو "-" اختيارية في البداية متبوعة بسلسلة أرقام تحتوي على محرف الفاصلة العشرية الاختياري وأُس `exponent` اختياري أيضًا. لا يُعترف على أي لاحقة عائمة (ما بعد الفاصلة العشرية)، وتُعدّ الفاصلة العشرية تابعةً لسلسلة الأرقام إذا وُجدت.
- في الدالة `strtol`: إشارة "+" أو "-" اختيارية في البداية متبوعة بسلسلة أرقام، وتُؤخذ هذه الأرقام من الخانات العشرية أو من أحرف صغيرة `lower case` أو كبيرة `upper case` ضمن النطاق `a` إلى `z` في الأبجدية الإنجليزية ويُعطى لكل من هذه الأحرف القيم ضمن النطاق 10 إلى 35 بالترتيب. يحدّد الوسيط `base` القيم المسموحة ويمكن أن تكون قيمة الوسيط صفر أو ضمن النطاق 2 إلى 36. يجري التعرّف على الخانات التي تبلغ قيمتها أقل من الوسيط `base`، إذ تسمح القيمة 16 للوسيط `base` على سبيل المثال للمحارف `0x` أو `0X` أن تتبع الإشارة الاختيارية، بينما يسمح `base` بقيمة صفر أن تكون المحارف المدخلة على هيئة أعداد صحيحة ثابتة في سي، ولا يُعترف على أي لاحقة عدد صحيح.
- في الدالة `strtoul`: مطابقة للدالة `strtol` إلا أنها لا تسمح بوجود إشارة.

يُخزّن عنوان أول محرف غير مُعترف عليه في الكائن الذي يشير إليه `endptr` في حال لم يكن فارغًا، وتكون هذه قيمة `nptr` إذا كانت السلسلة فارغة أو ذات تنسيق خاطئ.

تحوّل الدالة الرقم وتُعيده مع الأخذ بالحسبان كون وجود الإشارة البادئة مسموحًا أو لا، وذلك إذا كان إجراء التحويل ممكنًا، وإلا فإنها تعيد القيمة صفر. عند حدوث الطفحان أو حصول خطأ تُجرى العمليات التالية:

- في الدالة `strtod`: تُعيد عند الطفحان القيمة `±HUGE_VAL` وتعتمد الإشارة على إشارة النتيجة، وتُعيد القيمة صفر عند طفحان الحد الأدنى `underflow` ويُضبط `errno` في الحالتين إلى القيمة `ERANGE`.
 - في الدالة `strtol`: تُعيد القيمة `LONG_MAX` أو `LONG_MIN` عند الطفحان بحسب إشارة النتيجة، ويُضبط `errno` في كلا الحالتين إلى القيمة `ERANGE`.
 - في الدالة `strtoul`: تُعيد القيمة `ULONG_MAX` عند الطفحان، ويُضبط `errno` إلى القيمة `ERANGE`.
- قد يكون هناك سلاسل أخرى من الممكن التعرّف عليها في بعض التنفيذات إذا لم تكن الإعدادات المحلية هي إعدادات سي التقليدية.

9.15.2 توليد الأرقام العشوائية

تقدّم الدوال التالية طريقةً لتوليد الأرقام العشوائية الزائفة `pseudo-random`:

```
#include <stdlib.h>

int rand(void);
void srand(unsigned int seed);
```

تُعيد الدالة `rand` رقمًا عشوائيًا زائفًا ضمن النطاق من "0" إلى "RAND_MAX"، وهو ثابت قيمته على الأقل "32767".

تسمح الدالة `srand` بتحديد نقطة بداية للنطاق المُختار طبقًا لقيمة الوسيط `seed`، وهي ذات قيمة "1" افتراضيًا إذا لم تُستدعى `srand` قبل `rand`، ونحصل على سلسلة قيم مطابقة من الدالة `rand` إذا استخدمنا قيمة `seed` ذاتها.

يصف المعيار الخوارزمية المستخدمة في الدالتين `rand` و `srand`، وتستخدم معظم التنفيذات هذه الخوارزمية عادةً.

9.15.3 حجز المساحة

تُستخدم هذه الدوال لحجز وتحرير المساحة، إذ يُضمن للمساحة التي حصلنا عليها أن تكون كبيرةً كفاية لتخزين كائن من نوع معين ومُحاذاة ضمن الذاكرة بحيث لا تتسبب بتفعيل استثناءات العنوان `addressing exceptions`، ولا يجب افتراض أي شيء آخر بخصوص عملها.

```
#include <stdlib.h>

void *malloc(size_t size);
```

```
void *calloc(size_t nmem, size_t size);
void *realloc(void *ptr, size_t size);

void *free(void *ptr);
```

تُعيد جميع دوال حجز المساحة مؤشراً يشير إلى المساحة المحجوزة التي يبلغ حجمها size بايت، وإذا لم يكن هناك أي مساحة فارغة فهي تعيد مؤشراً فارغاً، إلا أن الفرق بين الدوال هو أن دالة calloc تأخذ وسيطاً يدعى nmem الذي يحدد عدد العناصر في مصفوفة، وكل عنصر في هذه المصفوفة يبلغ حجمه size بايت، ولذا فهي تحجز مساحة أكبر من التخزين عادةً من المساحة التي تحجزها malloc، كما أن المساحة المحجوزة بواسطة malloc غير مُهيئة بينما تُهيأ جميع بتات مساحة التخزين المحجوزة بواسطة calloc إلى الصفر، إلا أن هذا الصفر ليس تمثيلاً مكافئاً للصفر ضمن الفاصلة العائمة floating-point أو مؤشراً فارغاً null بالضرورة.

تُستخدم الدالة realloc لتغيير حجم الشيء المُشار إليه بواسطة المؤشر ptr مما يتطلب بعض النسخ لإنجاز هذا الأمر ومن ثم تُحرر مساحة التخزين القديمة. لا تُغيّر محتويات الكائن المُشار إليه بواسطة ptr بالنسبة للحجمن القديم والجديد، وتتصرف الدالة تصرفاً مماثلاً لتصرف malloc إذا كان المؤشر ptr مؤشراً فارغاً وذلك للحجم المُخصّص.

تُستخدم الدالة free لتحرير المساحة المحجوزة مسبقاً من إحدى دوال حجز المساحة، ومن المسموح تمرير مؤشر فارغ إلى الدالة free وسيطاً لها، إلا أن الدالة في هذه الحالة لا تنفذ أي شيء.

إذا حاولت تحرير مساحة لم تُحجز مسبقاً تحصل على سلوك غير محدد، ويتسبب هذا الأمر في العديد من التنفيذات باستثناء عنوان addressing exception مما يوقف البرنامج، إلا أن هذه ليست بدالة يمكن الاعتماد عليها.

9.15.4 التواصل مع البيئة

سنستعرض مجموعةً من الدوال المتنوعة:

```
#include <stdlib.h>

void abort(void);
int atexit(void (*func)(void));
void exit(int status);
char *getenv(const char *name);
int system(const char *string);
```

- دالة abort: تتسبب بإيقاف غير اعتيادي للبرنامج وذلك باستخدام الإشارة SIGABRT، ويمكن منع الإيقاف غير الاعتيادي فقط إذا حصلنا على الإشارة ولم يُعد معالج الإشارة signal handler أي قيمة.

وإلا ستُحذف ملفات الخرج وقد يمكن أيضًا إزالة الملفات المؤقتة بحسب تعريف التنفيذ، وتُعاد حالة "إنهاء برنامج غير ناجح unsuccessful termination" إلى البيئة المُستضافة، كما أن هذه الدالة لا يمكن أن تُعيد أي قيمة.

- دالة `atexit`: يصبح وسيط الدالة `func` دالةً يمكن استدعاؤها دون استخدام أي وسطاء عندما يُغلق البرنامج، ويمكن استخدام ما لا يقل عن 32 دالة مشابهة لهذه الدالة وأن تُستدعى عند إغلاق البرنامج بصورةٍ مُعكسة لتسجيل كلٍّ منها، ونحصل على القيمة المُعاداة صفر للدلالة على النجاح وإلا فقيمة غير صفرية للدلالة على الفشل.

- دالة `exit`: تُستدعى هذه الدالة عادةً لإنهاء البرنامج على نحوٍ اعتيادي، وتُستدعى عند تنفيذ الدالة جميع الدوال المُسجّلة باستخدام دالة `atexit`، لكن انتبه، إذ ستُعد الدالة `main` بحلول هذه النقطة قد أعادت قيمتها ولا يمكن استخدام أي كائنات ذات مدة تخزين تلقائي `automatic storage duration` على نحوٍ آمن، من ثم تُحذف محتويات جميع مجاري الخرج `output streams` وتُغلق وتُزال جميع الملفات التلقائية المنشأة بواسطة `tmpfile`، وأخيرًا يُعيد البرنامج التحكم إلى البيئة المُستضافة بإعادة حالة نجاح أو فشل محدّدة بحسب التنفيذ، وتعتمد الحالة على إذا ما كان وسيط الدالة `exit` مساوٍ للقيمة `EXITSUCCESS` (وهذه حالة النجاح) أو `EXITFAILURE` (حالة الفشل). للتوافق مع لغة سي القديمة، تُستخدم القيمة صفر بدلاً من `EXITFAILURE`، بينما يكون لباقي القيم تأثيرات معرّفة بحسب التنفيذ. **لا يمكن** أن تُعاد حالة الخروج.

- دالة `getenv`: يجري البحث في **لائحة البيئة environment list** المعرفة بحسب التنفيذ بهدف العثور على عنصر يوافق السلسلة النصية المُشار إليها بواسطة وسيط الاسم `name`، إذ تُعيد الدالة مؤشرًا إلى العنصر يشير إلى مصفوفة لا يمكن تعديلها من قبل البرنامج ويمكن تعديلها باستدعاء لاحق للدالة `getenv`، ونحصل على مؤشر فارغ إذا لم يُعثَر على عنصر موافق. يعتمد الهدف من لائحة البيئة وتنفيذها على البيئة المُستضافة.

- دالة `system`: تُمرّر سلسلة نصية إلى أمر معالج مُعرف حسب التنفيذ، ويتسبب مؤشر فارغ بإعادة القيمة صفر، وقيمة غير صفرية إذا لم يكن الأمر موجودًا، بينما يتسبب مؤشر غير فارغ بمعالجة الأمر. نتيجة الأمر والقيمة المُعاداة معرف حسب التنفيذ.

9.15.5 البحث والترتيب

هناك دالتان ضمن هذا التصنيف، أولهما دالة للبحث ضمن لائحة مُرتّبة والأخرى لترتيب لائحة غير مرتّبة، واستخدام الدالتان عام، إذ يمكن استخدامهما في مصفوفات من أي سعة وعناصرها من أي حجم.

يجب أن يلجأ المستخدم إلى دالة مقارنة إذا أراد مقارنة عنصرين عند استخدام الدوال السابقة، إذ تُستدعى هذه الدالة باستخدام المؤشرين الذين يشيران إلى العنصرين مثل وسطاء الدالة، وتُعيد الدالة قيمةً أقل من

الصفري إذا كانت قيمة المؤشر الأول أصغر من قيمة المؤشر الثاني، وقيمة أكبر من الصفري إذا كانت قيمة المؤشر الأول أكبر من المؤشر الثاني، والقيمة صفر إذا كانت قيمتا المؤشرين متساويين.

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base,
             size_t nmem, size_t size,
             int (*compar)(const void *, const void *));

void *qsort(const void *base, size_t nmem,
           size_t size,
           int (*compar)(const void *, const void *));
```

يمثل الوسيط `nmem` في كلٍّ من الدالتين السابقتين عدد العناصر في المصفوفة، ويمثل الوسيط `size` حجم عنصر المصفوفة الواحد بالبايت و `compar` هي الدالة التي تُستدعى للمقارنة بين العنصرين، بينما يشير المؤشر `base` إلى أساس المصفوفة أي بدايتها.

ترتب الدالة `qsort` المصفوفة ترتيبًا تصاعديًا.

تفترض الدالة `bsearch` أن المصفوفة مرتبة مسبقًا وتُعيد مؤشرًا إلى أي عنصر يتساوى مع العنصر المُشار إليه بالمؤشر `key`، وتُعيد الدالة مؤشرًا فارغًا إذا لم تجد أي عناصر متساوية.

9.15.6 دوال العمليات الحسابية الصحيحة

تقدّم هذه الدوال طريقةً لإيجاد القيمة المطلقة لوسيط يمثل عدد صحيح، إضافةً لحاصل القسمة والباقي من العملية لكلٍّ من النوعين `int` و `long`.

