https://simplifycpp.org

معيار ISO C23 لبرمجة الأنظمة

فهم معيار C في التطبيق العملي

إعداد: أيمن الحراكي

معيار ISO C23 لبرمجة الأنظمة

فهم معيار C في التطبيق العملي

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

يناير 2026

المحتويات

المقدمة

بدأ تعاملي مع لغة البرمجة C عام 1989 باستخدام بيئة Borland Turbo C، حيث كانت لغةً صارمة تتطلب دقة عالية وفهمًا عميقًا لسلوك الذاكرة والتنفيذ. ومع ظهور C++ لاحقًا، كان الانتقال إليها طبيعيًا بفضل البرمجة كائنية التوجه وتحسين تنظيم الشيفرة، لتصبح لغتي المهنية الأساسية لسنوات طويلة.

ومع تراكم الخبرة، تبيّن أن C لم تفقد يومًا مكانتها، رغم ظهور لغات أحدث. فهي لا تزال اللغة الأساسية في المجالات التي تتطلب قابلية تنبؤ عالية، وتجريدًا محدودًا، وتحكمًا مباشرًا في الذاكرة والتنفيذ، مثل أنظمة التشغيل، والأنظمة المدمجة، وتطوير المترجمات، والبرمجة منخفضة المستوى.

وانطلاقًا من ذلك، جاء هذا الكتاب معتمدًا على معيار ISO C23، لا بدافع الحداثة، بل لأنه الصيغة الأحدث والأوضح والمعتمدة رسميًا للغة C. ويركّز الكتاب على المجالات التي كانت C قوية فيها تاريخيًا، دون محاولة تحميلها أدوارًا لم تُصمَّم لها.

نظرة عامة على محتوى الكتاب

تم تنظيم الكتاب لبناء فهم مفاهيمي متين وكفاءة عملية، ويشمل:

- تطور لغة C وفلسفة تصميمها.
- فهم معيار ISO C23 وكتابة شيفرة صحيحة وقابلة للنقل.
- برمجة الأنظمة المدمجة وأنظمة التشغيل.

- تطوير المترجمات والأدوات.
- البرمجة منخفضة المستوى وإدارة الذاكرة.
- مشاريع عملية تعزز الانضباط والوضوح.

هدف هذا الكتاب

يهدف هذا الكتاب إلى تقديم C بصورتها الحقيقية: لغة تكافئ الانضباط والفهم العميق، وتضع المسؤولية كاملة على المبرمج. ومن خلال التركيز على C23، يتعلم القارئ كتابة شيفرة صحيحة، قابلة للنقل، وسهلة الصيانة في مجالات البرمجة على مستوى الأنظمة حيث لا تزال C غير قابلة للاستبدال.

كلمة ختامية

تستمر C لا لأنها عصرية، بل لأنها صادقة فيما تقدمه وما لا تقدمه. وقد كُتب هذا الكتاب لمن يسعى إلى فهم هذه الصراحة، وتحمل المسؤولية التي تفرضها، واستخدام C في مواضعها الصحيحة.

أيمن الحراكي

الفصل ١: مقدمة إلى C23

١.١ تاريخ وتطور لغة C

١.١.١ نشأة لغة C

نشأت لغة البرمجة C في أوائل سبعينيات القرن الماضي في مختبرات Bell Labs، وطوّرها **Dennis Ritchie** بوصفها خليفةً للغة **B**، والتي كانت بدورها مشتقة من **BCPL (Basic Combined Programming Language)**. وقد صُمِّمت C أساسًا لتنفيذ **UNIX operating system**، مع تحقيق توازن بين الوصول منخفض المستوى إلى العتاد ونموذج برمجة منظم وقابل للنقل.

- دوافع التصميم:
  أُنشئت C لتمكين برمجة الأنظمة بكفاءة، مع تحكم مباشر في الذاكرة، وأداء يمكن التنبؤ به، وتوافق وثيق بين تراكيب اللغة وتعليمات الآلة.

- المحطات المبكرة:

  - 1972: أول تنفيذ للغة C على PDP-11.
  - 1973: إعادة كتابة UNIX بلغة C، مما أثبت قابليتها للنقل.
  - 1978: نشر كتاب **The C Programming Language** (Kernighan & Ritchie)، والذي عرّف ما عُرف لاحقًا باسم **K&R C**.

٢.١.١ من K&R C إلى التقييس وفق ISO

- **K&R C** (1978):
المواصفة غير الرسمية الأصلية، وكانت تفتقر إلى فحص أنواع رسمي، ونماذج التصريح بالدوال (function prototypes)، ومكتبة قياسية موحّدة.

- **ANSI C / C89** (1989):
قنّنت اللغة، وقدّمت نماذج الدوال، ومكتبة قياسية معرّفة، وحسّنت قابلية النقل.

- **ISO C90** (1990):
تبنّت ANSI C مع تعديلات تحريرية طفيفة، ورسّخت C بوصفها معيارًا دوليًا.

٣.١.١ معايير C الحديثة

- **C99:**
  - دوال inline
  - مصفوفات بطول متغير (Variable Length Arrays — VLAs)
  - نوع الأعداد الصحيحة long long
  - النوع bool عبر <stdbool.h>
  - التعليقات أحادية السطر (//)

- **C11:**
  - دعم اختياري للخيوط (<threads.h>)
  - العمليات الذرّية (<stdatomic.h>)
  - _Generic للبرمجة المعمّمة بحسب النوع
  - واجهات فحص الحدود (الملحق Annex K الاختياري)

- **C17 (C18):**
  مراجعة لإصلاح العيوب في C11 دون إضافة ميزات لغوية جديدة.

- **C23:**
  إصدار تحديثي يركز على التنظيف، والاتساق، وافتراضات أكثر أمانًا، وتحسينات المكتبة، بدلاً من إعادة تصميم جذرية للغة.

١.١.٤ دور C في الحوسبة الحديثة

لا تزال C لغةً تأسيسية في:

- أنظمة التشغيل والنوى
- الأنظمة المدمجة والأنظمة ذات الزمن الحقيقي
- المترجمات، وبيئات التشغيل، والآلات الافتراضية
- مكدسات الشبكات ومكتبات النظام

ويعود بقاؤها إلى قابليتها للتنبؤ، واستقرار ABI، وافتراضاتها الدنيا حول بيئة التشغيل.

٢.١ ما الجديد في C23

١.٢.١ أهداف C23

- تبسيط اللغة عبر إزالة التراكيب المتقادمة
- تحسين الأمان دون تغيير الطبيعة منخفضة المستوى للغة C
- تحديث المكتبة القياسية
- تعزيز الاتساق مع الممارسات القائمة

١.٢.٤ التغييرات على مستوى اللغة

- إزالة التصريحات الضمنية للدوال
  يجب التصريح عن جميع الدوال قبل استخدامها.
- إزالة الكلمة المفتاحية register
  أصبحت متقادمة ولا تأثير لها.
- تحسين اتساق الأنواع
  قواعد أنظف لترقيات الأعداد الصحيحة والمؤهلات (qualifiers).
- السمات (**Attributes**) في **C**
  توحيد السمات باستخدام الصياغة [[...]] (مستعارة من C++ لكن معرّفة خصيصًا لـ C):

```
[[deprecated]] void old_api(void);
[[maybe_unused]] int debug_value;
```

٢.٢.٤ اليونيكود ومعالجة المحارف

- أصبح UTF-8 هو ترميز محارف المصدر الموصى به
- تم توحيد literals من الشكل u8"string"
- تقديم النوع char8_t بوصفه نوعًا مميزًا لوحدة ترميز UTF-8

٣.٢.٤ تحسينات المكتبة القياسية

- <stdbit.h> — أدوات لمعالجة البتّات
- <stdckdint.h> — حسابات أعداد صحيحة مع فحص التجاوز
- توضيح سلوك realloc

- تحسينات في التشخيص وفرض القيود

١.٢.٥ ما الذي لا تضيفه C23

- لا توجد كلمة مفتاحية nullptr
- لا توجد aliases باستخدام using
- لا توجد تعدادات ذات نطاق أو نوع (scoped/typed enums)
- لا توجد كلمة مفتاحية مدمجة bool (لا تزال عبر <stdbool.h>)
- لا توجد سلامة ذاكرة تلقائية

تتجنب C23 عمدًا التجريدات على نمط C++.

١.٢.٦ الميزات المتقادمة والمزالة

- gets — أُزيلت
- تعريفات الدوال بأسلوب K&R — أُزيلت
- واجهات مكتبية متقادمة — أُعلن تقادمها

١.٣ لماذا لا تزال C مهمة

١.٣.١ الأداء وقابلية التنبؤ

- لا تخصيصات مخفية
- لا نظام أنواع وقت التشغيل
- تحكم مباشر في تخطيط الذاكرة

٢.٣.١ قابلية النقل واستقرار ABI

- اتفاقيات نداء مستقرة
- توافق ثنائي طويل الأمد
- اللغة المهيمنة لواجهات الأنظمة

٣.٣.١ المعرفة التأسيسية

يُنمّي تعلم C فهمًا عميقًا لـ:

- نماذج الذاكرة
- تنفيذ المعالج
- اتفاقيات النداء
- حدود السلوك غير المعرّف

٤.١ إعداد بيئة C23

١.٤.١ المترجمات

- **GCC 13+**: دعم جزئي لـ C23 (-std=c23)
- **Clang 16+**: تغطية واسعة لميزات C23
- **MSVC**: دعم محدود وقيد التطور لـ C23

١.٤.٢ World Hello

```c
#include <stdio.h>

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

١.٤.٣ الترجمة

```
gcc -std=c23 hello.c -o hello
clang -std=c23 hello.c -o hello
```

١.٤.٤ الخلاصة

يُعد C23 معيار تنقيح لا إعادة تعريف؛ إذ يعزّز فلسفة C الأساسية بدل تغييرها. فهو يُحدّث الصياغة، ويحسّن التشخيص، وينظّف السلوكيات التراثية، ويؤكد دور C كلغة الأنظمة، والنوى، والأسس البرمجية.

الفصل ٢: أساسيات C23

١.٢ الصياغة الأساسية وبنية البرنامج

يُعد فهم الصياغة الأساسية وبنية برنامج C أمرًا جوهريًا لإتقان اللغة. وعلى الرغم من أن C23 يقدم تنقيحات وتحسينات في المكتبة، فإن الصياغة الأساسية ونموذج التنفيذ في C يظلان مستقرين ومتسقين مع المعايير السابقة. يعرّف هذا القسم البنية الأساسية لبرنامج C23 والقواعد التي تحكم صياغته.

١.١.٢ بنية برنامج C

يتكوّن برنامج C من توجيهات المعالج المسبق (preprocessor)، وتعريفات الدوال، والتعليميات (statements)، والتعابير (expressions). ويُبيّن المثال التالي أبسط بنية ممكنة:

```c
#include <stdio.h>

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

توجيهات المعالج المسبق

- تُعالَج قبل عملية الترجمة
- تُستخدم لتضمين الملفات، وتعريف الماكرو، والترجمة الشرطية

```
#include <stdio.h>
#define PI 3.14159
```

الدالة main

- يبدأ تنفيذ البرنامج من الدالة main
- التواقيع القياسية في البيئات المستضافة:

```
int main(void);
int main(int argc, char *argv[]);
```

تشير إعادة القيمة 0 إلى انتهاء البرنامج بنجاح.

التعليميات والتعابير

- التعليميات تنفّذ أفعالاً
- التعابير تُقيَّم إلى قيم

```
int x = 10;
int sum = x + 5;
printf("%d\n", sum);
```

التعليقات

```c
// Single-line comment

/* Multi-line
   comment */
```

٢.١.٢ قواعد الصياغة الأساسية

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
```

الكتل والأقواس

```c
if (x > 0) {
    printf("Positive\n");
}
```

المسافات البيضاء

المسافات البيضاء غالبًا غير مؤثرة دلاليًا، وتُستخدم لتحسين قابلية القراءة.

٣.١.٢ كتابة برنامج C23 بسيط

```c
#include <stdio.h>

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

٤.١.٢ الأخطاء الشائعة وأفضل الممارسات

أخطاء شائعة

- نسيان الفواصل المنقوطة
- استخدام متغيرات غير مهيأة
- نسيان تضمين الترويسات المطلوبة

أفضل الممارسات

- استخدام معرفات ذات معنى
- تهيئة المتغيرات
- الحفاظ على تنسيق متسق

٢.٢ أنواع البيانات والمتغيرات

يحافظ C23 على نظام الأنواع التقليدي في C مع قواعد أوضح ودعم مكتبي محسّن.

١.٢.٢ أنواع البيانات الأساسية

أنواع الأعداد الصحيحة

- char
- short
- int
- long
- long long

يتم التحكم في الإشارة عبر الكلمتين المفتاحيتين signed و unsigned.

أنواع الفاصلة العائمة

- float
- double
- double long

النوع المنطقي

```c
#include <stdbool.h>

bool flag = true;
```

٢.٢.٢ الأنواع المشتقة والمُعرّفة من قبل المستخدم

المصفوفات والمؤشرات

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

البُنى والاتحادات

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

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

أسماء الأنواع البديلة

```
typedef int Integer;
```

٣.٢.٢ المتغيرات، النطاق، والعمر

المتغيرات المحلية

```
void func(void) {
    int x = 10;
}
```

المتغيرات العامة

```
int counter = 0;
```

٤.٢.٢ الثوابت

استخدام const

```
const int MAX_SIZE = 100;
```

استخدام الماكرو

```
#define PI 3.14159
```

٥.٢.٢ تحويل الأنواع

التحويل الضمني

```
int x = 10;
double y = x;
```

التحويل الصريح

```
double y = 3.14;
int x = (int)y;
```

٣.٢ العوامل والتعابير

تتعامل العوامل مع البيانات، بينما تقوم التعابير بحساب القيم.

١.٣.٢ فئات العوامل

- حسابية
- علائقية
- منطقية
- إسناد
- بتّية
- شرطية

٢.٣.٢ أسبقية العوامل

تتبع العوامل تسلسل أسبقية ثابت. ويُستحسن استخدام الأقواس لضمان الوضوح والصحة.

