https://simplifycpp.org

إتقان C23

دليل شامل للبرمجة منخفضة المستوى، وأنظمة التشغيل، وتصميم المترجمات

إعداد: أيمن الحراكي

إتقان C23

دليل شامل للبرمجة منخفضة المستوى، وأنظمة التشغيل، وتصميم المترجمات

تمت الاستعانة بأدوات ذكاء اصطناعي حديثة في بعض مراحل الصياغة واستكشاف الأفكار، وذلك تحت إشراف كامل، مع المراجعة والتحقق والتصحيح، مع احتفاظ المؤلف الكامل بحقوق التأليف والمسؤولية العلمية.

إعداد : أيمن الحراكي

simplifycpp.org

فبراير 2026

المحتويات

مقدمة المؤلف

بدأت رحلتي مع لغة البرمجة **C** عام 1989 باستخدام **Borland Turbo C**—في وقتٍ كانت فيه البرمجة تعني التعامل المباشر مع قيود العتاد، وذاكرة محدودة، ومستوى تجريد شبه معدوم. وبعد بضع سنوات، في أوائل تسعينيات القرن الماضي، انتقلتُ بشكلٍ طبيعي إلى **C++**، حيث بدت خصائص البرمجة كائنية التوجه امتداداً قوياً لطريقة التفكير في C، لا بديلاً عنها.
لسنواتٍ طويلة، كنتُ—مثل كثيرين غيري—أفترض أن **C** قد تراجعت تدريجيًا إلى الخلف، وطغت عليها **C++** ولاحقًا لغات برمجة الأنظمة الحديثة مثل **Rust**. لكن الممارسة المهنية والعمل العميق على مستوى الأنظمة يكشفان واقعًا مختلفًا: لا تزال **C** واحدة من أكثر لغات البرمجة أهميةً وجذريةً في عصرنا الحالي.
تواصل C هيمنتها على المجالات الحرجة التي لا يكون فيها التنبؤ بالسلوك، والتحكم الكامل، وانخفاض مستوى التجريد خياراتٍ إضافية— بل متطلباتٍ أساسية لا غنى عنها.

لا تزال **C** لغةً أساسية في المجالات التالية:

- تطوير أنظمة التشغيل لا تزال النوى الأساسية، ومجدولات المهام، ومديرو الذاكرة، ومشغلات الأجهزة تعتمد بشكلٍ كبير على C بسبب سلوكها الحتمي وإمكانية الوصول المباشر إلى العتاد.
- الأنظمة المضمنة وأنظمة **Bare-Metal** من المتحكمات الدقيقة إلى الأنظمة ذات الزمن الحقيقي، توفر C تحكمًا لا يُضاهى في تخطيط الذاكرة، وتوقيت التنفيذ، وسجلات العتاد.
- تطوير المترجمات، والروابط، وسلاسل الأدوات لا يزال عدد كبير من المترجمات، والمجمِّعات، والروابط، وأدوات التحليل الساكن المستخدمة في الإنتاج مكتوبًا أساسًا بلغة C.

- البرمجيات منخفضة المستوى والحساسة للأداء تعتمد البرمجيات الثابتة، ومحمِّلات الإقلاع، والمشرفات الافتراضية، ومكتبات التشغيل على C لما توفره من شفافية ونموذج تجريد صفري التكلفة.

ولهذه الأسباب، كُتب هذا الكتاب باستخدام أحدث معيار للغة **C: C23**. وبدلاً من التعامل مع C كلغةٍ تراثية، يقدّمها هذا الكتاب بوصفها لغة أنظمة حديثة ومتطورة—لغةً تكيفت مع الزمن مع الحفاظ على مواطن قوتها الأصلية.

ما الذي يركّز عليه هذا الكتاب

يركّز هذا الكتاب عمداً على المجالات التي لا تزال **C** فيها دون منافس:

- التحكم الدقيق في الذاكرة وتمثيل البيانات
- التنفيذ القابل للتنبؤ والصحة منخفضة المستوى
- التفاعل المباشر مع العتاد وأنظمة التشغيل
- بناء الأدوات التأسيسية بدلاً من استهلاك التجريدات

خُصصت فصولٌ كاملة لتطوير أنظمة التشغيل، والبرمجة المضمنة، وبناء المترجمات، وتصميم البرمجيات منخفضة المستوى، وكل ذلك مستند إلى ممارسات حديثة وفق **C23**.

نظرة عامة على محتوى الكتاب

١. لغة **C** وتطورها

- السياق التاريخي وفلسفة تصميم C
- كيف تختلف C جذريًا عن C++ و Rust

- لماذا تواصل C البقاء رغم التحولات التقنية

٢. معايير **C** الحديثة — **C23**

- التحديثات اللغوية المقدّمة في C23
- تحديث الصياغة والدلالات وأفضل الممارسات
- كتابة شيفرة C قابلة للنقل ومتوافقة مع المستقبل

٣. برمجة الأنظمة المضمنة باستخدام **C**

- مفاهيم برمجة Bare-Metal
- الإدخال والإخراج المعتمد على الذاكرة والتحكم بالعتاد
- أمثلة عملية على المتحكمات الدقيقة

٤. تطوير أنظمة التشغيل

- دور C في تصميم النواة
- إدارة العمليات والذاكرة والمقاطعات
- بناء المكونات الأساسية لنظام التشغيل باستخدام C

٥. تطوير المترجمات والأدوات

- المحللات المعجمية، والمحللات النحوية، والتمثيلات الوسيطة
- كتابة مترجمات وأدوات لغوية بسيطة باستخدام C

٦. تقنيات البرمجة منخفضة المستوى

- إدارة الذاكرة اليدوية
- تخطيط البيانات، والمحاذاة، وفهم ABI
- الربط مع لغة التجميع والعتاد

٧. مشاريع عملية ودراسات حالة

- مشاريع متكاملة تجمع جميع المفاهيم
- تحسين الأداء واستراتيجيات الصحة البرمجية

هدف هذا الكتاب

لا يهدف هذا الكتاب إلى إقناع القارئ بأن C يجب أن تحل محل اللغات الحديثة—بل إلى توضيح أين ولماذا لا تزال **C** لغةً لا غنى عنها.
ومن خلال إتقان **C23**، سيكتسب القارئ:

- فهمًا أعمق لكيفية تفاعل البرمجيات مع العتاد
- القدرة على التفكير المنهجي في الأداء والصحة البرمجية
- المهارات اللازمة لبناء برمجيات الأنظمة التأسيسية

هذا الكتاب موجّه للمبرمجين الذين يرغبون في الاقتراب أكثر من الآلة، وفهم ما تخفيه التجريدات، وكتابة برمجيات تشكّل العمود الفقري للحوسبة الحديثة.

كلمات أخيرة

ليست C لغة الماضي—بل لغة المسؤولية. إنها تكافئ الانضباط، والوضوح، والفهم العميق.
ومن خلال هذا الكتاب، أسعى إلى تقديم C كما هي في واقعها اليوم: لغة برمجة أنظمة حديثة، دقيقة، وقوية.

أيمن الحراكي

التمهيد

على الرغم من أن C كلغة برمجة يزيد عمرها اليوم عن خمسة عقود، وعلى الرغم من ظهور لغات أحدث وأكثر تعقيداً مثل C++ وRust، لا تزال C تهيمن على مجالات البرمجة منخفضة المستوى المرتبطة مباشرة بالعتاد، ونوى أنظمة التشغيل، وبناء المترجمات وسلاسل الأدوات. وليس هذا التفوق صدفةً تاريخية، ولا مجرد نتيجة لقصور الإرث البرمجي، بل هو متجذر في أسباب تقنية عميقة ومستقرة عبر الزمن—ويأتي في مقدمتها استقرار واجهة التطبيق الثنائية (**Application Binary Interface — ABI**) عبر المعالجات وأنظمة التشغيل المختلفة.

استقرار ABI: حجر الأساس لتفوّق C

تُعد لغة C فريدةً تقريبًا في قدرتها على توفير واجهة **ABI** مستقرة، ومفهومة بعمق، وطويلة العمر على مدى عقود من التطور، وعبر طيفٍ واسع من المعماريات مثل x86 وARM وRISC-V، وكذلك أنظمة تشغيل متعددة تشمل Linux وBSD وWindows. ويعني هذا الاستقرار ما يلي:

- يمكن ربط الشيفرة المترجمة بلغة C وتنفيذها بشكل موثوق عبر إصدارات متعددة من المترجمات.
- لا تحتاج نوى أنظمة التشغيل والمكتبات الأساسية للنظام إلى إعادة تصميم مع كل إصدار لغوي جديد.
- تعتمد اللغات البرمجية الأخرى على C بوصفها طبقة الواجهة الثنائية الأساسية لها—وليس العكس.

وعلى النقيض من ذلك، لا تُعرّف C++ واجهة ABI مستقرة على مستوى معيار اللغة، كما أن Rust—على الرغم من قوتها التقنية—تتعامل مع عدم استقرار ABI بوصفه خيارًا تصميميًا مقصودًا. وهذا ما يجعل Rust، في الوقت الحالي على الأقل، غير مناسبة لتكون اللغة الأساسية لنوى أنظمة التشغيل، والمترجمات، والواجهات الثنائية طويلة العمر.

لماذا تتفوّق C على C++ وRust في أدنى المستويات

لا يعود تفوّق C في هذا المجال إلى بساطتها فحسب، بل إلى خصائص جوهرية يصعب تكرارها في لغات أخرى:

- نموذج ذاكرة شفاف ومباشر: لا توجد تجريدات مخفية، ولا مدمّرات ضمنية، ولا سلوك غير متوقع ناتج عن آليات تشغيل عالية المستوى.
- تطابق شبه كامل مع العتاد: ما يُكتب بلغة C يعكس بشكلٍ وثيق ما ينفذه المعالج فعليًا.
- تكامل سلس مع لغة التجميع: تُعد C الرفيق الطبيعي للغة التجميع في النوى، ومحمّلات الإقلاع، والمترجمات.
- لغة **FFI** العالمية: تختار جميع اللغات الحديثة تقريبًا C بوصفها الجسر الرسمي للتعامل مع العتاد ومكتبات النظام.

ولهذه الأسباب، تظل C الطبقة التأسيسية التي تُبنى فوقها اللغات الأخرى—حتى وإن قدّمت تلك اللغات ضمانات أمان أقوى أو تجريدات أعلى مستوى.

كيف أصبحت C الحديثة أقوى من أي وقت مضى

من المفاهيم الخاطئة الشائعة أن C قد توقفت عن التطور. لكن الواقع أن المعايير الحديثة—والتي بلغت ذروتها في C23—قد عززت اللغة بشكلٍ كبير دون المساس بفلسفتها الأساسية.
ومن أبرز الإضافات:

- أنواع صحيحة دقيقة مثل _BitInt(N) للتحكم الدقيق في عرض الأعداد الصحيحة.
- السمات مثل [[nodiscard]] التي تعزز الانضباط والصحة البرمجية دون فرض نماذج برمجة جديدة.
- مرافق أكثر أمانًا في المكتبة القياسية مثل reallocarray وstrlcpy.
- تحسين دعم التوازي والذرات بما يتناسب مع أنظمة المعالجات متعددة الأنوية الحديثة.
- دعم أفضل للتحليل الساكن، مما يتيح أدوات أكثر قوة لاكتشاف الأخطاء في مراحل مبكرة.

والأهم من ذلك أن جميع هذه التحسينات أُدخلت دون كسر التوافقية الخلفية ودون التضحية باستقرار **ABI**—وهو ما يشكّل ميزة حاسمة للغة C مقارنة بلغات الأنظمة المنافسة.

ملخص التمهيد

ليست C لغةً قديمة تعيش بالمصادفة؛ بل هي لغة مبنية على أسس هندسية صارمة أثبتت قدرتها على الصمود عبر الزمن. إن استمرار هيمنتها في برمجة العتاد، ونوى أنظمة التشغيل، وبنية المترجمات، لا يعود إلى غياب البدائل، بل لأن هذا المستوى من البرمجة يتطلب بطبيعته:

- استقراراً طويل الأمد،
- شفافيةً مطلقة في التفاعل مع العتاد،
- وتحكمًا كاملاً دون افتراضات مخفية.

ولهذه الأسباب، ستبقى C—إلى جانب تطورها المعياري الحديث—حجر الزاوية في برمجة الأنظمة الحرجة، بغضّ النظر عن كيفية تطور اللغات الأخرى أو تغيّر أدوات البرمجة مع مرور الزمن.

الفصل ١: مقدمة إلى C23

١.١ تاريخ وتطور لغة C

١.١.١ أصول لغة C

ظهرت لغة البرمجة C في أوائل سبعينيات القرن الماضي في **Bell Labs**، وقد قام بتطويرها **Dennis Ritchie** بوصفها خليفةً عملية للغة **B**، والتي كانت بدورها مشتقة من **BCPL (Basic Combined Programming Language)**. وقد صُممت C بالتزامن مع تطوير نظام التشغيل **UNIX**، بهدفٍ واضح يتمثل في إنشاء لغة قادرة على برمجة الأنظمة منخفضة المستوى، مع الحفاظ في الوقت ذاته على قابلية النقل عبر معماريات العتاد المختلفة.

وعلى خلاف لغة التجميع، قدّمت C تدفق تحكم منظّمًا وتجريدات قابلة للنقل؛ وعلى خلاف اللغات عالية المستوى في تلك الحقبة، احتفظت بإمكانية الوصول المباشر إلى الذاكرة وبُنى الآلة منخفضة المستوى. وقد شكّل هذا التوازن السمة الجوهرية والدائمة للغة C.

المحطات المبكرة

- 1972: أول تنفيذ للغة C على معالج PDP-11.
- 1973: إعادة كتابة UNIX بلغة C، مما أثبت أن أنظمة التشغيل يمكن كتابتها بلغة عالية المستوى.

- 1978: نشر كتاب *The C Programming Language* بواسطة Kernighan وRitchie، والذي شكّل أول توصيف واسع القبول للغة.

١.١.٢ التقييس: من K&R إلى ISO C

مع انتشار لغة C خارج Bell Labs، ظهرت لهجات غير متوافقة فيما بينها. وأصبح التقييس الرسمي ضرورةً أساسية للحفاظ على قابلية النقل والتوافق بين المنصات.

K&R C

كانت اللهجة الأصلية الموصوفة في كتاب K&R تفتقر إلى العديد من الميزات التي تُعد اليوم أساسية، مثل نماذج الدوال والمكتبة القياسية الموحّدة. ومع ذلك، فقد أرست الأساس الدلالي للغة وحددت جوهرها المفاهيمي.

ANSI C وISO C

- **ANSI C (C89/C90)** قدّمت نماذج الدوال، والرؤوس القياسية، وسلوكًا محدداً بدقة.
- **ISO C** (1990) اعتمدت معيار ANSI C على المستوى الدولي، مما رسّخ C كلغة قياسية عالمية.

١.١.٣ التطور الحديث: C99 وC11 وC17 وC23

تطورت لغة C بحذر شديد، حيث أُعطيت الأولوية للتوافقية الخلفية وقابلية التنبؤ على حساب التغييرات الجذرية السريعة.

C99

قدّمت C99 مجموعة من الميزات الحديثة الأساسية:

- دوال inline
- مصفوفات بطول متغير
- نوع الأعداد الصحيحة long long
- تعليقات السطر الواحد //

C11

ركّزت C11 على الصحة البرمجية والتوازي:

- نموذج ذاكرة رسمي
- دعم اختياري للتعددية (<threads.h>)
- عمليات ذرية
- _Generic للبرمجة العامة حسب النوع

C17

كانت C17 مراجعةً لإصلاح العيوب، دون إضافة ميزات لغوية جديدة، مع توضيح عدد من الغموض الموجود في C11.

C23

تمثل C23 عملية تنقية وتنظيف للغة أكثر من كونها إعادة تصميم شاملة. وتتمثل أهدافها في:

- توضيح السلوك غير المعرّف والسلوك المعتمد على التنفيذ
- تحسين اتساق المكتبة القياسية
- إزالة البُنى المتقادمة

- توافق أفضل مع أدوات التطوير الحديثة

١.٢ دور لغة C في البرمجيات الحديثة

على الرغم من قدمها، لا تزال لغة C تشكل حجر أساس في الحوسبة الحديثة:

- أنظمة التشغيل والنوى
- الأنظمة المضمنة وأنظمة الزمن الحقيقي
- المترجمات وبيئات التشغيل وسلاسل الأدوات
- البرمجيات الثابتة ومحمّلات الإقلاع والمشرفات الافتراضية

ولا تستمر C لأنها لغة رائجة، بل لأنها تمثل الآلة الأساسية بدقة متناهية.

٣.١ لماذا دراسة C في عصر اللغات الحديثة

تعلم لغة C لا يتعلق بالحنين إلى الماضي، بل بفهم الواقع التقني كما هو.

- شفافية الأداء: لا توجد تخصيصات خفية ولا سلوكيات تشغيل غير متوقعة.
- قابلية النقل: تعمل على جميع المعماريات تقريبًا.
- معرفة تأسيسية: الذاكرة، اتفاقيات النداء، تخطيط البيانات، ونماذج التنفيذ.
- الاستمرارية: شيفرة كُتبت قبل عقود لا تزال تعمل حتى اليوم.

٤.١ ما الجديد في C23

١.٤.١ فلسفة C23

لا تحاول C23 منافسة C++ أو Rust في مستوى التجريد أو ضمانات الأمان. وبدلاً من ذلك، تركز على ما يلي:

- إزالة الأخطاء التاريخية
- جعل النية البرمجية أوضح
- تحسين قابلية التحليل
- الحفاظ على النموذج الذهني البسيط للغة C

٢.٤.١ تحسينات اللغة والمكتبة

السمات (Attributes)

قامت C23 بتقييس عدد من السمات التي كانت متاحة سابقًا بوصفها امتدادات خاصة بالمترجمات:

```
[[nodiscard]] int acquire_resource(void);
[[deprecated("use new_api instead")]] void old_api(void);
[[maybe_unused]] static int debug_flag;
```

تُحسّن هذه السمات التشخيصات البرمجية دون إحداث أي تغيير في الدلالات اللغوية.

أدوات أمان الأعداد الصحيحة

تُقدّم C23 الرأس <stdckdint.h> لدعم العمليات الحسابية المُتحقَّق منها:

```c
#include <stdckdint.h>

int r;
if (ckd_add(&r, a, b)) {
    /* overflow detected */
}
```

يتيح هذا الأسلوب اكتشاف فيض العمليات الحسابية بشكلٍ صريح ومحمول، دون الوقوع في سلوك غير معرّف.

أدوات معالجة البِتّات

يوفّر الرأس الجديد <stdbit.h> عمليات قياسية لمعالجة البِتّات، مما يقلل الاعتماد على التعليمات الخاصة بالمترجمات.

توضيحات **Unicode**

تحسّن C23 الاتساق في التعامل مع UTF-8 وأنواع المحارف التي أُدخلت في المعايير السابقة، دون تغيير نموذج C القائم على التمثيل البايتّي.

٣.٤.١ الميزات المُزالة أو المُهمَلة

تواصل C23 عملية التنظيف طويلة الأمد للغة:

- إزالة الدالة gets بالكامل
- حظر التصريحات الضمنية للدوال
- إهمال الكلمة المحجوزة register
- تثبيط البُنى التراثية وغير محددة السلوك

١.٥ الأثر العملي لـ C23

بالنسبة للمبرمجين الممارسين، تعني C23:

- تشخيصات أوضح
- عمليات حسابية أكثر أمانًا عند الحاجة
- تحسين أدوات التحليل الساكن
- عدم إحداث أي تعطيل للشيفرة الجيدة المكتوبة مسبقًا بلغة C

١.٦ الملخص

تعيد C23 تأكيد هوية لغة C كلغة دقة ومسؤولية. فهي تُحدّث ما يحتاج إلى وضوح، وتزيل ما أثبت التاريخ ضرره، وتحافظ على المبادئ الجوهرية التي جعلت C لغة لا غنى عنها.
يتعامل هذا الكتاب مع C23 لا بوصفها منافسًا للغات أحدث، بل كأداة مُنقّحة للمبرمجين الذين يحتاجون إلى تحكم كامل بالآلة، وإلى الانضباط اللازم لاستخدامها بالشكل الصحيح.

١.٧ أهمية لغة C في البرمجة الحديثة

على الرغم من الانتشار الواسع للغات البرمجة عالية المستوى، لا تزال لغة البرمجة C واحدة من أكثر التقنيات أهميةً وتأثيرًا في تطوير البرمجيات الحديثة. ولا تعود هذه الأهمية المستمرة إلى المصادفة؛ إذ تحتل C موقعًا فريدًا على الحد الفاصل بين البرمجيات والعتاد، حيث تكون الدقة، وقابلية التنبؤ، والأداء عناصر أساسية لا يمكن التنازل عنها.
يشرح هذا القسم سبب بقاء C لغةً تأسيسية في الوقت الحاضر، والدور الذي تؤديه في الأنظمة الحديثة، ولماذا لا يزال تعلمها خطوةً محورية للمبرمجين الجادين.

١.٧.١ لغة C كأساس للحوسبة الحديثة

ليست C مجرد لغة برمجة أخرى، بل هي الطبقة التحتية التي بُنيت فوقها نسبة كبيرة من الحوسبة الحديثة.

- أنظمة التشغيل
  تُكتب المكونات الأساسية لأنظمة التشغيل الكبرى في الغالب باستخدام C. ويشمل ذلك النوى، ومجدولات المهام، ومديري الذاكرة، ومشغلات الأجهزة. توفّر C مستوى التحكم المطلوب للتفاعل المباشر مع العتاد، مع الحفاظ على قابلية النقل بين المعماريات المختلفة.

- المترجمات وبيئات تشغيل اللغات
  تعتمد العديد من تطبيقات اللغات الحديثة اعتماداً كبيرًا على C في أنظمة التشغيل الداخلية، والآلات الافتراضية، وسلاسل الأدوات. ويجعل الجمع بين الكفاءة، وقابلية النقل، والتحكم الدقيق C مناسبةً بشكل فريد لهذا الدور.

- التأثير في تصميم اللغات
  شكّلت C بناء الصياغة، والدلالات، ونماذج التنفيذ في عدد كبير من اللغات اللاحقة. ويساعد فهم C على توضيح مفاهيم مثل تخطيط الذاكرة، واتفاقيات النداء، وتمثيل البيانات، وهي مفاهيم لا تزال حاضرة في مختلف البيئات البرمجية.

١.٧.٢ لماذا لا تزال C ذات صلة اليوم

الأداء والشفافية

توفّر C مستوىً من الشفافية في الأداء لا تضاهيه معظم اللغات الحديثة.

- التطابق المباشر مع العتاد
  تتيح C تحكمًا صريحًا في الذاكرة، وتخطيط البيانات، وتدفق التنفيذ، مما يجعلها ضرورية في المجالات الحساسة للأداء.
- افتراضات تشغيلية محدودة
  تُترجم برامج C مباشرةً إلى شيفرة آلة دون الحاجة إلى نظام تشغيل وسيط، أو جامع نفايات، أو آلة افتراضية.

قابلية النقل

يسمح نموذج الآلة المجرّدة في C بترجمة الشيفرة عبر نطاق واسع من المنصات.

- من الحواسيب العملاقة إلى المتحكمات الدقيقة
- من البرمجيات الثابتة المضمنة إلى نوى أنظمة التشغيل
- مع دلالات متسقة يحددها معيار اللغة

المرونة

تُستخدم C في طيف متنوع من المجالات:

- تطوير الأنظمة والنوى
- البرمجيات المضمنة وأنظمة الزمن الحقيقي
- مكدسات الشبكات وقواعد البيانات
- المكتبات الحساسة للأداء

١.٧.٣ لغة C في المجالات التقنية الحديثة

الأنظمة المضمنة وإنترنت الأشياء

لا تزال C اللغة المهيمنة في تطوير الأنظمة المضمنة.

- تحكم دقيق في الذاكرة والتوقيت
- توافق مع بيئات Bare-Metal
- تكامل وثيق مع أنظمة التشغيل ذات الزمن الحقيقي

الحوسبة عالية الأداء

تُستخدم C على نطاق واسع في الحوسبة العلمية والعددية.

- شيفرة متجهة ومتوازية عالية الكفاءة
- تكامل مع MPI وOpenMP
- مكتبات عددية مستقرة وطويلة العمر

أنظمة التشغيل والنوى

تتطلب نوى أنظمة التشغيل سلوكًا حتميًا، وأداءً قابلاً للتنبؤ، وتحكمًا صريحًا—وهي متطلبات تتوافق مباشرةً مع فلسفة تصميم لغة C.

٤.٧.١ تعلم C كأساس مفاهيمي

تعلم C لا يقتصر على إتقان الصياغة، بل يتعداه إلى فهم كيفية عمل الحواسيب فعليًا.

- إدارة الذاكرة
  تعلّم الإدارة اليدوية للذاكرة مفاهيم أعمار الكائنات، والملكية، والانضباط في التعامل مع الموارد.

- نموذج التنفيذ
  تكشف C بشكلٍ صريح عن استدعاءات الدوال، وإطارات المكدس، وتخطيط البيانات.

- الانضباط الخوارزمي
  في غياب التجريدات الثقيلة، تشجّع C على التفكير المتأني في الخوارزميات وبُنى البيانات.

١.٧.٥ الملخص

تظل C لغةً أساسية، لا لأنها قديمة، بل لأنها تمثّل الآلة بأمانة ودقة. ويضمن دورها في أنظمة التشغيل، والأنظمة المضمنة، وسلاسل الأدوات، والبرمجيات الحساسة للأداء استمرار أهميتها.
إن فهم C يزوّد المبرمجين بالنماذج الذهنية اللازمة للاستدلال على الصحة البرمجية، والأداء، وسلوك الأنظمة—وهي مهارات تنتقل فائدتها إلى جميع اللغات الأخرى.

٨.١ إعداد بيئة تطوير حديثة لـ C23

قبل كتابة برامج C23، يجب إعداد بيئة تطوير صحيحة ومنضبطة. يوضّح هذا القسم إعداداً بسيطاً ومحمولاً مناسباً لتطوير C الاحترافي عبر المنصات المختلفة.

١.٨.١ اختيار مترجم يدعم C23

لا يزال دعم C23 في طور التطور ويختلف من مترجم لآخر.

GCC

- مجموعة مترجمات واسعة الاستخدام وناضجة
- دعم تجريبي لكنه في تطور مستمر لـ C23 (GCC 13+)
- شائع في بيئات Linux والأنظمة المضمنة

Clang

- مترجم حديث يتميز بتشخيصات ممتازة
- دعم جزئي لـ C23 (Clang 16+)
- تكامل قوي مع أدوات التحليل

MSVC

- يركز بشكل أساسي على C++
- دعم محدود وغير مكتمل لـ C23
- مناسب أساساً للتطوير الخاص بنظام Windows

٢.٨.١ المحررات وبيئات التطوير المتكاملة

لا تتطلب C استخدام بيئة تطوير متكاملة، لكن الأدوات المناسبة ترفع الإنتاجية.

- محررات خفيفة مع أدوات خارجية
- بيئات تطوير متكاملة للمشاريع الكبيرة
- تكامل المصحح والمترجم أهم من واجهة المستخدم

٣.٨.١ أدوات البناء

Make

- بسيط، وصريح، وشفاف
- شائع في المشاريع منخفضة المستوى والمضمنة

CMake

- نظام بناء متعدد المنصات
- واسع الاستخدام في مشاريع C الحديثة

٤.٨.١ أول برنامج C23

```c
#include <stdio.h>

int main(void) {
    puts("Hello, C23!");
    return 0;
}
```

٥.٨.١ الترجمة باستخدام وضع C23

```
gcc -std=c23 -Wall -Wextra -pedantic hello.c
```

```
clang -std=c23 -Wall -Wextra -pedantic hello.c
```

٦.٨.١ التنقيح

GDB

```
gcc -g hello.c
gdb ./a.out
```

LLDB

```
clang -g hello.c
lldb ./a.out
```

٧.٨.١ التحقق من دعم C23

يجب التحقق من دعم C23 باستخدام ميزات موجودة فعليًا في المعيار، مثل السمات أو العمليات الحسابية المُتحقَّق منها.

```c
[[nodiscard]] int f(void) { return 1; }

int main(void) {
    f();
}
```

تؤكد تشخيصات المترجم توفر الدعم.

١.٨.٨ الملخص

تركّز بيئة C23 الحديثة على الصحة البرمجية، وقوة التشخيصات، والانضباط في استخدام الأدوات، بدلاً من الراحة أو التسهيل الزائد. ومن خلال اختيار المترجمات المناسبة، وتفعيل التحذيرات الصارمة، وفهم سلسلة الأدوات، يمكن للمبرمجين كتابة شيفرة C محمولة، وفعّالة، وصحيحة.
ومع اكتمال إعداد البيئة، ستنتقل الفصول التالية إلى تناول البُنى الأساسية للغة وأساليب البرمجة المنضبطة باستخدام C23.

الفصل ٢: أساسيات C23

١.٢ الصياغة الأساسية وبنية البرنامج

يُعد الفهم الواضح لصياغة لغة C وبنية برامجها أمرًا أساسيًا قبل الانتقال إلى الموضوعات المتقدمة. فقد صُممت C عن قصد لتكون لغةً حدّية وبسيطة؛ إذ تعكس صياغتها نموذج تنفيذ الآلة الكامنة بدلاً من إخفائه خلف طبقات من التجريد.
يقدّم هذا القسم العناصر البنيوية لبرنامج C23، والقواعد التي تحكم الصياغة الصحيحة، والاصطلاحات التي تؤدي إلى شيفرة مقروءة وقابلة للصيانة.

١.١.٢ بنية برنامج C

يتكون برنامج C من تسلسل من التصريحات والتعريفات. ويبدأ التنفيذ من نقطة دخول محددة بوضوح، وهي الدالة main.

```c
#include <stdio.h>

int main(void) {
    puts("Hello, C23!");
    return 0;
}
```

يؤدي كل عنصر في هذا البرنامج غرضًا محددًا.

توجيهات المُعالج القبلي

تُعالج توجيهات المُعالج القبلي قبل بدء الترجمة. وهي لا تُعد جزءًا من لغة C بالمعنى الدقيق، لكنها عنصر أساسي في البرامج الواقعية.

- #include تُدرج محتوى ملف رأسي، مما يجعل التصريحات مرئية.

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

- #define تُنشئ بدائل نصية تُوسّع حرفيًا.

```
#define PI 3.1415926535897793
```

الدالة main

تُعد الدالة main نقطة الدخول للبرنامج.

```
int main(void) {
    return 0;
}
```

- يبدأ التنفيذ داخل main
- تُمرَّر قيمة الإرجاع إلى بيئة التشغيل
- تشير قيمة الإرجاع 0 تقليديًا إلى النجاح

تشير قائمة المعاملات void صراحةً إلى أن البرنامج لا يقبل معاملات من سطر الأوامر.

التعليمات والتعبيرات

- التعليمات تُنفّذ أفعالاً وتنتهي بفاصلة منقوطة.

```
int x = 10;
puts("C is explicit.");
```

- التعبيرات تحسب قيمًا، وقد تظهر داخل التعليمات.

```
int sum = x + 5;
```

التعليقات

تُستخدم التعليقات لتوثيق النية البرمجية وتوضيح السلوك غير الواضح، ويتجاهلها المترجم.

- تعليقات السطر الواحد:

```
// This is a single-line comment
```

- التعليقات متعددة الأسطر:

```
/* This is a
   multi-line comment */
```

٢.١.٢ قواعد الصياغة الأساسية

صياغة C صارمة بطبيعتها؛ فالمخالفات الصغيرة قد تؤدي إلى أخطاء ترجمة أو إلى سلوك غير معرّف.

حساسية حالة الأحرف

لغة C حساسة لحالة الأحرف.

```c
int value;
int Value;  // Different identifier
```

الفواصل المنقوطة

يجب أن تنتهي كل تعليمة بفاصلة منقوطة.

```c
int x = 10;   // Correct
int y = 20    // Error
```

الأقواس والنطاقات

تُستخدم الأقواس لتحديد التعليمات المركبة ونطاقات الرؤية.

```c
int main(void) {
    /* block scope */
}
```

المسافات البيضاء

المسافات البيضاء غير مؤثرة على المترجم غالبًا، لكنها مهمة جدًا للقارئ البشري.

```c
int a=10;      // Legal but unreadable
int b = 10;    // Preferred
```

٣.١.٢ أول برنامج C23 متكامل

```c
#include <stdio.h>

int main(void) {
    puts("Hello, C23!");
    return 0;
}
```

الترجمة والتنفيذ

- باستخدام GCC:

```
gcc -std=c23 -Wall -Wextra -pedantic hello.c
./a.out
```

- باستخدام Clang:

```
clang -std=c23 -Wall -Wextra -pedantic hello.c
./a.out
```

٤.١.٢ أخطاء شائعة

- نسيان الفواصل المنقوطة
- استخدام معرفات قبل التصريح عنها
- نسيان تضمين الرؤوس المطلوبة
- القراءة من متغيرات غير مهيأة

تُعد هذه الأخطاء من أكثر أسباب السلوك غير المعرّف والأخطاء الدقيقة.

٥.١.٢ أفضل الممارسات منذ البداية

- استخدام نماذج دوال صريحة
- تهيئة جميع المتغيرات
- تفعيل تحذيرات المترجم الصارمة
- تفضيل الوضوح على الحِيَل البرمجية

```
int student_count;      // Clear
int sc;                 // Avoid
```

٦.١.٢ الملخص

قدّم هذا الفصل الأسس البنيوية والصياغية للغة C23. تعلّمت كيف يُنظَّم برنامج C، وكيف يبدأ التنفيذ، وما القواعد التي تحكم الصياغة الصحيحة، وما الممارسات التي تؤدي إلى شيفرة نظيفة ومتينة.

هذه الأسس ليست تمهيدية فحسب، بل تشكّل النموذج الذهني اللازم للاستدلال الصحيح على برامج C. ومع ترسيخ هذه القاعدة، يمكننا الانتقال إلى استكشاف نظام الأنواع، وتمثيل البيانات، وتدفق التحكم في C23 بمزيد من العمق.

٢.٢ أنواع البيانات والمتغيرات

تشكل أنواع البيانات والمتغيرات الأساس لكل برنامج C. وعلى خلاف العديد من اللغات عالية المستوى، تكشف C نموذج الآلة الكامن بصورة صريحة؛ إذ تحدد أنواع البيانات كيفية تمثيل الكائنات في الذاكرة، وحجم التخزين الذي تشغله، وكيفية تنفيذ العمليات عليها.
يعرض هذا القسم نظام أنواع البيانات في C23، وقواعد التصريح والتهيئة، ودلالات النطاق، ومدة التخزين، والعمر، والتحويل.

١.٢.٢ أنواع البيانات في C23

في C، يحدد نوع البيانات كيفية تفسير القيمة وكيفية تخطيط الكائن في الذاكرة. ويُعد نظام الأنواع في C23 صغيرًا عمدًا، لكنه بالغ التعبير.

أنواع البيانات الأساسية

تمثل الأنواع الأساسية قيَمًا قياسية مدعومة مباشرة من اللغة.

الأنواع الصحيحة

- `char` — أصغر وحدة تخزين قابلة للعنونة
- `signed char`، `unsigned char`
- `short`، `unsigned short`
- `int`، `unsigned int`
- `long`، `unsigned long`
- `long long`، `unsigned long long`

الحجم الدقيق لهذه الأنواع معتمد على التنفيذ، لكن المعيار يضمن علاقات ترتيب مثل:

```
sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
```

أنواع الفاصلة العائمة

- `float` — دقة أحادية
- `double` — دقة مزدوجة
- `long double` — دقة موسعة

يتبع سلوك الأعداد العائمة عادةً معيار IEEE-754 عند توفره، لكن هذا غير مطلوب إلزاميًا من المعيار.

النوع المنطقي

- `bool` — يمثل قيم الحقيقة المنطقية

في C23، أصبح `bool` كلمةً محجوزة مدمجة، ولم يعد يتطلب تضمين `<stdbool.h>`.

أنواع البيانات المشتقة

تُبنى الأنواع المشتقة انطلاقًا من الأنواع الأساسية.

المصفوفات

تمثل المصفوفات تسلسلات متجاورة من الكائنات.

```
int values[5] = {1, 2, 3, 4, 5};
```

المؤشرات

تخزن المؤشرات عناوين الكائنات أو الدوال.

```
int *p = &values[0];
```

البُنى

تجمع البُنى كائنات مترابطة تحت نوع واحد.

```
struct Point {
    int x;
    int y;
};
```

الاتحادات

تسمح الاتحادات لأعضاء متعددين بمشاركة نفس مساحة التخزين.

```
union Number {
    int i;
    double d;
};
```

الأسماء المستعارة للأنواع

تسمح C بإنشاء أسماء مستعارة للأنواع باستخدام typedef.

```
typedef unsigned long size_type;
size_type length;
```

لا تُنشئ الأسماء المستعارة أنواعًا جديدة، بل تحسّن المقروئية ومستوى التجريد.

٢.٢.٢ المتغيرات والكائنات

يشير المتغير في C إلى كائن: منطقة تخزين لها نوع، وقيمة، وعمر، ونطاق رؤية.

التصريح والتعريف

```
int counter;
double average;
```

يُقدّم التصريح الاسم والنوع، بينما يخصص التعريف مساحة التخزين.

التهيئة

يمكن تهيئة الكائنات وقت التعريف.

```
int count = 10;
double rate = 0.75;
char grade = 'A';
```

تحمل الكائنات غير المهيأة ذات مدة التخزين التلقائية قيمًا غير محددة.

قواعد التسمية

- تبدأ المعرفات بحرف أو شرطة سفلية
- قد تحتوي على حروف وأرقام وشرطات سفلية
- حساسة لحالة الأحرف
- يجب ألا تتعارض مع الكلمات المحجوزة

٣.٢.٢ نطاق الرؤية ومدة التخزين والعمر

نطاق الكتلة (تخزين تلقائي)

```
void func(void) {
    int x = 42;
}
```

توجد الكائنات فقط أثناء تنفيذ الكتلة.

نطاق الملف (تخزين ساكن)

```
int global_value = 100;
```

توجد الكائنات طوال مدة تنفيذ البرنامج.

٤.٢.٢ الثوابت وعدم القابلية للتغيير

كائنات مؤهلة بـ const

```
const int max_connections = 32;
```

لا يمكن تعديل الكائن من خلال معرّفه.

ثوابت المُعالج القبلي

```
#define BUFFER_SIZE 1024
```

تُجري الماكروهات استبدالاً نصيًا ولا تُنشئ كائنات ذات نوع.

٥.٢.٢ معدّلات الأنواع

تُحدّد معدّلات الأنواع خصائص الأنواع الأساسية.

- `signed`، `unsigned`
- `short`، `long`

```
unsigned int flags;
long double precision;
```

٦.٢.٢ تحويل الأنواع

التحويلات الضمنية

تُنفَّذ تلقائيًا وفق قواعد اللغة.

```
int i = 10;
double d = i;
```

التحويلات الصريحة

تفرض التحويل بشكلٍ صريح.

```
double x = 3.7;
int y = (int)x;
```

يجب استخدام التحويلات الصريحة بحذر وعن قصد.

٧.٢.٢ أمثلة عملية

الأنواع القياسية

```c
#include <stdio.h>

int main(void) {
    int age = 30;
    double salary = 72000.5;
    bool active = true;

    printf("%d %.2f %d\n", age, salary, active);
    return 0;
}
```

المصفوفات

```c
#include <stdio.h>

int main(void) {
    int a[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; ++i) {
        printf("%d ", a[i]);
    }
    return 0;
}
```

٨.٢.٢ الملخص

يحدّد نظام الأنواع في C23 كيفية تمثيل البيانات وتخزينها ومعالجتها على مستوى الآلة. ويُعد فهم أنواع البيانات، والمتغيرات، ومدة التخزين، وقواعد التحويل، أمرًا أساسيًا لكتابة برامج C صحيحة وفعّالة.

تشكل هذه المعرفة الأساس للاستدلال على تخطيط الذاكرة، والأداء، والصحة البرمجية—وهي موضوعات تزداد أهميةً كلما تعمّقنا في البرمجة منخفضة المستوى وبرمجة الأنظمة.

٣.٢ المعاملات والتعبيرات

تحدد المعاملات والتعبيرات كيفية حساب القيم وإحداث الآثار الجانبية في برامج C. وفي C، لا تُعد التعبيرات مجرد صيغ رياضية، بل ترتبط ارتباطًا وثيقًا بنموذج الآلة المجرد، وقواعد التقييم، وقيود التتابع المعرّفة في المعيار.
إن فهم المعاملات في C23 لا يقتصر على حفظ الرموز، بل يتطلب فهم كيف ومتى تُقيَّم المعاملات، وكيف تنتشر النتائج عبر البرنامج.

١.٣.٢ المعاملات والتعبيرات

- المعاملات تحدد العمليات المطبّقة على المعاملات (كائنات، أو قيم، أو تعبيرات).
- التعبيرات تجمع بين المعاملات والمعاملات التابعة لها لإنتاج قيمة، أو أثر جانبي، أو كليهما.

كل تعبير في C يمتلك:

- نوعًا
- قيمةً (باستثناء تعبيرات void)
- آثارًا جانبية محتملة

٢.٣.٢ فئات المعاملات في C23

المعاملات الحسابية

تُطبَّق المعاملات الحسابية على المعاملات العددية.

المعامل	المعنى	مثال
+	الجمع	b + a
-	الطرح	b - a
*	الضرب	b * a
/	القسمة	b / a
%	باقي القسمة (للأعداد الصحيحة فقط)	b % a

```
int a = 10, b = 3;
int q = a / b;   // 3
int r = a % b;   // 1
```

في القسمة الصحيحة، يتم التقريب نحو الصفر.

معاملات المقارنة

تقارن معاملات المقارنة القيم وتُنتج نتيجة صحيحة (0 أو 1).

المعامل	المعنى	مثال
==	يساوي	b == a
=!	لا يساوي	b =! a
>	أصغر من	b > a
=>	أصغر أو يساوي	b => a
<	أكبر من	b < a
=<	أكبر أو يساوي	b =< a

```
if (x < y) {
    /* condition holds */
}
```

المعاملات المنطقية

تدمج المعاملات المنطقية الشروط وتنفذ تقييماً قصير الدائرة.

المعامل	المعنى	مثال
&&	و المنطقية	a && b
||	أو المنطقية	a || b
!	النفي المنطقي	!a

```
if (ptr != NULL && *ptr > 0) {
    /* safe: second operand evaluated only if first is true */
}
```

معاملات الإسناد

تُعدِّل معاملات الإسناد الكائنات وتُنتج قيمة.

المعامل	المعنى	مثال
=	إسناد	a = b
=+	جمع ثم إسناد	a =+ b
=-	طرح ثم إسناد	a =- b
=*	ضرب ثم إسناد	a =* b
=/	قسمة ثم إسناد	a =/ b
=%	باقي ثم إسناد	a =% b

```
int x = 5;
x += 2;   // x becomes 7
```

معاملات البِتّات

تتعامل معاملات البِتّات مع البتّات الفردية للأعداد الصحيحة.

المعامل	المعنى	مثال
&	و بتّية	b & a
|	أو بتّية	b | a
^	أو حصري بتّي	b ^ a
~	نفي بتّي	a
>>	إزاحة لليسار	1 >> a
<<	إزاحة لليمين	1 << a

```
unsigned x = 5;        // 0101
unsigned y = x << 1; // 1010
```

الزيادة والنقصان

```
int i = 10;
++i;   // prefix
i++;   // postfix
```

يُعدّل الشكل السابق القيمة أولاً، بينما يُعيد الشكل اللاحق القيمة القديمة.

المعامل الشرطي

يختار المعامل الشرطي بين تعبيرين.

```
int max = (a > b) ? a : b;
```

يُقيَّم فرع واحد فقط.

٣.٣.٢ الأولوية، الترابط، والتتابع

أولوية المعاملات

تحدد الأولوية كيفية تجميع التعبيرات نحويًا.

```
int x = 5 + 3 * 2;    // 11
```

الترابط

يحدد الترابط كيفية تجميع المعاملات عند تساوي الأولوية.

التتابع

الأولوية لا تحدد ترتيب التقييم.
التعبير التالي يمتلك سلوكًا غير معرّف:

```
i = i++ + 1;  // undefined behavior
```

يحدد C23 قواعد التتابع صراحةً، والاعتماد على ترتيب ضمني خطأ جسيم.

٤.٣.٢ أمثلة عملية

المنطقي مقابل البتّي

```
int a = 5, b = 3;

printf("%d\n", a && b);  // logical AND
printf("%d\n", a & b);   // bitwise AND
```

التعبير الشرطي

```
int abs = (x < 0) ? -x : x;
```

٥.٣.٢ الملخص

تحدد المعاملات والتعبيرات كيفية حساب البيانات، ومقارنتها، وتعديلها في برامج C. ويتطلب إتقان هذا الموضوع فهم المعاملات نفسها، إضافةً إلى الأولوية، والترابط، والآثار الجانبية، وقواعد التتابع. يُعد الاستخدام الصحيح للتعبيرات أمرًا أساسيًا لكتابة شيفرة C23 آمنة وفعّالة وقابلة للتنبؤ، خصوصًا في سياقات البرمجة منخفضة المستوى وبرمجة الأنظمة.

