现代C++速查手册

> *"作者：刘虹辰*
>
> *GitHub: https://github.com/JohnsonSii"*

该手册旨在以最易理解、最高效的手法向您介绍现代C++的基础知识及主流特性。我编写该手册采用了一种较为前沿的写作方法，以最简洁的语言陈述最直观的内容，从而规避大量`废话`所带来的认知难度，让您以一种如出生时认知世界的直观方式，低记忆成本的掌握繁琐复杂的语法知识。

目录

条件语句：

循环：

3. 函数

函数声明和定义：

函数参数：

返回值：

函数重载：

4. 类和对象

类的声明和定义：

构造函数和析构函数：

成员函数和数据成员：

静态成员：

5. 继承和多态

基类和派生类：

虚函数和多态：

抽象类和纯虚函数：

6. 友元函数和友元类

7. 操作符重载

8. 模板

函数模板：

类模板：

9. 模板元编程（TMP）

10. 异常处理

11. 命名空间

12. 标准模板库 (STL)

容器：

(1) 初始化

(2) 插入元素

(3) 访问元素

(4) 删除元素

(5) 大小和容量

(6) 其他常用操作

(7) 查找和算法

迭代器：

算法：

13. 动态内存管理

new 和 delete：

14. 指针和引用

指针：

引用：

15. 输入/输出 (I/O)

控制台I/O：

文件I/O：

16. 预处理器指令

包含头文件：

宏定义：

条件编译：

C++11 里程碑式的改变

1. 自动类型推导
2. 范围for循环
3. 初始化列表
4. Lambda表达式
5. 智能指针
6. 右值引用和移动语义

右值和左值：

右值引用：

移动语义：

移动构造函数和移动赋值运算符：

7. `nullptr`和强类型枚举

nullptr：

强类型枚举：

8. 继承构造

9. 委托构造函数
10. 默认和删除的函数
11. 模板元编程（TMP）增强
    基本概念：
    条件编译`std::enable_if`：
    TMP的应用：
12. `constexpr`常量表达式
13. `noexcept`无异常抛出规定
14. 继续增强的STL
    数组`std::array`：
    单链表`std::forward_list`：
    哈希表`std::unordered_map`和`std::unordered_set`：
    元组`std::tuple`：
        (1) 访问元素
        (2) 获取tuple的大小
        (3) 遍历tuple的元素
        (4) 比较tuples
        (5) 连接两个tuples
        (6) 应用函数到tuple (C++17特性)
    时间库`std::chrono`：
15. 变长模板参数
    基本定义：
    使用`sizeof...`获取参数数量：
    使用`std::tuple`与变长模板参数：
    递归展开：
    折叠表达式 (C++17特性)：
16. 异步和多线程
    线程库`std::thread`：
    互斥量和锁管理器：
    异步`std::future`和`std::async`：
    条件变量`std::condition_variable`：

C++98 基础概念

由于我尽可能的希望能向您提供可高效查询的手册，并非入门教程，所以默认您掌握一定的语言开发基础。接下来我将向您介绍C++98的相关基础知识和主流特性。

1. 基本数据类型

C++提供了一系列的基本数据类型来表示整数、浮点数、字符和布尔值等。任何一门语言都不能脱离计算机设计的底层逻辑，我们描述一切事物都必须由基本数据类型支撑。C++98与传统C99在基础数据类型上并无太大差异，其仍然受到操作系统的架构不同而产生差异，我将向您介绍一些常用的数据类型以便解决常规问题。

整型：

整数类型有多种，包括`int`，`short`，`long`和`long long`。它们的大小和范围取决于平台和编译器。

```
int a = 10;  // 通常是32位
short b = 5;  // 通常是16位
long c = 1000L;  // 通常是32位或64位
long long d = 1000000LL;  // 通常是64位
```

浮点型：

用于表示小数。`float`通常有32位，而`double`有64位。还有`long double`，其精度至少与`double`相同，但可能更高。

```
float x = 3.14f;
double y = 2.71828;
long double z = 3.14159265358979323 8L;
```

字符：

`char`用于表示单个字符。还有`wchar_t`，用于宽字符集。

```
char ch = 'A';
wchar_t wch = L'我';
```

布尔：

`bool`只有两个值：`true`和`false`。

```
bool isTrue = true;
bool isFalse = false;
```

2. 控制结构

一门语言最重要的除了数据类型无异于就是控制结构，了解一门编程语言我们也往往遵从于`先声明变量后分支循环`的原则去学习。C++98的分支与循环也与C99并无差异，只是从C++11起编译器提供了其他更易于遍历容器的分支语句及控制结构局部变量声明等更高级的用法，后面我们会详细介绍。

条件语句：

条件语句常见的即为`if`语句和`switch`语句，这与大多数编程语言类似。需要关注的点只有一个，那就是多分支条件判断仅执行到第一个满足条件的判例便跳出，我想这并不需要过多介绍。

```
if (a > b) {
    // 如果a大于b，则执行此代码块
} else if (a == b) {
    // 如果a等于b，则执行此代码块
} else {
    // 否则，执行此代码块
}

switch(a) {
    case 1:
        // 如果a等于1，执行此代码块
        break;
    case 2:
        // 如果a等于2，执行此代码块
        break;
    default:
        // 如果a不等于上述任何值，执行此代码块
}
```

循环：

`for`循环是几乎所有语言最常用的循环语法，C++98极大的依赖于容器长度，所以在使用`for`循环时我们需要预先知道循环执行的次数。切记，不要擅作主张的写出在循环体内改变循环变量这样大大降低可读性的代码，即便C++支持您这样操作，但可读性标志着您代码的可维护性。

```
for (int i = 0; i < 10; i++) {
    // 重复执行此代码块10次
}

int j = 0;
while (j < 10) {
    // 当j小于10时，重复执行此代码块
    j++;
}

int k = 0;
do {
    // 先执行此代码块，然后检查k是否小于10
    k++;
} while (k < 10);
```

3. 函数

函数是一组语句，它们在程序中被赋予一个名字。当这个名字在程序中被调用时，这组语句将被执行。我希望给您一个更直观的理解，在任何编程语言中，大括号`{}`包裹的语句一般被认为是一个具备作用域的结构范围，您可以在几乎任何地方见到它。

