



高等程序设计 - Qt/C++

第 2 章：C++ 语言基础与进阶

王培杰

长江大学地球物理与石油资源学院

2025 年 9 月 5 日



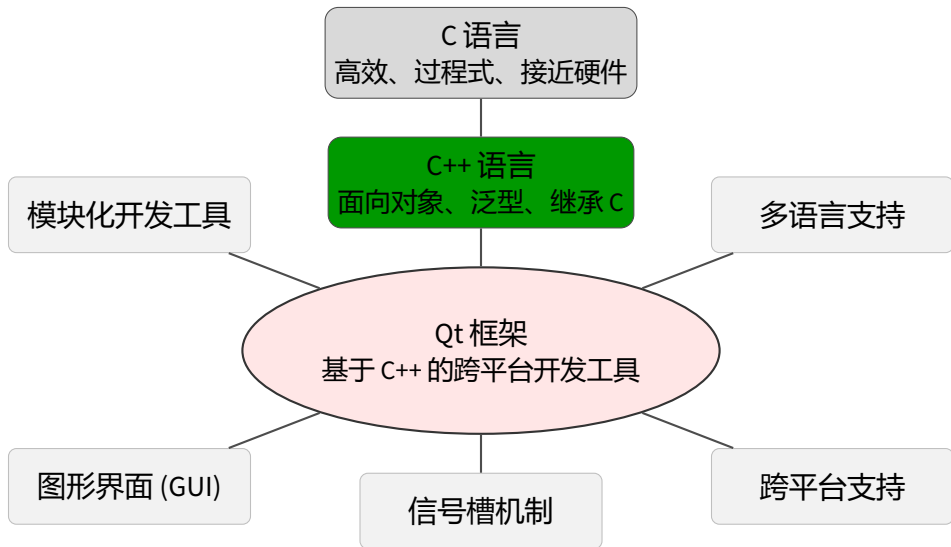
- 1 C、C++ 与 Qt
- 2 C 语言基础知识

- 3 C++ 基本语法
- 4 C++ 面向对象编程



- 1 C、C++ 与 Qt
- 2 C 语言基础知识

- 3 C++ 基本语法
- 4 C++ 面向对象编程



- **C 语言：**

- 诞生于 20 世纪 70 年代，是一种结构化、过程式的高级编程语言。
- 以高效、接近底层硬件著称，广泛应用于操作系统、嵌入式开发、驱动程序等领域。
- C 语言为 C++ 提供了坚实的基础，C++ 完全兼容 C 语言，是 C 的超集。

- **C++：**

- 由 Bjarne Stroustrup 于 20 世纪 80 年代初开发，在 C 语言基础上引入了面向对象编程 (OOP)、泛型编程等特性。
- 支持类、继承、多态、模板等高级特性，适合开发大型复杂系统。
- C++ 不仅能编写高效底层代码，还能实现高层次的抽象，兼容 C 语言代码，便于项目迁移和扩展。

- **Qt：**

- Qt 是一个基于 C++ 的跨平台应用程序开发框架，现由 Qt Company 维护。
- 提供了丰富的 GUI（图形用户界面）组件、信号槽机制、网络、数据库、多媒体等模块，极大提升了 C++ 开发效率。
- 支持 Windows、Linux、macOS、Android 等多平台，代码可移植性强。
- Qt 不仅适用于桌面应用开发，也广泛应用于嵌入式系统、移动端等领域，是 C++ 开发 GUI 和跨平台应用的首选框架之一。



● 第一阶段：C 语言基础

- 学习 C 语言的基本语法、数据类型、运算符、流程控制（顺序、选择、循环）。
- 掌握函数、数组、指针、结构体、文件操作等核心内容。
- 了解内存管理、编译与调试基础，为后续学习打下坚实基础。

● 第二阶段：C++ 进阶

- 在 C 语言基础上，学习 C++ 的面向对象编程（OOP）思想，包括类、对象、继承、多态、封装等。
- 掌握 C++ 的模板、STL（标准模板库）、异常处理、运算符重载等高级特性。
- 了解 C++11/14/17 等新标准的常用特性，为现代 C++ 开发做准备。

● 第三阶段：Qt 开发实践

- 学习 Qt 的基本概念、开发环境搭建、项目结构。
- 掌握 Qt 的信号与槽机制、常用控件、布局管理、事件处理等 GUI 开发基础。
- 进阶学习 Qt 的多线程、网络编程、数据库、多媒体、绘图等模块。
- 了解 Qt 的跨平台特性，能够在不同操作系统下进行开发与部署。
- 通过实际项目练习，提升综合开发能力。

C、C++ 与 Qt 学习路线（非常规路径）

直接上手 Qt

- 跳过传统 C/C++ 基础，直接从 Qt 入门，先体验图形界面开发的乐趣。
- 以实际项目驱动学习，边做边学，遇到 C++ 语法和概念时再查阅补充。
- 重点掌握 Qt 的信号与槽、常用控件、布局管理、事件处理等 GUI 开发核心。
- 随着项目深入，逐步接触 Qt 的多线程、网络、数据库、多媒体、绘图等高级模块。
- 利用 Qt 的跨平台特性，尝试在不同操作系统下编译和运行项目，提升实战能力。

以 Qt 为载体学习 C++

- 以 Qt 项目开发为主线，带动 C++ 语法、面向对象、模板等知识的学习。
- 在实际编码中理解类、继承、多态、信号槽等 C++ 与 Qt 结合的用法。
- 通过分析和修改 Qt 源码或示例，深入理解 C++ 的高级特性和 Qt 的设计思想。
- 结合 Qt 丰富的模块，逐步掌握 C++ 在实际工程中的应用场景。
- 以项目为驱动，理论与实践结合，快速提升 C++ 与 Qt 的综合开发能力。