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

٤.٢ تدفق التحكم

١.٤.٢ التعليميات الشرطية

```
if (x > 0) {
    puts("Positive");
```

```
} else {
    puts("Non-positive");
}
```

تعليمة switch

```
switch (day) {
    case 1: puts("Monday"); break;
    case 2: puts("Tuesday"); break;
    default: puts("Other");
}
```

٢.٤.٢ الحلقات

حلقة for

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

حلقة while

```
while (count < 5) {
    count++;
}
```

حلقة do-while

```
do {
    count++;
} while (count < 5);
```

٥.٢ الإدخال والإخراج

١.٥.٢ إدخال/إخراج وحدة التحكم

```
printf("Value: %d\n", x);
scanf("%d", &x);
```

٢.٥.٢ إدخال/إخراج الملفات

```
FILE *f = fopen("data.txt", "w");
if (f) {
    fprintf(f, "Hello\n");
    fclose(f);
}
```

٣.٥.٢ الإدخال/الإخراج الثنائي

```
fwrite(data, sizeof(int), 5, file);
fread(data, sizeof(int), 5, file);
```

٦.٢ الخلاصة

تظل أساسيات C23 متجذّرة في التصميم الأصلي للغة C: تحكم صريح، وسلوك يمكن التنبؤ به، وتجريد محدود. وقد أسّس هذا الفصل الصياغة الأساسية، وأنواع البيانات، والعوامل، وتدفق التحكم، ومرافق الإدخال والإخراج التي تشكّل الأساس لجميع برامج C23. وتبقى هذه المبادئ ضرورية لبرمجة الأنظمة، وتطوير الأنظمة المدمجة، والبرمجيات الحساسة للأداء.

الفصل ٣: الدوال في C23

١.٣ تعريف الدوال واستدعاؤها

تُعد الدوال الآلية الأساسية لتنظيم البرامج في لغة C. فهي تمكّن إعادة استخدام الشيفرة، والتجريد، والتقسيم المنطقي. ويحافظ معيار C23 على نموذج الدوال التقليدي في C، مع التأكيد على تدفق بيانات صريح، واتفاقيات نداء يمكن التنبؤ بها، وتجريد محدود.

١.١.٣ ما هي الدالة؟

الدالة هي كتلة مسماة من الشيفرة تؤدي مهمة محددة. وقد تستقبل الدالة قيم إدخال (معاملات)، وقد تُعيد قيمة واحدة إلى الجهة المستدعية.
تُحسّن الدوال:

- الوحدوية
- قابلية القراءة
- قابلية الصيانة

٢.١.٣ تعريف دالة

يحدّد تعريف الدالة نوع القيمة المعادة، واسم الدالة، وقائمة المعاملات، وجسم الدالة.

الصياغة العامة

```
return_type function_name(parameter_list) {
    // function body
}
```

- **return_type**: نوع القيمة المعادة، أو void
- **function_name**: مُعرِّف يتبع قواعد التسمية
- **parameter_list**: صفر أو أكثر من المعاملات المعرّفة بأنواع

مثال

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

٣.١.٣ استدعاء دالة

ينقل استدعاء الدالة التحكم إلى الدالة وقد يستقبل قيمة مُعادة.

```
int result = add(5, 10);
```

تُقيَّم المعاملات وتُنقل وفق ABI الخاص بالمنصة.

٣.١.٤ معاملات الدوال

التمرير بالقيمة

تُمرَّر جميع معاملات الدوال في C بالقيمة. أي تستقبل الدالة نسخة من القيمة المُمرَّرة.

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

التمرير عبر المؤشرات

للسماح للدالة بتعديل كائن ما، يجب تمرير عنوانه.

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

يُوصف هذا عادةً بأنه "تمرير عبر مؤشر"، وليس تمريرًا بالمرجع.

المصفوفات كمعاملات

تتحول المصفوفات إلى مؤشرات عند تمريرها إلى الدوال.

```
void print_array(const int arr[], int size);
```

٥.١.٣ قيم الإرجاع

إرجاع قيمة

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

الدوال من النوع void

```
void greet(void) {
    puts("Hello");
}
```

إرجاع عدة نتائج

لا تدعم C إرجاع عدة قيم مباشرة. ومن الأنماط الشائعة:

- إرجاع بنية (struct)
- تعديل الكائنات عبر المؤشرات

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

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

٦.١.٣ تصريحات الدوال (Prototypes)

يُعلن تصريح الدالة عن وجودها قبل تعريفها.

```
int add(int a, int b);
```

تُعد التصريحات إلزامية في C23؛ ولا يُسمح بالتصريحات الضمنية.

٧.١.٣ أفضل الممارسات

- إبقاء الدوال صغيرة ومركّزة
- تفضيل المعاملات على الحالة العامة
- استخدام const للمعاملات للقراءة فقط
- إعلان التصريحات في ملفات الترويسات

٢.٣ الدوال العودية

١.٢.٣ أساسيات العودية

تستدعي الدالة العودية نفسها لحل مشكلة عبر تقليلها إلى حالة أصغر.
يجب أن تحدد كل دالة عودية:

- حالة أساس
- حالة عودية

٢.٢.٣ مثال: المضروب (Factorial)

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

٣.٢.٣ مثال: فيبوناتشي

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

٤.٢.٣ مقايضات العودية

المزايا

- تعبير واضح عن المشكلات العودية
- مناسبة طبيعية للأشجار وخوارزميات التقسيم والتغلب

العيوب

- استهلاك المكدس مع كل نداء
- احتمال فيض المكدس
- غالبًا أبطأ من الحلول التكرارية

٥.٢.٣ العودية الذيلية

تحدث العودية الذيلية عندما يكون النداء العودي هو العملية الأخيرة. قد تُحسّنها بعض المترجمات، لكن لا تضمن C إزالة النداء الذيلي.

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

٣.٣ الدوال المُضمّنة (Inline) في C23

١.٣.٣ ماذا تعني inline في C؟

في C، تُعد inline محدِّدًا للربط والرؤية، وليست أمرًا مباشرًا للمترجم. فهي تُشير إلى إمكانية توسيع الدالة موضعيًا، دون إلزام.

٢.٣.٣ الصياغة

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

٣.٣.٣ قواعد inline والربط

يعتمد الاستخدام الصحيح على نوع الربط:

- inline static: مفضّل للاستخدام الداخلي وملفات الترويسات
- inline وحدها تتطلب تعريفًا خارجيًا في موضع آخر

```
static inline int max(int a, int b) {
    return (a > b) ? a : b;
}
```

٤.٣.٣ المزايا

- إزالة كلفة النداء (عند التوسيع الموضعي)
- تمكين تحسينات إضافية
- بديل آمن من حيث الأنواع عن الماكرو

٥.٣.٣ العيوب

- زيادة حجم الشيفرة
- تعقيد التنقيح
- قد يتجاهل المترجم التلميح

٦.٣.٣ الدوال المُضمّنة مقابل الماكرو

- الدوال المُضمّنة تلتزم بقواعد الأنواع
- الماكرو يقوم باستبدال نصّي خام
- الدوال المُضمّنة تحترم النطاق

```c
#define SQUARE(x) ((x) * (x))

static inline int square_fn(int x) {
    return x * x;
}
```

٤.٣ الخلاصة

تحافظ الدوال في C23 على نموذج التنفيذ الصريح والقابل للتنبؤ في C. وقد غطّى هذا الفصل تعريف الدوال، وتمرير المعاملات، وقيم الإرجاع، والعودية، والدوال المُضمّنة وفق دلالات C الصحيحة. ويُعد إتقان هذه المفاهيم أساسيًا لكتابة برامج C فعّالة، ووحدوية، وقابلة للصيانة.

الفصل ٤: المؤشرات وإدارة الذاكرة

٤.١ فهم المؤشرات

تُعد المؤشرات من أكثر المفاهيم جوهرية في لغة البرمجة C. فهي توفر وصولاً مباشرًا إلى الذاكرة وتمكّن من البرمجة منخفضة المستوى بكفاءة. وفي الوقت نفسه، يُعد الاستخدام غير الصحيح للمؤشرات المصدر الأساسي للسلوك غير المعرّف في برامج C.
يشرح هذا الفصل المؤشرات من منظور نموذج الذاكرة، لا بوصفها مجرد صياغة لغوية.

٤.١.١ ما هو المؤشر؟

المؤشر هو كائن تكون قيمته عنوان كائن آخر أو دالة.

- كل كائن يشغل حيّزًا من الذاكرة
- يخزّن المؤشر عنوان أول بايت من ذلك الكائن

٤.١.٢ تصريح المؤشرات وتهيئتها

صياغة التصريح

```
int *ptr;
```

يحدد النوع نوع الكائن الذي يشير إليه المؤشر.

التهيئة

يجب تهيئة المؤشرات قبل استخدامها.

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

٣.١.٤ فك الإشارة (Dereferencing)

يتيح فك الإشارة الوصول إلى الكائن المشار إليه.

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

يؤدي فك إشارة مؤشر غير صالح إلى سلوك غير معرّف.

٤.١.٤ المؤشرات والمصفوفات

تتحول تعابير المصفوفات إلى مؤشر لأول عنصر فيها في معظم التعابير.

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

تمييز مهم:

- المصفوفة ليست مؤشرًا
- اسم المصفوفة يتحلل إلى مؤشر في التعابير

٥.١.٤ حسابات المؤشرات

تُقاس حسابات المؤشرات بحجم النوع المشار إليه.

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

p++;  // advances by sizeof(int)
```

حسابات مؤشرات صالحة

- ضمن نفس كائن المصفوفة
- يُسمح بالمؤشر بعد آخر عنصر مباشرة (لكن لا يجوز فك إشارته)

٦.١.٤ طرح المؤشرات

ينتج عن طرح مؤشرين عدد العناصر بينهما.

```
int diff = &arr[3] - &arr[0];  // diff == 3
```

يجب أن يشير كلا المؤشرين إلى نفس المصفوفة.

٧.١.٤ مقارنة المؤشرات

تُعرّف مقارنة المؤشرات فقط ضمن نفس كائن المصفوفة.

٨.١.٤ مؤشرات إلى مؤشرات

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

يجب أن يكون كل مستوى من مستويات فك الإشارة صالحًا.

٩.١.٤ أخطاء شائعة في استخدام المؤشرات

مؤشرات غير مهيأة

```c
int *p;
*p = 5;  // Undefined behavior
```

مؤشرات معلّقة

```c
int *p = malloc(sizeof *p);
free(p);
*p = 10;  // Undefined behavior
```

الوصول خارج الحدود

```c
int arr[3];
int *p = arr + 3;  // valid
*p = 10;           // Undefined behavior
```

١٠.١.٤ أفضل الممارسات

- تهيئة المؤشرات
- استخدام NULL للمؤشرات غير الصالحة
- استخدام const حيثما كان مناسبًا
- الحفاظ على وضوح أعمار المؤشرات

٢.٤ التخصيص الديناميكي للذاكرة

يتيح التخصيص الديناميكي للذاكرة حجز الذاكرة وقت التشغيل.

١.٢.٤ malloc

```
int *arr = malloc(5 * sizeof *arr);
if (!arr) {
    return 1;
}
```

ملاحظة: لا حاجة إلى تحويل قيمة الإرجاع من malloc في C، بل يُعد ذلك غير مُستحسن.

٢.٢.٤ calloc

```
int *arr = calloc(5, sizeof *arr);
```

تُهيّأ الذاكرة بالقيم الصفرية.

realloc ٣.٢.٤

```
int *tmp = realloc(arr, 10 * sizeof *arr);
if (!tmp) {
    free(arr);
    return 1;
}
arr = tmp;
```

لا يجب الكتابة فوق المؤشر الأصلي حتى ينجح إعادة التخصيص.

free ٤.٢.٤

```
free(arr);
arr = NULL;
```

٥.٢.٤ مزالق التخصيص الديناميكي

- تسرّب الذاكرة
- التحرير المزدوج
- الاستخدام بعد التحرير

٣.٤ أنماط الملكية وإدارة الموارد في C

لا توفر C إدارة تلقائية للذاكرة. وبدلاً من ذلك، يجب التعبير عن الملكية صراحةً عبر الأعراف والانضباط.

١.٣.٤ نمط المالك الواحد

```
int *create_array(size_t n) {
    return malloc(n * sizeof(int));
}
```

يمتلك المستدعي المؤشر المُعاد، ويجب عليه تحريره.

٢.٣.٤ نمط التنظيف المحدود بالنطاق

```
int *arr = malloc(5 * sizeof *arr);
if (!arr) {
    return 1;
}

/* use arr */

free(arr);
```

٣.٣.٤ لماذا لا تحتوي C على مؤشرات ذكية

- لا توجد مُدمّرات
- لا يوجد تتبع تلقائي لأعمار الكائنات
- لا يوجد نموذج ملكية على مستوى اللغة

إن محاولة محاكاة المؤشرات الذكية في C++ داخل C تُدخل تعقيداً دون دعم لغوي حقيقي.

٤.٤ الخلاصة

تُعد المؤشرات أساس نموذج الذاكرة في C. ويتطلب الاستخدام الصحيح لها فهم أعمار الكائنات، ونطاقات العناوين الصالحة، وقواعد حسابات المؤشرات. يوفر التخصيص الديناميكي مرونة كبيرة، لكنه يتطلب انضباطًا صارمًا. يحافظ C23 على هذه المبادئ، مفضّلاً الملكية الصريحة والسلوك القابل للتنبؤ على الأتمتة.

الفصل ٥: المصفوفات والسلاسل النصية

١.٥ العمل مع المصفوفات

تُعد المصفوفات من أبسط وأهم هياكل البيانات في لغة C. فهي تمثل تسلسلاً ثابت الحجم من كائنات من النوع نفسه، مخزّنة في ذاكرة متجاورة.
ويتطلب الفهم الصحيح للمصفوفات فهم علاقتها بالذاكرة، وتحللها إلى مؤشرات (pointer decay)، وأعمار الكائنات.

١.١.٥ ما هي المصفوفة؟

المصفوفة هي كائن تجميعي (aggregate) يتكون من تسلسل متجاور من عناصر من النوع نفسه.

- يشغل كل عنصر موقعًا متجاورًا في الذاكرة
- المصفوفة نفسها ليست مؤشرًا
- للمصفوفة حجم ثابت يُحدد عند إنشائها

٢.١.٥ تصريح المصفوفات

الصياغة

```
type name[size];
```

يجب أن يكون حجم المصفوفة:

- تعبيرًا ثابتًا (للتخزين الساكن)
- أو قيمة وقت التشغيل (مصفوفات بطول متغير VLA، للتخزين التلقائي فقط)

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

٣.١.٥ تهيئة المصفوفات

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

التهيئة الجزئية

تُهيّأ العناصر غير المصرّح بها بالقيمة الصفرية.

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

استنتاج الحجم

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

٤.١.٥ الوصول إلى العناصر وتعديلها

```
numbers[0] = 42;
printf("%d\n", numbers[0]);
```

تبدأ فهارس المصفوفات من الصفر. ولا يتم إجراء أي فحص للحدود.

٥.١.٥ اجتياز المصفوفة

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

٦.١.٥ المصفوفات كمعاملات للدوال

عند تمرير المصفوفات إلى الدوال، تتحلل إلى مؤشرات لأول عنصر فيها.