函数声明和定义：

C++的任何标准中与C标准相同，在您使用函数之前需要在`main`函数或`main`函数之前进行声明，而不能像`Python`那样颠覆上下文顺序。如果您喜欢将函数名称和类型进行初始化的过程称为`声明`，那么实现其内容的过程则应该被称为`定义`，除此之外也可以被称为`定义`与`实现`。值得注意的是，在传统C++标准中我们一般在头文件中声明函数充当类似接口的作用，并在源函数中实现它，这是函数是编程步入面向对象（OOP）的阶梯，后续我们会详细介绍面向对象编程的精髓。

```
// 声明
int add(int a, int b);

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

函数参数：

函数的参数可以是值传递、引用传递或指针传递，您可以通过下面的案例对其进行直观的认识。

```
void byValue(int x) { x = 10; }
void byReference(int &x) { x = 10; }
void byPointer(int *x) { *x = 10; }
```

返回值：

函数可以返回一个值或返回`void`。

```
int getFive() { return 5; }
void printHello() { std::cout << "Hello"; }
```

函数重载：

同一个函数名可以有多个定义，只要它们的参数列表不同，即可完成重载。在`Python`中函数天生能够被进行重写，其结果呈现为简单的覆盖式声明或实现，而C++并不支持，请不要声明或实现相同名称、参数类型、返回值类型的函数。

```
void print(int i) { std::cout << i; }
void print(double d) { std::cout << d; }
```

4. 类和对象

类和对象被用作C++面向对象编程（OOP）的核心，它的作用在传统C++中起到了开创性作用，也为其他编程语言的设计奠定了扎实的基础。

类的声明和定义：

此处使用了C++98提供的类构造器初始化列表方式，声明了类的成员属性`width`和`height`的初始值，我们可以轻松在类内部或友元类中访问到它的私有属性。

```
class Rectangle {
public:
    Rectangle(int w, int h) : width(w), height(h)
{}
    int area() { return width * height; }
private:
    int width, height;
};
```

构造函数和析构函数：

用于初始化和清理对象。

```
class MyClass {
public:
    MyClass() { std::cout << "Constructor
called!"; }
    ~MyClass() { std::cout << "Destructor
called!"; }
};
```

成员函数和数据成员：

类中的函数和变量。

```
class Circle {
public:
    Circle(double r) : radius(r) {}
    double area() { return 3.14159 * radius *
radius; }
private:
    double radius;
};
```

静态成员：

属于类本身，而不是类的任何特定对象。被静态声明的属性或方法随着类模板的第一次构建而存在，其不受类实例的销毁而销毁。

```
class MyClass {
public:
    static int count;
    MyClass() { count++; }
};
int MyClass::count = 0;  // 初始化静态成员
```

5. 继承和多态

基类和派生类：

继承允许创建一个新类，继承现有类的属性和方法。这有助于实现代码重用和建立类之间的关系。

```
class Animal {
public:
    void eat() { std::cout << "Eating"; }
};

class Dog : public Animal {
public:
    void bark() { std::cout << "Barking"; }
};
```

虚函数和多态：

多态允许使用父类的指针或引用来调用子类的方法。这通常通过虚函数来实现。

```
class Base {
public:
    virtual void show() { std::cout << "Base"; }
};

class Derived : public Base {
public:
    void show() { std::cout << "Derived"; }
};

Base* obj = new Derived();
obj->show();  // 输出 "Derived"
```

抽象类和纯虚函数：

不能实例化的类。

```
class AbstractClass {
public:
    virtual void pureVirtualFunction() = 0;
};
```

6. 友元函数和友元类

友元函数是一个函数，它可以访问类的私有和受保护成员，尽管它不是类的成员。友元类的所有成员函数都是另一个类的友元。

```
class Box {
private:
    double width;
public:
    friend void printWidth(Box b);
};

void printWidth(Box b) {
    std::cout << "Width: " << b.width <<
std::endl;
}
```

7. 操作符重载

允许为用户定义的类型重载现有的操作符。

```
class Complex {
public:
    Complex(int r, int i) : real(r), imag(i) {}
    Complex operator + (Complex const &obj) {
        return Complex(real + obj.real, imag +
obj.imag);
    }
private:
    int real, imag;
};

Complex c1(3, 4), c2(1, 2);
Complex c3 = c1 + c2;
```

8. 模板

类模板作为C++的精髓，我们在后续介绍C++11标准以上的新特性时会详细讲解模板的概念。

函数模板：

允许为多种数据类型编写一个函数。

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

类模板：

允许为多种数据类型编写一个类。

```
template <typename T>
class Box {
public:
    Box(T i) : item(i) {}
    T getItem() { return item; }
private:
    T item;
};
```

9. 模板元编程（TMP）

模板元编程（TMP）是C++的一个高级特性，它允许开发者在编译时执行计算。尽管TMP在C++98中已经存在，但它在这个版本中主要是作为一种技巧或黑魔法被使用，因为它的语法和用法都相对复杂。

在C++98中，TMP主要依赖于模板特化和模板参数来实现编译时的递归和计算。例如，计算编译时的斐波那契数列和阶乘都是TMP的经典应用。

```
template<int N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

template<>
struct Factorial<0> {
    enum { value = 1 };
};
```

尽管C++98为TMP提供了基础，但真正的强大工具和特性是在C++11中引入的，这使得TMP变得更加易于使用和理解。为了深入了解TMP和其在C++11中的增强，我将在后续内容中进行详细讲解。

10. 异常处理

像其他编程语言一样，C++也提供了用于在程序中处理错误的语句，这让我们的代码编写风格多了一种自由度。在代码编写过程中提前预判异常产生的可能不仅能大大提高我们程序的可靠性，还能极大的增加开发效率。

