C++ 教程(C++11版)

零基础入门系列教程

作者: 魏明择

2026 年版

https://weimingze.com

目录

序

前言

C++ 是一个为效率而生的编程语言，这个效率包括开发效率和运行效率，但这个效率不包含学习效率。

先不要放弃，无论如何我都建议你认真的读完前几章，只有这样你才能真正的有所收获并爱上 C++。

C++ 语言是我学过的最难学的编程语言，但当你搞懂了 C++ 语言的各种语法后，你会发现他的精妙和作者的良苦用心。这绝对是一个门无可替代的经典的编程语言。近三十年是这样。三十年后应当也是如此。

我第一次学习使用 C++ 是大学时期，当时使用的是 Borland 公司的 C++ Builder 作为编辑器和编译器。工作以后最早拿来当作赚钱工具的编程语言也是 C++，我记得 2002年我在编写 PLC 编程器程序的界面时就是使用的 Visual Studio C++ 6.0，这绝对称得上是微软的巅峰之作。我在使用 C++ 的同时也逐渐明白了 C++ 的精妙之处，这为我后来学习各种编程语言打下了良好的基础。

此教程不同于大学课本里的枯燥的 C++ 教程。本教程将从实用出发，先讲解 C++ 的基本语法，然后逐步刨析语法的底层原理。无论你是初学者还是用过十几年的 C++ 老程序员，我相信你都会有所收获。

C++ 语言是基于 C 语言的编程语言。他是在 C 语言的基础上使用扩充的语法来实现更强大的功能。在学习 C++ 之前你需要先学习 C 语言。如果你还不够了解 C 语言请先看我的《C语言教程》。

学会了 C++ 语言，你就很快能学会 Java、Python 等编程语言。当然，你也可以先学习其他编程语言，然后反过来学习 C++ 语言也同样对学习有所帮助，编程语言都是相通的。而 Java 和 Python 都在仿照 C++ 的语法来进行设计的。

此教程会先讲解标准的 C++ 的特性，后面再讲解 C++11 的新特性。循序渐进进行讲解。

版权声明

C++ 语言简介

C++ 语言是一种编译型的编程语言。C++语言通过编译器将用户编写的 C++ 源程序的代码编译成处理器能够识别的机器指令，然后直接在处理器上执行。C++ 语言编写的程序可以在有操作系统的计算机系统上运行，也可以在没有操作系统的单片机上运行。

C++语言由本贾尼·斯特劳斯特卢普（Bjarne Stroustrup）于1979年在AT&T贝尔实验室研发。 C++ 在 C 语言的基础上扩充了面向对象和模板等新的特性，这让编写大型的软件程序变得更加轻松，同时也不会降低程序的执行效率。因此 C++ 可以称得上是最优秀的编译型编程语言。

C++ 作者：本贾尼·斯特劳斯特卢普（Bjarne Stroustrup）

为什么学 C++语言

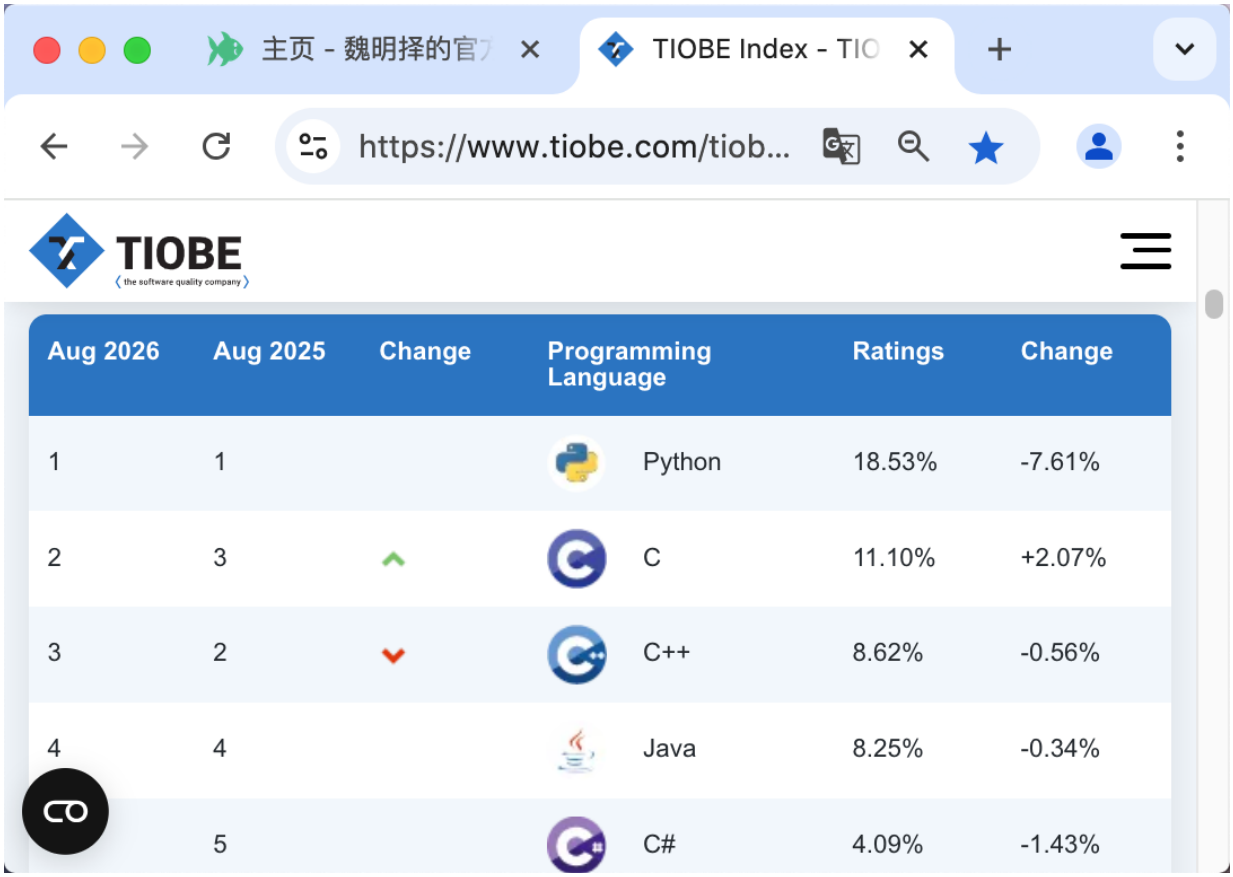
在国内大厂公司、军工企业的软件开发中，C++ 的使用占比较多，就业职位多。

C++的应用领域

C++语言在嵌入式、设备驱动程序、高性能计算（HPC）、游戏、云计算、桌面应用、移动应用、算法等领域有着不可替代的地位。

C++ 语言目前的状况

C++ 语言从诞生到现在有四十多年的历史。并且经久不衰，目前经 tiobe-index (最权威的编程语言排行网站) 的统计，截止 2026年8月，C++语言的排名是第三位，并且持续保持领先的位置。如下图所示：



Aug 2026	Aug 2025	Change	Programming Language	Ratings	Change
1	1		Python	18.53%	-7.61%
2	3	▲	C	11.10%	+2.07%
3	2	▼	C++	8.62%	-0.56%
4	4		Java	8.25%	-0.34%
	5		C#	4.09%	-1.43%

下面是自 2002 年 ~ 今的 C++ 语言排名趋势图，可见 进 20年来 C++ 语言一直处于 领先的位置。

C++ 语言优点

1. 执行效率高，能直接在硬件上执行指令集，效率仅次于 C 语言。
2. 支持面向对象等特性，适用于大型软件的开发和维护。
3. 有标准模板库，标准数据结构和算法都在其中，弥补了 C 语言的不足。

C++ 语言缺点

1. 语法比较复杂，学习 C++ 需要先精通 C 语言，学习成本高。如果不精通语法也能难写出高效率的程序。这也是有些人用 C++ 做了多年开发也不敢说精通的原因。
2. 使用困难，如果不能理解 C++ 语法的用意依旧写不出高效率的程序。
3. 有些修饰语只能从编译器级别实现编译时名字空间隔离，但运行时不能真正的实现内存隔离，因此依旧会出现内存泄露和崩溃的可能。

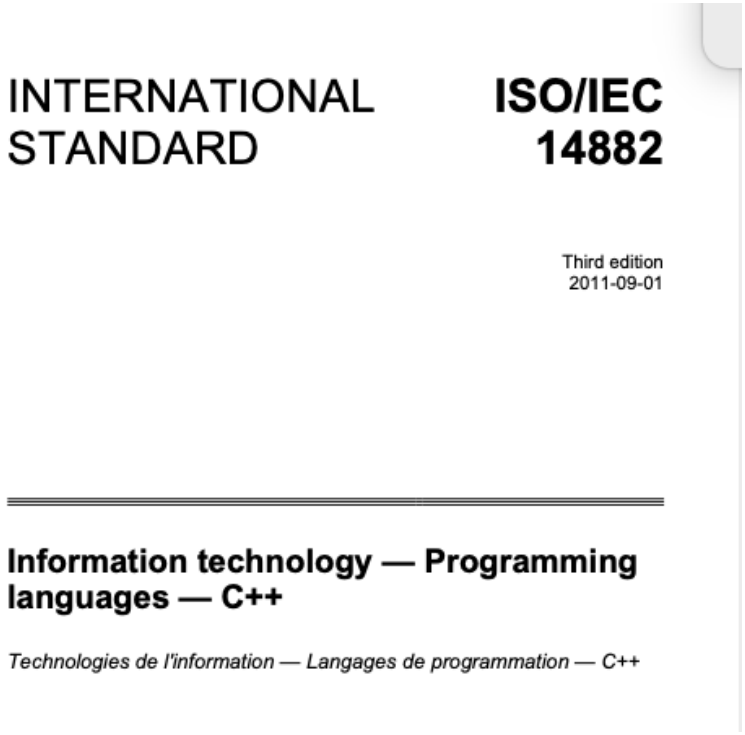
C++ 语言的标准

C++语言的标准经历了多个版本的演进，每个版本引入了新的特性和改进。以下列出几种主要的C++语言标准及其关键特性：

1. `C++98`（首个ISO标准）
   - 引入的标准模板库（STL）、异常处理机制、运行时类型识别 (RTTI) 等关键特性。
2. `C++03` (2003年)：
   - 技术性修正，主要修复C++98中的缺陷和Bug，没有引入新的语言特性.
3. `C++11`（ISO/IEC 14882:2011）
   - 引入auto类型推导、范围for循环、Lambda表达式、智能指针、移动语义等新特性，是现代 C++ 的代表作。
4. `C++14 / C++17 / C++20 / C++23`
   - 这四个版本在C++11的基础上进行了完善和少量扩展。

> 注: C++11 表示 2011 年发布的新标准。

本课程参照 C++11 标准（ISO/IEC 14882:2011）官方文档编写。请各位自行搜索下载。参照如下：


INTERNATIONAL STANDARD

ISO/IEC 14882

Third edition
2011-09-01

Information technology — Programming languages — C++

Technologies de l'information — Langages de programmation — C++

官方提供了 C++ 语言的官方文档，其内部记录的各种语法的详细解释以及不同版本的 C++ 标准的差别和解释。这个文档可以作为学习 C++ 标准的依据。

C++的官方文档：

https://en.cppreference.com/

第一章、开发环境的搭建

C++ 语言同 C 语言一样是编译型语言，我们编写 C++ 语言的程序时需要编辑器（如：vim、Visual Studio Code、记事本等）和编译器（如：g++、Clang、MSVC等）。当然我们也可以使用有些厂家提供的将编辑器、编译器、调试器等集成在一起的开发环境（如：XCode、Microsoft Visual Studio等）。

本课程的教学环境

本课程使用 Ubuntu 26.04 LTS Linux 操作系统作为教学环境。以 GNU 的 C 语言编译器集合中的 `g++` 作为编译器进行讲解。

本教程的案例均在 Ubuntu 26.04 LTS 上验证，学习本教程前请先自行安装 Ubuntu 26.04 LTS 操作系统或下载由本站制作的 vmware 虚拟机镜像。

Ubuntu26.04 VMware镜像:

- 百度网盘下载地址:
- https://pan.baidu.com/s/1mxbkL68C7EL7DnRTvLSkRA 提取码: angh

Ubuntu下载和安装方法详见: Ubuntu Linux 简介

1. Linux 安装 C++ 开发环境

Linux 下比较成熟的 C++ 语言编译器是 **gcc**编译器集合中的 `g++`。

gcc 是 GNU 项目组开发的，专门为 Linux 内核编译和应用程序编译而设计的编译器集合，其中包含编译 C++ 程序的编译器 `g++`。

1.1 g++ 安装

在 Ubuntu Linux 下使用 apt 命令。在 CentOS 下使用 yum 命令就可以很方便的安装 g++ 编译器。

有些 版本的 Linux 甚至已经将 g++ 预先安装好了，直接使用即可。

打开 Linux 的 **终端**，使用如下命令安装 g++。

Ubuntu 下安装 g++

```
sudo apt install g++
```

CentOS 下安装 g++

```
sudo yum install gcc-c++
```

验证 g++ 是否成功安装

在终端下，输入 g++ 命令，如果提示没有这个命令。则说明没有安装 g++。安装成功的效果如下：

```
weimz@mzstudio:~$ g++
g++: fatal error: no input files
compilation terminated.
```

> 以上提示是没有给出 C++ 语言的文件而导致的。

使用 g++ 的 `--version` 选项可以查看 g++ 的版本信息。如下所示：

```
weimz@mzstudio:~$ g++ --version
g++ (Ubuntu 15.2.0-16ubuntu1) 15.2.0
Copyright (C) 2025 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

> 当前 `g++` 的版本是 `15.2.0`。

至此，Linux 下的 C++ 语言编译器已经安装成功，你可以使用 g++ 将 C++ 语言的代码编译成可执行的应用程序了。

1.2 Vim 编辑器

C++ 语言程序都是文字信息，因此需要使用纯文本编辑器编写 C/C++ 语言的代码。

在 Linux 下可以使用 **Vim** 作为 C++ 语言的编辑器，也可以使用 Visual Studio Code 作为编辑器。以下介绍 Vim 编辑器的安装、配置和使用。

Ubuntu 下安装 Vim


```
sudo apt install vim
```

CentOS 下安装 Vim

```
sudo yum install vim
```

Vim 编辑器的用法详见我的 《Linux 教程》 的第五章的内容。

在使用 Vim 编写 C++ 语言代码时。可以修改 Vim 的配置文件 `.vimrc` 来更改默认配置，以便实现代码高亮，自动缩进等常用功能。

> 修改 `~/.vimrc` 只对本用户的配置生效。如果想对所有用户的vim配置，请修改 `/etc/vim/vimrc` 文件。

修改 .vimrc 方法:

终端内使用 Vim 打开用户主文件夹下的 `.vimrc` 文件。命令如下：

```
vim ~/.vimrc
```

以下是魏明择本人电脑的 Vim 配置，供参考。内容如下：

```
" 这是 .vimrc 配置文件的注释，注释以双引号(")开头直至行尾。
set number   "  设置显示行号
set tabstop=4      " 设置制表符(Tab)转换为4个空格
set shiftwidth=4   " 自动缩进时使用 4 个空格
set expandtab      " 将 Tab 转换为空格

set mouse=a   " 设置鼠标支持
set cursorline   " 高亮当前行
set showmatch   " 高亮显示匹配的括号

set cindent   " 设置 C 语言编写时自动缩进

syntax on   " 设置语法高亮
```

> 注： `.vimrc` 配置文件的注释是以 双引号(")开始直至行尾的内容。

至此，我们可以在 Linux 下编写 C++ 语言的代码了。

第二章、初步认识 C++ 语言

本章我们将从最简的C++语言程序开始了解 C++ 语言的语法规则、编写、编译及运行。

同 C 语言一样，C++程序需要使用文本编辑器进行编写，然后使用编译器将其编译成计算机能够识别的机器指令

> 注：不同的计算机需要使用不同的编译器进行编译。

C++ 语言的源文件的后缀通常是 `.cpp` 结尾。也可以使用 `.cxx`、`.cc`、`.C` 等作为 C++ 源文件的后缀名。本教程统一使用 `.cpp` 作为 C++ 的源程序文件的后缀。

1. 从 Hello World 开始

编写第一个 C++ 语言程序

使用编辑器编写一个文件 `hello.cpp`

内容如下：

```
#include <iostream>

int main(int argc, char * argv[]) {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

> 在 Linux 下可以使用 **Vim** 或 **文本编辑器** 编写。

使用 gcc 编译这个文本文件成为可执行文件。

打开一个 UNIX 或 Linux 终端，在终端中找到这个 `hello.cpp` 文件的位置。然后执行 `g++ hello.cpp` 来编译这个 `hello.cpp` 文件。如下所示：

```
weimz@mzstudio:~$ g++ hello.cpp
weimz@mzstudio:~$ ls
a.out  hello.cpp
```

可见此时多了一个可执行文件 `a.out`。

> 如果你是使用 MacOS 系统或 Windows 系统下的其它 C 语言开发环境。请找到相关的方法进行编译和运行。

在 Linux 运行编译后的可执行程序

```
weimz@mzstudio:~$ ./a.out
Hello World!
```

此时你可以看见在终端中打印了一行 `Hello World!`。至此你已经生成了第一个 C++ 语言编写的可执行程序（a.out）。

对比 C 语言编写同样功能的程序如下：

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    return 0;
}
```

从上述示例中我们能够看出 C 语言和 C++ 语言都使用标准 C 的程序入口函数 `main` 函数，并且返回值了类型和含义也同标准 C 语言的返回值意义完全相同。

我们发现在 C++ 语言中 `#include <iostream>` 代替了 C 语言中的 `#include <stdio.h>`，头文件 `iostream` 是 C++ 标准库中用于声明标准输入输出的头文件之一。

C++ 标准库中的头文件都是没有后缀名的头文件，这样做的目的是避免 C++ 头文件和 C 语言的头文件冲突。

再来看打印 `Hello World!` 的 `std::cout << "Hello World!" << std::endl;` 他的功能和 C 语言中的 `printf("Hello World!\n");` 是完全一样的。 其中 `cout` 是负责标准输出的一个变量（C++中叫对象），我们将字符串 `"Hello World!"` 通过 `<<` 这个**左移运算符** 交给变量`cout`，则这个字符串则会打印到控制台终端中。其中 `std` 是 C++ 标准库中的名字空间（后面才讲）。`::` 是名字空间的说明符，`std::cout` 表示名字空间 `std` 中的 `cout`变量（对象）。 `std::endl` 是 `std` 中的换行符 `endl`，它等同于 `"\n"`，因为字符串 `"Hello World!"` 中并没有换行符。我们将输出 `Hello World!` 的一行改成 `std::cout << "Hello World!";` ，则打印后则不会换行。如果你想再打印后直接换行也可以写成 `std::cout << "Hello World!\n";` ，将 换行符 `\n` 放在字符串中。

我们先将 C++ 了解 到这里，接下来我们完成下面的实验。

实验：

1. 编写上述 `hello.cpp` 然后编译运行。
2. 将 `hello.cpp` 中的内容全部更换成 C 语言代码如下，然后编译运行，看能否运行？为什么？

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    return 0;
}
```