```
void process(int arr[], size_t n);
```

داخل الدالة، يكون arr مؤشرًا وليس مصفوفة.

٧.١.٥ المصفوفات متعددة الأبعاد

المصفوفات متعددة الأبعاد هي مصفوفات تكون عناصرها مصفوفات أخرى.

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

يكون تخطيط الذاكرة على نمط الصفوف (row-major).

٨.١.٥ تمرير المصفوفات متعددة الأبعاد

يجب تحديد جميع الأبعاد ما عدا البعد الأول.

```
void process(int m[][3]);
```

٩.١.٥ أخطاء شائعة في المصفوفات

الوصول خارج الحدود

```
int a[5];
a[5] = 10;  // Undefined behavior
```

مصفوفات غير مهيأة

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

١٠.١.٥ أفضل الممارسات للمصفوفات

- استخدام ثوابت مسماة للأحجام
- تمرير الحجم صراحةً إلى الدوال
- تفضيل size_t للفهرسة
- تجنّب VLA في الواجهات العامة

٢.٥ السلاسل النصية وتمثيلها

لا تحتوي لغة C على نوع مدمج للسلاسل النصية. وتُمثَّل السلسلة النصية كتسلسل من المحارف ينتهي بمحرف الإنهاء '\٥'.

١.٢.٥ كائنات السلاسل النصية

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

ينشئ هذا مصفوفة قابلة للتعديل.

```
const char *name = "John";
```

يشير هذا إلى literal نصي، ولا يجوز تعديله.

٢.٢.٥ محرف الإنهاء الصفري

يجب أن تنتهي جميع سلاسل C بمحرف صفري. يؤدي غياب هذا المحرف إلى سلوك غير معرّف.

٣.٢.٥ الوصول إلى السلاسل النصية وتعديلها

```
char name[] = "John";
name[0] = 'j';
```

٤.٢.٥ دوال السلاسل النصية الشائعة

تعمل جميع دوال السلاسل النصية على تسلسلات بايت منتهية بمحرف صفري.

الطول

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

النسخ

```
strcpy(dest, src);  // Unsafe if dest is too small
```

مهم:
الدالة strncpy ليست بديلاً آمنًا عن strcpy، إذ لا تضمن إنهاء السلسلة بمحرف صفري.

الربط (Concatenation)

```
strcat(dest, src);  // Requires sufficient space
```

المقارنة

```
strcmp(a, b);
```

٥.٢.٥ البحث

```
strchr(s, 'x');
strstr(s, "abc");
```

٦.٢.٥ التجزئة (Tokenization)

```
strtok(str, " ,");
```

تحذير:

- تعدّل الدخل الأصلي
- تستخدم حالة عامة مخفية
- غير آمنة للخيوط

٧.٢.٥ تحويل السلاسل النصية إلى أعداد

الدوال القديمة (يُفضّل تجنّبها)

- atoi
- atof

لا توفر هذه الدوال أي إبلاغ عن الأخطاء.

الدوال الموصى بها

```
long v = strtol(s, &end, 10);
double d = strtod(s, &end);
```

٨.٢.٥ أخطاء شائعة في السلاسل النصية

فيض المخزن (Buffer Overflow)

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

غياب محرف الإنهاء

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

٩.٢.٥ أفضل الممارسات للسلاسل النصية

- تتبع حجم المخازن دائمًا
- تفضيل فحوصات الحدود الصريحة
- التعامل مع literals النصية كقراءة فقط
- استخدام strtod/strtol للتحليل

٣.٥ الخلاصة

المصفوفات في C هي كائنات متجاورة ثابتة الحجم، وليست مؤشرات. أما السلاسل النصية فهي مصفوفات من المحارف منتهية ببايت صفري. ويتطلب الاستخدام الصحيح التزامًا صارمًا بالحدود، وقواعد العمر، وعقود المكتبة القياسية. ويحافظ C23 على هذه الدلالات منخفضة المستوى، مفضّلاً التحكم الصريح وقابلية التنبؤ على أتمتة الأمان.

الفصل ٦: البُنى والاتحادات

١.٦ تعريف البُنى واستخدامها

تسمح البُنى (structures) بتجميع كائنات مترابطة من أنواع مختلفة ضمن كائن تجميعي واحد. وهي أساسية في نمذجة بيانات العالم الحقيقي وبناء تجريدات أعلى مستوى في لغة C.

١.١.٦ ما هي البنية؟

البنية هي نوع بيانات تجميعي يتكوّن من أعضاء مسماة، لكل عضو نوعه ومساحته التخزينية الخاصة.

- تشغل الأعضاء مواقع ذاكرة مميزة
- البنية نفسها كائن واحد
- يؤثر ترتيب الأعضاء في تخطيط الذاكرة

٢.١.٦ تعريف بنية

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

```
    float height;
};
```

٣.١.٦ تصريح كائنات من بنية

```
struct Person p;
```

يمكن أيضًا تعريف البنية والتصريح عن كائناتها في آنٍ واحد.

```
struct Point {
    int x;
    int y;
} p1, p2;
```

٤.١.٦ تهيئة البُنى

التهيئة الموضعية

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

التهيئة المسماة

تتوفر المهيئات المسماة منذ C99 وما تزال صالحة في C23.

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

٥.١.٦ الوصول إلى الأعضاء وتعديلهم

```
p.age = 31;
printf("%s %d\n", p.name, p.age);
```

٦.١.٦ البُنى المتداخلة

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

struct Person {
    char name[32];
    struct Address address;
};
```

٧.١.٦ مصفوفات من البُنى

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

٨.١.٦ أخطاء شائعة في البُنى

كائنات غير مهيأة

```
struct Person p;
```

```
printf("%s\n", p.name);  // Undefined behavior
```

٩.١.٦ أفضل الممارسات

- تهيئة جميع كائنات البُنى
- الحفاظ على تماسك البُنى وعدم تضخيمها
- تفضيل تمرير المؤشرات عند التعامل مع بُنى كبيرة
- استخدام const عندما لا يُقصد التعديل

٢.٦ مؤشرات إلى البُنى

تمكّن المؤشرات إلى البُنى الوصول غير المباشر، والتمرير الفعّال للمعاملات، والتخصيص الديناميكي.

١.٢.٦ التصريح والتهيئة

```
struct Person p = {"John", 30, 5.9f};
struct Person *ptr = &p;
```

٢.٢.٦ الوصول إلى الأعضاء عبر المؤشرات

```
printf("%s\n", ptr->name);
ptr->age++;
```

٣.٢.٦ تمرير البُنى إلى الدوال

بالقيمة

```
void print(struct Person p);
```

يؤدي ذلك إلى نسخ البنية.

بالمؤشر

```
void print(const struct Person *p);
```

يتجنب النسخ ويسمح بالوصول المتحكم به.

٤.٢.٦ التخصيص الديناميكي للبُنى

```
struct Person *p = malloc(sizeof *p);
if (!p) return 1;
```

لا تقم أبداً بتحويل قيمة الإرجاع من malloc في C.

٥.٢.٦ أخطاء شائعة في المؤشرات

مؤشر معلّق

```
struct Person *p = malloc(sizeof *p);
free(p);
p->age = 10;  // Undefined behavior
```

تسرّب الذاكرة

```
struct Person *p = malloc(sizeof *p);
/* missing free */
```

٣.٦ الاتحادات وتطبيقاتها

١.٣.٦ ما هو الاتحاد؟

الاتحاد (union) يضع عدة أعضاء في نفس موقع الذاكرة. ويُعد عضو واحد فقط نشطًا في أي وقت.

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

٢.٣.٦ حجم الاتحاد وتخطيطه

- يساوي الحجم أكبر عضو
- تشترك جميع الأعضاء في نفس العنوان

٣.٣.٦ التهيئة

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

٦.٣.٤ قواعد الوصول

لا يجوز قراءة سوى العضو الذي كُتب إليه مؤخرًا بشكل آمن، باستثناء حالات محدودة معرّفة بالتنفيذ.

٦.٣.٥ الاتحادات الموسومة (سجلات متغيّرة)

```
struct Variant {
    enum { INT, FLOAT, STRING } tag;
    union {
        int i;
        float f;
        char str[20];
    } data;
};
```

٦.٣.٦ مزالق الاتحادات

الوصول غير الصحيح إلى عضو

```
union Data d;
d.i = 10;
printf("%f\n", d.f);  // Undefined behavior
```

٦.٣.٧ أفضل الممارسات للاتحادات

- تتبع العضو النشط صراحةً
- تجنب type punning إلا عند الحاجة التي يفرضها ABI أو العتاد

- تفضيل الاتحادات الموسومة

٤.٦ أعضاء المصفوفات المرنة

تسمح أعضاء المصفوفات المرنة بتخزين لاحق متغير الحجم.

```
struct Buffer {
    size_t size;
    unsigned char data[];
};
```

١.٤.٦ التخصيص

```
struct Buffer *b =
    malloc(sizeof *b + n * sizeof b->data[0]);
```

٢.٤.٦ القواعد

- يجب أن يكون العضو الأخير
- لا يمكن نسخ البنية بأمان
- يتطلب تخصيصًا ديناميكيًا

٥.٦ الخلاصة

توفر البُنى نمذجةً تجميعية للبيانات بدلالات قيمة. وتمكّن المؤشرات إلى البُنى من الكفاءة والتحكم في العمر الديناميكي. أما الاتحادات فتوفر تراكبًا للذاكرة ويجب استخدامها بانضباط.

ويحافظ C23 على هذه الآليات دون إضافة أمان تلقائي، واضعًا كامل المسؤولية على عاتق المبرمج.

الفصل ٧: التعامل مع الملفات

١.٧ فتح الملفات وإغلاقها

يُمكّن التعامل مع الملفات البرامج من القراءة من التخزين الدائم والكتابة إليه. في لغة C، يتم إدخال/إخراج الملفات عبر مكتبة الإدخال والإخراج القياسية (stdio.h)، التي توفر وصولاً محمولاً ومُخزَّنًا (buffered) إلى الملفات من خلال تجريد FILE.

١.١.٧ ما هو التعامل مع الملفات؟

يشير التعامل مع الملفات إلى تنفيذ عمليات الإدخال والإخراج على ملفات مخزّنة ضمن نظام الملفات. وعلى عكس الذاكرة، تبقى الملفات بعد انتهاء البرنامج، وقد تُشارك بين عدة برامج.
توفر C التعامل مع الملفات عبر دوال المكتبة القياسية مثل:

- fopen، fclose
- fread، fwrite
- fgets، fputs
- fprintf، fscanf

٢.١.٧ فتح ملف

تُفتح الملفات باستخدام fopen، التي تُعيد مؤشرًا إلى كائن FILE.

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

إذا فشلت العملية، تُعاد القيمة NULL.

٣.١.٧ أنماط فتح الملفات

النمط	الوصف
"r"	قراءة ملف موجود
"w"	كتابة مع اقتطاع الملف أو إنشائه
"a"	الإلحاق، مع الإنشاء عند الحاجة
"r+"	قراءة وكتابة (يجب أن يكون الملف موجوداً)
"w+"	قراءة وكتابة مع الاقتطاع أو الإنشاء
"a+"	قراءة وإلحاق
"b"	مُعدِّل النمط الثنائي (مثل "rb")

ملاحظة: في أنظمة POSIX، لا يوجد فرق بين النمطين النصي والثنائي. أما في Windows، فيعطّل النمط الثنائي تحويل محارف نهاية السطر.

٤.١.٧ إغلاق ملف

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

يؤدي إغلاق الملف إلى تفريغ المخرجات المُخزَّنة وتحرير الموارد المرتبطة. وقد يؤدي عدم إغلاق الملفات إلى فقدان البيانات.

٥.١.٧ أفضل الممارسات

- التحقق دائمًا من fopen مقابل NULL
- إغلاق كل ملف تم فتحه
- إغلاق الملفات في جميع مسارات الخطأ

٢.٧ قراءة الملفات وكتابتها

تُميّز C بين:

- إدخال/إخراج مُنسّق (نصي)
- إدخال/إخراج غير مُنسّق (ثنائي)

١.٢.٧ الإدخال/الإخراج غير المُنسّق

fread

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

تقرأ بايتات خام دون تفسير.

fwrite

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

تكتب بايتات خام تمامًا كما هي مخزّنة في الذاكرة.

٢.٢.٧ الإدخال/الإخراج المُنسّق

fprintf

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

تكتب نصًا مُنسّقًا.

fscanf

تحذير:
الدالة fscanf هشّة وحساسة لاختلافات التنسيق.

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

عند الحاجة إلى المتانة، يُفضّل الإدخال المعتمد على الأسطر مع التحليل.

٣.٢.٧ الإدخال المعتمد على الأسطر

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

تُعد fgets أكثر دوال المكتبة القياسية أمانًا لقراءة الأسطر النصية.

٣.٧ تموضع الملف

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

تُستخدم للوصول العشوائي، خاصةً في الملفات الثنائية.

٤.٧ معالجة الأخطاء في عمليات الملفات

تعتمد عمليات إدخال/إخراج الملفات على موارد خارجية، ويجب دائمًا التحقق من الأخطاء.