```
#include <stdlib.h>

int abs(int j);
long labs(long j);

div_t div(int numerator, int denominator);
ldiv_t ldiv(long numerator, long denominator);
```

- الدالتان `abs` و `labs`: تُعيدان القيمة المطلقة لوسيطهما ويجب اختيار الدالة المناسبة بحسب احتياجاتك. نحصل على سلوك غير معرّف إذا لم تكن القيمة ممكنة التمثيل وقد يحدث ذلك في

الأنظمة التي تعمل بنظام المتمم الثنائي two's complement systems، إذ لا يوجد لأكثر رقم سلبية أي مكافئ إيجابي.

- الدالتان div وldiv: تُقسّمان الوسيط numerator على الوسيط denominator وتُعيدان هيكلًا structure للنوع المحدد، وفي أي حالة، سيحتوي الهيكل على عضو يدعى quot يحتوي على حاصل القسمة الصحيحة وعضوًا آخر يدعى rem يحتوي على باقي القسمة، ونوع العضوين هو int في الدالة div و long في الدالة ldiv، ويمكن تمثيل نتيجة العملية على النحو التالي:

```
quot*denominator+rem == numerator
```

9.15.7 الدوال التي تستخدم المحارف متعددة البايت

يؤثر تصنيف LC_CTYPE ضمن الإعدادات المحلية الحالية على سلوك هذه الدوال، إذ تُضبط كل دالة إلى حالة ابتدائية باستدعاء يكون وسيطها s الذي يمثل مؤشر المحرف فارغًا null، وذلك في حالة الترميز المُعتمد على الحالة state-dependent encoding، وتُغيّر حالة الدالة الداخلية وفق الضرورة عن طريق استدعاءات لاحقة عندما لا يكون s مؤشرًا فارغًا. تُعيد الدالة قيمةً غير صفرية إذا كان الترميز معتمدًا على الحالة، وإلا فتعيد القيمة صفر إذا كان المؤشر s فارغًا. تصبح حالة الإزاحة shift state الخاصة بالدوال غير محددة indeterminate إذا حدث تغيير للتصنيف LC_TYPE.

الدوال هي:

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

- الدالة mblen: تُعيد عدد البايتات المُحتواة بداخل محرف متعدد البايتات multibyte character المُشار إليه بواسطة المؤشر s أو تُعيد القيمة -1 إذا كان أول n بايت لا يشكّل محرف متعدد البايتات صالحًا، أو تُعيد القيمة صفر إذا كان المؤشر يشير إلى محرف فارغ.
- الدالة mbtowc: تُحوّل محرف متعدد البايتات يُشير إليه المؤشر s إلى الرمز الموافق له من النوع wchar_t وتُخزّن النتيجة في الكائن المُشار إليه بالمؤشر pwc، إلا إن كان pwc مؤشرًا فارغًا، وتُعيد الدالة عدد البايتات المُحوّلة بنجاح، أو -1 إذا لم تشكّل أول n بايت محرفًا متعدد البايت صالحًا، ولا يُفحص أكثر من n بايت يُشير إليه المؤشر s، ولن تتعدى القيمة المُعاداة قيمة n أو MB_CUR_MAX.

- دالة `wctomb`: تُحوّل رمز القيمة `wchar` إلى سلسلة من البايتات تمثل محرف متعدد البايتات وتخزن النتيجة في مصفوفة يشير إليها المؤشر `s` وذلك إن لم يكن `s` مؤشرًا فارغًا، وتُعيد الدالة عدد البايتات المحتواة داخل محرف متعدد البايتات، أو `-1` إذا كانت القيمة المخزنة في `wchar` لا تمثل محرف متعدد البايتات، ومن غير الممكن معالجة عدد بايتات يتجاوز `MB_CUR_MAX`.
- دالة `mbstowcs`: تحوّل سلسلة محارف متعددة البايتات بدءًا من الحالة الأولية للإزاحة `initial shift state` وذلك ضمن المصفوفة التي يشير إليها المؤشر `s` إلى سلسلة من الرموز الموافقة ومن ثم تخزنها في مصفوفة يشير إليها المؤشر `pwcs`، لا يُخزّن ما يزيد عن `n` قيمة في `pwcs`، وتُعيد الدالة `-1` إذا صادفت محرفًا متعدد البايت غير صالح، وإلا فإنها تعيد عدد عناصر المصفوفة المُعدّلة باستثناء رمز إنهاء المصفوفة.
- نحصل على سلوك غير معرّف إذا وجد كائنين متقاطعين.
- الدالة `wcstombs`: تُحوّل سلسلة من الرموز المُشار إليها بالمؤشر `pwcs` إلى سلسلة من المحارف متعددة البايتات بدءًا من الحالة الأولية للإزاحة وتُخزّن فيما بعد في مصفوفة مُشار إليها بالمؤشر `s`. تتوقف عملية التحويل عند مصادفة رمز فارغ، أو عند كتابة `n` بايت إلى `s`، وتُعيد الدالة `-1` إذا كان الرمز المُصادف لا يمثل محرفًا متعدد البايتات صالحًا، وإلا فيُعاد عدد البايتات التي كُتبت باستثناء رمز الإنهاء الفارغ.
- نحصل على سلوك غير محدد إذا وجد كائنين متقاطعين.

9.16 التعامل مع السلاسل النصية

هناك العديد من الدوال التي تسمح لنا بالتعامل مع السلاسل النصية، إذ تكون السلسلة النصية في لغة سي مؤلفةً من مصفوفة من المحارف تنتهي بمحرف فارغ `null`، وتتوقع الدوال في جميع الحالات تمرير مؤشر يشير إلى المحرف الأول ضمن السلسلة النصية، ويعرّف ملف الترويسة `<string.h>` هذا النوع من الدوال.

9.16.1 النسخ

يضم هذا التصنيف الدوال التالية:

```
#include <string.h>

void *memcpy(void *s1, const void *s2, size_t n);
void *memmove (void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
```

```
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
```

- دالة memcopy: تنسخ هذه الدالة n بايت من المكان الذي يشير إليه المؤشر s2 إلى المكان الذي يشير إليه المؤشر s1، ونحصل على سلوك غير محدد إذا كان الكائنات متداخلان overlapping objects. تعيد الدالة s1.
- دالة memmove: هذه الدالة مطابقة لعمل دالة memcopy إلا أنها تعمل على الكائنات المتداخلة، إلا أنها قد تكون أبطأ.
- دالتَي strcpy و strncpy: تنسخ كلا الدالتين السلسلة النصية التي يشير إليها المؤشر s2 إلى سلسلة نصية يشير المؤشر s1 إليها متضمنًا ذلك المحرف الفارغ في نهاية السلسلة. تنسخ strncpy سلسلة نصية بطول n بايت على الأكثر، وتحشو ما تبقى بمحارف فارغة إذا كانت s2 أقصر من n محرف، ونحصل على سلوك غير معرّف، إذا كانت السلسلتان متقاطعتين، وتُعيد كلا الدالتين s1.
- الدالتان strcat و strncat: تُضيف كلا الدالتين السلسلة النصية s2 إلى السلسلة s1 بالكتابة فوق overwrite المحرف الفارغ في نهاية السلسلة s1، بينما يُضاف المحرف الفارغ دائمًا إلى نهاية السلسلة. يمكن إضافة n محرف على الأكثر من السلسلة s2 باستخدام الدالة strncat مما يعني أن السلسلة النصية الهدف (أي s1) يجب أن تحتوي على مساحة لطولها الأصلي (دون احتساب المحرف الفارغ) زائد n+1 محرف للتنفيذ الآمن. تعيد الدالتين s1.

9.16.2 مقارنة السلسلة النصية والبايت

تُستخدم هذه الدوال في مقارنة مصفوفات من البايتات، وهذا يتضمن طبعًا السلاسل النصية في لغة سي إذ أنها سلسلة من المحارف char (أي البايتات) بمحرف فارغ في نهايتها. تعمل جميع هذه الدوال التي سنذكرها على مقارنة بايت تلو الآخر وتتوقف فقط في حالة اختلف بايت مع بايت آخر (في هذه الحالة تُعيد الدالة إشارة الفرق بين البايت والآخر) أو عندما تكون المصفوفتان متساويتين (أي لم يُعثر على أي فرق بينهما وكان طولهما مساوٍ إلى الطول المحدد أو -في حالة المقارنة بين السلاسل النصية- وُجد المحرف الفارغ في النهاية).

القيمة المُعادَة في جميع الدوال عدا الدالة strcmp هي أصغر من الصفر، أو تساوي الصفر، أو أكبر من الصفر وذلك في حالة كان الكائن الأول أصغر من الكائن الثاني، أو يساوي الكائن الثاني، أو أكبر من الكائن الثاني على الترتيب.

```
#include <string.h>
```

```
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *to, const char *from,
int strcoll(const char *s1, const char *s2);
```

- دالة `memcmp`: تُقارن أول `n` محرف في الكائن الذي يشير إليه المؤشر `s1` و `s2`، إلا أن مقارنة الهياكل بهذه الطريقة ليست مثالية، إذ قد تحتوي الاتحادات `unions` أو "الثقوب" `holes` المُسببة بواسطة محاذاة ذاكرة التخزين على بيانات غير صالحة.
 - دالة `strcmp`: تُقارن سلسلتين نصيتين وهي إحدى أكثر الدوال استخدامًا عند التعامل مع السلاسل النصية.
 - الدالة `strncmp`: تُطابق عمل الدالة `strcmp` إلا أنها تقارن `n` محرف على الأكثر.
 - الدالة `strxfrm`: تُحوّل السلسلة النصية المُمرّرة إليها (بصورة خاصة ومميّزة)، وتُخزّن إلى موضع المؤشر، ويكتب `maxsize` محرف على الأكثر إلى موضع المؤشر (متضمنًا المحرف الفارغ في النهاية)، وتضمن طريقة التحويل أننا سنحصل على نتيجة المقارنة ذاتها لسلسلتين نصيتين محوّلتين ضمن إعدادات المستخدم المحلية عند استخدام الدالة `strcmp` بعد تطبيق الدالة `strcoll` على السلسلتين النصيتين الأساسيتين.
- نحصل على طول السلسلة النصية الناتجة مثل قيمة مُعادة في جميع الحالات (باستثناء المحرف الفارغ في نهاية السلسلة)، وتكون محتويات المؤشر `to` غير معرّفة إذا كانت القيمة مساوية أو أكبر من `maxsize`، وقد تكون `s1` مؤشرًا فارغًا إذا كانت `maxsize` صفر، ونحصل على سلوك غير معرّف إذا كان الكائنان متقاطعين.
- دالة `strcoll` تُقارن هذه الدالة لسلسلتين نصيتين بحسب سلسلة الترتيب `collating sequence` المحدد في إعدادات اللغة المحلية.

9.16.3 دوال بحث الحروف والسلاسل النصية

يتضمن التصنيف الدوال التالية:

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
```

```
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(const char *s1, const char *s2);
```

- دالة memchr: تُعيد مؤشرًا يشير إلى أول ظهور ضمن أول n حرف من *s للمحرف c (من نوع unsigned char)، وتُعيد فراغًا null إن لم يُعثر على أي تطابق.
- دالة strchr: تُعيد مؤشرًا يشير إلى أول ظهور للمحرف c ضمن *s ويتضمن البحث المحرف الفارغ، وتُعيد فراغًا null إذا لم يُعثر على أي تطابق.
- دالة strcspn: تُعيد طول الجزء الأولي للسلسلة النصية s1 الذي لا يحتوي أيًا من محارف السلسلة s2، ولا يُؤخذ المحرف الفارغ في نهاية السلسلة s2 بالحسبان.
- دالة strpbrk: تُعيد مؤشرًا إلى أول حرف ضمن s1 يطابق أي حرف من محارف السلسلة s2 أو تُعيد فراغًا إن لم يُعثر على أي تطابق.
- دالة strrchr: تُعيد مؤشرًا إلى آخر حرف ضمن s1 يطابق المحرف c آخذة بالحسبان المحرف الفارغ على أنه جزء من السلسلة s1 وتُعيد فراغ إن لم يُعثر على تطابق.
- دالة strspn: تُعيد طول الجزء الأولي ضمن السلسلة s1 الذي يتألف كاملًا من محارف السلسلة s1.
- دالة strstr: تُعيد مؤشرًا إلى أول تطابق للسلسلة s2 ضمن السلسلة s1 أو تُعيد فراغ إن لم يُعثر على تطابق.
- دالة strtok: تقسم السلسلة النصية s1 إلى "رموز tokens" يُحدّد كل منها بمحرف من محارف السلسلة s2 وتُعيد مؤشرًا يشير إلى الرمز الأول أو فراغًا إن لم يوجد أي رموز. تُعيد استدعاءات لاحقة للدالة باستخدام (char *)0 قيمةً للوسيط s1 الرمز التالي ضمن السلسلة، إلا أن s2 (المُحدّد) قد يختلف عند كل استدعاء، ويُعاد مؤشر فارغ إذا لم يبق أي رمز.

9.16.4 دوال متنوعة أخرى

هناك بعض الدوال الأخرى التي لا تنتمي لأي من التصنيفات السابقة:

```
void *memset(void *s, int c, size_t n);
char *strerror(int errnum);
size_t strlen(const char *s);
```

- دالة `memset`: تضبط `n` بايت يشير إليها المؤشر `s` إلى قيمة المحرف `c` وهو من النوع `unsigned char`، وتُعيد الدالة المؤشر `s`.
- دالة `strlen`: تُعيد طول السلسلة النصية `s` دون احتساب المحرف الفارغ في نهاية السلسلة، وهي دالة شائعة الاستخدام.
- دالة `strerror`: تُعيد مؤشرًا يشير إلى سلسلة نصية تصف الخطأ رقم `errno`، وقد تُعدّل هذه السلسلة النصية عن طريق استدعاءات لاحقة للدالة `sterror`، وتعدّ هذه الدالة مفيدة لمعرفة معنى قيم `errno`.