2. 布尔类型

C++ 里新增了布尔类型（`bool`），用于表示布尔类型的数。

在 C 语言中，经常使用整数（`int`）或 字符型（`char`）来表示布尔类型的数，通常类型不统一（看程序编写人员的心情），并且数值表示的含义也不统一。比如 C 语言的关系运算符 `<` 通常返回 `1` 表示真，而有些函数则返回 `0` 表示执行成功。这给代码的阅读人员带来的极大的困惑。

C++ 统一的表示布尔的数据类型。其布尔类型名为**bool**，其值有两个 `true` 和 `false`。其本质还是一个整数，但比起使用 `int` 或 `char` 类型，类型更加明确。

布尔类型的值：

```
true   // 表示真（值为1）
false  // 表示假（值为0）
```

布尔类型相关关键字

```
bool    true   false
```

示例：

```
// filename: type_bool.cpp
#include <iostream>

using namespace std;

int main(int argc, char * argv[]) {
    bool b1 = false;  // b1 为布尔类型变量
```



```
    b1 = 200 > 100;
    cout << "b1:" << b1 << endl;

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o type_bool type_bool.cpp
weimz@mzstudio:~$ ./type_bool
b1:1
```

说明

布尔类型所真用的空间因编译器和硬件设备不同而不同，即 `sizeof(bool)` 的返回值可能是 `1`、`4` 等。

实验

尝试使用 `bool` 类型定义变量，然后将其赋值为 `100`，再打印此布尔类型变量的值。

第三章、名字空间

在 C 语言中，当多人合作开发一个项目是难免会出现名字冲突问题，如：张三写了一个文件 `a.c`，内部定义了一个全局变量 `int count = 100;`，同时李四写了一个文件 `b.c`，内部也定义了一个全局变量 `int count = 200;`，因为 C 语言中全局变量的名称是全局可见，因此在编译上述两个 `.c` 文件时并不会出错，但在链接阶段就会报错，提示类似于：**重复定义** 这类的错误。

张三写的 `a.c`

```
int count = 100;
```

李四写的 `b.c`

```
int count = 200;
```

如果要解决上述错误，我们可以让其中的一个全局变量 `count` 变为当前模块文件（`.c`）文件内可见即可。如，将李四定义的变量`count` 用 `static` 关键字修饰即可。修改如下：

李四写的 `b.c` 修改如下：

```
static int count = 200;
```

这样在程序链接阶段就不会出现错误了。这样虽然解决了当前名字冲突问题，但当王五、赵六参加到项目组的时候，又可能会出现名字冲突问题。

C++ 语言中给出了命名空间（`namespace`）的概念，他使用 `namespace` 关键字，将多个同名的变量、函数、类或模板定义在不同的名字空间中，在使用时名明确指定名字空间即可。例如开始的 `hello.cpp` 程序中的 `std::cout` 和 `std::endl`，就是 `std` 名字空间中的两个变量。

1. 命名空间定义

在现实世界中，名字叫 `魏明择` 的人可能有很多。比如我是那个在北京市朝阳区的**魏明择**，清华大学机电系还有一个**魏明择**，中科院研究所里还有一个**魏明择**。那如何区分这三个人呢？比如一个美女邮寄一本书给魏明择，那么该怎样指定这三个人中的一个呢？当然要在名字的前面加上定义，如 **北京市朝阳区的** 魏明择，这个**北京市朝阳区**就是一个名字空间。现实当中不同的名字空间内可以有多个不同名称的人而不会出现名字冲突。

在 C++ 语言中，每个人写的变量和函数等可以在自己的名字空间中。名字空间用自己的名字，其中可以放入多个不同明的变量、函数、类或模板。

名字空间也叫命名空间，是一个标识符的范围空间。

下面我们来看一个定义名字空间的语法：

语法:

```
namespace 命名空间名 {
   // 此处定义此空间内的变量和函数、类等。
}  // 注意：此处没有分号。
```

> 注：名字空间名必须是**标识符**。

示例:

在不同名字空间内定义名字为 `mystr` 的字符型指针和名字为 say_hello 的函数。

定义 `englishi_env` 名字空间来存放在**英文环境**下运行的数据和函数。

```
// 定义英文运行环境的名字空间
namespace english_env {
    const char * mystr = "Hello";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}
```

定义 `chinese_env` 名字空间来存放在**中文环境**下运行的数据和函数。

```
// 定义中文运行环境的名字空间
namespace chinese_env {
    const char * mystr = "你好";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}
```

这样我们就定义个两个可用的名字空间 `english_env` 和 `chinese_env`。

完整代码如下:

```
#include <iostream>

// 定义英文运行环境的名字空间
namespace english_env {
    const char * mystr = "Hello";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}

// 定义中文运行环境的名字空间
namespace chinese_env {
    const char * mystr = "你好";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}

int main(int argc, char * argv[]) {
    return 0;
}
```

实验

编译并运行上述代码，看是否能够正确编译。

> 此程序没有任何打印输出。

2. 名字空间的使用

我们声明了两个名字空间 `english_env` 和 `chinese_env` 内部都分别定义一个变量 `mystr` 和 一个函数 `say_hello`。

这个两个变量和两个函数同普通的全局变量和函数不同，它们是定义在名字空间内的标识符，在使用时需要手动指定名字空间的名称。下面我们来说一下如何使用这些变量和函数。

2.1 使用作用域限定符（::）

语法：

```
名字空间名::标识符
```

示例：

```
// 访问变量
std::cout << english_env::mystr << std::endl;
std::cout << chinese_env::mystr << std::endl;

// 调用函数
chinese_env::say_hello("魏老师");
english_env::say_hello("魏老师");
```

2.2 使用 `using` 声明（引入单个成员）

使用 `using` 关键字可以将名字空间中的一个标识符声明成为本地的标识符，然后就可以像使用全局变量一样使用这个标识符了。

语法：

```
using 名字空间名::标识符;
```

示例：

```
// 声明 english_env 空间内的  mystr 为当前作用域内的变量。
using english_env::mystr;
// 声明 chinese_env 空间内的  say_hello 函数为当前作用域内。
using chinese_env::say_hello;

// 访问 english_env::mystr 变量
std::cout << mystr << std::endl;
// 调用函数
say_hello("魏老师");
```

2.3 使用 `using namespace 指令`（引入整个名字空间的标识符）

上述做法仅仅在使用该名字空间内少量的标识符的时候比较方便。当某个名字空间内有大量的标识符声明，并都会频繁的使用时，则我们可以使用 `using namespace 指令`将该名字空间内的所有标识符导入到当前声明的作用域内。此时在当前作用域内使用该名字空间内的任何标识符都不需要指定名字空间名。

语法：

```
using namespace 名字空间名;
```

示例：

```
// 声明 chinese_env 内的所有标识符到当前作用域内。
using namespace chinese_env;

// 使用其中的标识符
std::cout << mystr << std::endl;
say_hello("老魏");
```

完整的可运行的示例代码：

```
#include <iostream>

// 定义英文运行环境的名字空间
namespace english_env {
    const char * mystr = "Hello";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}

// 定义中文运行环境的名字空间
namespace chinese_env {
    const char * mystr = "你好";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}

int main(int argc, char * argv[]) {
    // 访问变量
    std::cout << english_env::mystr << std::endl;
    std::cout << chinese_env::mystr << std::endl;

    // 调用函数
    chinese_env::say_hello("魏老师");
    english_env::say_hello("魏老师");

    // 声明 english_env 空间内的  mystr 为当前作用域内的变量。
    using english_env::mystr;
    // 声明 chinese_env 空间内的  say_hello 函数为当前作用域内。
    using chinese_env::say_hello;

    // 访问 english_env::mystr 变量
    std::cout << mystr << std::endl;
    // 调用函数
    say_hello("魏老师");

    // 声明 chinese_env 内的所有标识符到当前作用域内。
    using namespace chinese_env;

    // 使用其中的标识符
    std::cout << mystr << std::endl;
    say_hello("老魏");
```

```
    return 0;
}
```

实验

改写之前的 `hello.cpp`，使用 `using namespace std;` 将 std 名字空间内的所有标识符声明到 `main` 函数中，在将 原来的 `std::cout << "Hello World!" << std::endl;` 改写成 `cout << "Hello World!" << endl;` 看能否编译通过并查看运行结果。

可以参照如下代码进行改写：

```
#include <iostream>

using namespace std;

int main(int argc, char * argv[]) {
    cout << "Hello World!" << endl;
    return 0;
}
```

3. 名字空间的声明拆分

当我们项目的比较庞大，需要编写功能比较多，同时我们需要将这些标识符声明在同一个名字空间内，按我们之前的做法可能是仅仅写一个 `.cpp` 文件，然后定义一个名字空间（`namespace`），然后将所有的名字空间声明都写在其中，这种做法虽然可行，但当我们维护这些代码时就变得无比的麻烦。因此 C++ 中，我们可以用 **名字空间的声明拆分** 的方法将一个名字空间的声明拆分成多个部分，甚至放在不同的文件中。

例如，之前的名字空间 `chinese_env` 的声明中，将两个标识符 `mystr` 和 `say_hello` 声明在一段代码中，如下所示。

```
// 定义中文运行环境的名字空间
namespace chinese_env {
    const char * mystr = "你好";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}
```

其实我们可以将上述两个标识符的声明写成两部分，并放入到不同的文件中。如下所示：

```
// 定义中文运行环境的名字空间
namespace chinese_env {
```



```
    const char * mystr = "你好";
}
```

```
// 定义中文运行环境的名字空间
namespace chinese_env {
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}
```

这样仍然不影响其使用。

4. 名字空间嵌套声明

在声明名字空间时，可以实现嵌套声明，即名字空间内仍然可以再声明内部更深一层的名字空间，可以实现多层名字空间的声明。

示例：

在 `Outer` 名字空间内部嵌套声明 `Innter` 名字空间。

```
namespace Outer {
    int total = 100;
    void print_total(void) { std::cout << total << std::endl; }
    namespace Inner {
        int sub_count = 0;
        void other(void) {
            std::cout << "this is Inner namespace" << std::endl;
        }
    }
}
```

当有嵌套声明的名字空间时，可以使用作用域限定符（`::`） 来指定名字空间内部的标识符，也可以使用 `using` 或者 `using namespace` 来指定一个或全部标识符。用法几乎和没有嵌套的名字空间的用法一致。如：

1. 使用作用域限定符访问名字空间内的标识符。

```
std::cout << Outer::total << std::endl;
Outer::print_total();

std::cout << Outer::Inner::sub_count << std::endl;
Outer::Inner::other();
```

1. 使用 `using` 使用名字空间中的标识符

```
using Outer::total;
using Outer::print_total;
std::cout << total << std::endl;
print_total();

using Outer::Inner::sub_count;
using Outer::Inner::other;
std::cout << sub_count << std::endl;
other();
```

1. 使用 `using namespace` 使用名字空间中的标识符

```
using namespace Outer;
std::cout << total << std::endl;
print_total();

using namespace Outer::Inner;
std::cout << sub_count << std::endl;
other();
```

5. 命名空间声明和实现分离

在上述的示例中。我们通常在声明名字空间的时候就已经将函数的实现等加入到名字空间中。而在实际工作中，通常我们要将名字空间中的变量、函数、类的声明写在 `.h` 的头文件中，而将具体的函数、成员函数（后面才讲）等的实现写在 `.cpp` 文件中。

以如下的名字空间声明为例：

```
namespace english_env {
    const char * mystr = "Hello";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}

namespace chinese_env {
    const char * mystr = "你好";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}
```

通常我们拆分成声明部分和实现部分，假设我们将声明部分写在 `language.h` 头文件中，将实现部分写在 `language.cpp` 中。则拆分后的两个文件如下：

`language.h`文件内容如下：


```
#ifndef __LANGUAGE_H__
#define __LANGUAGE_H__

namespace english_env {
    extern const char * mystr;
    void say_hello(const char * name);
}

namespace chinese_env {
    extern const char * mystr;
    void say_hello(const char * name);
}

#endif /* __LANGUAGE_H__ */
```

`language.cpp` 文件内容如下：

```
#include <iostream>
#include "language.h"

namespace english_env {
    const char * mystr = "Hello";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}

namespace chinese_env {
    const char * mystr = "你好";
    void say_hello(const char * name) {
        std::cout << mystr <<" "<< name <<std::endl;
    }
}
```

这样我们将名字空间中变量和函数的声明部分写入到了 `.h` 文件中。接下来我们在任意地方只要包含头文件 `language.h` 就可以使用名字空间中的标识符了。

比如 使用的该名字空间的 `main.cpp` 中只要包含该头文件即可，如下所示：

`main.cpp` 文件内容如下：

```
#include <iostream>

#include "language.h"

int main(int argc, char * argv[]) {
    // 使用名字空间中的变量
    std::cout << english_env::mystr << std::endl;
    std::cout << chinese_env::mystr << std::endl;
```

```
    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -c language.cpp
weimz@mzstudio:~$ g++ -c main.cpp
weimz@mzstudio:~$ g++ -o mynamespace main.o language.o
weimz@mzstudio:~$ ./mynamespace
Hello
你好
```

实验：

复制上述的代码，对其进行修改，然后编译并运行。

第四章、字符串

当我们在 C 语言中记录字符串数据时，通常使用的是字符型的数组，比如:`char mystr[100];`，当我们要记录一个常量字符串时通常使用常字符型的指针来指向常量字符串，比如：`const char * pstr = "laowei";`。在 C 语言中如果将连个字符串拼接成一个更大的字符串就比较麻烦，需要先申请更大的内存空间，然后再使用 `strcpy` 将第一个字符串复制其中，然后再使用 `strcat` 将第二个字符串追加到字符串中。

C++ 的标准库给用户提供了一个 `string` 类型（C++中叫类），它使用动态数组对字符串进行了封装，让字符串的操作更加方便快捷。这个类封装在 `std` 名字空间中。

1. string 类型

C++ 标准库中的 std::string 在头文件 `string` 中声明，用于方便地处理字符串，相比C风格字符串数组，它更安全、易用。

下面我们来举例说明他的用法。

示例:

```cpp
// filename: mystring.cpp
#include <iostream>
#include <string>

int main(int argc, const char * argv[]) {
    std::string s1 = "Hello";
    std::string s2 = "World";

    // 拼接
    std::string s3 = s1 + " " + s2 + "!";  // "Hello World!"

    std::cout << "s1:" << s1 << std::endl;
    std::cout << "s2:" << s2 << std::endl;
    std::cout << "s3:" << s3 << std::endl;

    // 获取长度
    std::cout << "字符串s1的长度:" << s1.length() << std::endl;
    std::cout << "字符串s2的长度:" << s2.length() << std::endl;
    std::cout << "字符串s3的长度:" << s3.length() << std::endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o mystring mystring.cpp
weimz@mzstudio:~$ ./mystring
s1:Hello
s2:World
s3:Hello World!
字符串s1的长度:5
字符串s2的长度:5
字符串s3的长度:12
```

可见 C++ 中的 string 类型在存储字符串时非常的方便灵活。

虽然 C++ 提供了 `std::string` 字符串类型，但如果你不喜欢，你依旧可以使用 C 语言中的字符型数组来存储字符串，因为 C 语言的全部语法和标准库几乎都可以在 C++ 中使用。

实验

使用 `sizeof` 运算符查看上述 `s1`、 `s2`、`s3` 三个变量的占用的内存空间都是都少？为什么？请使用 AI 工具寻找答案（后面讲解类时才能理解）。

2. string 类型的成员函数

在上节课的示例中我们见到 `s1.length()` 可以返回 `s1` 字符串的长度为 `5`。这是字符串类型的变量（对象）的成员函数的用法。

成员函数是某个变量（对象）内部的函数。它可以借助于对应的变量（对象）来调用，并返回调用此成员函数的结果。

字符串常用的成员函数

成员函数	说明
容量相关	
`size` 或 `length`	返回字符串长度，不含尾零`'\0'`。
`max_size`	返回字符串能容纳的最大字符数。
`capacity`	返回为当前字符串分配的存储空间的大小。
`empty`	判断当前字符串是否为空。
修改相关	
`clear`	清空字符串中的内容。
`insert`	在字符串内插入一个或多个字符。
`erase`	删除一个或多个字符。
`push_back`	向字符串末尾加入一个字符。
`append`	向字符串末尾加入一个或多个字符。
`replace`	在指定位置替换字符串。
`copy`	将字符串的部分内容复制到指定的字符数组，并填充尾零`'\n'`。
`swap`	交换两个字符串的内容。
查找相关	
`find`	从字符串**左侧**查找指定的字符或字符串。
`rfind`	从字符串**右侧**查找指定的字符或字符串。
操作相关	
`compare`	比较字符串。
`substr`	返回子字符串。
`c_str`	以 C 语言中的 const char * 指针的形式返回字符串的内容，并以 `\0` 结尾。但不可以通过该指针修改字符串的内容。

其他成员函数

详见参考手册：https://en.cppreference.com/cpp/string/basic_string

示例

```
// filename: str_mem.cpp
#include <string>
#include <iostream>

int main(int argc, const char * argv[]) {
    std::string s1 = "Hello";
    std::string s2 = "c++";

    // 获取容量信息
    std::cout << "s1:" << s1 << std::endl;
    std::cout << "s2:" << s2 << std::endl;
    std::cout << "s1.size():" << s1.size() << std::endl;
    std::cout << "s2.length():" << s2.length() << std::endl;
    std::cout << "s1.max_size():" << s1.max_size() << std::endl;
    std::cout << "s1.capacity():" << s1.capacity() << std::endl;

    // 交换字符串
    s1.swap(s2);
    std::cout << "s1:" << s1 << std::endl;
    std::cout << "s2:" << s2 << std::endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o str_mem str_mem.cpp
weimz@mzstudio:~$ ./str_mem
s1:Hello
s2:c++
s1.size():5
s2.length():3
s1.max_size():9223372036854775806
s1.capacity():15
s1:c++
s2:Hello
```

因为字符串 std::string 是使用动态数组的方式来存储数据，数据存储在动态分配的内存中，因此为了计算方便。对字符串分配的内存空间可能会大于实际存储的字符的长度。如 `s1.size()` 返回 5，加上后面的尾零`'\n'`实际只占用 6 个字节，但实际为 s1 分配的存储空间是 15 个字节。

实验:

查看参考手册，尝试使用字符串其他的成员函数。

3. 字符串相关的运算符

C++ 的 std::string 字符串可以直接使用 `+` 运算符进行拼接操作，也可以使用 `==` 直接比较两个字符串的内容是否相同。这大大方便了字符串的基本运算操作。

以下列出了常用的字符串操作的运算符。

字符串常用的运算符操作

运算符	说明
`+`	用于拼接两个字符串（`string`）或一个字符串（`string`）一个 `char *` 字符串的拼接。
`==`	比较两个字符串是否相等。
`!=`、`<`、`<=`、`>`、`>=`	用于字符串比较，并返回布尔类型（`bool`）的数据。**(C++20标准已经将其删除)。**
`[]`	用于访问字符串中指定位置的字符。
`<<`	用于 `std::cout` 流对象的输出。
`>>`	用于 `std::cin` 流对象的输入。