٤.٢ تدفق التحكم: الشروط والحلقات

تحدد تراكيب تدفق التحكم ترتيب تنفيذ التعليمات. وفي C23، يكون تدفق التحكم صريحًا، وبسيطًا، وقريبًا من نموذج الآلة. لا توجد حمايات ضمنية، ولا فحوصات وقت تشغيل مخفية، ولا شبكات أمان تلقائية.

يتطلب إتقان تدفق التحكم فهم الصياغة، وطريقة تقييم التعبيرات، وتفسير القيم المنطقية، وكيفية انتقال التنفيذ داخل البرنامج.

١.٤.٢ التعليمات الشرطية

تختار التعليمات الشرطية بين مسارات تنفيذ بديلة اعتمادًا على قيمة تعبير. وفي C، يمكن لأي تعبير صحيح أن يُستخدم شرطًا: الصفر يعني خطأ، وأي قيمة غير صفرية تعني صحيح.

تعليمة if

تنفذ تعليمة if كتلة تعليمات عندما يُقيَّم الشرط على أنه صحيح.

```
if (condition) {
    /* executed if condition != 0 */
}
```

```
int age = 18;
if (age >= 18) {
    printf("Eligible to vote\n");
}
```

يُقيَّم الشرط مرةً واحدة فقط.

عبارة else

تُنفّذ عبارة else الاختيارية عندما يُقيَّم شرط if على أنه خطأ.

```
if (condition) {
    /* true branch */
} else {
    /* false branch */
}
```

```
int age = 16;
if (age >= 18) {
    printf("Adult\n");
} else {
    printf("Minor\n");
}
```

سلسلة if else

يمكن اختبار عدة شروط بالتتابع باستخدام if else. وتُقيَّم الشروط من الأعلى إلى الأسفل، ويتوقف التنفيذ عند أول تطابق صحيح.

```
int score = 85;

if (score >= 90) {
    printf("Grade A\n");
} else if (score >= 80) {
    printf("Grade B\n");
} else if (score >= 70) {
```

```
    printf("Grade C\n");
} else {
    printf("Fail\n");
}
```

تعليمة switch

تختار تعليمة switch فرع التنفيذ اعتماداً على قيمة تعبير صحيح.

```
switch (expression) {
    case constant:
        /* statements */
        break;
    default:
        /* fallback */
}
```

```
int day = 3;

switch (day) {
    case 1:
        printf("Monday\n");
        break;
    case 2:
        printf("Tuesday\n");
        break;
    case 3:
        printf("Wednesday\n");
        break;
```

```
    default:
        printf("Invalid day\n");
}
```

السقوط التتابعي

في حال غياب break، يستمر التنفيذ إلى الحالة التالية. ويُعد هذا السلوك مقصوداً، لكنه خطير إن لم يُوثَّق.

```
switch (x) {
    case 1:
    case 2:
        printf("One or two\n");
        break;
}
```

٢.٤.٣ الحلقات

تنفذ الحلقات كتلة تعليمات بشكل متكرر طالما تحقق الشرط. توفر C23 ثلاثة تراكيب حلقات بدلالات تقييم مختلفة.

حلقة for

تُعد حلقة for الأنسب للتكرار العددي.

```
for (initialization; condition; iteration) {
    /* loop body */
}
```

```
for (int i = 0; i < 5; ++i) {
    printf("%d\n", i);
}
```

يمكن حذف أي من التعبيرات الثلاثة، لكن حذف شرط التكرار ينشئ حلقة لا نهائية.

حلقة while

تختبر حلقة while الشرط قبل كل دورة.

```
int i = 0;

while (i < 5) {
    printf("%d\n", i);
    ++i;
}
```

إذا كان الشرط خاطئًا في البداية، فلن تُنفَّذ الحلقة.

حلقة do-while

تضمن حلقة do-while تنفيذًا واحدًا على الأقل.

```
int i = 0;

do {
    printf("%d\n", i);
    ++i;
} while (i < 5);
```

يُقيَّم الشرط بعد تنفيذ جسم الحلقة.

٣.٤.٢ أخطاء شائعة في تدفق التحكم

- الـ else المعلّقة: يجب استخدام الأقواس دائمًا لتجنب اللالتباس.
- نسيان break في switch: يؤدي إلى سقوط غير مقصود.
- الحلقات اللانهائية: تنتج عن شروط لا تتغير.
- الآثار الجانبية في الشروط: تقلل المقروئية وتزيد احتمال الخطأ.

٤.٤.٢ أفضل الممارسات

- استخدم الأقواس دائمًا، حتى مع تعليمة واحدة.
- اجعل الشروط بسيطة وخالية من الآثار الجانبية.
- فضّل switch للقيم المنفصلة.
- قلّل التداخل باستخدام الإرجاع المبكر أو إعادة الهيكلة.
- وثّق السقوط المقصود صراحةً.

٥.٤.٢ أمثلة عملية

شروط متداخلة

```
int age = 20;
char gender = 'M';

if (age >= 18) {
    if (gender == 'M') {
        printf("Adult male\n");
    } else {
```

```
        printf("Adult female\n");
    }
} else {
    printf("Minor\n");
}
```

التكرار على مصفوفة

```
int numbers[] = {10, 20, 30, 40, 50};
int sum = 0;

for (int i = 0; i < 5; ++i) {
    sum += numbers[i];
}

printf("Sum = %d\n", sum);
```

التحقق من الإدخال باستخدام while

```
int value;

do {
    printf("Enter a positive number: ");
    scanf("%d", &value);
} while (value <= 0);
```

٦.٤.٢ الملخص

تحدد تراكيب تدفق التحكم كيفية اتخاذ القرار وتنفيذ التكرار في برامج C. وفي C23، تكون الشروط والحلقات صريحة وقابلة للتنبؤ وقريبة من نموذج الآلة.

يُعد الاستخدام الصحيح لتدفق التحكم أمرًا أساسيًا لكتابة برامج مقروءة، وقابلة للصيانة، وموثوقة—خصوصًا في البرمجة منخفضة المستوى وبرمجة الأنظمة. ويُعد إتقان هذه التراكيب شرطًا مسبقًا لفهم الدوال، ومعالجة الأخطاء، وبنية البرامج في C الحديثة.

٥.٢ الإدخال والإخراج في C23

يشكّل الإدخال والإخراج (I/O) الحدّ الفاصل بين البرنامج والعالم الخارجي. وفي C23، صُمِّم الإدخال والإخراج ليكون صريحًا ومنخفض المستوى عن قصد، بما يعكس جذور اللغة في برمجة الأنظمة. لا توجد تدفقات ضمنية، ولا تحويلات تلقائية، ولا قواعد تخزين مؤقت مخفية.
يعتمد الإدخال والإخراج القياسي في C على مفهوم التدفّقات (streams)، وهي تمثل تسلسلات مرتّبة من البايتات مرتبطة بأجهزة مثل وحدة التحكم، أو الملفات، أو الأنابيب.

١.٥.٢ نموذج الإدخال والإخراج القياسي

تتفاعل برامج C مع البيانات الخارجية أساسًا من خلال مكتبة stdio.h. تُعرّف هذه المكتبة ما يلي:

- التدفّقات القياسية: stdin، stdout، stderr
- دوال الإدخال والإخراج المُنسّق: printf، scanf، fprintf، fscanf
- دوال الإدخال والإخراج غير المُنسّق: fread، fwrite، fgets، fputs

يُمثَّل كل تدفّق بكائن FILE * يدير التخزين المؤقت، وموضع القراءة أو الكتابة، وحالة الخطأ.

٢.٥.٢ إخراج وحدة التحكم باستخدام printf

تكتب الدالة printf إخراجًا مُنسّقًا إلى تدفّق الإخراج القياسي (stdout).

```
printf("format string", arguments);
```

محدّدات التنسيق

- %d — عدد صحيح بإشارة
- %u — عدد صحيح بدون إشارة

- %f — عدد بفاصلة عائمة
- %c — محرف
- %s — سلسلة منتهية بمحرف صفري
- %p — قيمة مؤشر
- %x — عدد صحيح ست عشري

```
int age = 25;
float height = 5.9f;

printf("Age: %d, Height: %.2f\n", age, height);
```

تُعيد الدالة printf عدد المحارف المكتوبة، أو قيمة سالبة في حال حدوث خطأ.

٣.٥.٢ إدخال وحدة التحكم باستخدام scanf

تقرأ الدالة scanf إدخالاً مُنسّقًا من stdin.

```
scanf("format string", &variable);
```

```
int age;
float height;

if (scanf("%d %f", &age, &height) == 2) {
    printf("Age: %d, Height: %.2f\n", age, height);
}
```

ملاحظات مهمة

- لا تقوم scanf بتخصيص الذاكرة
- تؤدي سلاسل التنسيق غير الصحيحة إلى سلوك غير معرّف
- يجب دائمًا فحص قيمة الإرجاع

٤.٥.٢ الإخراج المُنسّق إلى السلاسل والملفات

sprintf و snprintf

تكتب الدالة sprintf بيانات مُنسّقة إلى مصفوفة محارف، دون إجراء فحص للحدود.

```
char buffer[64];
sprintf(buffer, "Value: %d", 42);
```

لشيفرة أكثر أمانًا، يُفضّل استخدام snprintf:

```
snprintf(buffer, sizeof(buffer), "Value: %d", 42);
```

fprintf

تكتب الدالة fprintf إخراجًا مُنسّقًا إلى تدفّق ملف.

```
FILE *file = fopen("output.txt", "w");

if (file) {
    fprintf(file, "Age: %d\n", 25);
    fclose(file);
}
```

٥.٥.٢ الإدخال والإخراج من الملفات

يتم الإدخال والإخراج من الملفات في C عبر تدفّقات مرتبطة بالملفات.

فتح الملفات وإغلاقها

```
FILE *file = fopen("filename", "mode");
```

تشمل الأوضاع الشائعة:

- "r" — قراءة
- "w" — كتابة (مع التفريغ)
- "a" — إلحاق
- "rb"، "wb" — أوضاع ثنائية

يجب دائمًا إغلاق الملفات صراحةً:

```
fclose(file);
```

قراءة الملفات النصية

```
FILE *file = fopen("input.txt", "r");

if (file) {
    char buffer[128];
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);
    }
    fclose(file);
}
```

كتابة الملفات النصية

```
FILE *file = fopen("output.txt", "w");

if (file) {
    fprintf(file, "Hello, C23!\n");
    fclose(file);
}
```

٦.٥.٢ الإدخال والإخراج الثنائي

يتعامل الإدخال والإخراج الثنائي مع كتل الذاكرة الخام دون تفسير.

كتابة بيانات ثنائية

```
int data[] = {1, 2, 3, 4, 5};

FILE *file = fopen("data.bin", "wb");
if (file) {
    fwrite(data, sizeof(int), 5, file);
    fclose(file);
}
```

قراءة بيانات ثنائية

```
int data[5];

FILE *file = fopen("data.bin", "rb");
if (file) {
```

```
    fread(data, sizeof(int), 5, file);
    fclose(file);
}
```

يعتمد الإدخال والإخراج الثنائي على المعمارية، وهو حساس لترتيب البايتات (endianness)، والمحاذاة، وأحجام الأنواع.

٧.٥.٢ معالجة الأخطاء في الإدخال والإخراج

قد تفشل عمليات الإدخال والإخراج بسبب ملفات غير موجودة، أو أخطاء أذونات، أو بيانات غير صالحة.

```
FILE *file = fopen("input.txt", "r");

if (!file) {
    perror("Failed to open file");
    return 1;
}
```

- استخدم قيم الإرجاع لاكتشاف حالات الفشل
- استخدم feof() لاختبار نهاية الملف
- استخدم ferror() لاكتشاف أخطاء التدفّق

٨.٥.٢ مثال عملي

```
#include <stdio.h>

int main(void) {
```

```
    FILE *file = fopen("example.txt", "w");
    if (!file) return 1;

    fprintf(file, "C23 I/O example\n");
    fclose(file);

    file = fopen("example.txt", "r");
    if (!file) return 1;

    char buffer[64];
    fgets(buffer, sizeof(buffer), file);
    printf("%s", buffer);

    fclose(file);
    return 0;
}
```

٩.٥.٢ الملخص

يُعد الإدخال والإخراج في C23 صريحًا، وقويًا، وقريبًا من النظام الكامن. وتوفّر مكتبة stdio.h نموذجًا مرنًا قائمًا على التدفّقات يدعم الإدخال والإخراج المُنسّق وغير المُنسّق، والبيانات النصية والثنائية، والتفاعل مع الملفات والأجهزة.

يتطلب التعامل الصحيح مع الإدخال والإخراج الانتباه الدقيق لقيم الإرجاع، وأحجام المخازن، وحالات الخطأ. ويُعد إتقان هذه المفاهيم أساسيًا لكتابة برامج C متينة، وآمنة، وقابلة للنقل — خصوصًا في برمجة الأنظمة، والأنظمة المضمّنة، والبرمجيات منخفضة المستوى.

الفصل ٣: الدوال في C23

١.٣ تعريف الدوال واستدعاؤها

تُعد الدوال الوحدات الأساسية للتجريد والتنفيذ في لغة C. فهي تتيح تنظيم البرامج في مكونات منطقية، وتمكّن من إعادة استخدام الشيفرة، وتُعرّف واجهات واضحة بين الأجزاء المختلفة للبرنامج. وفي C23، صُمِّمت الدوال لتكون بسيطة وصريحة عن قصد، بما يعكس فلسفة اللغة القائمة على الشفافية والتحكم.

يستعرض هذا الفصل كيفية تعريف الدوال، والتصريح عنها، واستدعائها في C23، ويشرح كيفية تفاعل المعاملات، وقيم الإرجاع، والذاكرة أثناء استدعاءات الدوال.

١.١.٣ ما هي الدالة في C؟

الدالة هي كتلة شيفرة مسماة تنفّذ مهمة محددة. قد تستقبل صفرًا أو أكثر من المدخلات (المعاملات)، وقد تُعيد قيمة واحدة إلى الجهة المستدعية.

ومن منظور برمجة الأنظمة، تمثل الدالة ما يلي:

- منطقة شيفرة قابلة للاستدعاء
- واجهة استدعاء محددة بوضوح
- حدًا فاصلاً للنطاق، والعمر، واستخدام المكدس

يؤدي كل استدعاء لدالة إلى إنشاء سياق تنفيذ جديد، يُنفَّذ عادةً باستخدام مكدس الاستدعاءات (call stack).

٢.١.٣ تعريف دالة

يحدد تعريف الدالة نوع الإرجاع، واسم الدالة، والمعاملات، وتنفيذها.

الصياغة العامة

```
return_type function_name(parameter_list) {
    // Function body
    return value;  // Required for non-void functions
}
```

- **return_type** — نوع القيمة المعادة إلى الجهة المستدعية
- **function_name** — المعرّف المستخدم لاستدعاء الدالة
- **parameter_list** — المدخلات الممرّرة إلى الدالة مع أنواعها
- **function body** — التعليمات القابلة للتنفيذ

مثال: دالة بسيطة

```
int add(int a, int b) {
    return a + b;
}
```

تستقبل هذه الدالة عددين صحيحين وتُعيد مجموعهما.

٣.١.٣ استدعاء دالة

يتم استدعاء الدالة بكتابة اسمها متبوعاً بقائمة من الوسائط بين قوسين.

```
result = function_name(arguments);
```

مثال

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main(void) {
    int result = add(5, 10);
    printf("Sum: %d\n", result);
    return 0;
}
```

أثناء استدعاء الدالة:

- تُقيَّم الوسائط
- تُمرَّر القيم إلى المعاملات
- ينتقل التحكم إلى جسم الدالة
- تُنتج قيمة إرجاع (إن وُجدت)

٤.١.٣ معاملات الدوال وتمرير الوسائط

تستخدم C حصريًا دلالات التمرير بالقيمة (pass-by-value). أي أن كل معامل يستقبل نسخة من قيمة الوسيط.

التمرير بالقيمة

```c
void increment(int x) {
```

```
    x++;
}

int main(void) {
    int a = 5;
    increment(a);
    printf("%d\n", a);  // a remains 5
    return 0;
}
```

لا يتغير المتغير الأصلي لأن الدالة تعمل على نسخة.

محاكاة التمرير بالمرجع باستخدام المؤشرات

لتعديل بيانات الجهة المستدعية، يجب تمرير مؤشر بشكل صريح.

```
void increment(int *x) {
    (*x)++;
}

int main(void) {
    int a = 5;
    increment(&a);
    printf("%d\n", a);  // a is now 6
    return 0;
}
```

يجعل هذا الأسلوب تعديل البيانات صريحًا ومرئيًا في موقع الاستدعاء.

٥.١.٣ قيم الإرجاع

يمكن للدوال إرجاع قيمة واحدة باستخدام تعليمة return.

إرجاع قيمة

```
int square(int x) {
    return x * x;
}
```

```
int result = square(5);  // result is 25
```

تُنسخ القيمة المعادة إلى الجهة المستدعية.

الدوال من نوع void

تستخدم الدوال التي لا تُعيد قيمة نوع الإرجاع void.

```
void greet(void) {
    printf("Hello, World!\n");
}
```

```
greet();
```

٦.١.٣ نماذج الدوال (Function Prototypes)

يُصرّح نموذج الدالة عن واجهتها دون تقديم تنفيذها. وتُعد النماذج ضرورية عند استخدام الدوال قبل تعريفها.

صياغة النموذج

```
return_type function_name(parameter_list);
```

مثال

```c
#include <stdio.h>

int add(int a, int b);  // Prototype

int main(void) {
    printf("%d\n", add(5, 10));
    return 0;
}

int add(int a, int b) {
    return a + b;
}
```

تمكّن نماذج الدوال من:

- التحقق من أنواع الوسائط
- الفصل بين الواجهة والتنفيذ
- تنظيم البرامج متعددة الملفات

٧.١.٣ أفضل الممارسات في كتابة الدوال

- استخدام أسماء وصفية تكشف عن النية

- حصر كل دالة في مسؤولية واحدة
- تفضيل تمرير البيانات صراحةً على استخدام المتغيرات العامة
- التصريح عن النماذج في الملفات الرأسية
- توثيق المعاملات وقيم الإرجاع بوضوح

٨.١.٣ أمثلة عملية

مثال على دالة

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

تبديل القيم باستخدام المؤشرات

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int x = 10, y = 20;
swap(&x, &y);
```

٩.١.٣ الملخص

توفّر الدوال في C23 آلية بسيطة لكنها قوية لبناء البرامج والتحكم في تدفق التنفيذ. ومن خلال فهم كيفية تعريف الدوال، واستدعائها، وتمرير البيانات بينها، يكتسب المبرمج رؤية مباشرة لاستخدام المكدس، وسلوك الذاكرة، وبنية البرنامج.
ويُعد إتقان الدوال أمرًا أساسيًا لكتابة برامج C وحدوية، وفعّالة، وقابلة للصيانة، خصوصًا في برمجة الأنظمة، والأنظمة المضمّنة، والبرمجيات الحساسة للأداء.

٢.٣ وسائط الدوال وقيم الإرجاع

تُعرّف وسائط الدوال وقيم الإرجاع آلية التواصل الأساسية بين الدوال وبقية برنامج C. فهي تحدد حدود تدفق البيانات بوضوح، وتقرر ملكية القيم وأعمارها، وتؤثر مباشرةً في الأداء والصحة البرمجية. يقدم هذا القسم شرحًا دقيقًا وموجّهًا لبرمجة الأنظمة لكيفية تمرير الوسائط وإرجاع القيم في C23، بما في ذلك آثار دلالات التمرير بالقيمة، وتبادل البيانات باستخدام المؤشرات، وأنماط التصميم الشائعة لإرجاع نتائج معقدة.

١.٢.٣ وسائط الدوال

وسائط الدوال (وتُسمّى رسميًا المعاملات) هي قيم مسماة تستقبل البيانات من الجهة المستدعية. في C، صُمِّمت قواعد تمرير الوسائط لتكون بسيطة وصريحة عن قصد.

دلالات التمرير بالقيمة

تستخدم C حصريًا التمرير بالقيمة. عند استدعاء دالة، تُنسخ قيمة كل وسيط إلى معامل الدالة.

```
void increment(int x) {
    x++;
    printf("Inside function: %d\n", x);
}
```

```
int main(void) {
    int a = 5;
    increment(a);
    printf("Outside function: %d\n", a);
    return 0;
}
```

- المعامل x هو نسخة محلية
- التعديلات لا تؤثر على الجهة المستدعية
- هذا السلوك مضمون بواسطة نموذج لغة C

يمنع هذا التصميم الآثار الجانبية الخفية ويجعل استدعاءات الدوال قابلة للتنبؤ.

تمرير الوسائط باستخدام المؤشرات

للسماح للدالة بتعديل بيانات مملوكة للجهة المستدعية، يجب تمرير مؤشر بشكل صريح.

```
void increment(int *x) {
    (*x)++;
}
```

```
int main(void) {
    int a = 5;
    increment(&a);
    printf("%d\n", a);
    return 0;
}
```

هنا تعمل الدالة على ذاكرة مملوكة للجهة المستدعية، مما يجعل التعديل صريحًا ومرئيًا في موقع الاستدعاء.

تمرير المصفوفات إلى الدوال

في C، تتحلل المصفوفات إلى مؤشرات عند تمريرها كوسائط. تستقبل الدالة مؤشرًا إلى أول عنصر في المصفوفة.

```c
void print_array(const int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```c
int main(void) {
    int numbers[] = {1, 2, 3, 4, 5};
    print_array(numbers, 5);
    return 0;
}
```

استخدام const يعبّر عن النية ويمنع التعديل غير المقصود.

٣.٢.٣ قيم الإرجاع

يمكن للدالة إرجاع قيمة واحدة فقط. تُنسخ هذه القيمة إلى الجهة المستدعية باستخدام اتفاقية الاستدعاء (calling convention) الخاصة بالمنصة.

إرجاع قيمة واحدة

```c
int square(int x) {
    return x * x;
}
```

```c
int result = square(5);
```

القيم المعادة هي قيم عادية، وليست مراجع ولا أسماء مستعارة.

الدوال من نوع void

تستخدم الدوال التي لا تُنتج نتيجة نوع الإرجاع void.

```
void greet(void) {
    printf("Hello, World!\n");
}
```

٣.٢.٣ إرجاع عدة قيم: أنماط تصميم

لا تدعم C إرجاع عدة قيم مباشرةً. بدلاً من ذلك، يستخدم المبرمجون أنماطًا راسخة.

إرجاع بنية

```
struct Point {
    int x;
    int y;
};

struct Point make_point(int x, int y) {
    struct Point p = { x, y };
    return p;
}
```

```
struct Point p = make_point(10, 20);
```

تعالج واجهات ABI الحديثة إرجاع البنى الصغيرة بكفاءة عالية.

معاملات إخراج باستخدام المؤشرات

```
void get_min_max(const int arr[], int size, int *min, int *max) {
    *min = *max = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < *min) *min = arr[i];
        if (arr[i] > *max) *max = arr[i];
    }
}
```

```
int min, max;
get_min_max(numbers, 5, &min, &max);
```

يُستخدم هذا النمط على نطاق واسع في مكتبات الأنظمة وواجهات API.

٤.٢.٣ أفضل الممارسات لوسائط الدوال وقيم الإرجاع

- تفضيل إرجاع القيم على تعديل المدخلات
- استخدام المؤشرات فقط عند الحاجة للتعديل
- تعليم المعاملات غير القابلة للتعديل بـ const
- تجنب الاعتماد الخفي على الحالة العامة
- توثيق الملكية وتوقعات العمر بوضوح

٥.٢.٣ الدوال

الاستدعاء الذاتي (recursion) هو أسلوب تستدعي فيه الدالة نفسها لحل مشكلة عن طريق تقسيمها إلى مشكلات أصغر. في C، يكون الاستدعاء الذاتي صريحًا ومرتبطًا مباشرةً بسلوك المكدس.

بنية الدالة

```
return_type function(parameters) {
    if (base_case) {
        return value;
    }
    return function(smaller_problem);
}
```

يجب أن تُعرّف كل دالة حالة توقف (base case) لإنهاء التنفيذ.

مثال: المضروب

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

يستهلك كل استدعاء إطار مكدس جديد.

مثال: فيبوناتشي

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

هذا التنفيذ بسيط، لكنه غير فعّال بسبب تكرار الاستدعاءات.

٦.٢.٣ مزايا وحدود الاستدعاء الذاتي

المزايا

- تعبير طبيعي عن مشكلات التقسيم والحل
- تمثيل واضح لهياكل البيانات
- تحسين الوضوح المفاهيمي

الحدود

- ازدياد استهلاك المكدس مع العمق
- خطر تجاوز المكدس
- كلفة أداء أعلى مقارنةً بالتكرار

٧.٢.٣ الاستدعاء الذاتي الطرفي

تكون الدالة ذات استدعاء ذاتي طرفي عندما يكون الاستدعاء آخر عملية في الدالة.

```
int factorial_tr(int n, int acc) {
    if (n <= 1) {
        return acc;
    }
    return factorial_tr(n - 1, n * acc);
}
```

قد تُحسّن بعض المترجمات هذا النمط، لكن C لا تضمن ذلك.

٨.٢.٣ الملخص

تشكل وسائط الدوال وقيم الإرجاع العمود الفقري للتصميم الوحدوي في C23. ومن خلال فهم التمرير بالقيمة، وتبادل البيانات باستخدام المؤشرات، وأنماط الإرجاع المنظمة، يمكن تصميم واجهات واضحة، وفعّالة، وقابلة للصيانة.

يوفر الاستدعاء الذاتي قوة تعبيرية إضافية عند استخدامه بحكمة، لكن يجب تطبيقه بوعي كامل لاستهلاك المكدس ومقايضات الأداء. ويُعد إتقان هذه المفاهيم أساسيًا للبرمجة الاحترافية بلغة C، خصوصًا في برمجة الأنظمة، والأنظمة المضمّنة، والبرمجيات منخفضة المستوى.

الفصل ٤: المؤشرات وإدارة الذاكرة

٤.١ فهم المؤشرات

تُعد المؤشرات إحدى السمات الجوهرية للغة البرمجة C. فهي تتيح وصولاً مباشرًا إلى الذاكرة، مما يسمح بالتحكم الدقيق في تخطيط البيانات، والأداء، وموارد النظام. وفي الوقت نفسه، تُدخل المؤشرات قدرًا كبيرًا من التعقيد، لأنها تكشف المبرمج على أعمار الكائنات، وقواعد التداخل (aliasing), وسلوك غير معرّف (undefined behavior).
يقدّم هذا القسم شرحًا دقيقًا وحديثًا للمؤشرات كما يعرّفها معيار C23، مرتكزًا على كيفية نمذجة الذاكرة في الآلة التجريدية للغة C وكيفية تنفيذها على العتاد الحقيقي.

٤.١.١ ما هو المؤشر؟

المؤشر هو كائن تمثل قيمته عنوان كائن آخر أو دالة.
والأهم من ذلك: المؤشر لا يخزن القيمة نفسها، بل يخزن موقعًا توجد فيه القيمة.

- الكائن: منطقة تخزين لها نوع وعمر
- العنوان: الموقع التجريدي الذي يحدد ذلك الكائن
- المؤشر: قيمة ذات نوع تشير إلى ذلك العنوان

في C، يرتبط كل مؤشر بنوع الكائن الذي يشير إليه، وهذا النوع يحدد كيفية تفسير الذاكرة المشار إليها.

٢.١.٤ تصريح وتهيئة المؤشرات

صيغة تصريح المؤشر

```
type *pointer_name;
```

ترتبط النجمة بالمُصرِّح وليس بالنوع.

```
int *p;      // p       int
int* q;      //
```

التهيئة باستخدام عنوان

يجب تهيئة المؤشرات بعنوان صالح أو تعيينها صراحةً إلى NULL.

```
int x = 10;
int *ptr = &x;
```

يُرجع العامل & عنوان كائن له تخزين فعلي.

٣.١.٤ فك الإشارة (Dereferencing)

يتيح فك الإشارة الوصول إلى الكائن الذي يشير إليه المؤشر.

```
*ptr
```

```
int x = 10;
int *ptr = &x;
printf("%d\n", *ptr); // 10
```

يكون فك الإشارة صالحًا فقط إذا:

- كان المؤشر غير فارغ
- كان الكائن المشار إليه حيًا
- كان المؤشر محاذيًا بشكل صحيح

مخالفة أي من هذه الشروط تؤدي إلى سلوك غير معرّف.

٤.١.٤ حساب المؤشرات

يُعرَّف حساب المؤشرات فقط داخل حدود المصفوفات.

```
ptr + n
ptr - n
```

يزيد الإزاحة بعدد العناصر وليس بعدد البايتات.

```
int arr[] = {10, 20, 30};
int *p = arr;

p++; //       arr[1]
```

يُسمح بحساب المؤشرات فقط ضمن:

- عناصر المصفوفة نفسها
- العنصر التالي مباشرةً لآخر عنصر (ولا يجوز فك الإشارة إليه)

٥.١.٤ المؤشرات والمصفوفات

في معظم التعابير، يتحوّل اسم المصفوفة إلى مؤشر يشير إلى أول عنصر فيها.

```
int arr[5];
int *p = arr;
```

ومع ذلك:

- المصفوفات ليست مؤشرات
- sizeof(arr) يختلف عن sizeof(p)

```
sizeof(arr); //
sizeof(p);   //
```

٦.١.٤ المؤشرات إلى مؤشرات

يمكن أن تكون المؤشرات نفسها كائنات يُشار إليها.

```
int x = 10;
int *p = &x;
int **pp = &p;
```

```
printf("%d\n", **pp); // 10
```

يشيع هذا النمط في:

- واجهات التخصيص الديناميكي
- هياكل البيانات متعددة المستويات
- معاملات الإخراج

٧.١.٤ عمر الكائن والمؤشرات المعلّقة

يكون المؤشر صالحًا فقط طالما أن الكائن الذي يشير إليه ما زال حيًا.

```
int *p;
{
    int x = 5;
    p = &x;
}
*p = 10; //
```

انتهى عمر x بخروج نطاقه.

٨.١.٤ الذاكرة الديناميكية والمؤشرات

ينشئ التخصيص الديناميكي كائنات يُتحكم في أعمارها صراحةً من قبل البرنامج.

```
int *p = malloc(sizeof *p);
```

بعد استدعاء free، يصبح المؤشر غير صالح.

```
free(p);
p = NULL; //
```

٩.١.٤ أخطاء المؤشرات الشائعة

مؤشرات غير مهيأة

```
int *p;
*p = 10; //
```

مؤشرات معلّقة

```
int *p = malloc(sizeof *p);
free(p);
*p = 5; //
```

تسرّب الذاكرة

```
void leak(void) {
    int *p = malloc(sizeof *p);
}
```

١٠.١.٤ الصحة الثابتة مع المؤشرات

```
const int *p;   //
int *const q;   //
const int *const r;
```

استخدام const يوثق النية ويمنع التعديل غير المقصود.

١١.١.٤ أفضل الممارسات لاستخدام المؤشرات

- تهيئة المؤشرات فورًا
- تعيين المؤشرات المحررة إلى NULL
- احترام أعمار الكائنات
- تجنب حساب المؤشرات خارج المصفوفات
- استخدام const كلما أمكن

- تقليل التداخل (aliasing)

١٢.١.٤ أمثلة عملية

تبديل القيم

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

اجتياز مصفوفة باستخدام المؤشرات

```
int arr[] = {1, 2, 3, 4};
for (int *p = arr; p != arr + 4; ++p) {
    printf("%d\n", *p);
}
```

١٣.١.٤ الملخص

تشكل المؤشرات أساس قوة لغة C ومصدر تعقيدها الأساسي. فهي تكشف الذاكرة، وأعمار الكائنات، وقواعد التداخل بشكل مباشر للمبرمج. ويتطلب إتقان المؤشرات فهمًا عميقًا ليس للصياغة فقط، بل لقواعد الآلة التجريدية للغة C.
وعند استخدامها بانضباط ودقة، تمكّن المؤشرات من تحكم منخفض المستوى فعّال، وقابل للتنبؤ، ولا تضاهيه معظم اللغات الأخرى.

٢.٤ الحسابات والعمليات على المؤشرات

يُعد الحساب على المؤشرات إحدى السمات المميزة للغة C، إذ يسمح للمبرمجين بالتنقل داخل الذاكرة بكفاءة عالية باستخدام مؤشرات ذات أنواع محددة. ويشكّل هذا المفهوم الأساس لاجتياز المصفوفات، ومعالجة السلاسل النصية، ومعالجة البيانات منخفضة المستوى. ومع ذلك، فإن الحساب على المؤشرات يُعد أيضًا من أكثر أجزاء اللغة تقييدًا وسوء فهمًا.
يقدّم هذا القسم الحساب على المؤشرات تمامًا كما يعرّفه معيار C23، موضحًا العمليات المسموح بها، والافتراضات التي يبنيها المترجم، والمواضع التي يظهر فيها السلوك غير المعرّف.

١.٢.٤ القاعدة الأساسية للحساب على المؤشرات

يُعرَّف الحساب على المؤشرات في C فقط بالنسبة للمصفوفات.
يمكن للمؤشر بشكل قانوني أن:

- يشير إلى عنصر داخل مصفوفة
- يشير إلى عنصر واحد بعد آخر عنصر في نفس المصفوفة

تُعبَّر جميع عمليات الحساب على المؤشرات بوحدات نوع العنصر المشار إليه، وليس بالبايتات.

٢.٢.٤ جمع عدد صحيح مع مؤشر

عند جمع عدد صحيح مع مؤشر، يتقدّم المؤشر بعدد من العناصر يساوي قيمة العدد الصحيح.

```
pointer + integer
```

مثال

```
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr;
```

```
ptr = ptr + 1;
printf("%d\n", *ptr); // 20
```

الشرح

إذا كان ptr يشير إلى arr[i]، فإن ptr + n يشير إلى arr[i + n]. داخليًا، يضرب المترجم القيمة n في sizeof(int)، لكن هذا تفصيل تنفيذي غير مرئي للمبرمج.

٣.٢.٤ طرح عدد صحيح من مؤشر

يؤدي طرح عدد صحيح إلى تحريك المؤشر للخلف داخل نفس المصفوفة.

```
pointer - integer
```

مثال

```
int *ptr = &arr[3];
ptr = ptr - 2;
printf("%d\n", *ptr); // 20
```

يجب أن تبقى النتيجة داخل حدود نفس المصفوفة (أو في موضع العنصر الواحد بعد النهاية).

٤.٢.٤ طرح مؤشرين من بعضهما

يُرجع طرح مؤشرين عدد العناصر الفاصلة بينهما.

```
ptrdiff_t diff = pointer1 - pointer2;
```

القواعد

- يجب أن يشير المؤشران إلى نفس المصفوفة
- تُقاس النتيجة بعدد العناصر لا بالبايتات
- نوع النتيجة هو ptrdiff_t

مثال

```c
#include <stddef.h>

int arr[] = {10, 20, 30, 40, 50};
int *p1 = &arr[4];
int *p2 = &arr[1];

ptrdiff_t diff = p1 - p2;
printf("%td\n", diff); // 3
```

٥.٢.٤ مقارنة المؤشرات

تُعرَّف المقارنات العلاقية بين المؤشرات فقط عندما يشير كلا المؤشرين إلى عناصر داخل نفس المصفوفة.

```c
if (p1 < p2) { /* ... */ }
```

مثال

```c
int *p1 = &arr[1];
int *p2 = &arr[3];
```

```
if (p1 < p2) {
    printf("p1 precedes p2\n");
}
```

مقارنة مؤشرات تنتمي إلى مصفوفات مختلفة تؤدي إلى سلوك غير معرّف.

٦.٢.٤ فك الإشارة وصلاحية المؤشر

يتيح فك الإشارة الوصول إلى الكائن الذي يشير إليه المؤشر.

```
*pointer
```

يكون فك الإشارة صالحًا فقط إذا:

- كان المؤشر يشير إلى كائن حي
- لم يكن المؤشر عند موضع ما بعد النهاية
- لم ينتهِ عمر الكائن

٧.٢.٤ اجتياز المصفوفات باستخدام الحساب على المؤشرات

يُستخدم الحساب على المؤشرات غالبًا لاجتياز المصفوفات.

```
int arr[] = {10, 20, 30, 40, 50};

for (int *p = arr; p != arr + 5; ++p) {
    printf("%d\n", *p);
}
```

التعبير `arr + 5` قانوني، لكن لا يجوز أبدًا فك الإشارة إليه.

٨.٢.٤ الحساب على المؤشرات مع السلاسل النصية

السلاسل النصية هي مصفوفات من المحارف تنتهي بالمحرف '
.'0

```
const char *p = "Hello";

while (*p) {
    putchar(*p++);
}
```

يُعد اجتياز السلاسل النصية باستخدام المؤشرات أسلوبًا شائعًا وفعّالاً في C.

٩.٢.٤ الحساب على المؤشرات مع الذاكرة المخصصة ديناميكيًا

تتصرف الذاكرة التي تُرجعها malloc كمصفوفة من البايتات كبيرة بما يكفي لاحتواء عدد العناصر المطلوب.

```
int *arr = malloc(5 * sizeof *arr);

for (int i = 0; i < 5; ++i) {
    arr[i] = i * 10;
}
```

يجب أن تبقى جميع العمليات ضمن حدود الكتلة المخصصة.

١٠.٢.٤ قاعدة العنصر الواحد بعد النهاية

تسمح C صراحةً بأن يشير المؤشر إلى عنصر واحد بعد نهاية المصفوفة.

```
int *end = arr + size;
```

هذا المؤشر:

- يمكن مقارنته
- لا يجوز فك الإشارة إليه

١١.٣.٤ الأخطاء الشائعة والسلوك غير المعرّف

فك الإشارة خارج الحدود

```
int *p = arr + 5;
printf("%d\n", *p); //
```

الحساب على المؤشرات بين كائنات مستقلة

```
int a, b;
int *p = &a;
p++; //
```

حتى لو كانت الكائنات متجاورة في الذاكرة، فإن المعيار يمنع ذلك.

افتراضات نوع غير صحيحة

```
int x = 0x11223344;
char *p = (char *)&x;
p++; //
```

هذا ليس خطأ محاذاة، بل إعادة تفسير للنوع.

١٢.٢.٤ أفضل الممارسات للحساب على المؤشرات

- نفّذ الحساب فقط داخل المصفوفات أو الكتل المخصصة
- استخدم ptrdiff_t لفروق المؤشرات
- لا تفك الإشارة إلى مؤشر عند موضع ما بعد النهاية
- فضّل حدود الحلقات الواضحة على الحيل الذكية
- وثّق افتراضات تخطيط الذاكرة

١٣.٢.٤ أمثلة عملية

عكس مصفوفة

```
void reverse(int *arr, size_t n) {
    int *l = arr;
    int *r = arr + n - 1;

    while (l < r) {
        int t = *l;
        *l++ = *r;
        *r-- = t;
    }
}
```

حساب طول سلسلة نصية

```
size_t strlen_ptr(const char *s) {
    const char *p = s;
```

```
    while (*p) ++p;
    return (size_t)(p - s);
}
```

١٤.٢.٤ الملخص

الحساب على المؤشرات ليس حسابًا عامًا على الذاكرة، بل هو تنقّل منظّم داخل المصفوفات كما تعرّفها الآلة التجريدية للغة C. وعند استخدامه بشكل صحيح، يتيح اجتيازًا عالي الأداء دون أي كلفة إضافية. أما عند إساءة استخدامه، فإنه يؤدي مباشرةً إلى سلوك غير معرّف.

يتطلب إتقان الحساب على المؤشرات انضباطًا، واحترامًا صارمًا لحدود الكائنات، وفهمًا واضحًا لما يضمنه معيار C وما لا يضمنه.

٣.٤ التخصيص الديناميكي للذاكرة (malloc, calloc, realloc, free)

يُعد التخصيص الديناميكي للذاكرة إحدى القدرات الجوهرية للغة C. إذ يسمح للبرامج بإنشاء الكائنات، وتغيير حجمها، وتدميرها أثناء وقت التشغيل، مما يتيح بناء هياكل بيانات مرنة والاحتفاظ بالذاكرة لفترات أطول من مدة التخزين التلقائي.
وعلى عكس التخصيص عبر المكدس، تُدار الذاكرة المخصصة ديناميكيًا بشكلٍ صريح من قبل المبرمج. وتأتي هذه القوة مصاحبةً لقواعد صارمة تحكم عمر الكائن، وملكيته، وصحة الوصول إليه. إن انتهاك هذه القواعد يؤدي مباشرةً إلى سلوك غير معرّف.
يعرض هذا القسم التخصيص الديناميكي للذاكرة كما يعرّفه معيار C23، مع التركيز على الصحة، والسلامة، والاستخدام الاحترافي.

١.٣.٤ لماذا توجد الذاكرة الديناميكية في C

يُستخدم التخصيص الديناميكي عندما:

- لا يكون حجم الذاكرة المطلوب معروفًا وقت الترجمة
- يجب أن تعيش الكائنات بعد انتهاء الدالة التي أنشأتها
- تحتاج هياكل البيانات إلى التوسع أو الانكماش ديناميكيًا
- يجب نقل ملكية الذاكرة عبر حدود واجهات API

تُخزَّن الذاكرة المخصصة ديناميكيًا في المخزن الحر (ويُسمّى غالبًا heap)، والذي يُدار بواسطة مخصّص الذاكرة في بيئة تشغيل C.

٢.٣.٤ الدالة malloc

تُخصّص الدالة malloc كتلة خامًا من الذاكرة غير المُهيّأة.

```
void *malloc(size_t size);
```

الدلالات

- تخصّص على الأقل size بايت
- تُرجع مؤشرًا مُحاذيًا بشكلٍ مناسب لأي نوع كائن
- تُرجع NULL عند فشل التخصيص
- لا تقوم بتهيئة الذاكرة

نمط الاستخدام الصحيح

```
int *arr = malloc(5 * sizeof *arr);
if (!arr) {
    /* handle allocation failure */
}
```

ملاحظة: لا حاجة لتحويل نوع القيمة المعادة من malloc في لغة C، ويُعد ذلك ممارسة غير مستحسنة.

٣.٣.٤ الدالة calloc

تخصّص calloc ذاكرة لمصفوفة وتقوم بتهيئة جميع البايتات بالصفر.

```
void *calloc(size_t count, size_t size);
```

الدلالات

- تخصّص count × size بايت
- تهيئة كاملة بالصفر
- تمنع فيض الأعداد الصحيحة داخليًا

مثال

```c
int *arr = calloc(5, sizeof *arr);
if (!arr) {
    /* allocation failed */
}
```

تُضمن التهيئة بالصفر لجميع البِتّات، مما يجعل calloc مفيداً للمصفوفات من الأنواع العددية.

٤.٣.٤ الدالة realloc

تُعيد realloc تغيير حجم كتلة ذاكرة مخصّصة سابقاً.

```c
void *realloc(void *ptr, size_t new_size);
```

الدلالات الحرجة

- قد تنقل الكتلة إلى عنوان جديد
- تحافظ على البيانات الموجودة حتى الحد الأدنى بين الحجم القديم والجديد
- تُحرّر الكتلة الأصلية عند النجاح
- تترك الكتلة الأصلية دون تغيير عند الفشل

النمط الصحيح والآمن

```c
int *tmp = realloc(arr, 10 * sizeof *arr);
if (!tmp) {
    free(arr);
    /* handle failure */
```

```
}
arr = tmp;
```

التعيين المباشر للمؤشر الأصلي قد يؤدي إلى تسرب في الذاكرة.

٥.٣.٤ الدالة free

تُحرّر free الذاكرة المخصصة ديناميكيًا وتعيدها إلى المخصّص.

```
void free(void *ptr);
```

القواعد

- يجب أن يكون المؤشر ناتجًا عن malloc أو calloc أو realloc
- يجب ألا يكون المؤشر قد حُرّر سابقًا
- تمرير NULL آمن ولا يؤدي إلى أي إجراء

٦.٣.٤ عمر الكائن والملكية

تُدخل الذاكرة الديناميكية إدارةً صريحة لعمر الكائن:

- يبدأ العمر عند التخصيص
- ينتهي العمر عند استدعاء free
- الوصول خارج هذا النطاق يُعد سلوكًا غير معرّف

يجب تحديد ملكية الذاكرة بوضوح ونقلها بشكلٍ مقصود.

٧.٣.٤ الأخطاء الشائعة والسلوك غير المعرّف

تسرب الذاكرة

```
int *p = malloc(sizeof *p);
/* lost pointer - leak */
```

الاستخدام بعد التحرير

```
free(p);
*p = 42; // undefined behavior
```

التحرير المزدوج

```
free(p);
free(p); // undefined behavior
```

تحرير غير صالح

```
int x;
free(&x); // undefined behavior
```

٨.٣.٤ أفضل الممارسات للتخصيص الديناميكي

- خصّص فقط ما تحتاجه
- استخدم sizeof *ptr بدل الأنواع المرمّزة يدويًا
- تحقّق فورًا من نتائج التخصيص

- حرّر الذاكرة في نفس مستوى التجريد الذي خصّصها
- اضبط المؤشرات على NULL بعد التحرير
- قلّل عمليات realloc في المسارات الحرجة

٩.٣.٤ مثال عملي: مصفوفة ديناميكية

```
int *create_array(size_t n) {
    int *arr = malloc(n * sizeof *arr);
    if (!arr) return NULL;

    for (size_t i = 0; i < n; ++i) {
        arr[i] = (int)(i * 10);
    }
    return arr;
}
```

١٠.٣.٤ مثال عملي: مخزن قابل للنمو

```
size_t cap = 4;
size_t len = 0;
int *buf = malloc(cap * sizeof *buf);

if (!buf) return;

for (int i = 0; i < 10; ++i) {
    if (len == cap) {
        cap *= 2;
```

```
        int *tmp = realloc(buf, cap * sizeof *buf);
        if (!tmp) {
            free(buf);
            return;
        }
        buf = tmp;
    }
    buf[len++] = i;
}

free(buf);
```

١١.٣.٤ الملخص

يُعد التخصيص الديناميكي للذاكرة ركيزةً أساسية في قوة ومرونة لغة C. فهو يتيح بناء هياكل البيانات، والتحكم في عمر الكائنات، وبرمجة الأنظمة منخفضة المستوى، لكنه يتطلب انضباطًا ودقة عالية.

