## 课程目标

1. C++简介
2. 简单的C++程序
3. 面向过程和面向对象
4. C++对C的扩展
5. 作用域运算符
6. 命名空间
7. struct/const 增强
8. bool关键字
9. 引用
10. 内联函数
11. 函数的默认参数
12. 函数重载
13. extern 'C'
14. Lambda 表达式

## 课程实验

- 通过引用修改原始变量中的值
- 函数的占位参数
- 函数默认参数
- 函数重载实验
- `c++`调用`c`语言代码
- 使用`Lambda`表达式实现模板库排序

## 课堂引入

- **面向过程和面向对象:**
  - **面向过程:**
    - 使用函数或过程作为程序的基本单元,程序由一系列函数调用组成,数据和函数是分离的。
    - 通常通过将功能模块化、函数化来实现代码复用。缺乏明确的封装、继承、多态概念,通常通过模块化的方式来组织代码。
  - **面向对象:**
    - 使用对象作为程序的基本单元,程序由一组对象的交互组成,对象封装了数据和相关的操作。
    - 通过类和对象的继承关系来实现代码的重用。侧重于封装、继承、多态的概念,提供更灵活的代码组织和设计模式。

  总体而言,面向过程和面向对象的发展史是紧密交织的,两者在不同的时期都为解决不同的编程问题提供了有效的方法。在当今软件开

  发中,这两种范式仍然共存,而且很多现代编程语言支持混合使用这两种思维方式。

- **C++对C的扩展:**

  C++对C的扩展主要体现在面向对象编程的引入,以及对语言功能的丰富和增强,使得开发者能够更方便地进行复杂和灵活的编程。然而,C++仍然保留了对C语言的兼容性,因此C语言的代码通常可以在C++中运行。

# 授课进程

## C++简介

### 一、C++的概述

世界上第一种计算机高级语言是诞生于1954年的FORTRAN语言。之后出现了多种计算机高级语言。1970年,AT&T的Bell实验室的D.Ritchie和K.Thompson共同发明了C语言。研制C语言的初衷是用它编写UNIX系统程序,因此它实际上是UNIX的"副产品"。它充分结合了汇编语言和高级语言的优点,高效而灵活,又容易移植。

与C语言一样,C++也是在贝尔实验室诞生的,Bjarne Stroustrup(本贾尼·斯特劳斯特卢普)在20世纪80年代在这里开发了这种语言。

20世纪70年代中期,Bjarne Stroustrup在剑桥大学计算机中心工作。他使用过Simula和ALGOL,接触过C,深知运行效率的意义。既要编程简单、正确可靠,又要运行高效、可移植,是Bjarne Stroustrup的初衷。以C为背景,以Simula思想为基础,正好符合他的设想。1979年,Bjarne Stroustrup到了Bell实验室,开始从事将C改良为带类的C(C with classes)的工作。1983年该语言被正式命名为C++。

C++从最初的C with class,经历了从C++98、C++03、C++11、C++ 14、C++17再到C++20,经过多次标准化改造,功能得到了极大的丰富,已经演变为一门集面向过程、面向对象、函数式、泛型和元编程等多种编程范式的复杂编程语言。

## 二、C++的优缺点

**优点:**

1、可扩展性强,有可移植性,跨平台性,面向对象等特性

2、高效,简洁,快速

3、强大而灵活的表达能力,和不输于C语言的效率

4、支持硬件开发

5、程序模块间的关系更为简单,程序模块的独立性、数据的安全性就有了良好的保障

6、通过继承与多态性,可以大大提高程序的可重用性,使得软件的开发和维护都更为方便

**缺点:**

1、比较底层,易用性不是很好

2、多继承和友元机制

3、标准库涵盖范围不足

4、开发周期长，难度大

**C语言和c++语言的关系:**

> C++语言是在C语言的基础上,添加了面向对象、模板等现代程序设计语言的特性而发展起来的。两者无论是从语法规则上,还是从运算符的数量和使用上,都非常相似,所以我们常常将这两门语言统称为"C/C++"。
>
> C语言和C++并不是对立的竞争关系:
>
> - C++是C语言的加强,是一种更好的C语言。
> - C++是以C语言为基础的,并且完全兼容C语言的特性。
>
> C语言和C++语言的学习是可以相互促进。学好C语言,可以为我们将来进一步地学习C++语言打好基础,而C++语言的学习,也会促进我们对于C语言的理解,从而更好地运用C语言。

## 三、C++的应用领域

1. 服务器端开发:很多游戏或者互联网公司的后台服务器程序都是基于C++开发的,而且大部分是linux操作系统,你如果想做这样的工作,需要熟悉linux操作系统及其在上面的开发,熟悉数据库开发,精通网络编程。
2. 游戏:目前很多游戏客户端都是基于C++开发的,这个领域需要学习的东西就比较多,比如计算机图形、多媒体处理。
3. 数字图像处理:比如像 `AutoCAD` 的系统开发,像 `OpenCV` 的视觉识别等等。
4. 科学计算:在科学计算领域,但是近年来, `C++` 凭借先进的数值计算库、泛型编程等优势在这一领域也应用颇多。
5. 网络软件: `C++` 拥有很多成熟的用于网络通信的库,其中最具有代表性的是跨平台的、重量级的 `ACE` 库,该库可以说是 `C++` 语言最重要的成果之一,在许多重要的企业、部门甚至是军方都有应用。比如 `GOOGLE` 的 `chrome` 浏览器,就是使用 `C++` 开发。

6. 操作系统:在该领域,C语言是主要使用的编程语言。但是`C++`凭借其对C的兼容性,面向对象性质也开始在该领域崭露头角。
7. 移动设备,嵌入式系统，教育，科研,工业控制领域。

**四、C++的发展**

C++的版本发展可以分为以下几个阶段:

1. C++1.0:C++的第一个版本,于1985年发布。这个版本基于C语言,具有类、继承、多态和其他功能,主要用于Unix系统的编程。
2. C++2.0:在C++1.0的基础上添加了一些新特性,如虚函数、const成员函数、引用和默认参数等,于1989年发布。
3. C++3.0:主要改进了C++2.0版本中的模板功能,并引入了标准模板库(STL),于1991年发布。
4. C++4.0:原计划于1994年发布,但最终被取消。
5. C++98:为C++建立了第一个国际标准,于1998年发布。这个版本引入了多个新特性,如命名空间、bool类型、异常处理和类型转换等。
6. C++11:也被称为C++0x,增加了很多新特性,如智能指针、范围for循环、异常处理等,于2011年发布。
7. C++14:扩展了C++11版本,主要增加了一些漏洞修复和小的改进。
8. C++17:基于C++11版本引入了许多新特性,于2017年发布。包括结构化绑定、折叠表达式、constexpr if、inline变量、类模板参数推导、新的字符串字面量、更便捷的并行编程支持(std::execution)等。此外,C++17还对已有特性进行了改进,如lambda表达式、随机数生成器、std::optional、std::variant 等。
9. C++20:最新的C++标准,于2020年发布。这个版本引入了很多新的特性,如模块、协程、概念等。

总的来说,C++的版本发展历经了多个阶段,每个版本都有一些新的特性和改进。最新的C++23标准引入了很多新的特性,使得开发者可以更方便地编写可移植、可互操作的代码,提高代码的可读性和可维护性。

**五、C++支持的编译器和IDE**

**5.1 支持C++的编译器和IDE有很多,以下是一些常见的选择:**

1. GCC(GNU Compiler Collection):这是一个开源的编译器,支持多种语言,包括C++。GCC在Linux系统上广泛使用,并可用于Windows 和 Mac OS X。
2. Clang:这是另一个开源的编译器,专注于C++语言。Clang使用LLVM作为其后端,支持现代C++特性和优化。
3. MSVC(Microsoft Visual C++):微软开发的编译器,主要用于Windows平台。MSVC支持C++标准,具有丰富的调试和性能分析工具。
4. MinGW:这是一个在Windows上运行的GCC版本,使得在Windows系统上编译C++代码更加容易。
5. Cygwin:一个在Windows上运行的类Unix环境,通过模拟Unix系统调用和库,使在Windows上运行C++程序更加接近于Unix系统上运行。