示例

```cpp
// filename: string_operator.cpp
#include <iostream>

using namespace std;

int main(int argc, char * argv[]) {
    string s1 = "hello";
    string s2 = "world!";
    string s3;
    s3  = s1 + " " + s2; // 字符串拼接

    cout << s3 << endl;
    if (s1 == "hello")
        cout << "s1 字符串的内容是 \"hello\"" << endl;
    else
        cout << "s1 字符串的内容不是 \"hello\"" << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o string_operator string_operator.cpp
weimz@mzstudio:~$ ./string_operator
hello world!
s1 字符串的内容是 "hello"
```

实验

尝试下面的运算符操作是否能够成功？想想为什么？

```
#include <iostream>

using namespace std;

int main(int argc, char * argv[]) {
    // 1. string 字符串和字符串拼接
    string s1 = "ABC";
    string s2 = "123";
    cout << s1 + s2 << endl;
    // 2. string 字符串和 const char *字符串拼接
    cout << s1 + "abc" << endl;
    // 3. const char *字符串和 string 字符串拼接
    cout << "abc" + s1 << endl;
    // 4. const char *字符串和 const char * 字符串拼接
    cout << "abc" + "123" << endl;

    return 0;
}
```

第五章、标准输入输出

虽然 C 语言的 `printf` 和 `scanf` 等函数在 C++ 中依旧可用并且完全兼容，但 C++ 提供了一套自己的标准输入输出方式，以便更好的兼容 C++ 自己的标准库。

C++ 在 `std` 名字空间中提供了 `std::cout` 和 `std::cin` 两个对象（你可以理解成变量）用于负责控制台输出。在使用这两个对象之前需要先包含头文件（`#include<iostream>`）来添加这两个对象的声明。

1. C++的标准输出

C++ 中的 `std::cout` 对象负责将使用 `<<` 运算符交给他的数据输出到控制台终端上，同时有返回 `std::cout` 对象自身供下一个 `<<` 运算符使用。

例如，如下语句则可以在控制台终端打印 `魏明择35岁` 并在末尾换行。

```
std::cout << "魏明择" << 35 << "岁" << std::endl;
```

表达式 `std::cout << "魏明择"` 返回 `std::cout` 对象自身，因此 后面的 `<< 35` 是使用 std::cout 对象进行运算。因此上面的语句可以写成如下四条语句如下：

```
std::cout << "魏明择";
std::cout << 35;
std::cout << "岁";
std::cout << std::endl;
```

结合 `main` 函数则完整示例如下：

```
#include <iostream>

int main(int argc, char * argv[]) {
    std::cout << "魏明择" << 35 << "岁" << std::endl;
    return 0;
}
```

或者写成如下形式

```
#include <iostream>

int main(int argc, char * argv[]) {
    std::cout << "魏明择";
```

```
    std::cout << 35;
    std::cout << "岁";
    std::cout << std::endl;
    return 0;
}
```

两种形式运行结果一样，如下所示:

```
weimz@mzstudio:~$ g++ -o std_cout std_cout.cpp
weimz@mzstudio:~$ ./std_cout
魏明择35岁
```

练习

打印：**我是 xxx，我今年 yy 岁**。

2. C++的标准输入

C++ 语言提供了标准输入对象 `std::cin` 来替代 C 语言中的 `scanf` 进行输入。尽管 `scanf` 在 C++ 语言中依旧可用，但为了风格的同一，建议使用 C++ 推荐的 `std::cin` 来进行输入。

`std::cout` 对象负责从控制台终端读取字符，然后转化成相应的类型，使用 `>>` 运算符交给相应的变量。同时又返回 `std::cin` 对象自身供下一个 `>>` 运算符使用。

示例:

```
#include <iostream>

int main(int argc, char * argv[]) {
    std::string name;  // 创建一个string 对象，name
    int age;

    cout << "请输入姓名:";
    cin >> name;
    cout << "请输入年龄:";
    cin >> age;

    cout << name << "今年" << age << "岁" << endl;

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o std_cin std_cin.cpp
weimz@mzstudio:~$ ./std_cin
请输入姓名:魏明择
请输入年龄:35
魏明择今年35岁
```

第六章、C++的函数

C++ 语言的函数在 C 语言的基础上扩充了一些功能。这些功能包括函数的缺省参数、函数重载等。

1. 函数的缺省参数

在 C 语言中，如果我们要定义函数来实现两个整数、三个整数和四个整数相加并返回这些数的和，一般我们要写三个函数，如下所示：

```
int myadd2(int a, int b) {
    return a + b;
}

int myadd3(int a, int b, int c) {
    return a + b + c;
}

int myadd4(int a, int b, int c, int d) {
    return a + b + c + d;
}
```

在调用时也要根据名字依次调用，如：`myadd2(1,2)` 或 `myadd3(10, 20, 30)` 或 `myadd4(100, 200, 300, 400)`。

C++ 使用缺省参数的语法可以将上述问题简单的解决，这个解决方法就是 C++ 新增的语法：**函数的缺省参数**（default argument）。

函数的缺省参数 允许函数在声明时给定缺省参数，当函数调用没有给定足够的实际参数时，形式参数将使用缺省参数来代替实际参数对函数进行调用。

语法

```
返回类型 函数名/或成员函数名(形参类型 形参名1[ = 缺省值1],形参类型 形参名2[ = 缺省值
2], ...) {
    ...
}
```

示例

```
// filename: default_args.cpp
#include <iostream>

using namespace std;

int myadd(int a, int b, int c=0, int d=0) {
    return a + b + c + d;
}

int main(int argc, char * argv[]) {
    cout << myadd(1, 2) << endl;
    cout << myadd(10, 20, 30) << endl;
    cout << myadd(100, 200, 300, 400) << endl;

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o default_args default_args.cpp
weimz@mzstudio:~$ ./default_args
3
60
1000
```

说明

缺省参数必须自右至左依次存在。如果一个函数的形式参数有缺省参数，则其右侧的所有形式参数都必须有缺省参数。

如下列写法就是错误的。

```
int myadd(int a, int b=100, int c, int d=0) {
    return a + b + c + d;
}
```

因为 形参 `b` 有缺省参数，而 形参 `c` 却没有缺省参数，则编译会报错。

练习

写一个函数 mymul 来实现两个整数、三个整数、四个相乘，并返回相乘后的乘积。

如：

```
int mymul(...) {
    ...
}
```

请完善上述代码。

2. 函数重载

在 C 语言中，同一个 `.c` 文件中不可以用同名的函数或变量，否则会出现标识符冲突的错误。如果在不同的 `.c` 文件中有同名的函数存在，也需要使用 `static` 将多个函数修饰为静态函数，最多保留一个同名的非静态函数。但在 C++ 中则可以允许函数名相同，但函数名的形参个数或者形参类型不同的同名函数存在。

什么是函数重载：

函数重载(function overload)是指C++在一个应用程序内允许有相同的函数名称，但参数列表的个数和类型不同的函数定义（对返回值类型没有要求，不根据返回值进行区别）。

函数调用时会根据实际参数的类型和个数自动匹配相应函数。

区别：

- 参数个数不同
- 参数的类型不同
- 参数的顺序不同。

示例：

```
// filename: func_overload.cpp
#include <iostream>

using namespace std;

int myadd(int a, int b) {
    return a + b;
}

float myadd(float x, float y, float z) {
    return x + y + z;
}

int main(int argc, char * argv[]) {
    cout << myadd(1, 2) << endl;
    cout << myadd(3.4, 5.6, 7.8) << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o func_overload func_overload.cpp
weimz@mzstudio:~$ ./func_overload
3
16.8
```

根据运行结果可以看出，表达式 `myadd(1, 2)` 调用的函数是 `int myadd(int a, int b)`，而 表达式 `myadd(3.4, 5.6, 7.8)` 则调用的函数是 `float myadd(float x, float y, float z)`。

实现原理

C 语言的函数在经过编译阶段他的函数名不会改变，如 C 语言的函数 `int myadd(int a, int b)` 的函数名就一定是 `myadd` 在链接阶段也会根据函数名来匹配对应的函数，则 `static` 修饰后的函数则在编译阶则会隐藏此函数名为不可见。但在 C++ 语言中 任何的函数都会根据函数名和形参的类型在编译阶段重新定义函数名，在调用函数的地方编译器会根据实际参数来匹配对应的函数。

例如在编译后的目标文件中，可以使用 GCC 的 `nm` 命令查看函数的真实名字。 如下所示。

> 重载是编译器将函数名根据参数类型添加名称前缀和后缀来实现的。

以下是上述示例编译后的目标文件中的真实的函数名。

```
weimz@mzstudio:~$ gcc -o func_overload.o -c func_overload.cpp
weimz@mzstudio:~$ nm func_overload.o
                 U _GLOBAL_OFFSET_TABLE_
0000000000000040 T main
0000000000000018 T _Z5myaddfff
0000000000000000 T _Z5myaddii
                 U _ZNSolsEf
                 U _ZNSolsEi
                 U _ZNSolsEPFRSoS_E
                 U _ZSt21ios_base_library_initv
                 U _ZSt4cout
                 U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
```

其中字母 `T` 表示后面的名称是一个函数名（标签）。可见函数 `int myadd(int a, int b)` 编译后的函数名 `_Z5myaddii`， 而函数 `float myadd(float x, float y, float z)` 编译后的函数名为 `_Z5myaddfff`，而 main 函数的函数名依旧是 `main` 不变。

其中 `_Z5myaddii` 后面的 `ii` 表示两个整数型参数，`_Z5myaddfff` 后面的 `fff` 表示三个浮点数参数。

实验

自己尝试使用运算符重载定义函数名相同，参数不同并有缺省参数的多个函数，并尝试调用这些函数。

3. extern "C" 声明

C++ 语言的函数（`main`函数除外）在编译阶段都会被改成编译器能够识别的函数别名，这个函数别名只有生成此目标文件的编译器才能够识别，并且各种编译器为函数取的别名也各不相同。也就是说 C++ 编译出来的目标文件无法交给 C 语言的编译器来链接成为可执行文件。即 C 语言无法调用 C++ 语言编写的程序。解决这个问题的语法是 `extern "C"` 声明。

extern "C" 声明是 C++ 独有的声明（C语言不支持此声明），他的作用是告诉编译器，使用 `extern "C"` 声明的一个或多个函数的名称使用原始的 C 语言的标准名称，不添加前缀和后缀，以便于 C 语言 的程序调用（用来实现 C 和 C++ 语言的混合编程）。

语法有如下两种

声明一个函数的语法：

```
extern "C" 一个函数的声明
```

同时声明多个函数的语法：

```
extern "C" {
    函数的声明1;
    函数的声明2;
    ...
    函数的声明n;
}
```

示例：

```
// filename: extern_c.cpp
#include <iostream>

using namespace std;

// 声明 myadd 为 C 语言函数
extern "C" int myadd(int a, int b);

// 声明 mymul 为 C 语言函数
extern "C" {
    float mymul(float x, float y, float z);
}
```

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

float mymul(float x, float y, float z) {
    return x * y * z;
}

int main(int argc, char * argv[]) {
    cout << myadd(1, 2) << endl;
    cout << mymul(3.4, 5.6, 7.8) << endl;

    return 0;
}
```

链接结果和编译运行如下所示：

```
weimz@mzstudio:~$ gcc -o extern_c.o -c extern_c.cpp
weimz@mzstudio:~$ nm extern_c.o
                 U _GLOBAL_OFFSET_TABLE_
0000000000000040 T main
0000000000000000 T myadd
0000000000000018 T mymul
                 U _ZNSolsEf
                 U _ZNSolsEi
                 U _ZNSolsEPFRSoS_E
                 U _ZSt21ios_base_library_initv
                 U _ZSt4cout
                 U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
weimz@mzstudio:~$ g++ -o extern_c extern_c.o
weimz@mzstudio:~$ ./extern_c
3
148.512
```

可见 `myadd` 和 `mymul` 两个函数都编译成了 C 语言的标准函数。

第七章、引用

在 C 语言中我们经常使用指针用来记录一个变量的地址，然后使用解引用运算符（`*`）去访问或改变他指向的变量。使用指针这样的方法非常灵活且高效，但代码的可读性差，且可能出现无效的野指针，导致系统崩溃。

引用 是 C++ 引入一个新的语法，使用**引用**可以有效替代指针且使用非常方便。

1. 引用

引用（refrence）是为一个变量取一个别名，这个引用绑定原有的变量，使用这个别名等同于使用原有的变量。引用是指针的另一种用法。

声明引用变量的语法

```
变量类型 &变量名1=值,  &变量名2=值, ...;
```

说明

1. 引用型变量必须在创建时进行初始化。
2. 引用型变量在创建后无法改变绑定关系。
3. 引用型变量的类型必须和绑定的变量类型完全一致。

示例

```cpp
// filename: refrence.cpp
#include <iostream>

using namespace std;

int main(int argc, char * argv[]) {
    int x = 100;
    int &rx = x;  // rx为引用类型的变量，绑定x;
    cout << "x:" << x << " rx:" << rx << endl;
    x = 200;
    cout << "x:" << x << " rx:" << rx << endl;
    rx = 300;
    cout << "x:" << x << " rx:" << rx << endl;
```

```
    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o refrence refrence.cpp
weimz@mzstudio:~$ ./refrence
x:100 rx:100
x:200 rx:200
x:300 rx:300
```

上述示例中 `rx` 是变量 `x` 的引用，可以通过 `rx` 直接访问 `x`。

2. 引用的函数传递

引用可以作为函数的形式参数，在函数内部可以使用引用的形参变量访问实参。可以避免指针解引用的麻烦，并且使程序的可读性变得更好。

对于函数的实参比较庞大，比如一个结构体，如果以**传值**的方式传递到函数内部会进行内容复制，传递效率比较低；在这种情况下，可以考虑**传址**或**传引用**。

传引用示例

```
// filename: ref_args.cpp
#include <iostream>

using namespace std;

void myswap(int &pa, int &pb) {
    cout << "void myswap(int &pa, int &pb)" << endl;
    int temp = pa;
    pa = pb;
    pb = temp;
}
void myswap(float &pa, float &pb) {
    cout << "void myswap(float &pa, float &pb)" << endl;
    float temp = pa;
    pa = pb;
    pb = temp;
}

int main(int argc, char * argv[]) {
    int x = 100, y = 200;
    float a = 3.14, b = 2.71828;
    myswap(x, y);
    myswap(a, b);
    cout << "x:" << x << " y:" << y << endl;
```



```
    cout << "a:" << a << " b:" << b << endl;

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o ref_args ref_args.cpp
weimz@mzstudio:~$ ./ref_args
void myswap(int &pa, int &pb)
void myswap(float &pa, float &pb)
x:200 y:100
a:2.71828 b:3.14
```

可见两个 `myswap` 函数 都可以实现交换变量值的目的，且在函数内部通过引用的形参变量直接修改实参的变量还是比较方便的。

实验：

能否定义一个整型的引用 `rn` 去引用一个常数 `100`，如：`int & rn = 100;`。这样编译器会报错。报错的原因是引用 `rn` 和 `100` 的类型不匹配。请问如何修改才能让 `rn` 变量绑定常数 `100`（请借助于人工智能解决此问题）。

3. 常引用

在 C++ 语言中，可以使用 **const** 类型说明符来创建**常引用**。常引用可以绑定普通变量，也可以绑定常变量，甚至可以绑定常量或临时值（也是右值，后面才讲）。

语法

```
const 类型 & 常引用名=变量或常量;
```

说明

- 普通引用只能绑定变量，不能绑定常变量、常量和临时值。
- 常引用 可以绑定变量、常变量、常量、临时值。

正确用法

```
double e = 2.71828;  // e是变量
const double pi = 3.14159; // pi 是常变量
```

```
// 以下是合法的引用
const double &re = e;  // 引用变量
const double &rpi = pi;  // 引用常变量
const double &rc = 123.456;  // 引用常量
const double &rx = 1.2 + 3.4;  // 引用临时值（右值）
```

错误用法

```
const double pi = 3.14159; // pi 是常变量

double &rpi = pi;  // 错误的引用了常变量
double &rc = 123.456;  // 错误的引用了常量
double &rx = 1.2 + 3.4;  // 错误的引用了临时值
```

赋值语句不能通过常引用改变被引用变量的值。但变量可以通过赋值语句改变自身的值，一旦变量改变，常引用的内容也会被改变。

例如

```
double e = 2.71828;  // e是变量
const double &re = e;  // 引用变量

cout << "e:" << e << " re:" << re << endl;
e = 123.456; // 合法且e 和 re 的值都会改变
cout << "e:" << e << " re:" << re << endl;
// 以下是不合法的操作，不能通过常变量修改e的值
// re = 654.321; // 合法且e 和 re 的值都会改变
```

常引用用于函数的参数

当我们编写 C++ 的函数时，如果函数内部不通过该形参去修改实参（即函数内部对其只读），则可以使用常引用的方式传递实参的内容。此种方法尤其对占用内存较大的实参数据尤为有效，且有利于编译器优化。

示例

```
// filename: const_refrence.cpp
#include <iostream>

using namespace std;

// 传递常引用
double myadd1(const double &x, const double &y) {
    return x + y;
```

```
}
// 传递引用
double myadd2(double &x, double &y) {
    return x + y;
}

int main(int argc, char * argv[]) {
    double a = 1.2;
    double b = 3.4;

    cout << myadd1(a, b)<< endl;
    cout << myadd1(5.6, 7.8)<< endl;

    cout << myadd2(a, b)<< endl;
    // 以下一行会报错,常量不能赋值给普通引用
    // cout << myadd2(5.6, 7.8)<< endl;

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o const_refrence const_refrence.cpp
weimz@mzstudio:~$ ./const_refrence
4.6
13.4
4.6
```

实验

尝试使用常引用。

4. auto 自动类型推断

在 C++11 的标准中，原有的 C 语言的关键字 `auto` 已经放弃了自动存储期的变量的含义，该关键字重新定义为自动类型推断的关键字。

在 C++11 标准中 `auto` 关键字是类型占位符，用于自动推断类型。当书写的类型名称比较复杂且经常变化时，可以使用 `auto` 代表其类型，该类型会根据此变量的初始值的类型自动推断得出。

语法

```
auto 变量名 = 表达式;
```

说明

- auto 关键字定义的变量必须赋初始值。
- auto 类型的变量会自动丢弃 **const** 属性和 **引用** 属性，如果需要此属性则需要手动添加。
- C++11/14/17 标准中，不允许 `auto` 关键字来声明函数形参的数据类型。

示例

```
#include <iostream>

using namespace std;

int main(int argc, char * argv[]) {
    const int x = 100;
    const int &rx = x;

    auto pi = 3.14;  // auto 推断为 double 类型
    auto i = 200;  // auto 推断为int 类型
    auto b = true;  // b 的类型是 bool
    auto str = "hello";  // str 的类型是 const char*
    auto ax = x;  // auto 推断为int类型，丢弃const属性
    auto ay = rx; // auto 推断为int类型，丢弃const和引用属性
    const auto cx = rx;  // auto 推断为int类型，cx 为 const int 类型
    const auto &crx = rx;  // crx 为 const int &类型

    return 0;
}
```

实验

尝试使用 auto 关键字来定义数据类型。

第八章、类和对象

本章我们来学习 C++ 中的结构体、类以及他们常见的变量（对象）。

1. C++ 结构体

C++ 的结构体在 C 语言结构体的基础上进行的扩展，其内部可以定义函数，这些函数称之为成员函数。

语法