في هذا القسم، تناولنا malloc وcalloc وrealloc وfree من منظور عمر الكائن، والملكية، والسلوك غير المعرّف. إن إتقان إدارة الذاكرة الديناميكية ليس خيارًا في برمجة C الاحترافية، بل هو أساس لا غنى عنه.

٤.٤ أنماط شبيهة بالمؤشرات الذكية في C23

يُستخدم مصطلح المؤشر الذكي غالبًا في سياق C++، حيث تتيح ميزات اللغة مثل المُدمِّرات وأسلوب RAII إدارة الموارد بشكلٍ تلقائي وحتمي. أما لغة C، بما في ذلك C23، فلا توفّر مفهوم المؤشرات الذكية كميزة لغوية مدمجة.

ومع ذلك، فقد استخدم مبرمجو C منذ زمن طويل أنماطًا صريحة لإدارة الملكية والتنظيف تحقق أهدافًا مشابهة: منع تسرب الذاكرة، وتجنّب التحرير المزدوج، وجعل أعمار الموارد واضحة وقابلة للتحقق.

يعرض هذا القسم تجريدات شبيهة بالمؤشرات الذكية في C23، مع التركيز على ما هو ممكن، وما هو آمن، وما لا ينبغي الخلط بينه وبين دلالات C++.

١.٤.٤ ماذا تعني «المؤشرات الذكية» في سياق C

في لغة C، لا تُعد المؤشرات الذكية ميزة لغوية تلقائية، بل هي نمط برمجي يجمع بين:

- ملكية صريحة للذاكرة المخصصة ديناميكيًا
- دالة تنظيف محددة بوضوح
- بروتوكول منضبط لعمر الكائن

وبخلاف C++، لا تدعم C مُدمِّرات ولا تنظيفًا تلقائيًا عند الخروج من النطاق، ولذا تعتمد هذه الأنماط على تنظيف يدوي لكن منظم.

٢.٤.٤ تصميم مقابض قائمة على الملكية

يُعد تغليف المؤشر الخام مع دالة تنظيف داخل بنية أسلوبًا شائعًا وآمنًا. تمثل هذه البنية ملكية حصرية للمورد.

مقبض ملكية أساسي

```
typedef struct {
    void *ptr;
    void (*destroy)(void *);
} owned_ptr;
```

تعبّر هذه البنية عن ثابِتين مهمّين:

- للمورد مالك واحد فقط
- التدمير صريح وحتمي

٣.٤.٤ تعريف دالة تنظيف

يعرّف كل نوع مورد دالة التنظيف الخاصة به.

```
#include <stdlib.h>

void destroy_int_array(void *p) {
    free(p);
}
```

٤.٤.٤ إنشاء مورد مملوك

تُحدَّد الملكية عند لحظة الإنشاء.

```
owned_ptr make_int_array(size_t count) {
    owned_ptr o = {0};
```

```
    o.ptr = malloc(count * sizeof(int));
    if (!o.ptr) {
        o.destroy = NULL;
        return o;
    }

    o.destroy = destroy_int_array;
    return o;
}
```

٥.٤.٤ استخدام المورد وتحريره

يستخدم المستدعي المورد بشكلٍ اعتيادي ويحرّره بصورة صريحة.

```
#include <stdio.h>

int main(void) {
    owned_ptr arr = make_int_array(5);
    if (!arr.ptr) {
        return 1;
    }

    int *data = arr.ptr;
    for (int i = 0; i < 5; ++i) {
        data[i] = i * 10;
    }

    for (int i = 0; i < 5; ++i) {
        printf("%d ", data[i]);
```

```
    }
    printf("\n");

    arr.destroy(arr.ptr);
    arr.ptr = NULL;

    return 0;
}
```

٦.٤.٤ ما الذي يضمنه هذا النمط

- نقطة تحرير واحدة فقط
- دلالات واضحة لنقل الملكية
- لا تحرير خفي
- لا اعتماد على سحر المترجم

٧.٤.٤ ما الذي لا يوفّره هذا النمط

من الضروري فهم القيود:

- لا تنظيف تلقائي عند الخروج من النطاق
- لا عدّ مراجع
- لا أمان استثنائي (لأن C لا تدعم الاستثناءات)
- لا ضمانات عمر كائن مفروضة من المترجم

إطلاق وصف «مؤشر ذكي حقيقي» على هذا النمط يُعد مضلّلاً.

٨.٤.٤ لماذا يظل هذا مهمًا

رغم هذه القيود، يظل هذا النهج ذا قيمة لأنه:

- يوثّق الملكية بشكلٍ صريح
- يقلّل التسربات العرضية
- يتوسع جيدًا في الأنظمة الكبيرة
- يعكس أنماطًا مستخدمة في النوى، وقواعد البيانات، وبيئات التشغيل

تستخدم العديد من قواعد الشيفرة الإنتاجية في C تنويعات من هذا التصميم بدل الاعتماد على المؤشرات الخام فقط.

٩.٤.٤ نسخة متقدمة: ملكية قابلة للنقل فقط

لفرض الملكية الحصرية، يجب منع النسخ وفقًا للاتفاق.

```
owned_ptr owned_move(owned_ptr *src) {
    owned_ptr dst = *src;
    src->ptr = NULL;
    src->destroy = NULL;
    return dst;
}
```

يحاكي هذا الأسلوب دلالات النقل بشكلٍ صريح وآمن.

١٠.٤.٤ مقارنة مع المؤشرات الذكية في C++

الميزة	مقبض ملكية في C	مؤشر ذكي في C++
تنظيف تلقائي	لا	نعم
RAII	لا	نعم
عدّ المراجع	يدوي	مدمج
فرض من المترجم	لا	جزئي
قابلية التنبؤ	عالية	عالية
الكلفة	ضئيلة	متوسطة

١١.٤.٤ أفضل الممارسات

- فضّل الملكية الصريحة على الملكية المشتركة
- وحّد منطق التدمير
- تجنّب دوال «التحرير الخفي»
- لا تخلط نماذج الملكية
- وثّق قواعد العمر بوضوح

١٢.٤.٤ الملخص

لا تدعم C23 المؤشرات الذكية كميزة لغوية، ومحاولة تقليد دلالات C++ بشكل مباشر تؤدي إلى تصاميم هشّة. ومع ذلك، فإن التجريدات المبنية على الوعي بالملكية باستخدام البُنى ودوال التنظيف الصريحة توفر بديلاً منضبطًا وفعّالاً.

تحترم هذه الأنماط فلسفة C: تحكم صريح، تجريد محدود، وسلوك قابل للتنبؤ. وعند استخدامها بشكلٍ صحيح، تشكل الأساس لإدارة ذاكرة آمنة وقابلة للصيانة في الأنظمة الكبيرة المكتوبة بلغة C.

الفصل ٥: المصفوفات والسلاسل النصية

١.٥ العمل مع المصفوفات

تُعد المصفوفات إحدى أكثر تمثيلات البيانات أساسيةً في لغة C. فهي توفّر طريقةً مدمجة وفعّالة لتخزين عدة عناصر من نفس النوع داخل ذاكرة متجاورة. وعلى خلاف اللغات عالية المستوى، تكشف مصفوفات C عن تخطيطها في الذاكرة مباشرةً، مما يمنحها قوةً كبيرة ويقابلها قدرٌ عالٍ من المسؤولية.

يعرض هذا القسم المصفوفات كما هي فعليًا في C23: ككائنات ذاكرة ثابتة الحجم تخضع لقواعد صارمة تتعلق بالعمر والحدود. وبنهاية هذا القسم، ستفهم ليس فقط كيفية استخدام المصفوفات، بل كيف يفكّر النموذج التجريدي للغة C بشأنها.

١.١.٥ ما هي المصفوفة في C؟

المصفوفة في لغة C هي تسلسل من العناصر من نفس النوع مخزّنة في مواقع متجاورة في الذاكرة. المصفوفة نفسها كائن ذو حجم ثابت يُحدَّد وقت الترجمة (أو عند دخول الكتلة في حالة VLA).

- اسم المصفوفة يمثّل الكائن بالكامل وليس مؤشرًا
- كل عنصر يشغل تمامًا sizeof(type) بايت

- الفهرسة تبدأ من الصفر

٢.١.٥ تعريف المصفوفات

تُحدّد تصريحات المصفوفات نوع العنصر، واسم المصفوفة، وعدد العناصر.

الصيغة

```
element_type array_name[array_size];
```

- element_type: نوع كل عنصر
- array_size: تعبير ثابت (باستثناء VLA)

مثال

```
int numbers[5];
```

يحجز هذا التصريح مساحةً لخمسة أعداد صحيحة تمامًا في ذاكرة متجاورة.

٣.١.٥ تهيئة المصفوفات

يمكن تهيئة المصفوفات عند التصريح عنها باستخدام قوائم التهيئة.

التهيئة الكاملة

```
int numbers[5] = {10, 20, 30, 40, 50};
```

التهيئة الجزئية

تُهيَّأ العناصر غير المحددة بالصفر.

```
int numbers[5] = {10, 20};
```

استنتاج الحجم

عند وجود مُهيّئ، يمكن حذف حجم المصفوفة.

```
int numbers[] = {10, 20, 30, 40, 50};
```

٤.١.٥ الوصول إلى العناصر وتعديلها

يتم الوصول إلى عناصر المصفوفة باستخدام عامل الفهرسة.

```
array_name[index]
```

```
int numbers[5] = {10, 20, 30, 40, 50};
numbers[1] = 25;
printf("%d\n", numbers[1]);
```

٥.١.٥ اجتياز المصفوفات والعمليات الشائعة

تُعالج المصفوفات عادةً باستخدام الحلقات.

الاجتياز

```
for (int i = 0; i < 5; ++i) {
    printf("%d\n", numbers[i]);
}
```

البحث الخطي

```
int target = 30;
for (int i = 0; i < 5; ++i) {
    if (numbers[i] == target) {
        printf("Found at index %d\n", i);
        break;
    }
}
```

مثال على الفرز

```
void bubble_sort(int arr[], int size) {
    for (int i = 0; i < size - 1; ++i) {
        for (int j = 0; j < size - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
}
```

٦.١.٥ المصفوفات متعددة الأبعاد

تدعم لغة C المصفوفات التي تحتوي عناصرها على مصفوفات أخرى. وأكثر الأشكال شيوعًا هي المصفوفة ثنائية الأبعاد.

التصريح

```
int matrix[3][3];
```

التهيئة

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

الوصول

```
printf("%d\n", matrix[1][2]);
```

٧.١.٥ المصفوفات وواقع الذاكرة

المصفوفات ليست مؤشرات.

- اسم المصفوفة يشير إلى الكائن بالكامل
- لا يمكن إعادة إسناد المصفوفات

- sizeof(array) يُرجع الحجم الكلي للمصفوفة

ومع ذلك، في معظم التعابير، تتحلل المصفوفات إلى مؤشرات لأول عنصر فيها. هذا الفرق أساسي للفهم الصحيح.

٨.١.٥ الأخطاء الشائعة

الوصول خارج الحدود

```
int a[5];
a[5] = 42;  /* undefined behavior */
```

المصفوفات غير المهيّأة

```
int a[5];
printf("%d\n", a[0]); /* undefined behavior */
```

٩.١.٥ أفضل الممارسات

- هيّئ المصفوفات دائمًا
- لا تفترض وجود فحص للحدود
- مرّر حجم المصفوفة صراحةً إلى الدوال
- استخدم ثوابت للأحجام

```
#define ARRAY_SIZE 5
int numbers[ARRAY_SIZE];
```

١٠.١.٥ أمثلة عملية

جمع العناصر

```
int sum = 0;
for (int i = 0; i < 5; ++i) {
    sum += numbers[i];
}
```

إيجاد القيمة العظمى

```
int max = numbers[0];
for (int i = 1; i < 5; ++i) {
    if (numbers[i] > max) {
        max = numbers[i];
    }
}
```

١١.١.٥ الملخص

المصفوفات في C هي كائنات ذاكرة ثابتة الحجم ذات تطابق مباشر مع التخطيط الفيزيائي للذاكرة. وهي فعّالة، وقابلة للتنبؤ، وقوية، لكنها لا توفّر أي فحص تلقائي للحدود ولا إدارة تلقائية للعمر. إن فهم المصفوفات بوصفها ذاكرة وليس حاويات، أمرٌ أساسي لكتابة برامج C23 صحيحة، وعالية الأداء، وآمنة.

٢.٥ المصفوفات متعددة الأبعاد

المصفوفات متعددة الأبعاد في C هي مصفوفات تكون عناصرها مصفوفات أخرى. وهي توفّر طريقةً منظمة لتمثيل البيانات المرتّبة في أكثر من بُعد، مثل الجداول، والمصفوفات، والشبكات، والموترات. وأكثر الأشكال شيوعًا هي المصفوفة ثنائية الأبعاد، لكن C تدعم أي عدد من الأبعاد. وعلى خلاف اللغات عالية المستوى، تمتلك المصفوفات متعددة الأبعاد في C تخطيطًا ذاكراتيًا محددًا وقابلاً للتنبؤ. إن فهم هذا التخطيط ضروري لكتابة برامج C23 صحيحة، وفعّالة، وعالية الأداء.

١.٢.٥ ما هي المصفوفات متعددة الأبعاد؟

المصفوفة متعددة الأبعاد هي مصفوفة تكون كل عناصرها مصفوفات أخرى. مفهوميًا:

- المصفوفة ثنائية الأبعاد هي مصفوفة من مصفوفات أحادية البعد
- المصفوفة ثلاثية الأبعاد هي مصفوفة من مصفوفات ثنائية الأبعاد
- كل بُعد إضافي يُغلف البُعد السابق

فيزيائيًا، تُخزَّن جميع العناصر في كتلة واحدة متجاورة من الذاكرة باستخدام ترتيب الصفوف أولاً (row-major order).

٢.٢.٥ التصريح بالمصفوفات متعددة الأبعاد

المصفوفات ثنائية الأبعاد

```
element_type array_name[rows][columns];
```

- rows: عدد الصفوف
- columns: عدد العناصر في كل صف

```
int matrix[3][3];  /* 3 rows, 3 columns */
```

المصفوفات ثلاثية الأبعاد

```
element_type array_name[layers][rows][columns];
```

```
int cube[2][3][3];  /* 2 layers of 3x3 matrices */
```

٣.٢.٥ تخطيط الذاكرة: الصفوف أولاً

تخزّن لغة C المصفوفات متعددة الأبعاد بترتيب الصفوف أولاً. أي أن عناصر الصف الواحد تُخزَّن بشكل متجاور، ثم يليها الصف التالي.
لتصريح مثل:

```
int matrix[3][3];
```

يكون تخطيط الذاكرة كالتالي:

```
matrix[0][0] matrix[0][1] matrix[0][2]
matrix[1][0] matrix[1][1] matrix[1][2]
matrix[2][0] matrix[2][1] matrix[2][2]
```

يؤثر هذا التخطيط مباشرةً على حسابات المؤشرات، وسلوك الذاكرة المخبئية، وواجهات الدوال.

٤.٢.٥ تهيئة المصفوفات متعددة الأبعاد

التهيئة الكاملة

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

التهيئة الجزئية

تُهيَّأ العناصر غير المحددة بالصفر.

```
int matrix[3][3] = {
    {1, 2},
    {4, 5},
    {7, 8}
};
```

حذف البُعد الأول

يمكن حذف البُعد الأول فقط عند وجود مُهيّئ.

```
int matrix[][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

يستنتج المترجم عدد الصفوف تلقائيًا.

٥.٢.٥ الوصول إلى العناصر

يتطلب كل بُعد فهرسًا خاصًا به.

الوصول ثنائي الأبعاد

```
matrix[row][column]
```

```
printf("%d\n", matrix[1][2]);  /* prints 6 */
```

الوصول ثلاثي الأبعاد

```
cube[layer][row][column]
```

```
printf("%d\n", cube[1][2][1]);  /* prints 17 */
```

٦.٢.٥ تعديل العناصر

```
matrix[1][2] = 10;
```

تعمل الإسنادات مباشرةً على الذاكرة الأساسية.

٧.٢.٥ اجتياز المصفوفات متعددة الأبعاد

يُجرى الاجتياز عادةً باستخدام حلقات متداخلة.

```
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

٨.٢.٥ تمرير المصفوفات متعددة الأبعاد إلى الدوال

عند تمرير المصفوفات متعددة الأبعاد إلى الدوال، يجب تحديد جميع الأبعاد ما عدا الأول.

```
void process(int matrix[][3], int rows);
```

ويرجع هذا الشرط إلى حاجة المترجم لمعرفة حجم الصف لحساب الإزاحات الصحيحة في الذاكرة.

٩.٢.٥ مثال على عمليات المصفوفات

ضرب المصفوفات

```
void matrix_multiply(int a[2][2], int b[2][2], int result[2][2]) {
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            result[i][j] = 0;
            for (int k = 0; k < 2; ++k) {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

١٠.٢.٥ الأخطاء الشائعة

الوصول خارج الحدود

```
matrix[3][3];  /* undefined behavior */
```

المصفوفات غير المهيّأة

```
int matrix[3][3];
printf("%d\n", matrix[0][0]);  /* undefined behavior */
```

١١.٢.٥ أفضل الممارسات

- هيّئ المصفوفات متعددة الأبعاد دائمًا

- حافظ على الأبعاد صريحة ومتناسقة
- استخدم ثوابت مسماة للأحجام
- اجتز المصفوفات بترتيب الصفوف لتحسين أداء الذاكرة المخبئية

```
#define ROWS 3
#define COLS 3
int matrix[ROWS][COLS];
```

١٢.٢.٥ أمثلة عملية

جمع العناصر

```
int sum = 0;
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        sum += matrix[i][j];
    }
}
```

نقل المصفوفة (Transpose)

```
void transpose(int a[3][3], int result[3][3]) {
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            result[j][i] = a[i][j];
        }
    }
}
```

١٣.٢.٥ الملخص

المصفوفات متعددة الأبعاد في C هي كائنات ذاكرة منظمة ومتجاورة ذات تخطيط صارم وقابل للتنبؤ. وتنبع قوتها من التحكم المباشر بالذاكرة، لكن هذا التحكم نفسه يتطلب دقةً وانضباطًا. إن فهم كيفية تحويل الأبعاد إلى ذاكرة، وكيف تتحلل المصفوفات عند تمريرها إلى الدوال، وكيف يؤثر ترتيب الاجتياز على الأداء، أمرٌ أساسي للوصول إلى مستوى احترافي في برمجة C23.

٣.٥ السلاسل النصية ومعالجتها

تُعد السلاسل النصية إحدى التجريدات الأساسية في برامج C، وتُستخدم لتمثيل النصوص وتسلسلات المحارف. وعلى خلاف العديد من اللغات عالية المستوى، لا توفّر C نوع بيانات مخصصًا للسلاسل النصية. بدلاً من ذلك، تُنفَّذ السلاسل النصية على شكل مصفوفات من المحارف منتهية بمحرف صفري ('\٥').

يمنح هذا التصميم مبرمجي C تحكمًا كاملاً في تخطيط الذاكرة والأداء، لكنه في المقابل يحمّلهم مسؤولية إدارة الحدود، وضمان وجود المنهي الصفري، وضبط عمر البيانات بشكل صحيح. يقدّم هذا القسم نظرة شاملة على السلاسل النصية في C23، بما في ذلك تمثيلها، وتهيئتها، ومعالجتها، وأكثر الأخطاء الشائعة المرتبطة بها.

١.٣.٥ ما هي السلسلة النصية في C؟

السلسلة النصية في C هي تسلسل متجاور من المحارف يتبعه محرف إنهاء صفري ('\٥').

- المنهي الصفري يحدّد نهاية السلسلة
- جميع دوال السلاسل القياسية تعتمد عليه
- لا يُحسب المنهي الصفري ضمن طول السلسلة

من دون المنهي الصفري، لا تُعد مصفوفة المحارف سلسلةً نصيةً صحيحة في C.

٢.٣.٥ تعريف وتهيئة السلاسل النصية

تعريف مصفوفة محارف

```
char name[20];
```

يحجز هذا التعريف مساحةً لما يصل إلى 19 محرفًا بالإضافة إلى محرف الإنهاء الصفري.

التهيئة باستخدام ثابت نصي

```
char name[] = "John Doe";
```

- يضيف المترجم محرف الإنهاء '\0' تلقائيًا عند تعريف السلسلة.
- يُستنتج حجم المصفوفة مساويًا لـ `strlen("John Doe") + 1` لاحتساب محرف الإنهاء.

التهيئة محرفًا بمحرف

```
char name[20] = {'J', 'o', 'h', 'n', '\0'};
```

تجعل هذه الصيغة المنهي الصفري صريحًا وواضحًا.

٣.٣.٥ الثوابت النصية وقابلية التعديل

تُخزَّن الثوابت النصية في ذاكرة للقراءة فقط في معظم الأنظمة.

```
char *p = "Hello";   /* modifying *p is undefined behavior */
char s[] = "Hello"; /* safe: stored in writable memory */
```

يجب دائمًا استخدام مصفوفة إذا كان من الضروري تعديل السلسلة النصية.

٤.٣.٥ الوصول إلى المحارف وتعديلها

تُعامل السلاسل النصية كمصفوفات عادية.

```
char name[] = "John Doe";
printf("%c\n", name[0]);  /* J */
name[0] = 'j';
```

٥.٣.٥ دوال السلاسل النصية القياسية

تُعرّف واجهة السلاسل النصية القياسية في الترويسة <string.h>.

طول السلسلة (strlen)

```
size_t strlen(const char *str);
```

تُرجع عدد المحارف قبل المنهي الصفري.

نسخ السلاسل النصية

```
char *strcpy(char *dest, const char *src);
```

تحذير: لا تجري strcpy أي فحص للحدود.

```
char dest[20];
strcpy(dest, "Hello");
```

النسخ المحدود (strncpy)

```
char *strncpy(char *dest, const char *src, size_t n);
```

- لا تضمن إنهاء السلسلة بمحرف صفري
- يُساء استخدامها كثيرًا على أنها بديل ``آمن''

الاستخدام الصحيح:

```
strncpy(dest, src, sizeof(dest) - 1);
dest[sizeof(dest) - 1] = '\0';
```

الربط (Concatenation)

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

يجب أن تكون السلسلة الوجهة منتهيةً بمنهي صفري ومساحتها كافية.

المقارنة

```
int strcmp(const char *a, const char *b);
int strncmp(const char *a, const char *b, size_t n);
```

المقارنة معجمية وليست عددية.

البحث

```
char *strstr(const char *haystack, const char *needle);
char *strchr(const char *str, int c);
```

٦.٣.٥ الأخطاء الشائعة

تجاوز حدود الذاكرة

```
char name[5] = "John";
strcpy(name, "John Doe"); /* undefined behavior */
```

غياب المنهي الصفري

```
char name[4] = {'J','o','h','n'};
printf("%s\n", name); /* undefined behavior */
```

افتراض طول ثابت

جميع السلاسل النصية في C متغيرة الطول ويجب التحقق منها ديناميكيًا.

٧.٣.٥ أفضل الممارسات

- احجز دائمًا مساحة للمنهي الصفري
- تعامل مع المدخلات الخارجية على أنها غير موثوقة
- تتبّع أحجام المخازن صراحةً
- فضّل العمليات المعتمدة على الطول
- تجنّب تعديل الثوابت النصية

٨.٣.٥ أمثلة عملية

عكس سلسلة نصية

```
void reverse(char *s) {
    size_t len = strlen(s);
    for (size_t i = 0; i < len / 2; ++i) {
        char tmp = s[i];
        s[i] = s[len - i - 1];
        s[len - i - 1] = tmp;
    }
}
```

عدّ الكلمات

```
int count_words(const char *s) {
    int count = 0;
    int in_word = 0;

    while (*s) {
        if (*s == ' ' || *s == '\n' || *s == '\t') {
            in_word = 0;
        } else if (!in_word) {
            in_word = 1;
            ++count;
        }
        ++s;
    }
    return count;
}
```

٩.٣.٥ الملخص

سلاسل C منخفضة المستوى، فعّالة، وقوية — لكنها غير متسامحة مع الأخطاء. إن فهم المنهي الصفري، وملكية الذاكرة، وحدود المخازن أمرٌ أساسي لكتابة برامج C23 آمنة وصحيحة. إتقان السلاسل النصية ليس خيارًا في برمجة الأنظمة — بل هو أساسٌ لا غنى عنه.

٤.٥ دوال السلاسل النصية الشائعة في C23

توفر مكتبة C القياسية مجموعةً شاملة من الدوال لمعالجة السلاسل النصية، وجميعها معرّفة في الترويسة <string.h>. تعمل هذه الدوال على مصفوفات محارف منتهية بمنهي صفري، وتشكّل الأساس لمعالجة النصوص في برامج C.
وعلى خلاف اللغات عالية المستوى، لا تجري دوال السلاسل النصية أي فحص تلقائي للحدود. ولذلك، يتطلب استخدامها الصحيح فهمًا عميقًا لأحجام المخازن، وقواعد الإنهاء، وملكية الذاكرة. يعرض هذا القسم أكثر الدوال استخدامًا في C23، وسلوكها، وأنماط الاستخدام الصحيحة لها.

١.٤.٥ طول السلسلة (strlen)

تحسب strlen عدد المحارف في السلسلة، باستثناء المنهي الصفري.

```
size_t strlen(const char *str);
```

- زمن التنفيذ خطي: O(n)
- تتطلب وجود منهي صفري صحيح

```
printf("%zu\n", strlen("Hello")); /* prints 5 */
```

٢.٤.٥ نسخ السلاسل النصية

strcpy

```
char *strcpy(char *dest, const char *src);
```

تنسخ حتى الوصول إلى أول منهي صفري.
خطر: لا تجري أي فحص للحدود.

strncpy

```
char *strncpy(char *dest, const char *src, size_t n);
```

خصائص مهمة:

- لا تضمن إنهاءً صفريًا
- تملأ الباقي بالأصفار إذا كان المصدر أقصر
- يُساء فهمها على أنها ``آمنة''

٣.٤.٥ الربط النصي

strcat

```
char *strcat(char *dest, const char *src);
```

strncat

```
char *strncat(char *dest, const char *src, size_t n);
```

٤.٤.٥ تحويل السلاسل إلى أرقام

الدوال القديمة

```
int atoi(const char *str);
double atof(const char *str);
```

الدوال الموصى بها

```c
long strtol(const char *str, char **endptr, int base);
double strtod(const char *str, char **endptr);
```

٥.٤.٥ الملخص

توفر دوال السلاسل النصية في C23 قوةً وكفاءةً عالية، لكنها لا تقدّم أي ضمانات أمان تلقائية. وتقع مسؤولية الصحة على عاتق المبرمج بالكامل. إتقان هذه الواجهات أمرٌ أساسي في برمجة الأنظمة، والشبكات، وبناء المحللات، وتطوير أنظمة التشغيل بلغة C23.

الفصل ٦: الهياكل والاتحادات

١.٦ تعريف الهياكل واستخدامها

تُعد الهياكل (Structures) من أهم آليات التجريد في لغة C. إذ تتيح تجميع قيم متعددة من أنواع مختلفة ضمن وحدة منطقية واحدة، مما يسمح بنمذجة البيانات بشكل تعبيري مع الحفاظ على تحكم كامل في تخطيط الذاكرة.

في C23، تظل الهياكل بسيطة، وقابلة للتنبؤ، وعالية الكفاءة. وهي أنواع قيم (Value Types) ذات قواعد تخطيط محددة بوضوح، وسلوك عمر حياة حتمي. يقدّم هذا القسم الهياكل من منظور برمجة الأنظمة: التعريف، والتهيئة، والوصول، ودلالات الذاكرة، وأنماط الاستخدام الصحيحة.

١.١.٦ ما هي البنية؟

البنية هي نوع تجميعي مُعرّف من قبل المستخدم، يتكوّن من أعضاء مسمّاة، يمكن أن يختلف نوع كل منها. تُخزَّن جميع الأعضاء بشكل متجاور في الذاكرة، مع مراعاة قواعد المحاذاة والحشو (Alignment and Padding) التي يحددها ABI.

تُستخدم الهياكل على نطاق واسع لتمثيل الكيانات الواقعية، ورسائل البروتوكولات، وسجلات العتاد، وحالة البرامج الداخلية.

٦.١.٢ تعريف بنية

تُعرّف البنية باستخدام الكلمة المفتاحية struct متبوعة بقائمة تعريف الأعضاء.

```
struct Person {
    char name[50];
    int age;
    float height;
};
```

- ترتيب الأعضاء يحدّد تخطيط الذاكرة
- قد يُضاف حشو لأغراض المحاذاة
- لا توجد بُناة ولا تهيئة تلقائية

٦.١.٣ تعريف كائنات من بنية

بعد التعريف، يمكن إنشاء كائنات من البنية كما هو الحال مع أي نوع آخر.

```
struct Person p;
```

ويمكن تعريف عدة كائنات في سطر واحد:

```
struct Person p1, p2;
```

٤.١.٦ التعريف والتصريح معًا

يمكن تعريف البنية وإنشاء كائنات منها في تعليمة واحدة:

```
struct Person {
    char name[50];
    int age;
    float height;
} person1, person2;
```

٥.١.٦ تهيئة البنى

يمكن تهيئة البنى باستخدام قوائم تهيئة محاطة بالأقواس المعقوفة.

```
struct Person p = {"John Doe", 30, 5.9f};
```

التهيئة المعيّنة

يدعم معيار C23 التهيئة المعيّنة (Designated Initializers) التي تحسّن الوضوح وتقلّل الأخطاء.

```
struct Person p = {
    .name = "John Doe",
    .age = 30,
    .height = 5.9f
};
```

تُهيّأ الأعضاء غير المذكورة تلقائيًا بالصفر.

٦.١.٦ الوصول إلى أعضاء البنية

يتم الوصول إلى أعضاء البنية باستخدام معامل النقطة.

```
printf("%s\n", p.name);
printf("%d\n", p.age);
```

٧.١.٦ تعديل أعضاء البنية

يمكن تعديل الأعضاء مباشرةً.

```
p.age = 31;
```

٨.١.٦ الهياكل المتداخلة

يمكن أن تحتوي البنى على بنى أخرى كأعضاء.

```
struct Address {
    char city[32];
    int zip;
};

struct Person {
    char name[50];
    struct Address address;
};
```

يتم الوصول وفق قواعد التركيب الطبيعية:

```
printf("%s\n", p.address.city);
```

٩.١.٦ مصفوفات من البنى

تتكامل البنى بشكل طبيعي مع المصفوفات.

```
struct Person people[3] = {
    {"John", 30, 5.9f},
    {"Jane", 25, 5.5f},
    {"Alice", 28, 5.7f}
};
```

١٠.١.٦ المؤشرات إلى البنى

يمكن الوصول إلى البنى عبر المؤشرات باستخدام معامل السهم.

```
struct Person *ptr = &p;
printf("%s\n", ptr->name);
```

معامل السهم ليس سوى صيغة مختصرة لـ:

```
(*ptr).name
```

١١.١.٦ دلالات القيم والنسخ

تتمتع البنى في C بدلالات قيم (Value Semantics). أي أن الإسناد ينسخ جميع الأعضاء نسخًا سطحيًا محرفًا بمحرف.

```
struct Person a = p;
```

تُنسخ المؤشرات كعناوين فقط، ولا تُنسخ البيانات التي تشير إليها.

١٢.١.٦ الأخطاء الشائعة

بنى غير مهيّأة

```
struct Person p;
printf("%d\n", p.age); /* undefined behavior */
```

مؤشرات غير صالحة

```
struct Person *ptr;
ptr->age = 30; /* undefined behavior */
```

١٣.١.٦ أفضل الممارسات

- هيّئ البنى دائمًا
- فضّل التهيئة المعيّنة
- مرّر البنى الكبيرة عبر مؤشرات
- وثّق ملكية الأعضاء المؤشرية
- حافظ على تماسك البنية منطقيًا

١٤.١.٦ استخدام typedef لتحسين القراءة

```
typedef struct {
    char name[50];
    int age;
    float height;
```

```
} Person;

Person p = {"John Doe", 30, 5.9f};
```

١٥.١.٦ مثال عملي: الهندسة

```
struct Point {
    int x;
    int y;
};

struct Rectangle {
    struct Point top_left;
    struct Point bottom_right;
};
```

١٦.١.٦ الملخص

الهياكل هي أداة نمذجة البيانات الأساسية في C. فهي توفّر تجميعًا فعّالاً، وتخطيطًا متوقّعًا، وتحكمًا صريحًا في الذاكرة. وإتقانها ضروري لبرمجة الأنظمة، وتصميم البروتوكولات، والتعامل مع العتاد، والتطبيقات عالية الأداء في C23.

٢.٦ الاتحادات وتطبيقاتها

تُعدّ الاتحادات (Unions) بُنى بيانات منخفضة المستوى في لغة C تسمح لعدة أعضاء بمشاركة نفس موقع الذاكرة. وعلى عكس الهياكل، حيث يمتلك كل عضو مساحة تخزين مستقلة، فإن جميع أعضاء الاتحاد يتداخلون في نفس الذاكرة. وفي أي لحظة زمنية، يُعدّ عضو واحد فقط هو صاحب القيمة الصالحة.
تُستخدم الاتحادات بشكل أساسي من أجل كفاءة الذاكرة، وتمثيل البيانات المتغايرة المعلَّمة (Tagged Variants), ومعالجة البروتوكولات الثنائية، والتحكم الصريح في تمثيل الكائنات في الذاكرة.

١.٢.٦ ما هو الاتحاد؟

الاتحاد هو نوع بيانات مُعرّف من قبل المستخدم، تتشارك جميع أعضائه نفس عنوان الذاكرة. ويكون حجم الاتحاد مساويًا لحجم أكبر عضو فيه، مع مراعاة متطلبات المحاذاة.
لا يجوز الوصول إلى أكثر من عضو واحد بشكل ذي معنى في نفس الوقت.

٢.٢.٦ تعريف اتحاد

يُعرّف الاتحاد باستخدام الكلمة المفتاحية union متبوعة بقائمة الأعضاء.

```
union Data {
    int   i;
    float f;
    char  str[20];
};
```

تبدأ جميع الأعضاء من نفس عنوان الذاكرة.

٣.٢.٦ التصريح بمتغيرات من اتحاد

بعد التعريف، يمكن التصريح بمتغيرات الاتحاد كما هو الحال مع أي نوع آخر.

```
union Data data;
```

ويمكن التصريح بعدة متغيرات معًا:

```
union Data d1, d2;
```

٤.٢.٦ تهيئة الاتحادات

يمكن تهيئة الاتحاد بقيمة تخص أول عضو فيه.

```
union Data d = { 10 };
```

كما يدعم معيار C23 التهيئة المعيّنة، وهي المفضّلة بشدة.

```
union Data d = { .f = 3.14f };
```

٥.٢.٦ الوصول إلى أعضاء الاتحاد وتعديلهم

يتم الوصول إلى أعضاء الاتحاد باستخدام معامل النقطة.

```
d.i = 10;
printf("%d\n", d.i);
```

إسناد قيمة إلى عضو آخر يؤدي إلى الكتابة فوق نفس منطقة الذاكرة.

```
d.f = 3.14f;  /* overwrites previous integer */
```

لا يجوز الوصول إلا إلى العضو الذي كُتبت إليه القيمة مؤخرًا.

٦.٢.٦ نموذج الذاكرة والنوع الفعّال

عند كتابة قيمة في عضو من الاتحاد، يصبح هذا العضو هو العضو النشط (Active Member).
ويؤدي ذلك إلى تحديد النوع الفعّال (Effective Type) للكائن المخزَّن.
الوصول إلى عضو آخر بعد ذلك يُعد سلوكًا غير معرّف إلا في حالات محدودة جدًا ينص عليها معيار C.

٧.٢.٦ الاستخدامات الشائعة للاتحادات

كفاءة الذاكرة

تقلل الاتحادات استهلاك الذاكرة عندما تكون التمثيلات متبادلة الإقصاء.

```
union Value {
    int   i;
    float f;
};
```

لا يوجد إلا تمثيل واحد في أي لحظة.

الاتحادات المعلَّمة

غالبًا ما يُقرن الاتحاد بوسم (Tag) لتحديد العضو النشط.

```
struct Variant {
    enum { VT_INT, VT_FLOAT, VT_STRING } tag;
    union {
        int   i;
        float f;
        char  str[32];
    } value;
};
```

الاستخدام الصحيح يتطلب دائمًا فحص الوسم.

```
if (v.tag == VT_INT) {
    printf("%d\n", v.value.i);
}
```

هذا النمط أساسي في المفسّرات، والمحلّلات، وتصميم البروتوكولات.

البروتوكولات الثنائية وسجلات العتاد

تُستخدم الاتحادات لربط الذاكرة الخام بعرض مُنظّم.

```
union Register {
    uint32_t raw;
    struct {
        uint32_t enable : 1;
        uint32_t mode   : 3;
        uint32_t unused : 28;
    } bits;
};
```

وهذا شائع في الأنظمة المضمنة وبرمجة النواة.

٨.٢.٦ إعادة تفسير النوع: المسموح والممنوع

يُسمّى استخدام الاتحادات لإعادة تفسير الذاكرة **Type Punning**. وفي معيار C، تُعد معظم هذه الاستخدامات سلوكًا غير معرّف.

مثال على سلوك غير معرّف

```
union U {
    int   i;
    float f;
};

union U u;
u.f = 3.14f;
printf("%d\n", u.i);  /* undefined behavior */
```

هذا غير قابل للنقل وقد يفشل تحت التحسينات.

البديل الآمن: memcpy

الطريقة الوحيدة المحمولة لإعادة تفسير تمثيل كائن هي استخدام memcpy.

```
float f = 3.14f;
uint32_t bits;

memcpy(&bits, &f, sizeof bits);
```

٩.٢.٦ الأخطاء الشائعة

الوصول إلى عضو غير نشط

```
union Data d;
d.i = 42;
printf("%f\n", d.f);  /* undefined behavior */
```

افتراض قابلية نقل التخطيط

تخطيط الاتحاد والمحاذاة والحشو تعتمد على المنصة ولا يجوز افتراض ثباتها.

غياب الوسم

استخدام الاتحادات دون وسم يؤدي إلى كود هش وغير آمن.

١٠.٢.٦ أفضل الممارسات

- تتبّع العضو النشط دائمًا باستخدام وسم
- تجنّب إعادة تفسير النوع قدر الإمكان
- استخدم memcpy لإعادة تفسير البتات
- استخدم الاتحادات للتمثيل لا للتعددية الشكلية
- تعامل مع الاتحادات كأدوات منخفضة المستوى

١١.٢.٦ مثال عملي: اتحاد معلَّم

```c
#include <stdio.h>
#include <string.h>

struct Value {
    enum { INT, FLOAT, STRING } type;
    union {
        int   i;
        float f;
        char  s[32];
```

```
    } data;
};

int main(void) {
    struct Value v;

    v.type = INT;
    v.data.i = 10;

    if (v.type == INT) {
        printf("%d\n", v.data.i);
    }

    v.type = STRING;
    strcpy(v.data.s, "Hello");

    if (v.type == STRING) {
        printf("%s\n", v.data.s);
    }

    return 0;
}
```

١٢.٢.٦ الملخص

توفر الاتحادات تحكمًا صريحًا في تخطيط الذاكرة وتمثيل البيانات في لغة C. وهي أداة أساسية للكفاءة، والواجهات الثنائية، وبرمجة الأنظمة منخفضة المستوى. ويتطلب استخدامها الصحيح فهم العضو النشط، وقواعد النوع الفعّال، وتجنب السلوك غير المعرّف بشكل صارم. وعند استخدامها بانضباط، تُعد الاتحادات من أقوى أدوات C التعبيرية.

الفصل ٧: معالجة الملفات

١.٧ فتح الملفات وإغلاقها

تُعدّ معالجة الملفات إحدى القدرات الأساسية في برمجة الأنظمة، إذ تُمكّن البرامج من تخزين البيانات بشكل دائم، وتبادل المعلومات مع برامج أخرى، والتفاعل مع نظام التشغيل خارج عمر العملية نفسها.
في لغة C، تُنفَّذ معالجة الملفات من خلال مكتبة الإدخال والإخراج القياسية (<stdio.h>), والتي توفّر تجريدًا محمولاً ومُخزَّنًا مؤقتًا (Buffered) فوق واصفات الملفات الخاصة بنظام التشغيل. يركّز هذا القسم على فتح الملفات وإغلاقها بشكل صحيح في C23، مع شرح أنماط الفتح، وسلوك التخزين المؤقت، ومعالجة الأخطاء، وأفضل الممارسات.

١.١.٧ ما هي معالجة الملفات في C؟

تعتمد معالجة الملفات في C على كائن معتم (Opaque) يُدعى FILE، وهو يمثّل تدفّقًا مرتبطًا بملف أو جهاز. تتولى مكتبة C إدارة التخزين المؤقت، وترميز النصوص (في تدفقات النص)، والتفاعل مع بيئة الاستضافة.
تشمل العمليات الشائعة المتعلقة بالملفات:

- فتح ملف
- قراءة البيانات وكتابتها

- تفريغ المخازن المؤقتة
- إغلاق الملف وتحرير الموارد

٢.١.٧ فتح ملف باستخدام fopen

يتم فتح الملفات باستخدام الدالة fopen، والتي تربط اسم ملف بتدفّق (Stream) ونمط وصول محدد.

الصيغة

```
FILE *fopen(const char *filename, const char *mode);
```

- filename: مسار الملف
- mode: سلسلة نمط الوصول
- القيمة المعادة: مؤشر إلى FILE عند النجاح، أو NULL عند الفشل

أنماط فتح الملفات

تحدد سلسلة النمط كيفية فتح الملف وكيفية سلوك التدفّق.

النمط	المعنى
"r"	قراءة فقط؛ يجب أن يكون الملف موجوداً
"w"	كتابة فقط؛ يفرّغ الملف أو ينشئه
"a"	إلحاق؛ ينشئ الملف عند الحاجة
"r+"	قراءة وكتابة؛ يجب أن يكون الملف موجوداً
"w+"	قراءة وكتابة؛ يفرّغ الملف أو ينشئه
"a+"	قراءة وإلحاق
"b"	نمط ثنائي (مثل "rb")

وضع النص مقابل الوضع الثنائي

في الأنظمة المستضافة:

- قد يُجري وضع النص تحويلات على محارف نهاية السطر
- يعطّل الوضع الثنائي جميع هذه التحويلات

لضمان القابلية للنقل، يجب استخدام الوضع الثنائي صراحةً عند التعامل مع بيانات غير نصية.

مثال: فتح ملف

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("fopen");
        return 1;
    }

    printf("File opened successfully\n");

    fclose(file);
    return 0;
}
```

٣.١.٧ إغلاق ملف باستخدام fclose

يؤدي إغلاق الملف إلى تفريغ المخازن المؤقتة، وتحديث بيانات الملف الوصفية، وتحرير جميع الموارد المرتبطة بالتدفّق.

الصيغة

```
int fclose(FILE *stream);
```

- stream: مؤشر إلى تدفّق مفتوح
- القيمة المعادة: 0 عند النجاح، و EOF عند الفشل

دلالات مهمة

- يتم تفريغ المخرجات المخزّنة تلقائيًا
- يتم الإبلاغ عن أخطاء التفريغ عبر القيمة المعادة
- استخدام التدفّق بعد fclose سلوك غير معرّف

مثال: إغلاق ملف

```
#include <stdio.h>

int main(void) {
    FILE *file = fopen("example.txt", "w");
    if (!file) {
        perror("fopen");
        return 1;
    }

    if (fclose(file) == EOF) {
        perror("fclose");
        return 1;
    }
```

```
    return 0;
}
```

٤.١.٧ التخزين المؤقت والتفريغ

تُخزَّن تدفّقات C مؤقتًا بشكل افتراضي، وقد لا تُكتب البيانات مباشرةً إلى القرص.

- fclose يفرّغ تلقائيًا
- fflush يفرّغ يدويًا

```
fflush(file);
```

عدم تفريغ أو إغلاق الملف قد يؤدي إلى فقدان البيانات.