- 1 C、C++ 与 Qt
- 2 C 语言基础知识

- 3 C++ 基本语法
- 4 C++ 面向对象编程

C 语言的重要性

- **系统编程语言**：操作系统、驱动程序、嵌入式系统
- **高效性**：直接内存操作，编译型语言，执行效率高
- **强类型语言**：编译时类型检查，减少运行时错误
- **底层控制**：指针、位操作、内存管理
- **C++ 基础**：C++ 完全兼容 C 语言，是 C 的超集

C 语言特点

- **简洁高效**：语法简洁，编译后执行效率高
- **可移植性**：标准 C 代码可在不同平台编译运行
- **底层抽象**：提供硬件抽象，但不隐藏底层细节
- **过程式编程**：函数式编程范式，模块化设计

基本数据类型

- **整数类型：**

- int：标准整型，通常用于存储整数，大小一般为 4 字节（32 位系统/编译器）。
- short：短整型，通常为 2 字节，适合存储较小范围的整数。
- long：长整型，通常为 4 或 8 字节，能表示更大范围的整数。
- char：字符型，通常为 1 字节，用于存储单个字符或小整数。

- **浮点类型：**

- float：单精度浮点型，通常为 4 字节，适合存储小数，精度有限。
- double：双精度浮点型，通常为 8 字节，精度更高，适合需要更高精度的计算。

- **修饰符：**

- signed：有符号类型，能表示正数和负数（如 signed int）。
- unsigned：无符号类型，只能表示非负数，范围更大（如 unsigned int）。

- **类型大小：**不同平台和编译器下，类型所占字节数可能不同。

变量和常量

- **变量声明**: 类型 变量名;。变量用于存储可变的数据, 声明时可初始化 (如 `int b = 10;`)。
- **常量定义**:
 - `const` 关键字: 如 `const int MAX = 100;`, 定义只读变量, 编译期间检查不可修改。
 - `#define` 宏定义: 如 `#define PI 3.14`, 在预处理阶段进行文本替换, 常用于全局常量。
- **作用域**:
 - **局部变量**: 在函数或代码块内部声明, 只在其作用范围内有效。
 - **全局变量**: 在所有函数外部声明, 整个文件都可访问。
 - **静态变量**: 用 `static` 修饰, 静态变量在函数多次调用间保持值不变。
- **存储类**:
 - `auto`: 自动存储类, 局部变量默认类型, 现代 C 已很少显式使用。
 - `static`: 静态存储类, 延长变量生命周期, 见上。
 - `extern`: 外部变量声明, 用于引用其他文件的全局变量。
 - `register`: 将变量存储在 CPU 寄存器中, 提高访问速度, 现代编译器通常自动优化。

if-else 语句

- **基本语法:** `if (条件) {语句块 1} else {语句块 2}`
- **功能:** 根据条件表达式的真假, 选择执行不同的代码块。
- **嵌套:** if 语句可以嵌套使用, 实现多级判断。

```
9      int a = 5;
10     int b = 10;
11
12     // if-else 语句
13     if (a > 0) {
14         printf("a 是正数\n");
15     } else {
16         printf("a 不是正数\n");
17     }
```

```
19     // 嵌套 if 语句
20     if (a > 0) {
21         if (a > 10) {
22             printf("a 大于 10\n");
23         } else {
24             printf("a 在 1 到 10 之间\n");
25         }
26     }
```

C 语言条件语句：switch-case

switch-case 语句

- **基本语法**：switch (表达式) { case 常量: 语句; break; ... default: 语句; }
- **功能**：根据表达式的值，选择匹配的 case 分支执行。
- **注意事项**：每个 case 后通常加 break，否则会“贯穿”执行到下一个 case。

```
28 // switch-case 语句
29 switch (a) {
30     case 1:
31         printf("a 等于 1\n");
32         break;
33     case 5:
34         printf("a 等于 5\n");
35         break;
36     default:
37         printf("a 是其他值\n");
38 }
```

switch 条件表达式限定

- **switch 条件表达式**：必须是整数类型（如 int、char）。
- **case 常量**：必须是整数常量，且每个 case 的常量值必须不同。
- **default**：可选，用于处理未匹配的 case。

for 循环

- **基本语法**: `for(初始化; 条件; 更新){循环体}`
- **功能**: 适合已知循环次数的场景。
- **执行顺序**: 初始化 → 条件检查 → 循环体 → 更新 → 条件检查...

```
44 // for 循环
45 printf("for 循环输出: ");
46 for (int i = 0; i < 5; i++) {
47     printf("%d ", i);
48 }
49 printf("\n");
```

while 和 do-while 循环

- **while 循环语法:** while(条件){循环体}
- **特点:** 先判断条件, 后执行循环体, 可能一次都不执行。
- **do-while 循环语法:** do{循环体}while(条件);
- **特点:** 先执行循环体, 再判断条件, 至少执行一次。

```
51 // while 循环
52 printf("while 循环输出: ");
53 int i = 0;
54 while (i < 3) {
55     printf("%d ", i);
56     i++;
57 }
58 printf("\n");
```

```
60 // do-while 循环
61 printf("do-while 循环输出: ");
62 int j = 0;
63 do {
64     printf("%d ", j);
65     j++;
66 } while (j < 3);
67 printf("\n");
```

break 和 continue 在循环中的应用

- **break**: 跳出当前循环，循环提前结束。
- **continue**: 跳过本次循环剩余部分，直接进入下一次循环判断。

```
69 // break 和 continue
70 printf("break 和 continue 示例: ");
71 for (int k = 0; k < 10; k++) {
72     if (k == 3) {
73         continue; // 跳过 3
74     }
75     if (k == 7) {
76         break; // 在 7 处停止
77     }
78     printf("%d ", k);
79 }
80 printf("\n");
```


什么是函数？

- **函数**是实现特定功能的独立代码块，可以重复调用。
- 通过函数可以将复杂问题分解为若干小问题，便于模块化设计和代码复用。
- C 语言的函数包括**标准库函数**（如 printf、scanf）和**用户自定义函数**。

```
1  #include <stdio.h>
2  #include <windows.h>
3
4  // 函数声明：计算两个整数的和
5  int add(int a, int b) {
6      return a + b; // 返回 a 和 b 的和
7  }
```

```
9  int main() {
10     SetConsoleOutputCP(65001);
11     // 调用 add 函数，计算 2 和 3 的和
12     int result = add(2, 3);
13     // 输出结果
14     printf("2 + 3 = %d\n", result);
15     getchar();
16     return 0; // 程序正常结束
17 }
```