9.17 التاريخ والوقت

تتعامل هذه الدوال إما مع الوقت المُنقضي `elapsed time`، أو وقت التقويم `calendar time` ويحتوي ملف الترويسة `<time.h>` على تصريح كلا النوعين من الدوال بالاعتماد على التالي:

- القيمة `CLOCKS_PER_SEC`: عدد الدقات `ticks` في الثانية المُعادة من الدالة `clock`.
- النوعين `clock_t` و `time_t`: أنواع حسابية تُستخدم لتمثيل تنسيقات مختلفة من الوقت.
- الهيكل `struct tm`: يُستخدم لتخزين القيمة المُتمثلة لأوقات التقويم، ويحتوي على الأعضاء التالية:

```
int tm_sec      // الثواني بعد الدقيقة من 0 إلى 61، وتسمح 61 بثانيتين كبيستين - leap-second
int tm_min      // الدقائق بعد الساعة من 0 إلى 59
int tm_hour      // الساعات بعد منتصف الليل من 0 إلى 23
int tm_mday      // اليوم في الشهر من 1 إلى 31
int tm_mon       // الشهر في السنة من 0 إلى 11
int tm_year      // السنة الحالية من 1900
int tm_wday      // الأيام منذ يوم الأحد من 0 إلى 6
int tm_yday      // الأيام منذ الأول من يناير من 0 إلى 365
int tm_isdst     // مؤشر التوقيت الصيفي
```

يكون العنصر `tm_isdst` موجبًا إذا كان التوقيت الصيفي `daylight savings` فعّالًا، وصفر إن لم يكن كذلك، وسالبًا إن لم تكن هذه المعلومة متوفرة.

إليك دوال التلاعب بالوقت:

```
#include <time.h>

clock_t clock(void);
```

```
double difftime(time_t time1, time_t time2);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char *s, size_t maxsize,
    const char *format,
    const struct tm *timeptr);
```

تتشارك كل من الدوال `asctime` و `ctime` و `gmtime` و `localtime` و `strftime` بهياكل بيانات ساكنة `static` من نوع `struct tm` أو من نوع `char []`، وقد يتسبب استدعاء أحد منها بعملية الكتابة فوق البيانات المخزنة بسبب استدعاء سابق لإحدى الدوال الأخرى، ولذلك يجب على مستخدم الدالة نسخ المعلومات إذا كان هذا سيسبب أية مشاكل.

- الدالة `clock`: تُعيد أفضل تقريب للوقت الذي انقضى منذ تشغيل البرنامج مقدراً بدقات الساعة `ticks`، وتُعاد القيمة `1-(clock_t)` إذا لم يُعثَر على أي قيمة. من الضروري العثور على الفرق بين وقت بداية تشغيل البرنامج والوقت الحالي إذا أردنا إيجاد الوقت المُنقضي اللازم لتشغيل البرنامج، وهناك ثابتٌ معرفٌ حسب التنفيذ يعدل على القيمة المُعادة من `clock`. يجب تقسيم القيمة على `CLOCKS_PER_SEC` لتحديد الوقت بالثواني.
- الدالة `difftime`: تُعيد الفرق بين وقت تقويم ووقت تقويم آخر بالثواني.
- الدالة `mktime`: تُعيد وقت تقويم يوافق القيم الموجودة في هيكل يشير إلى المؤشر `timeptr`، أو تُعيد القيمة `1-(time_t)` إذا لم يكن من الممكن تمثيل القيمة. يُتجاهل العضوان `tm_wday` و `tm_yday`، ولا تُقَيّد باقي الأعضاء بقيمهم الاعتيادية، إذ يُضبط أعضاء الهيكل إلى قيم مناسبة ضمن النطاق الاعتيادي عند التحويل الناجح، وهذه الدالة مفيدة للعثور على التاريخ والوقت الموافق لقيمة من نوع `time_t`.
- الدالة `time`: تُعيد أفضل تقريب لوقت التقويم الحالي باستخدام ترميز غير محدد `unspecified encoding`، وتُعيد القيمة `1-(time_t)` إذا كان الوقت غير متوفر.
- الدالة `asctime`: تُحول الوقت ضمن هيكل يشير إليه المؤشر `timeptr` إلى سلسلة نصية بالتنسيق التالي:

Sun Sep 16 01:03:52 1973\n\0

المثال السابق مأخوذٌ من المعيار، إذ يعرّف المعيار الخوارزمية المستخدمة أيضًا، إلا أنه من المهم ملاحظة أن جميع الحقول ضمن السلسلة النصية ذات عرض ثابت وينطبق استخدامها على المجتمعات التي تتحدث باللغة الإنجليزية فقط. تُخزّن السلسلة النصية في هيكل ساكن `static structure` ويُمكن إعادة كتابته عن طريق استدعاءات لاحقة لأحد دوال التلاعب بالوقت (المذكورة أعلاه).

- الدالة `ctime`: تكافئ عمل `(localtime(timer))`. اقرأ عن الدالة `asctime` لمعرفة القيمة المُعادة.
- الدالة `gmtime`: تُعيد مؤشرًا إلى `struct tm`، إذ يُضبط هذا المؤشر إلى وقت التقويم الذي يشير إليه المؤشر `timer`، ويُمثل الوقت بحسب شروط التوقيت العالمي المُنسّق `Coordinated Universal Time` -أو اختصارًا `UTC`-، أو المسمى سابقًا بتوقيت جرينتش `Greenwich Mean Time`، ونحصل على مؤشر فارغ إذا كان توقيت `UTC` غير مُتاح.
- الدالة `localtime`: تحوّل الوقت الذي يشير إليه المؤشر `timer` إلى التوقيت المحلي وتُخزن النتيجة في `struct tm` وتُعيد مؤشرًا يشير إلى ذلك الهيكل.
- الدالة `strftime`: تملأ مصفوفة المحارف التي يشير إليها المؤشر `s` بمقدار `maxsize` محرف على الأكثر، وتُستخدم السلسلة النصية `format` لتنسيق الوقت المُمثّل في الهيكل الذي يشير إليه المؤشر `timeptr`، تُنسخ المحارف الموجودة في سلسلة التنسيق النصية (متضمنة المحرف الفارغ في نهاية السلسلة) دون أي تغيير إلى المصفوفة إلا إن كان وُجد توجيه تنسيق من التوجيهات التالية، فعندها تُنسخ القيمة المُحددة ضمن الجدول إلى المصفوفة الهدف بما يوافق الإعدادات المحلية.

الجدول 26: محددات الحجم

%a	اسم يوم الأسبوع باختصار
%A	اسم يوم الأسبوع كاملاً
%b	اسم الشهر باختصار
%B	اسم الشهر كاملاً
%c	تمثيل التاريخ والوقت
%d	تمثيل يوم الشهر عشرياً من 01 إلى 31
%H	الساعة من 00 إلى 23 (تنسيق 24 ساعة)
%I	الساعة من 01 إلى 12 (تنسيق 12 ساعة)
%j	يوم السنة من 001 إلى 366
%m	الشهر من 01 إلى 12
%M	الدقيقة من 00 إلى 59

%p	مكافئة PM أو AM المحلي
%S	الثانية من 00 إلى 61
%U	ترتيب الأسبوع ضمن السنة من 00 إلى 53 -الأحد هو اليوم الأول-
%w	يوم الأسبوع من 0 إلى 6 (الأحد مُمَثَّل بالرقم 0)
%W	ترتيب الأسبوع ضمن السنة من 00 إلى 53 -الاثنين هو اليوم الأول-
%x	تمثيل التاريخ محليًا
%X	تمثيل الوقت محليًا
%y	السنة دون سابقة القرن من 00 إلى 99
%Y	السنة مع سابقة القرن
%Z	اسم المنطقة الزمنية، لا نحصل على محارف إن لم يكن هناك أي منطقة زمنية
%%	محرف %

يُعاد عدد المحارف الكلية المنسوخة إلى `%s*` باستثناء محرف الفراغ في نهاية السلسلة، وتُعاد القيمة صفر إذا لم يكن هناك أي مساحة (بحسب قيمة `maxsize`) للمحرف الفارغ في النهاية.

9.18 الخاتمة

سيكون لتوحيد مكتبات وقت التشغيل `run-time libraries` التأثير الأكبر على قابلية نقل برامج لغة سي C بلا أي شك، ويجب أن يقرأ جميع مستخدمو لغة سي هذا الفصل بحرص وأن يألفوا محتوياته، إذ يُعد نقص عدد المكتبات القابلة للنقل من أكبر المشاكل لتحقيق برامج يمكن نقلها.

يا لحظك العاثر إن كنت تكتب برامجًا للأنظمة المدمجة، إذ أن المكتبات ليست مُعرّفة للتطبيقات المستقلة `stand-alone applications`، إلا أننا نتوقع من المزودين أن يُنتجوا حزم مكتبات للتطبيقات المستقلة أيضًا، وهي على الأغلب ستأتي دون إمكانية التعامل مع الملفات، ومع قولنا هذا إلا أنه لا يوجد هناك أي سبب في عدم عمل دوال التعامل مع السلاسل النصية -على سبيل المثال- على النحو ذاته التي تعمل في البيئات المُستضافة `hosted` وغير المُستضافة `unhosted`.



أكبر موقع توظيف عن بعد في العالم العربي

ابحث عن الوظيفة التي تحقق أهدافك وطموحاتك
المهنية في أكبر موقع توظيف عن بعد

[تصفح الوظائف الآن](#)

10. تطبيقات عملية

بعد أن تكلمنا عن المكتبات القياسية المعرفة حسب المعيار، لم يبقَ إلا توضيح هيئة البرامج الكاملة المكتوبة بهذه اللغة، وسنتطرق في هذا الفصل إلى بعض الأمثلة التي توضح كيفية جمع هذه العناصر لبناء البرامج.

إلا أن هناك بعض النقاط التي يجب مناقشتها في لغة سي قبل عرض هذه الأمثلة.

10.1 وسطاء الدالة main

تقدم وسطاء الدالة main فرصة مفيدة لكل من يكتب البرامج ويشغلها في بيئة مُستضافة hosted environment وذلك بإعطاء المعاملات للبرنامج، ويُستخدم ذلك عادةً لتوجيه البرنامج لكيفية تنفيذ مهامه، ومن الشائع أن تكون هذه المعاملات هي أسماء ملفات تُمرَّر مثل وسيط.

يبدو تصريح الدالة main على النحو التالي:

```
int main(int argc, char *argv[]);
```

يُشير التصريح إلى أن الدالة main تُعيد عددًا صحيحًا، وتُمرَّر هذه القيمة عادةً، أو حالة الخروج exit status في البيئات المُستضافة، مثل أنظمة دوس DOS أو يونيكس UNIX إلى مفسر سطر الأوامر command line interpreter، فعلى سبيل المثال تُستخدم حالة الخروج في نظام يونيكس للدلالة على إتمام البرنامج لمهامه بنجاح (تمثله القيمة صفر)، أو حدوث خطأ أثناء التنفيذ (تمثله قيمة غير صفرية). أتبع المعيار هذا الاصطلاح أيضًا، إذ تُستخدم exit(0) لإعادة حالة النجاح إلى البيئة المُستضافة وأي قيمة أخرى تدلّ على حدوث خطأ ما، وستُترجم exit القيمة لمعناها إذا كانت البيئة المُستضافة تستخدم اصطلاحًا مخالفًا. بما أن الترجمة معرفة

حسب التنفيذ، فمن الأفضل استخدام القيمتان المُعرّفتان في ملف الترويسة `<stdlib.h>`، وهما: `EXIT_SUCCESS` و `EXIT_FAILURE`.

هناك على الأقل وسيطان للدالة `main`، وهُما: `argc` و `argv`، إذ يدل أولهما على عدد الوسطاء المزودة للبرنامج، بينما يدل الثاني على مصفوفة من المؤشرات تشير إلى سلاسل نصية تمثل الوسطاء، وهي من النوع "مصفوفة من المؤشرات تشير إلى محرف `char`"، وتُمرّر هذه الوسطاء إلى البرنامج باستخدام مفسّر سطر الأوامر الخاص بالبيئة المُستضافة، أو لغة التحكم بالوظائف `job control language`.

يُعدّ تصريح الوسيط `argv` أول مصادفة للمبرمجين المبتدئين مع المؤشرات التي تشير إلى مصفوفات من المؤشرات، وقد يكون هذا محيرًا بعض الشيء في البداية، إلا أن الأمر بسيط الفهم. بما أن `argv` تُستخدم للإشارة إلى مصفوفة السلاسل النصية، يكون تصريحها على النحو التالي:

```
char *argv[]
```

تذكر أيضًا أن اسم المصفوفة يُحوّل إلى عنوان أول عنصر ضمنها عندما تُمرّر إلى دالة، وهذا يعني أنه يمكننا التصريح عن `argv` كما يلي: `char **argv` والتصريحان يؤديان الغرض ذاته في هذه الحالة.

