

C++新特性

C++11 标准为 C++ 语言带来了一系列革命性的改进和新特性，极大地提高了编程效率和代码质量。

```
int main()
{
    Complex c1, c2, result;

    cout<<"Enter first complex number:\n";
    c1.input();
    cout<<"Enter second complex number:\n";
    c2.input();
    // In case of operator overloading of binary operators in C++ prog
    // the object on right hand side
    of operator is always assumed as
    argument by compiler.
    result = c1 - c2;
    result.output();
    return 0;
}

<li><a href="#">Design</a></li>
<li><a href="#">Photo</a></li>
<li><a href="#">Art</a></li>
<li><a href="#">Game</a></li>
<li><a href="#">Film</a></li>
<li><a href="#">TV</a></li>

#posts-list {
    position: relative;
    width: 620px;
    float: left;

#posts-list article {
    position: relative;
    margin-bottom: 50px;
    padding: 30px;
    background: #f3e4e8;
    box-shadow: 3px 3px 0 0 rgba(0, 0, 0, 0.1);

#posts-list article .tape {
    position: absolute;
    top: -15px;
    left: 250px;
    display: block;
    width: 122px;
    height: 35px;
    background: url(/img/tape.png);

#main .page-navigation div {
    position: relative;
    right: 50%;
    box-shadow: 3px 3px 0 0 rgba(0, 0, 0, 0.1);

#main .page-navigation div span {
    margin-bottom: 15px;
    display: inline-block;

}; /* "Pervez Joarder" */
#Facebook "www.facebook.com/pjs.pervez"
#Twitter "https://twitter.com/pervezpjs"
}

Page Template designed by <a href="http://www.luiszuno.com">luiszuno.com</a>
tag << "i";

#main .page-navigation div span {
    margin-bottom: 15px;
    display: inline-block;
}
```

C++ 编译过程

预处理

词法分析

语法分析

语义分析

中间代码生成

优化

目标代码生成

链接

M学长的考研Top帮

预处理 (Preprocessing)

- 在编译源代码之前，预处理器首先对源文件进行操作。
- 这个阶段主要包括：
- **宏定义展开**：使用`#define`定义的宏会被替换为其内容。
- **处理头文件**：`#include`指令会将指定头文件的内容插入到源代码中。
- **删除注释**：预处理器会移除所有注释内容，以简化后续处理。

示例

```
#define PI 3.14
#include <iostream>

int main() {
    std::cout << "圆周率是: " << PI << std::endl;
    return 0;
}
```

预处理后，代码变为：

```
#include <iostream>

int main() {
    std::cout << "圆周率是: " << 3.14 << std::endl;
    return 0;
}
```

词法分析 (Lexical Analysis)



将预处理后的文本分解成一系列符号或标记 (tokens)。

这些tokens包括关键字、标识符、常量、运算符和分隔符等。



示例

对于语句 `int a = 5;`，会分解为以下tokens：

`int`、`a`、`=`、`5` 和 `;`。

语法分析 (Syntax Analysis 或 Parsing)

编译器根据C++的语法规则，将tokens组合成结构化的数据结构，即抽象语法树（AST）。抽象语法树的作用是 为后续的语义分析和代码生成提供便利，使编译器能更准确地理解程序的含义。

```
int main() {  
    int x = 5;  
    int y = 3;  
    int z = x + y;  
    return z;  
}
```

AST示例:

```
Program
|
+-- FunctionDeclaration (int main())
|
+-- CompoundStatement
|
+-- VariableDeclaration (int x)
|   +-- IntegerLiteral (5)
|
+-- VariableDeclaration (int y)
|   +-- IntegerLiteral (3)
|
+-- VariableDeclaration (int z)
|   +-- BinaryExpression (+)
|       +-- Identifier (x)
|       +-- Identifier (y)
|
+-- ReturnStatement
|   +-- Identifier (z)
```

语义分析 (Semantic Analysis)



语义检查

在构建 AST 的同时，编译器进行语义检查，以确保所有变量和函数的声明与使用符合语义规则。



静态语义规则验证

包括类型检查、作用域解析和其他静态语义规则的验证。例如，如果尝试将字符串赋值给整型变量，编译器将在这一阶段报告错误。

中间代码生成 (Intermediate Code Generation)

1

1. 生成中间表示形式

编译器生成一种与特定机器无关的中间表示形式，如字节码或三地址码。

2

2. 便于优化和移植

这种形式便于后续的优化和跨平台移植。

例如，将 `int z = x + y;` 转换为中间代码可能类似于：

```
t1 = x + y
```

```
z = t1
```

优化 (Optimization)