```
struct 结构体名 {
    类型1 成员变量名1;
    类型2 成员变量名2;
    ...
    返回类型1 成员函数名1(参数列表1) { }
    返回类型2 成员函数名2(参数列表1) { }
};
```

说明

1. 结构体内的成员函数在调用时，优先访问内部的局部变量，然后是结构体内的成员变量，再然后才是全局变量。
2. 成员函数的定义方式和全局函数的定义方式相同，也支持缺省参数、函数重载和内联。
3. 在 C++ 语言中，使用结构体定义变量时可以省略 `struct`关键字。即 C 语言中定义结构体类型的变量必须写成 `struct student s1;` 则在 C++ 语言中则可以写成 `student s1`。

成员函数的调用语法

C++ 的成员函数要借助于该类型的变量（对象）来调用，语法如下：

```
变量.成员函数名(调用实参列表)
```

示例

```
// filename: cpp_struct.cpp
#include <iostream>

using namespace std;
```

```
struct student {
    std::string name;
    int age;
    void set_age(int new_age) {
        age = new_age;
    }
    void show_info(void) {
        cout << "姓名:" << name << ",年龄:" << age << endl;
    }
};

int main(int argc, char * argv[]) {
    student s1 = {"张三", 20}, s2 = {"李四", 16};

    s1.show_info();
    s1.set_age(21);
    s1.show_info();

    s2.show_info();

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o cpp_struct cpp_struct.cpp
weimz@mzstudio:~$ ./cpp_struct
姓名:张三,年龄:20
姓名:张三,年龄:21
姓名:李四,年龄:16
```

练习

1. 写一个 C++ 的结构体 `struct school`，用于描述学校的信息，包含学校的名称，学校的学生数，以及学校学生的信息（数组）。学生的信息可以是另外一个结构体类型如 `struct student` 描述。学校有几个成员函数如下：
   - 添加一个新学生 `void add_student(void){...}`。
   - 根据名字删除一个学生 `bool del_student(std::string name){...}`。成功返回 true，失败返回 false。
   - 列出所有学生的信息 `void list_all(void) {...}`

2. 在主函数中创建两个学校类型的变量，然后对这两个学校的数据进行操作。

2. 类的概念

C 语言中已经定义好的基础数据类型有：

- 整数
  - `char`、`short int`、`int`、`long int`、`long long int`;
- 小数
  - `float`、`double`、`long double`。

标准 C++ 中又增添了一个布尔数据类型（`bool`）。

以上数据类型都定义在编译器内，他的操作行为不可以被修改。

在 C 语言中如果要定义复杂的组合的数据类型可以使用结构体（`struct`）和类（`class`）来定义新的数据类型。本节课先学习结构体的扩展用法。

在 C 语言中，上述这些称之为**数据类型**，在 C++ 中叫做**类**；C 语言中上述类型创建**变量**在 C++ 语言中则称之为**对象**。

类(class)和对象(object) 示意

```
int 是类         100、0、-2 是真实存在的对象

      / ------ 100
int  +  ------  0
      \ ------ -2

float 是类        3.14、2.71 是对象
```

结构体类型定义类和对象

```cpp
// 自定义类型 Car
struct Car{
    string brand; // 品牌
    string number; // 车牌号
    string color;  // 颜色
    void run(float speed) {}
};

// 创建 Car 类型的对象car1, car2
Car car1, car2;
```

C++ 中给出了类和对象的概念是为了让程序编写者能够更好地将代码和现实逻辑结合起来。比如，狗就是现实中的一个类，而你家的狗和邻居家的小狗都是具体的对象。

为了更好的模拟现实，C++ 的**结构体**和后面要讲的**类**都提供了**成员变量**和**成员函数**的概念。

现实世界中，每个对象都有很多的**属性**和**行为**特征。**属性**通常是用于描述一种类型中每个对象自己的数据。如每个人都有姓名（name）、性别（gender）、年龄（age）、身高（height）等，但通常各个对象的属性又大不相同。**行为**代表每种类型的对象共同的功能或动作，通常用成员函数来表示。比如：吃（eat）、走（walk）、学习（study）等。

C++类中使用成员变量来描述类的属性，使用成员函数来描述该类对象的行为。

- **属性：**用于保存一个对象的数据。C++ 用成员变量来存储。
- **行为：**用于描述一个对象的动作；C++ 用成员函数来实现。

以下定义一个人类（Human）的属性和行为。

```
struct Human{
    string name; // 姓名
    string gender; // 性别
    int age; // 年龄
    float height; // 身高
    void eat (string food) {};  // 吃
    void walk(float km) {}  // 走路
    int study(string subject){} // 学习
};
```

上述结构体就定义了人（Human）这个类的**属性**为`name`,`gender`、`age`、`height`，并且此类创建的每个对象（变量）都有不同的属性值。而人类相应的**行为**有`eat`、`walk`、`study`，他们都是函数，是定义在结构体内部的函数。这写函数只能由 Human 创建的对象来使用。如：

```
Human zhangsan = {"张三", "男", 18, 1.80};
zhangsan.eat("烤鸭");
zhangsan.walk(5);
zhangsan.study("C++");
```

实验

尝试完善上述代码，创建张三、李四两个对象，然后访问各自的成员变量，调用各自的成员函数。

3. 类 class

C++ 语言的**类（class）** 本质就是结构体，他用专有的关键字 `class` 声明称为类。类与结构体不同之处在于默认类内的成员函数和成员变量都是私有的（`private`），如果需要指定为特殊的访问权限，需要使用访问说明符 `public`、`protected`或 `private` 类指定访问权限。

C++ 类的作用同结构体一样，用来创建一个自定义的类型。

声明类语法

```
class 类名 {
    //访问说明符
    public:
        // 此处定义成员变量或成员函数。
    protected:
        // 此处定义成员变量或成员函数。
    private:
        // 此处定义成员变量或成员函数。
};
```

说明：

- 类名必须是一个标识符。按编码规范，类名都使用单词首字母大写的标识符（即大驼峰命名法）
- 类的定义的语法后必须加分号（`;`）（和结构体相同）。
- **访问说明符**用户控制类内成员的访问权限。有三种：
  - `public` 表示类内成员函数和非成员函数都可以访问。
  - `protected` 表示类内成员函数和子类成员函数可以访问，非成员函数不可以访问。
  - `private` 表示类内成员函数可以访问，子类成员函数和非成员函数不可以访问。
- 成员函数可以定义在类内和类的外部。
- 类和C++结构体的区别
  - 类的成员（变量或函数）默认的访问权限是私有（`private`）。
  - 结构体的成员（变量或函数）默认的访问权限是共有（`public`）。

成员变量

同 C 语言的结构体成员变量相同。

成员函数:

是定义在类内的函数，名字空间属于类，需要使用该类的对象来调用。

成员函数的调用语法

```
对象.成员函数名(实际参数列表...)
```

说明

- 成员函数同 C 语言的函数一样，可以有参数和返回值。
- 成员函数的调用是一个表达式（同C语言）。
- 按习惯，C++语言的成员函数一般使用驼峰命名法（建议使用小驼峰）。

示例

定义一个矩形类 `Rectangle`（即：长方形类）；此矩形有两个私有的成员变量`width`和 `height`用来保存此矩形的长和宽。此矩形有个三个成员函数，其中 `set_width_height` 用来修改句型的长和宽，`get_area` 用来获取此矩形的面积，`get_length` 用来获取矩形的周长。代码如下：

```cpp
// filename: cpp_class.cpp
#include <iostream>

using namespace std;

class Rectangle {
    public:
        void set_width_height(int w, int h) {
            width = w, height = h;
        }
        int get_area(void) {
            return width * height;
        }
        int get_length(void) {
            return  (width + height) * 2;
        }
    private:
        int width;
        int height;
};

int main(int argc, char * argv[]) {
    Rectangle r1;

    r1.set_width_height(10, 2);
    cout << "r1的周长:" << r1.get_length() << endl;
    cout << "r1的面积:" << r1.get_area() << endl;
    // cout << get_length() << endl; // 报错，get_length 是成员函数，不是函数。
    // cout << r1.width << endl; // 报错，main 函数不能访问私有成员变量。

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o cpp_class cpp_class.cpp
weimz@mzstudio:~$ ./cpp_class
r1的周长:24
r1的面积:20
```

以上是使用 C++ 的类来封装 Rectangle 类，在 C 语言中也同样可以使用结构体来实现，不同之处在于 C 语言的结构不能有成员函数，需要将成员函数改写成全局函数的方式来实现。改写后的代码如下：

```
// filename: c_struct.cpp
#include <iostream>

using namespace std;

struct rectangle {
    int width;
    int height;
};
void set_width_height(struct rectangle * r, int w, int h) {
    r->width = w, r->height = h;
}
int get_area(struct rectangle * r) {
    return r->width * r->height;
}
int get_length(struct rectangle * r) {
    return (r->width + r->height) * 2;
}

int main(int argc, char * argv[]) {
    struct rectangle r1;
    // r1.set_width_height(10, 2);
    set_width_height(&r1, 10, 2);
    cout << "r1的周长:" << get_length(&r1) << endl;
    cout << "r1的面积:" << get_area(&r1) << endl;
    cout << r1.width << endl;  // 正确，C 语言结构体不限制权限。

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o c_struct c_struct.cpp
weimz@mzstudio:~$ ./c_struct
r1的周长:24
r1的面积:20
10
```

可见在 C 语言中也同样可以使用结构体来实现 C++ 中类的定义，不同之处在于 C 语言的结构体所有的成员变量都是共有访问权限，任何代码都能修改成员变量，容易出现错误漏洞并不利于大型软件的编写。

实验

改写类 `class Rectangle` 中的成员变量 `width` 和 `height` 的访问权限。然后尝试在 `main` 函数中访问这些成员变量，总结 `public`、`protected` 和 `private` 的区别。

4. this 指针

`this` 指针是 C++ 成员函数内，指向调用对象地址的指针。在成员函数内部优先访问局部变量，然后才是成员变量。在成员函数如果存在同名的局部变量和成员变量，此时如果需要确切的访问成员变量，此时需要使用 `this` 指针显示的使用成员变量或成员函数。

作用

`this` 指针通常用来明确指示某个标识符是成员变量或成员函数，不是局部变量或全局变量。

关键字

```
this
```

说明：

`C++` 的每个成员函数都有一个 `this` 指针指向调用此函数的对象（默认成员函数的第一个参数就是 `this` 指针，只不过 C++ 的语法隐藏了 `this` 指针（第一个参数））；

使用 this 指针重新编写 `class Rectangle` 类示例

```
// filename: this_pointer.cpp
#include <iostream>
#include <stdio.h>

using namespace std;

class Rectangle {
    public:
        void set_width_height(int width, int height) {
            printf("this --> %p\n", this);
            // 注意此处形参和成员变量的名字相同，需要使用 this 指针区分。
            this->width = width, this->height = height;
        }
```

```
        int get_area(void) {
            return this->width * this->height;
        }
        int get_length(void) {
            return  (this->width + this->height) * 2;
        }
    private:
        int width;
        int height;
};

int main(int argc, char * argv[]) {
    Rectangle r1;
    printf("&r1 --> %p\n", &r1);

    r1.set_width_height(10, 2);
    cout << "r1的周长:" << r1.get_length() << endl;
    cout << "r1的面积:" << r1.get_area() << endl;

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o this_pointer this_pointer.cpp
weimz@mzstudio:~$ ./this_pointer
&r1 --> 0x7fff59230b80
this --> 0x7fff59230b80
r1的周长:24
r1的面积:20
```

练习

编写代码，尝试使用 this 指针。

5. 构造函数

在 C++ 语言中，任何一个对象都有自己的生命周期。在 C++ 语言中，这个生命周期完全可以自己控制。

C++ 语言中变量的声明周期要经历一下五个阶段：

1. 内存分配（出生）
2. 构造对象（出生后）
3. 使用对象
4. 析构对象（销毁前）
5. 内存释放（销毁）

这五个阶段的顺序是固定不变的，不能颠倒。

在创建对象（变量）方面，C++ 语言和 C 语言是一样的，这个对象可以建立在栈上的局部对象、可以是建立在数据段的全局对象、可以是动态内存分布的堆上的对象，也可以是临时对象。无论是在哪里构建对象，都要完整的经历上述五个阶段，否则可能会带来系统的不稳定。

阶段	状态	说明
内存分配	原始内存（未初始化）	分配一块内存，但内部数据不确定
构造对象	对象诞生	调用**构造函数**，确定成员变量的值。
使用对象	有效状态	成员变量和成员函数可正常调用。
析构对象	对象死亡	调用**析构函数**，释放对象占用的资源（但成员变量占用的内存依旧占用）。
内存释放	销毁	释放成员变量占用的内存块。

上面我们提到了两个函数：**构造函数**和**析构函数**，这是两个特殊的成员函数，这些成员函数由开发人员编写，在创建对象时由系统自动调用。

我们先了解**构造函数**。

构造函数(Constructor Function)的作用是对新创建的对象的成员变量进行初始化。

构造函数通常不是一个函数，而是一组函数。在创建对象分配内存后，对象会根据函数重载规则（根据形参的个数和类型）来调用其中的一个构造函数来初始化这个对象的成员。

构造函数的语法

构造函数的名称一定要和类名完全一致，并且构造函数一定没有返回类型。构造函数通常是共有的访问权限。定义语法如下所示：

```
class 类名:
{
    public:
        类名(形参列表1) : 构造函数的初始化列表 {...}
        类名(形参列表2) : 构造函数的初始化列表 {...}
```

```
        ...
};
```

说明

- 构造函数的函数名必须是类名。
- 默认一个对象在创建时都一定会调用构造函数。
- 每个类内默认都有一个无参的构造函数，这个默认的无参构造函数通常什么都不做。类的实现中如果有定义构造函数，则无参的构造函数失效。
- 构造函数可以重载，默认会根据实参列表调用相应的一个构造函数。
- 构造函数不需要有返回类型和返回值。
- C++的对象先分配内存空间，然后才调用构造函数进行初始化这段内存。
- 成员变量的初始化列表用来显式调用对象成员的构造函数，否则默认会调用成员变量无参的构造函数。

示例

定义一个矩形类 Rectangle，在创建对象时可以无参数的创建对象，也可以有参数的方式来创建对象。

```
// filename: constructor.cpp
#include <iostream>

using namespace std;

class Rectangle {
    public:
        Rectangle() {
            cout << "无参的构造函数被调用" << endl;
            width = 1, height = 1;
        }
        Rectangle(int w, int h) {
            cout << "有参的构造函数被调用，w:" << w << " h:" << h << endl;
            width = w, height = h;
        }
        int get_area(void) {
            return width * height;
        }
        int get_length(void) {
            return  (width + height) * 2;
        }
    private:
        int width;
        int height;
};
```

上面定义了两个构造函数，`Rectangle()` 和 `Rectangle(int w, int h)`，其中 `Rectangle()` 会在创建对象没有实参时调用。而 `Rectangle(int w, int h)` 会在有两个实参时调用。

下面我们来学习如何显式的调用对应的构造函数。

创建对象时调用构造函数的规则

1. 调用无参数的构造函数的语法:

```
类名 对象名;
```

2. 调用有参数的构造函数的语法:

```
类名 对象名(实参1, ...);
```

如:

```
Rectangle r1;  // 调用无参的构造函数，成员变量 width 和 height 都是 1。
Rectangle r2(5, 2);  // 调用两个参数的构造函数，width赋值为 5，height 赋值为 2
```

完整示例代码如下：

```
// filename: constructor.cpp
#include <iostream>

using namespace std;

class Rectangle {
    public:
        Rectangle() {
            cout << "无参的构造函数被调用" << endl;
            width = 1, height = 1;
        }
        Rectangle(int w, int h) {
            cout << "有参的构造函数被调用，w:" << w << " h:" << h << endl;
            width = w, height = h;
        }
        int get_length(void) {
            return  (width + height) * 2;
        }
    private:
        int width;
        int height;
};

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

```
    Rectangle r1;  // 调用无参的构造函数，成员变量 width 和 height 都是 1。
    Rectangle r2(5, 2);  // 调用两个参数的构造函数，width赋值为 5，height 赋值为 2

    cout << "r1的周长:" << r1.get_length() << endl;
    cout << "r2的周长:" << r2.get_length() << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o constructor constructor.cpp
weimz@mzstudio:~$ ./constructor
无参的构造函数被调用
有参的构造函数被调用，w:5 h:2
r1的周长:4
r2的周长:14
```

可见 `Rectangle r1;` 则调用无参的构造函数，`Rectangle r2(5, 2)` 则调用有参的构造函数。

简化构造函数

因为构造函数也可以有缺省参数，因此上述示例中的两个构造函数也可以使用缺省参数的方式将两个构造函数整合成一个构造函数，整合后的代码比较简洁，类的实现代码如下所示。

```
class Rectangle {
    public:
        Rectangle(int w=1, int h=1) {  // 使用缺省参数
            width = w, height = h;
        }
        int get_length(void) {
            return  (width + height) * 2;
        }
    private:
        int width;
        int height;
};
```

这样定义的类创建对象时可以无参构造，也可以用一个或两个参数来构造。创建对象的代码都可以不变。

实验

自己实现一个正方形类 `class Square`，其中只有一个属性为边长，在创建对象时如果没有给定边长则边长为1。然后编写两个获取周长和面积的成员函数。

6. 一个参数的构造函数

在 C++ 语言中，当创建一个对象时并且为其赋值一个初始值，则会调用具有一个参数的构造函数。

如下面的代码

```
Rectangle r3 = 100;
```

则会尝试调用 `Rectangle(const int & xxx)` 这样的构造函数，当然我们的示例中没有一个整数参数的构造函数则创建对象会出错。

示例

下面我们定义一个正方形类 `Square` 类来说明这样一个问题。

```
#include <iostream>

using namespace std;

class Square {
    public:
        Square(int l=1) {
            cout << "构造函数 Square(int) 被调用!" << endl;
            length = l;
        }
        Square(double l) {
            cout << "构造函数 Square(double) 被调用!" << endl;
            length = l;
        }
        double get_length(void) {
            return  length * 2;
        }
    private:
        double length;
};
```

当我们使用 Square 类创建对象时使用不同的创建方法则调用不同的构造函数。如下所示。

```
Square s1;  // 调用无参的构造函数
Square s2(100);  // 调用构造函数Square(int)
Square s3(1.2);  // 调用构造函数Square(double)
Square s4 = 200;  // 调用构造函数Square(int)
Square s5 = 3.4;  // 调用构造函数Square(double)
```