سترى تصريح الدالة `main` مكتوبًا بهذه الطريقة معظم الأحيان، والتصريح التالي مكافئ للتصريح السابق:

```
int main(int argc, char **argv);
```

تُهيئ وسطاء الدالة `main` عند بداية تشغيل البرنامج على نحوٍ موافق للشروط التالية:

- الوسيط `argc` أكبر من الصفر.
- يمثل `argv[argc]` مؤشرًا فارغًا `null`.
- تمثل العناصر بدءًا من `argv[0]` وصولاً إلى `argv[argc-1]` مؤشرات تشير إلى سلاسل نصية يُحدّد البرنامج معناها.
- يحتوي العنصر `argv[0]` السلسلة النصية التي تحتوي اسم البرنامج أو سلسلة نصية فارغة إذا لم تكن هذه المعلومة متاحة، وتمثل العناصر المتبقية من `argv` الوسطاء المزودة للبرنامج. يُزوّد محتوى السلاسل النصية إلى البرنامج بحالة الأحرف الصغيرة `lower-case` في حال توفر الدعم فقط للأحرف الوحيدة `single`.

لتوضيح هذه النقطة، إليك مثالًا عن برنامج يكتب وسطاء الدالة `main` إلى خرج البرنامج القياسي:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char **argv)
{
    while(argc--)
        printf("%s\n", *argv++);
    exit(EXIT_SUCCESS);
}
```

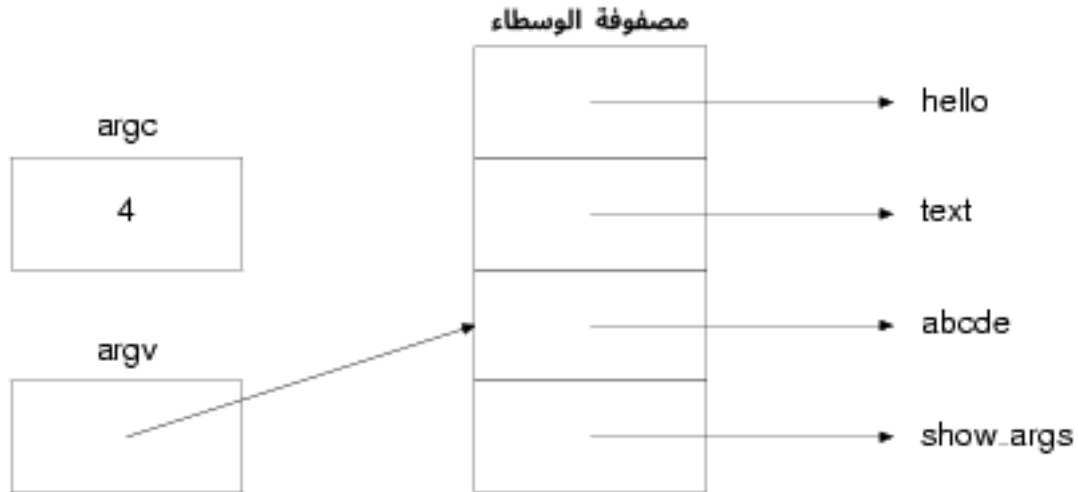
[مثال 1]

إذا كان اسم البرنامج `show_args` وكانت وسطاءه `abcde` و `text` و `hello` عند تشغيله، ستكون حالة الوسطاء وقيمة `argv` موضحة في الشكل التالي:



الشكل 14: وسطاء البرنامج

تنتقل `argv` إلى العنصر التالي عند كل زيادة لها، وبالتالي وبعد أول تكرار للحلقة سَتُشير `argv` إلى المؤشر الذي بدوره يشير إلى الوسيط `abcde`، وهذا الأمر موضح بالشكل التالي:



الشكل 15: وسطاء البرنامج بعد زيادة argv

سيعمل البرنامج على النظام الذي جربنا فيه البرنامج السابق عن طريق كتابة اسمه، ثم كتابة وسطاءه وفصلهم بمسافات فارغة. إليك ما الذي يحدث (الرمز \$ هو رمز الطرفية):

```
$ show_args abcde text hello
show_args
abcde
text
hello
$
```

10.2 تفسير وسطاء البرنامج

الحلقة المُستخدمة لفحص وسطاء البرنامج في المثال السابق شائعة الاستخدام في برامج سي C وستجدها في العديد من البرامج الأخرى، ويُعد استخدام "الخيارات options" للتحكم بسلوك البرنامج طريقة شائعة أيضًا (تُدعى أيضًا في بعض الأحيان المُبدلات switches أو الرايات flags)، إذ يدل الوسيط الذي يبدأ بالمحرف - على أنه وسيط يقدّم حرفًا وحيدًا أو أكثر يشير إلى خيار، ويمكن تشغيل الخيارات سويًا أو على نحو منفرد:

```
programe -abxu file1 file2
programe -a -b -x -u file1 file2
```

يحدّد كلاً من الخيارات جانبًا معيّنًا من مزايا البرنامج، وقد يُسمح لكل خيار بأخذ وسيط خاص به امتدادًا لهذه الفكرة، فعلى سبيل المثال إذا كان الخيار -x يأخذ وسيطًا خاصًا به، سيبدو ذلك على النحو التالي:

```
programe -x arg file1
```

وبذلك، فإن `arg` مرتبطة مع الخيار. تسمح لنا دالة `options` في الأسفل بأتمتة معالجة أسلوب الاستخدام هذا عن طريق الدعم الإضافي (شائع الاستخدام إلا أنه قد عفا عليه الزمن) لإمكانية تقديم خيار الوسيط مباشرةً بعد حرف الخيار كما يلي:

```
programe -xarg file1
```

تُعيد برامج الخيارات السابقة في كلِّ من الحالتين المحرف 'x' وتضبط المؤشر العام `global` المسمى `OptArg` ليشير إلى القيمة `arg`.

يجب أن يقدم البرنامج لائحةً من أحرف الخيارات الصالحة بهيئة سلسلة نصية حتى نستطيع استخدام برامج الخيارات، عندما يُلحق حرفٌ ضمن هذه السلسلة النصية بالنقطتين ':'. فهذا يعني أن ما يتبع حرف الخيار هو وسيط. يُستدعى برنامج `options` مرارًا عند تشغيل البرنامج حتى انتهاء أحرف الخيار.

يبدو أن الدوال التي تقرأ السلاسل النصية وتبحث عن تشكيلات مختلفة أو أنماط ضمن السلسلة صعبة القراءة، وإن كان في ذلك عزاءً لكن ليست عملية القراءة بتلك البساطة فعليًا، والشيفرة البرمجية التي تطبق الخيارات هي واحدةٌ من هذه الدوال، إلا أنها ليست الأضعف ضمن هذا التصنيف.

تفحص الدالة `options()` أحرف الخيار ووسطاء الخيار من قائمة `argv`، وتُعيد استدعاءات متتابعة للدالة أحرف خيار متتابعة متوافقة مع واحدة من بنود القائمة `legal`. قد تتطلب أحرف الخيار ووسطاء خيار ويُشار إلى ذلك بالنقطتين ':'. اللتين تتبعان الحرف في القائمة `legal`. على سبيل المثال، تشير لائحة `legal` التي تحتوي على "ab:c" على أن `a` و `b` و `c` جميعها خيارات صالحة وأن `b` تأخذ وسيط خيار، ويُمرّر وسيط الخيار فيما بعد إلى الدالة التي استُدعيت سابقًا في قيمة المؤشر العام المُسمى `OptArg`. يُعطي `OptIndex` السلسلة النصية التالية في مصفوفة `argv[]` التي لم تُعالج بعد من قبل الدالة `options()`.

تُعيد الدالة `options()` القيمة 1- إذا لم يكن هناك أي أحرف خيار أخرى، أو إذا عُثر على `SwitchChar` مضاعف، ويُجبر ذلك الدالة `options()` على إنهاء عملية معالجة الخيارات؛ بينما تُعيد 0 إذا كان هناك خيار لا ينتمي إلى مجموعة `legal`، أو إذا عُثر على خيار ما يحتاج لوسيط دون وجود وسيط يتبعه.

```
#include <stdio.h>
#include <string.h>

static const char SwitchChar = '-';
static const char Unknown = '?';

int OptIndex = 1; // يجب أن يكون أول خيار هو argv[1]
char *OptArg = NULL; // مؤشر عام لوسيط الخيار
```

```

int options(int argc, char *argv[], const char *legal)
{
    static char *posn = ""; // argv[OptIndex]  الموقع في
    char *legal_index = NULL;
    int letter = 0;

    if(!*posn){
        // لا يوجد المزيد من args أو SwitchChar أو حرف خيار
        if((OptIndex >= argc) ||
            (*(posn = argv[OptIndex]) != SwitchChar) ||
            !*++posn)
            return -1;

        // إيجاد SwitchChar مضاعف
        if(*posn == SwitchChar){
            OptIndex++;
            return -1;
        }
    }
    letter = *posn++;
    if(!(legal_index = strchr(legal, letter))){
        if(!*posn)
            OptIndex++;
        return Unknown;
    }
    if(*++legal_index != ':'){
        /* لا يوجد وسيط للخيار */
        OptArg = NULL;
        if(!*posn)
            OptIndex++;
    } else {
        if(*posn)
            // لا يوجد مسافة فارغة بين opt و opt arg
            OptArg = posn;
        else
            if(argc <= ++OptIndex){
                posn = "";
            }
    }
}

```

```

        return Unknown;
    } else
        OptArg = argv[OptIndex];

    posn = "";
    OptIndex++;
}
return letter;
}

```

[مثال 2]

10.3 برنامج لإيجاد الأنماط

نقدّم في هذا القسم برنامجًا كاملًا يستخدم أحرف الخيار مثل وسطاء للبرنامج بهدف التحكم بطريقة عمله. يُعالج البرنامج أولاً أي وسيط يمثل خيارًا، ويُحفظ الوسيط الأول -ليس خيار- بكونه "سلسلة بحث نصية search string"، في حين تُستخدم أي وسطاء أخرى متبقية لتحديد أسماء الملفات التي يجب أن تُقرأ دخلاً للبرنامج، وإذا لم يُعثر على أي اسم ملف فسيقرأ البرنامج من دخله القياسي بدلاً من ذلك، وإذا وُجد تطابق لسلسلة البحث النصية ضمن سطر من نص الدخل، يُطبع كامل السطر على الخرج القياسي. تُستخدم الدالة options لمعالجة جميع أحرف الخيار المزودة للبرنامج، ويميّز برنامجنا هنا خمسة خيارات، هي: -c و -i و -l و -n و -v، ولا يُشترط لأي من الخيارات السابقة أن تُتبع بوسيط اختياري. يحدد الخيار -أو الخيارات- سلوك البرنامج عند تشغيله على النحو التالي:

- الخيار -c: يطبع البرنامج عدد الأسطر الكلية الموافقة لسلسلة البحث النصية التي عُثر عليها في ملف - أو ملفات- الدخل، ولا تُطبع أي أسطر نصية.
 - الخيار -i: تُتجاهل حالة الأحرف لكل من سطر ملف الدخل وسلسلة البحث النصية عند البحث عن تطابق بينها.
 - الخيار -l: يُطبع كل سطر نصي على الخرج مسبقًا برقم السطر المفحوص في ملف الدخل الحالي.
 - الخيار -n: يُطبع كل سطر نصي على الخرج مسبقًا باسم الملف الذي يحتوي هذا السطر.
 - الخيار -v: يطبع البرنامج الأسطر فقط دون مطابقة سلسلة البحث النصية المزودة.
- يُعيد البرنامج بعد الانتهاء من تنفيذه حالةً تدل على واحدة من الحالات التالية:
- الحالة EXIT_SUCCESS: عُثر على تطابق واحد على الأقل.
 - الحالة EXIT_FAILURE: لم يُعثر على أي تطابق، أو حدث خطأ ما.

يعتمد البرنامج جدًّا على دوال المكتبة القياسية لإنجاز الجزء الأكبر من العمل، فعلى سبيل المثال تُعالج جميع الملفات باستخدام دوال `stdio`. لاحظ اعتماد جوهر البرنامج أيضًا على مطابقة السلاسل النصية باستخدام استدعاءات لدالة `strstr`.

إليك الشيفرة البرمجية الكاملة الخاصة بالبرنامج. حتى يعمل البرنامج، عليك طبقًا لتصريفه مع الشيفرة البرمجية الخاصة به التي تعالج أحرف الخيارات، والتي استعرضناها سابقًا.