C 语言函数的声明与定义

函数声明

- 作用：告诉编译器函数的名称、返回值类型和参数类型。
- 位置：一般写在文件开头或头文件中。
- 语法：返回类型 函数名 (参数类型列表);
- 例：`int add(int a, int b);`

函数定义

- 作用：给出函数的具体实现。
- 位置：函数声明之后，函数调用之前。
- 语法：返回类型 函数名 (参数列表){函数体}
- 例：`int add(int a, int b){ return a+b; }`

函数调用的基本过程

- 使用函数名（实参列表）的形式调用函数。
- 调用时，实参的值会传递给函数的形参。
- 函数执行完毕后，将结果（如果有）返回给调用处。

```
11 // 调用 add 函数，计算 2 和 3 的和  
12 int result = add(2, 3);
```

值传递 (pass by value)

- 调用函数时，将实参的值复制一份传递给形参。
- 在函数内部对形参的修改，不会影响到外部的实参变量。
- 适用于基本数据类型（如 `int`、`float` 等）。

值传递定义：

```
12 // 值传递示例: add 函数
13 int add(int a, int b) {
14     return a + b;
15 }
```

值传递示例：

```
46 // 函数调用——值传递
47 int result = add(5, 3);
```

指针传递（地址传递，pass by pointer/address）

- 通过传递变量的地址（指针）给函数，实现对外部变量的直接修改。
- 在函数内部通过解引用指针，可以改变外部变量的值。
- 适用于需要在函数内修改外部变量，或传递大型数据结构（如数组）。

指针传递定义：

```
17 // 引用传递示例：swap 函数
18 void swap(int *a, int *b) {
19     int temp = *a;
20     *a = *b;
21     *b = temp;
22 }
```

指针传递示例：

```
50 // 函数调用——引用传递
51 int x = 10, y = 20;
52 printf("交换前: x = %d, y = %d\n", x, y);
53 swap(&x, &y); // 传递变量地址，实现引用传递
54 printf("交换后: x = %d, y = %d\n", x, y);
```

C 语言函数的返回值与 void 类型

返回值

- C 语言函数可以有返回值，也可以没有返回值。返回值用于将函数内部计算的结果传递给调用者。
- 返回值类型在函数声明和定义时指定。例如，`int add(int a, int b)` 表示返回类型为 `int`。
- 使用 `return` 语句返回结果。例如：`return a + b;`。
- 如果函数有返回值，必须保证所有可能的执行路径都能返回一个与声明类型一致的值。
- 调用有返回值的函数时，通常用变量接收返回结果。
- 如果函数声明为非 `void` 类型但未返回值，编译器会发出警告或错误。

void 类型

- `void` 类型表示函数没有返回值。此类函数只执行操作，不向调用者返回数据。
- `void` 函数可以使用 `return;` 语句提前结束函数，但不能带返回值。

函数的优点

- 复用性
- 可读性
- 可维护性
- 可测试性
- 可扩展性
- 可重用性
- 可移植性
- 模块化

注意事项

- 函数名不能重复
- 函数声明与定义要一致
- 参数类型和个数要与声明一致
- 注意变量作用域和生命周期
- 返回值类型要匹配
- 避免递归陷入死循环
- 参数传递方式要明确
- 防止未初始化变量参与运算

什么是指针

- 指针是存储变量内存地址的变量。通过指针可以间接访问和操作内存中的数据。
- 声明指针：使用 `*` 声明指针类型，如 `int *p`；表示 `p` 是一个指向 `int` 类型的指针。
- 取地址操作符 `&`：`p = &x`；将变量 `x` 的地址赋值给指针 `p`。
- 解引用操作符 `*`：`*p` 访问指针所指向的内存单元的值。

```
9 // 基本指针操作
10 int x = 10;
11 int* ptr = &x;
12
13 printf("x 的值: %d\n", x);
14 printf("x 的地址: %p\n", (void*)&x);
15 printf("ptr 存储的地址: %p\n", (void*)ptr);
16 printf("ptr 指向的值: %d\n", *ptr);
```


指针与变量

指针与变量

- 通过指针可以修改变量的值。
- 指针本身也是一个变量，存储的是地址。

代码示例

```
9      // 基本指针操作
10     int x = 10;
11     int* ptr = &x;
12
13     printf("x 的值: %d\n", x);
14     printf("x 的地址: %p\n", (void*)&x);
15     printf("ptr 存储的地址: %p\n", (void*)ptr);
16     printf("ptr 指向的值: %d\n", *ptr);
17
18     // 通过指针修改值
19     *ptr = 20;
20     printf("修改后 x 的值: %d\n", x);
```

指针与数组

指针与数组

- 数组名本质上是首元素的指针，指针可以遍历数组元素。
- 指针可以进行加减运算（如 `p++`），步长为所指类型的字节数，常用于数组遍历。

指针访问数组代码示例

```
22 // 指针和数组
23 int arr[] = {1, 2, 3, 4, 5};
24 int* arrPtr = arr; // 数组名是指向第一个元素
    ↳ 的指针
25
26 printf("通过指针访问数组: ");
27 for (int i = 0; i < 5; i++) {
28     printf("%d ", *(arrPtr + i));
29 }
30 printf("\n");
```

指针修改数组代码示例

```
32 printf("通过指针修改数组: ");
33 for (int i = 0; i < 5; i++) {
34     *(arrPtr + i) = i * 10;
35 }
36 for (int i = 0; i < 5; i++) {
37     printf("%d ", arr[i]);
38 }
```