完整的示例代码如下：

```
// filename: constructor1.cpp
#include <iostream>

using namespace std;

class Square {
    public:
        Square(int l=1) {
            cout << "构造函数 Square(int) 被调用!" << endl;
            length = l;
        }
        Square(double l) {
            cout << "构造函数 Square(double) 被调用!" << endl;
            length = l;
        }
        double get_length(void) {
            return  length * 2;
        }
    private:
        double length;
};

int main(int argc, char * argv[]) {
    Square s1;  // 调用无参的构造函数
    Square s2(100);  // 调用构造函数Square(int)
    Square s3(1.2);  // 调用构造函数Square(double)
    Square s4 = 200;  // 调用构造函数Square(int)
    Square s5 = 3.4;  // 调用构造函数Square(double)

    cout << "s1的周长:" << s1.get_length() << endl;
    cout << "s2的周长:" << s2.get_length() << endl;
    cout << "s3的周长:" << s3.get_length() << endl;
    cout << "s4的周长:" << s4.get_length() << endl;
    cout << "s5的周长:" << s5.get_length() << endl;

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o constructor1 constructor1.cpp
weimz@mzstudio:~$ ./constructor1
构造函数 Square(int) 被调用!
构造函数 Square(int) 被调用!
构造函数 Square(double) 被调用!
构造函数 Square(int) 被调用!
构造函数 Square(double) 被调用!
s1的周长:2
s2的周长:200
s3的周长:2.4
```

```
s4的周长:400
s5的周长:6.8
```

实验

验证一个参数的构造函数的编写和调用规则。

7. 内建类型的构造函数

前面我们自己定义了 Rectangle 类 和 Square 类，这些类使我们自己定义的数据类型。在 C++ 语言中，char、int、float、double这些也同样是类，他们和我们自己定义的类是一样的。只是这些类的定义都定义在编译器内，任何开发人员都不能改变这些类的行为和属性。

虽然这些类都是编译器内构建好的，但当我们使用这些类时，这些类和我们用户自定的类一样都要经过分配内存、初始化、使用、析构、释放内存等几个步骤。并且这些类也都有自己的构造函数和析构函数（后面才讲）。

下面举例说明以下几个基础数据类型调用的构造函数。

示例

```
int a;  // 调用无参数的构造函数 `int()`。
int b = 100;  // 调用有一个int参数的构造函数 int(int)
int c = 3.1;  // 调用有一个double参数的构造函数 int(double)

float f1;  // 调用无参数的构造函数 `float()`。
float f2 = 200;  // 调用有一个int参数的构造函数 float(int)
float f3 = 3.2;  // 调用有一个float参数的构造函数 float(float)
```

实验

尝试自定义的类 `class MyInt` 和 `class MyFloat` 代替上述示例中的 `int` 和 `float` 使上述示例依旧能编译通过（先不考虑操作的逻辑，只保证语法正确即可）。

8. 构造函数的初始化列表

在 C++ 语言定义的类中，通常一个类内有多个成员变量。在一个类创建对象时，先分配对象总体占用的内存，然后在调用对象各个成员的构造函数来初始化成员变量自身，然后才是调用该类的构造函数。如下面定义 `Person` 这个类，他有三个成员变量 `name`、`age`。

```
class Person {
    private:
        string name; // 姓名
        int age; // 年龄
};
```

在使用 `Person` 创建对象时，会先调用 `name`、`age` 的构造函数，然后才会调用 `Person` 的构造函数。

例如

```
class Person {
    public:
        Person(const string &n, int a) {
            name = n;
            age = a;
        }
    private:
        string name; // 姓名
        int age; // 年龄
};
```

在使用 `Person` 类创建对象时如下所示：

```
Person p1("魏明择", 35);
```

上述在创建 `p1` 对象时是先调用 无参数的 `string(void)` 构造函数来创建 `name` 成员变量，再调用无参数的 `int(void)` 构造函数来创建 `age` 成员变量，然后才调用 `Person(const string &n, int a)` 构造函数，但是在构造函数内又实用赋值语句将 `name` 和 `age` 的值进行了修改。这样的做法虽然可行，但可能会影响效率。

那如何能在 创建 `name` 和 `age` 时能够直接调用有参的构造函数将形参 `n` 和 `a` 的值直接用来创建成员变量呢？我们可以使用 C++ 语法中的 **构造函数的初始化列表** 来实现这一功能。

语法

在构造函数形参列表的右小括号 （`)`） 和 左大括号（`{`）加一个冒号（`:`），冒号后显式调用成员变量的构造函数。

```
class 类名:
{
    public:
        类名(形参列表1) : 构造函数的初始化列表 {...}
```

```
        类名(形参列表2) : 构造函数的初始化列表 {...}
        ...
};
```

说明

1. 构造函数的初始化列表如果有多个成员变量需要使用逗号（,）分隔。

示例

```
class Person {
    public:
        Person(): name(""), age(1) { }
        Person(const string &n, int a): name(n), age(a) { }
    private:
        string name; // 姓名
        int age; // 年龄
};
```

这样 `name` 和 `age` 会在创建时使用 `n` 和 `a` 来构建自己，之后则不需要修改自己的值。

实验

使用上述 `Person` 类来创建对象，然后添加成员函数并调用成员函数来打印成员变量的值。

9. 析构函数

在 C++ 语言中，每个对象都有自己的生命周期，在释放对象自己占用的内存前一定会调用析构函数来释放该对象占用的资源。

析构函数 (Destructor)

析构函数会在对象销毁前自动调用。如果类内没有定义析构函数，则编译器也会为其添加一个析构函数，默认的析构函数什么都不做。

析构函数的作用是释放对象对象使用的资源。

析构函数的语法

```
class 类名{
    public:
    ~类名(void) {
```

```
        ... 做释放内存或关闭文件等操作。
    }
}
```

说明

- 析构函数只有一个，且不能重载。
- 析构函数的名称必须是 `~类名`。
- 析构函数没有参数和返回值。
- 析构函数会在对象销毁前自动调用。无需手动调用。

> 一般一个类内的成员变量有指针，则需要虚构函数类释放动态分配的内存资源。

示例

实现一个 Person 类用于描述人的信息。由于人的名字有长有短，因此采用动态分配内存的方式来保存人名信息。其中人的名字使用成员变量 `pname` 来保存。此示例中如果没有添加析构函数，则 `pname` 指向的内存不会释放，内存会有丢失。

```
// filename: destructor.cpp
#include <iostream>
#include <string.h>

using namespace std;

class Person {
    public:
        Person(const char * name, int a):age(a)  {
            if (NULL == name) { // 分配一个字节存放尾零
                pname = (char*)malloc(1);
                *pname = '\0';
            } else {
                pname = (char*)malloc(strlen(name)+1);
                strcpy(pname, name);
            }
            cout << "构造函数被调用" << endl;
        }
        ~Person(void) {
            // 释放内存
            free(pname);
            cout << "析构函数被调用" << endl;
        }
    public:
        void printInfo() {
            cout << pname << "今年" << age << "岁\n";

        }
    private:
```

```
        char * pname; // 姓名
        int age; // 年龄
};

int main(int argc, char * argv[]) {
    Person p1("wei", 35);  // 调用无参的构造函数
    p1.printInfo();

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o destructor destructor.cpp
.weimz@mzstudio:~$ ./destructor
构造函数被调用
wei今年35岁
析构函数被调用
```

练习：

尝试使用编写析构函数，尝试使用析构函数的语法。

10. 深拷贝和浅拷贝

浅拷贝 是指对象之间复制时，对应的成员变量依次复制。对于指针类型的变量只进行值复制，指针指向同一地址，指向的内存会共用，数据会互相影响。C++ 对象默认是浅拷贝。

深拷贝 是指在对象复制时，对应的普通成员变量依次复制。对于指针类型的变量指向的内容也会复制，致使各个对象的数据各自独立，互不影响。C++ 需要通过拷贝构造函数（后面会讲）和拷贝赋值函数（后面会讲）类实现。

浅拷贝示例

```
// filename: shallow_copy.cpp
#include <iostream>

using namespace std;

class Rectangle {
    public:
        Rectangle(int w=1, int h=1):width(w), height(h) {
            cout << "有参的构造函数被调用，w:" << w << " h:" << h << endl;
        }
        void showInfo() {
            cout << "width:" << width << " height:" << height << endl;
```



```
        }
    private:
        int width;
        int height;
};

int main(int argc, char * argv[]) {
    Rectangle r1(9, 8);  // 调用有参数的构造函数
    Rectangle r2(r1);  // 浅拷贝（调用默认的拷贝构造）
    Rectangle r3 = r1;  // 浅拷贝（调用默认的拷贝构造）
    Rectangle r4;  // 调用有参数的构造函数

    r4 = r1;    // 浅拷贝（调用默认的拷贝赋值函数）
    r1.showInfo();
    r2.showInfo();
    r3.showInfo();
    r4.showInfo();

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o shallow_copy shallow_copy.cpp
weimz@mzstudio:~$ ./shallow_copy
有参的构造函数被调用，w:9 h:8
有参的构造函数被调用，w:1 h:1
width:9 height:8
width:9 height:8
width:9 height:8
width:9 height:8
```

可见 对象 `r2`、`r3`、`r4`的成员变量宽（`width`）和高（`height`）值都是从 r1 对象两个成员变量赋值而来。

上述创建四个 `Rectangle` 类型的对象，但这里却只调用了两次构造函数，因为 `r2` 和 `r3` 创建时调用的是默认的拷贝构造函数。

带有指针类型成员变量的对象的浅拷贝示例

```
// filename: shallow_copy2.cpp
#include <iostream>
#include <string.h>

using namespace std;

class Person {
    public:
        Person(const char * name, int a):age(a)  {
            if (NULL == name) { // 分配一个字节存放尾零
                pname = (char*)malloc(1);
```

```
                *pname = '\0';
            } else {
                pname = (char*)malloc(strlen(name)+1);
                strcpy(pname, name);
            }
            cout << "构造函数被调用，动态分配了内存!" << endl;
        }
        ~Person(void) {
            // 释放内存
            cout << "析构函数被调用，即将释放内存" << endl;
            free(pname);
        }
    public:
        void printInfo() {
            cout << pname << "今年" << age << "岁\n";

        }
    private:
        char * pname; // 姓名
        int age; // 年龄
};

int main(int argc, char * argv[]) {
    Person p1("老魏", 35);
    Person p2 = p1;
    p1.printInfo();
    p2.printInfo();

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o shallow_copy2 shallow_copy2.cpp
weimz@mzstudio:~$ ./shallow_copy2
构造函数被调用，动态分配了内存!
老魏今年35岁
老魏今年35岁
析构函数被调用，即将释放内存
析构函数被调用，即将释放内存
free(): double free detected in tcache 2
已中止                    （核心已转储） ./shallow_copy2
```

在上述程序结构中可以看出，我们一共创建了两个 `Person` 类型的对象 `p1` 和 `p2`。从构造函数的调用情况来看，创建 `p1` 对象时调用了一次构造函数并只动态分配了一次内存。在创建 `p2` 对象时实际是调用的是默认的拷贝构造函数将 `p1` 成员变量的值依次赋值给 `p2` 的各个成员变量，因此 `p2.pname` 指针也同样指向了 `p1.pname` 指向的动态分配的内存。即保存名字的内存是两个对象共有的内存，并没有进行复制。

两个对象销毁时都会调用析构函数，第一次调用析构函数时会释放这段共享内存，当第二次调用析构函数时则会再次释放已经释放过的内存，因此就会出现程序异常崩溃的情况。

从上述示例可知，在使用 `p1` 对象创建 `p2` 时，`p2.pname` 指针应当指向重新开辟的新内存，避免内存共用，因此需要使用**深拷贝**。

在 C++ 中，实现深拷贝的最好方法是实现**拷贝构造**和**拷贝赋值**这两个特殊的成员函数。

实验

尝试画图理解 **带有指针类型成员变相的浅拷贝示例** 中 `p1` 和 `p2` 两个对象成员变量的值，以及动态分配内存和释放的过程。

11. 拷贝构造函数和拷贝赋值函数

C++ 两个同类型对象的复制时默认是成员变量的值依次赋值，这个由 C++ 编译器为每个类默认添加的两个成员函数：**拷贝构造函数**和**拷贝赋值函数**来决定了。如果需要打破这种默认的复制方法则需要自己编写这两个成员函数来实现自定义的功能。通常用这两个成员函数来控制对象赋值的过程。

拷贝构造函数的语法

```
class 类名 {
    public:
        类名(const 类名 & obj): 构造函数的初始化列表 {  // 拷贝构造函数
            ...
        }
    ...
};
```

说明

1. 拷贝构造函数的参数必须是同类对象的常引用（`const 类名 &`）。
2. 拷贝构造函数会在创建对象时直接用同类对象赋初始值时被自动调用。

以下两种情况才会调用拷贝构造函数

```
Person p1("老魏", 35);
Person p2 = p1;  // 调用拷贝构造函数
Person p3(p1);  // 调用拷贝构造函数
```


拷贝赋值函数的语法

```
class 类名 {
    public:
        类名& operator=(const 类名& obj) {  // 拷贝赋值函数
            ...
        }
    ...
}
```

说明

1. **拷贝赋值函数**的参数必须是同类对象的常引用（`const 类名 &`）。
2. **拷贝赋值函数**的返回值通常是对象自身的引用，类型为：`类名 &`。
3. 拷贝赋值函数会在对象创建后，再使用赋值语句对对象赋值时被调用。
4. 通常一个类内自定义了**拷贝构造函数**则一定要自定义**拷贝赋值函数**。

以下情况才会调用拷贝构造函数

```
Person p1("老魏", 35);
Person p4("魏明择", 1);

p4 = p1;  // 调用拷贝赋值函数
```

示例

以下重写`Person`类，使其能够实现深拷贝，让各个对象的数据各自独立。

```
// filename: copy_constructor.cpp
#include <iostream>
#include <string.h>

using namespace std;

class Person {
    public:
        // 构造函数
        Person(const char * name, int a):age(a)  {
            if (NULL == name) { // 分配一个字节存放尾零
                pname = (char*)malloc(1);
                *pname = '\0';
            } else {
                pname = (char*)malloc(strlen(name)+1);
                strcpy(pname, name);
            }
            cout << "构造函数被调用，动态分配了内存!" << endl;
        }
        // 拷贝构造函数
```

```
        Person(const Person &obj):age(obj.age) {
            this->pname = (char*)malloc(strlen(obj.pname)+1);
            strcpy(this->pname, obj.pname);
            cout << "拷贝构造函数被调用，重新动态分配了新内存!" << endl;
        }
        // 拷贝赋值函数
        Person& operator=(const Person &obj) {
            free(this->pname);
            this->pname = (char*)malloc(strlen(obj.pname)+1);
            strcpy(this->pname, obj.pname);
            this->age = obj.age;
            return *this;
        }
        // 析构函数
        ~Person(void) {
            // 释放内存
            cout << "析构函数被调用，即将释放内存" << endl;
            free(pname);
        }
    public:
        void printInfo() {
            cout << pname << "今年" << age << "岁\n";

        }
    private:
        char * pname; // 姓名
        int age; // 年龄
};

int main(int argc, char * argv[]) {
    Person p1("老魏", 35);
    Person p2 = p1;
    p1.printInfo();
    p2.printInfo();

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o copy_constructor copy_constructor.cpp
weimz@mzstudio:~$ ./copy_constructor
构造函数被调用，动态分配了内存!
拷贝构造函数被调用，重新动态分配了新内存!
老魏今年35岁
老魏今年35岁
析构函数被调用，即将释放内存
析构函数被调用，即将释放内存
```

实验

自定义一个使用动态分配内存来存储数据的类(类内含有指针成员变量），然后添加拷贝构造函数和拷贝赋值函数来实现深拷贝，实现对象的自动复制。

12. 标准 C++ 类总结

1. C 结构体和 C++ 结构体的区别

C 语言的结构体内部只能定义成员变量，不能定义成员函数，如果需要操作成员变量的值，需用定义全局函数来实现。而 C++ 的结构体则可以定义成员函数，使用成员函数可以操作成员变量。使用成员函数代替全局函数更有利于函数的归类，防止全局函数太多而混乱。

2. C++ 结构体 和 类 的区别

C++ 结构体和 C++ 的类是一样的功能，他的内部可以定义成员变量和成员函数，也可以重新定义和替代默认的构造函数和析构函数，不同之处在于 C++ 结构体内默认所有的成员变量和成员函数都是共有属性（`public`），而类内默认所有的成员变量和成员函数都是私有属性（`private`）。

3. 标准 C++ 结构体和 类的默认四个成员函数

1. **无参的构造函数：**在对象的内存分配后调用。默认的构造函数什么都不做，如果需要自定义初始化类内的成员变量，则需要用自定义的构造函数类替代默认的构造函数。
2. **析构函数：**在对象释放自己的内存前自动调用，通常用来释放对象占用的资源。默认的析构函数什么都不做。
3. **拷贝构造函数：**在对象的内存分配后，同时使用一个同类型对象进行初始化时自动调用。默认的拷贝构造函数执行浅拷贝，如果需要自定义复制过程则需要自定义此拷贝构造函数来替代默认的拷贝构造函数。
4. **拷贝赋值函数：**在创建对象已经创建后，在重新使用同类型的对象对此对象重新赋值时调用。默认的拷贝赋值函数执行浅拷贝，如果需要自定义赋值时的复制过程则需要自定义此拷贝赋值函数来替代默认的拷贝赋值函数。

第九章、类和对象（高级）

1. 全局变量（对象）的构造和析构函数调用规则

全局变量（对象）存在于数据段，而数据段是在 `main` 函数调用之前创建，因此全局变量（对象）会在 `main` 函数开始前创建，在 `main` 函数结束后销毁。因此全局变量（对象）的构造函数会先用 `main` 开始前调用。同样析构函数也会在 `main` 函数结束后调用。

示例

```
// filename: global_object.cpp
#include <iostream>

using namespace std;

class Square {
    public:
        Square(double l=1) {
            cout << "边长为" << l << "的正方形被创建!" << endl;
            length = l;
        }
        ~Square() {
            cout << "边长为" << length << "的正方形被销毁!" << endl;
        }
    private:
        double length;
};

Square s1(10);
Square s2(20);

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

    Square s3(3);
    Square s4(4);
    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o global_object global_object.cpp
weimz@mzstudio:~$ ./global_object
边长为10的正方形被创建!
边长为20的正方形被创建!
边长为3的正方形被创建!
边长为4的正方形被创建!
```

```
边长为4的正方形被销毁!
边长为3的正方形被销毁!
边长为20的正方形被销毁!
边长为10的正方形被销毁!
```

从上述运行结果中可以看出，全局对象 `s1` 和 `s2` 先创建，然后是 `main` 函数内的 `s3` 和 `s4` 创建。对象销毁的顺序和对象创建的顺序相反。先销毁 `main` 函数内的 `s4` 和 `s3`，然后是全局对象 `s2` 和 `s1`。

实验：

完成上述示例的编写和运行，体会对象创建和销毁的顺序。

2. 堆上动态创建对象

在 C 语言中，如果动态在堆上申请内存存放数据需要使用 `malloc` 和 `free` 一系列函数。

C++ 语言建议使用 `new` 和 `delete` 关键字在堆上创建对象用于代替 C 语言的 `malloc` 系列函数来动态创建对象。

关键字 `new` 用来在堆上动态创建对象。使用 `new` 创建对象时先调用 `malloc` 函数申请内存，然后调用相应的构造函数来初始化对象。

关键字 `delete` 用来销毁在堆上创建的对象，使用 `delete` 销毁对象时先调用析构函数进行反向初始化，然后再调用 `free` 函数来销毁对象占用的内存。