```

/*
برنامج بسيط يطبع الأسطر من ملف نصي بحيث يحوي ذلك السطر الكلمة المزودة في سطر الأوامر
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/*
تصاريح لبرنامج الأنماط
*
*/

#define CFLAG 0x001 // احص عدد الأسطر المتطابقة فقط
#define IFLAG 0x002 // تجاهل حالة الأحرف
#define LFLAG 0x004 // اعرض رقم السطر
#define NFLAG 0x008 // اعرض اسماء ملفات الدخل
#define VFLAG 0x010 // اعرض السطور التي لا تتطابق

extern int OptIndex; // الدليل الحالي للمصفوفة argv[]
extern char *OptArg; // مؤشر وسيط الخيار العام

/*
* main() الدالة إلى الأوامر إلى الدالة
*/

int options(int, char **, const char *);

/*

```

```
تسجيل الخيارات المطلوبة للتحكم بسلوك البرنامج
*/

unsigned set_flags(int, char **, const char *);

/*
تفقد كل سطر من الدخل لحالة المطابقة
*/

int look_in(const char *, const char *, unsigned);

/*
اطبع سطرًا من ملف الدخل إلى الخرج القياسي بالتنسيق المُحدد بواسطة خيارات سطر الأوامر
*/

void print_line(unsigned mask, const char *fname,
                int lno, const char *text);

static const char
    /* الخيارات الممكنة للنمط */
    *OptString = "cilnv",
    /* الرسالة التي ستعرض عندما تُدخل الخيارات بصورة غير صحيحة */
    *errmssg = "usage: pattern [-cilnv] word [filename]\n";

int main(int argc, char *argv[])
{
    unsigned flags = 0;
    int success = 0;
    char *search_string;

    if(argc < 2){
        fprintf(stderr, errmssg);
        exit(EXIT_FAILURE);
    }
}
```

```

flags = set_flags(argc, argv, OptString);

if(argv[OptIndex])
    search_string = argv[OptIndex++];
else {
    fprintf(stderr, errmssg);
    exit(EXIT_FAILURE);
}

if(flags & IFLAG){
    /* تجاهل حالة الحرف والتعامل فقط مع الأحرف الصغيرة */
    char *p;
    for(p = search_string ; *p ; p++)
        if(isupper(*p))
            *p = tolower(*p);
}

if(argv[OptIndex] == NULL){
    // stdin لم يُزود أي اسم ملف، لذا نستخدم stdin
    success = look_in(NULL, search_string, flags);
} else while(argv[OptIndex] != NULL)
    success += look_in(argv[OptIndex++],
                      search_string, flags);

if(flags & CFLAG)
    printf("%d\n", success);

exit(success ? EXIT_SUCCESS : EXIT_FAILURE);
}

unsigned set_flags(int argc, char **argv, const char *opts)
{
    unsigned flags = 0;
    int ch = 0;

    while((ch = options(argc, argv, opts)) != -1){

```

```
        switch(ch){
            case 'c':
                flags |= CFLAG;
                break;
            case 'i':
                flags |= IFLAG;
                break;
            case 'l':
                flags |= LFLAG;
                break;
            case 'n':
                flags |= NFLAG;
                break;
            case 'v':
                flags |= VFLAG;
                break;
            case '?':
                fprintf(stderr, errmssg);
                exit(EXIT_FAILURE);
        }
    }
    return flags;
}

int look_in(const char *infile, const char *pat, unsigned flgs)
{
    FILE *in;
    /*
    يخزن line[0] سطر الدخل كما يُقرأ
    بينما يحول line[1] السطر إلى حالة أحرف صغيرة إن لزم الأمر
    */
    char line[2][BUFSIZ];
    int lineno = 0;
    int matches = 0;
```

```

    if(infile){
        if((in = fopen(infile, "r")) == NULL){
            perror("pattern");
            return 0;
        }
    } else
        in = stdin;

    while(fgets(line[0], BUFSIZ, in)){
        char *line_to_use = line[0];
        lineno++;
        if(flgs & IFLAG){
            /* حالة تجاهل */
            char *p;
            strcpy(line[1], line[0]);
            for(p = line[1] ; *p ; *p++){
                if(isupper(*p))
                    *p = tolower(*p);
            }
            line_to_use = line[1];
        }

        if(strstr(line_to_use, pat)){
            matches++;
            if(!(flgs & VFLAG))
                print_line(flgs, infile, lineno,
line[0]);
        } else if(flgs & VFLAG)
            print_line(flgs, infile, lineno, line[0]);
    }
    fclose(in);
    return matches;
}

void print_line(unsigned mask, const char *fname,
                int lnno, const char *text)
{
    if(mask & CFLAG)

```

```

        return;
    if(mask & NFLAG)
        printf("%s:", *fname ? fname : "stdin");
    if(mask & LFLAG)
        printf(" %d :", lno);
    printf("%s", text);
}

```

[مثال 3]

10.4 مثال أكثر طموحا

أخيرًا نقدّم هنا مجموعةً من البرامج المصمّمة للتلاعب بملف بيانات وحيد والتعامل معه بطريقة مترابطة وسليمة؛ إذ تهدف هذه البرامج لمساعدتنا بتتبع نتائج عدة لاعبين يتنافسون مع بعضهم بعضًا في لعبة ما، مثل الشطرنج، أو الإسكواش على سبيل المثال.

يملك كل لاعب تصنيفًا من واحد إلى n ، إذ تمثل n عدد اللاعبين الكلي وواحد تصنيف أعلى لاعب. يستطيع اللاعبون من تصنيف منخفض تحدي لاعبين آخرين فوق تصنيفهم وينتقل اللاعب إلى تصنيف اللاعب الآخر الأعلى منه إذا انتصر عليه، وفي هذه الحالة يُنقل اللاعب الخاسر وأي لاعبين آخرين بين اللاعب الخاسر والفائز إلى تصنيف واحد أقل، وتبقى التصنيفات مثل ما هي إن لم ينتصر اللاعب الأقل تصنيفًا.

لتقديم بعض الضوابط للتوازن في التصنيف، يمكن لأي لاعب أن يتحدى لاعبًا أعلى منه تصنيفًا، إلا أن التحديات مع اللاعبين ذوي التصنيف الذي يزيد عن ثلاثة أو أقل مراتب هي الوحيدة التي ستسمح لهذا اللاعب بالتقدم في التصنيف، وهذا من شأنه أن يُجبر اللاعبين الجدد المضافين إلى أسفل التصنيف أن يلعبوا أكثر من لعبة واحدة للوصول إلى أعلى التصنيف.

هناك ثلاث مهام أساسية يجب تنفيذها للمحافظة على تتبع سليم لنتائج التصنيفات:

- طباعة التصنيف.
- إضافة لاعبين جدد.
- تسجيل النتائج.

سيأخذ تصميم برنامجنا هنا صورة ثلاثة برامج جزئية لتنفيذ كل واحدة من هذه المهام على حدة، ومن الواضح بعد اتخاذنا لهذا القرار أن هناك عددًا من العمليات التي يحتاجها كل برنامج على نحوٍ مشترك بين البرامج الثلاث؛ فعلى سبيل المثال، ستحتاج البرامج الثلاثة إلى قراءة سجلات اللاعب من ملف البيانات وستحتاج اثنان من البرامج على الأقل لكتابة سجلات اللاعب إلى ملف البيانات.

قد يكون الخيار الجيد هنا هو تصميم مكتبة من الدوال التي تتلاعب بسجلات اللاعبين وملف البيانات، واستخدام هذه المكتبة مع البرامج الثلاثة للتعامل مع تصنيف اللاعبين، إلا أننا بحاجة تعريف هيكل البيانات الذي سيمثل سجلات اللاعب قبل ذلك. تتألف المعلومات الدنيا اللازمة لإنشاء سجل لكل لاعب من اسمه وتصنيفه، إلا أننا سنحتفظ بعدد التحديات التي فاز بها اللاعب، إضافةً للتحديات التي خسرها وآخر لعبة لعبها لمنح بعض الإمكانات الإحصائية عند تشكيل لأحة التصنيف، ومن الواضح أن هذه المجموعة من المعلومات يجب تخزينها في هيكل ما.

نجد التصريح عن هيكل اللاعب والتصريح عن دوال مكتبة اللاعب في ملف الترويسة `player.h`، ونُخزّن البيانات في ملف البيانات مثل أسطر نصية، إذ يشير كل سطر إلى معلومات لاعب معين. يتطلب ذلك إجراء تحويلات الدخل والخرج، لكنها تقنية مفيدة إذا لم تكلف هذه التحويلات أداءً إضافيًا.

```

/*
 *
 * التصاريح والتعاريف للدوال التي تتلاعب بسجلات اللاعب بناءً على ترتيبهم
 *
 */

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NAMELEN 12          /* الطول الأعظمي لاسم اللاعب */

#define LENBUF 256         /* الطول الأعظمي لذاكرة الدخل المؤقتة */

#define CHALLENGE_RANGE 3 // عدد اللاعبين الأعلى تصنيفًا الذين من الممكن للاعب
أن يتحداهم ليزيد تصنيفه

extern char *OptArg;

typedef struct {
    char    name[NAMELEN+1];
    int     rank;
    int     wins;
    int     losses;

```

```

        time_t last_game;
    } player;

#define NULLPLAYER (player *)0

extern const char *LadderFile;

extern const char *WrFmt;          /* يُستخدم عند كتابة السجلات */
extern const char *RdFmt;          /* يُستخدم عند قراءة السجلات */

/*
تصاريح البرامج التي تُستخدم للتلاعب بسجلات اللاعب وملف لائحة التصنيف المعرفة في ملف
player.c
*/

int    valid_records(FILE *);
int    read_records(FILE *, int, player *);
int    write_records(FILE *, player *, int);
player *find_by_name(char *, player *, int);
player *find_by_rank(int, player *, int);
void   push_down(player *, int, int, int);
int    print_records(player *, int);
void   copy_player(player *, player *);
int    compare_name(player *, player *);
int    compare_rank(player *, player *);
void   sort_players(player *, int);

```

[مثال 4]

إليك شيفرة ملف `player.c` الذي يستخدم بعض الدوال العامة للتلاعب بسجلات اللاعبين وملف البيانات، ويمكن أن تُستخدم هذه الدوال مع برامج أخرى محدد لتشكل ثلاثة برامج تتعامل مع لائحة النتائج. لاحظ أنه يجب على كل برنامج أن يقرأ كامل البيانات من الملف إلى مصفوفة ديناميكية حتى نستطيع التلاعب بسجلات اللاعبين، ومن المفترض أن تكون جميع السجلات المحتواة داخل المصفوفة مُرتبةً حسب التصنيف قبل كتابتها مجدداً إلى ملف البيانات، وستؤدّ الدالة `push_down` بعض النتائج المثيرة للاهتمام إن لم تكن السجلات مرتبة.


```

/*
 * الدوال الاعتيادية المستخدمة للتلاعب ببيانات ملف لائحة النتائج وسجلات اللاعبين
 */

#include "player.h"

const char *LadderFile = "ladder";

const char *WrFmt = "%s %d %d %d %ld\n";
const char *RdFmt = "%s %d %d %d %ld";

/* تنبيه المستخدم بخصوص ضمّ السلاسل النصية */
const char *HeaderLine =
    "Player Rank Won Lost Last Game\n"
    "=====\\n";

const char *PrtFmt = "%-12s%4d %4d %4d %s\\n";

/* إعادة رقم السجلات الموجودة في الملف */

int valid_records(FILE *fp)
{
    int i = 0;
    long plrs = 0L;
    long tmp = ftell(fp);
    char buf[LENBUF];

    fseek(fp, 0L, SEEK_SET);

    for(i = 0; fgets(buf, LENBUF, fp) != NULL ; i++)
        ;

    /* استعادة مؤشر الملف إلى حالته الأصلية */

    fseek(fp, tmp, SEEK_SET);

```

```
        return i;
    }

    // قراءة القيمة num من سجل اللاعب من الملف fp إلى المصفوفة them

int read_records(FILE *fp, int num, player *them)
{
    int i = 0;
    long tmp = ftell(fp);

    if(num == 0)
        return 0;

    fseek(fp, 0L, SEEK_SET);

    for(i = 0 ; i < num ; i++){
        if(fscanf(fp, RdFmt, (them[i]).name,
                    &((them[i]).rank),
                    &((them[i]).wins),
                    &((them[i]).losses),
                    &((them[i]).last_game)) != 5)
            break;           // خطأ عند fscanf
    }

    fseek(fp, tmp, SEEK_SET);
    return i;
}

// كتابة num الخاص بسجل اللاعب إلى الملف fp من المصفوفة them

int write_records(FILE *fp, player *them, int num)
{
    int i = 0;

    fseek(fp, 0L, SEEK_SET);
```

```

    for(i = 0 ; i < num ; i++){
        if(fprintf(fp, WrFmt, (them[i]).name,
                    (them[i]).rank,
                    (them[i]).wins,
                    (them[i]).losses,
                    (them[i]).last_game) < 0)
            break;          // خطأ عند fprintf
    }

    return i;
}

/*
إعادة مؤشر يشير إلى اللاعب في المصفوفة them ذو اسم مطابق للقيمة name
*/

player *find_by_name(char * name, player *them, int num)
{
    player *pp = them;
    int i = 0;

    for(i = 0; i < num; i++, pp++)
        if(strcmp(name, pp->name) == 0)
            return pp;

    return NULLPLAYER;
}

/*
إعادة مؤشر يشير إلى لاعب في مصفوفة them تُطابق رتبته القيمة rank
*/

player *find_by_rank(int rank, player *them, int num)
{
    player *pp = them;
    int i = 0;

```

```

    for(i = 0; i < num; i++, pp++)
        if(rank == pp->rank)
            return pp;

    return NULLPLAYER;
}

/*
خفض رتبة جميع اللاعبين في مصفوفة them إذا كانت رتبتهم بين start و end
*/

void push_down(player *them, int number, int start, int end)
{
    int i;
    player *pp;

    for(i = end; i >= start; i--){
        if((pp = find_by_rank(i, them, number)) == NULLPLAYER){
            fprintf(stderr,
                "error: could not find player ranked %d\n",
                i);
            free(them);
            exit(EXIT_FAILURE);
        } else
            (pp->rank)++;
    }
}