```
try {
    int x = 10, y = 0;
    if(y == 0) throw "Division by zero!";
    int z = x / y;
} catch (const char* msg) {
    std::cerr << msg << std::endl;
}
```

11. 命名空间

命名空间用于组织代码并避免名称冲突，用法也很简单，您可以在下面的例子中获得直观理解。

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

int main() {
    int result = Mathematics::add(3, 4);
    std::cout << result << std::endl;  // 输出 7
}
```

12. 标准模板库 (STL)

STL提供了一系列的数据结构和算法，就像`Python`的库生态一样，C++一样具备可靠的生态和工具库，甚至大部分内置的标准库即可解决任何常规问题，还为用户自行构建第三方库提供了可靠的基础。

容器：

如`vector`, `list`, `map`, `set`等，我们将在现代C++标准中进行详细的介绍，因为新标准中为其添加了更多便捷易用的语法糖，现在讲解并不合时宜，我仅仅做一些简单的抛砖引玉。

(1) 初始化

`std::vector`可以通过多种方式进行初始化。

```
#include <vector>

std::vector<int> vec1;  // 默认初始化，空的vector
std::vector<int> vec2(5);  // 包含5个元素，每个元素的值都是0
std::vector<int> vec3(5, 10);  // 包含5个元素，每个元素的值都是10
std::vector<int> vec4 = {1, 2, 3, 4, 5};  // 列表初始化
```

(2) 插入元素

您可以在`vector`的末尾添加元素，或在任何其他位置插入元素。

```
vec1.push_back(6);  // 在末尾添加一个元素
vec1.insert(vec1.begin(), 0);  // 在开始位置插入一个元素
```

(3) 访问元素

可以使用下标操作符或`at()`方法访问`vector`中的元素。

```
int first = vec4[0];  // 使用下标操作符
int second = vec4.at(1);  // 使用at()方法
```

(4) 删除元素

```
vec4.pop_back();  // 删除最后一个元素
vec4.erase(vec4.begin());  // 删除第一个元素
vec4.erase(vec4.begin(), vec4.begin() + 2);  // 删除前两个元素
```

(5) 大小和容量

```
int size = vec4.size();  // 获取元素数量
int capacity = vec4.capacity();  // 获取当前分配的内存能容纳的元素数量
bool isEmpty = vec4.empty();  // 检查vector是否为空
```

(6) 其他常用操作

```
vec4.shrink_to_fit();  // 减少容器的容量，使其与大小相匹配

vec4.front();  // 获取第一个元素
vec4.back();   // 获取最后一个元素

vec4.assign(5, 20);  // 将vector的内容替换为5个值为20的元素
vec4.swap(vec1);  // 交换vec4和vec1的内容
```

(7) 查找和算法

与`std::vector`一起，STL还提供了一系列算法，这些算法可以与任何容器一起使用，包括`std::vector`。

```
#include <algorithm>

std::vector<int>::iterator it;
it = std::find(vec4.begin(), vec4.end(), 3);  //
查找值为3的元素
if(it != vec4.end()) {
    std::cout << "Element found: " << *it <<
std::endl;
} else {
    std::cout << "Element not found." <<
std::endl;
}

std::reverse(vec4.begin(), vec4.end());  // 反转
vector
std::sort(vec4.begin(), vec4.end());  // 对vector
进行排序
```

这些只是`std::vector`的一些基本操作。实际上，`std::vector`提供了许多其他功能，可以满足各种复杂的需求，我们后面会有更详细的讲解。

迭代器：

其本质就是指向容器首尾的指针，我们可以通过操作指针来遍历容器，C++98的迭代器 并未发挥其全部价值，后续的内容中将体现它的强大之处。

```
std::vector<int>::iterator it;
for(it = numbers.begin(); it != numbers.end();
++it) {
    std::cout << *it << std::endl;
}
```

算法：

如`sort`，`find`，这是两个极为常用的函数。

```
std::sort(numbers.begin(), numbers.end());  // 排
序
it = std::find(numbers.begin(), numbers.end(), 3);
 // 查找
```

13. 动态内存管理

new 和 delete：

用于在堆上分配和释放内存。

```
int* arr = new int[5];  // 分配一个整数数组
delete[] arr;  // 释放数组
```

14. 指针和引用

指针：

指针用于存储变量的地址，由于它作为C语言的基础和重点，相信您已经完全掌握了指针的相关知识，我们在现代C++中也基本不会进行太过底层的操作，所以这里不做重点赘述。

```
int x = 10;
int* ptr = &x;  // 获取x的地址
std::cout << *ptr << std::endl;  // 输出x的值
```

引用：

引用是另一个变量的别名，仅此而已，不建议过度思考其实现。

```
int y = 20;
int& ref = y;
ref = 30;  // y的值现在是30
```

15. 输入/输出 (I/O)

控制台I/O：

`cin`, `cout`, `cerr`, `clog`。

```
int num;
std::cout << "Enter a number: ";
std::cin >> num;
std::cerr << "This is an error message." <<
std::endl;
```

文件I/O：

- `ifstream`，`ofstream`。

```
std::ofstream outFile("example.txt");
outFile << "Writing to a file." << std::endl;
outFile.close();

std::ifstream inFile("example.txt");
std::string line;
while (getline(inFile, line)) {
    std::cout << line << std::endl;
}
inFile.close();
```

16. 预处理器指令

包含头文件：

- `#include`。

```
#include <iostream>
```

宏定义：

- `#define`。

```
#define PI 3.14159
```

条件编译：

- `#ifdef`, `#ifndef`, `#else`, `#endif`。

```
#ifdef DEBUG
    std::cout << "Debug mode is ON." << std::endl;
#else
    std::cout << "Debug mode is OFF." <<
std::endl;
#endif
```

C++11 里程碑式的改变

C++11，被誉为现代C++的开端，它不仅仅是一次简单的更新，更是一次对C++进行彻底现代化的尝试。在这个版本中，C++得到了许多新特性，这些特性旨在使编程更加简洁、高效和直观。或许从前我们经常认为C++是C语言的一个超集，但从C++11开始这个观念彻底被颠覆了。

1. 自动类型推导

在传统的C++编程中，我们总是需要明确地指定变量的类型。但随着模板和泛型编程的复杂性增加，有时确定变量的确切类型可能会变得非常复杂。这就是`auto`关键字的用武之地。