内存分区与动态分配

- C 程序运行时内存主要分为代码区、全局/静态区、栈区和堆区。
- 栈内存：局部变量和函数参数存储在栈区，由系统自动分配和释放，空间有限，生命周期随函数调用结束而结束。
- 堆内存：通过 `malloc()` 等函数动态分配，需手动用 `free()` 释放，适合存储生命周期较长或大小不确定的数据。

```
8 // 动态内存分配示例
9 int* ptr = (int*)malloc(sizeof(int));
10 *ptr = 100;
11 printf("动态分配的值: %d\n", *ptr);
12 free(ptr);
```

指针常见错误与良好习惯

常见错误

- 未初始化指针，访问已释放内存，指针越界等都可能導致程序崩溃或不可预期行为。
- 内存泄漏：动态分配的内存未及时释放会造成内存泄漏，长期运行会耗尽系统资源。
- 野指针：指针指向的内存已被释放或未初始化，访问野指针会导致不可预期的错误。
- 悬空指针：释放内存后未将指针置为 `NULL`，指针仍然指向原地址，极易出错。
- 内存越界：访问数组或内存块时超出分配范围，可能破坏数据或引发崩溃。

良好习惯

- 动态分配内存后及时检查返回值是否为 `NULL`，用完后立即 `free()` 并将指针赋为 `NULL`。
- 初始化指针为 `NULL`，避免野指针。

结构体概念

- **复合数据类型**: 将不同类型的数据组合在一起
- **成员访问**: 使用 . 运算符
- **指针访问**: 使用 -> 运算符
- **内存对齐**: 结构体成员的内存布局

结构体应用

- **数据封装**: 将相关数据组织在一起
- **函数参数**: 传递复杂数据结构
- **链表实现**: 自引用结构体
- **面向对象基础**: C++ 类的雏形

C 语言结构体示例

```
1  #include <stdio.h>
2  #include <windows.h>
3
4  // 定义结构体类型
5  struct Student {
6      char name[50]; // 姓名
7      int age;        // 年龄
8      float score;    // 分数
9  };
10
11 int main() {
12     SetConsoleOutputCP(65001);
13
14     // 定义并初始化结构体变量
```

```
15     struct Student stu = {"张三", 18, 90.5f};
16
17     // 访问结构体成员并输出
18     printf("姓名: %s\n", stu.name);
19     printf("年龄: %d\n", stu.age);
20     printf("分数: %.1f\n", stu.score);
21
22     // 使用结构体指针访问成员
23     struct Student *p = &stu;
24     printf("通过指针访问姓名: %s\n", p->name);
25
26     getchar();
27     return 0;
28 }
```



兼容性

- **完全兼容**: C++ 是 C 的超集, 几乎所有的 C 代码都是有效的 C++ 代码
- **语法兼容**: C 的基本语法在 C++ 中完全支持
- **库兼容**: C 标准库在 C++ 中可用 (需要适当的头文件)
- **编译兼容**: C++ 编译器可以编译 C 代码

C++ 对 C 的扩展

- **面向对象**: 类、继承、多态
- **函数重载**: 同名函数不同参数
- **引用**: 变量的别名
- **模板**: 泛型编程
- **异常处理**: try-catch 机制
- **命名空间**: 避免名称冲突

底层语言的特点

强类型语言的优势

- **编译时检查**: 类型错误在编译时发现, 减少运行时错误
- **性能优化**: 编译器可以根据类型信息进行优化
- **内存安全**: 类型系统帮助防止内存访问错误
- **代码可读性**: 类型信息使代码意图更清晰

底层语言的优势

- **直接内存操作**: 指针和位操作, 精确控制内存
- **高效执行**: 编译为机器码, 执行效率高
- **硬件抽象**: 提供硬件抽象但不隐藏底层细节
- **系统编程**: 适合操作系统、驱动程序开发
- **资源控制**: 精确控制 CPU、内存等资源

- 1 C、C++ 与 Qt
- 2 C 语言基础知识

- 3 C++ 基本语法
- 4 C++ 面向对象编程

C++ 的起源与设计理念

- **1980 年代**: Bjarne Stroustrup 在贝尔实验室开发
- **设计目标**: 结合 C 的高效性和 Simula 的面向对象特性
- **核心理念**: 零开销抽象原则 (Zero-Overhead Principle)
- **应用领域**: 系统编程、应用开发、嵌入式系统

C++ 标准版本演进

- **C++98** (1998 年) - 第一个国际标准, 确立基础语法
- **C++03** (2003 年) - 技术修正版本, 修复缺陷
- **C++11** (2011 年) - 现代 C++ 开始, 重大更新
- **C++14** (2014 年) - 功能完善和优化
- **C++17** (2017 年) - 标准库增强
- **C++20** (2020 年) - 最新标准, 重大特性

C++11 的重大创新

- **智能指针**: `std::unique_ptr`, `std::shared_ptr`
- **Lambda 表达式**: 函数式编程支持
- **移动语义**: 性能优化的革命性改进
- **auto 关键字**: 类型推导, 简化代码
- **范围 for 循环**: 更简洁的迭代语法
- **nullptr**: 类型安全的空指针

对编程范式的影响

- **从 OOP 到多范式**: 支持面向对象、泛型、函数式编程
- **性能优先**: 移动语义显著提升性能
- **安全性提升**: 智能指针减少内存泄漏
- **代码简化**: `auto` 和 `Lambda` 减少样板代码

C++14 增强

- **泛型 Lambda**: 支持 auto 参数
- **变量模板**: 模板变量声明
- **数字分隔符**: 提高可读性
- **std::make_unique**: 智能指针工厂

C++17 新特性

- **结构化绑定**: 多返回值处理
- **std::optional**: 可选值类型
- **std::variant**: 类型安全联合
- **并行算法**: 标准库并行化

C++20 重大更新

- **概念 (Concepts)**: 模板约束系统
- **协程 (Coroutines)**: 异步编程支持
- **模块 (Modules)**: 编译时依赖管理
- **三向比较**: `<=>` 操作符

多范式编程语言

- **面向对象编程**：封装、继承、多态，支持抽象和重用
- **泛型编程**：模板、STL，编译时多态
- **过程式编程**：函数、模块化设计（继承自 C 语言）
- **函数式编程**：Lambda 表达式、算法库