- 编译器对中间代码进行各种优化，以提高运行效率和减少资源消耗。
- 死代码删除
移除永远不会执行的代码。
- 循环优化
减少循环内重复计算的次数。
- 常量传播
将常量值直接替换到表达式中。

目标代码生成 (Code Generation)

生成目标代码

编译器将优化后的中间代码转换为特定计算机架构的目标代码 (Object Code)。

目标代码格式

目标代码通常为汇编语言或直接是二进制格式，能被CPU执行。

```
mov eax, x  
add eax, y  
mov z, eax
```

链接 (Linking)

合并目标文件

链接器将多个目标代码文件和必要的库文件合并，生成最终的可执行文件。

解决地址引用

链接过程中，链接器会解决函数调用和全局变量的地址引用问题。

关联函数

假设有两个源文件 `main.cpp` 和 `utils.cpp`，它们都包含函数 `void foo()`，链接器会将 `main.cpp` 中对 `foo()` 的调用与 `utils.cpp` 中的实现关联起来。

C++新特性

- 提高代码简洁性和可读性
- 增强性能和资源管理
- 支持现代编程范式
- 提升模板和泛型编程能力

自动类型推导 (auto)

原理：允许编译器基于变量的初始化表达式自动推导其类型

//使用前

```
std::vector<int> vec = {1, 2, 3, 4};
```

```
std::vector<int>::iterator it = vec.begin(); // 需要明确指定迭代器的类型
```

//使用后

```
std::vector<int> vec = {1, 2, 3, 4};
```

```
auto it = vec.begin(); // 自动推导迭代器类型
```

auto的优势

简化代码

auto 关键字允许编译器基于变量的初始化表达式自动推导其类型，从而简化代码编写。

减少编译错误

使用 auto 减少因类型错误导致的编译错误，提高代码的可靠性。

适用场景

auto 特别适用于迭代器和复杂函数返回类型的场景，提高开发效率。

维护性提升

减少重复类型声明，使代码更易于维护。

auto的缺点

使用 `auto` 可能会导致代码的**可读性降低**，特别是在代码较为复杂时。虽然 `auto` 简化了变量声明，但有时让人难以直观地理解变量的实际类型。

```
auto x = some_complex_function(); // x 的类型可能不容易从函数名推断
```


Lambda表达式

Lambda表达式是一种轻量级的、匿名的函数对象，允许你在需要函数的地方内联定义和使用函数逻辑。它们通常用于需要临时、小型函数的场景，如算法中的回调、事件处理等。

Lambda表达式的基本语法

```
[capture](parameters) -> return_type {  
    // 函数体  
};
```

- **capture**: 指定Lambda捕获外部变量的方式，可以是值捕获、引用捕获或混合捕获。
- **parameters**: 函数参数列表，与普通函数类似。
- **return_type** (可选): 指定返回类型，编译器通常可以自动推导。
- **函数体**: Lambda执行的代码块。

Lambda举例

以下是一个Lambda表达式的示例：

```
[](int n) { return n % 2 == 0; }
```

- 捕获列表 [] 表示不捕获任何外部变量。
- 参数列表 (int n) 表示Lambda表达式接受一个整数参数n。
- 函数体 { return n % 2 == 0; } 实现了一个简单的逻辑，检查n是否为偶数。

这个Lambda表达式等效于以下的普通函数：

```
bool isEven(int n) { return n % 2 == 0; }
```

Lambda 表达式使用情景

//使用前

```
bool descending(int a, int b) {  
    return a > b;  
}
```

```
std::vector<int> vec = {1, 2, 3, 4};
```

```
std::sort(vec.begin(), vec.end(), descending); // 需要单独定义比较函数
```

//使用后

```
std::vector<int> vec = {1, 2, 3, 4};
```

```
std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; }); // 直接使用 Lambda 表达式
```

捕获外部变量

```
// 假设有一个需要处理的外部变量
int externalValue = 5;
int main() {
    // 使用lambda捕获外部变量并进行操作
    auto lambdaFunc = [externalValue]() {
        std::cout << "Captured external value: " << externalValue << std::endl;
        // 在lambda函数体内部，我们可以直接使用这个被捕获的变量
        externalValue *= 2;
    };
    // 调用lambda表达式
    lambdaFunc();
    // externalValue仍为5
    return 0;
}
```