`auto`关键字告诉编译器："请你为我推导出这个变量的类型"。这听起来可能有点懒，但实际上，它可以使代码更加简洁，尤其是在处理复杂的模板类型时。

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
```

在传统的C++中，如果我们想要迭代这个向量，我们可能会这样写：

```
for(std::vector<int>::iterator it =
numbers.begin(); it != numbers.end(); ++it) {
    // ...
}
```

但在C++11中，我们可以简化为：

```
for(auto it = numbers.begin(); it !=
numbers.end(); ++it) {
    // ...
}
```

这样的代码不仅更简洁，而且更具可读性。此外，如果`numbers`的类型发生变化，我们不必修改迭代器的类型，因为`auto`会为我们处理这一切。

2. 范围for循环

说到迭代，C++11为我们提供了一种更加简洁的方法来遍历容器：范围for循环。这是一种新的循环结构，允许您直接遍历容器的元素，而不需要使用迭代器。

想象一下，您有一个整数向量，并希望打印出其中的每个元素。在C++11之前，您可能需要使用迭代器，但现在，您可以简单地这样做：

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
for(int num : numbers) {
    std::cout << num << std::endl;
}
```

这种新的循环结构使代码更加简洁和直观。您不再需要手动管理迭代器，也不需要担心超出容器的范围。范围for循环为您处理了所有这些细节。

此外，这种新的循环结构不仅限于标准容器。任何具有`begin()`和`end()`成员函数的对象都可以使用范围for循环进行迭代。

3. 初始化列表

C++11引入了一种新的初始化语法，使得初始化变得更加统一和直观。无论您正在初始化的是一个数组、一个结构体、一个类还是一个简单的变量，您都可以使用相同的语法。

```
int arr[] = {1, 2, 3, 4, 5};
std::vector<int> vec = {1, 2, 3, 4, 5};
```

这种新的初始化语法不仅使代码更加简洁，而且更具可读性。它消除了初始化过程中的歧义，并为复杂的数据结构提供了一种简单的初始化方法。

此外，C++11还为类提供了对初始化列表的支持。这意味着您可以在类的构造函数中使用初始化列表，以简洁的方式初始化类的成员。

```
class MyClass {
public:
    MyClass(std::initializer_list<int> list) {
        // ...
    }
};

MyClass obj = {1, 2, 3, 4, 5};
```

这种新的初始化方法为C++编程带来了巨大的便利性，使得代码更加简洁和直观。

4. Lambda表达式

Lambda表达式是C++11中引入的一个强大特性，它允许您定义一个匿名函数并直接在代码中使用。Lambda表达式可以捕获变量、接受参数并返回值，就像常规函数一样。

考虑以下示例，我们有一个整数向量，并希望对其进行排序。在C++11中，您可以使用Lambda表达式直接定义排序准则：

```
std::vector<int> numbers = {5, 2, 8, 1, 3};
std::sort(numbers.begin(), numbers.end(), [](int
a, int b) -> bool {
    return a > b;
});
```

在上述代码中，`[](int a, int b) -> bool { return a > b; }`就是一个Lambda表达式。它接受两个整数参数，并返回一个布尔值。这个Lambda表达式定义了一个降序排序准则。

Lambda表达式的语法可能看起来有点复杂，但一旦您习惯了，它会变得非常直观。它大大简化了短小功能的编写，使您的代码更加简洁。

5. 智能指针

在传统的C++编程中，动态内存管理是一个常见的问题。使用`new`和`delete`手动管理内存可能会导致内存泄漏或无效的内存访问。C++11引入了智能指针，以自动化内存管理。

C++11提供了三种类型的智能指针：`std::unique_ptr`、`std::shared_ptr`和`std::weak_ptr`。

- **std::unique_ptr**: 这是一个独占所有权的智能指针。当`unique_ptr`超出范围时，它所指向的对象会被自动删除。

```
std::unique_ptr<int> ptr(new int(5));
```

- **std::shared_ptr**: 允许多个`shared_ptr`共享同一个对象的所有权。当最后一个`shared_ptr`超出范围时，它所指向的对象会被自动删除。

```
std::shared_ptr<int> ptr1(new int(10));
std::shared_ptr<int> ptr2 = ptr1;  // 两个指针共享同一个对象
```

- **std::weak_ptr**: 是`shared_ptr`的伙伴，它不会增加引用计数。它用于防止智能指针之间的循环引用。

智能指针大大简化了C++的内存管理，使得内存泄漏和无效的内存访问变得更加罕见。

6. 右值引用和移动语义

C++11引入了右值引用，这是一种新的引用类型，用于引用即将被销毁的对象。这为C++带来了移动语义，允许对象在不进行昂贵的深拷贝的情况下进行转移。

考虑以下示例，我们有一个大型向量，并希望将其传递给另一个向量：

```
std::vector<int> vec1 = {1, 2, 3, 4, 5};
std::vector<int> vec2 = std::move(vec1);
```

在上述代码中，`std::move`将`vec1`转换为右值，这使得`vec2`可以直接接管`vec1`的内部数据，而无需进行深拷贝。这大大提高了性能，尤其是对于大型对象。

但要完全理解移动语义，我们需要进一步探讨右值引用的概念。

右值和左值：

在C++中，表达式可以是左值或右值。左值是一个对象的持久位置，通常可以在多个表达式中使用。右值是临时的，它的内容可以被移动到另一个对象中。

右值引用：

右值引用使用`&&`符号定义，并且它只能绑定到一个右值。这为开发者提供了一个明确的标志，表明对象的内容可以被安全地移动。

```
int&& rvalue_ref = 10 + 20;  // 10 + 20是一个右值
```

移动语义：

在传统的C++编程中，对象的复制通常涉及深拷贝，这可能非常昂贵，尤其是对于大型对象。移动语义允许资源从一个对象转移到另一个对象，而不是复制它们。这大大提高了性能。