// طباعة سجل اللاعب num بصورة مُنسقة من المصفوفة them

int print_records(player *them, int num)
{
    int i = 0;

    printf(HeaderLine);

    for(i = 0 ; i < num ; i++){

```

```

        if(printf(PrtFmt,
                (them[i]).name, (them[i]).rank,
                (them[i]).wins, (them[i]).losses,
                asctime(localtime(&(them[i]).last_game))) < 0)
           break;          /* error on printf! */
    }

    return i;
}

/* نسخ القيم من لاعب إلى آخر */

void copy_player(player *to, player *from)
{
    if((to == NULLPLAYER) || (from == NULLPLAYER))
        return;

    *to = *from;
    return;
}

/* مقارنة اسم اللاعب الأول مع اسم اللاعب الثاني */

int compare_name(player *first, player *second)
{
    return strcmp(first->name, second->name);
}

/* مقارنة رتبة اللاعب الأول مع رتبة اللاعب الثاني */

int compare_rank(player *first, player *second)
{
    return (first->rank - second->rank);
}

// ترتيب num الذي يدل على سجل اللاعب في المصفوفة them

```

```
void sort_players(player *them, int num)
{
    qsort(them, num, sizeof(player), compare_rank);
}
```

[مثال 5]

صُرِّفت الشيفرة السابقة عند تجربتها إلى كائن ملف `object file`، الذي كان مربوطًا (مع كائن ملف يحتوي على الشيفرة البرمجية الخاصة بالدالة `options`) بواحدٍ من البرامج الثلاثة الخاصة بالتعامل مع لائحة النتائج.

إليك الشيفرة البرمجية لواحدة من أبسط البرامج هذه، ألا وهو `"showladder"`، الذي يحتوي على الملف `"showladder.c"`. يأخذ هذا البرنامج خيارًا واحدًا وهو `-f` وقد تلاحظ أن هذا الخيار يأخذ وسيطًا اختياريًا أيضًا، والهدف من هذا الوسيط هو السماح بطباعة ملف بيانات لائحة التصنيف باستخدام اسم مغير للاسم الافتراضي `.ladder`.

يجب أن تُخزّن سجلات اللاعب في ملف البيانات قبل ترتيبها، إلا أن `showladder` يرتبها قبل أن يطبعها فقط بهدف التأكد.

```
/*
برنامج يطبع حالة لائحة النتائج الحالية
*/

#include "player.h"

const char *ValidOpts = "f:";

const char *Usage = "usage: showladder [-f ladder_file]\n";

char *OtherFile;

int main(int argc, char *argv[])
{
    int number;
    char ch;
    player *them;
    const char *fname;
    FILE *fp;
```

```
if(argc == 3){
    while((ch = options(argc, argv, ValidOpts)) != -1){
        switch(ch){
            case 'f':
                OtherFile = OptArg;
                break;
            case '?':
                fprintf(stderr, Usage);
                break;
        }
    }
} else if(argc > 1){
    fprintf(stderr, Usage);
    exit(EXIT_FAILURE);
}

fname = (OtherFile == 0)? LadderFile : OtherFile;
fp = fopen(fname, "r+");

if(fp == NULL){
    perror("showlldr");
    exit(EXIT_FAILURE);
}

number = valid_records (fp);

them = (player *)malloc((sizeof(player) * number));

if(them == NULL){
    fprintf(stderr, "showlldr: out of memory\n");
    exit(EXIT_FAILURE);
}

if(read_records(fp, number, them) != number){
    fprintf(stderr, "showlldr: error while reading"
        " player records\n");
}
```

```

        free(them);
        fclose(fp);
        exit(EXIT_FAILURE);
    }

    fclose(fp);

    sort_players(them, number);

    if(print_records(them, number) != number){
        fprintf(stderr, "showlldr: error while printing"
                    " player records\n");

        free(them);
        exit(EXIT_FAILURE);
    }

    free(them);
    exit(EXIT_SUCCESS);
}

```

[مثال 6]

يعمل البرنامج "showlldr" فقط إذا كان هناك ملف بيانات يحتوي على سجلات اللاعب بالتنسيق الصحيح، ويُنشئ البرنامج "newplyr" ملفاً إذا لم يكن هناك أي ملف مُسبقاً ومن ثم يضيف بيانات اللاعب الجديد بالتنسيق الصحيح إلى ذلك الملف.

يُدرج اللاعبون الجدد عادةً أسفل التصنيف إلا أن هناك بعض الحالات الاستثنائية التي يسمح فيها "newplyr" بإدراج اللاعبين وسط التصنيف.

يجب أن يظهر اللاعب مرةً واحدةً في التصنيف (إلا إذا تشابهت أسماء اللاعبين المستعارة) ويجب أن يكون لكل تصنيف لاعب واحد فقط، ولذا فإن البرنامج يفحص الإدخالات المتكررة وإذا كان من الواجب إدخال اللاعب الجديد إلى تصنيف مجاور لتصنيف اللاعبين الآخرين، يُزاح اللاعبون بعيداً عن تصنيف اللاعب الجديد.

يتعرّف البرنامج "newplyr" على الخيار -f بصورةً مشابهة للبرنامج "showlldr"، ويفسره على أنه طلب إضافة اللاعب الجديد إلى ملف يُسمى باستخدام وسيط الخيار بدلاً من اسم الملف الافتراضي ألا وهو "ladder". يتطلب البرنامج "newplyr" أيضاً خيارين إضافيين ألا وهما -n و -r ويحدد كل وسيط خيار اسم اللاعب الجديد وتصنيفه الأولي بالترتيب.


```
/*
برنامج يُضيف لاعب جديد إلى لائحة التصنيفات، ويفترض أن تُسند رتبةً بقيمة واقعية إلى اللاعب
*/

#include "player.h"

const char *ValidOpts = "n:r:f:";

char *OtherFile;

static const char *Usage = "usage: newplyr -r rank -n name [-f file]\n";

/* تصاريح مسبقة للدوال المعرفة في هذا الملف */

void record(player *extra);

int main(int argc, char *argv[])
{
    char ch;
    player dummy, *new = &dummy;

    if(argc < 5){
        fprintf(stderr, Usage);
        exit(EXIT_FAILURE);
    }

    while((ch = options(argc, argv, ValidOpts)) != -1){
        switch(ch){
            case 'f':
                OtherFile=OptArg;
                break;
            case 'n':
                strncpy(new->name, OptArg, NAMELEN);
                new->name[NAMELEN] = 0;
                if(strcmp(new->name, OptArg) != 0)
                    fprintf(stderr,
```

```

                                "Warning: name truncated to
%s\n", new->name);
        break;
    case 'r':
        if((new->rank = atoi(OptArg)) == 0){
            fprintf(stderr, Usage);
            exit(EXIT_FAILURE);
        }
        break;
    case '?':
        fprintf(stderr, Usage);
        break;
    }
}

if((new->rank == 0)){
    fprintf(stderr, "newplyr: bad value for rank\n");
    exit(EXIT_FAILURE);
}

if(strlen(new->name) == 0){
    fprintf(stderr,
                                "newplyr: needs a valid name for new player\
n");
    exit(EXIT_FAILURE);
}

new->wins = new->losses = 0;
time(& new->last_game); // أسند الوقت الحالي إلى last_game

record(new);

exit(EXIT_SUCCESS);
}

void record(player *extra)
{

```

```
int number, new_number, i;
player *them;
const char *fname =(OtherFile==0)?LadderFile:OtherFile;
FILE *fp;

fp = fopen(fname, "r+");

if(fp == NULL){
    if((fp = fopen(fname, "w")) == NULL){
        perror("newplyr");
        exit(EXIT_FAILURE);
    }
}

number = valid_records (fp);
new_number = number + 1;

if((extra->rank <= 0) || (extra->rank > new_number)){
    fprintf(stderr,
        "newplyr: rank must be between 1 and %d\n",
        new_number);
    exit(EXIT_FAILURE);
}

them = (player *)malloc((sizeof(player) * new_number));

if(them == NULL){
    fprintf(stderr,"newplyr: out of memory\n");
    exit(EXIT_FAILURE);
}

if(read_records(fp, number, them) != number){
    fprintf(stderr,
        "newplyr: error while reading player records\n
n");

    free(them);
    exit(EXIT_FAILURE);
}
```

```

    }

    if(find_by_name(extra->name, them, number) != NULLPLAYER){
        fprintf(stderr,
            "newplyr: %s is already on the ladder\n",
            extra->name);
        free(them);
        exit(EXIT_FAILURE);
    }

    copy_player(&them[number], extra);

    if(extra->rank != new_number)
        push_down(them, number, extra->rank, number);

    sort_players(them, new_number);

    if((fp = freopen(fname, "w+", fp)) == NULL){
        perror("newplyr");
        free(them);
        exit(EXIT_FAILURE);
    }

    if(write_records(fp, them, new_number) != new_number){
        fprintf(stderr,
            "newplyr: error while writing player records\n");
        fclose(fp);
        free(them);
        exit(EXIT_FAILURE);
    }
    fclose(fp);
    free(them);
}

```

[مثال 7]

البرنامج الأخير المطلوب هو البرنامج الذي يسجل نتائج الألعاب، ألا وهو برنامج "result".

يقبل "result" خيار -f كما هو الحال في البرنامجين الآخرين، مصحوبًا باسم الملف لتحديد بديل عن اسم ملف اللاعب الافتراضي.

يعرض برنامج "result" عملية الإدخال للاعبين الخاسرين والرابحين على نحوٍ تفاعلي على عكس برنامج "newplyr"، ويصرّ على أن الأسماء يجب أن تكون للاعبين موجودين مسبقًا. يجري التَحَقُّق من الأسماء بعد إعطاء اسمين صالحين، وذلك فيما إذا كان الخاسر أعلى تصنيفًا من الفائز، أو أن الفائز ذو تصنيفٍ مقارب مما يسمح له بتغيير تصنيفه؛ وإذا لُزم تغيير التصنيف، يأخذ المنتصر رتبة الخاسر ويُخفِّض الخاسر رتبةً واحدةً (إضافةً إلى أي لاعب من تصنيف متداخل).

إليك الشيفرة البرمجية الخاصة ببرنامج result.

```

/*
 * برنامج يسجل النتائج
 *
 */

#include "player.h"

/* تصريحات استباقية للدوال المعرفة في هذا الملف */
char *read_name(char *, char *);
void move_winner(player *, player *, player *, int);

const char *ValidOpts = "f:";

const char *Usage = "usage: result [-f file]\n";

char *OtherFile;

int main(int argc, char *argv[])
{
    player *winner, *loser, *them;
    int number;
    FILE *fp;
    const char *fname;
    char buf[LENBUF], ch;

    if(argc == 3){

```

```
while((ch = options(argc, argv, ValidOpts)) != -1){
    switch(ch){
        case 'f':
            OtherFile = OptArg;
            break;
        case '?':
            fprintf(stderr, Usage);
            break;
    }
}
} else if(argc > 1){
    fprintf(stderr, Usage);
    exit(EXIT_FAILURE);
}

fname = (OtherFile == 0)? LadderFile : OtherFile;
fp = fopen(fname, "r+");
if(fp == NULL){
    perror("result");
    exit(EXIT_FAILURE);
}

number = valid_records (fp);

them = (player *)malloc((sizeof(player) * number));

if(them == NULL){
    fprintf(stderr, "result: out of memory\n");
    exit(EXIT_FAILURE);
}

if(read_records(fp, number, them) != number){
    fprintf(stderr,
        "result: error while reading player records\n");
    fclose(fp);
    free(them);
}
```

```

        exit(EXIT_FAILURE);
    }

    fclose(fp);
    if((winner = find_by_name(read_name(buf, "winner"), them,
number))
        == NULLPLAYER){
        fprintf(stderr, "result: no such player %s\n", buf);
        free(them);
        exit(EXIT_FAILURE);
    }

    if((loser = find_by_name(read_name(buf, "loser"), them,
number))
        == NULLPLAYER){
        fprintf(stderr, "result: no such player %s\n", buf);
        free(them);
        exit(EXIT_FAILURE);
    }

    winner->wins++;
    loser->losses++;
    winner->last_game = loser->last_game = time(0);

    if(loser->rank < winner->rank)
        if((winner->rank - loser->rank) <= CHALLENGE_RANGE)
            move_winner(winner, loser, them, number);

    if((fp = freopen(fname, "w+", fp)) == NULL){
        perror("result");
        free(them);
        exit(EXIT_FAILURE);
    }
    if(write_records(fp, them, number) != number){
        fprintf(stderr, "result: error while writing player
records\n");
        free(them);

```