٥.١.٧ الأخطاء الشائعة

تجاهل NULL من fopen

```
FILE *file = fopen("missing.txt", "r");
if (!file) {
    perror("fopen");
}
```

نسيان إغلاق الملفات

تستهلك التدفّقات المفتوحة موارد النظام، وقد يؤدي عدم إغلاقها إلى:

- استنفاد واصفات الملفات
- فقدان البيانات بسبب عدم التفريغ

استخدام تدفّق مغلق

```
fclose(file);
fgetc(file);  // Undefined behavior
```

٦.١.٧ أفضل الممارسات

- تحقّق دائمًا من نتيجة fopen
- أغلق كل تدفّق مفتوح مرة واحدة فقط
- استخدم الوضع الثنائي للبيانات غير النصية
- تعامل مع أخطاء fclose
- نظّم الكود لضمان التنظيف في جميع مسارات الخروج

٧.١.٧ أمثلة عملية

قراءة ملف

```
#include <stdio.h>

int main(void) {
    FILE *file = fopen("example.txt", "r");
    if (!file) {
        perror("fopen");
        return 1;
    }

    int ch;
```

```
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);
    }

    fclose(file);
    return 0;
}
```

كتابة ملف

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("example.txt", "w");
    if (!file) {
        perror("fopen");
        return 1;
    }

    fputs("Hello, C23\n", file);
    fclose(file);
    return 0;
}
```

الوصول إلى ملف ثنائي

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

```c
int main(void) {
    FILE *file = fopen("data.bin", "wb");
    if (!file) {
        perror("fopen");
        return 1;
    }

    int value = 42;
    fwrite(&value, sizeof value, 1, file);

    fclose(file);
    return 0;
}
```

٨.١.٧ الملخص

يُعد فتح الملفات وإغلاقها بشكل صحيح أمرًا جوهريًا لبرامج موثوقة. في هذا القسم، استعرضنا كيفية عمل fopen و fclose، وتأثير أنماط الفتح، ودور التخزين المؤقت في صحة البرنامج. إتقان هذه الأساسيات شرطٌ أساسي قبل الانتقال إلى عمليات الملفات المتقدمة مثل الإدخال والإخراج المنسّق، والوصول العشوائي، وواجهات POSIX منخفضة المستوى.

٢.٧ قراءة الملفات وكتابتها

بعد فتح الملف بنجاح، يمكن للبرنامج قراءة البيانات منه أو كتابة البيانات إليه. توفّر مكتبة C القياسية واجهات إدخال وإخراج منخفضة المستوى موجّهة للبايتات، إضافةً إلى واجهات أعلى مستوى تعتمد على التنسيق. لكل فئة دلالاتها الخاصة، وخصائصها من حيث الأداء، وأنماط الفشل المحتملة. يقدّم هذا القسم عرضًا دقيقًا وعمليًا لعمليات قراءة وكتابة الملفات في C23، ويغطي الإدخال والإخراج الثنائي، وإدخال النصوص، والإدخال والإخراج المنسّق، وسلوك التخزين المؤقت، ومعالجة الأخطاء. بنهاية هذا القسم، ستفهم ليس فقط كيف تنفّذ عمليات الملفات، بل أيضًا متى تستخدم كل دالة بالشكل الصحيح.

١.٢.٧ القراءة من الملفات

تنقل عملية القراءة البيانات من تدفّق الإدخال إلى ذاكرة البرنامج. يعتمد اختيار الدالة المناسبة على ما إذا كانت البيانات ثنائية أو نصية، وعلى ما إذا كانت مهيكلة أو حرة الشكل.

الإدخال الثنائي باستخدام fread

تقوم الدالة fread بقراءة بايتات خام من التدفّق إلى مخزن ذاكرة، دون أي تفسير أو تحويل للبيانات.

الصيغة

```
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
```

الدلالات

- تحاول قراءة count عنصرًا، كل عنصر بحجم size بايت
- تعيد عدد العناصر التي تمت قراءتها فعليًا
- القراءة الجزئية لا تعني بالضرورة وجود خطأ

مثال

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("data.bin", "rb");
    if (!file) {
        perror("fopen");
        return 1;
    }

    int buffer[10];
    size_t n = fread(buffer, sizeof buffer[0], 10, file);

    printf("Objects read: %zu\n", n);

    fclose(file);
    return 0;
}
```

الإدخال المنسّق باستخدام fscanf

تقوم الدالة fscanf بقراءة نص من التدفّق وتحليله وفق سلسلة تنسيق محددة، مع إجراء تحويلات للأنواع.

الصيغة

```c
int fscanf(FILE *stream, const char *format, ...);
```

ملاحظات مهمة

- يجب أن يتطابق الإدخال تمامًا مع التنسيق
- المطابقات الجزئية تترك بيانات غير مقروءة في التدفّق
- معالجة الأخطاء معقّدة ومعرّضة للأخطاء

مثال

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("numbers.txt", "r");
    if (!file) {
        perror("fopen");
        return 1;
    }

    int value;
    if (fscanf(file, "%d", &value) == 1) {
        printf("Read value: %d\n", value);
    }

    fclose(file);
    return 0;
}
```

الإدخال السطري باستخدام fgets

تقوم الدالة fgets بقراءة سطر نصي واحد، بما في ذلك محرف نهاية السطر إن وُجد.

الصيغة

```
char *fgets(char *buffer, int size, FILE *stream);
```

مثال

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("example.txt", "r");
    if (!file) {
        perror("fopen");
        return 1;
    }

    char line[128];
    if (fgets(line, sizeof line, file)) {
        printf("Line: %s", line);
    }

    fclose(file);
    return 0;
}
```

٢.٣.٧ الكتابة إلى الملفات

تنقل عملية الكتابة البيانات من ذاكرة البرنامج إلى تدفّق الإخراج. وكما هو الحال في الإدخال، تُستخدم دوال مختلفة للبيانات الثنائية والنصية.

الإخراج الثنائي باستخدام fwrite

تكتب الدالة fwrite بايتات خام إلى التدفّق دون أي تفسير.

الصيغة

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);
```

مثال

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("data.bin", "wb");
    if (!file) {
        perror("fopen");
        return 1;
    }

    int values[] = {1, 2, 3, 4, 5};
    size_t n = fwrite(values, sizeof values[0], 5, file);

    printf("Objects written: %zu\n", n);

    fclose(file);
    return 0;
}
```

الإخراج المنسّق باستخدام fprintf

تكتب الدالة fprintf نصًا منسّقًا إلى تدفّق محدد.

الصيغة

```
int fprintf(FILE *stream, const char *format, ...);
```

مثال

```
#include <stdio.h>

int main(void) {
    FILE *file = fopen("log.txt", "w");
    if (!file) {
        perror("fopen");
        return 1;
    }

    fprintf(file, "Value = %d\n", 42);

    fclose(file);
    return 0;
}
```

إخراج السلاسل النصية باستخدام fputs

تكتب الدالة fputs سلسلة نصية منتهية بمحرف الإنهاء إلى التدفّق.

```
int fputs(const char *str, FILE *stream);
```

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

```c
int main(void) {
    FILE *file = fopen("example.txt", "w");
    if (!file) {
        perror("fopen");
        return 1;
    }

    fputs("Hello, C23\n", file);

    fclose(file);
    return 0;
}
```

٣.٢.٧ موضع الملف والوصول التسلسلي

يحتفظ كل تدفّق بمؤشّر موضع ملف يحدّد مكان عملية القراءة أو الكتابة التالية.

- القراءة تُقدّم الموضع
- الكتابة تُقدّم الموضع
- يمكن تغيير الموضع صراحةً

```c
fseek(file, 0, SEEK_SET);
```

إدارة الموضع بشكل خاطئ من أكثر مصادر الأخطاء الخفيّة شيوعًا.

٤.٢.٧ الأخطاء الشائعة

افتراض اكتمال القراءة أو الكتابة

لا تضمن fread ولا fwrite معالجة جميع العناصر المطلوبة.

```
if (fread(buf, size, count, file) != count) {
    if (ferror(file)) {
        perror("read error");
    }
}
```

تجاوز حدود المخازن

يجب دائمًا تقييد عمليات الإدخال بحجم المخزن.

```
fgets(buffer, sizeof buffer, file);
```

٥.٢.٧ أفضل الممارسات

- استخدم fwrite/fread للبيانات الثنائية
- فضّل fgets على fscanf لإدخال النص
- تحقّق دائمًا من القيم المعادة
- اعتبر الإدخال/الإخراج الجزئي سلوكًا طبيعيًا
- أغلق الملفات بسرعة ولمرة واحدة فقط

٦.٢.٧ مثال عملي: دورة ثنائية كاملة

```c
#include <stdio.h>

int main(void) {
    int out[] = {1, 2, 3, 4, 5};

    FILE *file = fopen("example.bin", "wb");
    fwrite(out, sizeof out[0], 5, file);
    fclose(file);

    int in[5];
    file = fopen("example.bin", "rb");
    fread(in, sizeof in[0], 5, file);
    fclose(file);

    for (int i = 0; i < 5; ++i) {
        printf("%d ", in[i]);
    }
    printf("\n");

    return 0;
}
```

٧.٢.٧ الملخص

تتطلّب قراءة وكتابة الملفات في C فهم واجهات البرمجة وفهم دلالات التدفّقات على حدّ سواء. في هذا القسم، استعرضنا الإدخال والإخراج الثنائي والنصي، والعمليات المنسّقة وغير المنسّقة، وسلوك التخزين المؤقت، وإدارة موضع الملف، ومعالجة الأخطاء بشكل متين. تشكّل

هذه الأساسيات قاعدة الانطلاق لموضوعات أكثر تقدّمًا مثل الوصول العشوائي، واستعادة الأخطاء، وأنظمة الإدخال والإخراج عالية الأداء.

٣.٧ معالجة الأخطاء في عمليات الملفات

تُعدّ معالجة الأخطاء متطلبًا أساسيًا لإنشاء برامج موثوقة عند التعامل مع الملفات. وعلى عكس الشيفرة الحسابية البحتة، تعتمد عمليات الملفات على موارد خارجية مثل نظام الملفات، وأجهزة التخزين، وخدمات نظام التشغيل. وقد تفشل هذه الموارد أو تصبح غير متاحة في أي وقت.
في C23، تُعدّ معالجة أخطاء الإدخال والإخراج عملية صريحة ويدوية. يقع على عاتق المبرمج اكتشاف حالات الفشل، وتفسير أسبابها، وضمان بقاء حالة البرنامج متّسقة. يقدّم هذا القسم منهجًا صارمًا وعمليًا لمعالجة الأخطاء في عمليات الملفات، مستندًا إلى الدلالات الرسمية لمكتبة C القياسية.

١.٣.٧ لماذا تفشل عمليات الملفات

قد تفشل عمليات الملفات لأسباب خارجة عن سيطرة البرنامج. من الأسباب الشائعة:

- عدم وجود الملف أو المسار
- نقص صلاحيات الوصول
- امتلاء القرص أو تجاوز الحصص
- أعطال العتاد أو نظام الإدخال والإخراج
- استخدام أوضاع ملفات غير صحيحة
- محاولة القراءة بعد نهاية الملف

يجب على أي برنامج صحيح أن يفترض إمكانية فشل أي عملية ملف، وأن يكون مستعدًا لمعالجة هذا الفشل بشكل صريح.

٢.٣.٧ آليات الكشف الأساسية عن الأخطاء

توفر لغة C ثلاث آليات متكاملة لاكتشاف أخطاء عمليات الملفات:

١. القيم المعادة من الدوال

٢. مؤشرات حالة التدفّق

٣. متغير الخطأ العام errno

تخدم كل آلية غرضًا مختلفًا، ويجب استخدامها بالطريقة الصحيحة.

٣.٣.٧ القيم المعادة: خط الدفاع الأول

تشير معظم دوال التعامل مع الملفات إلى الفشل من خلال قيمها المعادة.

- تعيد fopen القيمة NULL عند الفشل
- تعيد fclose القيمة EOF عند الفشل
- تعيد fread وfwrite عدد العناصر المعالجة
- تعيد fgets القيمة NULL عند الفشل أو نهاية الملف

يجب دائمًا فحص القيم المعادة فورًا.

```
FILE *file = fopen("example.txt", "r");
if (!file) {
    /* fopen failed */
}
```

يُعدّ تجاهل القيم المعادة من أكثر أسباب السلوك غير المعرّف شيوعًا في برامج C.

٤.٣.٧ فهم الإدخال والإخراج الجزئي

قد تقوم الدوال مثل fread وfwrite بمعالجة عدد أقل من العناصر المطلوبة دون الإشارة إلى وجود خطأ.

```
size_t n = fread(buf, sizeof *buf, count, file);
if (n < count) {
    if (ferror(file)) {
        /* an error occurred */
    }
    /* otherwise: end-of-file or short read */
}
```

لا يُعدّ الإدخال أو الإخراج الجزئي خطأً بحد ذاته، ويجب تفسيره ضمن السياق الصحيح.

٥.٣.٧ حالة التدفّق: feof وferror

يحتفظ كل تدفّق FILE بمؤشرات داخلية تعبّر عن حالة الخطأ وحالة نهاية الملف.

- feof(stream) لاختبار الوصول إلى نهاية الملف
- ferror(stream) لاختبار حدوث خطأ إدخال/إخراج

لا تكون هذه المؤشرات ذات معنى إلا بعد فشل عملية ما.

```
if (fgets(buf, sizeof buf, file) == NULL) {
    if (feof(file)) {
        /* normal end-of-file */
    } else if (ferror(file)) {
        /* I/O error */
    }
}
```

يُعدّ استدعاء feof قبل محاولة القراءة خطأً منطقيًا دائمًا.

٦.٣.٧ المتغير errno

يوفّر المتغير errno معلومات تشخيصية إضافية حول بعض حالات الفشل في الدوال القياسية واستدعاءات النظام.

- معرّف في <errno.h>
- صالح فقط مباشرة بعد الفشل
- قد يتغيّر بفعل استدعاءات مكتبية غير ذات صلة

من رموز الأخطاء الشائعة:

- ENOENT — الملف أو المسار غير موجود
- EACCES — رفض الصلاحيات
- ENOSPC — لا توجد مساحة كافية على الجهاز

```c
#include <errno.h>

FILE *file = fopen("example.txt", "r");
if (!file) {
    if (errno == ENOENT) {
        puts("File not found");
    }
}
```

يجب عدم فحص errno إلا عندما تشير الدالة صراحةً إلى الفشل.

٧.٣.٧ الإبلاغ عن الأخطاء: perror وstrerror

perror

تقوم الدالة perror بطباعة رسالة خطأ مقروءة للبشر استناداً إلى القيمة الحالية لـ errno.

```
if (!file) {
    perror("fopen");
}
```

تُكتب الرسالة إلى دفق الخطأ القياسي.

strerror

تحوّل الدالة strerror رمز الخطأ إلى سلسلة وصفية.

```
#include <string.h>

fprintf(stderr, "Error: %s\n", strerror(errno));
```

تُستخدم هذه الدالة عند بناء رسائل خطأ مهيكلة أو مخصّصة.

٨.٣.٧ تنظيف الموارد عند الفشل

يجب أن تضمن جميع مسارات الفشل تنظيف الموارد بشكل صحيح.

- يجب إغلاق الملفات المفتوحة
- يجب تحرير الذاكرة المحجوزة
- يجب التعامل مع عمليات الكتابة الجزئية بشكل متّسق

من الأنماط الشائعة تجميع منطق التنظيف في موضع واحد.

۹.۳.۷ مثال شامل

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

int main(void) {
    FILE *file = fopen("example.txt", "r");
    if (!file) {
        perror("fopen");
        return EXIT_FAILURE;
    }

    char buffer[128];

    if (!fgets(buffer, sizeof buffer, file)) {
        if (feof(file)) {
            puts("End of file reached");
        } else {
            perror("read error");
        }
        fclose(file);
        return EXIT_FAILURE;
    }

    printf("Read: %s", buffer);

    if (fclose(file) == EOF) {
        perror("fclose");
        return EXIT_FAILURE;
```

```
    }

    return EXIT_SUCCESS;
}
```

يوضّح هذا المثال الانضباط في فحص القيم المعادة، والتفسير الصحيح لحالة التدفّق، والتنظيف السليم في جميع مسارات التنفيذ.

١٠.٣.٧ ملخص أفضل الممارسات

- افحص دائمًا القيم المعادة من الدوال
- اعتبر الإدخال والإخراج الجزئي سلوكًا طبيعيًا
- استخدم feof وferror فقط بعد الفشل
- افحص errno فورًا وبشكل محدود
- أغلق الملفات مرة واحدة فقط
- اجعل مسارات معالجة الأخطاء واضحة وبسيطة

١١.٣.٧ الخلاصة

معالجة الأخطاء في عمليات الملفات ليست خيارًا، بل سمة أساسية للبرمجة الاحترافية بلغة C. إتقان القيم المعادة، وحالة التدفّقات، وآليات الإبلاغ عن الأخطاء، يمكّنك من بناء برامج تفشل بأناقة، وتحافظ على سلامة البيانات، وتبقى قابلة للصيانة في ظروف التشغيل الواقعية.

٤.٧ العمل مع الملفات الثنائية

تُعدّ الملفات الثنائية ركيزة أساسية في البرمجة منخفضة المستوى وبرمجة الأنظمة. وعلى عكس الملفات النصية التي تمثّل البيانات على شكل محارف قابلة للقراءة البشرية، تخزّن الملفات الثنائية البيانات كسلاسل بايت خام تعكس تمثيلها في الذاكرة بشكل مباشر.
يتيح هذا الأسلوب قراءة وكتابة البيانات بكفاءة عالية وبسلوك حتمي، لكنه في المقابل يفرض مسؤولية أكبر على المبرمج في ما يخص إدارة التخطيط، وقابلية النقل، وصحة التنفيذ.
يقدّم هذا القسم معالجة دقيقة ومتوافقة مع المعيار للتعامل مع الملفات الثنائية في C23، مع التركيز على الصحة، والأداء، وقابلية الصيانة على المدى الطويل.

١.٤.٧ ما الذي يعرّف الملف الثنائي

الملف الثنائي هو تسلسل من البايتات من دون أي ترميز نصي أو بنية أسطر ضمنية. لا يفرض معيار C أي تفسير على محتوى الملف الثنائي؛ ويعود تفسير البيانات كاملاً إلى البرنامج.
تُستخدم الملفات الثنائية بشكل شائع في:

- التخزين الدائم للبيانات المهيكلة
- التسلسل عالي الأداء
- لقطات الذاكرة والتفريغ
- صيغ الملفات ذات التخطيط الثابت

وبما أن الملفات الثنائية تكشف التمثيل الخام للبيانات، فإن صحة التنفيذ تعتمد على اتساق التخطيط، وأحجام الأنواع، وتناظر عمليات القراءة والكتابة.

٢.٤.٧ فتح الملفات الثنائية

تُفتح الملفات الثنائية باستخدام `fopen` مع تضمين المحدد `b` في سلسلة الوضع.

- "rb" — قراءة ثنائية
- "wb" — كتابة ثنائية (إنشاء أو اقتطاع)
- "ab" — إلحاق ثنائي
- "r+b" — قراءة/كتابة ثنائية
- "w+b" — قراءة/كتابة مع اقتطاع
- "a+b" — قراءة/إلحاق ثنائي

```
FILE *file = fopen("data.bin", "rb");
if (!file) {
    perror("fopen");
    return;
}
```

يُعدّ الوضع الثنائي ضروريًا على الأنظمة التي تقوم بوضع النص بإجراء تحويلات للأسطر أو الترميز.

٣.٤.٧ قراءة البيانات الثنائية باستخدام fread

تتم عملية الإدخال الثنائي باستخدام fread، التي تنقل بايتات خام من الملف إلى الذاكرة.

```
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
```

تحاول الدالة قراءة count من العناصر، حيث يبلغ حجم كل عنصر size بايت. تشير القيمة المعادة إلى عدد العناصر التي قُرئت بنجاح.

```
int buffer[10];
size_t n = fread(buffer, sizeof buffer[0], 10, file);
```

```c
if (n < 10) {
    if (feof(file)) {
        /* end-of-file reached */
    } else if (ferror(file)) {
        perror("fread");
    }
}
```

لا يُعدّ الإدخال الجزئي خطأً بحد ذاته، ويجب دائمًا تفسيره ضمن السياق المناسب.

٤.٤.٧ كتابة البيانات الثنائية باستخدام fwrite

تتم عملية الإخراج الثنائي باستخدام fwrite، التي تنقل بايتات خام من الذاكرة إلى الملف.

```c
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);
```

```c
int data[10] = {0,1,2,3,4,5,6,7,8,9};
size_t n = fwrite(data, sizeof data[0], 10, file);

if (n < 10) {
    perror("fwrite");
}
```

يسمح معيار C بالكتابة الجزئية، ويجب التعامل معها بشكل صريح.

٥.٤.٧ الملفات الثنائية والبُنى

تُستخدم الملفات الثنائية غالبًا لتخزين البُنى مباشرة. ويكون هذا الأسلوب صحيحًا فقط عندما يُستهلك الملف من نفس البرنامج أو من برامج تفترض تخطيط بيانات متطابقًا تمامًا.

```c
typedef struct {
    int id;
    char name[50];
    float salary;
} Employee;
```

```c
Employee e = {1, "John Doe", 75000.0f};
fwrite(&e, sizeof e, 1, file);
```

```c
Employee e;
fread(&e, sizeof e, 1, file);
```

قيود مهمة

الكتابة المباشرة للبُنى غير قابلة للنقل عبر:

- معماريات مختلفة
- مترجمات أو ABI مختلفة
- قواعد محاذاة أو حشو مختلفة
- اختلاف ترتيب البايتات

للتخزين طويل الأمد أو الصيغ العابرة للمنصات، يجب استخدام تسلسل صريح ومضبوط.

٦.٤.٧ الوصول العشوائي في الملفات الثنائية

تدعم الملفات الثنائية الوصول العشوائي بكفاءة عالية باستخدام fseek وftell.

```
int fseek(FILE *stream, long offset, int origin);
long ftell(FILE *stream);
```

```
fseek(file, 4L * sizeof(Employee), SEEK_SET);

Employee e;
fread(&e, sizeof e, 1, file);
```

يفترض الوصول العشوائي وجود سجلات ذات حجم ثابت وتخطيط مستقر.

٧.٤.٧ معالجة الأخطاء في الإدخال والإخراج الثنائي

يجب على جميع عمليات الملفات الثنائية فحص ما يلي:

- القيم المعادة من الدوال
- حالة التدفّق باستخدام feof وferror
- الأخطاء المبلّغ عنها عبر errno

```
FILE *file = fopen("data.bin", "rb");
if (!file) {
    perror("fopen");
    return;
}

int buffer[10];
```

```
size_t n = fread(buffer, sizeof buffer[0], 10, file);

if (n < 10) {
    if (feof(file)) {
        puts("End of file");
    } else {
        perror("read error");
    }
}

if (fclose(file) == EOF) {
    perror("fclose");
}
```

٨.٤.٧ أفضل الممارسات للملفات الثنائية

١. استخدم أنواعًا ثابتة العرض عندما يكون التخطيط مهمًا

٢. وثّق صيغ الملفات الثنائية بدقة

٣. لا تفترض أبدًا المحاذاة أو الحشو

٤. تحقّق دائمًا من عدد العناصر المقروءة أو المكتوبة

٥. افصل منطق التسلسل عن منطق العمل

٩.٤.٧ مثال شامل

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

```c
typedef struct {
    int id;
    char name[50];
    float salary;
} Employee;

void write_employees(const char *path) {
    FILE *f = fopen(path, "wb");
    if (!f) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    Employee list[3] = {
        {1, "Alice", 80000.0f},
        {2, "Bob",   85000.0f},
        {3, "Carol", 90000.0f}
    };

    if (fwrite(list, sizeof(Employee), 3, f) != 3) {
        perror("fwrite");
    }

    fclose(f);
}

void read_employee(const char *path, size_t index) {
    FILE *f = fopen(path, "rb");
```

```
    if (!f) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    fseek(f, (long)(index * sizeof(Employee)), SEEK_SET);

    Employee e;
    if (fread(&e, sizeof e, 1, f) == 1) {
        printf("%d %s %.2f\n", e.id, e.name, e.salary);
    }

    fclose(f);
}

int main(void) {
    write_employees("employees.bin");
    read_employee("employees.bin", 1);
    return 0;
}
```

١٠.٤.٧ الخلاصة

يوفّر التعامل مع الملفات الثنائية أعلى درجات الكفاءة والتحكّم، لكنه يتطلب دقة وانضباطًا عاليين. وعند استخدامه بالشكل الصحيح، يمكّن من تخزين واسترجاع البيانات المهيكلة بأداء عالٍ. أما عند إساءة استخدامه، فيؤدي إلى مشكلات دقيقة في القابلية للنقل وصحة التنفيذ.
يُعدّ إتقان الملفات الثنائية إحدى السمات الفارقة لمبرمجي C وبرمجة الأنظمة المحترفين.

الفصل ٨: البرمجة منخفضة المستوى

١.٨ فهم البرمجة منخفضة المستوى

تمثّل البرمجة منخفضة المستوى الأساس الذي تُبنى عليه أنظمة التشغيل، وبيئات التشغيل، والبرمجيات الحرجة من حيث الأداء. فهي تكشف الآليات التي يتفاعل بها البرنامج مع العتاد، والذاكرة، ونواة نظام التشغيل، بدلاً من إخفائها خلف تجريدات عالية المستوى.
وعلى عكس تطوير التطبيقات، تركّز البرمجة منخفضة المستوى على كيفية تنفيذ البرامج، وكيف تُرتَّب الذاكرة ويُتعامل معها، وكيف تُكتسب الموارد وتُحرَّر، وكيف ينتقل التحكم عبر الحد الفاصل بين نمط المستخدم ونمط النواة. إن إتقان هذه المفاهيم يتيح الاستدلال الدقيق حول الأداء، والصحة، وسلوك النظام.

١.١.٨ ما هي البرمجة منخفضة المستوى؟

تشير البرمجة منخفضة المستوى إلى كتابة برمجيات تعكس بشكل مباشر نموذج التنفيذ ونموذج الذاكرة الخاص بالآلة. وهي تقلّل من التجريد وتضع مسؤولية الصحة، وإدارة الموارد، والسلامة على عاتق المبرمج نفسه.
تشمل لغات البرمجة منخفضة المستوى عادةً لغة C ولغة التجميع، حيث توفّران وصولاً مباشرًا إلى تعليمات المعالج، وعناوين الذاكرة، وواجهات نظام التشغيل.
ومن خصائصها الأساسية:

- التفاعل المباشر مع العتاد: يمكن للبرنامج قراءة وكتابة عناوين الذاكرة، والوصول إلى السجلات، والتعامل مع الأجهزة.
- إدارة الموارد الصريحة: تُدار الذاكرة، وواصفات الملفات، والعمليات، وآليات التزامن يدويًا.
- ضمانات تشغيل محدودة: لا توجد حماية تلقائية من أخطاء الذاكرة أو الأخطاء المنطقية.
- تكلفة تنفيذ متوقعة: السلوك الأدائي ظاهر وقابل للتنبؤ إلى حد كبير.

تستبدل البرمجة منخفضة المستوى الراحة والسلامة بالتحكم، والشفافية، والكفاءة.

٢.١.٨ البرمجة منخفضة المستوى مقابل عالية المستوى

يعود الفرق الجوهري بين البرمجة منخفضة المستوى والعالية المستوى إلى مقدار التحكم في بيئة التنفيذ.

الجانب	منخفضة المستوى	عالية المستوى
التجريد	ضئيل، موجه للعتاد	مرتفع، موجه للمجال التطبيقي
التحكم	تحكم صريح في الذاكرة والتنفيذ	مُدار بواسطة بيئات تشغيل
الأداء	قابل للتنبؤ والضبط بدقة	مجرّد وغالبًا غير شفاف
سهولة الاستخدام	تتطلب معرفة عميقة بالنظام	تطوير أسرع وضمانات أعلى
قابلية النقل	غالبًا مرتبطة بالمعمارية	متعددة المنصات
أنماط الفشل	سلوك غير معرّف ممكن	أخطاء مُلتقطة ومبلّغ عنها

لا يُعد أيّ من النموذجين أفضل مطلقًا؛ بل يخدم كلٌ منهما متطلبات مختلفة.

٣.١.٨ لماذا تظل البرمجة منخفضة المستوى مهمة؟

رغم انتشار اللغات عالية المستوى، لا تزال البرمجة منخفضة المستوى ضرورية في الأنظمة الحديثة:

- برمجيات النظام: النواة، مشغلات الأجهزة، البرمجيات الثابتة، محمّلات الإقلاع، وأنظمة

المحاكاة.

- الأنظمة الحرجة للأداء: محركات الألعاب، أنظمة الزمن الحقيقي، والأحمال العددية.
- البيئات محدودة الموارد: الأنظمة المضمّنة التي تفتقر إلى ذاكرة افتراضية أو بيئات تشغيل ضخمة.
- الصحة والتشخيص: الفهم منخفض المستوى ضروري للتصحيح، والتحسين، والتحليل الأمني.

فجميع التجريدات عالية المستوى تعتمد في النهاية على آليات منخفضة المستوى.

٤.١.٨ المفاهيم الأساسية في البرمجة منخفضة المستوى

إدارة الذاكرة

في البرمجة منخفضة المستوى، تكون إدارة الذاكرة صريحة بالكامل. يتحكم المبرمج في توقيت الحجز وتوقيت التحرير.

```
int *ptr = malloc(sizeof *ptr * 10);
if (!ptr) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

/* use memory */

free(ptr);
```

تؤدي الإدارة الخاطئة إلى تسربات، أو فساد ذاكرة، أو سلوك غير معرّف.

المؤشرات والعنونة

تكشف المؤشرات عناوين الذاكرة الخام وتسمح بالتحكم المباشر في البيانات.

```
int x = 10;
int *p = &x;
*p = 20;
```

تُعد المؤشرات الأساس لهياكل البيانات الديناميكية والوصول إلى الذاكرة والعتاد.

العمليات على مستوى البت

تتعامل العمليات البتّية مع البتات الفردية وتُستخدم بكثافة في التحكم بالعتاد والأعلام وتمثيل البيانات المضغوط.

```
unsigned int flags = 0x0F;
flags &= ~0x02;
flags |=  0x10;
```

وتقابل هذه العمليات تعليمات المعالج مباشرةً.

التجميع المضمّن

يسمح التجميع المضمّن بإدراج تعليمات خاصة بالمعالج عندما لا تكفي تجريدات اللغة.

```
int result;
__asm__ volatile (
    ".intel_syntax noprefix\n"
    "mov eax, 10\n"
    "add eax, 20\n"
    "mov %0, eax\n"
```

```
    ".att_syntax prefix\n"
    : "=r"(result)
    :
    : "eax"
);
```

يقلّل هذا الأسلوب من قابلية النقل ويُستخدم عند الضرورة فقط.

نداءات النظام

تمثّل نداءات النظام الحد الفاصل بين نمط المستخدم ونمط النواة.

```
int fd = open("example.txt", O_RDONLY);
if (fd == -1) {
    perror("open");
}
close(fd);
```

وهي تفرض انتقالاً مضبوطًا في الامتيازات واعتماداً على المنصة.

٥.١.٨ أدوات تطوير البرمجة منخفضة المستوى

يعتمد التطوير الفعّال على أدوات متخصصة:

- المترجمات لتوليد شيفرة محسّنة
- المصححات لفحص السجلات والذاكرة
- محللات الأداء لدراسة الكاش والزمن
- مفككات الشيفرة لتحليل التعليمات

إتقان الأدوات لا يقل أهمية عن إتقان اللغة.

٦.١.٨ أفضل الممارسات

١. افهم المعمارية وABI المستهدف.

٢. تحقّق من افتراضات التخطيط والمحاذاة.

٣. قِس قبل التحسين.

٤. فعّل تحذيرات المترجم واحترمها.

٥. اختبر الحالات الحدّية بقسوة.

٧.١.٨ الخلاصة

تكشف البرمجة منخفضة المستوى الواقع الحقيقي لتنفيذ البرمجيات وتفاعلها مع الذاكرة ونظام التشغيل. وإتقانها لا يعني كتابة شيفرة سريعة فحسب، بل يعني فهم قواعد الآلة نفسها. وهذا الفهم يمكّن من تصميم واعٍ، وبرمجيات موثوقة، وقدرة دقيقة على تحليل الأداء والصحة والأمن.

٢.٨ الوصول إلى العتاد باستخدام المؤشرات

يُعد الوصول المباشر إلى العتاد إحدى السمات الجوهرية للبرمجة منخفضة المستوى. فالعديد من الأجهزة تُعرّف سجلات التحكم والحالة ضمن فضاء العناوين، مما يسمح بالتحكم بها عبر القراءة والكتابة إلى مواقع ذاكرة محددة.
في هذه الأنظمة، لا تُعد المؤشرات مجرد ميزة لغوية، بل هي الواجهة الفعلية للتحكم بالأجهزة الفيزيائية.

١.٢.٨ الإدخال والإخراج المعنون بالذاكرة

يتم في أسلوب Memory-Mapped I/O ربط سجلات الأجهزة بعناوين ذاكرة.

```
#define GPIO_DATA (*(volatile unsigned int *)0x1000)

void toggle_led(void) {
    GPIO_DATA ^= 0x01;
}
```

تمثّل القيمة 0x1000 عنوان سجل عتادي، وتؤثر القراءة أو الكتابة إليه مباشرةً على الجهاز.

٢.٢.٨ دور volatile

يُعلم المحدد volatile المترجم بأن قيمة الكائن قد تتغير خارج تدفق البرنامج.

```
volatile unsigned int *timer =
    (volatile unsigned int *)0x2000;

while (*timer == 0) {
    /* busy wait */
}
```

لا يوفّر volatile الذرّية ولا التزامن ولا ترتيب الذاكرة؛ بل يضمن فقط عدم حذف عمليات الوصول.

٣.٢.٨ الحساب على المؤشرات في الوصول للعتاد

تُعرض الأجهزة عادةً كمجموعة سجلات بإزاحات ثابتة عن عنوان أساسي.

```
#define UART_BASE 0x4000

volatile unsigned int *uart_data =
    (volatile unsigned int *)(UART_BASE + 0x00);
volatile unsigned int *uart_status =
    (volatile unsigned int *)(UART_BASE + 0x04);
```

يعكس هذا الأسلوب تخطيط السجلات كما هو موصوف في دليل العتاد.

٤.٢.٨ مثال عملي: التحكم في GPIO

```
#define GPIO_DATA (*(volatile unsigned int *)0x1000)
#define GPIO_DIR  (*(volatile unsigned int *)0x1004)

void gpio_init(void) {
    GPIO_DIR |= 0x01;
}

void led_on(void)      { GPIO_DATA |=  0x01; }
void led_off(void)     { GPIO_DATA &= ~0x01; }
void led_toggle(void)  { GPIO_DATA ^=  0x01; }
```

٥.٢.٨ قراءة المدخلات من العتاد

```
int button_pressed(void) {
    return (GPIO_DATA & 0x02) != 0;
```

```
}
```

٦.٢.٨ الخلاصة

يكشف الوصول إلى العتاد باستخدام المؤشرات نموذج التنفيذ الحقيقي للنظام. وتشكّل الإدخال المعنون بالذاكرة، والحساب على المؤشرات، وvolatile الواجهة الأساسية بين البرمجيات والأجهزة الفيزيائية.

وعند استخدامها بانضباط، تتيح هذه الآليات تحكمًا دقيقًا وحتميًا وعالي الكفاءة. أما إساءة استخدامها، فتؤدي إلى أخطاء دقيقة، وسلوك غير معرّف، أو عدم استقرار النظام. ولذلك يتطلب إتقانها انضباطًا، وفهمًا عميقًا لما تضمنه اللغة وما لا تضمنه.

٣.٨ استخدام التجميع المضمَّن في C

يسمح التجميع المضمَّن (Inline Assembly) بإدراج تعليمات تجميع خاصة بالمعالج مباشرةً داخل شيفرة C. ويوفّر هذا الأسلوب مخرجًا مضبوطًا من الآلة التجريدية للغة C عندما يكون الوصول المباشر إلى تعليمات المعالج، أو السجلات، أو ترتيب التنفيذ أمرًا ضروريًا.
لا يُعد التجميع المضمَّن جزءًا من معيار ISO C23، بل هو امتداد خاص بالمترجم تدعمه سلاسل أدوات مثل GCC وClang. ولهذا السبب، فإن استخدامه يُدخل تبعيات خاصة بالمعمارية وبالمترجم، ويجب أن يكون مقصودًا، ومحليًا، ومُبرَّرًا بعناية.

١.٣.٨ لماذا يوجد التجميع المضمَّن؟

يوجد التجميع المضمَّن لمعالجة قيود لغة C المحمولة. ومن أبرز المبررات لاستخدامه:

- الوصول إلى تعليمات خاصة: تعليمات لا مكافئ لها على مستوى C مثل rdtsc، cpuid، أو تعليمات الكاش وTLB.
- التفاعل الدقيق مع العتاد: التحكم الصريح في السجلات، أو الأعلام، أو الأجهزة المعنونة بالذاكرة.
- الترتيب والحواجز: فرض ترتيب صارم للتعليمات أو ضمانات رؤية الذاكرة.
- البيئات العارية والنواة: أنظمة لا تحتوي على بيئة تشغيل كاملة أو مكتبة قياسية.

لا يجب إدخال التجميع المضمَّن بدافع الحدس أو افتراض تحسين الأداء. بل يجب إثبات ضرورته وصحته عبر القياس والتحليل.

٢.٣.٨ صيغة التجميع المضمَّن (GCC / Clang)

تستخدم صيغة GCC Extended Inline Assembly البنية التالية:

```
asm volatile (
    "assembly instructions"
    : output_operands
    : input_operands
    : clobbered_registers
);
```

- asm تبدأ كتلة التجميع المضمَّن.
- volatile تمنع حذف أو إعادة ترتيب الكتلة.
- سلسلة التجميع تحتوي تعليمات التجميع الخام.
- المعاملات الخارجة تربط السجلات أو الذاكرة بمتغيرات C.
- المعاملات الداخلة تمرر القيم إلى التجميع.
- المخرِّبات تصف السجلات أو الأعلام أو الذاكرة التي يتم تعديلها.

أي إغفال في توصيف المعاملات أو الآثار الجانبية يؤدي إلى سلوك غير معرّف على الحد الفاصل بين اللغة والمترجم.

٣.٣.٨ مثال أساسي: جمع عددين صحيحين

```
#include <stdio.h>

int main(void) {
    int a = 5, b = 10, result;

    asm volatile (
        ".intel_syntax noprefix\n"
```

```
        "add %0, %2\n"
        ".att_syntax prefix\n"
        : "=r"(result)
        : "0"(a), "r"(b)
        : "cc"
    );

    printf("Result: %d\n", result);
    return 0;
}
```

الشرح:

- القيد "0" يُجبر المترجم على إعادة استخدام سجل الخرج كمعامل دخل.
- المخرّب cc يصرّح بتعديل أعلام المعالج.
- تخصيص السجلات متروك بالكامل للمترجم.

٤.٣.٨ الوصول إلى سجلات المعالج: قراءة عدّاد الزمن

```
#include <stdint.h>

static inline uint64_t read_tsc(void) {
    uint32_t lo, hi;
    asm volatile (
        ".intel_syntax noprefix\n"
        "rdtsc\n"
        ".att_syntax prefix\n"
        : "=a"(lo), "=d"(hi)
        :
```

```
        : "memory"
    );
    return ((uint64_t)hi << 32) | lo;
}
```

ملاحظات:

- القيود تربط القيم بسجلات معمارية محددة.
- المخرّب memory يمنع إعادة الترتيب حول التعليمة.
- التنفيذ خاص بمعمارية x86.

٥.٣.٨ المعاملات والقيود

تحدد القيود كيفية تمرير القيم بين C والتجميع:

- r — سجل عام
- m — معامل ذاكرة
- i — ثابت فوري
- a, d — سجلات معمارية ثابتة

الاختيار الصحيح للقيود أساسي للصحة وتخصيص السجلات وترميز التعليمات.

٦.٣.٨ المخرِّبات والآثار الجانبية

يجب التصريح بأي سجل، أو علم، أو حالة ذاكرة يتم تعديلها.

```
asm volatile (
    ".intel_syntax noprefix\n"
    "mov eax, %1\n"
    "add eax, %2\n"
    "mov %0, eax\n"
    ".att_syntax prefix\n"
    : "=r"(result)
    : "r"(a), "r"(b)
    : "eax", "cc"
);
```

عدم التصريح عن الآثار الجانبية يسمح للمترجم بافتراضات خاطئة، مما يؤدي إلى ترجمة غير صحيحة بصمت.

٧.٣.٨ التجميع المضمَّن ونداءات النظام

يمكن استخدام التجميع المضمَّن لاستدعاء نداءات النظام في بيئات محدودة. المثال التالي خاص بـ x86 Linux وغير قابل للنقل.

```
void write_stdout(const char *buf, unsigned len) {
    asm volatile (
        ".intel_syntax noprefix\n"
        "mov eax, 4\n"
        "mov ebx, 1\n"
        "mov ecx, %0\n"
        "mov edx, %1\n"
        "int 0x80\n"
        ".att_syntax prefix\n"
        :
        : "r"(buf), "r"(len)
```

```
        : "eax", "ebx", "ecx", "edx", "memory"
    );
}
```

هذا الأسلوب مناسب فقط للنواة، ومحمّلات الإقلاع، أو السياقات التعليمية التي تكون فيها ABI والمعمارية متحكمًا بها بالكامل.

٨.٣.٨ أفضل الممارسات للتجميع المضمَّن

١. استخدم التجميع المضمَّن فقط عند غياب بديل محمول.

٢. فضّل compiler intrinsics متى توفّرت.

٣. صرّح بجميع السجلات والأعلام وآثار الذاكرة.

٤. تجنّب تثبيت السجلات إلا إذا فرضه ABI.

٥. اعزل التجميع المضمَّن في وحدات صغيرة موثّقة جيدًا.

٤.٨ المعالجة المباشرة للذاكرة

تشير المعالجة المباشرة للذاكرة إلى الوصول إلى الذاكرة وتعديلها باستخدام العناوين الصريحة والعمليات المعتمدة على المؤشرات. وهي حجر الأساس في برمجة الأنظمة، والأنظمة المضمّنة، وبناء بيئات التشغيل.
بعكس التجريدات عالية المستوى، تكشف هذه الممارسات النموذج الحقيقي لتنفيذ الآلة وتخزينها.

١.٤.٨ المؤشرات والعنونة

تخزّن المؤشرات عناوين الذاكرة وتوفّر وصولاً غير مباشر إلى الكائنات.

```
int x = 10;
int *p = &x;
*p = 20;
```

تعتمد الصحة على كون المؤشر يشير إلى كائن صالح ذو عمر فعّال.

٢.٤.٨ الحساب على المؤشرات

يتقدم الحساب على المؤشرات بوحدات نوع المؤشر وليس بالبايتات.

```
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr;

for (int i = 0; i < 5; ++i) {
    printf("%d\n", *(p + i));
}
```

لا يُعرّف هذا الحساب إلا ضمن كائن واحد أو بعده مباشرة. وكل خروج عن ذلك يؤدي إلى سلوك غير معرّف.

٣.٤.٨ الحجز الديناميكي للذاكرة

يسمح الحجز الديناميكي بالتحكم في الذاكرة وقت التشغيل.

```
int *buf = malloc(10 * sizeof *buf);
if (!buf) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

/* use buffer */

free(buf);
```

يتحمل المبرمج مسؤولية الملكية والعمر وضمان تحرير الذاكرة مرة واحدة فقط.

٤.٤.٨ الوصول المباشر للذاكرة

يشيع الوصول إلى عناوين ذاكرة ثابتة في الأنظمة المضمّنة والتفاعل مع العتاد.

```
volatile uint32_t *reg = (volatile uint32_t *)0x1000;
*reg = 0xDEADBEEF;
```

يفترض هذا الأسلوب تخطيطًا عتاديًا صالحًا وهو خارج نموذج C المحمول.

٥.٤.٨ دوال الذاكرة القياسية

توفر مكتبة C عمليات على مستوى البايت مستقلة عن الأنواع.