١.٤.٧ قيم الإرجاع

- fopen تُعيد NULL عند الفشل
- fclose تُعيد EOF عند الفشل
- fwrite/fread عدد عناصر أقل عند الخطأ أو نهاية الملف

٢.٤.٧ errno

تكون errno ذات معنى فقط مباشرةً بعد حدوث الفشل.

```
#include <errno.h>
#include <string.h>

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

٣.٤.٧ feof و ferror

إن EOF ليست محرفًا.

```
if (fread(buf, size, count, file) < count) {
    if (feof(file)) {
        /* end of file */
```

```
    } else if (ferror(file)) {
        /* I/O error */
    }
}
```

٥.٧ الملفات الثنائية

١.٥.٧ الملفات الثنائية مقابل النصية

تُخزّن الملفات الثنائية بايتات خام دون تفسير. وهي:

- أسرع
- أكثر إحكاماً
- غير قابلة للقراءة البشرية

٢.٥.٧ إدخال/إخراج البُنى الثنائية (تحذير قابلية النقل)

```
typedef struct {
    int id;
    float salary;
} Record;
```

تحذير:

- قد يختلف الحشو (padding)
- قد تختلف ترتيبية البايتات (endianness)
- اختلافات ABI تُفسد قابلية النقل

يكون إدخال/إخراج البُنى الثنائية آمنًا فقط عندما يشترك القارئ والكاتب في المعمارية نفسها وABI ذاته.

٦.٧ أفضل الممارسات للإدخال/إخراج الملفات

- تفضيل fgets مع التحليل للنصوص
- استخدام الإدخال/الإخراج الثنائي فقط لتنسيقات مُعرّفة بدقة
- التحقق دائمًا من قيم الإرجاع
- توثيق تنسيقات الملفات صراحةً
- إغلاق الملفات في جميع مسارات الخروج

٧.٧ الخلاصة

يُعد التعامل مع الملفات في C صريحًا ومنخفض المستوى وقويًا. فهو يوفر إدخال/إخراجًا محمولاً ومُخزَّنًا عبر تجريد FILE، مع وضع كامل المسؤولية على عاتق المبرمج. ويتطلب إتقان إدخال/إخراج الملفات فهم آليات التخزين المؤقت، ومعالجة الأخطاء، وقيود قابلية النقل—خصوصًا عند التعامل مع البيانات الثنائية.

الفصل ٨: البرمجة منخفضة المستوى

١.٨ فهم البرمجة منخفضة المستوى

تُشكّل البرمجة منخفضة المستوى الأساس لبرمجيات الأنظمة والتطبيقات الحسّاسة للأداء. فهي تتيح التفاعل المباشر مع العتاد وأدوات نظام التشغيل الأولية، كاشفةً الذاكرة والسجلات وسلوك التنفيذ بأدنى قدر من التجريد. يقدّم هذا الفصل المبادئ الجوهرية للبرمجة منخفضة المستوى ويشرح لماذا يُعدّ إتقان هذه المفاهيم أمرًا أساسيًا لمبرمجي الأنظمة.

١.١.٨ ما هي البرمجة منخفضة المستوى؟

تشير البرمجة منخفضة المستوى إلى كتابة برمجيات تعمل قرب مستوى الآلة، مع تحكم صريح في تخطيط الذاكرة، وتمثيل البيانات، وتدفّق التنفيذ. وتوفّر لغات مثل C وAssembly حدًّا أدنى من التجريد فوق العتاد الأساسي، مما يسمح بالتفاعل المباشر مع سجلات المعالج، وعناوين الذاكرة، وواجهات النظام.
تشمل الخصائص الأساسية ما يلي:

- التفاعل المباشر مع العتاد: الوصول إلى السجلات، وإدخال/إخراج مُعيَّن بالذاكرة (MMIO)، وتعليمات المعالج.
- إدارة الذاكرة الصريحة: تخصيص وإلغاء تخصيص الذاكرة يدويًا مع تحمّل كامل المسؤولية عن الصحة.

- حدّ أدنى من التجريد: ضمانات تشغيلية وشبكات أمان أقل مقارنة باللغات عالية المستوى.
- أداء قابل للتنبؤ: تحكم دقيق في كلفة التنفيذ واستهلاك الموارد.

٢.١.٨ البرمجة منخفضة المستوى مقابل عالية المستوى

الجانب	البرمجة منخفضة المستوى	البرمجة عالية المستوى
التجريد	ضئيل ومتمركز حول العتاد	مرتفع ومدار وقت التشغيل
التحكم بالذاكرة	يدوي وصريح	تلقائي ومدار
الأداء	قابل للتنبؤ بدرجة عالية	مجرّد وغير مباشر
قابلية النقل	معتمد على المعمارية	مستقل عن المنصّة
مخاطر الأخطاء	مرتفعة دون انضباط	مخفّضة بفحوصات وقت التشغيل

٣.١.٨ أهمية البرمجة منخفضة المستوى

تُعد البرمجة منخفضة المستوى أساسيةً من أجل:

- أنظمة التشغيل والنوى
- مشغلات الأجهزة والبرمجيات الثابتة
- الأنظمة المضمنة وأنظمة الزمن الحقيقي
- المترجمات وبيئات التشغيل والآلات الافتراضية
- تحليل الأداء وأبحاث الأمن

كما يوضّح فهم السلوك منخفض المستوى كيفية عمل اللغات عالية المستوى داخليًا.

٢.٨ الوصول إلى العتاد باستخدام المؤشرات

يُجرى الوصول إلى الأجهزة عادةً عبر إدخال/إخراج مُعيَّن بالذاكرة (MMIO)، حيث تُعرَض سجلات الوحدات الطرفية عند عناوين ذاكرة ثابتة. في C، توفّر المؤشرات آلية التفاعل مع هذه العناوين.

١.٢.٨ إدخال/إخراج مُعيَّن بالذاكرة

مهم: تفترض الأمثلة التالية بيئةً مُتحكَّمًا بها مثل البرمجيات الثابتة دون نظام تشغيل (bare-metal) أو الشيفرة على مستوى النواة. إن الوصول إلى عناوين عشوائية في برامج مساحة المستخدم يُعد سلوكًا غير معرّف.

```
#define GPIO_DATA (*(volatile unsigned int *)0x1000)

void toggle_led(void) {
    GPIO_DATA ^= 0x01;
}
```

يمنع المُؤهِّل volatile المُصرِّف من تحسين عمليات القراءة والكتابة المتكررة، وهو أمرٌ جوهري للوصول إلى العتاد.

٢.٢.٨ الكلمة المفتاحية volatile

```
volatile unsigned int *timer_reg = (volatile unsigned int *)0x2000;

void wait_for_event(void) {
    while (*timer_reg == 0) {
        /* busy wait */
    }
}
```

بدون volatile قد يُخزّن المُصرِّف القيمة قانونيًا ويحوّل الحلقة إلى حلقة لا نهائية.

٣.٢.٨ حساب المؤشرات لكتل السجلات

```
#define UART_BASE 0x4000

#define UART_DATA   (*(volatile unsigned int *)(UART_BASE + 0x00))
#define UART_STATUS (*(volatile unsigned int *)(UART_BASE + 0x04))

void uart_send(char c) {
    while ((UART_STATUS & 0x01) == 0) { }
    UART_DATA = (unsigned int)c;
}
```

٣.٨ استخدام التجميع المضمَّن في C

يسمح التجميع المضمَّن بإدراج تعليمات خاصة بالمعمارية داخل شيفرة C. ويجب استخدامه بانضباط صارم لأنه يتجاوز كثيرًا من ضمانات المُصرِّف.

١.٣.٨ الصياغة العامة

```
asm volatile (
    "assembly instructions"
    : output_operands
    : input_operands
    : clobbered_registers
);
```

٢.٣.٨ مثال: حساب باستخدام السجلات

```
int add(int a, int b) {
    int result;
    asm volatile (
        "addl %1, %2\n\t"
        "movl %2, %0"
        : "=r"(result)
        : "r"(a), "r"(b)
        :
    );
    return result;
}
```

٣.٣.٨ قراءة عدّاد الطابع الزمني (x86)

ملاحظة معمارية: هذا المثال صالح فقط لمعالجات عائلة x86.

```
#include <stdint.h>

uint64_t read_tsc(void) {
    uint32_t lo, hi;
    asm volatile (
        "rdtsc"
        : "=a"(lo), "=d"(hi)
        :
        :
    );
    return ((uint64_t)hi << 32) | lo;
}
```

٤.٣.٨ استدعاءات النظام عبر التجميع المضمَّن (لينكس x86، 32-بت)

نطاق التطبيق: ينطبق هذا المثال فقط على لينكس 32-بت باستخدام int 0x80.

```
void write_stdout(const char *msg, unsigned len) {
    asm volatile (
        "movl $4, %%eax\n\t"
        "movl $1, %%ebx\n\t"
        "movl %0, %%ecx\n\t"
        "movl %1, %%edx\n\t"
        "int $0x80"
        :
        : "r"(msg), "r"(len)
        : "%eax", "%ebx", "%ecx", "%edx"
    );
}
```

٤.٨ التلاعب المباشر بالذاكرة

١.٤.٨ المؤشرات والعناوين

```
int x = 10;
int *p = &x;
*p = 20;
```

تمثّل المؤشرات عناوين ذاكرة، لا كائنات ولا ملكية. ويؤدي الاستخدام الخاطئ مباشرةً إلى سلوك غير معرّف.

٢.٤.٨ التخصيص الديناميكي للذاكرة

```c
int *arr = malloc(5 * sizeof(int));
if (!arr) {
    /* handle error */
}
free(arr);
```

٣.٤.٨ دوال التلاعب بالذاكرة

```c
#include <string.h>

char buf[16];
memset(buf, 0, sizeof(buf));
```

٤.٤.٨ مُخصِّص كتل ثابتة مخصّص

```c
#define BLOCKS 8
#define SIZE   128

static unsigned char pool[BLOCKS][SIZE];
static int used[BLOCKS];

void *alloc_block(void) {
    for (int i = 0; i < BLOCKS; ++i) {
        if (!used[i]) {
            used[i] = 1;
            return pool[i];
```

```
        }
    }
    return NULL;
}
```

٥.٨ الخاتمة

تكشف البرمجة منخفضة المستوى الكلفة الحقيقية وسلوك البرمجيات. فكل مؤشر، وكل تعليمة، وكل وصول إلى الذاكرة يحمل مسؤولية. ومن خلال إتقان هذه الأسس، لا يكتسب المبرمجون الأداء فحسب، بل الفهم — وهو أثمن مورد في تصميم الأنظمة.

الفصل ٩: التفاعل مع أنظمة التشغيل

١.٩ استدعاءات النظام في C

تُشكّل استدعاءات النظام (system calls) الحدّ الأساسي بين برامج مساحة المستخدم ونواة نظام التشغيل. فهي تتيح للتطبيقات طلب خدمات امتيازية مثل الوصول إلى الملفات، وإنشاء العمليات، وإدارة الذاكرة، والتواصل بين العمليات. يشرح هذا الفصل المفهوم العام لاستدعاءات النظام، وكيفية إتاحتها في برامج C، وكيفية استخدامها بأمان وبصورة صحيحة في برمجيات مستوى النظام.

١.١.٩ ما هي استدعاءات النظام؟

استدعاء النظام هو نقطة دخول مُتحكَّم بها إلى نواة نظام التشغيل. تعمل برامج مساحة المستخدم في وضعٍ مُقيَّد ولا يمكنها الوصول مباشرةً إلى العتاد أو إلى هياكل بيانات النواة. توفّر استدعاءات النظام آليةً آمنة ومُعرَّفة جيّدًا لطلب مثل هذه العمليات.
تُستخدم استدعاءات النظام من أجل:

- إدخال/إخراج الملفات والأجهزة
- إنشاء العمليات والتحكم بها
- إدارة الذاكرة الافتراضية

- التواصل بين العمليات
- الشبكات والمؤقّتات

٢.١.٩ الانتقال من وضع المستخدم إلى وضع النواة

عند تنفيذ استدعاء نظام، ينتقل التنفيذ من وضع المستخدم إلى وضع النواة. تختلف الآلية باختلاف المعمارية، لكن الخطوات المفهومية متشابهة:

١. تجهيز المعاملات وفقًا لـ ABI
٢. تنفيذ تعليمة خاصة تُحفّز الانتقال
٣. تتحقق النواة من المعاملات والصلاحيات
٤. تنفيذ الخدمة المطلوبة
٥. العودة إلى وضع المستخدم مع نتيجة

تشمل آليات الانتقال الشائعة:

- `syscall` (x86-64)
- `sysenter` (legacy x86)
- `int 0x80` (لينكس x86 الأقدم)

٣.١.٩ واجهات استدعاءات النظام في C

عمليًا، نادرًا ما تستدعي برامج C استدعاءات النظام مباشرةً. بدلاً من ذلك، تستخدم دوال تغليف (wrappers) من POSIX أو libc، والتي تتولى تفاصيل ABI، وترجمة الأخطاء، وقابلية النقل.
أمثلة على استدعاءات نظام شائعة مُتاحة عبر libc:

- open، read، write، close
- fork، exec، wait
- mmap، munmap
- pipe، socket

٤.١.٩ مثال: إدخال/إخراج الملفات عبر استدعاءات النظام

```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 الوصول الخام إلى الملفات دون إدخال/إخراج مُخزَّن (buffered I/O).

٥.١.٩ الاستدعاء المباشر لاستدعاءات النظام

في بيئات خاصة (بيئات تشغيل مصغّرة، محمّلات مخصّصة)، قد تُستدعى استدعاءات النظام مباشرةً باستخدام واجهة syscall.
ملاحظة نطاق: ينطبق هذا المثال على أنظمة لينكس فقط.

```
#include <unistd.h>
#include <sys/syscall.h>

int main(void) {
    const char msg[] = "Hello from syscall\n";
    syscall(SYS_write, STDOUT_FILENO, msg, sizeof(msg) - 1);
    return 0;
}
```

٦.١.٩ معالجة الأخطاء في استدعاءات النظام

تُشير استدعاءات النظام إلى الفشل عبر قيم الإرجاع والمتغيّر errno.