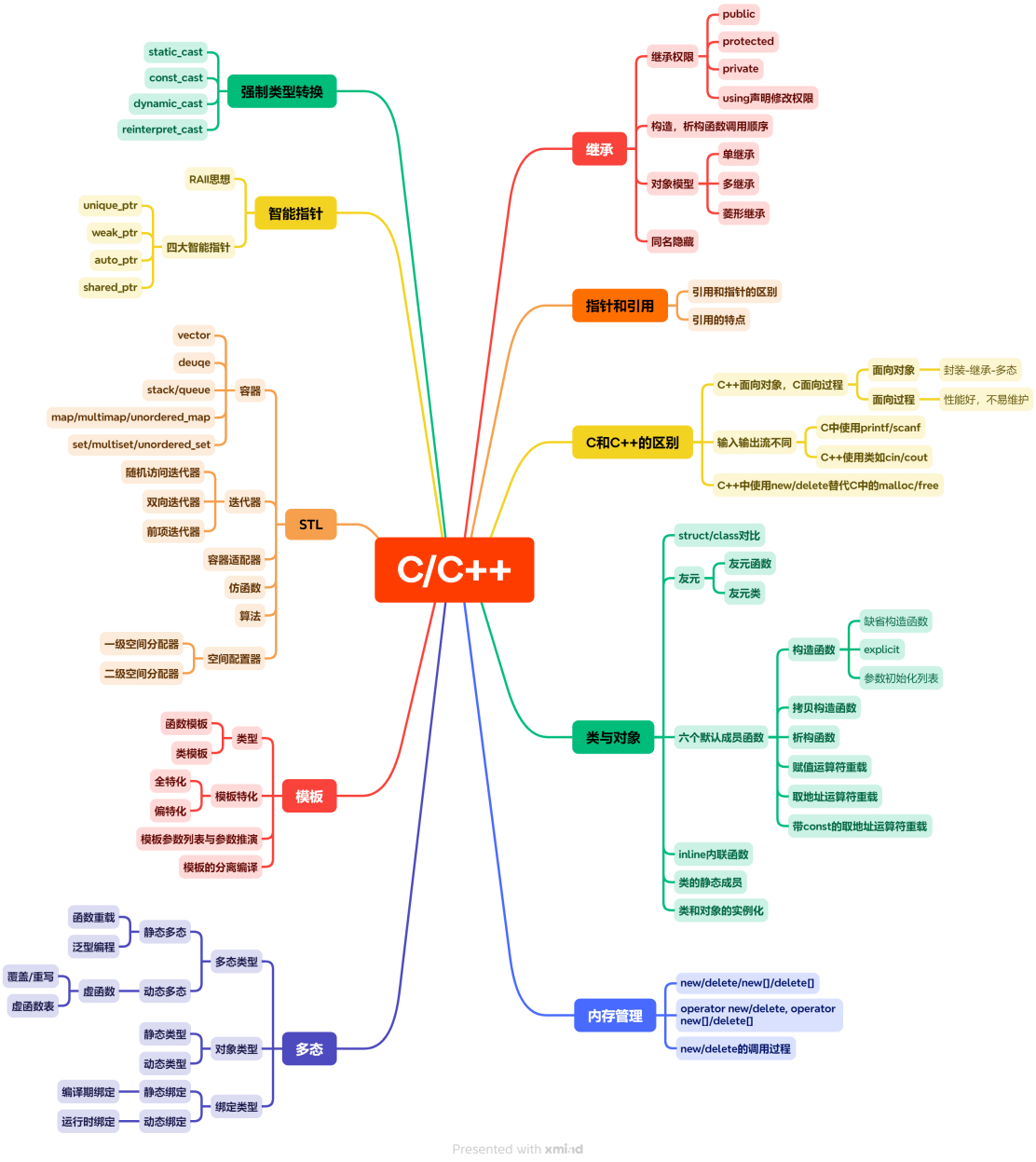
## 从C语言到C++

**需要扩展的语法内容**

1. 引用(reference):C++引入了引用的概念,可以通过&符号来声明引用。引用使变量具有别名,别名指向的地址与原变量地址相同,对引用赋值其实就是对原变量赋值,简化对变量的操作。
2. 函数重载:C++允许在同一类中使用相同函数名但参数不同的函数,这被称为函数重载。
3. 运算符重载:C++允许程序员定义新的运算符,这被称为运算符重载。
4. 内存管理:C++提供了新的关键字`new`和`delete`,用于替代C语言中的`malloc`和`free`函数,以支持对象的动态内存分配和释放。
5. 类型转换:C++提供了更安全的类型转换机制,包括显式类型转换运算符和类型转换构造函数。
6. 输入/输出流:C++提供了输入/输出流库,用于方便地进行输入输出操作。
7. 类和对象(class and object):C++是一种面向对象的语言,引入了类的概念,类是一种用户自定义的数据类型,它允许定义包含数据成员和成员函数的复合数据类型。类是对象的模板,它定义了对象的属性和方法。对象是类的实例,具有类定义的属性和行为。
8. 继承(inheritance):类可以继承其他类的属性和方法,这是代码重用的一种方式。
9. 多态(polymorphism):通过虚函数和纯虚函数实现,允许一个接口有多个实现方式。
10. 模板(template):C++引入了模板的概念,可以创建可处理不同数据类型的通用函数或类。
11. 异常处理(exception handling):C++提供了异常处理机制,允许在发生错误时抛出异常,并在上层代码中捕获处理。
12. 类型推导(type inference):C++11引入了类型推导的功能,可以根据表达式的值自动推断出变量的类型。
13. 智能指针(smart pointers):C++提供了智能指针的概念,它可以自动管理内存,避免内存泄漏。
14. Lambda表达式(lambda expressions):C++11引入了Lambda表达式,可以创建匿名函数,简化代码。
15. 标准模板库(STL):C++标准模板库是一套强大且通用的模板类库,包括如容器、迭代器、向量、列表、队列、栈等数据结构和常用算法。

这些扩展的语法内容使C++在编程范式和语言特性上与C语言有很大的不同,更加适用于面向对象的编程和复杂程序的开发。以上是从C语言到C++需要扩展的一些主要语法内容。当然,还有一些其他的特性和库可以使用,这取决于你的具体需求和学习方向。

## C++学习路线图




## C++学习推荐书籍

No.9 豆瓣热门编程图书TOP10

### C++ Primer 中文版（第 5 版）


作者: [美] Stanley B. Lippman / [美] Josée Lajoie / [美] Barbara E. Moo
出版社: 电子工业出版社
出品方: 博文视点
原作名: C++ Primer, 5th Edition
译者: 王刚 / 杨巨峰
出版年: 2013-9-1
页数: 838
定价: 128.00元
装帧: 平装
ISBN: 9787121155352

豆瓣评分

9.4 ★★★★½ 2188人评价

5星 78.5%
4星 17.9%
3星 2.5%
2星 0.4%
1星 0.6%

# 一 初识C++

## 1.1 简单的C++程序

### 1.1.1 `C++ Hello World`

```
#include <iostream>     // 预编译指令,引入头文件iostream
#include <stdio.h>
using namespace std;    // 使用标准命名空间

int main(int argc, char *argv[]) {
    cout << "hello world" << endl;       // 和printf功能一样,输出字符串"hello world"
    return 0;
}
```

**分析:**

```
- #include <iostream>                // 预编译指令,引入头文件iostream
- using namespace std;               // 使用标准命名空间
- cout << "hello world" << endl;     // 和printf功能一样,输出字符串"hello wrold"
```