```
memcpy(dest, src, size);
memset(buf, 0, size);
memcmp(a, b, size);
```

تعامل هذه الدوال الذاكرة كتخزين خام ويجب استخدامُها بانتباه للمحاذاة والحجم وعمر الكائنات.

٦.٤.٨ مثال: مُخصِّص ذاكرة بسيط

```
#define BLOCK_SIZE 1024
#define BLOCK_COUNT 8

static unsigned char pool[BLOCK_SIZE * BLOCK_COUNT];
static unsigned char *free_list[BLOCK_COUNT];
static int free_blocks = BLOCK_COUNT;

void allocator_init(void) {
    for (int i = 0; i < BLOCK_COUNT; ++i)
        free_list[i] = pool + i * BLOCK_SIZE;
}

void *alloc_block(void) {
    return free_blocks ? free_list[--free_blocks] : NULL;
}

void free_block(void *p) {
    free_list[free_blocks++] = p;
}
```

يوضح هذا المثال ملكية صريحة وحجزًا حتميًا دون الاعتماد على كومة النظام.

٧.٤.٨ أفضل الممارسات

١. لا تفك مرجع مؤشرات غير صالحة.

٢. استخدم volatile فقط للذاكرة المتغيرة خارجيًا.

٣. تجنّب انتهاك strict aliasing.

٤. حرّر الذاكرة مرة واحدة فقط.

٥. وثّق الملكية والتخطيط وعمر الذاكرة.

٨.٤.٨ الخلاصة

تكشف المعالجة المباشرة للذاكرة النموذج الحقيقي أسفل الآلة التجريدية للغة .C وتوفّر هذه التقنيات تحكمًا وتنبؤًا لا مثيل لهما، لكنها تتطلب دقةً، وانضباطًا، وفهمًا عميقًا للسلوك غير المعرّف. إن إتقان المؤشرات، والحجز، والوصول الخام للذاكرة يمكّن من بناء أنوية، وأنظمة مضمّنة، وبيئات تشغيل، وبرمجيات حرجة الأداء. أما إساءة استخدامها، فتؤدي إلى أخطاء دقيقة، وفساد صامت، وعدم استقرار النظام.

الفصل ٩: التفاعل مع أنظمة التشغيل

١.٩ نداءات النظام في C

تُشكّل نداءات النظام (System Calls) أدنى مستوى من الواجهة بين برامج فضاء المستخدم (User Space) ونواة نظام التشغيل (Kernel). وتسمح هذه النداءات للبرامج بطلب خدمات تتطلب صلاحيات مرتفعة، مثل إدخال/إخراج الملفات، وإدارة العمليات، وتخطيط الذاكرة، والاتصال بين العمليات.

لا تُعد نداءات النظام جزءاً من لغة ISO C23 نفسها. بل يتم تعريفها من قبل نظام التشغيل، ويتم الوصول إليها من برامج C عبر واجهات ثنائية خاصة بالنظام (ABI) ومغلّفات مكتبية (Library Wrappers).

إن فهم نداءات النظام ضروري لبرمجة الأنظمة، والتطبيقات الحساسة للأداء، والبيئات التي تكون فيها تجريدات المكتبة القياسية غير كافية أو غير متوفرة.

١.١.٩ ما هي نداءات النظام؟

نداء النظام هو نقطة دخول مضبوطة إلى نواة نظام التشغيل. تستخدم برامج فضاء المستخدم هذه النداءات لطلب خدمات لا يمكن تنفيذها مباشرةً بسبب آليات حماية العتاد.

تشمل الخدمات النموذجية التي توفّرها نداءات النظام:

- إدخال/إخراج الملفات والأجهزة — القراءة، والكتابة، وفتح، وإغلاق الملفات والأجهزة.

- إدارة العمليات — إنشاء العمليات، وإنهاؤها، ومزامنتها.
- إدارة الذاكرة — تخطيط الذاكرة الافتراضية وحمايتها.
- الاتصال بين العمليات — الأنابيب، والإشارات، والمقابس، والذاكرة المشتركة.

لا تنفّذ برامج المستخدم شيفرة النواة مباشرةً أبدًا؛ بل توفّر نداءات النظام الآلية الآمنة الوحيدة للانتقال.

٢.١.٩ وضع المستخدم مقابل وضع النواة

تفرض المعالجات الحديثة على الأقل نمطين للتنفيذ:

- وضع المستخدم — وصول مقيّد، حيث تعمل التطبيقات.
- وضع النواة — وصول كامل إلى العتاد والذاكرة، حيث يعمل نظام التشغيل.

تُحوّل نداءات النظام المعالج مؤقتًا من وضع المستخدم إلى وضع النواة. ويكون هذا الانتقال مضبوطًا بدقة من قبل المعالج ونظام التشغيل.

٣.١.٩ تدفق تنفيذ نداء النظام

يمر نداء النظام النموذجي بالمراحل التالية:

١. تجهيز المعاملات وفق واجهة ABI الخاصة بالمنصة.
٢. تنفيذ تعليمة خاصة تُطلق انتقال النمط.
٣. تتحقق النواة من المعاملات والصلاحيات.
٤. تنفّذ النواة الخدمة المطلوبة.
٥. يعود التنفيذ إلى وضع المستخدم بنتيجة أو خطأ.

تختلف التعليمة المستخدمة لدخول النواة حسب المعمارية:

- x86 (32-bit): `int 0x80`
- x86-64: `syscall`
- ARM: SVC (Supervisor Call)

٤.١.٩ نداءات النظام مقابل مكتبة C القياسية

من الضروري التمييز بين:

- نداءات النظام — واجهات النواة.
- دوال مكتبة C — مغلّفات ومساعدات في فضاء المستخدم.

تُعد دوال مثل open، read، وwrite في العادة مغلّفات رفيعة حول نداءات النظام، توفرها مكتبة النظام مثل glibc. أما دوال مثل printf فقد تستخدم عدة نداءات نظام داخليًا أو لا تستخدم أيًّا منها.
لا يعرّف معيار ISO C أي نداءات نظام.

٥.١.٩ نداءات النظام الشائعة في POSIX

في الأنظمة المتوافقة مع POSIX، تشمل نداءات النظام الشائعة:

- open, read, write, close
- fork, execve, wait
- pipe, dup, dup2
- mmap, munmap
- socket, bind, connect

يعتمد توفر هذه النداءات ودلالاتها على نظام التشغيل.

٦.١.٩ استخدام مغلّفات نداءات النظام في C

تستدعي معظم برامج C نداءات النظام عبر مغلّفات المكتبة القياسية، التي توفر التحقق من المعاملات، وتعديلات النقلية، ومعالجة الأخطاء.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    char buffer[128];
    ssize_t n = read(fd, buffer, sizeof buffer - 1);
    if (n == -1) {
        perror("read");
        close(fd);
        return 1;
    }

    buffer[n] = '\0';
    write(STDOUT_FILENO, buffer, n);

    close(fd);
    return 0;
}
```

يوضح هذا المثال التفاعل المباشر مع خدمات النواة مع البقاء ضمن C محمولة.

٧.١.٩ استدعاء نداءات النظام مباشرةً

في البيئات منخفضة المستوى مثل محمّلات الإقلاع والنواة وبيئات التشغيل المصغّرة، قد يكون من الضروري تجاوز المكتبة القياسية واستدعاء نداءات النظام مباشرةً.
في Linux، يمكن استخدام الدالة syscall للوصول إلى الواجهة الخام.

```
#include <unistd.h>
#include <sys/syscall.h>

int main(void) {
    const char *msg = "Hello, World!\n";
    syscall(SYS_write, STDOUT_FILENO, msg, 14);
    return 0;
}
```

هذا الأسلوب خاص بالنظام وABI ولا يجب استخدامه في التطبيقات المحمولة.

٨.١.٩ معالجة الأخطاء في نداءات النظام

تشير نداءات النظام عادةً إلى الفشل بإرجاع قيمة سالبة أو -1 وتعيين المتغير العالمي errno.

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main(void) {
    int fd = open("missing.txt", O_RDONLY);
```

```
    if (fd == -1) {
        perror("open");
        printf("errno value: %d\n", errno);
        return 1;
    }
    close(fd);
    return 0;
}
```

المتغير errno محلي لكل خيط تنفيذ ومعنوي فقط مباشرةً بعد حدوث الفشل.

٩.١.٩ إنشاء العمليات: fork و exec

يعتمد إنشاء العمليات في أنظمة Unix-like على نموذج من خطوتين:

- fork — إنشاء نسخة من العملية الحالية.
- exec — استبدال صورة العملية ببرنامج جديد.

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void) {
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    }
```

```
    if (pid == 0) {
        execlp("ls", "ls", "-l", NULL);
        perror("exec");
        _exit(1);
    }

    int status;
    wait(&status);
    printf("Child exited with status %d\n", status);
    return 0;
}
```

بعد نجاح exec، تزول الشيفرة الأصلية للبرنامج تماماً.

١٠.١.٩ اعتبارات الأداء

تُعد نداءات النظام مكلفة مقارنةً بالدوال العادية، لأنها:

- تُحدِث انتقال نمط للمعالج.
- تُعطّل خطوط الأنابيب والتكهّن.
- تتطلب تحقق النواة ومعالجة السياق.

تقلّل البرامج الكفؤة عدد نداءات النظام عبر تجميع العمليات أو استخدام إدخال/إخراج مُخزَّن.

١١.١.٩ أفضل الممارسات

١. تحقّق دائماً من قيم الإرجاع.

٢. فضّل مغلّفات المكتبة من أجل النقلية.

٣. عالج القراءات والكتابات الجزئية بشكل صحيح.

٤. لا تفترض الذرّية إلا إذا كانت مضمونة.

٥. وثّق السلوك الخاص بالنظام وABI.

١٢.١.٩ الخلاصة

تمثل نداءات النظام الحدّ الحقيقي بين شيفرة التطبيق ونواة نظام التشغيل. ويتطلب إتقانها فهم واجهاتها، ونموذج تنفيذها، وتكلفتها الأدائية، وقيودها النقلية.
عند استخدامها بشكل صحيح، تمكّن نداءات النظام برمجة منخفضة المستوى قوية، وبناء بيئات تشغيل مخصّصة، وبرمجيات أنظمة عالية الكفاءة. أما إساءة استخدامها، فتؤدي إلى شيفرة هشّة وغير محمولة. ويضع هذا الفصل الأساس الضروري للتفاعل الواعي والآمن مع أنظمة التشغيل باستخدام C23.

٢.٩ إدارة العمليات (fork، exec، wait)

تُعد إدارة العمليات أحد المسؤوليات الجوهرية لأنظمة التشغيل الحديثة. فهي التي تمكّن من إنشاء سياقات تنفيذ مستقلة، وتشغيلها، ومزامنتها، وإنهائها. وفي الأنظمة الشبيهة بـ Unix, تُبنى إدارة العمليات بشكل أساسي حول ثلاث نداءات نظام محورية: fork، exec، وwait.
تشكل هذه النداءات العمود الفقري لتعدد المهام، وبناء القِشَر (Shells), والتحكم في المهام (Job Control), والتعاون بين العمليات. ويُعد إتقانها شرطًا أساسيًا لبرمجة الأنظمة، وتصميم أنظمة التشغيل، وأي برمجيات تنسّق عمل عدة برامج مستقلة.

١.٢.٩ العمليات مقابل البرامج

البرنامج هو كيان ساكن: ملف تنفيذي موجود على القرص. أما العملية فهي كيان نشط: نسخة قيد التنفيذ من برنامج ما، تملك حالتها الخاصة.
تمتلك كل عملية:

- فضاء عناوين افتراضي خاص
- خيط تنفيذ واحد أو أكثر
- واصفات ملفات مفتوحة
- معرّف عملية (PID)
- بيانات جدولة ومحاسبة خاصة بالنواة

تتعامل نداءات إدارة العمليات مع العمليات، لا مع البرامج.

٢.٢.٩ نداء النظام fork

يقوم نداء النظام fork بإنشاء عملية جديدة عن طريق نسخ العملية المستدعية. وتُسمّى العملية الجديدة العملية الابنة (Child Process), بينما تُسمّى الأصلية العملية الأب (Parent Process).

من منظور المبرمج، يبدو أن fork يُعيد القيمة مرتين: مرة في الأب، ومرة في الابن.

التصريح

```
#include <unistd.h>

pid_t fork(void);
```

دلالات قيمة الإرجاع

- تُعيد 0 في العملية الابنة
- تُعيد PID الابن في العملية الأب
- تُعيد -1 عند الفشل وتضبط errno

بعد تنفيذ fork:

- يعمل الأب والابن بشكل مستقل
- تُنسخ الذاكرة منطقيًا (غالبًا باستخدام Copy-on-Write)
- تُشارك واصفات الملفات مع مشاركة مؤشرات الموضع

مثال: استخدام أساسي لـ fork

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

int main(void) {
    pid_t pid = fork();
```

```
    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid == 0) {
        printf("Child: PID=%d, PPID=%d\n", getpid(), getppid());
    } else {
        printf("Parent: PID=%d, Child PID=%d\n", getpid(), pid);
    }

    return 0;
}
```

٣.٢.٩ عائلة نداءات exec

تقوم عائلة نداءات exec باستبدال صورة العملية الحالية ببرنامج جديد. وعلى عكس fork، فإن exec لا تنشئ عملية جديدة.
بعد نجاح نداء exec:

- تزول شيفرة البرنامج الأصلي بالكامل
- يبقى PID دون تغيير
- تُستبدل الذاكرة، والمكدس، والسجلات
- تبقى واصفات الملفات مفتوحة ما لم يُطلب غير ذلك

إذا أعادت exec قيمة، فهذا يعني أنها فشلت.

أشهر صيغ exec

الدالة	الوصف
execl	قائمة معاملات، مسار صريح
execlp	قائمة معاملات، البحث في PATH
execv	مصفوفة معاملات، مسار صريح
execvp	مصفوفة معاملات، البحث في PATH
execle	بيئة تنفيذ مخصصة

مثال: استبدال العملية الابنة

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        execlp("ls", "ls", "-l", NULL);
        perror("exec");
        _exit(1);
    }

    return 0;
}
```

٤.٢.٩ نداء النظام wait

يسمح نداء النظام wait للعملية الأب بالانتظار حتى تنتهي إحدى عملياتها الأبناء، واسترجاع حالة انتهائها.

في حال عدم استدعاء wait، تبقى العمليات الأبناء المنتهية كـ عمليات زومبي (Zombie Processes) إلى أن يُعاد جمعها.

التصريح

```
#include <sys/wait.h>

pid_t wait(int *status);
```

ماكروات فحص الحالة

- WIFEXITED(status)
- WEXITSTATUS(status)
- WIFSIGNALED(status)
- WTERMSIG(status)

مثال: انتظار انتهاء عملية

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        sleep(1);
        return 42;
```

```
    }

    int status;
    wait(&status);

    if (WIFEXITED(status)) {
        printf("Child exited with code %d\n", WEXITSTATUS(status));
    }

    return 0;
}
```

٥.٢.٩ النمط القياسي لإنشاء العمليات

في التطبيق العملي، تتبع إدارة العمليات نمطًا شبه قياسي:

١. fork لإنشاء العملية الابنة

٢. exec في الابن لتحميل برنامج جديد

٣. wait في الأب للمزامنة

مثال: دورة حياة عملية كاملة

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();
```

```
    if (pid == 0) {
        execlp("ls", "ls", "-l", NULL);
        perror("exec");
        _exit(1);
    }

    int status;
    wait(&status);

    if (WIFEXITED(status)) {
        printf("Child exit code: %d\n", WEXITSTATUS(status));
    }

    return 0;
}
```

٦.٢.٩ اعتبارات الأداء والتصميم

- fork مُحسّن باستخدام Copy-on-Write
- exec عملية مكلفة وغير قابلة للتراجع
- إنشاء العمليات أثقل بكثير من إنشاء الخيوط
- الإفراط في fork يضر قابلية التوسع

لهذا تجمع الأنظمة الحديثة غالبًا بين العمليات والخيوط لتحقيق كفاءة أعلى.

٧.٢.٩ أفضل الممارسات في إدارة العمليات

١. تحقّق دائمًا من قيم الإرجاع.

٢. استخدم exit_ في مسارات الخطأ داخل الابن.

٣. اجمع العمليات الأبناء لتجنب الزومبي.

٤. تجنّب إنشاء عمليات غير ضرورية.

٥. وثّق بوضوح مسؤوليات الأب والابن.

٨.٢.٩ الخلاصة

تحدد إدارة العمليات باستخدام fork، exec، وwait نموذج التنفيذ في أنظمة Unix-like. وتوفر هذه النداءات آلية قوية وبسيطة في آن واحد لبناء القِشَر، والخوادم، والأنابيب، وبرمجيات الأنظمة المعقدة.

إن فهم دلالاتها الدقيقة — وليس مجرد طريقة استخدامها — ضروري لكتابة برامج منخفضة المستوى صحيحة، وكفؤة، وقابلة للصيانة. ويمثل هذا القسم الأساس المفاهيمي والعملي لموضوعات متقدمة مثل التحكم بالمهام، والإشارات، والعمليات الخلفية، ومجموعات العمليات.

٣.٩ إدارة الذاكرة في أنظمة التشغيل

تُعد إدارة الذاكرة إحدى أكثر المسؤوليات حساسيةً وأهمية في أي نظام تشغيل. فهي التي تتحكم في كيفية تخصيص الذاكرة، وحمايتها، وترجمتها، ومشاركتها، واسترجاعها، مع تحقيق توازن دقيق بين الأداء، والأمان، وقابلية التوسع. تعتمد أنظمة التشغيل الحديثة بشكل أساسي على آليات مدعومة عتاديًا تمنح كل عملية رؤية افتراضية خاصة للذاكرة مستقلة تمامًا عن التوزيع الفعلي لذاكرة RAM الفيزيائية.

يعرض هذا القسم رؤية دقيقة ومنظومية لإدارة الذاكرة، تشمل الذاكرة الافتراضية، والترقيم الصفحي، والتجزئة، والتخصيص الديناميكي، والحماية، والتجزؤ. ويُعد فهم هذه المفاهيم شرطًا أساسيًا لكتابة برمجيات أنظمة صحيحة، وكفؤة، ومتوقعة السلوك.

١.٣.٩ دور نظام التشغيل في إدارة الذاكرة

لا يقتصر دور نظام التشغيل على تلبية طلبات تخصيص الذاكرة. بل يتحكم بالذاكرة من خلال تصميم طبقي يشمل العتاد، وسياسات النواة، ومكتبات فضاء المستخدم.

يتحمل نظام التشغيل المسؤوليات التالية:

- تخصيص فضاءات عناوين افتراضية للعمليات
- ترجمة العناوين الافتراضية إلى عناوين فيزيائية
- فرض العزل والحماية بين العمليات
- إدارة الذاكرة الفيزيائية ووسائط التخزين الداعمة
- التعامل مع أخطاء الذاكرة والاستثناءات العتادية

وتتمحور أهداف التصميم حول:

- الكفاءة: تعظيم الاستفادة من RAM وتحسين المحلية (Locality)
- العزل: منع العمليات من الوصول إلى ذاكرة غيرها

- الأمان: فرض أذونات الوصول بدقة
- قابلية التوسع: دعم فضاءات عناوين ضخمة وأحمال عمل كبيرة

٢.٣.٩ الذاكرة الافتراضية

الذاكرة الافتراضية هي تجريد مدعوم عتاديًا يفصل فضاء عناوين العملية عن الذاكرة الفيزيائية. تعمل كل عملية ضمن فضاء عناوين افتراضي خاص بها، غالبًا ما يكون أكبر بكثير من حجم RAM المتوفر.
ولا تعني الذاكرة الافتراضية بحد ذاتها الاستبدال إلى القرص (Swapping). بل توفر بالأساس:

- عزل فضاءات العناوين
- حماية الذاكرة
- استخدامًا متفرقًا للذاكرة
- مشاركة مضبوطة بين العمليات

المفاهيم الأساسية للذاكرة الافتراضية

- فضاء العناوين الافتراضي: مجال عناوين متصل تراه العملية
- الصفحات :(Pages) كتل افتراضية ثابتة الحجم (غالبًا 4 كيلوبايت)
- الإطارات :(Frames) كتل فيزيائية بنفس الحجم
- جداول الصفحات: بنى تديرها النواة تربط الصفحات بالإطارات
- أخطاء الصفحات: استثناءات تُطلق عند غياب الترجمة أو انتهاك الأذونات

ولا يعني حدوث خطأ صفحة بالضرورة الوصول إلى القرص. فكثير من أخطاء الصفحات تنتج عن التحميل عند الطلب، أو انتهاك الأذونات، أو سلوك Copy-on-Write.

مثال: سلوك التحميل عند الطلب

```c
#include <stdlib.h>

int main(void) {
    int *buffer = malloc(1024 * 1024 * sizeof(int));
    if (!buffer) return 1;

    buffer[0] = 1;          /* First access triggers page allocation */
    buffer[1024 * 512] = 2;

    free(buffer);
    return 0;
}
```

في هذا المثال، لا تُخصّص الصفحات الفيزيائية عند استدعاء malloc، بل عند أول وصول فعلي إليها.

٣.٣.٩ الترقيم الصفحي والتجزئة

الترقيم الصفحي (Paging) والتجزئة (Segmentation) هما آليتان عتاديتان لإدارة الذاكرة. وهما ليستا واجهات برمجية، بل آليات داخلية ينفذها العتاد ونظام التشغيل.

الترقيم الصفحي

يقسم الترقيم الصفحي الذاكرة إلى وحدات ثابتة الحجم.

- يلغي التجزؤ الخارجي

- يسهّل الاستبدال والحماية
- يتطلب جداول صفحات وذاكرة TLB

يُعد الترقيم الصفحي الآلية السائدة في الأنظمة الحديثة.

التجزئة

تقسم التجزئة الذاكرة إلى مناطق منطقية مثل الشيفرة، والبيانات، والمكدس.

- توفر بنية دلالية واضحة
- تسمح بحماية دقيقة
- تعاني من التجزؤ الخارجي

تجمع الأنظمة الحديثة عادةً بين التجزئة (مفهوميًا) والترقيم الصفحي (فعليًا).

آثار الترقيم الصفحي المرئية

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

int main(void) {
    int *a = malloc(4096);
    int *b = malloc(8192);

    printf("%p %p\n", (void *)a, (void *)b);

    free(a);
    free(b);
    return 0;
}
```

تجاور العناوين الافتراضية لا يعني تجاورًا فيزيائيًا في الذاكرة.

٤.٣.٩ التخصيص الديناميكي للذاكرة

يُنفذ التخصيص الديناميكي في لغة C عبر مخصصات في فضاء المستخدم مثل malloc، وليس مباشرة من قبل النواة.
يتولى المخصص:

- إدارة بيانات وصفية للكومة
- طلب الذاكرة من نظام التشغيل على شكل كتل كبيرة
- إعادة استخدام الكتل المحررة

بينما توفر النواة الذاكرة باستخدام آليات مثل:

- sbrk / brk (تقنيات قديمة)
- munmap / mmap (الآلية الحديثة)

مثال: تخصيص كومة في C

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

int main(void) {
    int *arr = malloc(10 * sizeof(int));
    if (!arr) return 1;

    for (int i = 0; i < 10; i++) {
        arr[i] = i;
    }

    free(arr);
```

```
    return 0;
}
```

لا يعني استدعاء free بالضرورة إرجاع الذاكرة إلى نظام التشغيل، بل إعادتها إلى المخصص.

٥.٣.٩ حماية الذاكرة والعزل

تُفرض حماية الذاكرة عبر وحدة إدارة الذاكرة (MMU) باستخدام أذونات على مستوى الصفحات. يمكن وسم كل صفحة بأنها:

- قابلة للقراءة
- قابلة للكتابة
- قابلة للتنفيذ
- متاحة لفضاء المستخدم أو للنواة

وأي انتهاك يؤدي إلى استثناء عتادي.

مثال: انتهاك حماية

```
#include <stdlib.h>

int main(void) {
    int *p = malloc(10 * sizeof(int));
    p[10] = 42;  /* Undefined behavior: may trigger SIGSEGV */
    free(p);
    return 0;
}
```

يكتشف نظام التشغيل الوصول غير القانوني ويُنهي العملية.

٦.٣.٩ تجزؤ الذاكرة

يحدث التجزؤ عندما تؤدي أنماط التخصيص إلى تقليل الذاكرة المتصلة القابلة للاستخدام.

التجزؤ الداخلي

- مساحة مهدرة داخل الكتل المخصصة
- ناتجة عن المحاذاة وحجم الصفحات

التجزؤ الخارجي

- ذاكرة حرة لكنها غير متصلة
- يؤثر بشكل خاص على مخصصات الكومة

مثال: التجزؤ الخارجي

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

int main(void) {
    void *a = malloc(100);
    void *b = malloc(200);
    void *c = malloc(100);

    free(b);

    void *d = malloc(250);
    if (!d) {
        printf("Allocation failed due to fragmentation\n");
```

```
    }

    free(a);
    free(c);
    return 0;
}
```

٧.٣.٩ أفضل الممارسات في إدارة الذاكرة

١. افهم سلوك المخصص، لا تكتفِ باستخدام الواجهة.

٢. تحقّق دائمًا من نتائج التخصيص.

٣. تجنّب الوصول خارج حدود الذاكرة.

٤. قلّل عمليات التخصيص والتحرير المتكررة.

٥. استخدم mmap للمناطق الكبيرة أو المتخصصة.

٨.٣.٩ الخلاصة

إدارة الذاكرة هي تعاون طبقي بين العتاد، ونظام التشغيل، ومكتبات فضاء المستخدم. توفر الذاكرة الافتراضية العزل، والحماية، وقابلية التوسع، بينما يمكّن الترقيم الصفحي من استخدام فعّال للذاكرة الفيزيائية.
إن الفهم العميق لهذه الآليات يُمكّن المطور من تحليل الأداء، والتصرف الصحيح عند الفشل، وبناء برمجيات أنظمة موثوقة. ويؤسس هذا القسم للانتقال إلى موضوعات متقدمة مثل Copy-on-Write، والذاكرة المشتركة، وهياكل NUMA، وإدارة الذاكرة داخل النواة.

٤.٩ صلاحيات الملفات والعمليات

تُشكّل صلاحيات الملفات والعمليات الأساس الجوهري للأمان والعزل في أنظمة التشغيل متعددة المستخدمين. فهي التي تنظّم من يحق له الوصول إلى الملفات، وتنفيذ البرامج، وإرسال الإشارات للعمليات، وأداء العمليات ذات الامتيازات الخاصة. ولا تُعد الصلاحيات ميزةً اختيارية، بل هي جزء متكامل من منطق التحكم بالوصول داخل النواة، ويجري فرضها عند كل حدٍّ فاصل للاستدعاءات النظامية.
يعرض هذا القسم شرحًا دقيقًا ومنظوميًا لصلاحيات الملفات والعمليات، وكيفية تمثيلها، وتطبيقها، واستخدامها الصحيح في برامج C23.

١.٤.٩ مفهوم الصلاحيات في أنظمة التشغيل

توجد الصلاحيات لفرض التحكم بالوصول في البيئات المشتركة. ويستخدمها نظام التشغيل لضمان:

- السرية: منع الوصول غير المصرّح به للبيانات
- السلامة: منع التعديل غير المصرّح به
- الإتاحة: منع التخريب أو التعطيل المتعمّد
- المساءلة: ربط الأفعال بهويات محددة

يُقيّم كل قرار وصول من قِبل النواة اعتماداً على:

- بيانات اعتماد العملية (UID، GID، القدرات)
- ملكية الكائن (ملفات، مقابس، كائنات IPC)
- بتّات الصلاحيات والأعلام الأمنية

٢.٤.٩ صلاحيات الملفات

تُحدّد صلاحيات الملفات كيفية الوصول إلى الملفات والمجلدات. في الأنظمة الشبيهة بـ Unix، تُشفَّر الصلاحيات على شكل أقنعة بتّية مخزّنة في بيانات inode، وتُفرض من قبل النواة أثناء استدعاءات نظامية مثل open، وexecve، وunlink.

فئات الصلاحيات

يمتلك كل ملف صلاحيات معرّفة لثلاث فئات:

١. المالك (Owner)

٢. المجموعة (Group)

٣. الآخرون (Others)

ويمكن لكل فئة امتلاك صلاحيات القراءة، والكتابة، والتنفيذ.

بتّات الصلاحيات

الصلاحية	الرمز	المعنى
قراءة	r	قراءة محتوى الملف أو عرض محتويات المجلد
كتابة	w	تعديل محتوى الملف أو المجلد
تنفيذ	x	تنفيذ الملف أو اجتياز المجلد

على سبيل المثال، rwxr-xr-- تعني:

- المالك: وصول كامل
- المجموعة: قراءة وتنفيذ
- الآخرون: قراءة فقط

التمثيل الرقمي

يمكن تمثيل الصلاحيات أيضًا بالترميز الثماني:

- القراءة = 4
- الكتابة = 2
- التنفيذ = 1

وبذلك فإن rwxr-xr-- تقابل القيمة 754.

مثال: استعراض الصلاحيات

```
$ ls -l example.txt
-rw-r--r-- 1 user group 1024 Oct 10 12:34 example.txt
```

٣.٤.٩ تغيير صلاحيات الملفات

تُعدَّل صلاحيات الملفات باستخدام الاستدعاء النظامي chmod، الذي يُحدّث بيانات inode.

الصيغة

```c
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

مثال

```c
#include <sys/stat.h>
#include <stdio.h>
```

```c
int main(void) {
    if (chmod("example.txt",
              S_IRUSR | S_IWUSR |
              S_IRGRP |
              S_IROTH) == -1) {
        perror("chmod failed");
        return 1;
    }
    return 0;
}
```

يضبط هذا المثال الصلاحيات إلى rw-r--r-- (644).

٤.٤.٩ بيانات اعتماد العمليات وصلاحياتها

لا تمتلك العمليات صلاحيات مباشرة، بل تحمل بيانات اعتماد تستخدمها النواة عند تقييم طلبات الوصول.
تمتلك كل عملية:

- معرّف المستخدم الحقيقي (RUID)
- معرّف المستخدم الفعّال (EUID)
- المعرّف المحفوظ (SUID)
- ومعرّفات المجموعات المقابلة

مثال: فحص بيانات اعتماد العملية

```c
#include <unistd.h>
#include <stdio.h>
```

```
int main(void) {
    printf("RUID: %d\n", getuid());
    printf("EUID: %d\n", geteuid());
    printf("RGID: %d\n", getgid());
    printf("EGID: %d\n", getegid());
    return 0;
}
```

٥.٤.٩ تغيير بيانات اعتماد العملية

تقيّد النواة تغيير بيانات الاعتماد. فخفض الامتيازات مسموح دائمًا، أما رفعها فيتطلب امتيازًا مسبقًا.

الاستدعاءات المرتبطة

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

مثال: إسقاط الامتيازات

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

int main(void) {
    if (setuid(getuid()) == -1) {
        perror("setuid failed");
        return 1;
```

```
    }
    return 0;
}
```

يُعد إسقاط الامتيازات ممارسةً أمنية أساسية.

٦.٤.٩ بتّات الصلاحيات الخاصة

تدعم أنظمة ملفات Unix بتّات خاصة تغيّر سلوك التنفيذ.

الصلاحيات الخاصة

- Setuid (SUID) — التنفيذ بهوية المالك
- Setgid (SGID) — توريث المجموعة
- Bit Sticky — منع الحذف غير المصرّح به في المجلدات المشتركة

مثال: ضبط البتّات الخاصة

```
#include <sys/stat.h>
#include <stdio.h>

int main(void) {
    if (chmod("example",
              S_ISUID | S_ISGID |
              S_IRWXU |
              S_IRGRP | S_IXGRP |
              S_IROTH | S_IXOTH) == -1) {
        perror("chmod failed");
        return 1;
```

```
    }
    return 0;
}
```

٧.٤.٩ الآثار الأمنية للصلاحيات

يُعد سوء إدارة الصلاحيات أحد الأسباب الرئيسية لثغرات تصعيد الامتيازات. ومن المخاطر الشائعة:

- صلاحيات مفرطة
- استخدام خاطئ لـ SUID
- عدم إسقاط الامتيازات
- التعامل غير الآمن مع الملفات المؤقتة

٨.٤.٩ أفضل الممارسات

١. تطبيق مبدأ أقل الامتيازات
٢. تجنب ملفات SUID قدر الإمكان
٣. إسقاط الامتيازات مبكرًا
٤. التحقق من تغييرات الصلاحيات
٥. التدقيق الدوري لصلاحيات النظام

٩.٤.٩ الخلاصة

تُفرض صلاحيات الملفات والعمليات من قبل النواة عند كل نقطة وصول. ويُعد الفهم الدقيق لنموذج الصلاحيات، وتدفق بيانات الاعتماد، وقواعد الفرض أمرًا جوهريًا لبناء برمجيات أنظمة آمنة. ويؤسس هذا القسم للانتقال إلى موضوعات متقدمة مثل Capabilities، والعزل (Sandboxing)، وNamespaces، وآليات التحكم الإجباري بالوصول.

الفصل ١٠: أساسيات تصميم المترجمات

١.١٠ مقدمة في تصميم المترجمات

يُعد تصميم المترجمات أحد التخصصات الأساسية في برمجيات الأنظمة، إذ يربط لغات البرمجة بالسلوك التنفيذي الفعلي على مستوى العتاد. تقوم المترجمات بترجمة الشيفرة المصدرية المكتوبة بلغة عالية المستوى (مثل C23) إلى شيفرة آلة أو إلى تمثيل وسيط منخفض المستوى قابل للتنفيذ على معمارية مستهدفة.
يكشف فهم تصميم المترجمات كيف تُحلَّل تراكيب اللغة، وتُحوَّل، وتُحسَّن، ثم تُسقَط في النهاية على تعليمات المعالج. يقدّم هذا القسم المفاهيم الأساسية، والمراحل، والمكوّنات التي تشكّل العمود الفقري للمترجمات الحديثة.

١.١.١٠ ما هو المترجم؟

المترجم هو برنامج يقوم بتحويل الشيفرة المصدرية إلى تمثيل أدنى مستوى مع الحفاظ على دلالات البرنامج. وقد يكون الناتج: شيفرة آلة، أو لغة تجميع، أو بايت كود، أو تمثيلاً وسيطًا (IR).
تشمل الخصائص الأساسية للمترجم:

- الترجمة: تحويل البرامج المصدرية إلى شكل قابل للتنفيذ أو التحليل
- التحليل: التحقق من الصحة النحوية والدلالية

- التحسين: تحسين الأداء أو الحجم دون تغيير السلوك المرصود
- الاستهداف: مواءمة الشيفرة لمعمارية وبيئة محددة

وعلى خلاف المفسّرات، تُنجز المترجمات معظم التحليل والتحويل قبل وقت التنفيذ.

٢.١.١٠ لماذا ندرس تصميم المترجمات؟

لا يقتصر تصميم المترجمات على بناء لغات جديدة، بل يتجاوز ذلك إلى:

١. الوعي بالأداء: فهم الترجمة يساعد على كتابة شيفرة تتوافق بكفاءة مع العتاد

٢. بناء اللغات: كل لغة جديدة تتطلب توصيفًا معجميًا ونحويًا ودلاليًا

٣. فهم التحذيرات والأخطاء: سلوك المترجم يفسّر التحسينات والتحذيرات والسلوك غير المعرّف

٤. برمجة الأنظمة: المترجمات مرتبطة بـ ABI واتفاقيات الاستدعاء وأنظمة التشغيل

٥. الأدوات المتقدمة: المحللات الساكنة وأدوات الفحص تعتمد تقنيات المترجمات

٣.١.١٠ مراحل عمل المترجم

يعمل المترجم كسلسلة من المراحل المحددة بدقة. وتُقسّم عادة إلى: الواجهة الأمامية والواجهة الخلفية، يربط بينهما تمثيل وسيط.

الواجهة الأمامية

تُحلّل الواجهة الأمامية لغة المصدر وتنتج تمثيلاً مجرّداً مستقلاً عن العتاد.

١. التحليل المعجمي **(Lexical Analysis)**

- تحويل تدفّق المحارف إلى رموز (Tokens)

- تجاهل الفراغات والتعليقات
- **مثال:** `int x = 10;` → `int`, `x`, `=`, `10`, `;`

٢. التحليل النحوي (Parsing)

- تنظيم الرموز وفق قواعد اللغة
- إنتاج شجرة تحليل أو شجرة صياغة مجردة (AST)

٣. التحليل الدلالي

- ربط المعرّفات وتحديد النطاقات والأنواع
- فرض قواعد اللغة غير النحوية
- بناء جداول الرموز وتعليم الشجرة

الواجهة الخلفية

تحوّل الواجهة الخلفية التمثيل الوسيط إلى شيفرة تنفيذية لمعمارية محددة.

١. توليد التمثيل الوسيط

٢. التحسين

٣. توليد الشيفرة

٤. إخراج الشيفرة

٤.١.١٠ المكوّنات الأساسية للمترجم

المحلل المعجمي

- تقسيم النص إلى رموز

- يعتمد على آلات منتهية
- غالبًا يُولَّد من تعابير منتظمة

المحلل النحوي

- يفرض البنية النحوية
- يستخدم قواعد خالية من السياق
- ينتج AST

المحلل الدلالي

- يتحقق من الأنواع والنطاقات
- يكتشف الأخطاء الدلالية
- يدير جداول الرموز

توليد الشيفرة الوسيطة

```
int x = 10;
int y = x + 5;
```

تمثيل وسيط محتمل:

```
t1 = 10
x = t1
t2 = x + 5
y = t2
```

المُحسِّن

```
int x = 10 + 5;
```

بعد التحسين:

```
int x = 15;
```

مولّد الشيفرة

```
mov eax, 10
mov [x], eax
add eax, 5
mov [y], eax
```

٥.١.١٠ الأدوات والتقنيات

تعتمد المترجمات الحديثة على تقنيات راسخة:

- التعابير المنتظمة والآلات المنتهية
- خوارزميات LL و LR
- تحليل تدفّق التحكم والبيانات
- تمثيلات SSA
- تلوين الرسوم لتخصيص المسجلات

٦.١.١٠ مثال: مسار الترجمة الكامل

المصدر:

```
int main() {
    int x = 10;
    int y = x + 5;
    return y;
}
```

يمر عبر:

١. التحليل المعجمي
٢. بناء AST
٣. التحقق الدلالي
٤. توليد IR
٥. التحسين
٦. توليد الشيفرة النهائية

٧.١.١٠ الخلاصة

يؤطر تصميم المترجمات التحويل المنهجي من دلالات اللغة إلى التنفيذ الآلي. وهو يدمج النظرية بالهندسة منخفضة المستوى، ويربط لغات البرمجة بأنظمة التشغيل ومعمارية الحاسوب.
يمهّد هذا الفصل لفهم المترجمات الواقعية، ويؤسس للغوص العميق في التحليل المعجمي، والنحوي، والتمثيل الوسيط، وتوليد الشيفرة.

٢.١٠ التحليل النحوي (Syntax Analysis -- Parsing)

يُعد التحليل النحوي، المعروف باسم Parsing، المرحلة الرئيسية الثانية في عملية الترجمة. تستهلك هذه المرحلة تدفّق الرموز (Tokens) الناتج عن المحلل المعجمي، وتتحقق من أن ترتيب هذه الرموز يطابق القواعد النحوية الرسمية للغة البرمجة.
يقوم المُحلّل النحوي بإنشاء البنية التركيبية للبرنامج عبر بناء تمثيل هرمي، غالبًا على شكل شجرة تحليل أو تمثيل أكثر تجريدًا يُعرف بـ شجرة الصياغة المجرّدة (AST). ويشكّل هذا التمثيل الأساس للتحليل الدلالي، والتحسين، وتوليد الشيفرة.

١.٢.١٠ دور التحليل النحوي

يتحمّل التحليل النحوي المسؤوليات التالية:

١. فرض القواعد النحوية: التأكد من أن تسلسل الرموز يخضع لقواعد اللغة

٢. التنظيم البنيوي: تجميع الرموز في تراكيب ذات معنى

٣. الاكتشاف المبكر للأخطاء: تحديد الانتهاكات النحوية بدقة

إذا كان التحليل المعجمي يتعرّف على الكلمات، فإن التحليل النحوي يفهم الجُمل.

٢.٢.١٠ القواعد الخالية من السياق

تُعرَّف معظم لغات البرمجة باستخدام القواعد الخالية من السياق (Context-Free Grammars -- CFGs). وتتكوّن القاعدة من:

١. الرموز النهائية (Terminals): وهي الرموز القادمة من المُحلّل المعجمي

٢. الرموز غير النهائية (Non-terminals): فئات نحوية مجردة

٣. قواعد الإنتاج: قواعد إعادة الكتابة

٤. رمز البداية: يمثل البرنامج الكامل

مثال: قواعد تعبيرات حسابية

```
expression → expression + term
           | expression - term
           | term

term → term * factor
     | term / factor
     | factor

factor → number
       | '(' expression ')'
```

تعبّر هذه القواعد صراحةً عن أولوية العمليات والارتباطية.

٣.٢.١٠ شجرة التحليل و شجرة الصياغة المجرّدة

تمثل شجرة التحليل الاشتقاق النحوي الكامل للإدخال، بما في ذلك جميع الرموز غير النهائية. ورغم دقتها، فإنها غالبًا مطوّلة وغير مناسبة للمراحل اللاحقة.
أما شجرة الصياغة المجرّدة (AST) فتُزيل البنية النحوية الزائدة، وتحتفظ فقط بالعلاقات الدلالية الجوهرية.

مثال: 2 + 3 * 4

شجرة التحليل (مبسطة):

```
expression
  expression
    term
       factor
```

```
          number: 2
  '+'
  term
     term
       factor
          number: 3
     '*'
     factor
        number: 4
```

شجرة الصياغة المجرّدة المقابلة:

```
+
  2
  *
     3
     4
```

تعكس الـ AST ترتيب التقييم والنية الدلالية مباشرةً.

٤.٢.١٠ استراتيجيات التحليل النحوي

تُصنَّف خوارزميات التحليل وفق اتجاه بناء الاشتقاق.

التحليل من الأعلى إلى الأسفل

يبدأ هذا النوع من رمز البداية ويحاول اشتقاق الإدخال.

- التحليل التراجعي **(Recursive Descent):** سهل القراءة ومرن
- LL(k): تحليل تنبؤي يعتمد على نظرة مستقبلية بعدد k من الرموز

يتطلب هذا الأسلوب قواعد خالية من الاستدعاء اليساري.

التحليل من الأسفل إلى الأعلى

يبني هذا النوع التحليل ابتداءً من الرموز صعوداً إلى رمز البداية.

- **Shift-Reduce Parsing**
- LR(k): قوي وقادر على معالجة معظم اللغات

تستخدم مترجمات C الحديثة محللات قائمة على LR.

٥.٢.١٠ مولدات المحللات

يُعد بناء المحللات يدويًا عرضة للأخطاء في القواعد الكبيرة، لذا تُستخدم أدوات توليد.

Yacc و Bison

تقوم Yacc و Bison بتوليد محللات LR انطلاقًا من توصيف القواعد.

مثال: جزء من قواعد Bison

```
%token NUMBER

%%
expression:
      expression '+' term
    | expression '-' term
    | term
    ;

term:
      term '*' factor
```

```
    | term '/' factor
    | factor
    ;

factor:
      NUMBER
    | '(' expression ')'
    ;
%%
```

يمكن ربط أفعال دلالية لبناء عقد AST.

٦.٢.١٠ معالجة الأخطاء النحوية

تُعد معالجة الأخطاء جزءاً أساسيًا من تجربة المستخدم.

١. **Panic Mode:** تجاوز الإدخال حتى رمز تزامن

٢. **Error Productions:** قواعد خاصة للأخطاء الشائعة

٣. **Error Tokens:** استعادة مضبوطة ومواصلة التحليل

مثال: قاعدة خطأ

```
statement:
      expression ';'
    | error ';'
    ;
```

٧.٢.١٠ مثال عملي على التحليل

الشيفرة المصدرية:

```
int main() {
    int x = 10;
    return x;
}
```

تمثيل AST:

```
function_decl
  return_type: int
  name: main
  parameters: ()
  body
     declaration
        type: int
        name: x
        initializer: 10
     return_stmt
         expression: x
```

يُمرَّر هذا التمثيل إلى مرحلة التحليل الدلالي.

٨.٢.١٠ أفضل الممارسات

١. تصميم قواعد واضحة وغير ملتبسة

٢. فصل التحليل عن الدلالات

٣. تفضيل AST على شجرة التحليل الكاملة

٤. توفير رسائل أخطاء دقيقة وقابلة للاسترداد

٥. اختيار خوارزمية التحليل المناسبة لتعقيد اللغة

٩.٢.١٠ الخلاصة

يحوّل التحليل النحوي تدفّقًا خطيًا من الرموز إلى تمثيل هرمي منظم. وهو يتحقق من صحة البنية ويؤسس العلاقات اللازمة للاستدلال الدلالي.
المحلل النحوي الجيد يُبسّط جميع المراحل اللاحقة. ومع اكتمال التحليل، يصبح المترجم جاهزًا لفهم المعنى، والأنواع، والنطاقات، وهو مجال التحليل الدلالي.

٣.١٠ توليد الشيفرة والتحسين

تمثل مرحلتا توليد الشيفرة والتحسين المراحل النهائية والحاسمة في خط أنابيب الترجمة. في هذه النقطة، يكون المترجم قد تحقّق من الصحة النحوية والدلالية، وحول البرنامج إلى تمثيل وسيط يعكس معناه بدقة.
تتمثل مهمة الواجهة الخلفية في تحويل هذا التمثيل إلى شيفرة آلة صحيحة وفعّالة لمعمارية مستهدفة، مع تطبيق تحسينات ترفع الأداء أو تقلل الحجم دون تغيير السلوك.

١.٣.١٠ هدف توليد الشيفرة

يسعى توليد الشيفرة إلى:

١. الصحة الدلالية: الحفاظ على سلوك البرنامج
٢. الكفاءة المعمارية: استغلال خصائص المعالج
٣. قابلية النقل: دعم معماريات متعددة بأقل تعديل

٢.٣.١٠ خط أنابيب الواجهة الخلفية

١. اختيار التعليمات
٢. تخصيص المسجلات
٣. جدولة التعليمات
٤. إخراج الشيفرة النهائية

٣.٣.١٠ اختيار التعليمات

يحوّل هذا الطور عمليات IR إلى تعليمات حقيقية.
تمثيل وسيط:

```
t1 = a + b
t2 = t1 * c
```

تجميع x86 محتمل:

```
mov eax, [a]
add eax, [b]
imul eax, [c]
mov [t2], eax
```

٤.٣.١٠ تخصيص المسجلات

تُخصَّص القيم إلى عدد محدود من المسجلات السريعة.

٥.٣.١٠ جدولة التعليمات

تعيد هذه المرحلة ترتيب التعليمات المستقلة لتحسين التوازي وتقليل التوقفات.

٦.٣.١٠ التحسين

يهدف التحسين إلى رفع جودة الشيفرة دون تغيير معناها.

مثال: الطي الثابت

```
t1 = 10 + 20
t2 = t1 * 2
```

بعد التحسين:

```
t2 = 60
```

٧.٣.١٠ الخلاصة

تحوّل مرحلتا توليد الشيفرة والتحسين المعنى المجرّد إلى سلوك تنفيذي فعلي. وهما ما يميز مترجمًا يُنتج شيفرة صحيحة فقط عن مترجم يُنتج شيفرة فعّالة حقًا.
إتقان هذه المراحل هو الخط الفاصل بين بناء لغة وبين هندسة مترجمات احترافية حقيقية.

الفصل ١١: موضوعات متقدمة في C23

١.١١ البرمجة متعددة الخيوط في C23

تُعد البرمجة متعددة الخيوط (Multithreading) إحدى الركائز الأساسية في برمجة الأنظمة الحديثة، إذ تسمح بوجود عدة مسارات تنفيذ تعمل بالتوازي داخل عملية واحدة. ويمكّن ذلك البرامج من الاستفادة الكاملة من المعالجات متعددة الأنوية، وزيادة الإنتاجية، والحفاظ على الاستجابة تحت الضغط.

تواصل C23 البناء على نموذج الخيوط المعياري الذي أُدخل في C11، موفرةً واجهة برمجية محمولة ومحددة بدقة لإنشاء الخيوط، ومزامنتها، وتنسيقها. يقدّم هذا القسم الأسس المفاهيمية والتقنيات العملية اللازمة لكتابة برامج متعددة الخيوط صحيحة وفعّالة باستخدام C23.

١.١.١١ النموذج المفاهيمي للبرمجة متعددة الخيوط

يمثّل الخيط (Thread) مسار تنفيذ مستقل داخل العملية الواحدة. تشترك جميع الخيوط في نفس فضاء العناوين، بما في ذلك المتغيرات العامة والذاكرة الديناميكية، بينما يحتفظ كل خيط بمكدسه وسياق تنفيذه الخاص.

تشمل المزايا الرئيسية للبرمجة متعددة الخيوط:

١. التوازي: تنفيذ مهام مستقلة على أنوية متعددة

٢. إخفاء الكمون: التداخل بين الحساب وعمليات الإدخال والإخراج

٣. مشاركة الموارد: مشاركة الهياكل البيانية دون الحاجة لتواصل بين العمليات

غير أن هذه المزايا تأتي مع تعقيد إضافي، يتطلب مزامنة صريحة وتصميمًا منضبطًا.

٢.١.١١ إنشاء الخيوط ودورة حياتها

تُعرّف C23 واجهة موحدة لإدارة الخيوط عبر الترويسة <threads.h>, مما يسمح ببرمجة محمولة عبر المنصات المختلفة.

إنشاء الخيوط

يتم إنشاء الخيوط باستخدام الدالة thrd_create، والتي تطلق خيطًا جديدًا ينفّذ دالة يحددها المستخدم.