Lambda 表达式的优势

1

简洁性

Lambda 表达式提供了一种非常简洁的方式来定义一个函数对象。

2

方便的闭包

Lambda 允许方便地捕获外部变量，无需显式传递。

3

易于使用的回调

特别适用于定义回调函数，如作为算法的参数。

Variables in C

age = 20;

variable_name

20

Memory for variable

RAM

与函数的不同

特性	Lambda表达式	普通函数
定义方式	匿名定义，通常在使用的地方内联定义	具名定义，需要显式声明和定义
命名	没有名称，或者通过赋值给变量来调用	有明确的名称，可以在多个地方复用
捕获外部变量	可以通过捕获列表 <code>[]</code> 直接捕获外部变量	无法直接捕获外部变量，需要通过参数传递
作用域	通常在定义它的作用域内有效，可以捕获并持有外部变量	具有全局或类作用域，生命周期贯穿整个程序运行期间

与函数的不同

特性	Lambda表达式	普通函数
内联优化	编译器通常会内联优化，减少函数调用开销	编译器可以选择是否内联，取决于函数的复杂性和优化设置
灵活性	高，可以在定义时捕获不同的外部变量，适应不同的上下文需求	较低，逻辑固定，需要通过参数传递外部数据
语法简洁性	更简洁，适合短小的函数逻辑	需要完整的函数声明和定义，语法较为冗长
使用场景	算法回调、事件处理、临时操作等需要内联定义函数的场景	需要在多个地方复用的逻辑、复杂的功能模块等



智能指针

智能指针是一种特殊的指针类，它可以管理动态分配的内存，帮助减少内存泄漏的风险。C++标准库中提供了几种类型的智能指针，如`std::shared_ptr`和`std::weak_ptr`，它们使得内存管理更加方便和安全。智能指针通过重载了析构函数的方式，在对象超出作用域时自动释放内存。

M学长的考研Top帮

C++指针概念

C++指针有在程序中直接访问和操作内存地址的能力。

指针是一个变量，它存储了一个内存地址，可以通过解引用操作符 * 来获取该地址存储的值。

指针可以通过new和delete操作符动态地分配和释放内存。

```
int x = 10;
// 使用 new 动态分配一个整数的内存
int* ptr = new int;
*ptr = x; // 将 x 的值存储在动态分配的内存中
int value = *ptr; // value 等于 10，即动态分配内存中的值

// 使用完动态分配的内存后，记得释放它
delete ptr;
```

C++指针的优势

1

提高程序效率

使用指针可以直接操作内存地址，省去了复制变量的时间和空间。

2

灵活的内存管理

指针可以用于动态分配内存，使程序能够更加灵活地管理内存。

3

实现数据结构

指针可以用于实现数据结构，如链表和树等，以及处理大型数据集合时进行高效的

操作。

4

函数交互和多态

通过传递指针参数可以直接修改函数外的变量。指针还可以用于实现多态，提高编程的灵活性和效率。

原始指针的问题

メモリ

0x104FB6CCFFAF00A1		
0x104FB6CCFFAF00A2	5	int x = 5;
0x104FB6CCFFAF00A3		
0x104FB6CCFFAF00A4		
0x104FB6CCFFAF00A5		
0x104FB6CCFFAF00A6		
0x104FB6CCFFAF00A7		
0x104FB6CCFFAF00A8	0xA2	int* y = &x;
0x104FB6CCFFAF00A9	0x00	
0x104FB6CCFFAF00AA	0xAF	
0x104FB6CCFFAF00AB	0xFF	
0x104FB6CCFFAF00AC	0xCC	
0x104FB6CCFFAF00AD	0x86	
0x104FB6CCFFAF00AE	0x4F	
0x104FB6CCFFAF00AF	0x10	
0x104FB6CCFFAF00B0		

↑

1

内存泄漏

忘记释放动态分配的内存会导致内存泄漏。

2

悬挂指针

指向已释放内存的原始指针可能变成悬挂指针。

3

难以维护

手动管理内存容易出错，使代码难以维护。

空指针和野指针

空指针是一个指针变量，它不指向任何有效的内存地址。它通常被初始化为`nullptr`（C++11及以后版本）或`NULL`（传统C++），用于表示指针当前没有指向任何对象

野指针是指向已经被释放或未初始化的内存地址的指针。它们指向的内存可能已经被重新分配给其他变量或程序部分，导致未定义的行为。

智能指针

智能指针 (Smart Pointer) 是C++标准库提供的一种封装普通指针的类模板，用于自动管理动态内存的生命周期，避免手动delete，从而减少内存泄漏和悬空指针（野指针）的问题。

智能指针

使用 `unique_ptr` 前：

```
int* p = new int(10);  
// ... 使用 p  
delete p; // 必须手动删除
```