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

```
int main(void) {
    int fd = open("missing.txt", O_RDONLY);
    if (fd == -1) {
        perror("open failed");
        return 1;
    }
    return 0;
}
```

تحقق دائمًا من قيم الإرجاع قبل استخدام النتائج.

٢.٩ إدارة العمليات

تُعد إدارة العمليات مسؤوليةً أساسية لنظام التشغيل. وتُتيح الأنظمة الشبيهة بيونكس التحكم بالعمليات عبر مجموعة صغيرة لكنها قوية من استدعاءات النظام: fork، exec، وwait.

١.٢.٩ إنشاء العمليات باستخدام fork

تُنشئ fork عمليةً جديدة عبر نسخ العملية المُستدعية. ويحصل الابن على فضاء عناوين مستقل، يُنفَّذ باستخدام copy-on-write.

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

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
```

```
        printf("Child process\n");
    } else if (pid > 0) {
        printf("Parent process\n");
    } else {
        perror("fork");
    }
    return 0;
}
```

٢.٢.٩ استبدال البرنامج باستخدام exec

تستبدل عائلة exec صورة العملية الحالية ببرنامج جديد. ولا تُنشأ عملية جديدة.

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

int main(void) {
    execlp("ls", "ls", "-l", NULL);
    perror("exec failed");
    return 1;
}
```

إذا نجحت exec، فلن يعود التنفيذ أبدًا.

٣.٢.٩ مزامنة العمليات باستخدام wait

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        return 42;
    } else {
        int status;
        wait(&status);
        if (WIFEXITED(status)) {
            printf("Exit code: %d\n", WEXITSTATUS(status));
        }
    }
    return 0;
}
```

٣.٩ إدارة الذاكرة ونظام التشغيل

١.٣.٩ الذاكرة الافتراضية

تعمل كل عملية ضمن فضاء عناوين افتراضي خاص بها. ويقوم نظام التشغيل بربط العناوين الافتراضية بالذاكرة الفيزيائية باستخدام جداول الصفحات.
المفاهيم الأساسية:

- عزل فضاء العناوين الافتراضي
- الصفحات وأخطاء الصفحات
- التخصيص عند الطلب

- حماية الذاكرة

٢.٣.٩ التخصيص الديناميكي والنواة

تعتمد مخصّصات مساحة المستخدم في النهاية على آليات نواتية مثل brk أو mmap.

```
#include <stdlib.h>

int main(void) {
    int *p = malloc(1024 * sizeof(int));
    if (!p) return 1;
    free(p);
    return 0;
}
```

٣.٣.٩ حماية الذاكرة

يؤدي الوصول خارج حدود الذاكرة إلى سلوك غير معرّف وغالبًا ما تُنهيه النواة.

```
int *p = malloc(10 * sizeof(int));
p[10] = 5; /* invalid access */
```

٤.٩ صلاحيات الملفات والعمليات

١.٤.٩ نموذج الصلاحيات

تستخدم الأنظمة الشبيهة بيونكس نموذج صلاحيات قائمًا على:

- المالك
- المجموعة
- الآخرون

وتحدّد كل فئة صلاحيات القراءة والكتابة والتنفيذ.

٢.٤.٩ تغيير صلاحيات الملفات

```c
#include <sys/stat.h>

int main(void) {
    chmod("example.txt", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    return 0;
}
```

٣.٤.٩ هوية العملية

تملك العمليات مُعرّفات مستخدم ومجموعة حقيقية وفعّالة.

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

int main(void) {
    printf("UID=%d EUID=%d\n", getuid(), geteuid());
    return 0;
}
```

٤.٤.٩ بتّات الصلاحيات الخاصة

- Setuid
- Setgid
- Sticky bit

يجب استخدام هذه الميزات بحذر شديد نظرًا للمخاطر الأمنية.

٥.٩ الخاتمة

يُحدِّد التفاعل مع نظام التشغيل الحدّ الفاصل بين شيفرة التطبيق والآلة نفسها. إن فهم استدعاءات النظام، وإدارة العمليات، وعزل الذاكرة، ونماذج الصلاحيات يتيح للمبرمجين كتابة برمجيات منخفضة المستوى فعّالة وآمنة وصحيحة. ويُعدّ إتقان هذه المفاهيم أمرًا أساسيًا لبرمجة الأنظمة، وتصميم بيئات التشغيل، وتطوير أنظمة التشغيل.

الفصل ١٠: أساسيات تصميم المترجمات

١.١٠ مقدمة في تصميم المترجمات

يدرس تصميم المترجمات كيفية تحويل لغات البرمجة عالية المستوى إلى شيفرة آلة قابلة للتنفيذ. فالمترجم ليس مجرد أداة ترجمة، بل هو نظام مُنظَّم يقوم بالتحليل، والتحقق، والتحويل، والتحسين قبل التنفيذ. ويكشف فهم تصميم المترجمات كيف تُنفَّذ اللغات، وكيف ينشأ الأداء، وكيف تُفرض الصحة.

يقدّم هذا الفصل المفاهيم الأساسية، والمراحل، والبُنى الداخلية للمترجمات، مشكلاً الأساس للتطبيق العملي في الفصول اللاحقة.

١.١.١٠ ما هو المترجم؟

المترجم هو برنامج يحوّل الشيفرة المصدرية المكتوبة بلغة عالية المستوى إلى تمثيل أدنى مستوى مناسب للتنفيذ على آلة الهدف. وقد يكون الخرج شيفرة تجميع، أو شيفرة كائن، أو ملفًا تنفيذيًا، أو تمثيلاً وسيطًا (IR).

تشمل المسؤوليات الأساسية للمترجم ما يلي:

- تحويل بُنى اللغة إلى شكل مفهوم للآلة
- فرض الصحة النحوية والدلالية

- تحسين سلوك البرنامج
- دعم قابلية النقل عبر المعماريات

٢.١.١٠ لماذا ندرس تصميم المترجمات؟

يُعد تصميم المترجمات أساسيًا من أجل:

١. كتابة شيفرة واعية بالأداء

٢. تصميم لغات برمجة جديدة

٣. فهم السلوك غير المعرّف وتأثيرات التحسين

٤. تصحيح مشكلات التنفيذ منخفضة المستوى

٥. برمجة الأنظمة وتطوير سلاسل الأدوات

المبرمج الذي يفهم المترجمات يفهم نموذج التنفيذ الحقيقي للبرمجيات.

٢.١٠ بنية المترجم ومراحله

يُقسَّم المترجم تقليديًا إلى الواجهة الأمامية (front end) والواجهة الخلفية (back end)، ويربط بينهما تمثيل وسيط واحد أو أكثر.

١.٢.١٠ مراحل الواجهة الأمامية

تقوم الواجهة الأمامية بتحليل الشيفرة المصدرية وإنتاج تمثيل مُجرَّد ومستقل عن الآلة.

١. التحليل المعجمي

- يحوّل تدفّق المحارف إلى رموز (tokens)
- يزيل الفراغات والتعليقات

٢. التحليل النحوي

- يتحقق من البنية القواعدية
- يبني أشجار التحليل أو الأشجار المجردة AST

٣. التحليل الدلالي

- يفرض قواعد الأنواع
- يحلّ الأسماء والمجالات
- يكتشف الأخطاء الدلالية

٢.٢.١٠ مراحل الواجهة الخلفية

تُحوّل الواجهة الخلفية التمثيل الوسيط إلى شيفرة الهدف.

١. بناء التمثيل الوسيط

٢. التحسين

٣. توليد الشيفرة

٤. إخراج الشيفرة

٣.١٠ المكوّنات الأساسية للمترجم

١.٣.١٠ المحلّل المعجمي

يقوم المحلّل المعجمي بتجميع المحارف في رموز.

```
int x = 10;
```

ينتج الرموز التالية:

```
KEYWORD(int) IDENT(x) OP(=) INT_LITERAL(10) PUNCT(;)
```

٢.٣.١٠ المحلّل النحوي

يتحقق المُحلِّل من الصحة القواعدية ويبني البنية.

```
declaration
   type: int
   identifier: x
   initializer: 10
```

٣.٣.١٠ المحلّل الدلالي

يفرض التحليل الدلالي المعنى الصحيح:

- توافق الأنواع
- التصريح عن المتغيرات قبل الاستخدام
- صحة استدعاءات الدوال

٤.٣.١٠ التمثيل الوسيط

تقوم معظم المترجمات الحديثة بتحويل الشيفرة إلى تمثيل وسيط.

```
t1 = 10
x = t1
t2 = x + 5
y = t2
```

يفصل التمثيل الوسيط قواعد اللغة عن تفاصيل الآلة.

٥.٣.١٠ تحسين الشيفرة

تُحسّن التحسينات الكفاءة دون تغيير الدلالات.

```
int x = 10 + 5;
```

تصبح:

```
int x = 15;
```

٦.٣.١٠ توليد الشيفرة

يربط توليد الشيفرة التمثيل الوسيط بتعليمات الآلة.
ملاحظة معمارية: المثال التالي تجميعي تصوري شبيه بـ x86.

```
mov eax, 10
add eax, 5
mov [x], eax
```

٤.١٠ التحليل المعجمي

يحوّل التحليل المعجمي النص الخام إلى رموز.

١.٤.١٠ فئات الرموز

- الكلمات المفتاحية
- المعرّفات
- القيم الحرفية
- المعاملات
- علامات الترقيم

٢.٤.١٠ الآلات ذات الحالات المحدودة

تُنفَّذ المُحلِّلات المعجمية عادةً باستخدام آلات حالات محدودة.

- تمثل الحالات سياق المسح
- تعتمد الانتقالات على محارف الدخل
- تنتج حالات القبول الرموز

٣.٤.١٠ الأخطاء المعجمية

يجب الإبلاغ عن المحارف غير الصالحة أو القيم الحرفية المشوّهة والتعافي منها دون إنهاء الترجمة.

٥.١٠ التحليل النحوي

يتحقق التحليل النحوي من تسلسل الرموز باستخدام قواعد رسمية.

١.٥.١٠ القواعد الخالية من السياق

تُوصَف لغات البرمجة باستخدام قواعد خالية من السياق.

```
expr → expr + term | term
term → term * factor | factor
factor → number | ( expr )
```

٢.٥.١٠ أشجار التحليل مقابل الأشجار المجرّدة

تحافظ أشجار التحليل على بنية القواعد، بينما تُزيل الأشجار المجرّدة العُقد غير الضرورية.

```
  +
 / \
2   *
   / \
  3   4
```

٣.٥.١٠ تقنيات التحليل

- التحليل من الأعلى إلى الأسفل (recursive descent, LL)
- التحليل من الأسفل إلى الأعلى (shift-reduce, LR)

٤.٥.١٠ معالجة أخطاء النحو

يجب على المُحلِّلات اكتشاف الأخطاء والتعافي منها لمتابعة المعالجة.

٦.١٠ توليد الشيفرة والتحسين

١.٦.١٠ اختيار التعليمات

تُربط عمليات التمثيل الوسيط بتعليمات الآلة.

```
t1 = a + b
t2 = t1 * c
```

```
mov eax, [a]
add eax, [b]
imul eax, [c]
```

٢.٦.١٠ تخصيص السجلات

السجلات موارد نادرة.

- تلوين الرسوم البيانية
- المسح الخطي

يقلّل التخصيص الفعّال من حركة الذاكرة.

٣.٦.١٠ جدولة التعليمات

تُعاد ترتيب التعليمات لتعظيم استخدام خط أنابيب المعالج.

٤.٦.١٠ مستويات التحسين

- تحسين الثغرات الصغيرة (peephole)
- تحسين محلي
- تحسين شامل
- تحسين عبر الإجراءات

٥.٦.١٠ توليد الشيفرة المعتمد على LLVM

توفّر LLVM بنيةً قابلة لإعادة الاستخدام للتمثيل الوسيط، والتحسين، وتوليد الشيفرة.

```
% LLVM IR
% add i32 10, 20
```

٧.١٠ الخاتمة

يُعد تصميم المترجمات أساس التنفيذ البرمجي الحديث. فهو يربط نظرية اللغات، وبرمجة الأنظمة، والمعمارية، والتحسين. ويُمكّن إتقان أساسيات المترجمات المبرمجين من فهم سلوك الأداء، وقيود الصحة، والكلفة الحقيقية للتجريد. وقد وضع هذا الفصل الأساس المفهومي اللازم لبناء مترجمات ومفسّرات حقيقية.

الفصل ١١: موضوعات متقدمة في C23

١.١١ البرمجة متعددة الخيوط في C23

تسمح البرمجة متعددة الخيوط بتنفيذ عدة خيوط تنفيذ بشكل متزامن داخل عملية واحدة، مع مشاركة نفس مساحة العناوين. أُدخل هذا الدعم في معيار C11 وتم الحفاظ عليه دون تغيير في C23. لا تعيد C23 تعريف نموذج الخيوط، بل تستمر في تقديم واجهة برمجية قياسية وبسيطة مناسبة لبرامج متزامنة قابلة للنقل.

١.١.١١ مقدمة في البرمجة متعددة الخيوط

الخيط هو مسار تنفيذ واحد داخل العملية. تشترك الخيوط في الذاكرة، وموصِّفات الملفات، وحالة العملية، بينما يمتلك كل خيط سياق تنفيذ خاصًا به.
تشمل فوائد تعدد الخيوط:

- التنفيذ المتوازي على الأنظمة متعددة الأنوية
- تحسين الاستجابة
- مشاركة فعّالة للبيانات داخل الذاكرة

لكنها تُدخل أيضًا مخاطر مثل سباقات البيانات، وحالات الجمود (deadlocks)، والسلوك غير المعرّف عند انتهاك قواعد التزامن.

٢.١.١١ إنشاء الخيوط وإدارتها

يوفّر معيار ISO C دعمًا قياسيًا للخيوط عبر <threads.h>.

إنشاء الخيوط

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

int thread_function(void *arg) {
    int value = *(int *)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;
}
```

إنهاء الخيوط

تنتهي الخيوط بالعودة من دالة الدخول أو باستدعاء thrd_exit.