```c
#include <threads.h>
#include <stdio.h>

int thread_function(void *arg) {
    int *value = arg;
    printf("Thread running with value: %d\n", *value);
    return 0;
}

int main(void) {
    thrd_t thread;
    int value = 42;

    if (thrd_create(&thread, thread_function, &value) != thrd_success) {
        return 1;
    }
```

```
    thrd_join(thread, NULL);
    return 0;
}
```

يبدأ الخيط التنفيذ فور إنشائه، ويشارك فضاء العناوين مع الخيط الأب.

إنهاء الخيوط

ينتهي الخيط إما بإرجاع قيمة من دالة الدخول، أو باستدعاء thrd_exit صراحةً.

```
int thread_function(void *arg) {
    printf("Thread running\n");
    thrd_exit(0);
}
```

لا يؤدي إنهاء الخيط إلى المزامنة التلقائية مع الخيوط الأخرى.

٣.١.١١ بدائيات المزامنة

نظرًا لأن الخيوط تتشارك الذاكرة، فإن المزامنة ضرورية لتجنب تضارب البيانات (Data Races) والسلوك غير المعرّف.

الأقفال المتبادلة (Mutexes)

يفرض القفل المتبادل الإقصاء المتبادل، بحيث لا يسمح إلا لخيط واحد بالوصول إلى المورد المحمي.

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

mtx_t mutex;
```

```
int shared_data = 0;

int thread_function(void *arg) {
    mtx_lock(&mutex);
    shared_data++;
    printf("Thread %ld: shared_data = %d\n",
           (long)arg, shared_data);
    mtx_unlock(&mutex);
    return 0;
}
```

يُعرّف القفل نقطة إقصاء متبادل ونقطة مزامنة في نموذج الذاكرة لـ C.

متغيرات الشرط (Condition Variables)

تسمح متغيرات الشرط للخيوط بالانتظار بكفاءة حتى يتغير وضع معين، دون الحاجة إلى الانتظار النشط.

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

mtx_t mutex;
cnd_t cond;
int ready = 0;

int producer(void *arg) {
    mtx_lock(&mutex);
    ready = 1;
    cnd_signal(&cond);
    mtx_unlock(&mutex);
```

```
    return 0;
}

int consumer(void *arg) {
    mtx_lock(&mutex);
    while (!ready) {
        cnd_wait(&cond, &mutex);
    }
    printf("Consumer: ready = %d\n", ready);
    mtx_unlock(&mutex);
    return 0;
}
```

توفّر متغيرات الشرط تنسيقًا فعّالاً دون إهدار للمعالج.

٤.١.١١ التخزين المحلي للخيوط

يوفر التخزين المحلي للخيوط (Thread-Local Storage -- TLS) نسخًا مستقلة من المتغيرات لكل خيط، مما يقلل الحاجة للمزامنة.

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

_Thread_local int thread_local_var;

int thread_function(void *arg) {
    thread_local_var = (int)(long)arg;
    printf("Thread %ld: TLS = %d\n",
           (long)arg, thread_local_var);
    return 0;
```

```
}
```

يُستخدم TLS بكثرة في التخزين المؤقت، والمخازن المؤقتة، وحالات الخطأ.

٥.١.١١ مبادئ تصميم البرامج متعددة الخيوط

البرمجة متعددة الخيوط هي مشكلة تصميم قبل أن تكون مشكلة تركيب لغوي.

١. تقليل الحالة المشتركة قدر الإمكان
٢. تحديد ملكية البيانات بوضوح
٣. استخدام الأقفال باعتدال
٤. تجنب دورات ترتيب الأقفال
٥. افتراض ترتيب ذاكرة ضعيف

٦.١.١١ مثال عملي: حساب الأعداد الأولية بالتوازي

```
#include <threads.h>
#include <stdbool.h>
#include <stdio.h>

#define NUM_THREADS 4
#define RANGE 100000

mtx_t mutex;
int prime_count = 0;

bool is_prime(int n) {
```

```
    if (n < 2) return false;
    for (int i = 2; i * i <= n; ++i)
        if (n % i == 0) return false;
    return true;
}

int worker(void *arg) {
    int id = (int)(long)arg;
    int start = id * (RANGE / NUM_THREADS);
    int end   = start + (RANGE / NUM_THREADS);

    for (int i = start; i < end; ++i) {
        if (is_prime(i)) {
            mtx_lock(&mutex);
            prime_count++;
            mtx_unlock(&mutex);
        }
    }
    return 0;
}

int main(void) {
    thrd_t threads[NUM_THREADS];

    mtx_init(&mutex, mtx_plain);

    for (int i = 0; i < NUM_THREADS; ++i)
        thrd_create(&threads[i], worker, (void *)(long)i);
```

```
    for (int i = 0; i < NUM_THREADS; ++i)
        thrd_join(threads[i], NULL);

    mtx_destroy(&mutex);

    printf("Total prime numbers: %d\n", prime_count);
    return 0;
}
```

يعالج كل خيط نطاقًا مستقلاً، وتتم المزامنة فقط عند تحديث الحالة المشتركة.

٧.١.١١ الخلاصة

توفر البرمجة متعددة الخيوط في C23 أساسًا معياريًا ومحمولاً للبرمجة المتزامنة على مستوى الأنظمة. ورغم توفر الأدوات الأساسية، تبقى الصحة مسؤولية المبرمج.
يتطلب إتقان هذا المجال فهم تداخلات التنفيذ، ورؤية الذاكرة، وانضباط ملكية البيانات. وعند استخدامها بشكل صحيح، تمكّن هذه الأدوات من بناء برمجيات عالية الأداء وقابلة للتوسع تستغل العتاد الحديث بكفاءة كاملة.

٢.١١ الشبكات باستخدام المقابس (Sockets)

تُعد الشبكات مكوّنًا أساسيًا في برمجيات الأنظمة الحديثة، إذ تمكّن العمليات من التواصل عبر الأجهزة والشبكات. في C23، تُنفّذ الشبكات باستخدام واجهة المقابس (POSIX Socket API)، والتي توفّر واجهة منخفضة المستوى ومباشرة لمكدس الشبكة في النظام.
تُعامل المقابس ككائنات نواة تشبه الملفات، مما يسمح للبرامج بإنشاء الاتصالات، وتبادل البيانات، وإدارة الموارد الشبكية بدقة عالية.

١.٢.١١ تجريد المقابس

يمثل المقبس نقطة نهاية اتصال تُدار من قبل النواة، ويُعرَّف بواسطة:

- عائلة العناوين
- نوع المقبس
- البروتوكول

٢.٢.١١ نظرة عامة على واجهة المقابس

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
```

تشمل العمليات الأساسية:

١. socket
٢. bind
٣. listen

٤. accept

٥. connect

٦. recv / send

٧. close

٣.٢.١١ مثال عملي: خادم TCP بسيط

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
addr.sin_addr.s_addr = INADDR_ANY;

bind(sockfd, (struct sockaddr *)&addr, sizeof addr);
listen(sockfd, 5);

int client = accept(sockfd, NULL, NULL);

char buf[1024];
ssize_t n = recv(client, buf, sizeof buf - 1, 0);
buf[n] = '\0';

send(client, "Hello from server", 17, 0);

close(client);
close(sockfd);
```

٤.٢.١١ الخلاصة

تكشف برمجة المقابس التفاعل المباشر مع مكدس الشبكة في نظام التشغيل. ورغم أن الواجهة منخفضة المستوى وصريحة، فإنها تمنح تحكمًا غير مسبوق في الأداء، وإدارة الموارد، وسلوك الاتصال.

يتطلب إتقان الشبكات في C23 فهمًا دقيقًا لتفاعل النواة، ودلالات الحجب، وملكية البيانات، ومعالجة الأخطاء. وتُعد هذه المهارات ضرورية لبناء الخوادم، والأنظمة الموزعة، والبنية التحتية عالية الأداء.

٣.١١ معالجة الإشارات

تُعد معالجة الإشارات (Signal Handling) إحدى الركائز الأساسية في برمجة الأنظمة، إذ تتيح للعملية الاستجابة لأحداث غير متزامنة تُسلَّم من قبل نظام التشغيل. تعمل الإشارات كمقاطعات برمجية (Software Interrupts) يمكنها تعليق مسار التنفيذ الطبيعي ونقل التحكم إلى دالة معالجة مخصصة.
وعلى عكس استدعاءات الدوال أو الاستثناءات، يمكن أن تصل الإشارات في أي لحظة تقريبًا أثناء تنفيذ البرنامج، مما يجعلها من أكثر آليات التحكم دقةً وخطورةً في لغة C. يقدّم هذا القسم نموذجًا دقيقًا ومنضبطًا لمعالجة الإشارات في C23، مرتكزًا على سلوك نظام التشغيل ودلالات POSIX.

١.٣.١١ ما هي الإشارات؟

تمثل الإشارة إخطارًا ترسله النواة إلى عملية للدلالة على حدوث حدث معين. وقد تُولَّد الإشارات من:

- نظام التشغيل (أعطال عتادية، مؤقتات)
- عمليات أخرى
- العملية نفسها

وتشمل الفئات الشائعة:

١. طلبات الإنهاء: SIGTERM، SIGKILL
٢. الأعطال والأخطاء: SIGSEGV، SIGFPE، SIGILL
٣. تفاعل المستخدم: SIGINT، SIGTSTP
٤. إشارات معرفة من المستخدم: SIGUSR1، SIGUSR2

تقطع الإشارات تدفق التنفيذ الطبيعي، ويجب التعامل معها بحذر بالغ.

٢.٣.١١ دعم الإشارات في C23

تعرف ISO C23 دعمًا أساسيًا للإشارات من خلال الترويسة <signal.h>. أما على أنظمة POSIX، فيُمدَّد هذا الدعم بواجهات أكثر تطورًا مثل sigaction و sigprocmask.
تُسلَّم الإشارات بشكل غير متزامن، وقد تقاطع التنفيذ داخل دوال المكتبة، أو نداءات النظام، أو شيفرة المستخدم.

٣.٣.١١ الإشارات الشائعة

الإشارة	المعنى
SIGINT	مقاطعة من الطرفية (Ctrl+C)
SIGTERM	طلب إنهاء متدرج
SIGKILL	إنهاء قسري (لا يمكن معالجته)
SIGSEGV	وصول غير صالح للذاكرة
SIGFPE	خطأ حسابي
SIGALRM	انتهاء مؤقت
SIGUSR1	إشارة معرفة من المستخدم
SIGUSR2	إشارة معرفة من المستخدم

٤.٣.١١ المعالجة الأساسية باستخدام signal

تُستخدم الدالة signal لتثبيت معالج لإشارة معينة. ويُستدعى المعالج بشكل غير متزامن عند تسليم الإشارة.

```
#include <signal.h>
#include <unistd.h>

volatile sig_atomic_t interrupted = 0;
```

```c
void handler(int sig) {
    interrupted = 1;
}

int main(void) {
    signal(SIGINT, handler);

    while (!interrupted) {
        /* work */
    }

    _exit(0);
}
```

ملاحظات مهمة:

- يُسمح فقط بتعديل كائنات sig_atomic_t بأمان
- معظم دوال المكتبة غير آمنة داخل معالجات الإشارات
- واجهة signal تعاني من مشاكل تاريخية في القابلية للنقل

٥.٣.١١ لماذا signal غير كافية؟

توفر واجهة signal ضمانات محدودة، وقد يختلف سلوكها بين الأنظمة. كما أنها لا تتيح:

- تحكمًا دقيقًا في إخفاء الإشارات
- سلوكًا موثوقًا لإعادة تشغيل النداءات
- الوصول إلى بيانات وصفية للإشارة

لبرمجة أنظمة احترافية، يجب استخدام sigaction.

٦.٣.١١ المعالجة المتقدمة باستخدام sigaction

توفر واجهة sigaction تحكمًا صريحًا في تسليم الإشارات، وإخفائها، وسلوك المعالج.

```c
#include <signal.h>
#include <string.h>
#include <unistd.h>

void handler(int sig, siginfo_t *info, void *ucontext) {
    (void)ucontext;
    write(STDOUT_FILENO, "SIGINT received\n", 16);
}

int main(void) {
    struct sigaction sa;
    memset(&sa, 0, sizeof sa);

    sa.sa_sigaction = handler;
    sa.sa_flags = SA_SIGINFO;

    sigaction(SIGINT, &sa, NULL);

    while (1) pause();
}
```

ملاحظات:

- الدالة write آمنة داخل معالج الإشارة، بينما printf ليست كذلك
- siginfo_t توفر معلومات عن مصدر الإشارة
- pause تعلّق التنفيذ حتى وصول إشارة

١١.٣.٧ الدوال الآمنة داخل معالجات الإشارات

يُسمح باستدعاء عدد محدود فقط من الدوال داخل معالج الإشارة، منها:

- write
- _exit
- signal
- عمليات sig_atomic_t

استدعاء أي دالة أخرى يؤدي إلى سلوك غير معرّف.

١١.٣.٨ إرسال الإشارات

يمكن إرسال الإشارات برمجيًا:

```
#include <signal.h>

raise(SIGUSR1);
```

أو بين العمليات:

```
kill(pid, SIGTERM);
```

١١.٣.٩ إخفاء الإشارات والمناطق الحرجة

يمكن حجب الإشارات مؤقتًا لحماية مناطق حرجة.

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGINT);

sigprocmask(SIG_BLOCK, &set, NULL);
/* critical section */
sigprocmask(SIG_UNBLOCK, &set, NULL);
```

لا تُفقد الإشارات عند الحجب، بل يؤجَّل تسليمها حتى رفع الحجب.

١٠.٣.١١ التعامل مع إشارات متعددة

```
void handler(int sig) {
    if (sig == SIGINT)  write(1, "INT\n", 4);
    if (sig == SIGTERM) write(1, "TERM\n", 5);
}

int main(void) {
    signal(SIGINT, handler);
    signal(SIGTERM, handler);
    while (1) pause();
}
```

١١.٣.١١ قواعد تصميم معالجة الإشارات

١. التعامل مع معالجات الإشارات كسياق مقاطعة

٢. عدم تخصيص ذاكرة داخل المعالج

٣. عدم استدعاء دوال غير آمنة

٤. استخدام أعلام للتواصل مع المنطق الرئيسي

٥. تفضيل sigaction على signal

١٢.٣.١١ الخلاصة

توفر الإشارات آلية تحكم غير متزامنة منخفضة المستوى ومدمجة بعمق مع نظام التشغيل. وهي قوية لكنها خطرة، وغالبًا ما يؤدي سوء استخدامها إلى سباقات، أو حالات توقف، أو سلوك غير معرّف.

تتطلب المعالجة الصحيحة للإشارات في C23 انضباطًا صارمًا، وبساطة متعمدة، وفصلاً واضحًا بين منطق المقاطعة والتنفيذ الطبيعي. وعند استخدامها بشكل سليم، تمكّن الإشارات من بناء برمجيات أنظمة سريعة الاستجابة مثل الخدمات، والخوادم، والمفسرات، وبيئات التشغيل.

٤.١١ التواصل بين العمليات (IPC)

يشير التواصل بين العمليات (Inter-Process Communication -- IPC) إلى الآليات التي يوفرها نظام التشغيل لتمكين العمليات المستقلة من تبادل البيانات وتنسيق التنفيذ. وبما أن العمليات لا تتشارك فضاء العناوين افتراضيًا، فإن IPC هو الوسيلة الوحيدة الآمنة لبناء أنظمة متعددة العمليات.
يُعد IPC أساسًا لأنظمة التشغيل الحديثة، ويقف خلف المفسرات، والخوادم، وقواعد البيانات، والمتصفحات، والأنظمة الموزعة. يعرض هذا القسم آليات IPC كما تُستخدم في برامج C23 على أنظمة POSIX، مع التركيز على الصحة، والأداء، ومفاضلات التصميم.

١.٤.١١ ما هو IPC؟

يمكّن IPC العمليات من:

- تبادل البيانات
- إرسال الإشعارات
- تنسيق التنفيذ
- مشاركة الموارد بأمان

وعلى عكس الخيوط، تكون العمليات معزولة بالكامل، وأي تواصل بينها يمر عبر النواة.

٢.٤.١١ نظرة عامة على آليات IPC

تشمل الآليات الشائعة:

١. الأنابيب (Pipes)
٢. الأنابيب المسماة (FIFOs)
٣. طوابير الرسائل

٤. الذاكرة المشتركة

٥. المقابس

٦. الإشارات

تمثل كل آلية موازنة مختلفة بين الأداء، والتعقيد، ونطاق الاستخدام.

٣.٤.١١ الأنابيب

توفر الأنابيب تيار بايتات أحادي الاتجاه بين عمليات مرتبطة عادةً بعلاقة أب-ابن.

إنشاء واستخدام الأنابيب

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

int main(void) {
    int pipefd[2];
    pipe(pipefd);

    pid_t pid = fork();

    if (pid == 0) {
        close(pipefd[0]);
        write(pipefd[1], "Hello from child", 16);
        close(pipefd[1]);
    } else {
        close(pipefd[1]);
```

```
        char buf[32];
        read(pipefd[0], buf, sizeof buf);
        printf("Received: %s\n", buf);
        close(pipefd[0]);
    }
}
```

٤.٤.١١ الذاكرة المشتركة

تسمح الذاكرة المشتركة لعدة عمليات بربط نفس المنطقة الفيزيائية، مما يوفّر أسرع شكل من أشكال IPC.

مهم: لا توفّر الذاكرة المشتركة أي مزامنة. المزامنة الصريحة إلزامية.

```
#include <sys/shm.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int shmid = shmget(IPC_PRIVATE, 1024, IPC_CREAT | 0666);

    if (fork() == 0) {
        char *p = shmat(shmid, NULL, 0);
        strcpy(p, "Hello");
        shmdt(p);
    } else {
        char *p = shmat(shmid, NULL, 0);
        printf("Received: %s\n", p);
        shmdt(p);
```

```
        shmctl(shmid, IPC_RMID, NULL);
    }
}
```

٥.٤.١١ الخلاصة

يمثل التواصل بين العمليات العمود الفقري للأنظمة متعددة العمليات. وتعكس كل آلية IPC فلسفة مختلفة لتبادل البيانات، من تدفقات البايتات إلى الذاكرة المشتركة وتمرير الرسائل.
يتطلب إتقان IPC فهمًا عميقًا لواجهات البرمجة، وسلوك النواة، وقواعد المزامنة، وأوضاع الفشل. وعند استخدامه بشكل صحيح، يمكّن IPC من بناء معماريات قابلة للتوسع، ومعيارية، وقوية — أما عند إساءة استخدامه، فيتحول إلى مصدر للأخطاء الدقيقة والتعقيد الخفي.

الفصل ١٢: الأمن والتحسين

١.١٢ أفضل الممارسات للبرمجة الآمنة

تمثل البرمجة الآمنة (Secure Coding) انضباطًا هندسيًا يهدف إلى كتابة برمجيات تبقى صحيحة، ومتينة، ومقاومة لإساءة الاستخدام حتى في وجود مدخلات خبيثة، ومسارات تنفيذ غير متوقعة، وبيئات تشغيل عدائية.
في الأنظمة الحديثة، نادرًا ما تنشأ الإخفاقات الأمنية من خطأ واحد فقط. بل غالبًا ما تظهر كنتيجة لسلسلة من الافتراضات الصغيرة: مدخلات غير مُتحقَّق منها، ملكية غير واضحة، سلوك غير معرّف، أو صلاحيات مفرطة. وفي اللغات منخفضة المستوى مثل C23، يجب أن تكون هذه الافتراضات صريحة ومحمية عمدًا.
يقدّم هذا القسم البرمجة الآمنة بوصفها منهجًا هندسيًا منهجيًا، لا مجرد مجموعة من الحلول الترقيعية.

١.١.١٢ البرمجة الآمنة كمبدأ تصميم

تبدأ البرمجة الآمنة قبل كتابة أي سطر شيفرة. إذ تتطلب تصميم البرمجيات بحدود ثقة واضحة، وقواعد ملكية صريحة، وافتراضات دفاعية تجاه كل مدخل خارجي.
تشمل المبادئ الأساسية:

١. الدفاع متعدد الطبقات — لا يجب أن يعتمد الأمن على فحص واحد

٢. أقل قدر من الصلاحيات — تشغيل المكونات بأدنى سلطة ممكنة

٣. الفشل المغلق — يجب أن تؤدي الأخطاء إلى الرفض لا السماح

٤. حدود الثقة الصريحة — لا تفترض أبدًا أن المدخلات آمنة

٥. أنماط فشل متوقعة — السلوك غير المعرّف خلل أمني

٢.١.١٢ الثغرات الشائعة في برامج C

فهم أصناف الثغرات المتكررة أمر أساسي لمنعها بصورة منهجية.

تجاوز سعة المخازن

يحدث تجاوز السعة (Buffer Overflow) عندما تتجاوز عمليات الكتابة حدود الذاكرة المخصصة، مما يؤدي إلى إتلاف الذاكرة المجاورة وقد يسمح بالسيطرة على تدفق التنفيذ.
استراتيجيات التخفيف:

- تجنب الدوال غير المحدودة (strcpy ،gets)
- إجراء فحوص حدود صريحة
- تفعيل تحصين المترجم (-D_FORTIFY_SOURCE)
- تفضيل واجهات برمجة مدركة للطول

مثال: نسخ نصي مضبوط

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

int main(void) {
```

```
    char buf[10];
    const char *src = "Untrusted input";

    strncpy(buf, src, sizeof buf - 1);
    buf[sizeof buf - 1] = '\0';

    printf("%s\n", buf);
}
```

ملاحظة مهمة: الدالة strncpy ليست آمنة بذاتها؛ الأمان يتحقق فقط عند فرض الإنهاء الصريح وفهم المقصود من الاستخدام.

ثغرات الحقن

تنشأ ثغرات الحقن عندما تُفسَّر البيانات غير الموثوقة على أنها أوامر، أو تعليمات، أو شيفرة قابلة للتنفيذ.
استراتيجيات التخفيف:

- التحقق الصارم من المدخلات
- عدم دمج المدخلات غير الموثوقة في سياقات تنفيذية
- تفضيل الواجهات المهيكلة على استدعاء القشرة

مثال: بناء أوامر دفاعي

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

int is_safe(const char *s) {
    return strpbrk(s, ";|&$") == NULL;
```

```
}

int main(void) {
    const char *input = "hello";

    if (!is_safe(input)) {
        fprintf(stderr, "Rejected input\n");
        return 1;
    }

    printf("Safe input: %s\n", input);
}
```

تسرب الذاكرة وأخطاء العمر

تؤدي تسربات الذاكرة وأخطاء الاستخدام بعد التحرير إلى تدهور الاعتمادية، وقد تتحول إلى هجمات حجب خدمة.
استراتيجيات التخفيف:

- تحديد قواعد الملكية بوضوح
- تحرير الذاكرة مرة واحدة فقط
- ربط التخصيص والتحرير في نفس طبقة التجريد
- استخدام أدوات الفحص أثناء التطوير

```
#include <stdlib.h>

int main(void) {
```

```
    int *p = malloc(10 * sizeof *p);
    if (!p) return 1;

    free(p);
}
```

٣.١.١٢ التحقق من المدخلات وتنقيتها

يجب اعتبار كل مدخل خارجي عدائيًا افتراضيًا.

- معاملات سطر الأوامر
- متغيرات البيئة
- بيانات الشبكة
- الملفات ووسائط IPC

مثال: تحقق قائم على القائمة البيضاء

```
#include <ctype.h>

int valid_id(const char *s) {
    for (; *s; ++s)
        if (!isalnum((unsigned char)*s))
            return 0;
    return 1;
}
```

٤.١.١٢ الإعدادات الافتراضية الآمنة

لا يُحدَّد الوضع الأمني بالشيفرة فقط، بل بالسلوك الافتراضي أيضًا.

- أذونات ملفات مقيدة
- تعطيل الميزات الاختيارية
- اختيار البروتوكولات صراحة
- حالة بدء تشغيل متوقعة

مثال: أذونات ملفات مقيدة

```
#include <sys/stat.h>
#include <stdio.h>

int main(void) {
    FILE *f = fopen("secure.txt", "w");
    chmod("secure.txt", S_IRUSR | S_IWUSR);
    fprintf(f, "confidential\n");
    fclose(f);
}
```

٥.١.١٢ معالجة الأخطاء وتسريب المعلومات

رسائل الخطأ تمثل سطح هجوم.

- تجنب كشف مسارات الملفات
- تجنب إظهار عناوين الذاكرة

- فصل رسائل المستخدم عن سجلات التشخيص

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("missing.txt", "r");
    if (!f) {
        fprintf(stderr, "Operation failed\n");
        return 1;
    }
}
```

٦.١.١٢ تحصين المترجم وسلسلة الأدوات

يعتمد أمن C بشكل كبير على دعم سلسلة الأدوات.

- حراس المكدس (Stack Canaries)
- AddressSanitizer
- UndefinedBehaviorSanitizer
- حماية تدفق التحكم

يُفرض الأمن أثناء البناء كما أثناء التشغيل.

٧.١.١٢ ملخص أفضل الممارسات

١. اعتبر السلوك غير المعرّف خللاً أمنيًا

٢. تحقق من جميع المدخلات الخارجية

٣. اجعل الملكية والعمر صريحين

٤. فضّل المسارات البسيطة القابلة للتدقيق

٥. استخدم تحصين المترجم بقوة

٦. افترض أن المهاجم يفهم شيفرتك

٨.١.١٢ الخلاصة

البرمجة الآمنة في C23 لا تهدف إلى إلغاء المخاطر، بل إلى جعلها مرئية، ومضبوطة، ومحدودة. توفر اللغة القوة، لكن الانضباط هو ما يوفر الأمان.
ولا تزال البرامج المكتوبة بلغة C من بين أكثر الأنظمة أمانًا وأعلى أداءً عندما يتحمل المطور المسؤولية الكاملة عن الصحة، والحدود، وأوضاع الفشل.

٢.١٢ تقنيات تحسين الشيفرة

تحسين الشيفرة (Code Optimization) هو انضباط هندسي يهدف إلى تحسين الأداء، والكفاءة، واستخدام الموارد دون تغيير الدلالات السلوكية للبرنامج. في برمجة الأنظمة منخفضة المستوى، يؤثر التحسين مباشرةً على زمن الاستجابة، ومعدل الإنتاجية، واستهلاك الطاقة، وقابلية التوسع. في C23، لا يعني التحسين استخدام حيل ذكية، بل فهم نموذج التكلفة للمعالج، والمترجم، وهرمية الذاكرة.

١.٢.١٢ مقدمة في تحسين الشيفرة

يهدف التحسين إلى تقليل زمن التنفيذ، وحركة الذاكرة، وأعباء التزامن، مع الحفاظ الكامل على الصحة وقابلية الصيانة.
المبادئ الأساسية للتحسين الفعّال هي:

١. القياس أولاً — التحسين دون قياس مجرد تخمين

٢. تحسين ما يهم فعلاً — التركيز على المسارات الساخنة

٣. احترام المترجم — المترجمات الحديثة تنفذ العديد من التحسينات تلقائيًا

٤. الحفاظ على الوضوح — الشيفرة غير المقروءة نادرًا ما تكون سريعة على المدى الطويل

٢.٢.١٢ التحليل الإحصائي والاختبار المعياري

قبل تعديل أي شيفرة، يجب تحديد أين يُستهلك الزمن فعليًا. التحليل الإحصائي (Profiling) يكشف الاختناقات الحقيقية، بينما يثبت الاختبار المعياري (Benchmarking) جدوى التحسينات.

أدوات التحليل الإحصائي

- gprof — توزيع الزمن على مستوى الدوال

- Valgrind (Callgrind) — رسوم استدعاء وتعليمات مفصلة
- perf — عدادات المعالج، أخطاء الذاكرة المخبئية، أخطاء التنبؤ بالتفرعات

أدوات الاختبار المعياري

- time — قياس زمني إجمالي تقريبي
- أطر اختبار مخصصة — اختبارات دقيقة وقابلة للتكرار

مثال: التحليل باستخدام gprof

```c
#include <stdio.h>

void f1(void) {
    for (volatile int i = 0; i < 1000000; ++i);
}

void f2(void) {
    for (volatile int i = 0; i < 500000; ++i);
}

int main(void) {
    f1();
    f2();
}
```

سير العمل:

```
gcc -pg -O2 prog.c -o prog
./prog
gprof prog gmon.out
```

يحدد هذا الإجراء الدوال المسيطرة على زمن التنفيذ، مما يوجّه جهد التحسين بشكل عقلاني.

٣.٢.١٢ تقنيات التحسين الشائعة

تحسين الحلقات

غالبًا ما تهيمن الحلقات على زمن التنفيذ بسبب عدد التكرارات والوصول المكثف للذاكرة.
تقنيات فعّالة:

- تقليل عدد التكرارات
- تحسين محلية البيانات
- إزالة الحسابات الثابتة داخل الحلقة

ملاحظة مهمة: فكّ الحلقات يدويًا نادراً ما يكون مفيداً إلا عند توجيه ذلك بالقياس. المترجم عادةً يقوم به عندما يكون مجديًا.

مثال: نقل الكود الثابت خارج الحلقة

```
for (int i = 0; i < n; ++i) {
    int limit = n * 4;
    a[i] = i < limit ? i : limit;
}
```

يسمح هذا للمترجم برفع التعبيرات الثابتة وتبسيط تدفق التحكم.

إدراج الدوال (Inlining)

يؤدي الإدراج إلى إزالة كلفة الاستدعاء وتمكين تحسينات إضافية، لكن الإفراط فيه يزيد حجم الشيفرة ويضغط على ذاكرة التعليمات.

```
static inline int add(int a, int b) {
    return a + b;
}
```

توجيه: قم بإدراج الدوال الصغيرة متكررة الاستدعاء، ودع القرار النهائي للمترجم في أغلب الحالات.

تحسين الوصول إلى الذاكرة

في المعالجات الحديثة، الوصول إلى الذاكرة هو العامل المسيطر على الأداء.
الأهداف الأساسية:

- تعظيم المحلية المكانية والزمانية
- تقليل أخطاء الذاكرة المخبئية
- تجنب الحركة غير الضرورية للذاكرة

مثال: اجتياز صديق للذاكرة المخبئية

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        sum += a[i][j];
```

يتوافق الاجتياز الصفّي مع تخطيط الذاكرة، ويقلل أخطاء الذاكرة المخبئية.

محاذاة البيانات

يمكن أن تحسن المحاذاة الأداء، لكن يجب استخدامها بوعي.

```
struct __attribute__((aligned(16))) data {
    int a;
    double b;
};
```

تحذير: قد تحسن المحاذاة أداء SIMD والوصول الذري، لكنها قد تزيد حجم البنية واستهلاك الذاكرة.

التحسين الخوارزمي

اختيار الخوارزمية يفوق جميع التحسينات الدقيقة.

```
int binary_search(const int *a, int n, int key) {
    int l = 0, r = n - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (a[m] == key) return m;
        if (a[m] < key) l = m + 1;
        else r = m - 1;
    }
    return -1;
}
```

خفض التعقيد من $O(n)$ إلى $O(\log n)$ يتفوق على أي تحسين منخفض المستوى.

٤.٢.١٢ تحسينات المترجم

تنفذ المترجمات الحديثة تحسينات عدوانية تلقائيًا.

الخيارات الشائعة

- -O2 — توازن بين الأداء والحجم
- -O3 — تحسينات متقدمة وفكّ الحلقات
- -Os — تحسين موجه للحجم

- -march=native — استغلال خصائص المعالج الهدف

```
gcc -O2 -march=native prog.c
```

تحذير: الخيار -Ofast قد ينتهك دلالات اللغة، خصوصاً في العمليات العشرية.

١٢.٢.٥ أفضل الممارسات للتحسين

١. قم بالتحليل قبل التعديل

٢. حسن الخوارزميات قبل التفاصيل الدقيقة

٣. فضّل تحسين تخطيط البيانات

٤. ثق بالمترجم ما لم يثبت العكس

٥. أعد القياس بعد كل تغيير

١٢.٢.٦ مثال عملي: ضرب المصفوفات الواعي للذاكرة المخبئية

```
#define N 100

void matmul(int A[N][N], int B[N][N], int C[N][N]) {
    for (int i = 0; i < N; ++i)
        for (int k = 0; k < N; ++k)
            for (int j = 0; j < N; ++j)
                C[i][j] += A[i][k] * B[k][j];
}
```

إعادة ترتيب الحلقات تحسن إعادة استخدام الذاكرة وتقلل أخطاء الذاكرة المخبئية.

٧.٢.١٢ الخلاصة

التحسين في C23 لا يتعلق بالحيل الذكية، بل بفهم تكلفة العتاد، وسلوك المترجم، ومحلية الذاكرة. الصحة أولاً. القياس ثانيًا. التحسين يأتي فقط عندما يكون مبررًا. والإتقان الحقيقي لا يكمن في كتابة «شيفرة سريعة»، بل في معرفة لماذا هي سريعة.

الفصل ١٣: الميزات الجديدة والتغييرات في C23

١.١٣ نظرة عامة على الميزات الجديدة في C23

تواصل لغة البرمجة C تطوّرها بأسلوب محافظ ومدروس. يقدّم معيار C23 مجموعة من التحسينات، والتوضيحات، والإضافات المحددة بعناية، التي تهدف إلى تحسين التعبيرية، والسلامة، وقابلية التكامل، مع الحفاظ على الفلسفة الأساسية للغة: التحكم الصريح، الأداء المتوقع، والتقارب الوثيق مع نموذج العتاد.
يعرض هذا القسم أهم الإضافات والتغييرات في C23، مع التركيز على تأثيرها العملي في برمجة الأنظمة الواقعية.

١.١.١٣ مقدمة إلى C23

يبني C23 على معياري C11 وC17 من خلال معالجة تناقضات قديمة، وتوحيد ممارسات شائعة، وإضافة ميزات جديدة ذات نطاق محدود. بدلاً من إعادة تعريف اللغة، يقوم C23 بصقلها.
الأهداف الرئيسية لمعيار C23 هي:

١. تحسينات سلامة تدريجية — تقليل مصادر السلوك غير المعرّف

٢. التحديث — مواءمة C مع سلاسل الأدوات الحديثة

٣. التوضيح والاتساق — إزالة الغموض والسلوكيات غير المحددة

٤. الاستقرار طويل الأمد — التطور دون كسر القواعد البرمجية القائمة

تظل C23 لغة تتحقق صحتها بالانضباط، لا بالأتمتة.

٣.١.٣ الإضافات والتحسينات الرئيسية في C23

السمات المعيارية **(Standardized Attributes)**

يعتمد C23 رسميًا صيغة السمات ذات الأقواس المزدوجة، بما يتوافق مع C++ ويحسّن قابلية نقل التوصيفات.

```
[[nodiscard]] int compute_value(void) {
    return 42;
}
```

الغاية التصميمية:

- التعبير عن نية المبرمج للمترجم
- تمكين تشخيصات أقوى دون تغيير الدلالات
- تعزيز الاتساق بين اللغات

السمات لا تفرض سلوكًا، بل تمكّن التحليل.

دعم الأعداد الموسّعة: _BitInt

يوحّد C23 النوع _BitInt(N) لتمثيل أعداد صحيحة بعرض بتات محدد بدقة.

```
_BitInt(128) counter;
```

حالات الاستخدام:

- التشفير والحسابات ذات الدقة العالية
- التحكم الدقيق في ABI
- نمذجة سجلات العتاد

تزيد هذه الميزة التعبيرية دون فرض كلفة تشغيلية ضمنية.

تحسين تمثيل الأخطاء

يشجّع C23 على الإبلاغ الصريح عن الأخطاء باستخدام أنواع إرجاع محددة ورموز أخطاء معيارية.

```
#include <errno.h>

errno_t safe_divide(int a, int b, int *out) {
    if (!out || b == 0)
        return EINVAL;
    *out = a / b;
    return 0;
}
```

النية التصميمية:

- فصل تدفق البيانات عن الإبلاغ عن الخطأ
- تحسين التحليل الساكن
- تجنب القيم الدلالية الغامضة

لا يستبدل هذا الأنماط التقليدية، بل يضفي عليها طابعًا أوضح وأكثر أمانًا.

إدارة ذاكرة أكثر أمانًا: reallocarray

يوحّد C23 الدالة reallocarray التي تحمي من فيض الضرب أثناء حساب حجم الذاكرة.

```
int *p = reallocarray(NULL, count, sizeof *p);
```

الفائدة:

- منع فيض الضرب الصامت
- جعل نية التخصيص صريحة
- تحسين قابلية التدقيق والتحليل

هذا تحسين في الصحة، لا في الأداء.

تحسينات المكتبة القياسية

تم توحيد عدة دوال كانت مستخدمة على نطاق واسع لكنها غير معيارية سابقًا.

```
strlcpy(dst, src, sizeof dst);
strlcat(dst, suffix, sizeof dst);
```

المنطق التصميمي:

- الوعي الصريح بحجم المخزن
- دلالات اقتطاع متوقعة
- تقليل الاعتماد على السلوك غير المعرّف

تحسن هذه الدوال المتانة، لكنها لا تعفي من الانضباط.

توضيح التخزين المحلي للخيوط

يوضّح C23 دلالات التخزين المحلي للخيوط باستخدام thread_local.

```
thread_local int tls_counter;
```

الخصائص:

- نسخة مستقلة لكل خيط
- عمر ثابت أو تلقائي
- دون تزامن ضمني

يسهّل هذا عزل الحالة، لا التحكم في التزامن.

تحسينات البرمجة العامة

تظل _Generic الأساس للماكروات العامة للأنواع، مع اندماج أفضل في الأنماط الحديثة.

```
#define print(x) _Generic((x), \
    int: print_int, \
    double: print_double)(x)
```

يوفّر هذا اختياراً في وقت الترجمة دون كلفة تشغيلية.

٣.١.١٣ الاستخدام المسؤول لميزات C23

ميزات C23 أدوات، وليست ضمانات. الاستخدام الفعّال يتطلب انضباطًا.

١. إدخال ميزات C23 تدريجيًا

٢. تفضيل الوضوح على الحداثة

٣. استخدام ميزات السلامة للمساعدة لا للاستبدال

٤. الحفاظ على قابلية النقل

٥. اعتبار المعيار عقداً لا شبكة أمان

٤.١.١٣ الخلاصة

يمثّل C23 تحسينًا دقيقًا للغة C لا إعادة تصميم جذرية. يعزز الدقة، والتعبيرية، والصحة المدعومة بالأدوات، مع الحفاظ على قابلية التنبؤ، والشفافية، والتحكم.
فهم ما يضيفه C23 — وما يتعمد عدم إضافته — يسمح بكتابة شيفرة C حديثة فعّالة، قابلة للنقل، ومناسبة للأنظمة طويلة العمر.

٢.١٣ التغييرات في المكتبة القياسية

يقدّم معيار C23 مجموعة مركّزة من التغييرات على المكتبة القياسية. بدل التوسع الكبير، يركّز C23 على توحيد الممارسة الشائعة، وتحسين الصحة، وزيادة الأمان تدريجيًا، مع الحفاظ على الطبيعة منخفضة المستوى للغة.

١.٢.١٣ تطور المكتبة في C23

تطورت المكتبة القياسية تاريخيًا ببطء وحذر. كثير من الدوال ظهرت خارج المعيار قبل اعتمادها رسميًا بعد سنوات من الاستخدام العملي.
الأهداف الرئيسية لتغييرات المكتبة في C23:

١. واجهات أكثر أمانًا
٢. توحيد رسمي للممارسات الشائعة
٣. توضيح السلوك
٤. الحفاظ على قابلية النقل

يحسّن C23 الصحة بالمواصفة، لا بالتجريد.

٢.٢.١٣ الدوال المعيارية الجديدة

strlcpy و strlcat

يوحّد C23 الدالتين المعروفتين في أنظمة BSD كبدائل أكثر أمانًا.

```
char dest[20];
strlcpy(dest, "Hello", sizeof dest);
strlcat(dest, " World", sizeof dest);
```

الخصائص:

- إنهاء صفري مضمون (عند الحجم > 0)
- إرجاع الطول الكامل المتوقع
- كشف الاقتطاع

reallocarray

اعتماد رسمي لدالة تحقق الضرب قبل التخصيص.

```
int *p = reallocarray(NULL, count, sizeof *p);
```

تعالج هذه الدالة فئة حقيقية من ثغرات الأمان التاريخية.

٣.٢.١٣ واجهات موضّحة ومحسّنة

إنشاء الملفات الحصري

يوضّح C23 دلالة الوضع "x" في fopen.

```
FILE *f = fopen("example.txt", "wx");
```

يمنع هذا حالات السباق ويضمن عدم الاستبدال الضمني.

توضيح الإبلاغ عن الأخطاء في تعدد الخيوط

يحسّن C23 وضوح مواصفات دوال مثل strerror.

١٣.٢.٤ واجهات أُزيلت أو أصبحت متقادمة

إزالة gets

إزالة كاملة ونهائية لدالة لا يمكن استخدامها بأمان.

تقادم tmpnam

تأكيد تقادمها بسبب حالات السباق والتنبؤ بالأسماء.

١٣.٢.٥ ما لا يفعله C23

- لا إدارة ذاكرة تلقائية
- لا حاويات مع فحص حدود
- لا استثناءات
- لا إدارة عمر تلقائية

تظل C23 صارمة، وصريحة، ولا ترحم.

١٣.٢.٦ أفضل الممارسات

١. استخدم الواجهات الأحدث عند الحاجة
٢. اعتبر كل استدعاء قابلاً للفشل
٣. لا تفترض أمان الخيوط
٤. تجنب الواجهات المتقادمة
٥. اجمع المكتبة مع تصميم منضبط

٧.٢.١٣ الخلاصة

تتطور مكتبة C23 بالتحسين الدقيق لا التوسع. ومن خلال توحيد الواجهات المجربة، وتوضيح السلوك، وإزالة الإرث الخطر، يعزّز C23 لغة C دون المساس بهويتها الأساسية.

فهم هذه التغييرات يمكّنك من كتابة شيفرة C حديثة أكثر أمانًا، أوضح، وقابلة للنقل، مع البقاء في قلب برمجة الأنظمة.

٣.١٣ التوافق مع الإصدارات السابقة وترقية الشيفرة

يُعدّ التوافق مع الإصدارات السابقة (Backward Compatibility) إحدى السمات الجوهرية التي تميّز لغة C. وعلى عكس كثير من اللغات الحديثة، تتطور C ضمن قيود صارمة تهدف إلى الحفاظ على قواعد برمجية قد تمتد لعقود، وتُستخدم في صناعات مختلفة، وعلى أجيال متعددة من العتاد. يواصل معيار C23 هذا النهج. فعلى الرغم من تقديمه تحسينات وقدرات جديدة، إلا أنه لا يُبطل الشيفرة الصحيحة المكتوبة وفق المعايير السابقة. يتناول هذا القسم كيفية الحفاظ على التوافق الخلفي في C23، وكيف يمكن للمطورين ترقية قواعدهم البرمجية بأسلوب مسؤول ومنضبط.