C 语言兼容性

- **完全兼容**：C++ 是 C 的超集，几乎所有的 C 代码都是有效的 C++ 代码
- **底层控制**：保留 C 语言的指针、内存管理、位操作等底层特性
- **性能优先**：零开销抽象原则，高级特性不带来性能损失
- **系统编程**：适合操作系统、驱动程序、嵌入式系统开发

主要应用领域

- **系统软件**：操作系统、驱动程序
- **游戏开发**：引擎、图形渲染
- **嵌入式系统**：实时控制、IoT 设备
- **高性能计算**：科学计算、金融交易
- **桌面应用**：Qt、MFC、WPF

技术优势

- **跨平台**：一次编写，多处运行
- **类型安全**：编译时类型检查
- **向后兼容**：C 语言兼容性，平滑过渡
- **标准化**：ISO 标准，长期稳定
- **生态系统**：丰富的库和工具链
- **底层控制**：直接内存操作，高效执行

与其他语言的比较

- **vs C**：更强的类型安全，面向对象支持，但保持底层控制能力
- **vs Java**：更高的性能，更直接的内存控制，无虚拟机开销
- **vs Python**：编译型语言，执行效率更高，强类型检查
- **共同特点**：都是强类型语言 编译时检查 适合系统编程

Qt 框架的 C++ 基础

- **原生 C++ 框架**: Qt 完全用 C++ 编写, 无虚拟机依赖
- **面向对象设计**: 充分利用 C++ 的封装、继承、多态特性
- **现代 C++ 支持**: 支持 C++11 及以后特性, 包括智能指针、Lambda 等
- **跨平台抽象**: 统一不同操作系统的 API, 实现真正的跨平台

Qt 的 C++ 特性应用

- **信号槽机制**: 基于函数指针和回调
- **元对象系统**: 运行时类型信息和反射
- **内存管理**: 父子对象关系, 自动清理
- **模板应用**: 容器类、算法库

学习 C++ 的优势

- **类型安全**: 编译时错误检查, 减少运行时错误
- **性能优化**: C++ 的高效执行
- **生态系统**: 丰富的第三方库和工具支持

Qt 与 C++ 的协同优势

开发效率提升

- **快速原型**: Qt Designer 可视化设计
- **代码生成**: uic 工具自动生成 UI 代码
- **调试支持**: 集成调试器和性能分析
- **文档完善**: 详细的 API 文档和示例

性能与稳定性

- **编译优化**: C++ 编译器深度优化
- **内存安全**: RAII 和智能指针
- **异常处理**: C++ 异常机制
- **线程安全**: Qt 的线程模型

学习路径建议

- **基础阶段**: 掌握 C++ 基本语法和面向对象概念
- **进阶阶段**: 学习现代 C++ 特性和 Qt 框架
- **实践阶段**: 结合项目开发, 深入理解两者结合
- **高级阶段**: 性能优化、设计模式、架构设计

C++ 对 C 的语法扩展

- **完全兼容 C**: C++ 是 C 的超集, 所有 C 代码都是有效的 C++ 代码
- **面向对象扩展**: 类、对象、继承、多态
- **函数重载**: 同名函数不同参数类型
- **引用类型**: 变量的别名, 避免指针的复杂性
- **命名空间**: 避免名称冲突
- **异常处理**: try-catch 机制

C++ 语法特点

- **强类型**: 编译时类型检查, 类型安全
- **静态类型**: 类型在编译时确定
- **编译型语言**: 直接编译为机器码
- **多范式**: 支持过程式、面向对象、泛型编程

C++ 数据类型扩展

- **bool 类型**: true 和 false, C++ 原生支持
- **wchar_t**: 宽字符类型, 支持 Unicode
- **引用类型**: int&、double& 等
- **类类型**: 用户自定义类型
- **模板类型**: 泛型类型

类型安全增强

- **类型转换**: 显式类型转换, 避免隐式转换错误
- **const 修饰符**: 常量类型, 防止意外修改
- **类型推导**: auto 关键字, 编译器自动推导类型
- **nullptr**: 类型安全的空指针 (C++11)

C++ 变量声明增强

- **引用声明**: `int& ref = x;` 创建变量的别名
- **const 引用**: `const int& ref = x;` 只读引用
- **auto 关键字**: `auto x = 42;` 自动类型推导
- **decltype**: `decltype(expr)` 推导表达式类型

常量声明

- **const 常量**: `const int MAX = 100;`
- **constexpr**: 编译时常量 (C++11)
- **constexpr**: 编译时初始化 (C++20)
- **const 成员函数**: `void func() const;`



```
1  #include <iostream>
2  #include <windows.h>
3
4  int main() {
5      SetConsoleOutputCP(65001);
6
7      int var = 10;
8      const int CONST_VAR = 20;
9
10     std::cout << "变量: " << var << std::endl;
11     std::cout << "常量: " << CONST_VAR << std::endl;
12
13     std::cin.get();
14     return 0;
15 }
```

C++ 运算符扩展

- **作用域解析运算符**: `::` 访问全局变量或类成员
- **成员访问运算符**: `.` 和 `->` 访问对象成员
- **类型转换运算符**: `static_cast`、`dynamic_cast`
- **条件运算符**: `?:` 三元运算符

表达式增强

- **函数调用表达式**: 支持函数重载和默认参数
- **成员函数调用**: `obj.func()` 或 `ptr->func()`
- **模板实例化**: `vector<int> v;`
- **Lambda 表达式**: `[] (int x) { return x * 2; }`

C++ 控制结构增强

- **范围 for 循环**: `for (auto& item : container)` (C++11)
- **初始化语句**: `if (auto it = find(x); it != end())`
- **结构化绑定**: `auto [x, y] = pair;` (C++17)
- **switch 增强**: 支持初始化语句和 fallthrough

异常处理

- **try-catch**: 异常捕获和处理
- **throw**: 抛出异常
- **noexcept**: 指定函数不抛出异常
- **RAII**: 资源获取即初始化