例如，当使用`std::move`函数时，它将其参数转换为右值，这通常意味着资源可以从源对象移动到目标对象。

移动构造函数和移动赋值运算符：

为了支持移动语义，许多C++类（尤其是STL容器）都定义了移动构造函数和移动赋值运算符。

```
class MyClass {
public:
    MyClass(MyClass&& other) noexcept { /*...*/ }
 // 移动构造函数
    MyClass& operator=(MyClass&& other) noexcept {
/*...*/ }  // 移动赋值运算符
};
```

这些函数允许类的实例在不进行深拷贝的情况下进行转移，从而提高性能。

总之，右值引用和移动语义为C++编程带来了巨大的性能提升，特别是在处理大型对象和资源密集型操作时。这些特性使得C++代码更加高效，同时也更加简洁和直观。

7. `nullptr`和强类型枚举

nullptr：

在传统的C++中，我们使用`NULL`表示空指针。但`NULL`实际上只是一个整数值0，这可能会导致类型混淆。C++11引入了`nullptr`，这是一个特殊的类型，专门用于表示空指针。

```
int* ptr = nullptr;
```

强类型枚举：

C++11引入了新的枚举类型，称为强类型枚举。这些枚举是强类型的，这意味着它们的值不会隐式转换为其他类型。

```
enum class Color {
    Red,
    Green,
    Blue
};

Color myColor = Color::Red;
```

强类型枚举提供了更好的类型安全性，并防止了不必要的类型转换。

8. 继承构造

派生类可以继承基类的构造函数。

```
class Base {
public:
    Base(int) {}
};

class Derived : public Base {
    using Base::Base;
};
```

9. 委托构造函数

C++11允许一个构造函数调用同一个类中的另一个构造函数。这称为委托构造。

```
class Box {
public:
    Box() : Box(1, 1, 1) {}  // 委托构造函数
    Box(double l, double w, double h) : length(l),
width(w), height(h) {}
private:
    double length, width, height;
};
```

通过委托构造函数，您可以避免重复代码，并确保所有构造函数逻辑都位于一个地方。

10. 默认和删除的函数

C++11允许您明确地指定函数为默认实现或删除。

```
class MyClass {
public:
    MyClass() = default;  // 使用编译器的默认实现
    MyClass(const MyClass&) = delete;  // 禁止复制
构造
};
```

这为类提供了更好的控制，确保了某些操作（如复制或赋值）不会被误用。

11. 模板元编程（TMP）增强

模板元编程（TMP）是一种在编译时执行计算的技术，它使用模板作为其主要工具。TMP可以用于生成编译时的常量、类型计算和编译时的条件分支。TMP在C++98和C++03中就已经存在，但随着C++11和后续标准的引入，TMP得到了进一步的增强和简化。C++11引入了许多新的TMP工具，如`decltype`、`constexpr`、`std::integral_constant`和`std::enable_if`等。这些工具使得TMP变得更加强大和易于使用。

基本概念：

- **类型计算**：在TMP中，我们经常使用类型来表示值。例如，`std::integral_constant`是一个模板，它可以用于表示编译时的整数值。

```
typedef std::integral_constant<int, 3> three;
```

- **编译时递归**：TMP中的递归是通过模板特化和模板参数来实现的。

条件编译`std::enable_if`：

`std::enable_if`是TMP中的一个重要工具，它可以用于条件编译。它根据一个布尔模板参数来启用或禁用某个模板的特定实例。

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };
```

使用`std::enable_if`可以实现函数模板的条件重载。

TMP的应用：

TMP的一个常见应用是在编译时生成查找表或执行复杂的类型计算。例如，`std::tuple`和`std::variant`的实现都依赖于TMP。

12. `constexpr`常量表达式

`constexpr`是一个新的关键字，用于指定常量表达式。这些表达式在编译时计算，而不是运行时。

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

int arr[square(4)];  // 创建一个大小为16的数组
```

使用`constexpr`可以确保某些值在编译时是已知的，这有助于优化和错误检查。

13. `noexcept`无异常抛出规定

`noexcept`是一个新的关键字，用于指定函数不会抛出异常。

```
void myFunction() noexcept {
    // 这个函数保证不会抛出异常
}
```

这为异常处理提供了更好的控制，并可以优化不抛出异常的函数。

14. 继续增强的STL

C++11对STL进行了许多增强。例如，**`std::array`**是一个固定大小的数组，**`std::forward_list`**是一个单向链表。

数组`std::array`：

`std::array`是一个固定大小的数组，它提供了与传统数组相似的性能，但增加了一些有用的成员函数，如**`size()`**和**`fill()`**。

```
#include <array>

std::array<int, 5> arr = {1, 2, 3, 4, 5};
arr.fill(10);  // 将所有元素设置为10
int first = arr.front();  // 获取第一个元素
int last = arr.back();  // 获取最后一个元素
```

单链表`std::forward_list`：

`std::forward_list`是一个单向链表，它比**`std::list`**更加轻量，但不支持双向迭代。

```
#include <forward_list>

std::forward_list<int> flist = {1, 2, 3, 4};
flist.push_front(0);  // 在链表前面插入一个元素
flist.pop_front();  // 删除链表的第一个元素
```

哈希表`std::unordered_map`和`std::unordered_set`:

`std::unordered_map`和`std::unordered_set`是基于哈希表的容器，通常提供比其有序对应项更快的查找时间。

```
#include <unordered_map>
#include <unordered_set>

std::unordered_map<std::string, int> umap =
{{"apple", 1}, {"banana", 2}};
umap["cherry"] = 3;  // 插入或更新键值对

std::unordered_set<int> uset = {1, 2, 3, 4};
uset.insert(5);  // 插入元素
```

元组`std::tuple`:

`std::tuple`允许您将多个不同类型的元素组合成一个单一对象。

```
#include <tuple>

std::tuple<int, std::string, double> myTuple =
std::make_tuple(1, "apple", 3.14);
int a;
std::string b;
double c;
std::tie(a, b, c) = myTuple;  // 解包元组
```

(1) 访问元素

使用`std::get`来访问tuple的元素。你需要提供元素的索引作为模板参数。

```
auto firstElement = std::get<0>(t);  // 获取第一个元
素
```

(2) 获取tuple的大小

使用`std::tuple_size`。

```
constexpr auto size =
std::tuple_size<decltype(t)>::value;
```