```

        exit(EXIT_FAILURE);
    }
    fclose(fp);
    free(them);
    exit(EXIT_SUCCESS);
}

void move_winner(player *ww, player *ll, player *them, int number)
{
    int loser_rank = ll->rank;
    if((ll->rank - ww->rank) > 3)
        return;
    push_down(them, number, ll->rank, (ww->rank - 1));
    ww->rank = loser_rank;
    sort_players(them, number);
    return;
}

char *read_name(char *buf, char *whom)
{
    for(;;){
        char *cp;
        printf("Enter name of %s : ",whom);
        if(fgets(buf, LENBUF, stdin) == NULL)
            continue;
        /* حذف السطر الجديد */
        cp = &buf[strlen(buf)-1];
        if(*cp == '\n')
            *cp = 0;
        /* محرف واحد على الأقل */
        if(cp != buf)
            return buf;
    }
}

```

[مثال 8]

10.5 الخاتمة

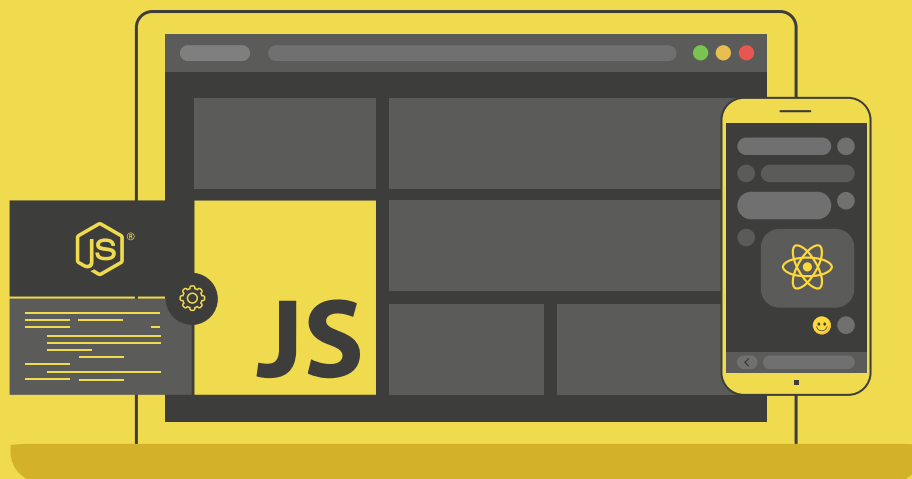
يجب أن تقدم لك البرامج المعروضة في هذا الفصل فهماً أعمق لبرامج منتصف الطريق التي يمكن أن تُنجز باستخدام لغة سي C والمكتبات المعرفة في المعيار.

ما الذي نقصده بكلمة "منتصف الطريق"؟ ببساطة، صمّمنا وطبقنا وفحصنا ووثّقنا البرامج بحيث تناسب البرامج الصغيرة المحتواة داخل بعضها بعضاً، والتي لا تتطلب مرحلةً متقدمةً من الوثوقية والمتانة، إذ لا تتطلب الكثير من البرامج هذه المعايير، وفعل هذا يعني تعقيد الأمور على نحوٍ غير ضروري، ويعتمد الأمر بطبيعة الحال على الهدف النهائي من البرنامج.

هناك بعض الحالات التي لا بدّ فيها تحقيق بعض الأمور المتطلّبة، وهذا النوع من البرامج مُهندَس بصورةٍ دقيقة ويتطلب قدرًا أكبر بكثير من الجهد سواءً بهدف الفحص أو التجربة، إضافةً إلى التحكم بإمكانية الوصول إلى الشيفرة المصدرية على عكس متطلبات بعض البرامج البسيطة التي استعرضناها. تبدو الشيفرة المصدرية لهذا النوع من البرامج مختلفةً كثيرًا، فعلى الرغم من أنها قد تستخدم لغة البرمجة ذاتها إلا أنها تعتمد أكثر على التحقق من الأخطاء وتصحيحها، ونحن لم ننظر إلى هذا النوع من البرامج إطلاقًا هنا.

بغض النظر عن البيئة التي تعمل بها، نتمنى أن هذا الكتاب قد ساعدك في فهم لغة سي. حظًا موفقًا.

دورة تطوير التطبيقات باستخدام لغة JavaScript



احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقاً من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن



11. حلول التمارين

11.1 الفصل الأول

11.1.1 التمرين الثاني

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int this_number, divisor, not_prime;
    int last_prime;

    this_number = 3;
    last_prime = 3;

    printf("1, 3 is a prime pair\n");

    while(this_number < 10000){
        divisor = this_number / 2;
        not_prime = 0;
        while(divisor > 1){
            if(this_number % divisor == 0){
                not_prime = 1;
                divisor = 0;
            }
        }
    }
}
```

```

    }
    else
        divisor = divisor-1;
}

if(not_prime == 0){
    if(this_number == last_prime+2)
        printf("%d, %d is a prime pair\n",
            last_prime, this_number);
        last_prime = this_number;
}
this_number = this_number + 1;
}
exit(EXIT_SUCCESS);
}

```

11.1.2 التمرين الثالث

```

#include <stdio.h>
#include <stdlib.h>

main(){
    printf("Type in a string: ");
    printf("The value was: %d\n", getnum());
    exit(EXIT_SUCCESS);
}

getnum(){
    int c, value;;
    value = 0;
    c = getchar();
    while(c != '\n'){
        value = 10*value + c - '0';
        c = getchar();
    }
    return (value);
}

```

11.1.3 التمرين الرابع

```
#include <stdio.h>
#include <stdlib.h>

/* حجم المصفوفة */
#define NUMBER 10

main(){
    int arr[NUMBER], count, lo, hi;

    count = 0;
    while(count < NUMBER){
        printf("Type in a string: ");
        arr[count] = getnum();
        count = count+1;
    }
    lo = 0;
    while(lo < NUMBER-1){
        hi = lo+1;
        while(hi < NUMBER){
            int tmp;
            if(arr[lo] > arr[hi]){
                tmp = arr[lo];
                arr[lo] = arr[hi];
                arr[hi] = tmp;
            }
            hi = hi + 1;
        }
        lo = lo + 1;
    }
    /* طباعة العناصر */
    count = 0;
    while(count < NUMBER){
        printf("%d\n", arr[count]);
        count = count+1;
    }
}
```

```

    }
    exit(EXIT_SUCCESS);
}

getnum(){
    int c, value;;

    value = 0;
    c = getchar();
    while(c != '\n'){
        value = 10*value + c - '0';
        c = getchar();
    }
    return (value);
}

```

11.1.4 التمرين الخامس

```

#include <stdio.h>
#include <stdlib.h>

/*
علينا بناء مصفوفة من المحارف لطباعة قيمة int بالنظام الثنائي والست عشري والعشري بالترتيب
توجد القيم على أساس أقل الخانات أهمية أولاً، وتطبع أكثر الخانات أهمية أولاً.
*/
#define NDIG    32      /* افتراض العدد الأعظمي من الخانات */

int getnum(void);

main(){
    int val, i, count;
    char chars[NDIG];

    i = getnum();

    /* الطباعة في النظام الثنائي */
    val = i;

```

```
count = 0;
do{
    chars[count] = val % 2;
    val = val / 2;
    count = count + 1;
}while(val);
count = count - 1; /* تزايدت للتو في الأعلى */

while(count >= 0){
    printf("%d", chars[count]);
    count = count - 1;
}
printf("\n");

/* الطباعة في النظام العشري */
val = i;
count = 0;
do{
    chars[count] = val % 10;
    val = val / 10;
    count = count + 1;
}while(val);
count = count - 1; /* تزايدت للتو في الأعلى */

while(count >= 0){
    printf("%d", chars[count]);
    count = count - 1;
}
printf("\n");

/* الطباعة بالنظام الست عشري */
val = i;
count = 0;
do{
    chars[count] = val % 16;
    val = val / 16;
```

```

        count = count + 1;
    }while(val);
    count = count - 1; /* تزايدت للتو أعلاه */

    while(count >= 0){
        if(chars[count] < 10)
            printf("%d", chars[count]);
        else{
            // بفرض `A` إلى `F` على نحوٍ تتابعي
            chars[count] = chars[count]-10+'A';
            printf("%c", chars[count]);
        }
        count = count - 1;
    }
    printf("\n");
    exit(EXIT_SUCCESS);
}

getnum(){
    int c, value;;

    value = 0;
    c = getchar();
    while(c != '\n'){
        value = 10*value + c - '0';
        c = getchar();
    }
    return (value);
}

```

11.2 الفصل الثاني

11.2.1 التمرين الأول

تُستخدم ثلاثيات المحارف trigraphs عند استخدام جهاز الدخل أو مجموعة محارف نظام الاستضافة الأصيلة host system's native character set التي لا تدعم محارف مختلفة كافية لتمثيل محارف لغة سي.

11.2.2 التمرين الثاني

لن تُستخدم ثلاثيات المحارف في نظام يحتوي على محارف مختلفة كافية لحجز محرف منفصل لكل من رموز لغة سي؛ وللحصول على قابلية نقل أفضل، قد نرى استخدام تمثيل ثلاثيات المحارف في برنامج بلغة سي عند توزيعه للسماح بمعظم الأنظمة التي لا تستخدم آسكي ASCII بقراءة البيانات المُرمزة بالآسكي وترجمتها إلى ترميزها الأصلي، ومن ثم تُصَرَّف باستخدام مصرّف لغة سي قياسي مباشرةً.

11.2.3 التمرين الثالث

لا تتكافأ محارف المسافة الفارغة مع بعضها داخل السلاسل النصية وثوابت المحارف، إذ أن محرف السطر الجديد له معنى مميز في المعالج المُسبق preprocessor.

11.2.4 التمرين الرابع

لاستكمال سطر طويل، خاصةً في الأنظمة التي لا تحتوي على حدٍ أعظمي لطول السطر الفيزيائي.

11.2.5 التمرين الخامس

تُصبح متصلة.

11.2.6 التمرين السادس

لأن `/*` التي تُنهي التعليق الداخلي تُنهي أيضًا التعليق الخارجي.

11.2.7 التمرين السابع

31 محرّفًا للمتغيرات الداخلية وستة محارف للمتغيرات الخارجية، ولا يجب أن يعتمد الاسم ذو الست محارف على اختلاف الأحرف الكبيرة والصغيرة.

11.2.8 التمرين الثامن

يقدم التصريح اسمًا ونوعًا لشيء ما، إلا أنه لا يحجز بالضرورة المساحة اللازمة له.

11.2.9 التمرين التاسع

التعريف هو تصريح يحجز مساحة.

11.2.10 التمرين العاشر

الحالة المحققة دائمًا هي أن النطاق الأكبر من القيم يُمكن تخزينه في عدد كبير مُضاعف long double على الرغم من أنه قد لا يختلف عن نوع ما من أنواع الفاصلة العائمة الأصغر منه.

11.2.11 التمرين الحادي عشر

تنطبق الإجابة السابقة أيضًا على النوع ذي الدقة الأكبر: قيمة كبيرة مُضاعفة `long double`، إذ لا تسمح سي لمنقذ اللغة `language implementor` باستخدام نفس عدد البتات في القيم المُضاعفة `double` والقيم الكبيرة المُضاعفة `long double` على سبيل المثال، ومن ثم حجز بتات أكثر للدقة في نوع ما وبتات أكثر للنطاق في نوع آخر.

11.2.12 التمرين الثاني عشر

لا يوجد هناك أي مشاكل بخصوص إسناد نوع فاصلة عائمة صغير إلى نوع آخر أكبر منه.

11.2.13 التمرين الثالث عشر

قد يتسبب إسناد نوع فاصلة عائمة كبير إلى نوع أقصر منه بحدوث طفحان وسلوك غير محدد.

11.2.14 التمرين الرابع عشر

السلوك غير المحدد هو سلوك لا يمكن توقعه إطلاقًا، إذ من الممكن حدوث أي شيء، وقد يبدو في بعض الأحيان أنه ما من شيء يحدث سوى القيم الحسابية الغريبة الناتجة.

11.2.15 التمرين الخامس عشر

1. النوع `signed int` عن طريق ترقية الأعداد الصحيحة.
2. لا يمكن توقع ذلك دون معرفة التطبيق؛ فإذا كان من الممكن للنوع `int` تخزين جميع القيم الممكنة تخزينها في `unsigned char`، فهذا يعني أن النتيجة ستكون `int` عن طريق ترقية الأعداد الصحيحة أيضًا، وإلا فستكون `unsigned int`.
3. النوع `unsigned int`.
4. النوع `long`.
5. النوع `unsigned long`.
6. النوع `long`.
7. النوع `float`.
8. النوع `float`.
9. النوع `long double`.

11.2.16 التمرين السادس عشر

1. التعبير `i1 % i2`.
2. التعبير `i1 % (int)f1`.
3. إذا لم يكن أي من المعاملين operand سالبين، فهذا يعني أن الإشارة معرفةً بحسب التطبيق، وإلا فهي موجبة، وبالتالي لا يمكنك التنبؤ بالإشارة حتى لو كانت إشارتا المعاملين سالبة.
4. اثنان، نفي أحادي وطرح ثنائي.
5. التعليمة `i1 &= 0xf`.
6. التعليمة `i1 |= 0xf`.
7. التعليمة `i1 &= ~0xf`.
8. التعليمة `i1 = ((i2 >> 4) & 0xf) | ((i2 & 0xf) << 4)`.
9. لا يمكن توقُّع النتيجة، إذ من الواجب عليك عدم استخدام المتغير نفسه أكثر من مرة داخل تعبير إذا كان التعبير يغيّر من قيمته.