١.٣.١٣ مفهوم التوافق الخلفي في C

في سياق معيار ISO C، لا يعني التوافق الخلفي تشجيع استخدام الشيفرة القديمة، بل يعني أن الشيفرة المتوافقة يجب أن تستمر في الترجمة والتصرف كما هو محدد.
على وجه التحديد، يضمن C23 ما يلي:

- بقاء برامج C11 و C18 الصحيحة صالحة في C23
- الحفاظ على الدلالات القائمة ما لم يُعاد تعريفها صراحة
- أن الواجهات المُزالة كانت أصلاً غير قابلة للاستخدام الآمن أو ذات سلوك غير معرّف

تتطور معايير C بأسلوب محافظ عن قصد.

٢.٣.١٣ الميزات المُزالة مقابل الميزات المتقادمة

من الضروري التمييز بين الميزات المُزالة والميزات المتقادمة.

الواجهات المُزالة

بعض الواجهات أُزيلت قبل C23 ولم تعد جزءاً من اللغة.

- gets — أُزيلت بسبب انعدام الأمان الجوهري
- التصريحات الضمنية للدوال — أُزيلت لفرض سلامة الأنواع

الشيفرة التي تعتمد عليها كانت غير متوافقة حتى قبل C23.

الواجهات المتقادمة

يعزّز C23 التحذير من استخدام واجهات معروفة بعدم أمانها أو كثرة أخطائها.

- tmpnam — توليد غير آمن لأسماء الملفات المؤقتة
- واجهات سلاسل نصية قديمة دون فحص حدود

التقادم إشارة نية، لا إزالة فورية.

٣.٣.١٣ استبدال الواجهات التراثية غير الآمنة

استبدال gets

```
fgets(buffer, sizeof buffer, stdin);
```

كان هذا الاستبدال إلزاميًا قبل C23 بوقت طويل.

استبدال tmpnam

```
char template[] = "/tmp/fileXXXXXX";
int fd = mkstemp(template);
```

توضيح مهم:

- mkstemp دالة POSIX وليست جزءاً من ISO C

- لا يوفّر ISO C واجهة معيارية لإنشاء ملفات مؤقتة آمنة
- قابلية النقل قد تتطلب طبقة تجريد خاصة بالمنصة

لا يُدخل C23 سلوك POSIX إلى معيار ISO C.

٤.٣.١٣ دمج ميزات C23 بشكل آمن

ميزات C23 تحسينات اختيارية، وليست ترقية إجبارية.

استراتيجية التبني التدريجي

يُفضّل إدخال الميزات الجديدة بشكل محلي وتدريجي.

```
[[nodiscard]] int compute_value(void) {
    return 42;
}
```

تحسّن السمات التشخيص دون تغيير الدلالات.

الاستخدام الصحيح لاكتشاف الميزات

لا تعتمد فقط على __STDC_VERSION__ لاكتشاف توفر الميزات.

```
#if defined(__has_c_attribute)
#  if __has_c_attribute(nodiscard)
#    define NODISCARD [[nodiscard]]
#  endif
#endif

#ifndef NODISCARD
```

```
#  define NODISCARD
#endif

NODISCARD int compute_value(void) {
    return 42;
}
```

أهمية ذلك:

- دعم المترجم لا يعني الالتزام الكامل بالمعيار
- سلاسل الأدوات تتأخر غالبًا عن المعايير
- اكتشاف الميزات أدق من تخمين الإصدار

٥.٣.١٣ أوضاع المترجم وإعدادات البناء

الانتقال إلى C23 قرار على مستوى نظام البناء، لا فرضًا على الشيفرة المصدرية.

تحديد المعيار صراحة

```
gcc -std=c23 -Wall -Wextra -Wpedantic
```

يُفعّل هذا تشخيصات قوية دون إجبار تعديل الشيفرة.

دعم أكثر من معيار

تدعم كثير من المشاريع عدة معايير في آن واحد.

- C11 للأنظمة القديمة
- C23 للأدوات الحديثة

- شيفرة مشتركة عبر حجب الميزات

هذا نمط طبيعي ومتوقع في مشاريع C.

٦.٣.١٣ مثال عملي على الترقية

شيفرة تراثية

```
char buf[100];
gets(buf);
char *name = tmpnam(NULL);
```

هذه الشيفرة غير متوافقة أصلاً.

شيفرة مُحدّثة

```
if (fgets(buf, sizeof buf, stdin)) {
    /* use input */
}
```

يجب أن تكون إدارة الملفات المؤقتة معتمدة على المنصة أو مجرّدة.

٧.٣.١٣ أفضل الممارسات للتوافق طويل الأمد

١. اعتبر المعيار عقداً، لا موضة

٢. استبدل الواجهات غير الآمنة بغض النظر عن الإصدار

٣. استخدم اكتشاف الميزات لا تخمين الإصدارات

٤. افصل قضايا قابلية النقل عن تطور اللغة

٥. حدّث سلاسل الأدوات قبل إعادة كتابة الشيفرة

٨.٣.١٣ الخلاصة

لا يكسر C23 لغة C — بل يعزّزها.
يبقى التوافق الخلفي قيداً تصميميًا أساسيًا، يضمن استمرار الأنظمة القائمة، مع السماح بتحديث تدريجي ومدروس. الترقية إلى C23 ليست إعادة كتابة، بل تحسينًا للتشخيص، وتوضيحًا للنية، وتبنّيًا لأنماط أكثر أمانًا حيثما كان ذلك مناسبًا.
الشيفرة الجيدة في C لا تشيخ — بل تتراكم فيها الخبرة والانضباط.

الفصل ١٤: التطبيقات العملية ودراسات الحالة

١.١٤ بناء نواة نظام تشغيل بسيطة

يُعدّ بناء نواة نظام تشغيل (Operating System Kernel) من أكثر التمارين تطلبًا وإثراءً في البرمجة منخفضة المستوى. إذ يتطلب فهمًا دقيقًا لأوضاع تنفيذ المعالج، وتخطيط الذاكرة، واتفاقيات الاستدعاء، وواجهات العتاد، وسلوك سلاسل الأدوات البرمجية.
يقدّم هذا القسم نواة تعليمية مصغّرة، وليست نظام تشغيل إنتاجيًا. الهدف هو الوضوح المفاهيمي: إظهار كيفية استخدام C23 بشكل صحيح في بيئة مستقلة (Freestanding Environment) مع التفاعل مع العتاد عبر شيفرة منخفضة المستوى مضبوطة بعناية.

١.١.١٤ ما هي نواة نظام التشغيل (وما ليست عليه)

نواة نظام التشغيل هي القلب ذي الامتيازات العليا للنظام، وهي المسؤولة عن التحكم في الوصول إلى العتاد وتوفير التجريدات الأساسية.
في أبسط صورها، تتولى النواة المهام التالية:

- التحكم في المعالج وإدارة أوضاع التنفيذ
- تخطيط الذاكرة وحمايتها
- معالجة المقاطعات والاستثناءات

- التفاعل مع العتاد عبر برامج التشغيل

تتجاهل هذه النواة التعليمية عمدًا:

- العمليات للمستخدم
- الذاكرة الافتراضية
- أنظمة الملفات
- تعدد المهام

فالبساطة هنا خيار مقصود.

٢.١.١٤ بيئة التطوير وسلسلة الأدوات

يتطلب تطوير النواة نموذج ترجمة مستقل (Freestanding). أي أن المترجم لا يفترض وجود نظام تشغيل ولا بيئة تشغيل قياسية.

متطلبات المترجم

- دعم C المستقلة (-ffreestanding)
- تحكم صريح بمرحلة الربط
- عدم الاعتماد على libc

توضيح مهم:

- gcc-multilib لا يُنشئ مترجمًا عابرًا للمنصات
- المترجم العابر الحقيقي يستهدف معمارية مختلفة
- لأغراض تعليمية، يكفي البناء الأصلي على x86

بيئة المحاكاة

يُستخدم QEMU لمحاكاة العتاد بشكل آمن.

```
sudo apt install qemu-system-x86
```

يمنع ذلك السلوك غير المعرّف على الأجهزة الحقيقية.

٣.١.١٤ نظرة عامة على عملية الإقلاع

في أنظمة x86 التقليدية المعتمدة على BIOS:

- يبدأ المعالج في نمط 16-بت الحقيقي
- يقوم BIOS بتحميل أول قطاع (512 بايت) إلى العنوان 0x7C00
- يبدأ التنفيذ من هذا العنوان

يجب على محمّل الإقلاع:

- أن يتناسب مع 512 بايت
- تحميل النواة
- نقل التحكم إليها صراحة

٤.١.١٤ محمّل إقلاع مصغّر (x86 BIOS)

```
[BITS 16]
[ORG 0x7C00]

start:
```

```
    cli
    xor ax, ax
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov sp, 0x7C00

    mov ah, 0x02
    mov al, 1
    mov ch, 0
    mov cl, 2
    mov dh, 0
    mov bx, 0x1000
    int 0x13

    jmp 0x1000:0x0000

times 510-($-$$) db 0
dw 0xAA55
```

ملاحظات:

- تُستخدم خدمات BIOS للقرص
- تُحمَّل النواة في عنوان ثابت
- لا توجد معالجة أخطاء (تبسيط تعليمي)

٥.١.١٤ شيفرة النواة باستخدام C23 المستقلة

لا يغيّر C23 قواعد برمجة النواة، بل يُحسّن الوضوح والتشخيص.

القيود الأساسية

- لا مكتبة قياسية
- لا كومة (Heap) إلا إذا نُفذت يدويًا
- لا تسامح مع السلوك غير المعرّف

نواة مصغّرة

```c
#include <stdint.h>

static volatile uint16_t *const vga =
    (uint16_t *)0xB8000;

void clear_screen(void) {
    for (int i = 0; i < 80 * 25; ++i)
        vga[i] = 0x0F00 | ' ';
}

void print(const char *s) {
    while (*s)
        *vga++ = 0x0F00 | *s++;
}

void kernel_main(void) {
    clear_screen();
    print("Hello from C23 kernel");
    for (;;);
}
```

٦.١.١٤ الربط وبناء صورة النظام

يجب ربط النواة ووضعها في عناوين محددة بدقة.

```
gcc -ffreestanding -c kernel.c -o kernel.o
ld -Ttext 0x1000 --oformat binary kernel.o -o kernel.bin
cat boot.bin kernel.bin > os.img
```

أهمية ذلك:

- لا يوجد محمّل ELF
- تخطيط الذاكرة يجب أن يكون دقيقًا
- نقطة الدخول تُحدَّد يدويًا

٧.١.١٤ الاختبار عبر المحاكاة

```
qemu-system-x86_64 -drive format=raw,file=os.img
```

يتيح ذلك تنفيذ النواة في بيئة معزولة وآمنة.

٨.١.١٤ التوسّع التدريجي للنواة

بعد استقرار التنفيذ، يمكن إضافة الأنظمة الفرعية بشكل تدريجي.

جدول واصفات المقاطعات

```
struct idt_entry {
    uint16_t offset_low;
    uint16_t selector;
    uint8_t zero;
```

```
    uint8_t flags;
    uint16_t offset_high;
} __attribute__((packed));
```

معالجة المقاطعات مرتبطة بالمعمارية وتتطلب حذرًا شديدًا.

كومة مصغّرة (Bump Allocator)

```
static uint8_t heap[1024 * 1024];
static size_t used;

void *kmalloc(size_t n) {
    if (used + n > sizeof heap) return 0;
    void *p = &heap[used];
    used += n;
    return p;
}
```

هذه الكومة محدودة عمدًا ولا تدعم التحرير.

٩.١.١٤ أفضل الممارسات للنوى التعليمية

١. لا تفترض شيئًا عن البيئة

٢. اعتبر السلوك غير المعرّف خطأً قاتلاً

٣. افصل الشيفرة الخاصة بالمعمارية

٤. قدّم الوضوح على التحسين

٥. تحقّق من كل خطوة قبل إضافة ميزات

١٠.١.١٤ الخلاصة

كتابة نواة نظام تشغيل مصغّرة لا تهدف إلى بناء نظام كامل، بل إلى فهم الحد الفاصل بين البرمجيات والعتاد.

لا تجعل C23 تطوير النواة أسهل أو أصعب، بل تجعله أوضح. ومن خلال الجمع بين C مستقلة منضبطة وشيفرة تجميع مضبوطة بعناية، يكتسب المطوّر فهمًا دقيقًا لكيفية عمل الأنظمة فعليًا. تؤسس هذه الدراسة قاعدة صلبة يمكن البناء عليها لاحقًا لنوى أكثر تقدمًا: الترقيم الصفحي، وتعدد المهام، وتعدد المعالجات بثقة وانضباط.

٢.١٤ تصميم مترجم أساسي

يُعدّ تصميم المترجم (Compiler) من أكثر المهام تعقيداً من الناحية الفكرية في برمجة الأنظمة. فالمترجم يحوّل الشيفرة المصدرية القابلة للقراءة البشرية إلى تمثيل منخفض المستوى قابل للتنفيذ على الآلة، مع الحفاظ على معنى البرنامج وتطبيق قواعد اللغة بدقة.
يقدّم هذا القسم مترجماً تعليمياً مبسّطاً عن قصد. هدفه ليس بناء مترجم إنتاجي عالي التحسين، بل شرح بنية المترجم وتدفّق البيانات داخله بوضوح.

١.٢.١٤ ما الذي يفعله المترجم فعلياً؟

المترجم ليس تحويلاً واحداً، بل سلسلة مراحل (Pipeline) محددة جيداً. كل مرحلة تستهلك تمثيلاً منظماً وتنتج تمثيلاً أكثر تقييداً.
يتكوّن خط الأنابيب التقليدي للمترجم من:

١. التحليل المعجمي — النص ← رموز (Tokens)

٢. التحليل النحوي — الرموز ← شجرة بنية مجردة (AST)

٣. التحليل الدلالي — AST ← AST مشروحة

٤. التمثيل الوسيط (IR) — AST ← IR

٥. توليد الشيفرة — IR ← شيفرة الهدف

٦. التحسين — تحويلات اختيارية على IR أو الشيفرة الهدف

خلاصة أساسية:

- المترجمات الحقيقية لا تولّد الشيفرة الآلية مباشرة من النص
- التمثيلات الوسيطة ضرورية
- يجب أن تكون كل مرحلة صحيحة قبل التفكير في التحسين

٢.٢.١٤ بيئة التطوير

لا يتطلب تطوير المترجمات عتاداً خاصًا، لكنه يتطلب انضباطًا عاليًا ووضوحًا في الأدوات.

سلسلة الأدوات

يكفي وجود مترجم يدعم C23. تحسّن C23 التشخيصات ووضوح الأنواع، لكنها لا تغيّر نظرية المترجمات.

```
sudo apt install gcc
```

مولدات المحللات (اختياري)

أدوات مثل Flex و Bison هي أدوات مساعدة وليست متطلبات أساسية.

- تُؤتمت بناء المحللات المعجمية والنحوية
- لا تفرض بنية المترجم
- يمكن كتابة محللات يدوية بالكامل

تُستخدم هنا لأغراض تعليمية فقط.

٣.٢.١٤ التحليل المعجمي

يحوّل التحليل المعجمي سلسلة المحارف إلى سلسلة من الرموز (Tokens). والرموز هي أصغر الوحدات ذات المعنى في اللغة.
أمثلة على الرموز:

- الكلمات المحجوزة

- المعرّفات
- القيم الثابتة
- العوامل

وصف التحليل المعجمي (Flex)

```
%{
#include <stdio.h>
%}

%%

"int"       { printf("KEYWORD(int)\n"); }
"return"    { printf("KEYWORD(return)\n"); }
[a-zA-Z_][a-zA-Z0-9_]* { printf("IDENT(%s)\n", yytext); }
[0-9]+      { printf("NUMBER(%s)\n", yytext); }
[ \t\n]+    ;
.           { printf("UNKNOWN(%s)\n", yytext); }

%%

int main(void) {
    yylex();
    return 0;
}
```

مهم:

- المحلل المعجمي لا يقوم بالتحليل النحوي

- يتعرف على الأنماط لا على البنية
- معالجة الأخطاء في هذه المرحلة يجب أن تكون محدودة

٤.٢.١٤ التحليل النحوي

يتحقق التحليل النحوي من أن تسلسل الرموز يتبع القواعد النحوية للغة. ناتج هذه المرحلة هو شجرة بنية مجردة ،(AST) وليست شيفرة تنفيذية.

تعريف القواعد (Bison)

```
%{
#include <stdio.h>
%}

%token INT RETURN IDENT NUMBER

%%

program:
    statement
    ;

statement:
    declaration
  | return_stmt
    ;

declaration:
    INT IDENT ';'
```

```
    ;

return_stmt:
    RETURN NUMBER ';'
    ;

%%

int main(void) {
    yyparse();
    return 0;
}
```

تمييز أساسي:

- التحليل النحوي يحدّد البنية
- لا يتحقق من الأنواع
- لا يولّد شيفرة

٥.٢.١٤ التحليل الدلالي

يفرض التحليل الدلالي قواعد اللغة التي لا يمكن للنحو وحده التعبير عنها.
تشمل مسؤولياته عادةً:

- التحقق من الأنواع
- إدارة النطاقات
- ربط المعرّفات
- ضمان التصريح قبل الاستخدام

مثال بسيط على جدول الرموز

```c
#include <string.h>
#include <stdio.h>

typedef struct {
    const char *name;
    int type;
} Symbol;

static Symbol table[64];
static int count;

void add_symbol(const char *name, int type) {
    table[count++] = (Symbol){ name, type };
}

int lookup(const char *name) {
    for (int i = 0; i < count; ++i)
        if (strcmp(table[i].name, name) == 0)
            return table[i].type;
    return -1;
}
```

ملاحظة:

- يعمل التحليل الدلالي على AST
- الأخطاء هنا أخطاء لغوية وليست أخطاء نحوية

١٤.٢.٦ التمثيل الوسيط

تُدخل المترجمات الحديثة تمثيلاً وسيطًا (IR) بين AST والشيفرة الآلية.
فوائد IR:

- استقلالية عن المعمارية
- تسهيل التحسين
- تبسيط توليد الشيفرة

مثال: شيفرة ثلاثية العناوين

```
t1 = 5
t2 = 10
t3 = t1 + t2
```

هذا الشكل أسهل للتحليل والتحويل من أشجار الصياغة المجردة.

١٤.٢.٧ توليد الشيفرة

يحوّل توليد الشيفرة الـ IR إلى شيفرة خاصة بالهدف. لأغراض تعليمية، يكفي توليد شيفرة تجميع بسيطة.

```c
#include <stdio.h>

void emit_add(const char *dst, const char *a, const char *b) {
    printf("mov eax, %s\n", a);
    printf("add eax, %s\n", b);
    printf("mov %s, eax\n", dst);
}
```

توضيح مهم:

- هذا إخراج توضيحي فقط
- المترجمات الحقيقية يجب أن تلتزم باتفاقيات الاستدعاء وواجهات ABI

٨.٢.١٤ التحسين

يقوم التحسين بتحويل الـ IR مع الحفاظ على الدلالات.

مثال: طيّ الثوابت

```
int fold_add(int a, int b) {
    return a + b;
}
```

يمكن للمترجم حساب هذه القيمة زمن الترجمة إذا كانت المعطيات معروفة.
قاعدة أساسية:

- يجب ألا يغيّر التحسين السلوك المرصود
- الصحة تسبق الأداء

٩.٢.١٤ أفضل الممارسات للمترجمات التعليمية

١. الفصل الصارم بين مراحل المترجم

٢. بناء AST قبل توليد الشيفرة

٣. إدخال IR مبكرًا

٤. رفض البرامج غير الصحيحة بوضوح

٥. عدم التحسين قبل إثبات الصحة

١٠.٢.١٤ الخلاصة

تصميم المترجم ليس مسألة أدوات، بل مسألة تمثيل وتحويل. توفر C23 لغة واضحة ومعبرة لبناء بنية المترجم، لكنها لا تغيّر نظرية المترجمات.

من خلال بناء مترجم صغير ومنظّم جيدًا، يكتسب المطوّر فهمًا عميقًا للغات البرمجة، ونماذج التنفيذ، والكلفة الحقيقية للتجريد. وهذه المعرفة تنطبق بالتساوي على C و C++ و Rust واللغات الحديثة المعتمدة على الآلات الافتراضية.

٣.١٤ كتابة برنامج تشغيل جهاز Device Driver باستخدام C

تُعدّ كتابة برنامج تشغيل جهاز (Device Driver) من أكثر المهام تطلبًا في برمجة الأنظمة. يعمل برنامج التشغيل عند الحدّ الفاصل بين البرمجيات والعتاد، حيث يترجم تجريدات نظام التشغيل إلى عمليات فعلية على الجهاز.
يقدّم هذا القسم مثالاً تعليميًا لبرنامج تشغيل جهاز محارف في لينكس (Linux Character Device Driver). وعلى الرغم من أن التنفيذ يعتمد على واجهات نواة لينكس، فإن المبادئ المعمارية تنطبق على برامج التشغيل في جميع أنظمة التشغيل الحديثة.

١.٣.١٤ ما هو برنامج تشغيل الجهاز فعليًا؟

برنامج تشغيل الجهاز هو شيفرة تعمل داخل النواة تتوسط الوصول إلى جهاز عتادي. وعلى عكس برامج فضاء المستخدم، فإن برامج التشغيل:

- تعمل بامتيازات مرتفعة
- يجب ألا تتعطّل أو تحجب التنفيذ بشكل غير صحيح
- لا يمكنها الاعتماد على مكتبة C القياسية
- تتفاعل مباشرة مع العتاد أو البرمجيات الثابتة

تشمل مسؤولياتها الأساسية:

١. التهيئة — اكتشاف الجهاز وضبطه
٢. وساطة الإدخال/الإخراج — توفير وصول مضبوط للعتاد
٣. معالجة المقاطعات — الاستجابة للأحداث غير المتزامنة
٤. احتواء الأخطاء — منع فشل النظام ككل
٥. إدارة الطاقة ودورة الحياة

توضيح مهم:

- برامج التشغيل تعتمد على نظام التشغيل
- يسمح باستخدام صياغة C23، لكن C في النواة ليست *Hosted C*
- تستخدم نواة لينكس لهجة C مقيدة

٢.٣.١٤ بيئة التطوير

يتطلب تطوير برامج التشغيل سلسلة أدوات مدركة لبنية النواة.

سلسلة الأدوات والترويسات

```
sudo apt install build-essential linux-headers-$(uname -r)
```

- ترويسات النواة تعرّف واجهاتها الداخلية
- ترويسات فضاء المستخدم غير كافية

شيفرة النواة (اختياري)

الاطلاع على شيفرة النواة مفيد للفهم والتصحيح:

```
sudo apt install linux-source
```

٣.٣.١٤ نوع برنامج التشغيل: أجهزة المحارف

توفر أجهزة المحارف تيارات بايت غير منظمة. ومن أمثلتها:

- المنافذ التسلسلية
- الأجهزة الافتراضية
- واجهات التحكم

تُعرض هذه الأجهزة في فضاء المستخدم كملفات تحت /dev، لكن هذا مجرد تجريد — وليس تخزينًا فعليًا للملفات.

٤.٣.١٤ بنية برنامج التشغيل

يتكوّن برنامج تشغيل محارف بسيط من:

١. تسجيله لدى النواة
٢. دوال ردّ نداء لعمليات الملفات
٣. تحرير الموارد عند الإزالة

٥.٣.١٤ مثال: برنامج تشغيل محارف بسيط

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#define DEVICE_NAME "my_device"
#define BUFFER_SIZE 1024
```

```
static int major;
static char buffer[BUFFER_SIZE];
static size_t offset;

static int dev_open(struct inode *inode, struct file *file) {
    pr_info("device opened\n");
    return 0;
}

static int dev_release(struct inode *inode, struct file *file) {
    pr_info("device closed\n");
    return 0;
}

static ssize_t dev_read(struct file *file, char __user *ubuf,
                        size_t count, loff_t *pos) {
    size_t available = BUFFER_SIZE - offset;
    size_t len = min(count, available);

    if (len == 0)
        return 0;

    if (copy_to_user(ubuf, buffer + offset, len))
        return -EFAULT;

    offset += len;
    return len;
}
```

```
static ssize_t dev_write(struct file *file, const char __user *ubuf,
                         size_t count, loff_t *pos) {
    size_t space = BUFFER_SIZE - offset;
    size_t len = min(count, space);

    if (len == 0)
        return -ENOSPC;

    if (copy_from_user(buffer + offset, ubuf, len))
        return -EFAULT;

    offset += len;
    return len;
}

static struct file_operations fops = {
    .owner   = THIS_MODULE,
    .open    = dev_open,
    .release = dev_release,
    .read    = dev_read,
    .write   = dev_write,
};

static int __init driver_init(void) {
    major = register_chrdev(0, DEVICE_NAME, &fops);
    if (major < 0)
        return major;

    pr_info("device registered with major %d\n", major);
```

```
    return 0;
}

static void __exit driver_exit(void) {
    unregister_chrdev(major, DEVICE_NAME);
    pr_info("device unregistered\n");
}

module_init(driver_init);
module_exit(driver_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Educational character device driver");
```

٦.٣.١٤ الترجمة والتحميل

تُترجم وحدات النواة مقابل النواة العاملة:

```
make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
sudo insmod my_device.ko
```

ملاحظة:

- لا يُستخدم gcc مباشرة
- نظام بناء النواة يتحكم بالأعلام وواجهة ABI

٧.٣.١٤ اختبار الجهاز

بعد إنشاء عقدة الجهاز:

```
sudo mknod /dev/my_device c <major> 0
```

الاختبار:

```
echo "hello" > /dev/my_device
cat /dev/my_device
```

مهم:

- عمليات القراءة والكتابة تستدعي دوال النواة
- لا تُستخدم أي دوال من libc

٨.٣.١٤ معالجة المقاطعات

غالبًا ما تُشير الأجهزة إلى الأحداث باستخدام المقاطعات.

مثال على معالج مقاطعة

```
#include <linux/interrupt.h>

#define IRQ_LINE 17

static irqreturn_t irq_handler(int irq, void *dev) {
    pr_info("interrupt received\n");
    return IRQ_HANDLED;
}

static int __init irq_init(void) {
    return request_irq(IRQ_LINE, irq_handler,
                       IRQF_TRIGGER_RISING,
```

```
                        "my_device", NULL);
}

static void __exit irq_exit(void) {
    free_irq(IRQ_LINE, NULL);
}
```

قواعد أساسية:

- يجب أن تكون معالجات المقاطعات سريعة
- يُمنع النوم أو الحجب
- يُسمح فقط بأعمال حدّية

٩.٣.١٤ قيود حرجة على برامج التشغيل

- لا عمليات فاصلة عائمة
- لا تخصيص ذاكرة دون حذر
- لا حجب في السياقات الذرية
- لا ثقة بمدخلات فضاء المستخدم

مخالفة هذه القواعد قد تؤدي إلى تعطّل النظام بالكامل.

١٠.٣.١٤ أفضل الممارسات

١. اعتبر شيفرة النواة غير آمنة افتراضيًا

٢. تحقّق من كل تفاعل مع فضاء المستخدم

٣. افصل منطق العتاد عن السياسات

٤. فضّل البساطة على الأداء

٥. اختبر دائمًا داخل آلات افتراضية

١١.٣.١٤ الخلاصة

كتابة برنامج تشغيل جهاز ليست مسألة صياغة C فقط، بل مسألة مسؤولية. تعمل برامج التشغيل في أكثر سياقات النظام امتيازًا، حيث يمكن لخطأ واحد أن ينهار النظام بأكمله.
من خلال فهم الدور المعماري لبرامج التشغيل، وقيود تنفيذ النواة، والفصل الصارم بين فضاء المستخدم والنواة، يكتسب المطوّر تقديرًا عميقًا لكيفية إدارة أنظمة التشغيل للعتاد بأمان.
هذه المعرفة تنتقل مباشرة إلى الأنظمة المضمنة، والمشرفات الافتراضية، وأنظمة الإدخال/الإخراج عالية الأداء.

٤.١٤ برمجة الأنظمة المضمّنة

برمجة الأنظمة المضمّنة (Embedded Systems Programming) هي تخصص يركّز على كتابة البرمجيات التي تعمل على وحدات حوسبة متخصصة مضمّنة داخل أنظمة كهربائية أو ميكانيكية أكبر. وعلى عكس الحواسيب العامة، تخضع الأنظمة المضمّنة لقيود صارمة في الذاكرة، دورات المعالج، عرض حزمة الإدخال/الإخراج، وغالبًا ميزانية الطاقة. وفي العديد من التطبيقات، تكون البرمجيات المضمّنة زمنية حقيقية (Real-Time) و حرجة من حيث السلامة (Safety-Critical)، ما يجعل الصحة والتنبؤية مهمتين بقدر الأداء.
يقدّم هذا القسم منظورًا عمليًا على مستوى برمجة الأنظمة لتطوير الأنظمة المضمّنة باستخدام C23 كلغة مرجعية، مع تأكيد تمييز جوهري: البرمجيات الثابتة (Firmware) عادة ما تكون *Freestanding C* — دون بيئة تشغيل مستضافة ودون مكتبة قياسية كاملة — كما أن نموذج البناء، والربط، والبدء يختلف جذريًا عن برامج سطح المكتب.

١.٤.١٤ ما الذي يميّز الأنظمة المضمّنة؟

تُصمَّم الأنظمة المضمّنة لأداء مجموعة ضيقة من المهام تحت قيود قوية. ومن أمثلتها: وحدات التحكم في السيارات، وحدات التحكم الصناعية، الأجهزة الطبية، الموجّهات، الحساسات، والإلكترونيات الاستهلاكية.
تشمل خصائصها الأساسية:

١. قيود الموارد: ذاكرة RAM و Flash محدودة، قدرة معالجة محدودة، وميزانيات طاقة صارمة.

٢. السلوك الزمني الحقيقي: قد تكون المهل الزمنية إلزامية (Hard Real-Time) أو مفضلة بشدة (Soft Real-Time).

٣. التحكم المباشر بالعتاد: الشيفرة تتعامل مباشرة مع السجلات، والوحدات الطرفية، وDMA، ومتحكمات المقاطعات.

٤. الموثوقية والسلامة: الفشل قد يعني ضررًا ماديًا، خسارة مالية، أو خطرًا على البشر.

توضيح بالغ الأهمية:

- معظم البرمجيات المضمّنة هي Freestanding: لا يمكن افتراض malloc، أو إدخال/إخراج الملفات، أو العمليات، أو حتى printf.
- يبدأ البرنامج من معالج إعادة الضبط (Reset Handler)، وليس من مُحمِّل نظام تشغيل.
- يجب فهم ملف الربط (Linker Script)، وخريطة الذاكرة ،(Flash/RAM) وشيفرة البدء.

٢.٤.١٤ إعداد بيئة التطوير

يتطلب تطوير الأنظمة المضمّنة سلسلة أدوات عابرة، وآلية تحميل للبرمجيات، وواجهة تصحيح.

سلسلة الأدوات والمترجم العابر

المترجم العابر (Cross-Compiler) ينتج شيفرة لهيكلية مختلفة عن جهاز التطوير.

```
sudo apt-get install gcc-arm-none-eabi
```

- arm-none-eabi-gcc يستهدف أجهزة ARM Cortex-M دون نظام تشغيل.
- ``none-eabi'' تعني عدم وجود نظام تشغيل واستخدام واجهة ARM EABI.

أدوات التصحيح والاختبار

يتم التصحيح في الأنظمة المضمّنة عادة عبر JTAG أو SWD.

```
sudo apt-get install openocd
```

- يوفّر OpenOCD جسرًا بين أداة التصحيح و GDB.
- يمكن إيقاف التنفيذ، فحص الذاكرة، والتنفيذ خطوة بخطوة.

٣.٤.١٤ نموذج تنفيذ البرمجيات الثابتة

البرمجيات الثابتة هي الشيفرة منخفضة المستوى التي تعمل مباشرة على المتحكم. ويحتوي المشروع الأدنى عادة على:

١. شيفرة البدء: إعداد المكدس، تهيئة أقسام data وbss، وجدول المتجهات.

٢. تهيئة العتاد: شجرة التردد، GPIO، والوحدات الطرفية.

٣. الحلقة الرئيسية أو المجدول: السلوك المستقر للنظام.

٤. معالجات المقاطعات: الأحداث غير المتزامنة والتوقيت.

أنماط شائعة:

- Super-loop: حلقة while(1) واحدة.
- معتمد على المقاطعات: المقاطعة تشير، والعمل يتم لاحقًا.
- مبني على RTOS: مهام وتزامن وهيكلة.

٤.٤.١٤ مثال: وميض صمام LED

وميض LED ليس تمرينًا بدائيًا فقط، بل اختبارًا لسلسلة الأدوات، وضبط التردد، والـ GPIO، وآلية التحميل.

وميض LED على Cortex-M (على مستوى السجلات)

```
#include <stdint.h>
#include "stm32f4xx.h"

static void delay(volatile uint32_t count) {
```

```
    while (count--) {
        __asm__ volatile ("nop");
    }
}

int main(void) {
    /* Enable GPIOA clock */
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

    /* Configure PA5 as output */
    GPIOA->MODER &= ~(3u << (5u * 2u));
    GPIOA->MODER |=  (1u << (5u * 2u));

    while (1) {
        GPIOA->ODR ^= (1u << 5u);  /* Toggle LED */
        delay(1000000u);
    }
}
```

- البرمجة على مستوى السجلات حتمية وخالية من كلفة خفية.
- حلقة التأخير غير دقيقة زمنيًا، وهي لغرض العرض فقط.

الترجمة والتحويل والتحميل

```
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -g -O0 -ffreestanding \
  -nostdlib -o firmware.elf firmware.c

arm-none-eabi-objcopy -O binary firmware.elf firmware.bin
```

```
st-flash write firmware.bin 0x08000000
```

ملاحظات:

- المشاريع الحقيقية تتطلب ملف بدء وملف ربط.
- -ffreestanding و-nostdlib تعكسان افتراضات العتاد المجرد.

٥.٤.١٤ المقاطعات والحتمية

المقاطعات آلية أساسية للاستجابة للأحداث الخارجية: المؤقتات، UART، ADC، وغيرها.
قواعد الانضباط:

- إبقاء ISR قصيرة وقابلة للتنبؤ
- تجنّب الحلقات الطويلة داخل ISR
- ``المقاطعة تضع إشارة، والحلقة الرئيسية تنفّذ العمل''
- حماية البيانات المشتركة بين ISR والشيفرة الرئيسية

٦.٤.١٤ أنظمة التشغيل الزمنية الحقيقية (RTOS)

مع ازدياد التعقيد، يصبح نمط Super-loop صعب التوسّع. يوفّر RTOS تزامنًا منظّمًا: مهام، أولويات، واستباقية.

مفاهيم RTOS

- مهام/خيوط ذات أولويات
- الجدولة (استباقية أو تعاونية)

- **التزامن:** mutex، semaphore، event groups
- **الاتصال:** queues، message buffers
- **التوقيت:** delays، timers

مثال: مهمتان باستخدام FreeRTOS

```
#include <stdint.h>
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"

static void vTaskLED1(void *arg) {
    (void)arg;
    while (1) {
        GPIOA->ODR ^= (1u << 5u);
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

static void vTaskLED2(void *arg) {
    (void)arg;
    while (1) {
        GPIOA->ODR ^= (1u << 6u);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int main(void) {
```

```
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

    GPIOA->MODER &= ~(3u << (5u * 2u));
    GPIOA->MODER |=  (1u << (5u * 2u));
    GPIOA->MODER &= ~(3u << (6u * 2u));
    GPIOA->MODER |=  (1u << (6u * 2u));

    xTaskCreate(vTaskLED1, "LED1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(vTaskLED2, "LED2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    vTaskStartScheduler();

    while (1) { }
}
```

- كل مهمة تملك سلوكها الزمني.
- المجدول يضمن تداخلاً متوقعًا.

٧.٤.١٤ إدارة الطاقة

الطاقة غالبًا هي القيد المسيطر في الأنظمة المعتمدة على البطارية. تحسين الطاقة يتطلب تفكيرًا معماريًا.

تقنيات الطاقة

١. أوضاع النوم

٢. إيقاف تردد الوحدات غير المستخدمة

٣. التصميم المعتمد على الأحداث

٤. تخفيض التردد عند عدم الحاجة

مثال: الدخول في وضع النوم

```c
#include <stdint.h>
#include "stm32f4xx.h"

static void enter_sleep(void) {
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;
    __WFI();
}

int main(void) {
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

    GPIOA->MODER &= ~(3u << (5u * 2u));
    GPIOA->MODER |=  (1u << (5u * 2u));