```
int thread_function(void *arg) {
    thrd_exit(0);
}
```

٣.١.١١ مزامنة الخيوط

تضمن المزامنة الوصول الصحيح إلى البيانات المشتركة.

الأقفال (Mutexes)

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

mtx_t mutex;
int shared = 0;

int worker(void *arg) {
    mtx_lock(&mutex);
    shared++;
    mtx_unlock(&mutex);
    return 0;
}

int main(void) {
    thrd_t t1, t2;
    mtx_init(&mutex, mtx_plain);

    thrd_create(&t1, worker, NULL);
    thrd_create(&t2, worker, NULL);
```

```
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    mtx_destroy(&mutex);
    return 0;
}
```

متغيرات الشرط

```
#include <threads.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);
    }
```

```
    mtx_unlock(&mutex);
    return 0;
}
```

٤.١.١١ التخزين المحلي للخيط

يوفّر التخزين المحلي للخيط نسخة مستقلة من المتغير لكل خيط.

```
_Thread_local int counter = 0;
```

يصل كل خيط إلى نسخة مستقلة من counter.

٥.١.١١ أفضل الممارسات للبرمجة متعددة الخيوط

- تقليل الحالة المشتركة القابلة للتغيير
- مزامنة جميع البيانات المشتركة
- تجنب الأقسام الحرجة الطويلة
- تفضيل تمرير الرسائل عند الإمكان
- الاختبار تحت حالات التنافس

٢.١١ الشبكات باستخدام المقابس

واجهات الشبكات ليست جزءاً من **ISO C**. تُقدَّم برمجة المقابس عبر POSIX وامتدادات خاصة بالمنصات. تنطبق الأمثلة التالية على الأنظمة الشبيهة بـ Unix.

١.٢.١١ أساسيات المقابس

تمثل المقابس نقاط اتصال تُعرَّف بعنوان ومنفذ.

- مقابس تيارية (TCP)
- مقابس رزمية (UDP)

٢.٢.١١ إنشاء مقبس

```
#include <sys/socket.h>

int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

٣.٢.١١ الربط والاستماع

```
#include <netinet/in.h>
#include <string.h>

struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
addr.sin_addr.s_addr = INADDR_ANY;
```

٤.٢.١١ قبول الاتصالات

```
int client = accept(sockfd, NULL, NULL);
```

١١.٢.٥ إرسال واستقبال البيانات

```
send(client, msg, len, 0);
recv(client, buf, sizeof(buf), 0);
```

١١.٢.٦ أفضل الممارسات لبرمجة المقابس

- فحص جميع قيم الإرجاع
- التعامل مع الإرسال والاستقبال الجزئي
- استخدام الإدخال/الإخراج غير الحاجز عند الحاجة
- إغلاق المقابس فور الانتهاء

١١.٣ معالجة الإشارات

الإشارات إشعارات غير متزامنة تُسلَّم إلى العملية. تُعرَّف معالجة الإشارات في ISO C، مع تحكم موسّع يوفّره POSIX.

١١.٣.١ دلالات الإشارات

تُقاطع الإشارات تدفق التنفيذ الطبيعي وقد تحدث في أي وقت. لا يُسمح داخل معالِجات الإشارات إلا بمجموعة محدودة من العمليات الآمنة.

١١.٣.٢ المعالجة الأساسية للإشارات

```
#include <signal.h>
```

```c
void handler(int sig) {
    (void)sig;
}

int main(void) {
    signal(SIGINT, handler);
    for (;;) { }
}
```

١١.٣.٣ قواعد أمان الإشارات

داخل معالِج الإشارة:

- لا تستدعِ دوال غير آمنة للإشارات
- تجنب التخصيص الديناميكي
- تجنب بدائيات القفل

يؤدي خرق هذه القواعد إلى سلوك غير معرّف.

١١.٣.٤ حجب الإشارات

```c
#include <signal.h>

sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigprocmask(SIG_BLOCK, &set, NULL);
```

٤.١١ التواصل بين العمليات (IPC)

يسمح IPC بالتنسيق بين عمليات مستقلة. توفّر آليات IPC عبر نظام التشغيل، وليس عبر ISO C.

١.٤.١١ الأنابيب

```c
#include <unistd.h>

int fd[2];
pipe(fd);
```

٢.٤.١١ الأنابيب المسماة (FIFOs)

```c
#include <sys/stat.h>

mkfifo("/tmp/myfifo", 0666);
```

٣.٤.١١ طوابير الرسائل

```c
#include <mqueue.h>

mqd_t mq = mq_open("/queue", O_CREAT | O_RDWR, 0666, NULL);
```

٤.٤.١١ الذاكرة المشتركة

```c
#include <sys/shm.h>

int id = shmget(IPC_PRIVATE, 1024, IPC_CREAT | 0666);
```

٥.٤.١١ أفضل الممارسات لـ IPC

- اختيار أبسط آلية تؤدي الغرض
- مزامنة الوصول إلى الذاكرة المشتركة
- تنظيف موارد IPC
- تعريف بروتوكولات تواصل واضحة

٥.١١ الخاتمة

تجمع البرمجة المتقدمة في C23 بين خصائص اللغة القياسية ومرافق النظام التي توفّرها المنصة. وبينما تحافظ C23 على نواة بسيطة وقابلة للنقل، تتطلب برمجة الأنظمة الواقعية استخداماً منضبطاً للخيوط، والإشارات، والشبكات، وIPC. ويتطلّب إتقان هذه الموضوعات فهماً دقيقاً لنماذج التنفيذ، والمزامنة، وحدود السلوك غير المعرّف.

الفصل ١٢: الأمن والتحسين

١.١٢ أفضل الممارسات للبرمجة الآمنة

البرمجة الآمنة هي منهجية كتابة برامج تبقى صحيحة، متينة، وقابلة للتنبؤ حتى في وجود مدخلات خاطئة، استخدام غير متوقع، أو بيئات عدائية. في اللغات منخفضة المستوى مثل C23، لا ينفصل الأمن عن الصحة: فكثير من الثغرات الأمنية هي نتيجة مباشرة للسلوك غير المعرّف.

١.١.١٢ مبادئ البرمجة الآمنة

تُبنى برامج C الآمنة على المبادئ التالية:

- السلوك المعرّف فقط: تجنّب جميع أشكال السلوك غير المعرّف والسلوك المعتمد على التنفيذ.
- الدفاع متعدد الطبقات: الجمع بين الانضباط اللغوي، وفحوصات المترجم، والتحقق وقت التشغيل.
- أقل امتياز ممكن: تقييد الوصول إلى الموارد والقدرات.
- الفشل الآمن: اكتشاف الأخطاء مبكرًا وإنهاء التنفيذ بأمان.
- الملكية الصريحة: تحديد من يملك كل مورد ومن المسؤول عن تحريره.

٢.١٢ تجنّب الثغرات الشائعة

١.٢.١٢ تجاوز سعة المخازن

يحدث تجاوز سعة المخزن عندما يكتب البرنامج خارج حدود كائن مُخصّص. في C، يؤدي ذلك مباشرةً إلى سلوك غير معرّف وقد يُفسد تدفق التحكم أو البيانات أو الحدود الأمنية.

الأسباب الشائعة

- عمليات نسخ دون فحص
- حسابات حجم غير صحيحة
- الثقة بمدخلات خارجية

توضيح مهم

الدوال مثل strncpy ليست آمنة بطبيعتها. فهي تحدّ فقط عدد البايتات المنسوخة ولا تضمن الإنهاء الصفري أو الصحة.

إستراتيجية التخفيف الصحيحة

- تتبّع أحجام المخازن صراحةً
- استخدام أنماط نسخ محدودة
- فرض الإنهاء الصفري يدويًا
- رفض المدخلات كبيرة الحجم مبكرًا

مثال صحيح

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

void safe_copy(char *dst, size_t dst_size, const char *src) {
    if (dst_size == 0) return;
    size_t n = strlen(src);
    if (n >= dst_size) n = dst_size - 1;
    memcpy(dst, src, n);
    dst[n] = '\0';
}

int main(void) {
    char buffer[10];
    safe_copy(buffer, sizeof(buffer), "This string is too long");
    printf("%s\n", buffer);
    return 0;
}
```

٢.٢.١٢ المؤشرات المتدلية

يشير المؤشر المتدلي إلى ذاكرة لم تعد مملوكة للبرنامج. الوصول إليها يؤدي إلى سلوك غير معرّف.

الأسباب النموذجية

- تحرير الذاكرة مع بقاء مراجع لها
- إرجاع عناوين متغيرات محلية

- أخطاء التحرير المزدوج

قاعدة أساسية

تعيين المؤشر المحلي إلى NULL لا يؤثر على النسخ الأخرى منه. يجب أن تكون الملكية صريحة.

نمط صحيح

```
#include <stdlib.h>

void free_and_clear(int **p) {
    if (*p) {
        free(*p);
        *p = NULL;
    }
}

int main(void) {
    int *data = malloc(10 * sizeof(int));
    free_and_clear(&data);
    return 0;
}
```

٣.١٢ التحقق من المدخلات

يجب التعامل مع جميع المدخلات الخارجية على أنها عدائية حتى يثبت العكس.

```
#include <ctype.h>
```

```
int is_alnum_string(const char *s) {
    for (; *s; ++s) {
        if (!isalnum((unsigned char)*s))
            return 0;
    }
    return 1;
}
```

يجب أن يسبق التحقق أي تحليل أو تخصيص أو تنفيذ.

٤.١٢ الإعدادات الافتراضية الآمنة

تضعف الإعدادات الافتراضية المتساهلة مستوى الأمان.

- أذونات ملفات مقيّدة
- تعطيل ميزات التصحيح
- واجهات مصدّرة بالحد الأدنى

```
#include <sys/stat.h>

chmod("file.txt", S_IRUSR | S_IWUSR);
```

٥.١٢ معالجة الأخطاء

يجب ألا تكشف رسائل الخطأ عن الحالة الداخلية.

```
if (!file) {
    fprintf(stderr, "Operation failed\n");
    return 1;
}
```

٦.١٢ تقنيات تحسين الشيفرة

يُحسّن التحسين الأداء دون تغيير الدلالات. تأتي الصحة أولاً.

١.٦.١٢ مبادئ التحسين

- القياس قبل التحسين
- تحسين الخوارزميات قبل التحسينات الدقيقة
- الثقة بالمترجم قبل الحيل اليدوية

٢.٦.١٢ تحسين خوارزمي

```
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;
}
```

٣.٦.١٢ تحسين الوصول إلى الذاكرة

الوصول المتسلسل للذاكرة يُحسّن تموضع البيانات في الذاكرة المخبئية.

٤.٦.١٢ الإدراج (Inlining)

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

الإدراج قرار للمترجم وليس ضماناً.

٥.٦.١٢ خيارات تحسين المترجم

```
gcc -O2 -march=native program.c
```

٧.١٢ التنقيح والتهيئة

التنقيح والتهيئة ضروريان للصحة والأداء.

١.٧.١٢ التنقيح باستخدام GDB

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

٢.٧.١٢ تنقيح الذاكرة

```
valgrind --leak-check=full ./a.out
```

٣.٧.١٢ التهيئة (Profiling)

```
gcc -pg program.c
gprof a.out gmon.out
```

٨.١٢ الخاتمة

لا يتحقق الأمن والتحسين في C23 عبر كلمات محجوزة خاصة أو مكتبات سحرية، بل عبر الانضباط، وفهم نموذج التنفيذ، واحترام حدود السلوك غير المعرّف. تنشأ برامج C الآمنة والسريعة من الصحة أولاً، والقياس ثانيًا، والتحسين أخيرًا.

الفصل ١٣: الميزات الجديدة والتغييرات في C23

١.١٣ نظرة عامة على C23

يُعد C23 إصدارًا تراكميًا من لغة ISO C. وعلى خلاف الانتقالات الكبرى مثل C89 إلى C99 أو C99 إلى C11، يركّز C23 على التوضيح، والتنقية، والاتساق، والتحديث المحدود، بدل إدخال منظومات جديدة كبيرة.
يحافظ C23 على فلسفة التصميم الأساسية للغة C: التحكم الصريح، وتوليد شيفرة قابل للتنبؤ، والتطابق الوثيق مع تنفيذ الآلة.

١.١.١٣ أهداف تصميم C23

- تحسين وضوح المواصفات واتساقها
- إزالة البنى المتقادمة والخطِرة
- إدخال تحسينات حديثة محدودة
- الحفاظ على التوافق الرجعي

- الإبقاء على ملاءمة اللغة لبرمجة الأنظمة

لا يهدف C23 إطلاقًا إلى تحويل C إلى لغة آمنة للذاكرة أو عالية التجريد.

٢.١٣ تغييرات مستوى اللغة في C23

١.٢.١٣ السمات (Attributes)

يوحّد C23 صيغة السمات باستخدام الأقواس المربعة المزدوجة، مقرّبًا اللغة من الممارسات الشائعة لدى المترجمات مع الحفاظ على القابلية للنقل.

مثال: [[nodiscard]]

```
[[nodiscard]] int compute(void) {
    return 42;
}

int main(void) {
    compute(); /* diagnostic encouraged */
    return 0;
}
```

تقدّم السمات تلميحات للمترجم ولا تغيّر دلالات اللغة.

٢.٢.١٣ الأعداد الصحيحة دقيقة البِت

يقدّم C23 النوع _BitInt(N)، الذي يسمح بتحديد عرض البِت بدقة.

```
_BitInt(17) counter;
unsigned _BitInt(128) big;
```

ملاحظات مهمة:

- الإشارة (Signedness) صريحة
- الحسابات تتبع قواعد الأعداد الصحيحة
- التمثيل معتمد على التنفيذ

٣.٢.١٣ تحسين تحويلات القيم المنطقية والأعداد الصحيحة

يوضّح C23 عدداً من الالتباسات التاريخية المتعلقة بترقية الأعداد، والتحويلات، ودلالات القيم المنطقية، ما يحسّن تشخيصات المترجم وقابلية النقل.

٣.١٣ ما الذي لا يضيفه C23

فهم ما لا يضيفه C23 لا يقل أهمية عن معرفة ما يضيفه.