`new` 创建对象的语法

```
// 创建单个对象
new 类名;  // 调用无参数的构造函数
new 类名(实参列表); // 根据重载规则调用有参数的构造函数

// 创建对象数组
new 类名[对象个数(整数)]
// 创建对象数组，并指定构造函数
new 类名[对象个数(整数)]{每个对象的初始化构造函数}
```

`delete` 销毁对象的语法

```
// 销毁单个对象
delete 对象地址
```

```
// 销毁对象数组
delete [] 对象地址
```

示例

此示例示意使用 `new` 和 `delete` 来动态申请单个对象和动态申请对象的数组。

```
// filename: new_delete.cpp
#include <iostream>

using namespace std;

class Square {
    public:
        Square(double l=1) {
            cout << "边长为" << l << "的正方形被创建!" << endl;
            length = l;
        }
        ~Square() {
            cout << "边长为" << length << "的正方形被销毁!" << endl;
        }
        double get_length(void) {
            return  length * 2;
        }
    private:
        double length;
};

int main(int argc, char * argv[]) {
    Square * ps = NULL;

    cout << "动态分配一个对象" << endl;
    ps = new Square(2);  // 创建一个 Square 对象
    cout << ps->get_length() << endl;
    delete ps;  // 销毁这个 Square 对象

    // 创建三个 Square 对象
    cout << "动态分配多个对象" << endl;
    ps = new Square[3]{Square(3), Square(4), Square(5)};
    for (int i = 0; i < 3; i++) {
        ps[i].get_length();
    }
    delete [] ps;
    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o new_delete new_delete.cpp
weimz@mzstudio:~$ ./new_delete
动态分配一个对象
边长为2的正方形被创建!
```

```
4
边长为2的正方形被销毁!
动态分配多个对象
边长为3的正方形被创建!
边长为4的正方形被创建!
边长为5的正方形被创建!
边长为5的正方形被销毁!
边长为4的正方形被销毁!
边长为3的正方形被销毁!
```

实验

使用 `new` 动态创建 10 个整数类型的对象数组。使用后再使用 `delete` 进行销毁。

3. 成员函数的类内声明和类外实现

在 C++ 语言中，在编写代码时通常将类（和结构）拆分成声明和实现两部分。声明部分通常写在 `.h` 文件中，内部只写类的成员声明，供类的实现成员函数和使用此类创建对象时使用；而成员函数的实现部分（和静态成员变量的定义和初始化，后面才讲）通常写在 `.cpp` 文件中，通常实现部分会单独编译和链接。

以下面代码为例，此 `main.cpp` 文件中类 `Square` 的声明和实现都实现在一起，并且和使用此类的 `main` 函数写在同一个文件中。这样不利于模块化编程，也不利于类的重复利用。

```cpp
// filename: main.cpp
#include <iostream>

using namespace std;

class Square {
    public:
        Square(double l=1) : length(l) {
            cout << "边长为" << l << "的正方形被创建!" << endl;
        }
        ~Square() {
            cout << "边长为" << length << "的正方形被销毁!" << endl;
        }
    private:
        double length;
};

int main(int argc, char * argv[]) {
    Square s1(3);

    return 0;
}
```

下面将上述的类 `Square` 拆成声明放在文件 `square.h` 中，将类 `Square` 的实现部分放在文件 `square.cpp` 中。将 `main` 函数放在 `main.cpp` 中。拆分结构如下:

文件：`square.h`

```
#ifndef __SQUARE_H__
#define __SQUARE_H__

class Square {
    public:
        Square(double l=1);
        ~Square();
    private:
        double length;
};

#endif  /* __SQUARE_H__ */
```

文件：`square.cpp`

```
#include <iostream>

#include "square.h"

using namespace std;

Square::Square(double l) : length(l) {
    cout << "边长为" << l << "的正方形被创建!" << endl;
}

Square::~Square() {
    cout << "边长为" << length << "的正方形被销毁!" << endl;
}
```

成员在类外实现时，需要使用 作用域限定符`::` 来说明当前的函数属于某个类的成员函数。如果成员函数有缺省参数，则只能在声明时说明缺省参数的值，而在现实部分则不用给出缺省参数。

构造函数的初始化列表需要写在成员函数的实现部分。

文件：`main.h`

```
#include "square.h"

int main(int argc, char * argv[]) {
    Square s1(3);

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o square.o -c square.cpp
weimz@mzstudio:~$ g++ -o main.o -c main.cpp
weimz@mzstudio:~$ g++ -o main main.o square.o
weimz@mzstudio:~$ ./main
边长为3的正方形被创建!
边长为3的正方形被销毁!
```

实验

将你之前写过的类拆解成类内声明和类外实现。

4. 临时对象（匿名对象）

临时对象（又叫匿名对象）是指直接由构造函数创建的对象。他通常用于表达式中或函数传参等临时创建一个对象进行操作。

临时对象的生命周期短，通常在表达式结束后就会被销毁。

示例

```cpp
// filename: temp_obj.cpp
#include <iostream>

using namespace std;

class Square {
    public:
        Square(double l=1) {
            cout << "边长为" << l << "的正方形被创建!" << endl;
            length = l;
        }
        ~Square() {
            cout << "边长为" << length << "的正方形被销毁!" << endl;
        }
        double get_length(void) {
            return  length * 2;
        }
    private:
        double length;
};

int main(int argc, char * argv[]) {
    // 创建临时对象
    Square(55);
    // 创建临时对象并调用成员函数
    cout << Square(6).get_length() << endl;
```

```
    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o temp_obj temp_obj.cpp
weimz@mzstudio:~$ ./temp_obj
边长为55的正方形被创建!
边长为55的正方形被销毁!
边长为6的正方形被创建!
12
边长为6的正方形被销毁!
```

从构造函数和析构函数的调用过程来看，临时对象在创建此对象的表达式执行完毕后临时对象就会被销毁。

5. 静态成员变量

静态成员变量 是类本身的变量，他属于类，不属于类的创建的对象。

静态成员变量与普通变量的区别：

1. 普通成员变量：该类创建的对象各自用于一份副本，没有对象都有自己的同名成员变量。
2. 静态成员变量：全局只有一个副本，该类创建的对象都共用这个变量。

静态成员变量 需要在类内声明，在类外单独定义，因此要使用静态成员变量需要在类的声明部分进行声明，同时在类外实现部分的代码处进行定义和初始化。

静态成员变量声明的语法:

```
class 类名{
    访问说明符:
        static 变量类型 变量名;
};
```

静态成员变量定义的语法:

静态成员变量 只有在定义时才会单独为其分配内存，并赋予初始值（如果有初始值）。

```
变量类型 类名::变量名[ = 初始值];
```

静态成员变量的使用方法

```
类名::变量名
// 或
对象名.变量名
```

示例

记录一个类创建对象的个数;

```
// filename: static_member_var
#include <iostream>
#include <string>

using namespace std;

class Person {
    public:
        static int total_count; // 声明静态成员变量
        Person(const string & n, int a): name(n), age(a)  {
            cout << "创建: Person(" << n << ", " << a << ");\n";
            // 成员函数内使用静态成员变量
            total_count++;
        }
        ~Person(void) {
            cout << "析构: ~Person(" << name << ", " << age << ");\n";
            total_count--;
        }
    private:
        string name; // 姓名
        int age; // 年龄
};

// 定义静态成员变量
int Person::total_count = 0;

int main(int argc, char * argv[]) {
    Person p1("老魏", 35);
    Person p2("魏明择", 35);
    // 两种使用静态成员变量的方法
    cout << "Person对象的个数:" << Person::total_count << endl;
    cout << "Person对象的个数:" << p2.total_count << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o static_member_var static_member_var.cpp
weimz@mzstudio:~$ ./static_member_var
```

```
创建: Person(老魏, 35);
创建: Person(魏明择, 35);
Person对象的个数:2
Person对象的个数:2
析构: ~Person(魏明择, 35);
析构: ~Person(老魏, 35);
```

从上述运行结果可知，静态成员变量全局只有一个，每个对象的成员函数（包括构造函数和析构函数）都可以修改和访问这个成员变量。如果该成员变量是公有访问权限，则其他函数也可以访问这个静态成员变量。

静态成员变量通常用于记录类全局的信息，而不是对象的信息。

6. 静态成员函数

静态成员函数 是属于类的函数，他只能访问类的静态成员变量，不能访问对象的成员变量。

说明

1. 静态成员函数没有 `this` 指针，不能访问对象的成员变量。
2. 静态成员函数的作用域属于类。

静态成员函数的类内声明，类内实现的语法

```
class 类名{
   public:
      // 类内声明，类内实现。
      static 返回类型 函数名1(形式参数列表) { ... };
}
```

静态成员函数的类内声明，类外实现的语法

```
class 类名{
   public:
      // 类内声明。
      static 返回类型 函数名2(形式参数列表);
}
```

类外实现的语法

```
返回类型 类名::函数名2(形式参数列表) { ... }
```

调用的语法

```
// 借助于类名来调用
类名::函数名2(实际参数列表)
// 借助于该类的对象来调用
对象.函数名2(实际参数列表)
```

示例

```
// filename: static_member_func.c
#include <iostream>
#include <string>

using namespace std;

class Person {
    public:
        static int total_count; // 声明静态成员变量
        Person(const string & n, int a): name(n), age(a)  {
            cout << "创建: Person(" << n << ", " << a << ");\n";
            // 成员函数内使用静态成员变量
            total_count++;
        }
        ~Person(void) {
            cout << "析构: ~Person(" << name << ", " << age << ");\n";
            total_count--;
        }
        // 静态成员函数
        static int getTotalCount(void); // 类内声明
    private:
        string name; // 姓名
        int age; // 年龄
};

// 定义静态成员变量
int Person::total_count = 0;

// 静态成员函数的类外实现
int Person::getTotalCount(void) {
    return total_count;
}

int main(int argc, char * argv[]) {
    Person p1("老魏", 35);
    Person p2("魏明择", 35);
    // 调用使用静态成员变量的方法
    cout << "Person对象的个数:" << Person::getTotalCount() << endl;
    cout << "Person对象的个数:" << p2.getTotalCount() << endl;

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o static_member_func static_member_func.cpp
weimz@mzstudio:~$ ./static_member_func
创建：Person(老魏, 35);
创建：Person(魏明择, 35);
Person对象的个数:2
Person对象的个数:2
析构：~Person(魏明择, 35);
析构：~Person(老魏, 35);
```

7. 常成员函数

常成员函数 是使用**常声明（const）**方法声明一个普通的成员函数只能读取其对象的成员变量，但不能修改其成员变量。一旦在函数内修改成员变量，则编译器报错。

常成员函数内的 this 指针被设定为只读（添加了 const 属性），因此无法修改其成员变量。

常成员函数的使用场景：

1. 没有常属性的对象可以调用普通成员函数，也可以调用常成员函数。
2. 具有常属性（`const`）的常对象只能调用常成员函数。

与普通成员函数一样，常成员函数也有**类内声明类内实现**和**类内声明类外实现**两种语法形式。

常成员函数类内声明类内实现的语法

```
class 类名{
   public:
      // 类内声明，类内实现。
      返回类型 成员函数名(形式参数列表) const { ... };
}
```

常成员函数类内声明，类外实现的语法

```
class 类名{
   public:
      // 类内声明。
      返回类型 函数名2(形式参数列表) const;
}
```

类外实现的语法

```
返回类型 类名::函数名2(形式参数列表) const { ... }
```

示例

```
// filename: const_member_func
#include <iostream>
#include <string>

using namespace std;

class Person {
    public:
        Person(const string & n, int a): name(n), age(a)  {
            cout << "创建: Person(" << n << ", " << a << ");\n";
        }
        ~Person(void) {
            cout << "析构: ~Person(" << name << ", " << age << ");\n";
        }
        int getAge(void) const; // 类内声明
    private:
        string name; // 姓名
        int age; // 年龄
};

// 常成员函数的类外实现
int Person::getAge(void) const {
    return this->age;
}

int main(int argc, char * argv[]) {
    Person p1("老魏", 35);
    const Person p2("魏明择", 25);

    cout << "p1的年龄:" << p1.getAge() << endl;
    cout << "p2的年龄:" << p2.getAge() << endl;

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o const_member_func const_member_func.cpp
.weimz@mzstudio:~$ ./const_member_func
创建: Person(老魏, 35);
创建: Person(魏明择, 25);
p1的年龄:35
p2的年龄:25
```

```
析构: ~Person(魏明择, 25);
析构: ~Person(老魏, 35);
```

实验

尝试将自己编写的类的只读成员函数都改成常成员函数。

第十章、面向对象编程

1. 面向对象编程思想

面向对象编程（Object Oriented Programming，OOP）的思想是把一切看成现实世界的物体（对象），通过对象和对象之间的关系来构建软件系统。

面向对象编程思想的组成要素是**类**和**对象**。

类是对象的描述（蓝图）

对象是由类创建的实例，真正占用内存的变量。

思想是用类来描述对象的行为。用类封装对象数据。

描述方法示例

```
有两个人（Human），张三(zhang3) 和 李四(li3)

张三工作赚钱 1000 元
李四借张三 300 元
李四买游戏机花了270元;
张三教李四玩王者荣耀
李四教张三 C++
```

用 C++ 语言来描述面相对象的思想，如下所示。

示例代码

```
// filename: oop_demo.cpp
#include <iostream>
#include <string.h>

using namespace std;

class Human {
    public:
        Human(const char * n) : name(n),money(0){}
        void showInfo(void) {
            cout << name << ", 有钱:" << money << "元, 技能:" << skill << endl;
        }
```

```
        void work(int m) {
            money += m;
            cout << name << " 工作赚了" << m << "元，共有"  << money << "元" << end
l;
        }
        void borrowFrom(Human & h, int m) {
            if (m > h.money) {
                cout << h.name << "没有" << m << "元这么多钱，借钱失败!" << endl;
                return;
            }
            h.money -= m;
            money += m;
            cout << h.name << "借给" << name << m << "元钱!" << endl;
        }
    private:
        string name;
        int money;
        string skill; // 技能 :w

};

int main(int argc, char * argv[]) {
    // 有两个人（Human），张三(zhang3) 和 李四(li3)
    Human zhang3("张三");
    Human li4("李四");

    // 张三工作赚钱 1000 元
    zhang3.work(1000);
    // 李四借张三 300 元
    li4.borrowFrom(zhang3, 300);
    zhang3.showInfo();
    li4.showInfo();
    // 李四买游戏机花了270元;
    // li4.buy("游戏机", 270);
    // 李四教张三玩王者荣耀
    // li4.teach(zhang3, "王者荣耀");
    // 张三教李四 C++
    // zhang3.teach(li4, "C++");

    cout << "程序结束!" << endl;
    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o oop_demo oop_demo.cpp
weimz@mzstudio:~$ ./oop_demo
张三 工作赚了1000元，共有1000元
张三借给李四300元钱!
张三, 有钱:700元, 技能:
李四, 有钱:300元, 技能:
程序结束!
```

练习

完成面向对象思想示例的张三、李四的其他逻辑。

2. 封装

封装是指隐藏对象的实现细节，让使用者不关心这些细节。从而保证数据的安全性。

封装后的类使用者只需要调用简单的接口（成员函数）即可使用。

C++ 通过访问说明符实现封装。

访问说明符

- `public`
- `protected`
- `private`

访问说明符	成员函数	子类成员函数	其他
public	可以访问	可以访问	可以访问
protected	可以访问	可以访问	无法访问
private	可以访问	无法访问	无法访问

3. 继承

继承是允许一个类（子类或派生类）基于另外一个类（父类或基类）创建，并继承基类的成员变量和成员函数。

继承的作用：

1. 将公有部分放入基类，实现代码共享。
2. 子类可以在原有的基础上添加新的功能。

继承的语法：

```
class 子类名: 继承方式 基类名 {
    ...
};
```

继承方式

- 共有继承（`public`）
- 保护继承（`protected`）
- 私有继承（`private`）

继承后的访问权限表

基类访问权限	继承方式	本类访问权限
public	public	public
protected	public	protected
private	public	不可访问
public	protected	protected
protected	protected	protected
private	protected	不可访问
public	private	private
protected	private	private
private	private	不可访问

显式调用父类的构造函数的语法

```
class 子类名: 继承方式 基类名 {
     public:
          子类名(形参列表): 基类名(实参),成员变量1(实参),成员变量2(实参){...}
};
```

示例

```
/ filename: inherit.cpp
#include <iostream>

using namespace std;

// 点类(描述一个点的位置，面积等信息)
class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
```

```
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
        float getArea(void) { // 获取图形面积
            return 0;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) {
            //  x = ax; y = ay; r = radius;
            cout << "Circle(" << x << "," << y << "," << r << ")\n";
        }
        void info(void) {
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    Point p1(3, 5);
    Circle c1(4, 6, 10);

    cout << "sizeof(p1)" << sizeof(p1) << endl;
    cout << "sizeof(c1)" << sizeof(c1) << endl;
    c1.info();
    p1.info();
    p1.moveTo(100, 105);
    // 子类对象不存在moveTo，则调用父类的方法
    c1.moveTo(200, 210);
    p1.info();
    c1.info();
    cout << "程序结束!" << endl;
    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o inherit inherit.cpp
weimz@mzstudio:~$ ./inherit
Point(3,5)
Point(4,6)
Circle(4,6,10)
sizeof(p1)8
sizeof(c1)12
圆(4,6,10)
```

```
点(3,5)
点(100,105)
圆(200,210,10)
程序结束!
```

4. 覆盖

覆盖是指子类实现了同名，参数列表也完全相同的成员函数，在子类对象调用该方法时，实际调用的是子类的方法（父类的方法不会被调用），此种现象成为制覆盖（或屏蔽）。

> 子类成员函数的调用规则是向上查找规则，即子类先查找自己是否有这个成员函数，如果没有就查找父类，如果再没有就查找父类的父类... 以此类推。都没有找到则报错。

示例

父类 `Point` 实现了 `getArea` 和 `getLength` 两个成员函数，子类也是现实了同样名称和参数的成员函数进行了覆盖。此时父类对象调用父类的成员函数，子类则调用子类的成员函数。

```
// filename: inherit.cpp
#include <iostream>

using namespace std;

#define PI (3.1415926)
// 点类(描述一个点的位置，面积等信息)
class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
        float getArea(void) { // 获取图形面积
            return 0;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
```

```
            : Point(ax, ay), r(radius) {
            //  x = ax; y = ay; r = radius;
            cout << "Circle(" << x << "," << y << "," << r << ")\n";
        }
        void info(void) {
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
        float getArea(void) {
            return PI*r*r;
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    Point p1(3, 5);
    Circle c1(4, 6, 10);

    cout << "sizeof(p1)" << sizeof(p1) << endl;
    cout << "sizeof(c1)" << sizeof(c1) << endl;
    c1.info();
    p1.info();
    p1.moveTo(100, 105);
    // 子类对象不存在moveTo，则调用父类的方法
    c1.moveTo(200, 210);
    p1.info();
    c1.info();
    cout << "p1的面积" << p1.getArea() << endl;
    cout << "c1的面积" << c1.getArea() << endl;
    cout << "程序结束!" << endl;
    return 0;
}
```