**问题1**:`c++`头文件为什么没有`.h`?

> 在c语言中头文件使用扩展名.h,将其作为一种通过名称标识文件类型的简单方式。但是c++得用法改变了,c++头文件没有扩展名。
>
> 但是有些c语言的头文件被转换为c++的头文件,这些文件被重新命名,丢掉了扩展名.h(使之成为c++风格头文件),并在文件名称前面加上前缀c(表明来自c语言)。例如c++版本的math.h为cmath.

| 头文件类型 | 约定 | 示例 | 说明 |
| --- | --- | --- | --- |
| c++旧式风格 | 以.h结尾 | iostream.h | c++程序可用 |
| c旧式风格 | 以.h结尾 | math.h | c/c++程序可用 |
| c++新式风格 | 无扩展名 | iostream | c++程序可用,使用namespace std |
| 转换后的c | 加上前缀c,无扩展名 | cmath | c++程序可用,可使用非c特性,如namespace std |

## 1.2 面向过程

- 面向过程是一种以过程为中心的编程思想。通过分析出解决问题所需要的步骤,然后用函数把这些步骤一步一步实现,使用的时候一个一个依次调用就可以了。
- 面向过程编程思想的核心:功能分解,自顶向下,逐层细化(程序 = 数据结构 + 算法)。
- 面向过程编程语言主要缺点是不符合人的思维习惯,而是要用计算机的思维方式去处理问题,而且面向过程编程语言重用性低,维护困难。

### 1.3 面向对象

**面向对象编程**(OOP,Object Oriented Programming)思想是一种编程范式,也是一种计算机编程架构和方法。OOP通过将程序视为对象的集合来进行设计,将现实世界中的各种实体抽象为对象,并通过类(class)来定义对象的属性和行为。

OOP还引入了封装(encapsulation)、继承(inheritance)、多态(polymorphism)和抽象(abstraction)等概念,这些概念可以帮助程序员更好地组织和管理代码,提高代码的可重用性、可维护性和可扩展性。在OOP中,对象之间通过消息传递相互通信,来模拟现实世界中不同实体间的联系。OOP被广泛应用于各种编程语言中,如C++、Java、Python、C#等。OOP的目标是提高代码的可重性、可维护性和可扩展性。

在面向对象中,算法与数据结构被看做是一个整体,称作对象,现实世界中任何类的对象都具有一定的属性和操作,也总能用数据结地来描述,所以可以用下面的等式来定义对象和程序:

```
对象 = 算法 + 数据结构
程序 = 对象 + 对象 + ......
```

从上面的等式可以看出,程序就是许多对象在计算机中相继表现自己,而对象则是一个个程序实体。 面向对象编程思想的核心:应对变化,提高复用。

### 1.4 面向对象几大特性

1. **封装**:封装是指将数据和操作数据的函数捆绑在一起,形成一个独立的实体,即对象。通过封装可以隐藏对象的内部实现细节,控制对数据的访问,只通过对象提供的接口与外界交互,这使得代码更加模块化,更容易理解和维护。C++中可以使用类来实现封装。
2. **继承**:继承是面向对象编程中的一个重要概念,它所表达的是类之间相关的关系,这种关系使得对象可以继承另外一类对象的特征和能力。一个子类可以继承父类的属性和方法,同时还可以添加自己特有的属性和方法。这有助于避免公用代码的重复开发,减少代码和数据冗余。在C++中,使用冒号(:)和访问修饰符来定义继承关系。
3. **多态**:意味着可以使用相同的接口表示不同的类型,这使得代码更加灵活和可扩展。通过使用继承和接口实现多态性,可以轻松地扩展和维护代码,以适应新的需求和变化。在C++中,可以通过虚函数来实现多态。它是面向对象编程领域的核心概念。
4. **抽象**:抽象是指只提供接口而不提供具体实现的方法。抽象类是一种特殊的类,可以包含纯虚函数。抽象类主要用于定义接口。C++中可以使用纯虚函数来定义抽象类。OOP编程通过使用抽象类和接口来实现高层次的抽象,这有助于隐藏实现细节,只暴露必要的接口。这使得代码更加灵活,可以更容易地适应不同的需求和变化。

## 二 C++对C的扩展

### 2.1 作用域运算符(域名操作符 `::`)

通常情况下,如果有两个同名变量,一个是全局变量,另一个是局部变量,那么局部变量在其作用域内具有较高的优先权,它将屏蔽全局变量。

```
// 全局变量
int a = 10;
void test() {
    // 局部变量
    int a = 20;
    // 全局a被隐藏
    cout << "a:" << a << endl;
}
```

程序的输出结果是: `a:20`

在test函数的输出语句中,使用的变量a是test函数内定义的局部变量,因此输出的结果为局部变量a的值。

作用域运算符可以用来解决局部变量与全局变量的重名问题

```
// 全局变量
int a = 10;
// 局部变量和全局变量同名
void test() {
    int a = 20;
    // 打印局部变量a
    cout << "局部变量a:" << a << endl;
    // 打印全局变量a
    cout << "全局变量a:" << ::a << endl;
}
```

这个例子可以看出,作用域运算符可以用来解决局部变量与全局变量的重名问题,即在局部变量的作用域内,可用 `::` 对被屏蔽的同名的全局变量进行访问。

### 2.2 命名空间

创建名字是程序设计过程中一项最基本的活动,当项目很大时,会不可避免地包含大量名字。c++允许我们对名字的产生和名字的可见性进行控制。在c语言中可以通过 `static` 关键字来使得名字只得在本编译单元内可见,在c++中我们将通过一种通过**命名空间**来控制对名字的访问。在c++中,名称(name)可以是符号常量、变量、函数、结构、枚举、类和对象等等。工程越大,名称互相冲突性的可能性越大。另外使用多个厂商的类时,也可能导致名称冲突。为了避免在大规模程序的设计中,以及在程序员使用各种各样的C++库时,这些标识符的命名发生冲突,标准C++引入关键字 `namespace` ,可以更好地控制标识符的作用域。

#### 2.2.1 命名空间的语法

**创建一个命名空间**:

```
namespace A {
    int a = 10;
}

namespace B {
    int a = 20;
}

void test() {
    cout << "A::a : " << A::a << endl;
    cout << "B::a : " << B::a << endl;
}
```

**命名空间只能全局范围内定义(以下是错误写法)**

```
void test() {
    namespace A {
        int a = 10;
    }

    namespace B {
        int a = 20;
    }
    cout << "A::a : " << A::a << endl;
    cout << "B::a : " << B::a << endl;
}
```

**命名空间可嵌套命名空间**

```
namespace OuterNamespace {
    int x = 10;
    namespace InnerNamespace {
        int y = 20;
        void foo() {
            std::cout << "Hello from InnerNamespace!" << std::endl;
        }
    }

    void bar() {
        std::cout << "Hello from OuterNamespace!" << std::endl;
    }
}

int main(int argc, char *argv[]) {
    OuterNamespace::x = 30; // 访问外部命名空间内的变量
    OuterNamespace::InnerNamespace::y = 40; // 访问嵌套命名空间内的变量
    OuterNamespace::InnerNamespace::foo();  // 调用嵌套命名空间内的函数
    OuterNamespace::bar(); // 调用外部命名空间内的函数
    return 0;
}
```

**命名空间是开放的,即可以随时把新的成员加入已有的命名空间中**

```
namespace A {
    int a = 10;
}

namespace A {
    void func() {
        cout << "hello namespace!" << endl;
    }
}

void test() {
    cout << "A::a : " << A::a << endl;
    A::func();
}
```

**声明和实现可分离**

```
namespace MySpace {
    void func1();
    void func2(int param);
}

void MySpace::func1() {
    cout << "MySpace::func1" << endl;
}

// 全局函数
void func1() {
    cout << "MySpace::func1" << endl;
}

void MySpace::func2(int param) {
    cout << "MySpace::func2 : " << param << endl;
}
```