(3) 遍历tuple的元素

由于tuple可以包含不同类型的元素，所以不能使用传统的循环来遍历它。但是，可以使用递归模板函数来实现这一点，这里看不懂没关系，下面会介绍变长模板参数，请结合来分析。

```
template<std::size_t Index = 0, typename...
TupleTypes>
typename std::enable_if<Index == sizeof...
(TupleTypes), void>::type
printTuple(const std::tuple<TupleTypes...>& tuple)
{
    // 结束递归
}

template<std::size_t Index = 0, typename...
TupleTypes>
typename std::enable_if<Index < sizeof...
(TupleTypes), void>::type
printTuple(const std::tuple<TupleTypes...>& tuple)
{
    std::cout << std::get<Index>(tuple) <<
std::endl;
    printTuple<Index + 1>(tuple);  // 递归调用
}
```

在`**storeInTuple**`函数内部，您可以调用`**printTuple(t)**`来打印tuple的内容。

(4) 比较tuples

可以使用关系运算符（如`==`，`!=`，`<`，`<=`，`>`，`>=`）来比较两个tuples，前提是它们的元素类型都支持这些运算符。

(5) 连接两个tuples

使用`std::tuple_cat`来连接两个或多个tuples。

```
auto t2 = std::make_tuple(4, 5);
auto t3 = std::tuple_cat(t, t2);  // t3现在包含1,
'a', 3.14, 4, 5
```

(6) 应用函数到tuple (C++17特性)

这里为了避免强行割裂tuple的完整使用案例，决定提前介绍C++17特性，使用`std::apply`来将函数应用于tuple的元素。

```
auto sum = std::apply([](auto... args) { return
(... + args); }, t);
```

时间库`std::chrono`:

`std::chrono`是一个时间库，允许您以各种方式测量和表示时间。

```
#include <chrono>
#include <thread>

auto start =
std::chrono::high_resolution_clock::now();
std::this_thread::sleep_for(std::chrono::seconds(1
));
auto end =
std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed = end -
start;
```

15. 变长模板参数

变长模板参数是C++11的一个强大特性，它允许模板接受任意数量的参数，这为模板编程带来了巨大的灵活性。以下是关于变长模板参数的详细介绍：

基本定义：

变长模板参数使用`...`来定义，并可以接受任意数量的参数。

```
template<typename... Args>
void function(Args... args) {
    // 使用args...
}
```

使用`sizeof...`获取参数数量：

`sizeof...`运算符可以用来获取变长模板参数的数量。

```
template<typename... Args>
void function(Args... args) {
    std::cout << sizeof...(args) << std::endl;  // 打印参数数量
}
```

使用`std::tuple`与变长模板参数：

`std::tuple`是一个非常有用的模板类，它可以与变长模板参数一起使用，以存储任意数量和类型的值。

```
template<typename... Args>
void storeInTuple(Args... args) {
    std::tuple<Args...> t(args...);
    // 使用t...
}
```

递归展开：

由于变长模板参数可以接受任意数量的参数，因此常常需要递归来处理它们。以下是一个简单的例子，展示如何递归地打印所有参数：

```
template<typename T>
void print(T t) {
    std::cout << t << std::endl;
}

template<typename T, typename... Args>
void print(T t, Args... args) {
    std::cout << t << std::endl;
    print(args...);
}
```

在上述代码中，当`print`函数被调用时，它会首先打印第一个参数，然后递归地调用自己来打印剩余的参数。

这里我们要摒弃严格的文章结构，避免强行割裂本该聚合的内容。我将使用一种在C++17中引入的编译时条件判断来避免上述需要单独定义递归函数出口的麻烦。

```
template<typename... Args>
void print(Args... args) {
    if constexpr (sizeof...(args) > 0) {
        std::cout << std::get<0>
(std::tuple<Args...>(args...)) << std::endl;
        print(std::get<1>(std::tuple<Args...>
(args...))...);
    }
}
```

折叠表达式 (C++17特性)：

折叠表达式是C++17中引入的，用于简化变长模板参数的处理，它允许对参数进行折叠操作，如加法、乘法等。

```
template<typename... Args>
auto sum(Args... args) {
    return (... + args);  // C++17折叠表达式
}
```

在上述代码中，`sum`函数使用折叠表达式来计算所有参数的和。

总的来说，变长模板参数为模板编程提供了巨大的灵活性和强大的功能。通过递归、折叠表达式和其他技巧，可以高效地处理和操作这些参数。

16. 异步和多线程

线程库`std::thread`：

`std::thread`是C++11中引入的一个类，用于表示一个可执行的线程。它允许您并发地运行函数或可调用对象。

```
#include <thread>
#include <iostream>

void functionToRunInThread() {
    std::cout << "Running in a separate thread!"
<< std::endl;
}

int main() {
    std::thread t(functionToRunInThread);
    t.join();  // 等待线程完成
}
```

互斥量和锁管理器：

为了避免多个线程同时访问共享资源造成的数据竞争，我们需要使用互斥量（mutex）。**`std::mutex`**提供了一个互斥量，而**`std::lock_guard`**是一个便捷的RAII风格的锁管理器。

```
#include <thread>
#include <iostream>
#include <mutex>

std::mutex mtx;
```

```
void printWithMutex(int id) {
    std::lock_guard<std::mutex> lock(mtx);
    std::cout << "Thread " << id << " is running."
<< std::endl;
}

int main() {
    std::thread t1(printWithMutex, 1);
    std::thread t2(printWithMutex, 2);
    t1.join();
    t2.join();
}
```

异步`std::future`和`std::async`：

`**std::future**`表示一个异步操作的结果，它可以在未来某个时间点获得。`**std::async**`是一个函数，它可以异步地运行一个函数，并返回一个`**std::future**`对象。

```
#include <future>
#include <iostream>

int compute() {
    return 42;
}

int main() {
    std::future<int> result = std::async(compute);
    std::cout << "Result: " << result.get() << 
std::endl;  // 阻塞直到结果可用
}
```

条件变量`std::condition_variable`:

条件变量允许线程等待某个条件成立。它通常与`std::mutex`一起使用。

```
#include <thread>
#include <iostream>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
```

```
bool ready = false;

void printWhenReady() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, []{ return ready; });  // 等待直到ready为true
    std::cout << "Ready!" << std::endl;
}

int main() {
    std::thread t(printWhenReady);
    {
        std::lock_guard<std::mutex> lock(mtx);
        ready = true;
    }
    cv.notify_one();  // 通知等待的线程
    t.join();
}
```

原子操作：

原子操作是一种在多线程环境中不需要额外同步机制就能确保数据安全的操作。`std::atomic`提供了一种表示原子类型的方法。

```
#include <atomic>
```

```
#include <thread>
#include <iostream>

std::atomic<int> counter(0);

void increment() {
    for (int i = 0; i < 100000; ++i) {
        ++counter;
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << "Counter: " << counter.load() <<
std::endl;
}
```

这些只是C++11中异步和多线程支持的一部分。这个标准为并发编程提供了强大的工具和抽象，使得在C++中进行多线程编程变得更加简单和安全。

17. 类型推导

`decltype`关键字用于推导表达式的类型。

```
int x = 10;
decltype(x) y = 20;  // y的类型是int
```

18. 用户定义的字面量

C++11 允许用户定义自己的字面量。这使得您可以为基本数据类型或自定义类型创建自己的表示方法。

```
constexpr long double operator"" _cm(long double
x) { return x * 10; }
constexpr long double operator"" _m(long double x)
{ return x * 1000; }
constexpr long double operator"" _mm(long double
x) { return x; }

double height = 3.4_cm;  // height = 34.0
```

这里，我们定义了三个自定义字面量，用于表示厘米、米和毫米。

19. 对齐规定

`alignas`和`alignof`关键字用于指定和查询变量的对齐要求。

```
alignas(16) int array[4];
```

20. 正则表达式

C++11 提供了对正则表达式的支持，使得文本处理和验证变得更加容易。

```
#include <regex>

std::string s = "C++11 is awesome!";
std::regex e ("(\\w+)\\s(\\d+)");  // 匹配单词和数字
std::smatch m;
std::regex_search(s, m, e);
for(auto x : m) std::cout << x << " ";
```

这将输出 `C++11 11`，因为它匹配了单词 "C++11" 和数字 "11"。

C++17 正式步入现代化编程

C++17，作为现代C++的进一步发展，继续完善了C++的特性，使其更加强大和灵活。这个版本中，C++得到了许多新的增强和改进，这些特性进一步简化了编程，使得代码更加简洁、高效和直观。

1. 结构化绑定

C++17引入了结构化绑定，这允许您从数组、结构体或元组中解包多个值，并将它们绑定到变量上。

```
std::pair<int, std::string> p = {1, "apple"};
auto [id, name] = p;  // id = 1, name = "apple"
```

这种新的绑定方式使得处理复合数据结构变得更加简洁和直观。

2. `if`和`switch`的初始化器

C++17为`if`和`switch`语句引入了一个新的初始化器语法，允许您在条件检查之前执行一个初始化操作。

```
if (int value = compute(); value > 10) {
    // ...
}
```

这种新的语法使得代码更加紧凑，同时确保了初始化和条件检查在同一个逻辑块中。

3. 内联变量

C++17引入了内联变量的概念，这允许您在头文件中定义变量，而不会导致多重定义错误。

```
inline int globalValue = 42;
```

这为库开发者提供了更大的灵活性，使得变量可以像内联函数一样在头文件中定义。

4. `constexpr`的增强

C++17进一步增强了`constexpr`，使得更多的函数和算法可以在编译时执行。

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

这种增强为编译时计算提供了更大的灵活性，使得代码更加高效。

5. 折叠表达式

如前文所述，C++17引入了折叠表达式，这是一个强大的特性，允许您简洁地对变长模板参数进行操作。

```
template<typename... Args>
auto sum(Args... args) {
    return (... + args);
}
```

这种新的表达式大大简化了模板编程，使得处理变长参数变得更加简洁。

6. `std::optional`可选值

C++17引入了**`std::optional`**，这是一个模板类，表示一个可选的值。它可以持有一个值，或者不持有任何值。

```
std::optional<int> getResult() {
    if (/* some condition */) {
        return 42;
    }
    return std::nullopt;
}
```

这为表示可能不存在的值提供了一种类型安全的方式。

7. `std::variant`

C++17还引入了**`std::variant`**，这是一个模板类，可以持有多种不同类型的值，但在任何时候只能持有其中一种。

```
std::variant<int, std::string> v;
v = 42;
v = "hello";
```

这为表示多种可能的值类型提供了一种灵活的方式。

8. `std::any`

与`std::variant`相反，`std::any`可以持有任何类型的值，但在任何时候只能持有一个值。

```
std::any a = 42;
a = std::string("hello");
```

这为存储任意类型的值提供了一种灵活的方式。

9. `std::string_view`

C++17引入了`std::string_view`，这是一个非修改的字符串引用，它提供了一种高效的方式来查看字符串的子串，而不需要创建新的字符串。

```
std::string s = "hello world";
std::string_view sv = s.substr(0, 5);  // "hello"
```

这为处理字符串提供了一种更加高效和灵活的方式。

10. 文件系统库

C++17引入了一个新的文件系统库，提供了一种跨平台的方式来操作文件和目录。

```
#include <filesystem>
namespace fs = std::filesystem;

fs::path p = "/path/to/file";
if (fs::exists(p)) {
    // ...
}
```

这为文件操作提供了一种现代和类型安全的方式。

11. 并行算法

C++17的并行算法是STL中的一个重要扩展，它为开发者提供了一个简单的方式来利用多核处理器的能力，而无需直接管理线程或其他低级并发结构。这使得编写高效且并行的代码变得更加简单。然而，您仍然需要注意数据竞争和性能问题，确保正确且高效地使用这些算法。