可见虽然 `c1` 对象的父类是 `Point` 类。但 `c1.getArea()` 则调用自己的 `Circle` 类自己的 `getArea` 成员函数。

5. 显式调用父类的成员函数

在有继承关系的类中，一旦子类对父类同一个成员函数进行了覆盖，则子类对象以默认调用的一定是子类的成员函数，而父类的成员函数则被覆盖（屏蔽）。如果再子类成员函数中需要显示调用父类的成员函数，则需要使用类名和作用域限定符（`::`）来显式的调用被覆盖的成员函数。

语法

```
class 子类名: public 父类名 {
    返回类型 成员函数名(形式参数列表) {
        //  调用父类的成员函数(用父类名指定名字空间)
        父类名::成员函数名(实际调用参数列表)
```

```
    }
};
```

示例

```
// filename: call_demo.cpp
#include <iostream>

using namespace std;

#define PI (3.1415926)
// 点类(描述一个点的位置，面积等信息)
class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
        float getArea(void) {
            return 0;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) {
            //  x = ax; y = ay; r = radius;
            cout << "Circle(" << x << "," << y << "," << r << ")\n";
        }
        void info(void) {
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
        void showBaseInfo(void) {
            // 显式调用Point类的info 而非自己的info函数
            Point::info();
        }
        float getArea(void) {
            return PI*r*r;
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    Point p1(3, 5);
```

```
    Circle c1(4, 6, 10);

    c1.info();
    p1.info();
    p1.moveTo(100, 105);
    // 子类对象不存在moveTo，则调用父类的方法
    c1.moveTo(200, 210);
    p1.info();
    c1.info();
    c1.showBaseInfo();
    cout << "程序结束!" << endl;
    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o call_demo call_demo.cpp
weimz@mzstudio:~$ ./call_demo
Point(3,5)
Point(4,6)
Circle(4,6,10)
圆(4,6,10)
点(3,5)
点(100,105)
圆(200,210,10)
点(200,210)
程序结束!
```

从上述结果可以看出 `c1.showBaseInfo();` 实际打印的是 `点(200,210)`，此成员函数调用的是 `Point::info` 成员函数。

实验

自己写一个显式调用被覆盖的父类成员函数的示例。

6. 多态

多态（Polymorphism）是不同的对象展现出不同的状态。**多态** 允许程序使用同一的接口（通常是基类的指针）来操作不同的派生类对象。而具体执行那个类的成员函数则在运行时根据不同的对象来决定。

多态是指使用父类的指针，在指针指向不同的子类对象时，调用的成员函数是子类对象的成员函数。即调用由指向的类型决定，而不是由指针类型决定，这种现象叫做多态。

C++ 中的多态是使用**虚（成员）函数**来实现的。

虚函数 是指在成员函数声明时使用关键字 `virtual` 声明的成员函数。

语法

```
class 类名{
    // 声明是加virtual,  实现时不用加virtual.
    virtual 返回类型  成员函数名(形式参数列表) { ... }
};
```

说明:

- 虚函数的 `virtual` 关键字默认是继承的，即父类如果是虚函数，则子类的覆盖成员函数也一定是虚函数，即便子类不使用 `virtual` 关键字声明，则此子类相同的成员函数也是虚函数。

示例

```
// filename: poly.cpp
#include <iostream>

using namespace std;

#define PI (3.1415926)
// 点类(描述一个点的位置，面积等信息)
class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        // 定义为虚函数
        virtual void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
        // 定义为虚函数
        virtual float getArea(void) {
            return 0;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) {
            //  x = ax; y = ay; r = radius;
            cout << "Circle(" << x << "," << y << "," << r << ")\n";
```

```
        }
        void info(void) {  // 子类的此函数也为虚函数
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
        void showBaseInfo(void) {
            // 显式调用Point类的info 而非自己的info函数
            Point::info();
        }
        float getArea(void) {
            return PI*r*r;
        }
    public:
        float r; // 半径
};
// 定义一个矩形类（长方形）
class Rect : public Point {
    public:
        Rect(int ax, int ay, int aw, int ah)
            : Point(ax, ay), w(aw), h(ah) {
                cout << "Rect(" << x << ","
                    << y << ", " << w << ", "
                    << h << ")\n";
            }
        void info(void) { // 子类的此函数也为虚函数
            cout << "长方形(" << x << ","
                << y << ", " << w << ", "
                << h << ")\n";
        }
        float getArea(void) { // 虚函数
            return w * h;
        }
    protected:
        int w; // 宽
        int h; // 高
};

int main(int argc, char * argv[]) {
    Point * shapes[4];
    Circle c1(4, 6, 10);
    Point * ps;

    shapes[0] = new Rect(1, 2, 3, 4);
    shapes[1] = &c1;
    shapes[2] = new Point(8, 9);
    shapes[3] = new Circle(10, 11, 12);
    for (int i = 0; i < 4; i++) {
        ps = shapes[i];
        ps->info();
        cout << ps->getArea() << endl;
    }
    cout << "程序结束!" << endl;
    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o poly poly.cpp
weimz@mzstudio:~$ ./poly
Point(4,6)
Circle(4,6,10)
Point(1,2)
Rect(1,2, 3, 4)
Point(8,9)
Point(10,11)
Circle(10,11,12)
长方形(1,2, 3, 4)
12
圆(4,6,10)
314.159
点(8,9)
0
圆(10,11,12)
452.389
程序结束!
```

可见 `main` 函数中的 `ps` 指针的类型是 `Shape*` 但因为他指向了不同类型的子类对象，因此 `ps->info()` 和 `ps->getArea()` 都调用的不同的成员函数。此现象为多态。

说明

- C++ 多态是运行时动态的选择调用的成员函数（不是在编译阶段确定的）。

7. 虚函数表

虚函数表 （又称**虚表**）是 C++ 编译器为实现运行时多态而建立的一张查找表，此查找表记录是记录该类对象的所有虚成员函数地址的表。

虚函数表是一个函数指针的数组，每个指针都指向他的虚成员函数。

虚函数表由 C++ 编译器在编译期间确定，并在运行时保存于内存中，当创建含有虚函数的对象时，对象内存的起始地址会存放一个（或多个）指针指向这个虚函数表。

虚函数表是 C++ 实现运行时多态的基础。

说明

- 在单继承的对象中，一个类可以有多个虚函数，但虚函数表的指针只有一个。
- C++ 是使用虚函数表实现的多态，如果一个类有虚成员函数，则此类对象的起始地址是一个虚函数表指针，指向对象的类型对相应的虚函数表。

示例1

没有虚函数表的类创建的对象的内存结构分析

```
// filename: vtable1.cpp
#include <iostream>

using namespace std;

// 点类(描述一个点的位置，面积等信息)
class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) { }
        // 定义为普通成员函数
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
        // 定义为普通成员函数
        float getArea(void) {
            return 0;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) { }
        void info(void) {  // 子类的此函数也为虚函数
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
        float getArea(void) {
            return 3.14*r*r;
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    Point p1(1, 2);
    Circle c1(3, 4, 5);
    Point * p = &c1;

    cout << "sizeof(p1):" << sizeof(p1) << endl;
    cout << "sizeof(c1):" << sizeof(c1) << endl;
    p->info();

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o vtable1 vtable1.cpp
weimz@mzstudio:~$ ./vtable1
sizeof(p1):8
sizeof(c1):12
点(3,4)
```

指针 `p` 的类型是 `Point*`，它执行 `Circle` 类型的对象 `c1`，因为 `Point::info()` 不是虚函数，因此 `p->info()` 会根据 `p` 的类型来确定执行 `Point::info()` 函数。

内存结构分析

此时 `p1` 和 `c1` 两个对象的内存结构如下:

```
点 p1 的内存结构
+--------+
|  x: 1  |
|  y: 2  |
+--------+

圆 c1 的内存结构
+--------+
|  x: 3  |
|  y: 4  |
|  r: 5  |
+--------+
```

示例2

将 `void Point::info()` 成员函数声明为虚函数，查看运行结果并分析内存结构，代码修改如下：

```
// filename: vtable2.cpp
#include <iostream>

using namespace std;

// 点类(描述一个点的位置，面积等信息)
class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) { }
        // 定义为虚函数
        virtual void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
        float getArea(void) {
```

```
            return 0;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) { }
        void info(void) {  // 子类的此函数也为虚函数
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
        float getArea(void) {
            return 3.14*r*r;
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    Point p1(1, 2);
    Circle c1(3, 4, 5);
    Point * p = &c1;

    cout << "sizeof(p1):" << sizeof(p1) << endl;
    cout << "sizeof(c1):" << sizeof(c1) << endl;
    p->info();
    cout << "p->getArea() 返回结果:" << p->getArea() << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o vtable2 vtable2.cpp
weimz@mzstudio:~$ ./vtable2
sizeof(p1):16
sizeof(c1):24
p->getArea() 返回结果:0
圆(3,4,5)
```

从运行结果可以看出， `p1` 和 `p2` 两个对象的内存结构变大。`p->getArea()` 依旧调用 `Point::getAear()` 成员函数，而 `p->info()` 则调用 `Circle::info()` 这个虚函数。

内存结构分析

此时 `p1` 和 `c1` 两个对象的内存结构如下:

```
点 p1 的内存结构     虚表
+--------+      +--------+
| __vptr | ---> |   [0]  | --->  "void Point::info()"
|  x: 1  |      +--------+
|  y: 2  |
|  r: 5  |
+--------+

圆 c1 的内存结构     虚表
+--------+      +--------+
| __vptr | ---> |   [0]  | --->  "void Circle::info()"
|  x: 3  |      +--------+
|  y: 4  |
|  r: 5  |
+--------+
```

示例3

将 `void Point::info()` 和 `float Point::getArea(void)` 两个成员函数都声明为虚函数，查看运行结果并分析内存结构。

```
// filename: vtable3.cpp
#include <iostream>

using namespace std;

// 点类(描述一个点的位置，面积等信息)
class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) { }
        // 定义为虚函数
        virtual void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
        // 定义为虚函数
        virtual float getArea(void) {
            return 0;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) { }
        void info(void) {  // 子类的此函数也为虚函数
            cout << "圆(" << x << "," << y << "," << r << ")\n";
```

```
        }
        float getArea(void) {
            return 3.14*r*r;
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    Point p1(1, 2);
    Circle c1(3, 4, 5);
    Point * p = &c1;

    cout << "sizeof(p1):" << sizeof(p1) << endl;
    cout << "sizeof(c1):" << sizeof(c1) << endl;
    p->info();
    cout << "p->getArea() 返回结果:" << p->getArea() << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o vtable3 vtable3.cpp
weimz@mzstudio:~$ ./vtable3
sizeof(p1):16
sizeof(c1):24
圆(3,4,5)
p->getArea() 返回结果:78.5
```

从运行结果可以看出， `p->getArea()` 调用 `Circle::getAear()` 成员函数，且 `p->info()` 也调用 `Circle::info()` 这个虚函数。

内存结构分析

此时 `p1` 和 `c1` 两个对象的内存结构如下:

```
点 p1 的内存结构     虚表
+--------+      +--------+
| __vptr | ---> |   [0]  | --->  "void Point::info()"
|  x: 1  |      |   [1]  | --->  "float Point::getArea()"
|  y: 2  |      +--------+
|  r: 5  |
+--------+

圆 c1 的内存结构     虚表
+--------+      +--------+
| __vptr | ---> |   [0]  | --->  "void Circle::info()"
|  x: 3  |      |   [1]  | --->  "float Circle::getArea()"
|  y: 4  |      +--------+
```

```
|  r: 5  |
+--------+
```

8. 纯虚函数和抽象类

纯虚函数 是指在类内只有存虚函数声明但一定没有实现的虚成员函数。

语法

```
class 类名{
    // 声明是加virtual,  不用实现
    virtual 返回类型  成员函数名(形式参数列表)=0;
};
```

纯虚函数的作用是声明一个虚成员函数，继承此类的子类必须实现并覆盖此纯虚函数，如果子类不进行覆盖并实现此纯虚函数则子类对象则不能创建该类的对象。

抽象类

类内包含纯虚函数的类称为**抽象类（或纯虚类）**，抽象类并不能用来创建对象。抽象类仅用来派生子类，要求派生自抽象类的子类则必须重写并实现这些纯虚函数，否则不能创建该类的对象。

抽象类的主要作用是定义接口规范，让此抽象类的子类创建的对象都有统一的成员函数（接口），通过这些成员函数可以统一调用，又根据多态的原理实现个性化的功能。

例

定义一个抽象类如下:

```
class Shape{ // 抽象类(Abstract Class);
    public:
        // 纯虚函数的声明
        virtual void moveTo(float x, float y)=0;
        virtual void info(void) =0;
        virtual float getArea(void) =0;
};
```

则派生自此类的子类则必须要重写这三个成员函数，即该类的子类一定有这三个成员函数。

示例

```cpp
// filename: abstract_cls.cpp
#include <iostream>

using namespace std;

#define PI (3.1415926)
// 抽象类
class Shape{ // 抽象类(Abstract Class);
    public:
        // 纯虚函数的声明
        virtual void moveTo(float x, float y)=0;
        virtual void info(void) =0;
        virtual float getArea(void) =0;
};
// 点类(描述一个点的位置，面积等信息)
class Point: public Shape {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        // 定义为虚函数
        virtual void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
        // 定义为虚函数
        virtual float getArea(void) {
            return 0;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) {
            //  x = ax; y = ay; r = radius;
            cout << "Circle(" << x << "," << y << "," << r << ")\n";
        }
        void info(void) {  // 子类的此函数也为虚函数
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
        void showBaseInfo(void) {
            // 显式调用Point类的info 而非自己的info函数
            Point::info();
        }
        float getArea(void) {
            return PI*r*r;
        }
    public:
        float r; // 半径
```

```
};
// 定义一个矩形类（长方形）
class Rect : public Point {
    public:
        Rect(int ax, int ay, int aw, int ah)
            : Point(ax, ay), w(aw), h(ah) {
                cout << "Rect(" << x << ","
                    << y << ", " << w << ", "
                    << h << ")\n";
            }
        void info(void) { // 子类的此函数也为虚函数
            cout << "长方形(" << x << ","
                << y << ", " << w << ", "
                << h << ")\n";
        }
        float getArea(void) { // 虚函数
            return w * h;
        }
    protected:
        int w; // 宽
        int h; // 高
};

int main(int argc, char * argv[]) {
    // Shape s; // 报错，抽象类不能创建对象
    Shape * shapes[4];
    Circle c1(4, 6, 10);
    Shape * ps;

    shapes[0] = new Rect(1, 2, 3, 4);
    shapes[1] = &c1;
    shapes[2] = new Point(8, 9);
    shapes[3] = new Circle(10, 11, 12);
    for (int i = 0; i < 4; i++) {
        ps = shapes[i];
        ps->info();
        cout << ps->getArea() << endl;
    }
    cout << "程序结束!" << endl;
    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o abstract_cls abstract_cls.cpp
.weimz@mzstudio:~$ ./abstract_cls
Point(4,6)
Circle(4,6,10)
Point(1,2)
Rect(1,2, 3, 4)
Point(8,9)
Point(10,11)
Circle(10,11,12)
长方形(1,2, 3, 4)
12
```

```
圆(4,6,10)
314.159
点(8,9)
0
圆(10,11,12)
452.389
程序结束!
```

9. 虚析构函数

如果一个类可能被继承，并且会通过基类指针来删除（`delete`）派生类的对象，那么他的析构函数必须是虚函数。否则只会调用基类的析构函数，不会调用派生类的析构函数，导致部分资源泄露。

虚析构函数的作用是保证在使用 `delete` 来销毁对象是一定能够调用该类的析构函数来防止资源丢失。

语法

```
class 类名{
    // 声明虚析函数。
    virtual ~类名(void){...}
};
```

示例

```
// filename: v_decons.cpp
#include <iostream>

using namespace std;

#define PI (3.1415926)
// 抽象类
class Shape{ // 抽象类(Abstract Class);
    public:
        virtual ~Shape() {cout << "~Shape()" << endl;};
        // 纯虚函数的声明
        virtual void moveTo(float x, float y)=0;
};
// 点类(描述一个点的位置，面积等信息)
class Point: public Shape {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        ~Point() {
            cout << "~Point(" << x << "," << y << ")\n";
```

```
        }
        void moveTo(float new_x, float new_y) {
            x = new_x; y = new_y;
        }
    public:
        float x;
        float y;
};
// 圆类(描述圆的位置、半径、面积等信息)
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) {
            cout << "Circle(" << x << "," << y << "," << r << ")\n";
        }
        ~Circle() {
            cout << "~Circle(" << x << "," << y << "," << r << ")\n";
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    Point * p;
    p = new Point(1, 2);
    delete p;
    Shape *p2;
    p2 = new Circle(3, 4, 5);
    delete p2; // 调用圆的析构函数

    cout << "程序结束!" << endl;
    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o v_decons v_decons.cpp
weimz@mzstudio:~$ ./v_decons
Point(1,2)
~Point(1,2)
Point(3,4)
Circle(3,4,5)
~Circle(3,4,5)
~Point(3,4)
程序结束!
```

实验

将上述示例的抽象类 `Shape` 的 虚析构函数 `virtual ~Shape(){...}` 改为 `~Shape(){...}`, 然后运行此示例，查看器运行结果，想想为什么?