**匿名命名空间,意味着命名空间中的标识符只能在本文件内访问,相当于给这个标识符加上了static,使得其可以作为内部连接**

```
namespace {
    int x = 10; // 匿名命名空间内的变量
    void foo() {
        std::cout << "Hello from anonymous namespace!" << std::endl;
    }
}

int main(int argc, char *argv[]) {
    x = 20;     // 访问匿名命名空间内的变量
    foo();      // 调用匿名命名空间内的函数
    return 0;
}
```

**命名空间别名**

除了在代码中显式定义命名空间外，`C++` 还支持使用 `using` 关键字来简化对命名空间内元素的访问。例如:

```
namespace MyNamespace {
    int x = 10;
    void foo() {
        std::cout << "Hello from MyNamespace!" << std::endl;
    }
}

int main(int argc, char *argv[]) {
    using MyNamespace::x;       // 简化访问命名空间内的变量
    using MyNamespace::foo;     // 简化调用命名空间内的函数
    x = 20;     // 访问命名空间内的变量
    foo();      // 调用命名空间内的函数
    return 0;
}
```

在上面的示例中,我们使用`using`关键字来简化对命名空间内变量和函数的访问。这样可以在不使用命名空间前缀的情况下直接访问和调用它们。但需要注意的是,过度使用`using`关键字可能会降低代码的可读性,应谨慎使用。

```
namespace veryLongName {
    int a = 10;
    void func() {
        cout << "hello namespace" << endl;
    }
}

void test() {
    namespace shortName = veryLongName;
    cout << "veryLongName::a : " << shortName::a << endl;
    veryLongName::func();
    shortName::func();
}
```

### 2.2.2 using声明

**using声明可使得指定的标识符可用。**

```
namespace A {
    int paramA = 20;
    int paramB = 30;
    void funcA() {
        cout << "hello funcA" << endl;
    }
    void funcB() {
        cout << "hello funcA" << endl;
    }
}

void test() {
```

```
    // 1.通过命名空间域运算符
    cout << A::paramA << endl;
    A::funcA();
    // 2.using声明
    using A::paramA;
    using A::funcA;
    cout << paramA << endl;
    // cout << paramB << endl; //不可直接访问
    funcA();
    // 3.同名冲突
    // int paramA = 20; // 相同作用域注意同名冲突
}
```

#### 2.2.3 命名空间使用

**使用命名空间需要注意以下几点:**

1. 命名空间的作用域:命名空间的作用域可以是全局的、局部的或私有的。在命名空间中定义的变量和函数只在该命名空间内部可见,而在命名空间外部则需要使用命名空间前缀来访问。
2. 命名空间的成员:命名空间可以包含变量、函数、类和其他实体。这些实体可以是全局的、静态的或内部的。如果一个命名空间定义在另一个命名空间内部,那么该命名空间只在该内部命名空间中可见。在命名空间内部,可以使用作用域限定符`::`来访问其他命名空间的成员。
3. 命名空间的继承:命名空间可以继承其他命名空间的成员。通过使用关键字`using`和`export`,可以在一个命名空间中导出和导入其他命名空间的成员。但是要注意避免命名冲突和循环引用的问题。当两个或多个命名空间相互引用时,会导致循环引用的问题。这可能会导致链接错误或运行时错误。为了避免这种情况,可以使用`using`指令或别名来简化访问命名空间内的实体。
4. 命名空间的冲突:由于不同的库或文件可能使用相同的名称,因此在使用命名空间时需要确保不会发生命名冲突。一种解决方式是使用前缀或命名空间限定符来区分不同的命名空间中的同名实体。
5. 命名空间的嵌套:一个命名空间可以嵌套在另一个命名空间中。这样可以更好地组织和管理代码,并避免命名冲突。

命名空间的分离也可以用于模块化编程。通过将不同的模块定义为不同的命名空间,可以更好地隔离模块之间的接口和实现,并避免命名冲突。需要注意的是,命名空间的分离应该遵循一定的规范和约定,例如使用一致的命名规则、避免命名冲突等。此外,在分离命名空间时应该考虑到代码的可维护性和可扩展性,以便于后续的代码维护和扩展。

### 2.3 更严格的类型转换

不同类型的变量一般是不能直接赋值的,需要相应的强制转换。

```
typedef enum COLOR {
    GREEN,
    RED,
    YELLOW
} color;

enum class Color {
    GREEN,
    RED,
```

```
    YELLOW
};

Color::GREEN;

int main(int argc, char *argv[]) {
    color mycolor = GREEN;
    mycolor = 10;
    printf("mycolor:%d\n", mycolor);
    return 0;
}
```

以上代码c编译器编译可通过,`c++`编译器无法编译通过。

### 2.4 `struct` 类型加强

- C语言中定义结构体变量需要加上`struct`关键字,`c++`不需要。
- C语言中的结构体只能定义成员变量,不能定义成员函数。`c++`即可以定义成员变量,也可以定义成员函数。

```
// 1. 结构体中即可以定义成员变量,也可以定义成员函数
struct Student {
    string m_strName;
    int m_nAge;

    void setName(string name) {
        m_strName = name;
    }

    void setAge(int age) {
        m_nAge = age;
    }

    void showStudent() {
        cout << "Name:" << m_strName << " Age:" << m_nAge << endl;
    }
};

// 2. c++中定义结构体变量不需要加struct关键字
int main(int argc, char *argv[]) {
    Student student;
    student.setName("John");
    student.setAge(20);
    student.showStudent();
}
```

### 2.5 新增 `bool` 类型关键字

`C++` 中, `bool` 是一种基本的数据类型,它只能存储两个值: `true` 和 `false` 。 `bool` 类型的变量通常用于逻辑比较,如条件判断等。

以下是关于 `C++ bool` 类型的一些基本特性:

- **存储值**: `bool` 类型的变量只能存储两个值: `true` 或 `false` 。
- **默认值**:未初始化的 `bool` 变量的默认值是 `false` 。
- **关系运算符**: `C++` 提供了以下关系运算符,可以用于比较两个 `bool` 类型的值:

> `==` :等于
>
> `!=` :不等于
>
> `<` :小于
>
> `>` :大于
>
> `<=` :小于等于
>
> `>=` :大于等于

**逻辑运算符**: `C++` 提供了以下逻辑运算符,可以用于组合或比较多个 `bool` 类型的值:

> `&&` :逻辑与(AND)
>
> `||` :逻辑或(OR)
>
> `!` :逻辑非(NOT)

**类型转换**:当一个 `bool` 类型的值转换为其他数据类型时, `true` 会被转换为1,而 `false` 会被转换为0。

**在 `switch` 语句中的使用**:在 `switch` 语句中, `bool` 类型的值可以代替 `case` 标签。例如:

```
bool myBool = true;
switch(myBool) {
case true:  //等同于 case 1:
    cout << "myBool is true";
    break;
case false: //等同于 case 0:
    cout << "myBool is false";
    break;
default:
    cout << "This should not happen";
}
```

**内存占用**:一个 `bool` 变量通常占用1个字节的内存空间。

**注意:** 在标准的C语言中并未定义 `bool` 类型,如果需要使用 `bool` 类型,程序员可以通过宏定义来自定义一个 `bool` 类型,定义语句如下:

```
#define bool int
#define false 0
#define true 1

typedef enum {
    false,
    true
} bool;
```

> C语言中的bool类型:
>
> C语言中也有bool类型,在C99标准之前是没有bool关键字,C99标准已经有bool类型,包含头文件stdbool.h,就可以使用和c++一样的bool类型。

### 2.6 三目运算符功能增强

C语言三目运算表达式返回值为数据值,为右值,不能赋值。

```
int a = 10;
int b = 20;
printf("ret:%d\n", a > b ? a : b);
// 思考一个问题,(a > b ? a : b) 三目运算表达式返回的是什么?
// (a > b ? a : b) = 100;
// 返回的是右值
```