11.2.17 التمرين السابع عشر

1.

```
(c = (( u * f) + 2.6L);
(int = ((float) + long double);
(int = (long double));
(int);
```

قد لا تكون ترقية الأعداد الصحيحة للنوع char إلى int من النوع unsigned char بناءً على التطبيق.

2.

```
(u += (((--f) / u) % 3));
(unsigned += ((float / unsigned) % int));
(unsigned += (float % int));
(unsigned += float);
(unsigned);
```

3.

```
(i <<= (u * (- (++f))));
(int <<= (unsigned * (- float)));
(int <<= (unsigned * float));
(int <<= float);
(int);
```

تنص قوانين عوامل الإزاحة على أن المعامل الواقع على يمين العامل يجب تحويله دائمًا إلى `int`، إلا أن ذلك لا يؤثر على النتيجة التي تكون من النوع المُحدد بحسب نوع المعامل الواقع على يسار العامل دائمًا، وهذا هو الحال في مثالنا بما أننا استخدمنا عامل الإسناد.

.4

```
(u = (((i + 3) + 4) + 3.1));
```

تنص القواعد على أن التعابير الجزئية التي تحتوي على `+` يمكن أن يُعاد تجميعها اعتباطيًا طالما أنه لا يوجد أي تغيير للأنواع. والأنواع في التعبير السابق، هي:

```
(unsigned = (((int + int) + int) + double))
```

بالتالي يمكن إعادة تجميع عمليتي الجمع أقصى اليسار والعمل من اليسار:

```
(unsigned = ((int + int) + double));
(unsigned = (int + double));
(unsigned = double);
(unsigned);
```

.5

```
(u = (((3.1 + i) + 3 ) + 4));
```

راجع ما قلناه في الإجابة السابقة عن إعادة التجميع.

```
(unsigned = (((double + int) + int) + int));
```

يمكن إعادة تجميع عمليتي الجمع أقصى اليمين.

```
(unsigned = ((double + int) + int));
(unsigned = (double + int));
(unsigned = double);
(unsigned);
```

.6

```
(c = ((i << (- (--f))) & 0xf));
(char = ((int << (- (--float))) & int ));
(char = ((int << (- float)) & int ));
(char = ((int << float) & int));
(char = (int & int));
(char);
```

11.3 الفصل الثالث

11.3.1 التمرين الأول

تعطي جميعها نتيجةً صحيحةً من النوع `int` بقيمة 1 للدلالة على صواب `true` و0 للدلالة على خطأ `false`.

11.3.2 التمرين الثاني

تعطي جميعها نتيجةً صحيحةً من النوع `int` بقيمة 1 للدلالة على صواب `true` و0 للدلالة على خطأ `false`.

11.3.3 التمرين الثالث

تضمن ترتيب التقييم من اليسار إلى اليمين وتتوقف عملية التقييم حالما تصبح النتيجة الكلية ممكنة التحديد.

11.3.4 التمرين الرابع

يمكن استخدام `break` لتحويل تعليمة `switch` إلى مجموعة من الخيارات الاستثنائية.

11.3.5 التمرين الخامس

لا يوجد للتعليمة `continue` أي معنى مميز في تعليمة `switch`، ولها معنى فقط في تعليمة `do, while` أو خارجية.

11.3.6 التمرين السادس

يمكن أن يسبب استخدام `continue` داخل تعليمة `while` عدم تحديث قيمة متغير الحلقة، ويقع تفادي هذا الخطأ على عاتق المبرمج.

11.3.7 التمرين السابع

لا يمكنك استخدام تعليمة `goto` للقفز من دالة إلى أخرى وذلك لأن نطاق التسمية لا يمتد خارج الدالة التي تحتوي عليها، إلا أنه يمكنك استخدام دالة المكتبة `longjmp` التي استعرضناها في الفصل التاسع لتشكيل قفزة من دالة إلى أخرى.

11.4 الفصل الرابع

11.4.1 التمرين الأول

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int i, abs_val(int);

    for(i = -10; i <= 10; i++)
        printf("abs of %d is %d\n", i, abs_val(i));
    exit(EXIT_SUCCESS);
}

int
abs_val(int x){

    if(x < 0)
        return(-x);
    return(x);
}
```

11.4.2 التمرين الثاني

تشكل الإجابة على هذا التمرين من ملفين، ما يلي هو الملف الأول:

```
#include <stdio.h>
#include <stdlib.h>

int curr_line(void), curr_col(void);
void output(char);
```

```
main(){
    printf("line %d\n", curr_line());
    printf("column %d\n", curr_col());

    output('a');
    printf("column %d\n", curr_col());

    output('\n');
    printf("line %d\n", curr_line());
    printf("column %d\n", curr_col());
    exit(EXIT_SUCCESS);
}
```

يحتوي الملف الثاني على الدوال والمتغيرات الساكنة:

```
#include <stdio.h>

int curr_line(void), curr_col(void);
void output(char);

static int lineno=1, colno=1;

int
curr_line(void){
    return(lineno);
}

int
curr_col(void){
    return(colno);
}

void
output(char a){
    putchar(a);
    colno++;
}
```

```

if(a == '\n'){
    colno = 1;
    lineno++;
}
}

```

11.4.3 التمرين الثالث

الدالة التعاودية:

```

#include <stdio.h>
#include <stdlib.h>

void recur(void);

main(){
    recur();
    exit(EXIT_SUCCESS);
}

void
recur(void){
    static ntimes;

    ntimes++;
    if(ntimes < 100)
        recur();
    printf("%d\n", ntimes);
    ntimes--;
}

```

11.4.4 التمرين الرابع

وأخيرًا، أطول الإجابات:

```

#include <stdio.h>
#include <stdlib.h>

```



```
#define PI 3.141592
#define INCREMENT (PI/20)
#define DELTA .0001

double sine(double), cosine(double);
static unsigned int fact(unsigned int n);
static double pow(double x, unsigned int n);

main(){
    double arg = 0;

    for(arg = 0; arg <= PI; arg += INCREMENT){
        printf("value %f\t sine %f\t cosine %f\n", arg, sine(arg),
        cosine(arg));
    }
    exit(EXIT_SUCCESS);
}

static unsigned int
fact(unsigned int n){
    unsigned int answer;

    answer = 1;
    while(n > 1)
        answer *= n--;

    return(answer);
}

static double
pow(double x, unsigned int n){
    double answer;

    answer = 1;
    while(n){
        answer *= x;
        n--;
    }
}
```

```
    }
    return(answer);
}

double
sine(double x){
    double difference, thisval, lastval;
    unsigned int term;
    int sign;

    sign = -1;
    term = 3;

    thisval = x;
    do{
        lastval = thisval;
        thisval = lastval + pow(x, term)/fact(term) * sign;
        term += 2;
        sign = -sign;
        difference = thisval - lastval;
        if(difference < 0)
            difference = -difference;
    }while(difference > DELTA && term < 16);

    return(thisval);
}

double
cosine(double x){
    double difference, thisval, lastval;
    unsigned int term;
    int sign;

    sign = -1;
    term = 2;
```

```

thisval = 1;
do{
    lastval = thisval;
    thisval = lastval + pow(x, term)/fact(term) * sign;
    term += 2;
    sign = -sign;
    difference = thisval - lastval;
    if(difference < 0)
        difference = -difference;
}while(difference > DELTA && term < 16);

return(thisval);
}

```

11.5 الفصل الخامس

11.5.1 التمرين الأول

من 0 إلى 9.

11.5.2 التمرين الثاني

لا شيء، من المضمون أنه سيكون عنوان صالح ويمكن استخدامه للتحقق من مؤشر في نهاية المصفوفة.

11.5.3 التمرين الثالث

فقط عندما تشير إلى المصفوفة أو الكائن ذاته.

11.5.4 التمرين الرابع

يمكن استخدامه بأمان لتخزين قيمة المؤشر الذي يشير إلى أي نوع من الكائنات.

11.5.5 التمرين الخامس

.1

```

int
st_eq(const char *s1, const char * s2){

    while(*s1 && *s2 && (*s1 == *s2)){

```

```

    s1++; s2++;
}

return(*s1-*s2);
}

```

.2

```

const char *
find_c(char c, const char *cp){

while(*cp && *cp != c)
    cp++;
if(*cp)
    return(cp);

return(0);
}

```

.3

```

const char *
sub_st(const char *target, const char *sample){

/*
تجربة سلسلة نصية جزئية باستخدام كل محرف في sample
*/

while(*sample){
    const char *targ_p, *sample_p;

    targ_p = target;
    sample_p = sample;
    /* مقارنة السلسلة النصية */
while(*targ_p && *sample_p && (*targ_p == *sample_p)){
    targ_p++; sample_p++;
}
}
/*
إذا وصلت إلى نهاية الهدف، فلديك سلسلة فرعية *
*/
}

```

```

    */
    if(*targ_p == 0)
        return(sample);
    /* وإلا جرب مكانًا آخر */
    sample++;
}
return(0); /* في حال عدم العثور على تطابق */
}

```

11.5.6 التمرين الخامس

لا يمكن تقديم أي إجابة.

11.6 الفصل السادس

11.6.1 التمرين الأول

```

struct {
    int a,b;
};

```

11.6.2 التمرين الثاني

لا يخدم تصريح الهيكل أي استخدام دون الوسم، أو أي متغيرات أخرى معرّفة، ولا يمكن الإشارة إليه لاحقًا.

11.6.3 التمرين الثالث

```

struct int_struct{
    int a,b;
}x,y;

```

11.6.4 التمرين الرابع

```

struct int_struct z;

```

11.6.5 التمرين الخامس

```

p = &z;
p->a = 0;

```

11.6.6 التمرين السادس

مباشرةً، على سبيل المثال:

```
struct x;
```

أو ضمناً، إذا لم يوجد أي تصاريح خارجية:

```
struct x *p;
```

11.6.7 التمرين السابع

لا يُعامل مثل مؤشر بل مثل طريقة مختصرة لتهيئة عناصر المصفوفة بصورة منفردة.

11.6.8 التمرين الثامن

لا يوجد أي شيء خارج عن المألوف، إذ تُعامل السلسلة النصية مثل سلسلة نصية ثابتة من النوع

```
.const char *
```

11.6.9 التمرين التاسع

نعم، الأمر أسهل في هذه الحالة.

11.7 الفصل السابع

11.7.1 التمرين الأول

```
#define MAXLEN 100
```

11.7.2 التمرين الثاني

قد يكون هناك بعض المشاكل المتعلقة بالأسبقية في بعض التعابير، والتعريف الأكثر أماناً في هذه

الحالة هو:

```
#define VALUE (100+MAXLEN)
```

11.7.3 التمرين الثالث

```
#define REM(a,b) ((a)%(b))
```

11.7.4 التمرين الرابع

```
#define REM(a,b) ((long)(a)%(long)(b))
```

11.7.5 التمرين الخامس

غالبًا ما يشير لوجود ملف ترويسة لمكتبة.

11.7.6 التمرين السادس

غالبًا ما يشير لوجود ملف ترويسة خاص.

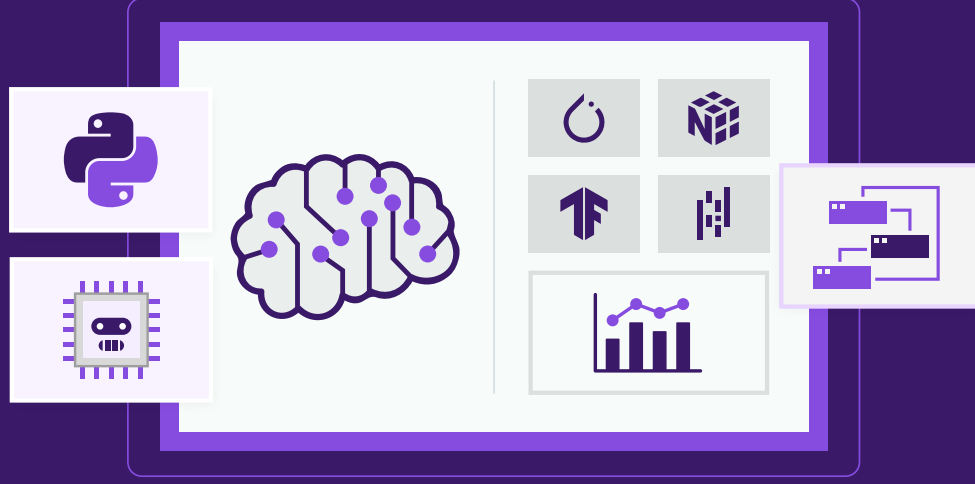
11.7.7 التمرين السابع

باستخدام توجيهات التصريف الشرطي، والأمثلة موضحة في محتويات الفصل.

11.7.8 التمرين الثامن

يستخدم النوع `long int` مكان `int` و `unsigned long int` مكان `unsigned int` باستخدام البيئة الحاسوبية المقدّمة من المترجم، وليس البيئة الهدف، ويجب أن تُقدّم على الأقل النطاقات المحددة في ملف الترويسة `<limits.h>`.

دورة الذكاء الاصطناعي



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



أحدث إصدارات أكاديمية حسوب