使用 `unique_ptr` 后：

```
std::unique_ptr<int> up(new int(10)); // 自动管理内存  
// 当 up
```

离开作用域时，它指向的内存会被自动释放

智能指针分类

- ① `std::unique_ptr`
独占拥有权的智能指针，只能有一个 `unique_ptr` 指向对象。
- ② `std::shared_ptr`
共享拥有权的智能指针，多个 `shared_ptr` 可以指向相同的对象。
- ③ `std::weak_ptr`
配合 `shared_ptr` 使用，作为观察者，不影响引用计数。

智能指针的优势

自动内存管理

智能指针自动管理内存，避免了内存泄漏和悬挂指针的问题。

1

2

方便且安全

使用智能指针可以更方便地处理对象的生命周期，使代码更安全。

3

资源管理

智能指针不仅可以用于内存管理，还可以用于管理其他资源。

智能指针的核心原理

1. RAII (Resource Acquisition Is Initialization): 智能指针将资源管理封装在其对象的生命周期内, 当智能指针对象被构造时获取资源 (分配内存), 当智能指针对象被销毁时释放资源 (释放内存)。
2. 引用计数和控制块: `shared_ptr` 使用引用计数来跟踪所有 `shared_ptr` 实例对同一对象的引用, 确保在最后一个 `shared_ptr` 离开作用域时自动释放对象。
3. 析构函数: 智能指针的析构函数会自动调用 `delete` 或其他自定义删除器, 确保所管理对象被正确释放。

C++11 范围for循环

范围for循环允许方便地遍历容器，使得代码更加简洁易读。

使用传统for循环遍历数组：

```
#include <iostream>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; ++i) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

C++11 范围for循环

使用C++11范围for循环遍历数组：

```
#include <iostream>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    for (int num : arr) {
        std::cout << num << " ";
    }
    return 0;
}
```



C++11 初始化列表

初始化列表用于容器和对象的统一初始化方式。

```
int arr[] = {1, 2, 3}; // 数组初始化  
std::vector<int> vec = {1, 2, 3}; // 容器初始化
```

C++11 线程支持库

线程支持库支持多线程编程，提高程序的并发性能。

C++11 原子操作库

原子操作库提供了原子操作的支持，用于多线程中的数据同步。

C++11 正则表达式库

正则表达式库用于字符串模式匹配和搜索。

C++11 类型推导 decltype

decltype 用于获取表达式的类型，进一步简化代码。

C++14 新特性

返回类型推导

函数返回类型可以用 auto 关键字自动推导。

泛型 Lambda 表达式

Lambda 表达式可以使用 auto 在参数中实现参数类型推导。

二进制字面量

支持二进制数表示。

C++17 新特性

结构化绑定

允许将对象或数组的多个成员绑定到一组变量。

编译时 if

允许在编译时进行条件编译。

字符串视图

提供了对字符串的轻量级非拥有引用。

文件系统库

提供了对文件系统的操作能力。



面试常问问题

M学长的考研Top帮

Q:为什么要使用智能指针，而不是原生指针？

A: 智能指针自动管理内存，避免了手动 `delete`，减少了内存泄漏、悬空指针和双重释放等问题。在 C++ 中，常用的智能指针有 `std::unique_ptr`、`std::shared_ptr` 和 `std::weak_ptr`。

Q:循环引用的问题如何避免

A: 循环引用发生在两个或多个 `std::shared_ptr` 互相引用对方，导致引用计数无法归零，从而内存无法被释放。

- 避免循环引用的方法：

使用 `std::weak_ptr`：

使用 `std::weak_ptr` 代替 `std::shared_ptr`，打破循环引用。`weak_ptr` 只持有对象的弱引用，不会增加引用计数

Q: `nullptr` 和 `NULL` 有什么区别?

回答: `nullptr` 和 `NULL` 的主要区别在于**类型安全**和**表达的语义**:

- `NULL` 是 `0` 的宏:

在 C++ 中, `NULL` 通常被定义为整数 `0`, 这意味着 `NULL` 是一个**整数常量**。所以, `NULL` 可以被隐式转换为任何指针类型、整数类型, 甚至可以导致类型不明确的问题。

```
#define NULL 0
```

- `nullptr` 是一个**指针类型**:

在 C++11 中引入的 `nullptr` 是一个新的关键字, 表示一个**空指针常量**。它有一种特殊的类型, 叫做 `std::nullptr_t`, 可以隐式转换为任意指针或成员指针类型, 但不能转换为整数类型。这让 `nullptr` 更加类型安全, 避免了 `NULL` 带来的歧义。



谢谢大家

M学长的考研Top帮