C++语言三目运算表达式返回值为变量本身(引用),为左值,可以赋值。

```
int a = 10;
int b = 20;
printf("ret:%d\n", a > b ? a : b);
// 思考一个问题,(a > b ? a : b) 三目运算表达式返回的是什么?

cout << "b:" << b << endl;
// 返回的是左值,变量的引用
(a > b ? a : b) = 100;  // 返回的是左值,变量的引用
cout << "b:" << b << endl;
```

> [左值和右值概念]
>
> - 在c++中可以放在赋值操作符左边的是左值,可以放到赋值操作符右面的是右值。
> - 有些变量即可以当左值,也可以当右值。
> - 左值为Lvalue,L 代表 Location,表示内存可以寻址,可以赋值。
> - 右值为Rvalue,R 代表 Read,就是可以知道它的值。
>
>   比如:int temp = 10; temp在内存中有地址,10没有,但是可以Read到它的值。

### 2.7 C/C++中的`const`

#### 2.7.1 const概述

`const`单词字面意思为常数,不变的。它是c/c++中的一个关键字,是一个限定符,它用来限定一个变量不允许改变,它将一个对象转换成一个常量。例如:

```
const int a = 10;
a = 100;    // 编译错误,const是一个常量,不可修改
```

#### 2.7.2 C/C++中const的区别

1. C/C++中const异同总结

在c中,编译器对待const如同对待变量一样,只不过带有一个特殊的标记,意思是"你不能改变我"。在c++中定义const时,编译器为它创建空间,所以如果在两个不同文件定义多个同名的const,链接器将发生链接错误。简而言之,const在c++中用的更好。

2. 尽量以`const`替换`#define`

在旧版本`C`中,如果想建立一个常量,必须使用预处理器`#define MAX 1024`

我们定义的宏`MAX`从未被编译器看到过,因为在预处理阶段,所有的`MAX`已经被替换为了1024,于是`MAX`并没有将其加入到符号表中。但我们使用这个常量获得一个编译错误信息时,可能会带来一些困惑,因为这个信息可能会提到1024,但是并没有提到`MAX`。如果`MAX`被定义在一个不是你写的头文件中,你可能并不知道1024代表什么,也许解决这个问题要花费很长时间。

解决办法就是用一个常量替换上面的宏。

```
const int max = 1024;
```

> const 和 #define 区别总结:
>
> 1．const有类型,可进行编译器类型安全检查。#define无类型,不可进行类型检查.
>
> 2．const有作用域,而#define不重视作用域,默认定义处到文件结尾。如果定义在指定作用域下有效常量,那么#define就不能用。

**问题: 宏常量可以有命名空间吗?**

```
namespace MySpace {
    #define num 1024
}

void test() {
    // cout << MySpace::NUM << endl; // 错误
    // num = 100; // 1024 = 100 ❎
    cout << num << endl;
}
```

### 2.8 引用(Reference)

引用(Reference)是C++语言相对于C语言的又一个扩充,是C++常用的一个重要内容之一。类似于指针,只是在声明的时候用"&"取代了"*"。正确、灵活地使用引用,可以使程序简洁、高效。

#### 2.8.1 引用的特性

1. 引用被声明后就必须被初始化。一旦引用被初始化后,相当于目标变量名有两个名称,即该目标原名称和引用

   名,它就不能被重新初始化。
2. 引用在内存中并没有产生返回值的副本。
3. 引用本身就是一个目标变量的别名,对引用的操作就是对目标变量的操作。
4. 引用可以作为函数参数使用,以达到不产生副本、提高效率的目的。
5. 不能返回局部变量的引用，因为局部变量会在函数返回后被销毁，因此被返回的引用就成了无所指的引用，程序会进入未知状态。

#### 2.8.2 引用的应用

1. 引用的声明方法
2. 常引用
3. 引用作为参数(最常用)
4. 引用作为返回值
5. 引用和数组
6. 引用和指针

**1) 引用的声明方法**

类型标识符 &引用名 = 被引用对象

[例13]

```
int a = 10;
int &b = a;     // 引用
cout << a <<  " " << b << endl;          //
cout << &a << " " << &b << endl;         // 取地址
```

在本例中变量b就是变量a的引用,程序运行结果如下:

```
   10              10
0x61fe14        0x61fe14
```

从这段程序中我们可以看出变量a和变量b都是指向同一地址的,也即变量b是变量a的另一个名字,也可以理解为0x64fe68空间拥有两个名字:a和b。由于引用和原始变量都是指向同一地址的,因此通过引用也可以修改原始变量中所存储的变量值,如例2所示,最终程序运行结果是输出 两个20,可见原始变量a的值已经被引用变量b修改。

```
// 1. 认识引用
void test_01() {
```

```
    int a = 10;
    // 给变量a取一个别名b
    int &b = a;
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
    cout << "------------" << endl;
    // 操作b就相当于操作a本身
    b = 100;
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
    cout << "------------" << endl;
    // 一个变量可以有n个别名
    int &c = a;
    c = 200;
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
    cout << "c:" << c << endl;
    cout << "------------" << endl;
    // a,b,c的地址都是相同的
    cout << "a:" << &a << endl;
    cout << "b:" << &b << endl;
    cout << "c:" << &c << endl;
}

// 2. 使用引用注意事项
void test_02() {
    //1) 引用必须初始化
    //int &ref; // 报错:必须初始化引用
    //2) 引用一旦初始化,不能改变引用
    int a = 10;
    int b = 20;
    int &ref = a;
    ref = b;    // 不能改变引用
    //3) 不能对数组建立引用
    int arr[10];
    //int& ref3[10] = arr;
}
```

**2) 常引用**

- 如果我们不希望通过引用来改变原始变量的值时,我们可以按照如下的方式声明引用:
- `const` 类型标识符 & 引用名 = 被引用的变量名

这种引用方式称为常引用。如例15所示,我们声明b为a的常引用,之后尝试通过b来修改a变量的值,结果编译报错。虽然常引用无法修改原始变量的值,但是我们仍然可以通过原始变量自身来修改原始变量的值。我们用a=20;语句将a变量的值由10修改为20,这是没有语法问题的。

**[例15]** 通过常引用来修改原始值

```
int a = 10;
const int &b = a;
b = 20;  // compile error
a = 20;  // ✅
```

通过上述例子可以知道,通过引用我们可以修改原始变量的值,引用的这一特性使得它用于函数传递参数或函数返回值时非常有用。

### 3) 函数中的引用

如果我们在声明或定义函数的时候将函数的形参指定为引用,则在调用该函数时会将实参直接传递给形参,而不是将实参的拷贝传递给形参。如此一来,如果在函数体中修改了该参数,则实参的值也会被修改。这跟函数的普通传值调用还是有区别的。

参数传递分为值传递和引用传递:

1. 值传递: 按值传递,按值传递如果传递很大的数据项,赋值数据将导致较长的执行时间
2. 引用传递: 避免复制大量数据的开销,可以提高性能

传递引用给函数与传递指针的效果一样,用引用作为参数比使用指针更有清晰的语法

**[例16]**

```
void swap(int &x, int &y);         // 引用作为参数
void swap(int &x, int &y) {        // 函数实现几乎和原来一样
    int temp = x;
    x = y;
    y = temp;
}

void swap(int *x, int *y) {
    int z = *x;
    *x = *y;
    *y = z;
}
```

**[例17]**

```
#include <iostream>
using namespace std;

void foo(int val) {
    val = 10;
}

void bar(int &val) {
    val = 10;
}

void zoo(int *pval) {
    *pval = 10;
}

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

```
    int a = 1;
    int b = 1;
    int c = 1;
    foo(a);
    bar(b);
    zoo(&c);
    cout << a << " " << b << " " << c << endl;

    return 0;
}
```

#### 4) 引用作为返回值

函数返回值时,要生成一个值的副本。而引用返回值时,不生成值的副本,所以提高了效率。

**[例18]**

```
int result = 0;
int &func(int r) {    // 返回引用
    result = r * r;
    return result;
}
```

**注意:如果返回不在作用域内的变量或者对象的引用就有问题了。这与返回一个局部作用域指针的性质一样严重。**

**[例19]**