- لا نموذج جديد لأمان الذاكرة
- لا نظام ملكية أو استعارة
- لا استثناءات
- لا قوالب عامة تتجاوز _Generic
- لا إعادة تصميم لنموذج التعددية

تظل التعددية كما عُرّفت في C11 دون تغيير.

٤.١٣ وضع التعددية في C23

يحافظ C23 على واجهة التعددية في C11 دون تعديل.

```c
#include <threads.h>

_Thread_local int tls_value;
```

توضيح:

- الكلمة المفتاحية هي _Thread_local وليس thread_local
- لم تُضَف أدوات تزامن جديدة
- لم تُدخَل تغييرات على نموذج الذاكرة

٥.١٣ تغييرات المكتبة القياسية

١.٥.١٣ إزالة الواجهات الخطِرة

يواصل C23 إزالة الواجهات التي تشجّع على السلوك غير المعرّف.

واجهات مُزالة أو متقادمة

- gets (أُزيلت قبل C23)
- tmpnam (متقادمة)
- التصريحات الضمنية للدوال

٢.٥.١٣ أمان إنشاء الملفات

يُوحِّد C23 الإنشاء الحصري للملفات عبر سلاسل أوضاع الفتح.

```
FILE *f = fopen("data.txt", "wx");
```

يمنع ذلك الكتابة فوق ملفات موجودة دون قصد.

٣.٥.١٣ توضيح دلالات الدوال

يحسّن C23 صياغة وضمانات العديد من دوال المكتبة، ما يقلّل اختلافات التنفيذ.
توضيح مهم:

- strlcpy وstrlcat ليستا ضمن C ISO
- reallocarray ليست ضمن C ISO
- تبقى هذه امتدادات POSIX/BSD

٦.١٣ التوافق الرجعي

يُعد C23 متوافقًا رجعيًا إلى حدٍّ كبير مع C11 وC18.

١.٦.١٣ اكتشاف إصدار اللغة

```
#if __STDC_VERSION__ >= 202311L
    /* C23 */
#else
    /* Earlier C */
#endif
```

٢.٦.١٣ اعتماد الميزات تدريجيًا

يمكن إدخال السمات والبنى الجديدة تدريجيًا دون كسر الشيفرات القديمة.

```
#if __STDC_VERSION__ >= 202311L
[[nodiscard]]
#endif
int compute(void) {
    return 42;
}
```

٣.٦.١٣ البنى المتقادمة

ينبغي على قواعد الشيفرة إزالة ما يلي:

- التصريحات الضمنية
- تعريفات الدوال بأسلوب K&R
- الاعتماد على السلوك غير المحدد

٧.١٣ مثال عملي للترقية

قبل

```
char buf[100];
gets(buf);
```

بعد

```c
char buf[100];
if (fgets(buf, sizeof(buf), stdin)) {
    /* process input */
}
```

يحسّن هذا التغيير الأمان دون الاعتماد على ميزات لغوية جديدة.

٨.١٣ أفضل الممارسات للاعتماد C23

- التعامل مع C23 بوصفه تنقيحًا لا ثورة
- إعطاء أولوية للسلوك المعرّف على الصياغة الجديدة
- الفصل بين ISO C وامتدادات POSIX
- استخدام تحذيرات المترجم بكثافة
- ترقية الشيفرة تدريجيًا

٩.١٣ الخاتمة

يحدّث C23 لغة C بحذر ومحافظة. تكمن قيمته ليس في ميزات درامية، بل في دقة أعلى، وإعدادات افتراضية أكثر أمانًا، ودلالات أنظف.
تظل شيفرة C23 الجيدة هي C المنضبطة: صريحة، حدّية، وقريبة من الآلة. وتواصل اللغة مكافأة من يفهم التنفيذ، والذاكرة، وحدود السلوك غير المعرّف — ومعاقبة من لا يفهم.

الفصل ١٤: التطبيقات العملية ودراسات الحالة

١.١٤ بناء نواة نظام تشغيل مصغّرة

يُعدّ بناء نواة نظام تشغيل تمرينًا تأسيسيًا في برمجة الأنظمة منخفضة المستوى. فهو يتطلب فهم نماذج التنفيذ، وتخطيط الذاكرة، وواجهات العتاد، والتمييز الدقيق بين قواعد اللغة وقواعد المنصّة.
يقدّم هذا القسم تصميمًا تصوريًا ومصغّرًا لنواة نظام تشغيل، ولا يهدف إلى إنتاج نظام تشغيل كامل.

١.١.١٤ ما هي النواة (وما ليست عليه)

نواة نظام التشغيل برنامج ذو امتيازات عالية مسؤول عن:

- إدارة أوضاع المعالج
- إدارة الذاكرة
- معالجة المقاطعات
- تجريد العتاد

توضيح مهم:

- النوى لا تعمل في بيئة C مستضافة
- معيار ISO C (بما في ذلك C23) لا يعرّف النوى
- تُكتب شيفرة النواة بلغة *Freestanding C*

٢.١.١٤ بيئة Freestanding C

يجب ترجمة شيفرة النواة مع:

- عدم استخدام المكتبة القياسية
- عدم وجود تهيئة تشغيلية (runtime)
- تحكم صريح بنقاط الدخول

```
gcc -ffreestanding -fno-builtin -nostdlib
```

٢.١٤ مفاهيم الإقلاع (Bootstrapping)

يقوم مُحمِّل الإقلاع بتهيئة بيئة التنفيذ ونقل التحكم إلى النواة.
تحديد نطاق مهم:

- محمّلات BIOS ذات النمط الحقيقي أصبحت تراثية
- الأنظمة الحديثة تستخدم UEFI
- المثال هنا تعليمي فقط

٣.١٤ نقطة دخول نواة مصغّرة

١.٣.١٤ نقطة دخول النواة

لا تبدأ النواة من الدالة main.

```
void kernel_main(void) {
    for (;;) {
        /* halt */
    }
}
```

لا تتوفر رؤوس قياسية.

٢.٣.١٤ مثال وصول مباشر إلى العتاد

```
volatile unsigned short *vga =
    (unsigned short *)0xB8000;

void write_char(char c) {
    *vga = (0x0F << 8) | c;
}
```

تفترض هذه الشيفرة وضع النص VGA على معمارية x86.

٤.١٤ معالجة المقاطعات (تصوريًا)

معالجة المقاطعات تعتمد على المعمارية ولا يعرّفها معيار C.

- جداول الواصفات

- تعليمات المعالج
- شيفرة ربط بلغة التجميع

تُستخدم C فقط لمنطق المعالِجات.

٥.١٤ مفاهيم إدارة الذاكرة

مخصّصات الذاكرة القياسية في ISO C غير متاحة داخل النواة.

١.٥.١٤ مخصّص تراكمي بسيط

```
static unsigned char heap[4096];
static unsigned long offset;

void *kmalloc(unsigned long size) {
    if (offset + size > sizeof(heap))
        return 0;
    void *p = &heap[offset];
    offset += size;
    return p;
}
```

لا يمكن لهذا المخصّص تحرير الذاكرة وهو للتوضيح فقط.

٦.١٤ أفضل ممارسات تطوير النواة

- تجنّب الرؤوس القياسية

- تجنّب السلوك غير المعرّف
- فصل منطق C عن شيفرة التجميع
- لا تفترض أي شيء عن بيئة التشغيل

٧.١٤ تصميم مُصرِّف (Compiler) أساسي

بناء المصرّفات مستقل عن ميزات C23 ويعتمد على نظرية اللغات أكثر من واجهات النظام.

١.٧.١٤ خط أنابيب المصرّف

- التحليل المعجمي
- التحليل النحوي
- التحليل الدلالي
- توليد الشيفرة

٢.٧.١٤ التحليل المعجمي (تصوريًا)

تحوّل المُحلِّلات المعجمية سلاسل المحارف إلى رموز.

```
[a-zA-Z_][a-zA-Z0-9_]*    IDENTIFIER
[0-9]+                    NUMBER
```

٣.٧.١٤ التحليل النحوي

يحوّل الرموز إلى أشجار تركيبية وفق قواعد اللغة.

٤.٧.١٤ التحليل الدلالي

يفرض التحليل الدلالي:

- قواعد الأنواع
- قواعد النطاق
- حل الرموز

٥.٧.١٤ توليد الشيفرة

يُنتج توليد الشيفرة:

- لغة التجميع
- الشيفرة البايتية
- تمثيلات وسيطة

٦.٧.١٤ التحسين

تشمل التحسينات:

- طيّ الثوابت
- إزالة الشيفرة الميتة
- إزالة التعابير المشتركة

٨.١٤ كتابة مُشغِّلات الأجهزة

١.٨.١٤ توضيح حاسم

مُشغِّلات الأجهزة ليست برامج **ISO C**.

- تستخدم مشغّلات Linux امتدادات GNU C
- واجهات النواة غير موحّدة
- لا تنطبق C23

٢.٨.١٤ نطاق مشغّلات لينكس

تعمل مشغّلات لينكس:

- في فضاء النواة
- دون مكتبة C قياسية
- وفق قواعد صارمة للنواة

الأمثلة المعروضة خاصة بلينكس.

٣.٨.١٤ مسؤوليات المُشغِّل

- تهيئة العتاد
- معالجة المقاطعات
- نقل البيانات

٩.١٤ برمجة الأنظمة المُضمّنة

تستخدم الأنظمة المُضمّنة C كلغة للتحكم بالعتاد.

١.٩.١٤ خصائص البيئة

- لا نظام تشغيل (بيئة عارية)
- وصول مباشر إلى السجلات
- رؤوس خاصة بالمورّد

٢.٩.١٤ بنية **Bare-Metal**

```
int main(void) {
    init_hardware();
    for (;;) {
        run();
    }
}
```

تُوفّر أداة البناء شيفرة البدء، وليس معيار ISO C.

٣.٩.١٤ الوصول إلى سجلات العتاد

```
#define GPIO_ODR (*(volatile unsigned int *)0x40020014)

GPIO_ODR ^= (1 << 5);
```

٤.٩.١٤ توضيح حول RTOS

أنظمة RTOS:

- ليست جزءاً من C
- توفّر الجدولة والتزامن
- تتطلب انضباطًا صارمًا

٥.٩.١٤ إدارة الطاقة

إدارة الطاقة يحدّدها العتاد لا اللغة.

- أوضاع السكون
- تعطيل الساعات
- الاستيقاظ بالمقاطعات

١٠.١٤ أفضل الممارسات عبر جميع المجالات

- افهم بيئة التنفيذ
- افصل قواعد اللغة عن قواعد المنصّة
- لا تفترض قابلية النقل
- احترم حدود السلوك غير المعرّف
- اقرأ أدلة المعمارية

١١.١٤ الخاتمة

البرمجة منخفضة المستوى العملية ليست مسألة ميزات لغوية، بل مسألة فهم بيئات التنفيذ. تظل C23 أداة قوية لبرمجة الأنظمة، لكن النوى، والمشغّلات، والمصرّفات، والأنظمة المُضمّنة تقع خارج ضمانات معيار C. وتتحقق الإتقان باحترام هذه الحدود واستخدام C بوعي — لا بافتراض أنها تقدّم ما لم تعد به قط.

الفصل ١٥: مستقبل لغة C

١.١٥ اتجاهات تطوّر لغة C

حافظت لغة البرمجة C على أهميتها لأكثر من خمسة عقود ليس عبر التوسّع السريع، بل عبر ضبط النفس والانضباط. ويُوجَّه تطوّرها بالاستقرار، وقابلية التنبؤ، واحترام قواعد الشيفرات القائمة. يتطلّب فهم مستقبل C التمييز بين:

- معيار لغة ISO C
- تطوّر المترجمات وسلاسل الأدوات
- ممارسات النظام البيئي والمجتمع

وهذه عناصر مترابطة، لكنها ليست متطابقة.

١.١.١٥ فلسفة التقييس

تعطي لجنة ISO C (WG14) الأولوية لما يلي:

- التوافق العكسي
- الحد الأدنى من الاضطراب الدلالي

- المواصفة الدقيقة
- القابلية للنقل عبر المعماريات

يواصل C23 هذا التقليد؛ فهو تحسين تدريجي وليس تحوّلاً نموذجيًا جذريًا.

٢.١.١٥ ما الذي يمثّله C23 فعليًا

يُحدِّث C23 اللغة بشكل محافظ:

- يوضّح القواعد القائمة
- يزيل البُنى المتقادمة
- يقدّم ميزات محدودة منخفضة المخاطر
- يُحسّن اتساق المواصفة

ولا يحاول C23 إعادة تصميم C كلغة آمنة للذاكرة أو عالية التجريد.

٢.١٥ الأمان والسلامة: الواقع مقابل التوقّع

لم يكن الأمان في C يومًا متكفّلاً به من قِبل اللغة نفسها.

١.٢.١٥ لا أمان مضمّن للذاكرة

معيار ISO C، بما في ذلك C23:

- لا يُجري فحصًا للحدود
- لا يمنع تجاوز سعة المخازن

- لا يتتبّع الملكية أو الأعمار

ويتحقق الأمان في C عبر:

- الانضباط في كتابة الشيفرة
- التحليل الساكن
- تصميم واجهات برمجية حذِر
- أدوات خارج نطاق اللغة

٢.٢.١٥ دور الأدوات

تأتي تحسينات السلامة الحديثة من:

- تشخيصات المترجم
- أدوات Sanitizers
- المحلّلات الساكنة
- ممارسات مراجعة الشيفرة

وهذه غير معرّفة في معيار ISO C وتتطوّر باستقلالية.

٣.١٥ التزامن والتوازي

قدّم C11 نموذج الخيوط والذريّات، ولا يوسّعه C23.

١.٣.١٥ استقرار نموذج الخيوط

- لا بدائيات تزامن جديدة
- لا ضمانات للمجدول
- لا تجريدات لا-متزامنة

يبقى التزامن صريحًا ومنخفض المستوى عمدًا.

٢.٣.١٥ الذريّات كأساس

توجد العمليات الذرّية لدعم:

- بُنى بيانات دون أقفال
- تزامن على مستوى العتاد
- قابلية التشغيل البيني مع واجهات النظام

وهي ليست إطارًا عالي المستوى للتزامن.