C++ 函数特性

- **函数重载**: 同名函数不同参数类型或数量
- **默认参数**: `void func(int x = 0, int y = 0);`
- **内联函数**: `inline` 关键字
- **函数模板**: 泛型函数
- **Lambda 表达式**: 匿名函数

函数调用约定

- **值传递**: `void func(int x);` 传递副本
- **引用传递**: `void func(int& x);` 避免拷贝
- **const 引用**: `void func(const int& x);` 只读
- **右值引用**: `void func(int&& x);` 移动语义

C++ 函数示例

```
1  #include <iostream>
2  #include <string>
3  #include <windows.h>
4
5  // 函数声明
6  int add(int a, int b);
7  void printMessage(const std::string& message);
8  int factorial(int n);
9
10 int main() {
11     SetConsoleOutputCP(65001);
12
13     int result = add(5, 3);
14     std::cout << "5 + 3 = " << result << std::endl;
15
16     std::cin.get();
17     return 0;
18 }
```

```
20 // 函数定义
21 int add(int a, int b) {
22     return a + b;
23 }
24
25 // 引用参数示例
26 void printMessage(const std::string& message) {
27     std::cout << "Message: " << message << std::endl;
28 }
29
30 // 递归函数示例
31 int factorial(int n) {
32     if (n <= 1) {
33         return 1;
34     }
35     return n * factorial(n - 1);
36 }
```


C++ 函数重载

```
1  #include <iostream>
2  #include <string>
3  #include <windows.h>
4
5  // 重载函数: 不同参数类型
6  int add(int a, int b) {
7      return a + b;
8  }
9
10 double add(double a, double b) {
11     return a + b;
12 }
13
14 // 重载函数: 不同参数数量
15 int add(int a, int b, int c) {
16     return a + b + c;
17 }
18
19 // 重载函数: 不同参数类型
20 std::string add(const std::string& a, const
    ↪ std::string& b) {
```

```
21     return a + b;
22 }
23
24 void print(int i) { std::cout << "整数: " << i <<
    ↪ std::endl; }
25 void print(double f) { std::cout << "浮点数: " << f <<
    ↪ std::endl; }
26 void print(const char* s) { std::cout << "字符串: " <<
    ↪ s << std::endl; }
27
28 int main() {
29     SetConsoleOutputCP(65001);
30
31     print(10);
32     print(3.14);
33     print("Hello");
34
35     std::cin.get();
36     return 0;
37 }
```

Lambda 表达式

Lambda 表达式语法

- `[capture](parameters) -> return_type { body }`
- `capture`: 捕获外部变量
- `parameters`: 参数
- `return_type`: 返回类型
- `body`: 函数体

Lambda 表达式捕获

- `[]`: 不捕获任何外部变量。若在 Lambda 体内使用未捕获的外部变量会导致编译错误。
- `[x, &y]`: `x` 以值捕获, `y` 以引用捕获。
- `[&]`: 所有被用到的外部变量都以引用方式捕获。
- `[=]`: 所有被用到的外部变量都以值方式捕获。
- `[&, x]`: `x` 以值方式捕获, 其余变量以引用方式捕获。
- `[=, &z]`: `z` 以引用方式捕获, 其余变量以值方式捕获。

C++ Lambda 表达式示例

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5  #include <windows.h>
6
7  int main() {
8      SetConsoleOutputCP(65001);
9
10     std::vector<int> numbers = {1, 2, 3, 4, 5};
11
12     // Lambda 表达式
13     std::for_each(numbers.begin(), numbers.end(), [](int n) {
14         std::cout << n << " ";
15     });
16     std::cin.get();
17     return 0;
18 }
```

C++ 数组增强与细节

- **原生数组**: 如 `int arr[5];`, 大小固定, 不能自动推断长度, 越界不安全。
- **`std::array`**: C++11 引入, 固定大小、类型安全, 支持标准库算法, 如 `std::array<int, 5> arr;`。
- **`std::vector`**: 动态数组, 自动管理内存, 可动态扩容, 常用操作有 `push_back`、`size`、`at` 等。
- **初始化列表**: 可用于原生数组、`std::array`、`std::vector`, 如 `int arr[] = {1, 2, 3};` 或 `std::vector<int> v = {1, 2, 3};`。
- **范围 for 循环**: C++11 起支持, 简化遍历, 如 `for (auto x : arr) {}`, 适用于原生数组、`std::array`、`std::vector`。
- **迭代器遍历**: `std::vector` 和 `std::array` 支持迭代器, 可用 `begin()` 和 `end()` 进行灵活遍历。
- **内存安全**: `std::vector` 的 `at()` 方法有越界检查, 原生数组无越界保护。
- **多维数组**: 原生数组、`std::array`、`std::vector` 均可实现多维数组。



```
1  #include <iostream>
2  #include <array>
3  #include <windows.h>
4
5  int main() {
6      SetConsoleOutputCP(65001);
7
8      // 原生数组示例
9      int arr[5] = {1, 2, 3, 4, 5};
10
11     for (int i = 0; i < 5; i++) {
12         std::cout << arr[i] << " ";
13     }
14     std::cout << std::endl;
15
16     std::cin.get();
17     return 0;
18 }
```

C++std::array 示例

```
1  #include <array>
2  #include <iostream>
3  #include <windows.h>
4
5  int main() {
6      SetConsoleOutputCP(65001);
7
8      std::array<int, 5> arr = {1, 2, 3, 4, 5};
9
10     for (auto num : arr) {
11         std::cout << num << " ";
12     }
13     std::cout << std::endl;
14
15     std::cin.get();
16     return 0;
17 }
```

C++std::vector 示例