```
int &func(int r) {
    int result = 0;
    result = r * r;
    return result ;            // 返回局部变量的引用
}

int main(int argc, char *argv[]) {
    int &val = func(5);        // 返回的引用是个局部变量
    return 0;
}
```

在例4和例5中,我们就是为了避免这一点才采用的引用传递参数。普通的传值返回则不存在这样的问题,因为编译器会将返回值拷贝到临时存储空间后再去销毁b变量的。

#### 5) 引用和数组

在C++中,引用可以与数组一起使用。我们可以使用引用来表示数组,使得对数组的操作更加简洁和高效。

需要注意的是,当使用引用来表示数组时,我们实际上是在使用数组的首地址来初始化引用。因此,当我们修改引用所代表的值时,也就修改了

数组中对应的元素的值。

不能遍历引用的数组

**[例23]**

```
int a[10] = {0};
int &ra[10] = a;     // error不能建立一个引用类型的数组
```

### 6) 引用和指针

C++中的引用和指针是两个不同的概念,尽管它们在某些方面具有相似的特性。

引用(Reference)可以看作是一个已经存在的变量的别名,使用引用的语法是在变量名前面加上&符号。一个引用被声明后,它必须被初始化,

且一旦一个引用被初始化后,它就不能被重新初始化。

指针(Pointer)则是一个变量,其值为另一个变量的地址。指针可以被视为是引用概念的扩展,它允许我们间接地访问和修改变量的值。

以下是一些关于C++引用和指针的对比:

1. 初始化:引用在声明时必须被初始化，而指针则可以在任何时候被初始化。
2. 间接访问:引用没有间接访问操作符(*),而指针则使用间接访问操作符来访问指针指向的值。
3. 修改值:引用没有指针算术操作符(+、-),而指针则可以使用这些操作符来改变指针指向的地址。
4. 空值:引用不能是空值,而指针则可以是空值。
5. 类型安全:引用比指针更类型安全。例如,一个int类型的引用不能被赋予一个char类型的值。但是,一个int类型的指针可以指向一个char类型的值。
6. 内存管理:引用本身就是一个目标变量的别名,对引用的操作就是对目标变量的操作,所以它不需要进行任何的内存申请和释放操作。而指针则需要显式地分配和释放内存以避免内存泄漏。
7. 没有引用的指针和引用的引用

总的来说,引用和指针都是C++中重要的概念,它们各有各的应用场景。引用通常用于简化代码和提高效率,而指针则常常用于动态内存分配和实现复杂的数据结构。

注意: 由于指针也是变量,所以可以有指针变量的引用,有空指针没有空引用,每一个引用都是有效的

**[例22]**

```
int *a = nullptr;   // nullptr是指空指针,NULL
(int *) &ptr = a;     // 表示int*的引用ptr初始化为a
int b = 8;
ptr = &b;           // ok, ptr是a的别名,是一个指针

void &a = 3;        // 注意这是不合法的
// void只是在语法上相当于一个类型,本质上不是类型,没有任何一个变量或对象类型为void
```

**[例24]**

```
int a;
int &ra = a;
int & *ptr = &ra;     // error企图定义一个引用的指针
```

**7) 引用的本质**

引用的本质在`c++`内部实现是一个指针常量

```
Type &ref = val;    // Type *const ref = &val;
```

c++编译器在编译过程中使用常指针作为引用的内部实现,因此引用所占用的空间大小与指针相同,只是这个过程是编译器内部实现,用户不可见。

```
// 发现是引用,转换为 int *const ref = &a;
void testFunc(int& ref) {
    ref = 100; // ref是引用,转换为*ref = 100
}

int main(int argc, char *argv[]) {
    int a = 10;
    int& aRef = a;  // 自动转换为 int *const aRef = &a;这也能说明引用为什么必须初始化
    aRef = 20;      // 内部发现aRef是引用,自动帮我们转换为: *aRef = 20;
    cout << "a:" << a << endl;
    cout << "aRef:" << aRef << endl;
    testFunc(a);
    return 0;
}
```

> [const引用使用场景]
>
> 常量引用主要用在函数的形参,尤其是类的拷贝/复制构造函数。
>
> 将函数的形参定义为常量引用的好处:
>
> - 引用不产生新的变量,减少形参与实参传递时的开销。
> - 由于引用可能导致实参随形参改变而改变,将其定义为常量引用可以消除这种副作用。
>
> 如果希望实参随着形参的改变而改变,那么使用一般的引用,如果不希望实参随着形参改变,那么使用常引用。

### 2.8.3 总结

引用是**别名，在声明时必须初始化，在实际代码中主要用作函数的形参**（有一个地方例外:类中可以声明，在其他地方初始化）

（1）**&**在此不是求地址运算，而是起引用的标识作用，类型标识符是指目标变量的类型。

（2）声明引用时，必须同时对其进行初始化。

（3）引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，**且不能再把该引用名作为其他变量名的别名。**

（4）声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。

（5）**不能建立数组的引用。**因为数组是一个由若干个元素所组成的集合，所以无法建立一个数组的别名。

### 2.9 内联函数

`C++`语言新增关键字`inline`,用于将一个函数声明为内联函数。在程序编译时,编译器会将内联函数调用处用函数体替换,这一点类似于C语言中的宏扩展。

内联函数的主要优势在于能够减少函数调用的开销,从而提高程序的执行效率。当程序中频繁调用某个函数时,内联函数可以避免每次调用时进行函数调用的开销,从而减少程序的运行时间。此外,内联函数还可以避免因为函数调用而产生的栈帧开销,从而节省内存空间。这对于内存资源受限的环境(如嵌入式系统)来说尤为重要。

使用内联函数的缺点就是,如果被声明为内联函数的函数体非常大,则编译器编译后程序的可执行码将会变得很大。另外,如果函数体内出现循环或者其它复杂的控制结构的时候,这个时候处理这些复杂控制结构所花费的时间远大于函数调用所花的时间,因此如果将这类函数声明为内联函数,意义不大,反而会使得编译后可执行代码变长。**通常在程序设计过程中,我们会将一些频繁被调用的短小且简单的函数声明为内联函数。为了使得inline声明内联函数有效,我们必须将inline关键字与函数体放在一起才行,否则inline关键字是不能成功将函数声明内联函数的。**如例11所示,inline关键字则无丝毫作用,而例12中则成功将swap函数声明为了一个内联函数。

**[例11]** inline 关键字放在函数声明处不会起作用:

```
inline void swap(int &a, int &b);         // 不是内联函数,也不会报错,只不过inline没有意义

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

**[例12]** inline 关键字应该与函数体放在一起:

```
void swap(int &a, int &b);

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

**注意：在类/结构体中的成员函数，如果在声明的时候并实现了该函数，不管加不加inline关键字，该函数都默认为内联函数。**

### 2.10 函数默认参数

c++在声明函数原型的时可为一个或者多个参数指定默认(缺省)的参数值,当函数调用的时候如果没有指定这个值,编译器会自动用默认值代替。

```
void TestFunc01(int a = 10, int b = 20) {
    cout << "a + b  = " << a + b << endl;
}
TestFunc01();       // 30
TestFunc01(15);     // 35
TestFunc01(15, 22); // 37
TestFunc01(, 35);   // ❎
void TestFunc02(int a, int b = 10, int c = 10) {}
TestFunc02(16);
TestFunc02(17,22);
TestFunc02(55,22,31);