٤.١٥ قابلية التشغيل البيني: ما الذي تضمنه C

توجد قابلية التشغيل البيني في C بسبب:

- تمثيلات بيانات بسيطة
- اتفاقيات نداء مستقرة (حسب المنصّة)
- افتراضات تشغيلية محدودة

١.٤.١٥ ما الذي لا يعرّفه ISO C

- واجهات الثنائيات للتطبيقات (ABI)
- واجهات الاستدعاء الأجنبي (FFI)
- اتفاقيات تشويه الأسماء

وهذه مسؤوليات المنصّات والمترجمات.

٥.١٥ الأداء كقيمة جوهرية

يبقى الأداء السمة الفارقة للغة C.

١.٥.١٥ الأداء على مستوى اللغة

توفّر C:

- تنفيذًا قابلاً للتنبؤ
- وصولاً مباشرًا للذاكرة
- حدًا أدنى من كلفة التجريد

وتتجنّب اللغة عمدًا الميزات التي تُخفي الكلفة.

٢.٥.١٥ التحسين المعتمد على المترجم

تأتي التحسينات المتقدمة من المترجمات:

- التوجيه المتّجه (Vectorization)

- التحسين وقت الربط (LTO)
- التحسين الموجّه بالملفّات التعريفية (PGO)

وهذه مستقلة عن مواصفة اللغة.

٦.١٥ قابلية النقل والاستخدام عبر المنصّات

تظل C من أكثر اللغات قابليةً للنقل على الإطلاق.

١.٦.١٥ ماذا تعني القابلية للنقل في C

- قابلية نقل على مستوى الشيفرة المصدرية
- تعريف واضح للسلوك غير المعرّف
- تكيّف صريح مع المنصّات

ولا تعني القابلية للنقل سلوكًا متطابقًا عبر الأنظمة.

٧.١٥ دور C في البرمجيات الحديثة

تبقى C حجر الأساس في:

- أنظمة التشغيل
- الأنظمة المُضمّنة
- المترجمات وبيئات التشغيل
- المكتبات عالية الأداء

- البرمجيات الحرِجة أمنيًا

وغالبًا ما يكون دورها غير مرئي لكنه لا غنى عنه.

٨.١٥ التعليم والقيمة طويلة الأمد

يوفّر تعلّم C:

- فهمًا للذاكرة
- فهمًا لنماذج التنفيذ
- احترامًا لحدود السلوك غير المعرّف
- وعيًا بقيود العتاد

وتنتقل هذه المهارات إلى كل لغات الأنظمة الأخرى.

٩.١٥ الاتجاه المستقبلي للغة C

مستقبل C ليس في التوسّع، بل في الحفظ.

- ستبقى C صغيرة
- ستبقى C صريحة
- ستبقى C قريبة من العتاد
- ستقاوم تضخّم الميزات

وهذا ليس جمودًا، بل استقرارًا مقصودًا.

١٠.١٥ الخاتمة

تدوم C لأنها تعرف ماهيتها.

يعزّز C23 هذه الهوية عبر تنقية اللغة دون الإفراط في توسيعها. ولا يُحدَّد مستقبل C بالاتجاهات، بل بالثقة: الثقة في أن تبقى اللغة قابلة للتنبؤ، وقابلة للنقل، وصادقة بشأن ما تقدّمه وما لا تقدّمه.

ولبرمجة المستويات المنخفضة، وأنظمة التشغيل، والمترجمات، والأنظمة المُضمّنة، تظل C ليست ذات صلة فحسب — بل غير قابلة للاستبدال.

الملاحق

الملحق أ: نظرة عامة على مكتبة ISO C القياسية

تُعد مكتبة ISO C القياسية مجموعةً محددة رسميًا من الترويسات (headers)، والأنواع (types)، والماكرو (macros)، والدوال (functions)، كما هو معرّف في المعيار ISO/IEC 9899. وهي توفر المرافق الأساسية المطلوبة لكتابة برامج C قابلة للنقل، مع تجنّبٍ متعمّد لاتخاذ قرارات تتعلق بأنظمة التشغيل، أو واجهات المستخدم، أو الشبكات، أو سلامة الذاكرة.
يقدم هذا الملحق نظرةً مفاهيمية على المكتبة القياسية كما عرّفها معيار ISO C، بما في ذلك C23، دون إسناد أي إضافات غير قياسية أو خاصة بمنصات معينة إلى اللغة نفسها.

مبادئ التصميم

تم تصميم مكتبة C القياسية بحيث تكون:

- بسيطة ومترابطة بشكل متعامد
- قابلة للنقل عبر المعماريات وأنظمة التشغيل
- صريحة في التعامل مع الفشل والسلوك غير المعرّف
- مستقلة عن بيئات التشغيل

ولا يُقصد بها توفير تجريدات عالية المستوى أو ضمانات أمان تلقائية.

الترويسات القياسية الأساسية

<stdio.h>
مرافق الإدخال والإخراج المُخزَّن باستخدام التدفقات (streams).

- إدخال/إخراج منسق: printf، scanf
- عمليات الملفات: fopen، fread، fwrite
- إدخال/إخراج الأحرف: getchar، putchar

<stdlib.h>
أدوات عامة الغرض.

- إدارة الذاكرة الديناميكية: malloc، free
- إنهاء البرنامج: exit، abort
- التحويلات العددية: strtol، strtod

<string.h>
عمليات على تسلسلات البايت والسلاسل المنتهية بمحرف NULL.

- النسخ والمقارنة: memcpy، strcmp
- البحث والطول: strchr، strlen

<stdint.h>
أنواع أعداد صحيحة بعرض ثابت لضمان قابلية النقل.

- الأنواع الموقّعة: int32_t، int64_t
- الأنواع غير الموقّعة: uint32_t، uint64_t

<stddef.h>
تعريفات أساسية.

- size_t، ptrdiff_t، NULL

<assert.h>
دعم التشخيص.

- التأكيدات أثناء التشغيل باستخدام assert

ملاحظات حول C23

يعمل معيار C23 بشكل أساسي على تنقيح وتوضيح المواصفات القائمة. ولا يعيد تعريف المكتبة بوصفها إطارًا عالي المستوى أو واعيًا بـ Unicode. وأي وظائف موسّعة تتعلق بالتدويل، أو الأمان، أو سلامة الذاكرة تبقى خارج نطاق ISO C، ويتم التعامل معها عبر مكتبات خارجية أو واجهات برمجة تطبيقات خاصة بالمنصات.

الملحق ب: الأخطاء الشائعة في برمجة C

تمنح لغة C تحكمًا مباشرًا في الذاكرة والتنفيذ، مما يجعلها قوية وخطرة في الوقت نفسه. وتنشأ معظم العيوب الخطيرة في برامج C ليس من الصياغة، بل من انتهاك القواعد الدلالية للغة أو الاعتماد على السلوك غير المعرّف.

أخطاء إدارة الذاكرة

تسرّب الذاكرة

```c
void leak(void) {
    int *p = malloc(10 * sizeof *p);
    /* missing free */
}
```

المؤشرات المعلّقة

```c
int *bad(void) {
    int *p = malloc(sizeof *p);
    free(p);
    return p;
}
```

تجاوز سعة المصفوفة

```c
void overflow(void) {
    char buf[8];
    strcpy(buf, "too long");
}
```

السلوك غير المعرّف

- استخدام متغيرات غير مهيأة
- فك إشارة مؤشرات NULL أو غير صالحة
- فيض الأعداد الصحيحة الموقّعة
- انتهاك أعمار الكائنات

السلوك غير المعرّف ليس خطأً وقت التشغيل، بل هو خرقٌ لعقد اللغة نفسه.

أخطاء متعلقة بالأمان

- ثغرات سلاسل التنسيق
- سباقات التحقق ثم الاستخدام (Time-of-check vs Time-of-use)
- إدخال غير مُتحقق منه وثقة ضمنية

يتحقق الأمان في C عبر الانضباط، لا عبر ضمانات لغوية.

الملحق ج: سلاسل الأدوات وممارسة التطوير

يعرّف معيار ISO C لغةً، لا بيئة تطوير. ويعتمد تطوير C في العالم الحقيقي على سلاسل أدوات خارجية تحكمها أعراف المنصات أكثر من المعيار اللغوي.

المترجمات

تشمل مترجمات C الشائعة:

- GNU Compiler Collection (GCC)
- Clang/LLVM
- Microsoft Visual C

توفر هذه المترجمات تشخيصات وتحسينات وامتدادات ليست جزءاً من ISO C.

التنقيح والتحليل

يتضمن تطوير C الاحترافي عادةً:

- منقحات على مستوى الشيفرة المصدرية
- أدوات تحليل ساكن
- تحليل ديناميكي (المعقمات، أدوات القياس)

تُكمّل هذه الأدوات فهم اللغة، لكنها لا تُغني عنه.

أنظمة البناء

تقوم أنظمة البناء بأتمتة عمليتي الترجمة والربط. وتشمل الأساليب الشائعة الأنظمة المعتمدة على make والأنظمة المعتمدة على المولّدات. وهذه الأنظمة متعامدة مع لغة C نفسها.

الملحق د: مشاريع تعليمية نموذجية

تهدف المشاريع التالية إلى الأغراض التعليمية. وهي ليست تطبيقات مرجعية، ولا تمثل أفضل الممارسات لأنظمة الإنتاج دون مزيد من التنقيح.

المستوى المبتدئ

- آلة حاسبة تعمل من سطر الأوامر
- لعبة تخمين الأرقام

تعزز هذه المشاريع فهم تدفق التحكم، والإدخال/الإخراج، ومعالجة البيانات الأساسية.

المستوى المتوسط

- إدارة السجلات باستخدام الملفات
- تنفيذ قائمة مترابطة

تُدخل هذه المشاريع مفاهيم البُنى، والذاكرة الديناميكية، وانضباط المؤشرات.

المستوى المتقدم

- صدفة أوامر مصغّرة
- محلل لغوي (Lexical Analyzer)

تكشف هذه المشاريع استدعاءات النظام، والتحكم في العمليات، وأسس بناء المترجمات.

ملاحظة ختامية

تكون المشاريع المكتوبة بلغة C أكثر قيمة عندما تُعامل كتجارب لفهم الذاكرة، والتنفيذ، وتحمل المسؤولية. فالصحة، والوضوح، والانضباط أهم من عدد الميزات أو سهولة الاستخدام.

المراجع

معايير لغة C

- **ISO/IEC 9899 — Programming Languages — C**
  - المعيار الدولي الرسمي الذي يعرّف لغة البرمجة C.
  - يغطي صياغة اللغة، ودلالاتها، والمكتبة القياسية.
  - تُنشر جميع الإصدارات الحديثة (C11، C17، C23) تحت هذا المعيار.

البرمجة الحديثة بلغة C

- Jens Gustedt — *Modern C*
  - معالجة صارمة لممارسات البرمجة الحديثة في C.
  - يركز على الاستخدام الصحيح لميزات حقبة C11 والانضباط في البرمجة بلغة C.
- K. N. King — *C Programming: A Modern Approach*
  - مقدمة شاملة ومنظمة للغة C.
  - مناسبة للتعلّم التأسيسي وللاستخدام كمرجع طويل الأمد.

البرمجة منخفضة المستوى ومعمارية الحاسوب

- David R. ،Randal E. Bryant — *Computer Systems: A Programmer's Perspective* O'Hallaron
  - يشرح كيفية تفاعل البرمجيات مع العتاد على جميع المستويات.
  - تركيز قوي على الذاكرة، ونماذج التنفيذ، والأداء.
- Jonathan Bartlett — *Programming from the Ground Up*
  - يقدم مفاهيم البرمجة منخفضة المستوى من خلال لغة التجميع (assembly).
  - يركز على فهم التنفيذ بدلاً من الاعتماد على التجريدات.
- Dennis M. Ritchie ،Brian W. Kernighan — *The C Programming Language*
  - المقدمة الأصلية والمرجعية المعتمدة للغة C.
  - ترسخ الفلسفة الأساسية التي قامت عليها اللغة.

أنظمة التشغيل

- Greg Gagne ،Peter B. Galvin ،Abraham Silberschatz — *Operating System Concepts*
  - معالجة أكاديمية شاملة لمبادئ أنظمة التشغيل.
- Andrew S. Tanenbaum — *Modern Operating Systems*
  - يغطي معمارية أنظمة التشغيل وتصميمها وتنفيذها.
- Andrea C. Arpaci-،Remzi H. Arpaci-Dusseau — *Operating Systems: Three Easy Pieces* Dusseau
  - يقدم مفاهيم أنظمة التشغيل بوضوح وأساس مفاهيمي قوي.

تصميم المترجمات

- Ravi ،Monica S. Lam ،Alfred V. Aho — *Compilers: Principles, Techniques, and Tools* Jeffrey D. Ullman ،Sethi
  - المرجع الكلاسيكي لنظرية وبناء المترجمات.
- Linda Torczon ،Keith Cooper — *Engineering a Compiler*
  - يركز على التنفيذ العملي وتحسين المترجمات.
- Andrew W. Appel — *Modern Compiler Implementation in C*
  - منهج عملي لبناء المترجمات باستخدام لغة C.

التطبيق العملي والتعلّم التطبيقي

- تصميم وتنفيذ مترجم أو مفسّر صغير للغة مخصّصة.
- كتابة برامج C مستقلة (freestanding) لبيئات bare-metal أو على مستوى النواة.
- دراسة وقراءة قواعد شيفرة كبيرة وواقعية مكتوبة بلغة C، مثل نوى أنظمة التشغيل، والمكتبات القياسية، وسلاسل الأدوات.
- تطبيق التحليل الساكن، والمراجعة الدقيقة للشيفرة، والاختبار المنضبط على مشاريع حقيقية.

ملاحظة ختامية

تؤكد المراجع المدرجة هنا على المعرفة التأسيسية، والأهمية طويلة الأمد، والانضباط قبل الموضة. وقد اختيرت لدعم الفهم العميق للغة C، والبرمجة منخفضة المستوى، وأنظمة التشغيل، وبناء المترجمات — بمعزل عن الأدوات العابرة، أو الدروس السريعة، أو الصيحات المؤقتة.