```
1  #include <iostream>
2  #include <vector>
3  #include <windows.h>
4
5  int main() {
6      SetConsoleOutputCP(65001);
7      std::vector<int> vec = {1, 2, 3, 4, 5}; // 初始化
8      for (auto x : vec) {
9          std::cout << x << " " << std::endl;
10     }
11     std::cout << "vec.size() = " << vec.size() <<
    ↵ std::endl;
12     std::cout << "vec.front() = " << vec.front() <<
    ↵ std::endl;
13     std::cout << "vec.back() = " << vec.back() <<
    ↵ std::endl;
14     std::cout << "vec.at(2) = " << vec.at(2) <<
    ↵ std::endl;
15     std::cout << "vec[2] = " << vec[2] << std::endl;
16     std::cout << "vec.empty() = " << vec.empty() <<
    ↵ std::endl;
17     //迭代器
```

```
18     for (auto it = vec.begin(); it != vec.end();
    ↵ ++it) {
19         std::cout << *it << " " << std::endl;
20     }
21     //插入元素
22     vec.insert(vec.begin() + 2, 10);
23     for (auto x : vec) {
24         std::cout << x << " " << std::endl;
25     }
26     //删除元素
27     vec.erase(vec.begin() + 2);
28     for (auto x : vec) {
29         std::cout << x << " " << std::endl;
30     }
31     //清空元素
32     vec.clear();
33     for (auto x : vec) {
34         std::cout << x << " " << std::endl;
35     }
36     std::cin.get();
37     return 0;
38 }
```



C++ 指针增强

- **nullptr**: 类型安全的空指针 (C++11)
- **智能指针**: `std::unique_ptr`、`std::shared_ptr`
- **void 指针**: `void*` 通用指针
- **函数指针**: 指向函数的指针

C++ 引用

- **左值引用**: `int& ref = x;`
- **常量引用**: `const int& ref = x;`
- **右值引用**: `int&& ref = 42;` (C++11)
- **引用 vs 指针**: 引用更安全, 不能为空

C++ 指针示例

```
1  #include <iostream>
2  #include <windows.h>
3
4  int main() {
5      SetConsoleOutputCP(65001);
6
7      int var = 10;
8      int* ptr = &var;
9
10     std::cout << "变量值: " << var << std::endl;
11     std::cout << "指针值: " << *ptr << std::endl;
12
13     *ptr = 20;
14     std::cout << "修改后变量值: " << var << std::endl;
15
16     std::cin.get();
17     return 0;
18 }
```

C++ 引用示例

```
1  #include <iostream>
2  #include <windows.h>
3
4  int main() {
5      SetConsoleOutputCP(65001);
6
7      int x = 10;
8      int& ref = x;
9
10     std::cout << "x = " << x << std::endl;
11     std::cout << "ref = " << ref << std::endl;
12
13     ref = 20;
14     std::cout << "修改后 x = " << x << std::endl;
15 }
```

- 1 C、C++ 与 Qt
- 2 C 语言基础知识
- 3 C++ 基本语法
- 4 C++ 面向对象编程

什么是类？

- 类是对象的蓝图或模板
- 类定义了对象的结构和行为
- 类包含数据成员和成员函数
- 类是抽象的，对象是具体的

什么是对象？

- 对象是类的实例
- 对象具有类定义的结构和行为
- 对象是具体的，有自己的状态和行为

面向对象的优点

- 封装：将数据和方法封装在类中，隐藏实现细节，只暴露接口
- 继承：继承父类的方法和属性，实现代码复用
- 多态：实现接口的统一，不同实现方式

C++ 类的定义、对象的创建与使用

```
1  #include <iostream>
2  #include <string>
3  #include <windows.h>
4
5  class Student {
6  public:
7      Student(std::string name, int age) : name(name),
        ↪ age(age) {}
8
9      void display() {
10         std::cout << "姓名: " << name << ", 年龄: " <<
        ↪ age << std::endl;
11     }
12
13 private:
14     std::string name;
15     int age;
16 };
```

```
18 int main() {
19     SetConsoleOutputCP(65001);
20
21     Student student("张三", 20);
22     student.display();
23
24     std::cin.get();
25     return 0;
26 }
```

- 基类：父类，被继承的类，基类中定义的属性和方法可以被派生类继承
- 派生类：子类，继承父类的类，在基类的基础上扩展新的属性和方法
- 继承：派生类继承基类的属性和方法，派生类可以重写基类的方法

```
1  #include <iostream>
2  #include <windows.h>
3
4  class Animal {
5  public:
6      void eat() {
7          std::cout << "动物在吃" << std::endl;
8      }
9  };
10
11 class Dog : public Animal {
12 public:
13     void bark() {
14         std::cout << "狗在叫" << std::endl;
15     }
16 };
```

```
18  int main() {
19      SetConsoleTitle("C++ 继承");
20
21      Dog dog;
22      dog.eat();
23      dog.bark();
24
25      std::system("pause");
26      return 0;
27 }
```

```
1  #include <iostream>
2  #include <windows.h>
3
4  class Base1 {
5  public:
6      void show1() { std::cout << "Base1 show" <<
7          ↵ std::endl; }
8  };
9
10 class Base2 {
11 public:
12     void show2() { std::cout << "Base2 show" <<
13         ↵ std::endl; }
14 };
15
16 class Derived : public Base1, public Base2 {
17 public:
18     void show() { std::cout << "Derived show" <<
19         ↵ std::endl; }
20 };
21
```

```
19 int main() {
20     SetConsoleOutputCP(65001);
21
22     Derived d;
23     d.show1();
24     d.show2();
25     d.show();
26
27     std::cin.get();
28     return 0;
29 }
```

C++ 继承的规则

- 公有继承：基类的 public 和 protected 成员在派生类中保持原样，private 成员不可访问
- 保护继承：基类的 public 和 protected 成员在派生类中变为 protected，private 成员不可访问
- 私有继承：基类的 public 和 protected 成员在派生类中变为 private，private 成员不可访问

```
1  #include <iostream>
2  #include <windows.h>
3
4  // 基类，包含 public、protected、private 成员
5  class Base {
6  public:
7      int publicVar;
8      void publicFunc() { std::cout << "Base
9      ↪ publicFunc" << std::endl; }
10 protected:
11     int protectedVar;
12     void protectedFunc() { std::cout << "Base
13     ↪ protectedFunc" << std::endl; }
14 private:
15     int privateVar;
16     void privateFunc() { std::cout << "Base
17     ↪ privateFunc" << std::endl; }
18 };
19
```


C++ 继承的规则