// 注意点:
// 1. 形参b设置默认参数值,那么后面位置的形参c也需要设置默认参数
void TestFunc02(int a, int b = 10, int c = 10) {}
// 2. 如果函数声明和函数定义分开,函数声明设置了默认参数,函数定义不能再设置默认参数
void TestFunc03(int a = 0, int b = 0);
void TestFunc04(int a, int b) {}

int main(int argc, char *argv[]) {
    // 1.如果没有传参数,那么使用默认参数
    TestFunc01();
    // 2. 如果传一个参数,那么第二个参数使用默认参数
    TestFunc01(100);
    // 3. 如果传入两个参数,那么两个参数都使用我们传入的参数
    TestFunc01(100, 200);
    return 0;
}
```

> 注意:
>
> 函数的默认参数从左向右,如果一个参数设置了默认参数,那么这个参数之后的参数都必须设置默认参数。
>
> 如果函数声明和函数定义分开写,函数声明和函数定义不能同时设置默认参数。

### 2.11 函数的占位参数

c++在声明函数时,可以设置占位参数。占位参数只有参数类型声明没有参数名声明。一般情况下,在函数体内部无法使用占位参数。

```
void TestFunc01(int a, int b, int) {
    // 函数内部无法使用占位参数
    cout << "a + b = " << a + b << endl;
}

// 占位参数也可以设置默认值
void TestFunc02(int a, int b, int = 20) {
    // 函数内部依旧无法使用占位参数
```

```
    cout << "a + b = " << a + b << endl;
}

int main(int argc, char *argv[]) {
    // 错误调用,占位参数也是参数,必须传参数
    // TestFunc01(10, 20);
    // 正确调用
    TestFunc01(10, 20, 30);
    // 正确调用
    TestFunc02(10, 20);
    // 正确调用
    TestFunc02(10, 20, 30);
    return 0;
}
```

### 2.12 函数重载(`overload`)

`C++`函数重载(`Overloading`)是指在同一作用域内可以定义多个同名函数,它们的参数列表(参数的个数或类型)必须不同。函数重载是一种多态的表现形式,常见于面向对象的编程语言中。

> 函数重载的基本条件:
>
> - 同一个作用域
> - 函数名相同
> - 参数个数不同
> - 参数类型不同
> - 注意：函数名和参数相同，返回值不同的函数不能作为重载条件

在C++中,函数重载可以按照以下方式进行:

1. 按照参数的个数进行重载:例如,可以定义两个同名函数,一个接受一个整数参数,另一个接受两个整数参数。

```
void func(int x) {
    // 函数体
}

void func(int x, int y) {
    // 函数体
}
```

2. 按照参数的类型进行重载:例如,可以定义两个同名函数,一个接受一个整数参数,另一个接受一个字符串参数。

```
void func(int x) {
    // 函数体
}

void func(std::string str) {
    // 函数体
}
```

3. 按照函数返回类型进行重载:例如,可以定义两个同名函数,一个返回整数类型,另一个返回字符串类型。

```
int func() {
    // 函数体
    return 0;
}

std::string func() {
    // 函数体
    return "";
}
```

需要注意的是,函数重载的参数列表必须不同，否则会导致编译错误。此外，函数重载不能根据函数的返回类型进行区分,只能根据参数列表进行区分。

```
// 1. 函数重载条件
namespace A {
    void MyFunc() { cout << "无参数!" << endl; }
    void MyFunc(int a) { cout << "a: " << a << endl; }
    void MyFunc(string b) { cout << "b: " << b << endl; }
    void MyFunc(int a, string b) { cout << "a: " << a << " b:" << b << endl;}
    void MyFunc(string b, int a) {cout << "a: " << a << " b:" << b << endl;}
}

// 2.返回值不作为函数重载依据
namespace B {
    void MyFunc(string b, int a) {}
    // int MyFunc(string b, int a) {}   // 无法重载仅按返回值区分的函数
}
```

**注意:** 函数重载和默认参数一起使用，需要额外注意二义性问题的产生。

```
void MyFunc(string b) {
    cout << "b: " << b << endl;
}

// 函数重载碰上默认参数
void MyFunc(string b, int a = 10) {
    cout << "a: " << a << " b:" << b << endl;
}

int main(int argc, char *argv[]) {
    MyFunc("hello");    // 这时,两个函数都能匹配调用,产生二义性
    return 0;
}
```

### 2.13 extern "C"浅析

以下在`Linux`下测试:

> c函数: void MyFunc() {}，被编译成函数: MyFunc
> c++函数: void MyFunc() {}，被编译成函数: _Z6Myfuncv

通过这个测试，由于`c++`中需要支持函数重载，所以`c`和`c++`中对同一个函数经过编译后生成的函数名是不相同的，这就导致了一个问题，如果在`c++`中调用一个使用c语言编写模块中的某个函数，那么`c++`是根据`c++`的名称修饰方式来查找并链接这个函数，那么就会发生链接错误，以上例`c++`中调用`MyFunc`函数，在链接阶段会去找Z6Myfuncv，结果是没有找到的，因为这个MyFunc函数是c语言编写的，生成的符号是MyFunc。

**那么如果我想在c++调用c的函数怎么办?**

extern "C"的主要作用就是为了实现`c++`代码能够调用其他`c`语言代码。加上extern "C"后，这部分代码编译器按c语言的方式进行编译和链接,而不是按`c++`的方式。

**MyModule.h**

```
#ifndef MYMODULE_H
#define MYMODULE_H

#include <stdio.h>

#if __cplusplus     // 在C++环境中
extern "C" {
#endif
    void func1();
    int func2(int a,int b);
#if __cplusplus
}
#endif
#endif
```

`MyModule.c`

```
#include "MyModule.h"

void func1() {
    printf("hello world!\n");
}

int func2(int a, int b) {
    return a + b;
}
```

TestExternC.cpp

```
#include <iostream>
using namespace std;

extern "C" void func1();
extern "C" int func2(int a, int b);

int main(int argc, char *argv[]) {
    func1();
    cout << func2(10, 20) << endl;
    return EXIT_SUCCESS;
}
```

## 2.14 Lambda表达式

### 2.14.1 什么是`lambda`表达式

匿名函数是许多编程语言都支持的概念,有函数体,没有函数名。`C++ lambda`表达式是一种匿名函数,是一种简洁、强大的语法糖,灵活且强大的函数定义方式,极大地简化代码。它使用简单的语法,就可以方便地定义一个函数对象,并使用该对象进行函数调用。

### 2.14.2 `lambda`表达式的历史

C++ lambda表达式的历史可以追溯到C++11的引入。在C++11之前,函数对象和函数指针是实现函数封装和回调的主要方式。然而,这些方式往往导致代码冗长和复杂,同时也不易于维护。

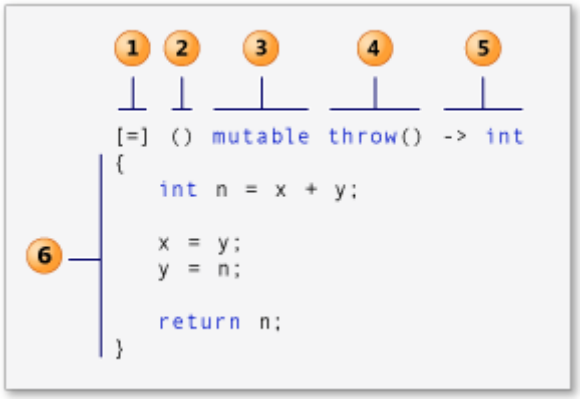
为了解决这些问题,C++11引入了lambda表达式。lambda表达式具有简单的语法和易于理解的结构,使得代码更加清晰和简洁。在C++14中,lambda表达式得到了进一步的改进和扩展,增加了默认参数、模板参数和广义捕获等功能,使其更加灵活和强大。这些改进使得lambda表达式在各种场景中更加方便使用,包括算法和容器的操作、自定义排序、函数回调等。随着时间的推移,lambda表达式逐渐成为C++中重要的特性之一,被广泛应用于各种场景中。它不仅简化了代码,提高了代码的可读性和可维护性,同时也为程序员提供了更多的编程方式和灵活性。

### 2.14.3 Lambda表达式的语法

Lambda表达式由三部分组成:捕获列表、参数列表和函数体。捕获列表用于指定需要捕获的外部变量,参数列表指定了函数的参数类型和名称,函数体包含了实现函数功能的语句。

Lambda表达式的语法如下:

```
[capture list](parameter list) { Statement }
[capture list](parameters list) mutable ->return-type { Statement };
```




```
capture clause(捕获)
lambda-parameter-declaration-list (变量列表)
mutable-specification (捕获的变量可否修改)
exception-specification (异常设定)
lambda-return-type-clause (返回类型)
compound-statement(函数体)
```

### 2.14.4 Capture List

Capture list是可选的,用于捕获外部变量。它指定了lambda表达式中使用的外部变量的访问权限。有以下三种类型的capture list:

```
[&]:默认情况下,如果外部变量在lambda表达式中被修改,那么它们将被按引用捕获。这意味着在lambda表达式中修改外部变量的值将影响原始变量。
[=]:如果外部变量在lambda表达式中不需要被修改,那么它们可以被按值捕获。这意味着在lambda表达式中修改外部变量的值不会影响原始变量。
[&x, y]:可以指定按引用捕获某些变量,按值捕获其他变量。例如,[&x, y=z]表示按引用捕获变量x,按值捕获变量 y,并将其初始化为 z 的值。
```

```
int t = 15;
[&](int x, int y) {
	t = 20;
}

int t = 15;
[=](int x, int y) {
	cout << t << endl;
}
```

### 2.14.5 Parameter List

Lambda表达式的参数列表指定了函数的参数类型和名称。参数列表可以是空的,也可以包含一个或多个参数。参数类型可以是任意类型,包括基本类型和自定义类型。以下是一些示例:

```
auto sum = [](int a, int b) { return a + b; }; // 两个整数参数
上面函数等价于：int add(int a, int b) { return a + b; }
auto average = [](double x, double y) { return (x + y) / 2; }; // 两个浮点数参数
auto concatenate = [](string str1, const string& str2) { return str1 + str2; };
// 一个字符串参数和一个常量字符串参数
```

#### 2.14.6 Statement

Lambda表达式的主体是函数的实现,它包含了一组语句,用于定义lambda表达式的行为。可以使用任何有效的C++语句,包括条件语句、循环语句、函数调用等。->return-type表示返回类型,如果没有返回类型,则可以省略这部分。这涉及到c++11的另一特性,参见自动类型推导,最后就是函数体部分。

**[例1]**

```
auto isEven = [](int n) { return n % 2 == 0; }; // 判断一个数是否为偶数
int num = 6;
if (isEven(num)) {
    cout << num << " is even" << endl;
}
else {
    cout << num << " is odd" << endl;
}
```

在这个例子中,我们定义了一个lambda表达式 `[](int n) { return n % 2 == 0; }`,用于判断一个整数是否为偶数。然后我们调用这个lambda表达式来检查变量 `num` 是否为偶数,并输出相应的信息。

**[例2]**

```
auto sum = [](int a, int b) { return a + b; };
int result = sum(3, 4); // result为7
```

在这个例子中,我们定义了一个lambda表达式 `[](int a, int b) { return a + b; }`,它接受两个整数参数,并返回它们的和。我们将这个lambda表达式赋值给变量 `sum`,然后调用它来计算3和4的和。

**[例3]**

```
#include <algorithm>    // 标准模板库算法库
#include <cmath>        // 数学库
#include <iostream>
using namespace std;

// 绝对值排序
void abssort(float *x, unsigned n) {
    // 模板库排序函数
    sort(x, x + n,
        // Lambda 开始位置
        [](float a, float b) {
            return (abs(a) < abs(b));
        } // lambda表达式结束
    );
}
```

```
int main(int argc, char *argv[]) {
    float a[5] = { 2.1f, 3.5f, 4.0f, 5.2f, 3.3f };
    abssort(a, 5);
    for (auto & x : a) {
        cout << x << endl;
    }
    return 0;
}
```

结果如下:

```
2.1
3.3
3.5
4
5.2
```

### 2.14.7 外部变量的捕获规则

默认情况下,即捕获字段为[]时,lambda表达式是不能访问任何外部变量的,即表达式的函数体内无法访问当前作用域下的变量。

如果要设定表达式能够访问外部变量,可以在 [] 内写入 & 或者 = 加上变量名,其中 & 表示按引用访问,= 表示按值访问,变量之间用逗号分隔,比如 [=factor, &total] 表示按值访问变量 factor,而按引用访问 total。

不加变量名时表示设置默认捕获字段,外部变量将按照默认字段获取,后面在书写变量名时不加符号表示按默认字段设置,比如下面三条字段都是同一含义:

```
[&total, factor]
[&, factor]
[=, &total]
```

**[例4]**

```
#include <functional>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    // lambda函数对象
    auto f1 = [](int x, int y) { return x + y; };
    cout << f1(2, 3) << endl;

    // function<int(int, int)>f2 = [](int x, int y) { return x + y; };
    auto f2 = [](int x, int y) { return x + y; };
    cout << f2(3, 4) << endl;
    return 0;
}
```

```
5
7
```

**不能访问局部变量**

```
int test = 100;
// lambda函数对象捕获局部变量
auto f1 = [](int x, int y) { return test + x + y; };
cout << f1(2, 3) << endl;
```

```
ciyeer@ubuntu:~$ gcc test2.cpp -o test2
test2.cpp: In lambda function:
test2.cpp:8:39: error: ‘test’ is not captured
    8 |     auto f1 = [](int x, int y){return test+x+y;}
      |                                       ^~~~
test2.cpp:8:16: note: the lambda has no capture-default
    8 |     auto f1 = [](int x, int y){return test+x+y;}
      |                ^
test2.cpp:6:9: note: ‘int test’ declared here
    6 |     int test = 100;
      |         ^~~~
test2.cpp: In function ‘int main()’:
test2.cpp:9:5: error: expected ‘,’ or ‘;’ before ‘cout’
    9 |     cout << f1(2, 3) << endl;
      |     ^~~~
ciyeer@ubuntu:~$
```

**如何传进去局部变量**

```
int test = 100;
// lambda函数对象捕获局部变量
auto f1 = [test](int x, int y) { return test + x + y; };
cout << f1(2, 3) << endl;
```

**默认访问所有局部变量**

```
int test = 100;
// lambda函数对象捕获所有局部变量
auto f1 = [=](int x, int y) { return test + x + y; };
cout << f1(2, 3) << endl;
```

# 课堂小结

- 本篇介绍了C++的基础知识,面向过程和面向对象的特性以及C++对C语言的扩展等内容。通过对本篇内容的学习可以掌握什么是面向对象的编程思想,和我们之前学的C语言面向过程思维的区别。
- C++对C的扩展主要体现在面向对象编程的引入,以及对语言功能的丰富和增强,这使得 C++ 相较于C语言而言更适合大型和复杂的软件系统的设计和开发,使得开发者能够更方便地进行复杂和灵活的编程。然而,C++仍然保留了对C语言的兼容性,因此C语言的代码通常可以在C++中运行。
- C++ 通常具有比 Java和C#等纯面向对象的语言有更好的性能,因为 C++ 的执行方式更接近底层硬件。C++ 允许直接操作内存,提供更细粒度的控制,适用于对性能和底层控制有较高要求的场景和应用,如游戏开发、嵌入式系统等。

# 随堂作业

编写一个程序，实现以下功能：

1. **函数重载：**
   - 编写多个 `calculate` 函数，支持两个整数、两个浮点数、一个整数和一个浮点数的加法操作（函数需要重载）。
2. **Lambda 表达式：**
   - 编写一个函数 `applyOperation`，该函数接收一个 Lambda 表达式和两个整数，调用 Lambda 表达式执行相应的运算并返回结果。
3. **引用：**
   - 编写一个函数 `swap`，使用引用参数交换两个整数的值。
   - 在 `main` 函数中调用 `swap` 函数，并输出交换前后的值。

**要求：**

- `calculate` 函数需要进行函数重载。
- 使用 Lambda 表达式实现不同的运算（如加法、减法）。
- 使用引用传递参数的方式实现 `swap` 函数。