执行策略：

C++17引入了三种执行策略，它们决定了算法如何执行：

- `std::execution::seq`：顺序执行。这与传统的STL算法的行为相同。
- `std::execution::par`：并行执行。算法将尽可能地并行执行。
- `std::execution::par_unseq`：并行和向量化执行。除了并行执行外，还可能使用SIMD指令进行向量化。

并行算法示例：

以下是一些并行算法的示例：

(1) 排序

```
std::vector<int> v = {5, 3, 4, 1, 2};
std::sort(std::execution::par, v.begin(),
v.end());
```

(2) 查找

```
auto result = std::find(std::execution::par,
v.begin(), v.end(), 3);
```

(5) 累加

```
int sum = std::accumulate(std::execution::par,
v.begin(), v.end(), 0);
```

(6) 变换:

```
std::vector<int> output(v.size());
std::transform(std::execution::par, v.begin(),
v.end(), output.begin(), [](int n) { return n * 2;
});
```

注意事项：

- **数据竞争**：当使用并行算法时，必须确保没有数据竞争。例如，如果一个函数修改了一个全局变量，那么在并行版本中可能会出现问题。
- **性能**：并不是所有的算法都适合并行化。小的数据集或简单的操作可能不会从并行化中受益，因为线程管理的开销可能会超过并行执行的好处。
- **兼容性**：并行算法需要一个支持并行执行的编译器和标准库。此外，并不是所有的STL算法都有并行版本。

C++20 新特性

C++20是现代C++编程又一个重大的版本更新，它引入了许多新特性和改进，其向着平稳的学习曲线进行了优化，也标志着C++20正式走向未来。

1. 概念 (Concepts)

概念是一种约束模板参数的方式，它可以确保模板参数满足某些特定的属性或行为。

```
template<typename T>
concept Integral = std::is_integral_v<T>;

template<Integral T>
T square(T value) {
    return value * value;
}
```

在上述代码中，我们定义了一个`Integral`概念，它确保传递给`square`函数的参数是整数类型。

2. 范围 (Ranges)

范围提供了一种新的方式来操作容器和其他序列数据。它们可以与管道操作符一起使用，使代码更加直观。

```
#include <ranges>

std::vector<int> numbers = {1, 2, 3, 4, 5};
auto evenNumbers = numbers | std::views::filter([]
(int n) { return n % 2 == 0; });
for (int n : evenNumbers) {
    std::cout << n << " ";
}
```

上述代码将输出所有的偶数：`2 4`。

3. 协程 (Coroutines)

协程是一种特殊类型的函数，它可以在执行过程中被暂停，并在后续的某个时间点从暂停的地方恢复执行。这种能力使得协程成为异步编程、生成器、以及其他需要非线性控制流的场景的理想选择。

基本概念：

(1) 协程 vs 线程

与线程不同，协程是协作式的，意味着在任何给定时间只有一个协程在执行，而线程是抢占式的，可以被操作系统在任何时候中断。协程的切换是由协程自己控制的，而线程的切换是由操作系统控制的。

(2) `co_yield`

这是一个关键字，用于在协程中产生一个值并暂停协程的执行。当协程再次被恢复时，它会从`co_yield`之后的代码继续执行。

(3) `co_return`

用于从协程返回一个值或表示协程结束。

(4) `co_await`

用于暂停当前协程并等待某个异步操作完成。

协程的优势：

- **简化异步代码**：与传统的回调或`std::future`相比，协程提供了一种更加直观和简洁的方式来编写异步代码。
- **高效的性能**：协程的切换开销远小于线程的切换，使得它们非常适合于高并发的场景。
- **灵活的控制流**：协程可以轻松地实现生成器、迭代器、以及其他复杂的控制流结构。

使用示例：

```
#include <iostream>
#include <experimental/coroutine>
#include <future>

// 使用协程实现异步任务
std::future<int> asyncTask() {
    co_await std::async(std::launch::async, []() {

 std::this_thread::sleep_for(std::chrono::seconds(
1));
    });
    co_return 42;
}

// 使用协程实现生成器
auto range(int start, int end) ->
std::generator<int> {
    for (int i = start; i <= end; ++i) {
        co_yield i;
    }
}

int main() {
    // 异步任务示例
    auto result = asyncTask();
```

```
    std::cout << "Async result: " << result.get()
<< std::endl;

    // 生成器示例
    for (int n : range(1, 5)) {
        std::cout << n << " ";
    }
}
```

在上述代码中，我们首先使用协程实现了一个异步任务，该任务在1秒后返回值`42`。接着，我们使用协程实现了一个生成器，该生成器生成一个指定范围内的整数序列。

协程为C++编程带来了革命性的改变，它提供了一种全新的方式来处理异步操作、生成器和其他复杂的控制流结构。随着更多的编译器和库的支持，协程的应用将变得越来越广泛。

4. 模块 (Modules)

模块是一种新的代码组织方式，它可以提高编译速度并使代码结构更清晰。

```
// myModule.ixx
export module myModule;

export int myFunction() {
    return 42;
}

// main.cpp
import myModule;

int main() {
    std::cout << myFunction() << std::endl;  // 输出 42
}
```

5. 三元比较 (Three-way comparison)

新的`<=>`运算符可以同时进行等于、小于和大于的比较。

```
struct Point {
    int x, y;
    auto operator<=>(const Point& other) const {
        if (x < other.x) return
std::strong_ordering::less;
        if (x > other.x) return
std::strong_ordering::greater;
        return y <=> other.y;
    }
};

Point p1{1, 2}, p2{1, 3};
bool isLess = p1 < p2;  // true
```

总结

C++20还有很多内容值得大家去研究，在使用C++20的过程中我们也应该对现代化编程语言有所感悟。在面对C++多年来日益完善的生态时，我们也不难发现其高度灵活的社区诞生了一个个优秀的工具，但值得警醒的是，灵活的编程风格让数据结构和组件间的通信变得尤为困难，我们在使用多种处理不同任务的第三方库时，

其数据结构非常难以互相转化。这种现象也最终导向了`Python`的诞生，`Python`由C++构建并天生为了解决这个问题，它也确实做的不错，所以发展到今天，我强烈大家积极面向未来，在合适的任务中选择合适的编程语言进行开发，切忌固执己见耽误开发效率。