```
17 // 公有继承
18 class PublicDerived : public Base {
19 public:
20     void test() {
21         publicVar = 1;           // 可以访问
22         protectedVar = 2;       // 可以访问
23         // privateVar = 3;      // 无法访问
24         publicFunc();           // 可以访问
25         protectedFunc();        // 可以访问
26         // privateFunc();       // 无法访问
27         std::cout << "PublicDerived test" <<
            ↵ std::endl;
28     }
29 };
30
31 // 保护继承
32 class ProtectedDerived : protected Base {
33 public:
34     void test() {
35         publicVar = 1;           // 可以访问
36         protectedVar = 2;       // 可以访问
37         // privateVar = 3;      // 无法访问
```

```
38         publicFunc();           // 可以访问
39         protectedFunc();        // 可以访问
40         // privateFunc();       // 无法访问
41         std::cout << "ProtectedDerived test" <<
            ↵ std::endl;
42     }
43 };
44
45 // 私有继承
46 class PrivateDerived : private Base {
47 public:
48     void test() {
49         publicVar = 1;           // 可以访问
50         protectedVar = 2;       // 可以访问
51         // privateVar = 3;      // 无法访问
52         publicFunc();           // 可以访问
53         protectedFunc();        // 可以访问
54         // privateFunc();       // 无法访问
55         std::cout << "PrivateDerived test" <<
            ↵ std::endl;
56     }
57 };
```

多态

- 多态：派生类可以重写基类的方法，实现不同的行为
- 虚函数：基类中定义的虚函数，派生类可以重写
- 纯虚函数：基类中定义纯虚函数，派生类必须重写

```
1  #include <iostream>
2  #include <windows.h>
3
4  class Shape {
5  public:
6      virtual void draw() { std::cout << "绘制形状" <<
        ↪ std::endl; }
7      virtual ~Shape() = default; // 添加虚析构函数，防止
        ↪ 内存泄漏
8  };
9
10 class Circle : public Shape {
11 public:
12     void draw() override { std::cout << "绘制圆形" <<
        ↪ std::endl; }
13 };
14
15 class Triangle : public Shape {
16 public:
17     void draw() override { std::cout << "绘制三角形"
        ↪ << std::endl; }
```

多态的作用

- 代码复用：派生类直接使用基类的属性和方法，不需要重新实现
- 接口统一：派生类和基类具有相同的接口，可以相互替换
- 扩展性：新增子类时，不需要修改父类，只需要重写子类的方法

```
19
20 int main() {
21     SetConsoleOutputCP(65001);
22
23     Shape* shape1 = new Circle();
24     shape1->draw();
25
26     Shape* shape2 = new Triangle();
27     shape2->draw();
28
29     delete shape1;
30     delete shape2;
31
32     std::cin.get();
33     return 0;
```

智能指针的类型

- **std::unique_ptr** - 独占所有权，不能复制
- **std::shared_ptr** - 共享所有权，引用计数
- **std::weak_ptr** - 弱引用，不增加引用计数
- **std::auto_ptr** - 已废弃，C++17 移除

智能指针的优势

- **自动内存管理** - 避免内存泄漏
- **异常安全** - 异常时自动清理
- **RAII** - 资源获取即初始化
- **线程安全** - shared_ptr 线程安全

什么是引用计数？

- 创建对象时，引用计数为 1
- 当新的指针或者引用指向该对象时，引用计数加 1
- 当指针或者引用不再指向该对象时，引用计数减 1
- 当引用计数为 0 时，对象被销毁，资源被释放

C++ 智能指针示例

```
1  #include <iostream>
2  #include <memory>
3  #include <vector>
4  #include <windows.h>
5  class Resource {
6  public:
7      Resource() { std::cout << "资源创建" << std::endl;
8          ↪ }
9      ~Resource() { std::cout << "资源销毁" <<
10         ↪ std::endl; }
11      void use() { std::cout << "资源使用" << std::endl;
12         ↪ }
13 };
14 int main() {
15     SetConsoleOutputCP(65001);
16     // unique_ptr - 独占所有权
17     std::unique_ptr<Resource> unique =
18         ↪ std::make_unique<Resource>();
19     unique->use();
20     // unique_ptr 不能被复制, 只能被移动
21     std::unique_ptr<Resource> unique2 =
22         ↪ std::move(unique);
23     // shared_ptr - 共享所有权
```

```
19     std::shared_ptr<Resource> shared1 =
20         ↪ std::make_shared<Resource>();
21     {
22         std::shared_ptr<Resource> shared2 = shared1;
23         ↪ // 引用计数 +1
24         shared2->use();
25         std::cout << "Reference count: " <<
26         ↪ shared1.use_count() << std::endl;
27     } // shared2 超出作用域, 引用计数 -1
28     std::cout << "Reference count: " <<
29     ↪ shared1.use_count() << std::endl;
30     // weak_ptr - 弱引用
31     std::weak_ptr<Resource> weak = shared1;
32     std::cout << "weak_ptr reference count: " <<
33     ↪ weak.use_count() << std::endl;
34     // 检查 weak_ptr 是否有效
35     if (auto locked = weak.lock()) { // 如果有效, 使
36         ↪ 用资源
37         locked->use();
38     }
39     std::cin.get();
40     return 0;
41 }
```

- 本章介绍了 C 的基本语法、C++ 的基本语法、Lambda 表达式、数组、指针和引用、面向对象编程、智能指针等概念，通过示例代码，我们了解了这些概念的用法和实现方式
- 在实际开发中，我们可以根据需求选择合适的数据结构和算法，提高代码的效率和可读性
- C++ 的语法还有很多，比如模板、异常处理、多线程、网络编程、数据库编程、图形编程等，这些内容可以参考其他书籍或者网络资源