    while (1) {
        GPIOA->ODR ^= (1u << 5u);
        enter_sleep();
    }
}
```

مهم:

- النوم فعّال فقط عند وجود مصدر إيقاظ.
- الانتظار المشغول يهدر الطاقة.

٨.٤.١٤ الموثوقية والسلامة والاختبار

أخطاء الأنظمة المضمّنة غالبًا ما تكون مادية. لذلك، الانضباط الهندسي جزء من الصحة.
مصادر فشل شائعة:

- فيض المكدس
- سباقات بين ISR والشيفرة الرئيسية
- سلوك غير معرّف
- إعادة تشغيل المراقب

٩.٤.١٤ أفضل الممارسات

١. التصميم حول القيود
٢. تفضيل الحتمية
٣. تقليل ISR
٤. جعل التوقيت صريحًا
٥. الاختبار على العتاد الحقيقي
٦. توثيق خريطة الذاكرة وتدفق البدء

١٠.٤.١٤ الخلاصة

برمجة الأنظمة المضمّنة ليست C`` على حاسوب صغير''، بل نموذج تنفيذ مستقل يعتمد على بدء من إعادة الضبط، وترجمة ،Freestanding وقيود زمنية صارمة، وتحكم مباشر بالعتاد.
بإتقان بنية البرمجيات الثابتة، وانضباط المقاطعات، وسير عمل الأدوات، والتصميم الواعي للطاقة، يمكن بناء برمجيات موثوقة وفعّالة في مجالات السيارات، والصناعة، والاستهلاك، والأنظمة الحرجة.

الفصل ١٥: مستقبل لغة C

١.١٥ اتجاهات تطوّر لغة C

منذ ظهورها في أوائل سبعينيات القرن الماضي، شكّلت لغة C العمود الفقري البنيوي للحوسبة الحديثة. فأنظمة التشغيل، والمترجمات، وشبكات الاتصال، وقواعد البيانات، والبرمجيات الثابتة تعتمد جميعها اعتماداً عميقًا على .C وعلى الرغم من الادعاءات المتكررة بزوالها، لا تزال لغة C تتطوّر بمنهجية مدروسة ومحافظة. ويمثّل معيار C23 ليس إعادة ابتكار للغة، بل استجابة هندسية دقيقة لمتطلبات العصر الحديث في مجالات السلامة، وقابلية النقل، وأدوات التطوير، والأداء.
إن فهم مستقبل C يبدأ من فهم فلسفتها: لغة تتطوّر ببطء، وتقدّم الاستقرار، وقابلية التنبؤ، والتحكم الصريح على التجريد السريع. والاتجاهات التي تشكّل C اليوم تعكس هذه الفلسفة، مع ضمان بقائها لغة حيّة وقابلة للاستخدام في أنظمة الحاضر والمستقبل.

١.١.١٥ التحديث دون التخلّي عن الهوية

أحد أبرز ملامح تطوّر لغة C هو التحديث المضبوط. فعلى عكس اللغات التي تُدخل تحولات جذرية في النماذج البرمجية، تتقدّم C عبر تعزيز تدريجي لأسسها.

- C23 كمعيار تطوّري: يقدّم C23 تحسينات في دعم Unicode، وتوحيد السمات (Attributes)، وتحديداً أوضح للأنواع العددية، وواجهات مكتبية أكثر أماناً، وصياغة معيارية

أدق. وتُحدّث هذه التغييرات لغة C دون المساس بطابعها منخفض المستوى أو نموذجها الذهني.

- الحفاظ على الدلالات الأساسية: تظل المؤشرات، وإدارة الذاكرة الصريحة، والتخطيط الحتمي للبيانات، والتفاعل المباشر مع العتاد عناصر جوهرية. ولا يتم إدخال جامع نفايات، أو بيئة تشغيل خفية، أو تجريدات إلزامية.

- التوازن بين البيئات المستضافة والمجرّدة: يواصل C23 دعم البيئات المستضافة (التطبيقات) والبيئات المجرّدة (النوى، البرمجيات الثابتة، محمّلات الإقلاع)، مما يعزّز دور C كلغة تمتد عبر كامل طبقات البرمجيات.

إن التحديث في C إضافي لا هدّام، وتبقى الشيفرة القديمة ذات معنى، ومقروءة، وذات قيمة حتى بعد عقود من كتابتها.

٢.١.١٥ التوافق الخلفي كأولوية استراتيجية

التوافق الخلفي ليس صدفة في C، بل قيد تصميمي مقصود.

- قواعد شيفرة طويلة العمر: قطاعات مثل الطيران، والاتصالات، والتمويل، وأنظمة التشغيل تحتوي على شيفرات C يعود عمرها إلى عشرات السنين. كسر التوافق سيؤدي إلى اضطراب اقتصادي وتقني جسيم.

- نموذج الإهمال التدريجي: الدوال والبنى الخطرة يتم وسمها كمهجورة ببطء، مع توفير بدائل أكثر أمانًا ودعم أدواتي مكثّف، بدل الإزالة القسرية.

- الهجرة المدفوعة بالمترجم: التحذيرات، والسمات، والتحليل الساكن، والتشخيصات توجّه المطوّرين نحو أنماط أكثر أمانًا دون إبطال الشيفرة القائمة.

تضمن هذه الاستراتيجية استمرار قابلية C للاستخدام في المجالات التي يتجاوز فيها عمر البرمجيات عمر العتاد نفسه.

٣.١.١٥ السلامة والأمان كاهتمامات على مستوى اللغة

أصبحت السلامة في تطوير C الحديث هدفًا تصميميًا أساسيًا، دون التضحية بالأداء أو التحكم.

- واجهات مكتبية أكثر أمانًا: يوسّع C23 نطاق الدوال التي تقلّل من أخطار فيض المخازن، وتجاوز السعة العددية، والسلوك غير المعرّف المرتبط بإدارة الذاكرة.
- سمات معيارية موحّدة: سمات مثل [[nodiscard]]، [[maybe_unused]] تحسّن التحقق الساكن دون تغيير دلالات التنفيذ.
- السلامة المدعومة بالأدوات: بدل تضمين آليات سلامة داخل بيئة التشغيل، تعتمد C على المترجمات، والمُعقّمات (Sanitizers)، وأدوات التحليل الساكن، والتحقق الشكلي لفرض الصحة.

تركّز استراتيجية السلامة في C على قابلية التحقق، لا على تقييد المطوّر. تظل المسؤولية بيد المبرمج، لكن مع دعم أدواتي أقوى من أي وقت مضى.

٤.١.١٥ التزامن والتوازي كمتطلبات أساسية

العتاد الحديث متوازي بطبيعته، وقد انعكس ذلك في تطوّر C.

- نموذج خيوط معياري: قدّم معيار C11 واجهة موحّدة للتزامن، ويعمل C23 على صقلها وتوضيحها.
- توسيع دعم العمليات الذرّية: يواصل C23 تحسين وضوح وتعبيرية العمليات الذرّية، وترتيبات الذاكرة، وبرمجة الأنظمة الخالية من الأقفال.
- نموذج أداء قابل للتنبؤ: على عكس التجريدات عالية المستوى، تكشف C نموذج الذاكرة صراحة، مما يجعلها مناسبة للأنظمة الزمنية الحقيقية وعالية الأداء.

لا تزال C من اللغات القليلة التي تكون فيها تكلفة التزامن مرئية وقابلة للتحكم.

٥.١.١٥ قابلية التكامل كميزة استراتيجية

أعظم ميزة طويلة الأمد لـ C ليست الصياغة ولا الأداء، بل قابلية التكامل.

- المعيار الفعلي لـ FFI: معظم اللغات الحديثة — Java، Swift، Go، Rust، Python، — تتفاعل مع النظام عبر واجهات C.
- تمثيلات بيانات مستقرة: بساطة واجهة ABI في C تجعلها اللغة الطبيعية كنقطة فصل بين البيئات التنفيذية.
- أساس سلاسل الأدوات الحديثة: LLVM، وأنظمة التشغيل، ومحمّلات التنفيذ، وبيئات اللغات تعتمد جميعها على واجهات متوافقة مع C.

بدل أن تنافس C اللغات الأحدث، فهي تمكّنها.

٦.١.١٥ الأدوات والمترجمات ونمو المنظومة

مستقبل C لا ينفصل عن منظومة أدواتها.

- ابتكار المترجمات: تواصل GCC وClang وMSVC تطوير التشخيصات، وسلاسل التحسين، وأدوات السلامة، مع الالتزام الصارم بالمعايير.
- أنظمة البناء وإدارة الاعتماديات: أدوات مثل CMake، Meson، Conan حسّنت بشكل كبير قابلية التوسّع وقابلية إعادة الإنتاج في المشاريع الكبيرة.
- تكامل بيئات التطوير: خوادم اللغة، والفهرسة الدلالية، وأدوات إعادة الهيكلة قلّصت الفجوة الإنتاجية بين C واللغات الأعلى مستوى.

لم تعد C لغة فقيرة بالأدوات، بل لغة غنيّة بها.

٧.١.١٥ الأداء كجزء من الهوية

لم يكن الأداء يومًا خيارًا في C، بل سبب وجودها.

- تحسينات مترجم متقدمة: تحسين الربط، والتحسين المعتمد على الملفات الشخصية، والتوجيه التلقائي للتوازي، وتحليل العابر للوحدات أصبحت ممارسات قياسية.
- البرمجة الواعية بالعتاد: يواصل C23 دعم المحاذاة الصريحة، والتعامل مع البتات، والتحكم منخفض المستوى الضروري للأنظمة المضمّنة والحوسبة عالية الأداء.
- نموذج تكلفة قابل للفهم: لا تزال C غير منافسة في القدرة على تحليل عدد التعليمات، وتخطيط الذاكرة، وسلوك الذاكرة المخبئية.

ولا تزال C اللغة التي تُقاس بها أداءات اللغات الأخرى.

٨.١.١٥ قابلية النقل وطول العمر

قابلية نقل C ليست مصادفة، بل نتيجة هندسة معيارية.

- معيارية صارمة: سلوك محدّد بوضوح عبر المعماريات يضمن دلالات متسقة.
- فصل واضح بين المحمول والخاص بالمنصة: تشجّع C على حدود صريحة، مما يمكّن تصميمًا متعدد المنصات موثوقًا.
- الاستمرارية عبر أجيال العتاد: شيفرة كُتبت قبل عقود لا تزال تُترجم وتعمل على أنظمة حديثة بتعديلات طفيفة.

قليل من اللغات تستطيع الادعاء بهذا المستوى من الاستمرارية الزمنية.

٩.١.١٥ المجتمع والتعليم ونقل المعرفة

لا تزال C لغة تأسيسية في التعليم والتدريب المهني.

- التفكير على مستوى الأنظمة: تعلّم C يعني فهم نماذج الذاكرة، وتمثيل البيانات، وحدود ABI، والتفاعل مع العتاد.
- خبرة قابلة للنقل: إتقان C يخفض بشكل كبير حاجز الانتقال إلى C++، Rust، التجميع، وبرمجة الأنظمة عمومًا.
- إشراف مفتوح المصدر: مشاريع البنية التحتية الكبرى تضمن تطوّرًا مستمرًا موجّهًا بمتطلبات العالم الحقيقي.

C ليست مجرد لغة، بل انضباط هندسي.

١٠.١.١٥ الخلاصة

مستقبل لغة C ليس ركوداً ولا إعادة ابتكار جذرية، بل تطوّرًا منضبطًا. ويمثّل C23 تجسيداً واضحًا لهذا النهج: تحديث حيث يلزم، وتوحيد حيث يفيد، وحفاظ على التحكم الصريح منخفض المستوى الذي يعرّف هوية C.
وطالما أن البرمجيات تحتاج إلى التفاعل المباشر مع العتاد، أو بناء أسس التنفيذ، أو تصميم المترجمات، أو فرض السلوك الحتمي، ستظل C لغة أساسية. ودورها لا يتضاءل بظهور لغات أحدث، بل يزداد ترسيخًا بها. إن إتقان C ليس مهارة موروثة، بل استثمار طويل الأمد في فهم كيفية عمل البرمجيات فعليًا.

٢.١٥ مصادر التعلّم والخطوات التالية

إن إتقان لغة البرمجة C — ولا سيما بالمستوى المطلوب لبرمجة الأنظمة، وأنظمة التشغيل، وبناء المترجمات — ليس مهمة محدودة بزمن، بل هو رحلة فكرية طويلة الأمد. وعلى الرغم من أن هذا الكتاب يقدّم أساسًا منظّمًا وعميقًا لمعيار C23، فإن الإتقان الحقيقي لا يتحقق إلا عبر الدراسة المستمرة، والممارسة المنضبطة، والاحتكاك الدائم بالأنظمة الحقيقية. يستعرض هذا القسم مصادر تعلّم موثوقة، ويقترح خطوات واضحة ترشدك لما بعد هذا الكتاب.

١.٢.١٥ الكتب الموثوقة والمواصفات الرسمية

تظل المراجع المطبوعة والمعايير الرسمية أدقّ وأكثر مصادر المعرفة موثوقية لمبرمجي C.

- المراجع التأسيسية: تُعدّ الأعمال الكلاسيكية مثل The C Programming Language لـ Brian J. Kernighan و Dennis M. Ritchie مرجعًا أساسيًا لبناء النموذج الذهني الصحيح للغة C، ويجب قراءتها بتأنٍّ لا بعجلة. تركّز هذه الكتب على الوضوح، والانضباط، والصحة البرمجية.

- مراجع C الحديثة: كتب مثل C Programming: A Modern Approach لـ K. N. King تربط بين C التقليدية والممارسات الحديثة، وتغطي موضوعات السلوك غير المعرّف، وقابلية النقل، والأنماط الأكثر أمانًا.

- معايير C الرسمية: تُعد مواصفات ISO/IEC 9899 (C11، C17، C23) المرجع الأعلى سلطة. ورغم أنها ليست تعليمية بطبيعتها، فإن قراءة المعيار — ولو جزئيًا — تُنمّي فهمًا دقيقًا للضمانات، والقيود، والحالات الحدّية للغة.

- القراءة المتقدمة والتجريبية: كتب مثل C Expert Programming: Deep C Secrets تسلّط الضوء على سوء الفهم الشائع، والقرارات التاريخية، والفخاخ الدقيقة التي تفصل بين المبرمج الجيد والمبرمج الخبير.

ومع الوقت، يتعلّم المبرمج الجاد في C التعامل مع المعيار كمرجع أساسي، لا كمصدر ثانوي.

٢.٢.١٥ التعلّم عبر الإنترنت والدورات المنهجية

توفّر المنصات الإلكترونية مرونةً ودعمًا بنيويًا، خصوصًا عند استخدامها إلى جانب المراجع الموثوقة.

- الدورات الأكاديمية: الدورات ذات الطابع الجامعي تركّز غالبًا على الصحة، والأسس النظرية، والفهم منخفض المستوى، بدل التطوير السريع للتطبيقات.
- منصات التعلّم التفاعلي: تفيد في تثبيت الصياغة والأنماط الأساسية، لكن يجب اعتبارها مكمّلة لا مرجعًا أساسيًا.
- الشروحات المرجعية: المواقع التعليمية الجيدة مناسبة للمراجعة السريعة أو البحث عن موضوع محدد، خصوصًا عند التحقق منها بمقارنتها مع المعيار.

يكون التعلّم عبر الإنترنت أكثر فاعلية عندما يقترن بالقراءة المتأنية، والتجريب، والتحقق المستمر.

٣.٢.١٥ أدوات التطوير وسير العمل الاحترافي

لا يقتصر إتقان C على اللغة نفسها، بل يشمل منظومة أدواتها.

- المترجمات: تُنفّذ GCC وClang وMSVC المعيار مع فروق دقيقة. والقدرة على فهم التحذيرات، وتفعيل أدوات السلامة، وتفسير التشخيصات مهارة احترافية أساسية.
- أدوات التصحيح والتحليل: المصححات، وأدوات فحص الذاكرة، والمحللات الساكنة ضرورية لفهم سلوك البرامج خارج حدود الشيفرة المصدرية.
- أنظمة البناء: Make، CMake، Meson ليست أدوات اختيارية في المشاريع الجادة، بل تمثل منطق البناء، وافتراضات النقل، وعلاقات الاعتماد.

إن تطوير البرمجيات بلغة C منفصل عن الأدوات فصلاً غير ممكن.

١٥.٢.٤ المشاريع مفتوحة المصدر والمشاركة المجتمعية

يتطوّر الإتقان الحقيقي عبر الاحتكاك بقواعد شيفرة حقيقية.

- المشاريع واسعة النطاق: دراسة مشاريع C ناضجة — مثل أنظمة التشغيل، وسلاسل الأدوات، والمكتبات النظامية — تكشف أنماطًا معمارية وأعرافًا برمجية لا تنقلها الكتب بالكامل.
- المساهمة كوسيلة للتعلّم: حتى المساهمات الصغيرة تعرّضك لمراجعة الشيفرة، والالتزام بالمعايير، وقضايا قابلية النقل، والتفكير طويل الأمد في الصيانة.
- المجتمعات التقنية: المنتديات، وقوائم البريد، ومتتبّعات الأخطاء تقدّم رؤية عملية لكيفية تفكير الخبراء في الصحة، والأداء، والمفاضلات الهندسية.

وغالبًا ما يكون متابعة نقاشات الخبراء تعليمية بقدر كتابة الشيفرة نفسها.

١٥.٢.٥ الممارسة عبر مشاريع هادفة

الممارسة بلا هدف تؤدي إلى التكرار، أما الممارسة الهادفة فتؤدي إلى الإتقان.

- مشاريع الأنظمة: بناء نواة صغيرة، أو مكتبة تشغيل، أو مُحاكي يفرض عليك فهم نماذج الذاكرة، وقواعد ABI، والتفاعل مع العتاد.
- أدوات اللغة: كتابة مترجم، أو مفسّر، أو محلل ساكن تعزّز فهمك للصياغة، والدلالات، والتحسين.
- تمارين الأداء: المشاريع التي تركّز على سلوك الذاكرة المخبئية، والتزامن، أو القيود الزمنية الحقيقية تُنمّي الحدس منخفض المستوى.

يجب اختيار المشاريع بناءً على المفاهيم التي تجبرك على إتقانها، لا على تعقيدها الظاهري.

٦.٢.١٥ التخصصات المتقدمة

بعد بناء قاعدة قوية، يصبح التخصص أمرًا ممكنًا وضروريًا.

- برمجة الأنظمة والنوى: تعميق المعرفة في الذاكرة الافتراضية، والجدولة، والمزامنة، وطبقات تجريد العتاد.
- بنية المترجمات واللغات: دراسة نظريات التحليل النحوي، والتمثيلات الوسيطة، ومراحل التحسين، واستراتيجيات توليد الشيفرة.
- الأنظمة المضمّنة والزمن الحقيقي: التركيز على السلوك الحتمي، وإدارة الطاقة، ومعالجة المقاطعات، والبيئات المقيّدة.

وغالبًا ما يتقاطع التخصص في C مع لغة التجميع ومعمارية الحاسوب.

٧.٢.١٥ التعلّم المستمر والنمو المهني

تتطور C ببطء، لكن المنظومة المحيطة بها لا تتوقف.

- متابعة تطور المعايير: متابعة المقترحات ونقاشات اللجان تساعد على استشراف الاتجاهات المستقبلية.
- الانخراط المهني: المؤتمرات، والأبحاث التقنية، ونقاشات المعايير تصقل الفهم عند تقاطع النظرية مع التطبيق.
- التحسين مدى الحياة: إتقان C لا يعني حفظ التفاصيل، بل صقل الحكم الهندسي، والانضباط، والتفكير التقني مع مرور الزمن.

٨.٢.١٥ الخلاصة

إتقان لغة C لا يعني تعلّم صياغة، بل فهم كيفية عمل البرمجيات في أساسها. تكافئ هذه اللغة الصبر، والدقة، والنزاهة الفكرية. ومن خلال الجمع بين المراجع الموثوقة، والممارسة المنضبطة،

والمشاريع الواقعية، والتعلّم المستمر، تنتقل من كتابة شيفرة C إلى فهم الأنظمة بعمق وتصميمها بمسؤولية.

أما الخطوات التالية، فلا يحدّدها كتاب واحد، بل تحدّدها المشكلات التي تختار حلّها، والصرامة التي تطبّقها، والعمق الذي تسعى إليه في الفهم.

الملاحق

الملحق :A مرجع مكتبة C23 القياسية

تشكّل مكتبة C القياسية الأساس الذي تُبنى عليه جميع برامج C القابلة للنقل بين الأنظمة. فهي تحدد مجموعة من الرؤوس (Headers)، والأنواع، والماكروهات، والدوال التي توفّر خدمات أساسية مثل الإدخال والإخراج، وإدارة الذاكرة، ومعالجة السلاسل النصية، والحسابات الرياضية، والتعامل مع الزمن، والتحكم في تنفيذ البرامج.

ومع اعتماد معيار C23، تواصل المكتبة تطورها مع الحفاظ الصارم على التوافق العكسي. فبدل إعادة ابتكار المنظومة، يعمل C23 على تنقيح المرافق القائمة، وتحسين الاتساق، وتعزيز دعم Unicode، وإدخال سمات حديثة للغة تمكّن المبرمج من كتابة شيفرة أوضح، وأكثر أمانًا، وأسهل صيانة—من دون التضحية بالأداء أو قابلية التنبؤ.

يهدف هذا الملحق إلى أن يكون مرجعًا موجزًا وموثوقًا لأهم مكوّنات مكتبة C23 القياسية. وهو مخصص للمراجعة السريعة والتوجيه المفاهيمي، وليس بديلاً عن المعيار الرسمي.

نظرة عامة على مكتبة C23 القياسية

تُنظَّم مكتبة C23 القياسية في ملفات رؤوس، يختص كل منها بفئة محددة بوضوح من الوظائف. تضمن هذه البنية المعيارية قابلية النقل، والاستقلال عن المترجم، وسلوكًا متوقعًا عبر منصات تمتد من الأنظمة المضمّنة وصولاً إلى أنظمة التشغيل واسعة النطاق.

الرؤوس القياسية الأساسية

<stdio.h> — الإدخال والإخراج
توفّر إمكانات الإدخال والإخراج المؤقت وغير المؤقت للملفات، والتدفّقات، والأجهزة القياسية.

- **printf, fprintf, snprintf** — إخراج منسّق
- **scanf, fscanf, sscanf** — إدخال منسّق
- **fopen, fclose, fread, fwrite** — التعامل مع الملفات
- **fgets, fputs, getchar, putchar** — إدخال/إخراج محارف وأسطر

<stdlib.h> — الأدوات العامة
تعرّف الأدوات الأساسية لإدارة الذاكرة، والتحكم في تنفيذ البرامج، وتحويل القيم العددية.

- **malloc, calloc, realloc, free** — إدارة الذاكرة الديناميكية
- **exit, abort, atexit** — التحكم في إنهاء البرنامج
- **strtol, strtoll, strtod** — تحويلات عددية موثوقة

<string.h> — السلاسل النصية وكتل الذاكرة
توفّر عمليات لمعالجة السلاسل النصية المنتهية بالمحرف الصفري ومناطق الذاكرة الخام.

- **strcpy, strncpy, strcat, strncat** — نسخ وربط السلاسل
- **strcmp, strncmp, strchr, strrchr** — المقارنة والبحث
- **memcpy, memmove, memset, memcmp** — معالجة الذاكرة

<math.h> — الدوال الرياضية
تعرّف عمليات رياضية تعتمد على الأعداد العائمة بسلوك محدد بدقة من حيث التقريب والأخطاء.

- **sin, cos, tan, sqrt, pow** — دوال مثلثية وأسّية
- **exp, log, log10** — دوال أسية ولوغاريتمية
- **ceil, floor, fabs, fmod** — تحويلات الأعداد العائمة

<time.h> — التاريخ والوقت
توفّر أدوات للتعامل مع الزمن التقويمي، وزمن المعالج، وتنسيق الوقت.

- **time, difftime, clock** — قياس الزمن
- **ctime, asctime** — تحويل إلى صيغة قابلة للقراءة
- **strftime** — إخراج منسّق للتاريخ والوقت

<ctype.h> — تصنيف المحارف
تدعم تصنيف المحارف وتحويل حالة الأحرف بشكل مستقل عن تفاصيل الترميز.

- **isalpha, isdigit, isspace, isalnum** — اختبارات المحارف
- **toupper, tolower** — تحويل حالة الأحرف

<stdint.h> — أنواع أعداد صحيحة بعرض ثابت
تعرّف أنواع أعداد صحيحة بعروض دقيقة أو دنيا، وهي ضرورية للبرمجة منخفضة المستوى القابلة للنقل.

- **int8_t, int16_t, int32_t, int64_t**
- **uint8_t, uint16_t, uint32_t, uint64_t**

<stddef.h> — تعريفات أساسية
توفّر أنواعاً وماكروهات مشتركة عبر اللغة.

- NULL ptrdiff_t, size_t,

<assert.h> — التشخيص
توفّر تأكيدات وقت التشغيل لاكتشاف أخطاء البرمجة أثناء التطوير.

- assert

التحسينات الحديثة في C23

تحسين دعم Unicode والمحارف
يواصل C23 إضفاء الطابع الرسمي على التعامل مع Unicode من خلال أنواع محارف وأدوات تحويل موحّدة.

- <uchar.h> — أدوات محارف Unicode
- **char8_t, char16_t, char32_t** — وحدات ترميز UTF-8 و UTF-16 و UTF-32
- **mbrtoc8, c8rtomb** — تحويلات UTF-8 متعددة البايتات

السمات القياسية
يعتمد C23 صياغة السمات المتوافقة مع الممارسات الحديثة للمترجمات، ما يحسّن التشخيص والتعبير عن نية المبرمج.

- [[deprecated]] — تمييز الواجهات المتقادمة
- [[nodiscard]] — فرض استخدام القيم المعادة
- [[maybe_unused]] — كبح تحذيرات عدم الاستخدام

تنقيحات المكتبة
يعمل C23 على توحيد أو توضيح عدد من المرافق الشائعة الاستخدام.

- aligned_alloc — تخصيص ذاكرة مع وعي بالمحاذاة
- تحسين الاتساق عبر واجهات السلاسل والذاكرة

أمثلة استخدام

مثال: إخراج إلى ملف

```c
#include <stdio.h>

int main(void) {
    FILE *file = fopen("example.txt", "w");
    if (!file) {
        perror("fopen");
        return 1;
    }
    fprintf(file, "Hello, C23!\n");
    fclose(file);
    return 0;
}
```

مثال: الذاكرة الديناميكية

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

int main(void) {
    int *data = malloc(10 * sizeof *data);
    if (!data) {
        perror("malloc");
```

```
        return 1;
    }
    for (int i = 0; i < 10; ++i)
        data[i] = i * i;

    for (int i = 0; i < 10; ++i)
        printf("%d ", data[i]);

    free(data);
    return 0;
}
```

مثال: معالجة السلاسل النصية

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

int main(void) {
    char src[] = "C23 Standard Library";
    char dst[64];

    strcpy(dst, src);
    printf("%s\n", dst);

    return 0;
}
```

الخلاصة

تظل مكتبة C23 القياسية محافظةً عن قصد، إذ تعطي الأولوية لقابلية النقل، والحتمية، والكفاءة على حساب التصميم القائم على التجريد المكثّف. وتكمن قوتها ليس في إخفاء التعقيد، بل في كشفه بطريقة منضبطة وموحّدة.

إن الفهم المتين لمكتبة C23 ضروري لكتابة برامج C صحيحة، وعالية الأداء، وقابلة للصيانة— خصوصًا في مجالات البرمجة منخفضة المستوى، والأنظمة المضمّنة، وبرمجة الأنظمة.

الملحق B: أخطاء شائعة في برمجة C وكيفية تجنبها

توفّر لغة البرمجة C مستوى فريدًا من التحكم، والأداء، وقابلية التنبؤ. غير أن هذه المزايا تأتي بثمن: تفترض اللغة أن المبرمج يفهم نموذج الآلة، وبيئة التنفيذ، وقواعد آلة C المجرّدة بدقة.
إن معظم الأخطاء الخطيرة في برامج C لا تنشأ من أخطاء نحوية، بل من افتراضات خاطئة تتعلق بالذاكرة، وعمر الكائنات، وترتيب التقييم، وحدود البيانات. يسلّط هذا الملحق الضوء على أكثر المزالق شيوعًا وخطورة في البرامج الواقعية المكتوبة بلغة C، ويعرض تقنيات حديثة ومنضبطة—ومتوافقة مع C23—لتجنبها.

مزالق إدارة الذاكرة

تُعد إدارة الذاكرة اليدوية أعظم قوة في C وأكبر مصدر خطر فيها في الوقت ذاته. فالأخطاء في هذا المجال تؤدي كثيرًا إلى انهيارات البرامج، وتلف البيانات، أو ثغرات أمنية قابلة للاستغلال.

تسرب الذاكرة

يحدث تسرب الذاكرة عندما تُخصَّص ذاكرة ديناميكيًا ثم تُفقد جميع المراجع إليها دون تحريرها.
مثال على المشكلة

```
void leak(void) {
    int *data = malloc(10 * sizeof *data);
    /*              */
}
```

الانضباط الصحيح

- يجب أن يكون لكل عملية تخصيص نموذج ملكية واضح.
- حرّر الذاكرة في نفس الطبقة المنطقية التي خصصتها.
- انتبه إلى الإرجاع المبكر حتى لا تتجاوز استدعاء free.

المؤشرات المعلّقة

يشير المؤشر المعلّق إلى ذاكرة تم تحريرها مسبقًا أو خرجت عن نطاقها الزمني.
مثال على المشكلة

```
int *bad_pointer(void) {
    int *p = malloc(sizeof *p);
    *p = 42;
    free(p);
    return p; /*        */
}
```

الانضباط الصحيح

- لا تُرجِع مؤشرات إلى ذاكرة محررة أو إلى متغيرات محلية.
- اضبط المؤشر إلى NULL مباشرة بعد free.
- عرّف نقل الملكية صراحةً في واجهات البرمجة.

تجاوز حدود المصفوفات (Buffer Overflows)

الكتابة خارج حدود كائن ما تؤدي إلى سلوك غير معرّف، وهي من أهم مصادر الثغرات الأمنية.
مثال على المشكلة

```
void overflow(void) {
    char buf[10];
    strcpy(buf, "This string is far too long");
}
```

الانضباط الصحيح

- مرّر دائمًا حجم المخزن بشكل صريح.

- فضّل الدوال المقيّدة (fgets ,snprintf).
- اعتبر كل مدخل خارجي معاديًا افتراضيًا.

السلوك غير المعرّف

يحدث السلوك غير المعرّف (UB) عندما لا يفرض معيار C أي متطلبات على سلوك البرنامج. وعند حدوثه، يصبح البرنامج بأكمله غير موثوق.

الكائنات غير المهيأة

قراءة كائن غير مهيأ تؤدي إلى نتائج غير متوقعة.
مثال على المشكلة

```
void uninit(void) {
    int x;
    printf("%d\n", x);
}
```

الانضباط الصحيح

- هيّئ جميع الكائنات عند تعريفها.
- فعّل تحذيرات المترجم بأعلى مستوى.
- عامل التحذيرات كأخطاء أثناء التطوير.

فك إشارة مؤشر NULL

فك إشارة مؤشر NULL يؤدي دائمًا إلى سلوك غير معرّف.
مثال على المشكلة

```c
void null_deref(void) {
    int *p = NULL;
    *p = 10;
}
```

الانضباط الصحيح

- تحقّق من صحة المؤشرات قبل استخدامها.
- استخدم التأكيدات (assertions) للثوابت الداخلية.
- صمّم الواجهات لتجنّب المؤشرات القابلة لأن تكون NULL قدر الإمكان.

فيض الأعداد الصحيحة

فيض الأعداد الصحيحة الموقّعة يُعد سلوكًا غير معرّف في C.
مثال على المشكلة

```c
void overflow(void) {
    int x = INT_MAX;
    x++;
}
```

الانضباط الصحيح

- استخدم الحساب غير الموقّع فقط عند وضوح الدلالة.
- تحقّق من الحدود قبل العمليات الحسابية.
- فضّل أنواعًا أوسع للمتغيرات التراكمية.

الأخطاء المنطقية والدلالية

تُترجم هذه الأخطاء بنجاح، لكنها تنتج سلوكًا غير صحيح.

الإسناد داخل الشروط

مثال على المشكلة

```
if (x = 1) {
    /*            */
}
```

الانضباط الصحيح

- فعّل التحذيرات الخاصة بالإسناد المشبوه.
- استخدم المقارنات الصريحة.
- استخدم المقارنات ذات الثابت في اليسار فقط عند الحاجة.

أخطاء Off-by-One

مثال على المشكلة

```
for (int i = 0; i <= 10; ++i) {
    arr[i] = i;
}
```

الانضباط الصحيح

- اعتبر حدود المصفوفات فترات نصف مفتوحة.
- استخدم أنماط حلقات متسقة.
- اشتق حدود الحلقات من حجم الكائن لا من أرقام ثابتة.

تجاهل القيم المعادة

مثال على المشكلة

```
FILE *f = fopen("data.txt", "r");
fclose(f);
```

الانضباط الصحيح

- افحص كل دالة يمكن أن تفشل.
- استخدم [[nodiscard]] حيثما أمكن.
- مرّر الأخطاء صراحةً.

أخطاء حرجة أمنيًا

ثغرات سلاسل التنسيق

مثال على المشكلة

```
printf(user_input);
```

الانضباط الصحيح

- لا تسمح أبدًا باستخدام مدخل المستخدم كسلسلة تنسيق.
- استخدم سلاسل تنسيق ثابتة.

استخدام دوال محذوفة أو خطرة

دوال مثل gets أُزيلت لأسباب أمنية جوهرية.
الانضباط الصحيح

- تجنّب الواجهات المتقادمة كليًا.
- فضّل الواجهات الصريحة والمدركة للأحجام.

حالات السباق و TOCTOU

مثال على المشكلة

```
if (access("file", W_OK) == 0) {
    fopen("file", "w");
}
```

الانضباط الصحيح

- استخدم العمليات الذرّية وآليات المزامنة.
- تجنّب أنماط ``التحقق ثم الاستخدام''.

مزالق الأداء

اختيار هياكل بيانات غير مناسبة

- طابق بنية البيانات مع نمط الوصول.
- ضع في الحسبان محلية الذاكرة وترتيبها.

الإفراط في الحالة العامة **(Global State)**

- قلّص النطاق قدر الإمكان.
- فضّل تدفق البيانات الصريح عبر المعاملات.

حلقات غير فعّالة

- قلّل الحلقات المتداخلة.
- أخرج الحسابات الثابتة خارج الحلقات.
- قِس الأداء قبل التحسين.

الخلاصة

لغة C لا تغفر عدم الدقة—لكنها تكافئ الانضباط.
تنشأ معظم مزالق C ليس من اللغة نفسها، بل من انتهاك نموذج التنفيذ وحدود التجريد. ومن خلال فهم عمر الكائنات، وملكية الذاكرة، وقواعد التقييم، ومخاطر التزامن، يمكنك كتابة برامج C ليست سريعة فحسب، بل صحيحة، وآمنة، وقابلة للصيانة.
ينبغي قراءة هذا الملحق لا كقائمة أخطاء، بل كدليل على التفكير الاحترافي في برمجة C.

الملحق :C الأدوات والمراجع لمطوري لغة C

لا يمكن فصل برمجة C عن منظومة الأدوات المحيطة بها. وبما أن C تعمل قريبًا جدًا من العتاد، فإن جودة الأدوات المستخدمة في الترجمة، والتحليل، والتصحيح، والتحقق، تؤثر مباشرةً على صحة البرنامج، وأمنه، وأدائه.
يقدّم هذا الملحق عرضًا مُنتقى بعناية لأهم الأدوات والمراجع التي يستخدمها مطورو C المحترفون اليوم، مع التركيز على سير عمل حديث ومتوافق مع معيار **C23**. الهدف ليس حصر جميع الأدوات المتاحة، بل إبراز تلك الشائعة، والناضجة تقنيًا، وذات الصلة بتطوير البرمجيات النظمية، والمضمنة، والحرجة للأداء.

بيئات التطوير المتكاملة والمحررات

تمثّل بيئة التطوير أو المحرر الواجهة الأساسية بين المطور وقاعدة الشفرة. في تطوير ،C تكون جودة الأدوات أهم من المظهر البصري.

Visual Studio Code

محرر خفيف وقابل للتوسعة، يُستخدم غالبًا كواجهة أمامية لسلاسل أدوات خارجية.

- دعم قائم على الإضافات لتطوير C
- ،IntelliSense تمييز دلالي، وتنقّل ذكي داخل الشفرة
- تصحيح أخطاء مدمج عبر أدوات تصحيح خارجية
- مناسب لسير العمل المخصص والمعتمد على الأدوات

CLion

بيئة تطوير متكاملة كاملة الميزات ومتعددة المنصات، موجهة لـ C++ و C.

- تحليل عميق للشفرة ودعم متقدم لإعادة الهيكلة
- مصحح أخطاء مدمج وأدوات تشخيص للذاكرة
- نموذج مشاريع مبني أصلاً على CMake
- مناسب للمشاريع الكبيرة والمعقدة

CDT Eclipse

بيئة تطوير مفتوحة المصدر تركّز على تطوير C++ و C.

- تكامل مع GDB ودعم أدوات القياس Profiling
- منظومة إضافات واسعة
- شائعة في البيئات المضمنة والمؤسسية

المترجمات وسلاسل الأدوات

المترجم هو الذي يحدد كيفية تفسير آلة C المجرّدة، ويؤثر مباشرة على الصحة والأداء.

GCC

مجموعة مترجمات مفتوحة المصدر، ناضجة وواسعة الانتشار.

- دعم واسع للمنصات والمعماريات
- قدرات تحسين قوية ومتقدمة
- دعم مستمر لمعايير C الحديثة بما فيها C23
- شائع في لينكس، الأنظمة المضمنة، والبناء المتقاطع

(LLVM) Clang

واجهة ترجمة حديثة مبنية على بنية LLVM.

- رسائل تشخيص عالية الجودة وواضحة
- بنية معيارية تمكّن أدوات تحليل متقدمة
- تكامل قوي مع أدوات التحليل الساكن والمُعقّمات Sanitizers
- مستخدم على نطاق واسع في البحث والإنتاج

MSVC

مترجم مايكروسوفت لتطوير البرمجيات على نظام Windows.

- تكامل عميق مع منصة Windows
- أدوات تصحيح وقياس أداء متقدمة
- خيار شائع للتطبيقات الأصلية على Windows

أدوات تصحيح الأخطاء والتحليل أثناء التنفيذ

تسمح أدوات التصحيح بفحص حالة التنفيذ الداخلية، وهو أمر حاسم في لغة لا توفر ضمانات أمان مدمجة.

GDB

المصحح القياسي للأنظمة الشبيهة بـ Unix.

- نقاط توقف، مراقبة متغيرات، وتنفيذ خطوة بخطوة

- دعم تصحيح البرامج متعددة الخيوط
- قابل للبرمجة والتوسعة

LLDB

مصحح حديث مبني لمنظومة LLVM.

- بدء أسرع وتجربة استخدام محسّنة
- تكامل مباشر مع معلومات التصحيح التي يولدها Clang
- قابل للتوسعة عبر السكربتات

Valgrind

إطار عمل ديناميكي لتحليل التنفيذ الثنائي.

- كشف تسربات الذاكرة والوصولات غير الصحيحة
- قياس سلوك الذاكرة المخبئية والتفرعات
- تحليل الخيوط والمزامنة

أدوات التحليل الساكن وجودة الشفرة

يكمل التحليل الساكن تشخيصات المترجم عبر اكتشاف فئات من الأخطاء قد لا تظهر أثناء الاختبار.

Clang-Tidy و Clang Static Analyzer

- تحليل مسارات التنفيذ الحساسة
- كشف السلوك غير المعرّف وسوء استخدام الواجهات
- فرض معايير الترميز وأفضل الممارسات

Cppcheck

محلل ساكن خفيف يركز على صحة الشفرة.

- كشف أخطاء الذاكرة والأخطاء المنطقية
- مناسب لبيئات التكامل المستمر

Splint

أداة تحليل ساكنة تعتمد على المواصفات والتعليقات التوضيحية.

- فحص الصحة بناءً على توصيفات صريحة
- ذات تأثير تاريخي في برمجة C الآمنة
- ذات طابع أكاديمي وتراثي أكثر من كونها عملية اليوم

أنظمة البناء

تنظّم أنظمة البناء عمليات الترجمة، والربط، وإدارة الاعتماديات.

Make

- بسيط وشفاف
- لا يزال شائعًا في المشاريع منخفضة المستوى والمضمنة
- يتطلب صيانة يدوية دقيقة

CMake

- نظام توليدي متعدد المنصات
- معتمد على نطاق واسع في مشاريع C الحديثة
- مناسب للأنظمة الكبيرة ومتعددة البيئات

Meson

- مصمم للسرعة والبساطة
- يشجع على بناء قابل للإعادة الإنتاج وواضح
- غالبًا ما يُستخدم مع Ninja

المكتبات والأطر العملية

رغم نزعة C إلى البساطة، تظل المكتبات ضرورية في المشاريع غير البسيطة.

GLib

- مكتبة أدوات عامة توفر هياكل بيانات ودعم قابلية النقل
- شائعة في برمجيات النظم وسطح المكتب

OpenSSL

- خوارزميات تشفير واتصال آمن
- مستخدمة على نطاق واسع لكنها تتطلب انضباطًا عاليًا في الاستخدام

SDL

- وصول متعدد المنصات للوسائط وأجهزة الإدخال
- شائعة في الألعاب والتطبيقات التفاعلية

المراجع والمجتمعات الإلكترونية

- منتديات تقنية وأرشيفات أسئلة وأجوبة
- مستودعات مفتوحة المصدر وتطبيقات مرجعية
- مراجعات مجتمعية وتغذية راجعة من الأقران

يجب استخدام هذه الموارد بحذر: فالتوثيق الرسمي، ومعايير اللغة، والاختبار التجريبي، يجب أن تظل دائمًا المرجع الأعلى مقارنة بالنصائح غير الموثقة.

الخلاصة

لا تُعرَّف برمجة C باللغة وحدها، بل بالانضباط والأدوات المحيطة بها. يعتمد سير العمل الحديث في C23 على مزيج مختار بعناية من المترجمات، والمحللات، والمصححات، وأنظمة البناء.
إتقان هذه الأدوات ليس خيارًا لمطوري C المحترفين—بل هو جزء لا يتجزأ من كتابة برمجيات صحيحة، وآمنة، وعالية الأداء. يوفّر هذا الملحق توجيهًا عمليًا نحو تلك المنظومة، ويشكّل أساسًا لتطوير C حديث ومنضبط.

الملحق D: مشاريع تطبيقية وأمثلة برمجية

تُعد الخبرة العملية عنصرًا أساسيًا لإتقان لغة C. فعلى عكس اللغات التي تُخفي التعقيد خلف طبقات من التجريد، تتطلب C من المبرمج استيعاب نموذج التنفيذ، وسلوك الذاكرة، وواجهات النظام بشكل مباشر. ولهذا السبب، تُعد المشاريع المختارة بعناية من أكثر الوسائل فاعلية لبناء نماذج ذهنية صحيحة ودقيقة.
يقدّم هذا الملحق مجموعة منظمة من المشاريع التطبيقية مرتبة حسب تزايد العمق التقني. صُمم كل مشروع لتعزيز مفاهيم محددة تم تناولها في هذا الكتاب، بدءًا من تدفق التحكم الأساسي وصولاً إلى استدعاءات النظام وبناء مكوّنات المترجمات. الأمثلة متعمدة القِصر، وتركّز على الوضوح والصحة بدلاً من الاكتمال الوظيفي.

مشاريع تأسيسية

تركّز هذه المشاريع على البنى الأساسية للغة وتدفق البرنامج، وهي مناسبة للقرّاء الجدد في C أو لمن ينتقلون من لغات عالية المستوى.

آلة حاسبة عبر سطر الأوامر

الهدف تقديم مفاهيم الإدخال والإخراج القياسي، وتدفق التحكم، والعمليات الحسابية، ومعالجة الأخطاء الأساسية.

```c
#include <stdio.h>

int main(void) {
    char op;
    double a, b;

    if (scanf(" %c %lf %lf", &op, &a, &b) != 3) {
        puts(   "    "   );
```

```
        return 1;
    }

    switch (op) {
        case '+': printf("%g\n", a + b); break;
        case '-': printf("%g\n", a - b); break;
        case '*': printf("%g\n", a * b); break;
        case '/':
            if (b == 0.0) {
                puts(   "    " );
                return 1;
            }
            printf("%g\n", a / b);
            break;
        default:
            puts(  "    "   );
    }

    return 0;
}
```

المفاهيم الأساسية

- الإدخال والإخراج المنسّق
- التحكم في التدفق باستخدام switch
- البرمجة الدفاعية

لعبة تخمين الأرقام

الهدف توضيح الحلقات التكرارية، وتوليد الأرقام شبه العشوائية، والتفاعل مع المستخدم.

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

int main(void) {
    int secret, guess, attempts = 0;

    srand((unsigned)time(NULL));
    secret = rand() % 100 + 1;

    do {
        if (scanf("%d", &guess) != 1)
            return 1;

        attempts++;

        if (guess < secret)
            puts(  "    "    );
        else if (guess > secret)
            puts(  "    "    );
        else
            printf(  "    %d    \n", attempts);

    } while (guess != secret);

    return 0;
}
```

المفاهيم الأساسية

- التحكم في الحلقات

- تدفق التنفيذ الحتمي
- العشوائية الأساسية وتهيئة البذرة

مشاريع متوسطة المستوى

تقدّم هذه المشاريع هياكل البيانات، والذاكرة الديناميكية، والتخزين الدائم.

مدير سجلات الطلاب

الهدف تقديم البُنى (Structures)، والإدخال/الإخراج الثنائي، وحفظ البيانات بشكل دائم.

```c
#include <stdio.h>

typedef struct {
    char name[32];
    int id;
    float grade;
} Student;

int main(void) {
    Student s = { "Alice", 1, 92.5f };
    FILE *fp = fopen("students.bin", "ab");

    if (!fp)
        return 1;

    fwrite(&s, sizeof s, 1, fp);
    fclose(fp);
```

```
    return 0;
}
```

المفاهيم الأساسية

- تمثيل البيانات المهيكلة
- العمليات الثنائية على الملفات
- تخطيط الكائنات وقضايا قابلية النقل

قائمة مترابطة أحادية الاتجاه

الهدف تعليم التخصيص الديناميكي، والتعامل مع المؤشرات، وإدارة العمر الافتراضي يدويًا.

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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *push(Node *head, int v) {
    Node *n = malloc(sizeof *n);
    if (!n) return head;
    n->value = v;
    n->next = head;
    return n;
}
```

```
void print(const Node *n) {
    for (; n; n = n->next)
        printf("%d -> ", n->value);
    puts("NULL");
}
```

المفاهيم الأساسية

- انضباط الملكية والعمر الافتراضي
- أنماط التخصيص في الكومة
- ثوابت هياكل البيانات

مشاريع نظم متقدمة

تكشف هذه المشاريع واجهات نظام التشغيل وبُنى المترجمات. وهي متعمدة البساطة للتركيز على المفاهيم بدل الميزات.

قشرة يونكس مصغّرة

الهدف تقديم العمليات (Processes)، واستدعاءات النظام، وتنفيذ البرامج.

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void) {
    char *args[] = { "ls", NULL };

    if (fork() == 0) {
```

```
        execvp(args[0], args);
        _exit(1);
    }

    wait(NULL);
    return 0;
}
```

المفاهيم الأساسية

- إنشاء العمليات
- استبدال صورة البرنامج
- تزامن الأب والابن

نموذج أولي لمحلل لغوي

الهدف تقديم مفهوم الترميز (Tokenization) وأساسيات الواجهة الأمامية للمترجم.

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

void tokenize(const char *s) {
    while (*s) {
        if (isalpha(*s)) {
            putchar('I');
            while (isalnum(*s)) s++;
        } else if (isdigit(*s)) {
            putchar('N');
            while (isdigit(*s)) s++;
```

```
        } else {
            putchar(*s++);
        }
    }
    putchar('\n');
}
```

المفاهيم الأساسية

- البنية المعجمية للغات البرمجة
- المسح الحتمي للنص
- أساسيات خط أنابيب المترجم

الخلاصة

لا تهدف هذه المشاريع التطبيقية إلى أن تكون تطبيقات مكتملة. بل هي تمارين صغيرة ومركّزة صُممت لتعزيز المفاهيم الجوهرية في C: التحكم الصريح، والسلوك المعرّف بدقة، واحترام النظام الأساسي.
من خلال تنفيذ هذه المشاريع وتعديلها وتوسيعها، يطوّر القارئ الانضباط اللازم للبرمجة الاحترافية بلغة C. فالإتقان لا يأتي من كتابة شيفرة أكثر، بل من فهم دقيق لما تعنيه كل سطر برمجي—وما تكلفته الفعلية.
يوفّر هذا الملحق جسرًا عمليًا بين النظرية وبرمجة النظم الواقعية في C23 الحديثة.

المراجع

يعتمد هذا الكتاب على مزيج من معايير اللغة الرسمية، والمراجع الأكاديمية، والمصادر المهنية الراسخة عبر عقود طويلة من برمجة النظم. وقد تم اختيار المواد المدرجة في هذا الفصل بناءً على دقتها التقنية، وطول عمرها، وأهميتها المباشرة لبرمجة النظم الحديثة باستخدام C23.
يُنصح القرّاء بالتعامل مع المواصفات الرسمية والنصوص الأساسية بوصفها مصادر مرجعية موثوقة، والتعامل مع المواد المتاحة على الإنترنت بحس نقدي عالٍ.

C23 وبرمجة C الحديثة

- فريق عمل ISO للغة C (WG14)
  - مسودات العمل الرسمية، وتقارير العيوب، والمقترحات الخاصة بمعيار لغة C.
  - المرجع المعياري الأساسي لجميع دلالات لغة C، بما في ذلك C23.
- Gustedt Jens — C Modern
  - معالجة حديثة وموجهة بالمعيار الرسمي لبرمجة C.
  - يركّز على الاستخدام المنضبط، والسلوك المعرّف، والأساليب المعاصرة المتوافقة مع تطور ما بعد C11.
- King N. K. — C Programming: A Modern Approach

- مرجع شامل ومنهجي يغطي الأساسيات والموضوعات المتقدمة.
- مفيد بشكل خاص لترسيخ النماذج الذهنية الصحيحة للغة .C

البرمجة منخفضة المستوى وبرمجة النظم

- O’Hallaron D. Bryant، R. — Perspective Programmer’s A Systems: Computer
  - مرجع تأسيسي يشرح كيفية تفاعل البرامج مع العتاد.
  - أساسي لفهم الذاكرة، والتنفيذ، والأداء على مستوى النظام.
- Bartlett Jonathan — Up Ground the from Programming
  - يركّز على العلاقة بين C، ولغة التجميع، ونموذج الآلة.
  - ذو قيمة خاصة للقراء المتجهين نحو البرمجة منخفضة المستوى.
- Ritchie M. Dennis Kernighan، W. Brian — Language Programming C The
  - العرض الأصلي للغة C.
  - ذو أهمية تاريخية كبيرة، ولا يزال قيّمًا من حيث الوضوح، مع ضرورة الرجوع إلى المعايير الحديثة لضمان الدقة.

أنظمة التشغيل

- Gagne G. Galvin، P. Silberschatz، A. — Concepts System Operating
  - مرجع أكاديمي شامل لمبادئ وتصميم أنظمة التشغيل.
- Tanenbaum S. Andrew — Systems Operating Modern
  - يركّز على المفاهيم المعمارية وبنية الأنظمة.
  - مفيد لفهم المفاضلات في تصميم أنظمة التشغيل.
- Arpaci- A. Arpaci-Dusseau، R. — Pieces Easy Three Systems: Operating Dusseau
  - شرح مبسط وواضح لمفاهيم أنظمة التشغيل الأساسية.
  - فعّال بشكل خاص عند دمجه مع التجارب العملية.

تصميم المترجمات وتنفيذ اللغات

- J. Sethi، R. Lam، M. Aho، A. — Tools and Techniques, Principles, Compilers: Ullman
  - المرجع الكلاسيكي في نظرية المترجمات.
  - يوفّر الأسس الرسمية للتحليل المعجمي، والتحليل النحوي، والتحسين، وتوليد الشيفرة.
- Torczon Linda Cooper، Keith — Compiler a Engineering
  - منظور عملي يركّز على تنفيذ المترجمات الحديثة.
- Appel W. Andrew — C in Implementation Compiler Modern
  - دليل عملي لبناء المترجمات باستخدام لغة C.
  - ذو صلة مباشرة بمحاور هذا الكتاب.

سلاسل الأدوات والتوثيق التقني

- توثيق GCC
  - مرجع لسلوك المترجم، والتحسينات، والامتدادات.
  - أساسي لفهم الترجمة الواقعية لبرامج C.
- توثيق Clang وLLVM
  - يقدّم نظرة عميقة على بنية المترجمات الحديثة والتشخيصات.
  - مفيد بشكل خاص للتحليل وبناء الأدوات.
- Explorer Compiler
  - أداة لفحص الشيفرة المجمّعة الناتجة وتأثيرات التحسين.
  - قيّمة لتحليل الأداء والتعلّم.
- Wiki OSDev
  - قاعدة معرفة مجتمعية لتطوير أنظمة التشغيل.
  - مفيدة عند استخدامها جنبًا إلى جنب مع المراجع الأولية.

التطبيق العملي والتعلّم التطبيقي

- مشاريع لغوية ومترجمات
  - تصميم مترجم أو مفسّر صغير يعزّز فهم التحليل النحوي، والدلالات، وتوليد الشيفرة.
- تجارب أنظمة التشغيل
  - كتابة شيفرة الإقلاع، أو مديري الذاكرة، أو المجدولات توفّر فهمًا عميقًا لبرمجة النظم.
- المساهمة في البرمجيات مفتوحة المصدر
  - دراسة والمشاركة في مشاريع ناضجة تكشف القيود الواقعية، والمعايير، ومفاضلات التصميم.

ملاحظة ختامية

لا يمكن لأي مرجع واحد أن يعوّض القراءة المتأنية لمعيار لغة C، أو التجريب العملي، أو التفكير المنضبط. تشكّل الأعمال المدرجة هنا أساسًا متينًا يمكن بناء الكفاءة المهنية في C وبرمجة النظم عليه.

يُنصح القرّاء بالعودة إلى هذه المراجع بشكل دوري مع تعمّق فهمهم وتطوّر خبرتهم.