10.多继承

C++允许一个子类继承自多个父类，这种现象叫做**多继承**。

多继承的语法

```
class 子类名: 继承方式 父类名1, 继承方式 父类名2 {
};
```

示例

```
// filename: multi_inherit.cpp
#include <iostream>
#include <string.h>

using namespace std;

class Plane {
    public:
        void fly(int height) {
            cout << "飞机以" << height<< "的高度飞行\n";
        }
    private:
        int xxx;
};

class Car{
    public:
        void run(int speed) {
            cout << "汽车以" << speed<< "km/h的速度行使\n";
        }
    private:
        int yyy;
};

class PlaneCar: public Car, public Plane{
};

int main(int argc, char * argv[]) {
    PlaneCar pc;
    pc.fly(10000);
    pc.run(210);
    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o multi_inherit multi_inherit.cpp
weimz@mzstudio:~$ ./multi_inherit
```

```
飞机以10000的高度飞行
汽车以210km/h的速度行使
```

11. 虚继承

在继承的过程中有时会出现用一个类派生出多个子类，又通过多继承将多个子类派生出一个子类。这种创建的子类对象中可能会包含多个基类对象。

以下是多继承中的一个著名案例，菱形继承（也叫钻石继承）。

```
  A
 / \
B   C
 \ /
  D
```

B 类继承自 A类，因此 B 类对象即包含自身的成员，也会包含 A 类的成员。同理 C 类对象也会包含 A 类的成员。又因为 D 类继承自 B 类和 C 类，因此 D 类的对象就有两份 A 类的成员变量（一份来自于 B 类，一份来自于 C 类）。而 D 类根本就不需要两份 A 类成员（会出现冲突）。解决这个问题的语法是使用**虚继承**。

虚继承的语法

```
class 子类名: virtual 继承方式 父类名 {
};
```

示例

```
// filename: diamond.cpp
#include <iostream>

using namespace std;

class A {
    public:
        A() : a(1) {}
        void method() {
            cout << "a:" << a << endl;
        }
    private:
        int a;
};
class B : virtual public A {
    public:
        B() : A(), b(2) {}
```

```
    private:
        int b;
};
class C : virtual public A {
    public:
        C() : A(), c(3) {}
    private:
        int c;
};
class D : public B, public C {
    public:
        D() : B(), C(), d(4) {}
    private:
        int d;
};

int main(int argc, char * argv[]) {
    D d;
    d.method();
    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o diamond diamond.cpp
weimz@mzstudio:~$ ./diamond
a:1
```

上述代码中， B 类和 C 类都是虚继承自 A 类，因此在多继承创建 D 类时D 类中只有一个 A 类的对象，否则编译则会出现成员函数或者成员变量出现争议（`ambiguous`） 的编译错误。

第十一章、友元

在 C++ 的语法中，友元（`friend`）是指让声明的一个函数、类或成员函数可以访问该类的私有和保护成员。

友元的作用

让全局函数、其他类或其他类的成员函数可以访问该类的私有和保护成员。

友元是一种通过关键 `friend` 来实现的声明。

关键字

```
friend
```

1. 友元声明

友元声明的分类

1. 友元函数
2. 友元成员函数
3. 友元类

语法

```
class 类名 {
   // 友元函数的声明语法
   friend 返回类型 友元函数名(形式参数列表);
   // 友元成员函数的声明语法
   friend 返回类型 其他类::类名友元函数名(形式参数列表);
   // 友元类的声明语法
   friend class 其他类;
};
```

示例

```
#include <iostream>

using namespace std;
```

```
class Children; // 类声明
class Parent{
    public:
        Parent(const string & n, int a)
            : name(n),age(a),money(0){}
        void work(int m);
        void showInfo(void);
        void giveMoneyTo(Children & c, int m); // 给孩子钱
    private:
        string name;
        int age;
        int money;
    // 声明show_money函数可以访问本类的所有成员
    friend void show_money(const Parent &);
    friend class Children;
};

class Children {
    public:
        Children(const string & n, int a)
            : name(n),age(a),money(0){}
        void showInfo(void);
        void borrowTo(Parent & p, int m);
    private:
        string name;
        int age;
        int money;
// 仅声明Parent类的giveMoneyTo函数可以访问本类的所有成员
        friend void Parent::giveMoneyTo(Children & c, int m);
};

void Parent::work(int m) {
    money += m;
}

void Parent::showInfo(void) {
    cout << age << "岁的" << name << "有钱" << money
        << "元" << endl;
}

void Children::showInfo(void) {
    cout << age << "岁的" << name << "有钱" << money
        << "元" << endl;
}

void Parent::giveMoneyTo(Children & c, int m) {
    if (m > this->money) {
        cout << "钱不够" << endl;
        return;
    }
    this->money -= m;
    c.money += m;
}

void Children::borrowTo(Parent & p, int m) {
    if (m > this->money) {
```

```
        cout << "本宝宝没钱，不借" << endl;
        return;
    }
    this->money -= m;
    p.money += m;
}

// 全局函数，用来打印家长的钱数;
void show_money(const Parent & p) {
    cout << p.age << "岁的" << p.name << "有钱"
        << p.money << "元" << endl;
}

int main(int argc, char * argv[]) {
    Parent p("老张", 31);
    Children c("小张", 12);
    p.work(1000);
    show_money(p);
    p.giveMoneyTo(c, 700);
    p.showInfo();
    c.showInfo();
    c.borrowTo(p, 200);
    p.showInfo();
    c.showInfo();

    return 0;
}
```

第十二章、运算符重载

运算符重载是指在自定义的类或普通函数对已有的运算符进行重新定义，在使用时用运算符代替成员函数进行操作。

1. 运算符的重载原理

在说明运算符重载之前我门先以数学中复数为例来封装一个 Complex 类，简化实现复数的加、减、乘等操作。

复数简介

每个复数都分为**实部**（`real`）和 **虚部**（`image`）两部分，如：`(1+2i)` 的实部是 `1`，虚部是 `2`。

他们的运算规则如下：

```
(1+2i) + (3+4i) = (4+6i);
(3+4i) - (1+2i) = (2+2i)
(1+2i) * (3+4i) = (1 * 3 + 1 * 4i + 3 * 2i + 2i*4i) = (-5 + 10i);
```

下面来封装一个 `Complex` 类，并实现成员函数 `add`、`sub`、`mul` 来实现负数的**加**、**减**、**乘**等操作。并通过 `info` 成员函数打印负数的值。

示例代码如下：

```
// filename: operator_demo.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
        Complex(double r=0, double i=0): real(r), image(i){}
        void info(){
            cout << "(" << real << "+" << image << "i)\n";
        }
        Complex add(const Complex & right) const {
            return Complex (
                    this->real + right.real,
                    this->image + right.image);
        }
        Complex sub(const Complex & right) const {
            return Complex (
```

```
                    this->real - right.real,
                    this->image - right.image);
        }
        Complex mul(const Complex & right) const {
            return Complex (
                    this->real * right.real - this->image * right.image,
                    this->real * right.image + this->image * right.real);
        }
    private:
        double real;
        double image;
};

int main(int argc, char * argv[]) {
    Complex c1(1, 2);
    Complex c2(3, 4);
    Complex c3;
    c3 = c1.add(c2);
    c3.info();
    c3 = c2.sub(c1);
    c3.info();
    c3 = c1.mul(c2);
    c3.info();

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o operator_demo operator_demo.cpp
weimz@mzstudio:~$ ./operator_demo
(4+6i)
(2+2i)
(-5+10i)
```

下面我们将成员函数 `add` 换成 `operator +`、`sub` 换成 `operator -`、`mul` 换成 `operator *`，然后 `main` 函数的 `c1.add(c2)` 替换成 `c1 + c2`、`c2.sub(c1)` 替换成 `c2 - c1`、`c1.mul(c2)` 替换成 `c1 * c2`

示例代码如下:

```
// filename: operator_add.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
        Complex(double r=0, double i=0): real(r), image(i){}
        void info(){
            cout << "(" << real << "+" << image << "i)\n";
        }
        Complex operator +(const Complex & right) const {
```

```
            return Complex (
                    this->real + right.real,
                    this->image + right.image);
        }
        Complex operator -(const Complex & right) const {
            return Complex (
                    this->real - right.real,
                    this->image - right.image);
        }
        Complex operator *(const Complex & right) const {
            return Complex (
                    this->real * right.real - this->image * right.image,
                    this->real * right.image + this->image * right.real);
        }
    private:
        double real;
        double image;
};

int main(int argc, char * argv[]) {
    Complex c1(1, 2);
    Complex c2(3, 4);
    Complex c3;
    c3 = c1 +c2;
    c3.info();
    c3 = c2 - c1;
    c3.info();
    c3 = c1 *c2;
    c3.info();

    return 0;
}
```

编译和运行结果同没有替换之前也是一样的。结果如下：

```
weimz@mzstudio:~$ g++ -o operator_add operator_add.cpp
weimz@mzstudio:~$ ./operator_add
(4+6i)
(2+2i)
(-5+10i)
```

可见，运算符实际就是函数，这个函数可以是类的成员函数，也可以全局函数。

运算符重载的语法

```
class 类名 {
    返回类型 operator 运算符(形参列表) {}
};
```

运算符说明

- 运算符重载实质就是函数重载。
- 运算符重载只能使用现有的运算符，不能创造新的运算符。
- 运算符重载的参数的个数是固定的，类型可以自定义。

不支持重载的运算符

- `::`作用域运算符
- `?:` 条件运算符
- `.` 成员访问运算符
- `.*` 成员指针访问运算符

2. 二元运算符重载

C++ 语言中可以重载的二元运算符有：

算术运算符

```
+   -   *   /   %
```

关系运算符

```
<   <=   >   >=   ==   !=
```

逻辑运算符

```
&&   ||
```

位运算符

```
&   |   ^   <<   >>
```

赋值运算符

```
=   +=   -=   *=   /=   %=   &=   |=   ^=   <<=   >>=
```

逗号运算符

```
,
```

成员指针运算符

```
->*   .*
```

以成员函数重载运算的语法格式为

```
class 类名 {
    返回类型 operator 二元运算符(数据类型 右操作数) {...}
};
```

说明

1. 运算符重载的返回类型可以根据实际需要来确定。
2. 以成员函数的方式重载二元运算符的函数参数只能有一个，这个参数的类型根据重载规则可以是任意类型。

3. 运算符重载的方式

C++ 中有三种运算符重载的方式：

1. 成员函数方式重载。
2. 友元全局函数方式重载。
3. 全局函数方式重载。

以之前定义的 `Complex` 类为例，现在我们已经创建了一个对象 `Complex c1(1, 2)`，现在需要实现 `c1 + 1` 和 `5 + c1` 这样的操作，下面来用三种方法重载加号运算符。

需要注意的是一个虚数和整数相加时是实部相加，虚部不变。

1. 成员函数方式重载

```
// filename: op_methed1.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
```

```
        Complex(double r=0, double i=0): real(r), image(i){}
        void info(){
            cout << "(" << real << "+" << image << "i)\n";
        }
        Complex operator +(int right) const {
            return Complex ( this->real + right, this->image);
        }
    private:
        double real;
        double image;
};
int main(int argc, char * argv[]) {
    Complex c1(1, 2);
    Complex c3;
    c3 = c1 + 1;
    c3.info();
    // c3 = 5 + c1; // 报错
    // c3.info();

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o op_methed1 op_methed1.cpp
weimz@mzstudio:~$ ./op_methed1
(2+2i)
```

上述示例成功的重载了 `+` 运算符能够是一个 `Complex` 类型的对象和一个整数相加。但是如果是一个整数和一个 `Complex` 类型的对象相加（如：`5 + c1`）则会报错，因为 左操作数是 `int` 类，`int` 类中没有重载能和 `Complex` 相加的成员函数。要解决这个问题则需要使用友元全局函数的方式重载 加号（`+`）运算符。

2. 友元全局函数方式重载

去掉原来的成员函数 `Complex::operator+(int)`，改用两个全局函数如下：

```
// filename: op_methed2.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
        Complex(double r=0, double i=0): real(r), image(i){}
        void info(){
            cout << "(" << real << "+" << image << "i)\n";
        }
    private:
        double real;
        double image;
```

```
    friend Complex operator +(const Complex & left, int right);
    friend Complex operator +(int left, const Complex & right);
};
// 重载加号运算符(Complex + 整数)
Complex operator +(const Complex & left, int right) {
    return Complex ( left.real + right, left.image);
}
// 重载加号运算符(整数 + Complex)
Complex operator +(int left, const Complex & right) {
    return Complex (left + right.real, right.image);
}
int main(int argc, char * argv[]) {
    Complex c1(1, 2);
    Complex c3;
    c3 = c1 + 1;
    c3.info();
    c3 = 5 + c1; // 报错
    c3.info();

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o op_methed2 op_methed2.cpp
weimz@mzstudio:~$ ./op_methed2
(2+2i)
(6+2i)
```

可见使用 友元方式重载运算符和同样可以实现相同的目的，并且可以不用修改 `int`类（`int` 类本来就无法修改）。

3. 全局函数方式重载

上述以友元函数的方式重载的示例中使用友元来重载加号运算符是因为 `Complex` 类中的两个成员变量 `real` 和 `image` 是私有成员，其他函数则无法方法，如果将其改为公有成员，则不需要使用友元，这时候则可以使用全局函数方式重载加号运算符。示例代码如下：

```
// filename: op_method3.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
        Complex(double r=0, double i=0): real(r), image(i){}
        void info(){
            cout << "(" << real << "+" << image << "i)\n";
        }
    public:
    // private:
```

```
        double real;
        double image;
};
// 重载加号运算符(Complex + 整数)
Complex operator +(const Complex & left, int right) {
    return Complex ( left.real + right, left.image);
}
// 重载加号运算符(整数 + Complex)
Complex operator +(int left, const Complex & right) {
    return Complex (left + right.real, right.image);
}
int main(int argc, char * argv[]) {
    Complex c1(1, 2);
    Complex c3;
    c3 = c1 + 1;
    c3.info();
    c3 = 5 + c1; // 报错
    c3.info();

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o op_method3 op_method3.cpp
weimz@mzstudio:~$ ./op_method3
(2+2i)
(6+2i)
```

特殊情况：

以下运算符只能使用成员函数的方式进行重载。

```
=  ()  []  ->
```

练习

使用上述三种方法重载 减号运算符。

4. 输出运算符的重载

本节来讲解如何让 `std::out` 能够对用户自定义的 `Complex` 类 的对象进行输出。即可以实现如下代码的功能

```
Complex c1(1, 2);

std::cout << c1;  // 直接输出 c1 对象。
```

由于做操作数是 `std::cout`，因此只能使用 友元全局函数的方式重载 `<<` 运算符来实现。

这里 `std::cout` 是 `ostream` 类型的对象，并且此运算符重载后需要返回 `ostream` 对象自身，这个返回的对象供下一个 `<<` 运算进行在进行输出。

示例

重载 `<<` 运算符，实现标准输出。

```
// filename: std_cout.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
        Complex(double r=0, double i=0): real(r), image(i){}
        Complex operator +(const Complex & right) const {
            return Complex (
                    this->real + right.real,
                    this->image + right.image);
        }
    private:
        double real;
        double image;

    // 友元全局函数声明
    friend ostream & operator << (ostream & o, const Complex & c);
};

ostream & operator << (ostream & o, const Complex & c) {
    o << "(" << c.real << "+" << c.image << "i)";
    return o;
}

int main(int argc, char * argv[]) {
    Complex c1(1, 2);
    Complex c2(3, 4);
    Complex c3;

    c3 = c1 + c2;
    cout << c1 << "+" << c2 << "=" << c3 << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o std_cout std_cout.cpp
weimz@mzstudio:~$ ./std_cout
(1+2i)+(3+4i)=(4+6i)
```

5. 一元运算符的重载

C++ 能够重载的一元运算符有：正号（`+`）、负号（`+`）、取反（`~`）、取地址（`&`）、解引用（`*`）、指针成员访问（`->`）、逻辑非（`!`）、自增（`++`）、自减（`--`）运算符。

以成员函数方式重载一元运算符时，成员函数通常不需要给定参数，因为他们操作数都是 `this`。

示例

以下以重载负号（`-`）运算符为例，可以使用 `-` 运算将负数的将 `Complex` 类型的对象的实部和虚部都取符号的负向操作。

```
// filename: op_minus.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
        Complex(double r=0, double i=0): real(r), image(i){}
        Complex operator -() {
            return Complex(-real, -image);
        }
    private:
        double real;
        double image;

    // 友元全局函数声明
    friend ostream & operator << (ostream & o, const Complex & c);
};

ostream & operator << (ostream & o, const Complex & c) {
    o << "(" << c.real << "+" << c.image << "i)";
    return o;
}

int main(int argc, char * argv[]) {
    Complex c1(1, 2);
    cout << "+c1:" << c1 << endl;
    cout << "-c1:" << -c1 << endl;

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o op_minus op_minus.cpp
weimz@mzstudio:~$ ./op_minus
+c1:(1+2i)
-c1:(-1+-2i)
```

可见 `-c1` 将 `c1` 实部和虚部的符号都进行了反向变化。

6. 自增、自减运算符重载

这一节讲解自增（`++`）和自减（`--`）运算符的重载。

自增运算符分为前置自增和后置自增运算，两个运算符同为一元运算符，C++ 编译器是为后置运算的重载函数添加一个 `int` 类型的参数来进行区分。

对于自减运算符也是如此，以下给出了这些运算符的重载的语法。

前置语法

```
class 类名{
   返回类型 operator++(void) {}
   返回类型 operator--(void) {}
}
```

后置语法

```
class 类名{
   返回类型 operator++(int) {}
   返回类型 operator--(int) {}
}
```

示例

以 Complex 类为例，以下设置规则，如果自增则实部自增，虚部不变。自减运算符也是如下。

```
// filename: incr_redu.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
        Complex(double r=0, double i=0): real(r), image(i){}
```

```
        // 前置自增
        const Complex & operator ++() {
            real += 1.0;
            return *this;
        }
        // 前置自减
        const Complex & operator --() {
            real -= 1.0;
            return *this;
        }
        // 后置自增
        Complex operator ++(int) {
            Complex last_val(*this);
            real += 1.0;
            return last_val;
        }
        // 后置自减
        Complex operator --(int) {
            Complex last_val(*this);
            real -= 1.0;
            return last_val;
        }
    private:
        double real;
        double image;

    // 友元全局函数声明
    friend ostream & operator << (ostream & o, const Complex & c);
};

ostream & operator << (ostream & o, const Complex & c) {
    o << "(" << c.real << "+" << c.image << "i)";
    return o;
}

int main(int argc, char * argv[]) {
    Complex c1(1, 2);
    Complex c2;

    c2 = ++c1;
    cout << "c1:" << c1 << " c2:" << c2 << endl;
    c2 = c1++;
    cout << "c1:" << c1 << " c2:" << c2 << endl;
    c2 = --c1;
    cout << "c1:" << c1 << " c2:" << c2 << endl;
    c2 = c1--;
    cout << "c1:" << c1 << " c2:" << c2 << endl;

    return 0;
}
```

编译和运算结果如下

```
weimz@mzstudio:~$ g++ -o incr_redu incr_redu.cpp
weimz@mzstudio:~$ ./incr_redu
```

```
c1:(2+2i) c2:(2+2i)
c1:(3+2i) c2:(2+2i)
c1:(2+2i) c2:(2+2i)
c1:(1+2i) c2:(2+2i)
```

7. 索引运算符重载

重载索引运算符（[]） 可以将一个对象作为一维数组来使用。

如，我们可以像如下代码一样使用。

```
MyClass m;

int x = c[100];  //  需要重载 [] 运算符。
```

示例

以下定义一个动态生成整数等差数列的容器类 `MyRange`，该类给出在创建时给出开始的整数值（`start`）、结束的整数值（`stop`，不包含 `stop`）和步长（`step`，默认为1）。其内部动态生成相应的整数。如：

```
MyRange r1(3, 6);  // 生成 3、4、5
MyRange r2(1, 10, 3);  // 生成 1、4、7
MyRange r3(10, 0, -2);  // 生成 10、8、6、4、2
```

接下来我们使用索引运算符就可以获取其中的数据，如

```
cout << r2[0] << endl; // 打印 1
cout << r2[1] << endl; // 打印 4
cout << r2[100] << endl; // 打印 0, 索引越界返回 0
```

完整示例代码

```
// filename: op_index.cpp
#include <iostream>

using namespace std;

class MyRange{
    public:
        MyRange(int begin, int end, int stp=1): start(begin), stop(end), step(st
p) {}
        // 重载 [] 运算符
        int operator [] (int index) {
            int value = start + index * step;
```

```
            if (step > 0 && value > stop)
                return 0; // 越界
            else if (step < 0 && value < stop)
                return 0; // 越界

            return value;
        }
    private:
        int start;
        int stop;
        int step;
    friend ostream & operator << (ostream & o, const MyRange &c);
};

ostream & operator << (ostream & o, const MyRange &c) {
    o << "[";
    if (c.step > 0)
        for (int v = c.start; v < c.stop; v += c.step) {
            if (v != c.start)
                o << ",";
            o << v;
        }
    else
        for (int v = c.start; v > c.stop; v += c.step) {
            if (v != c.start)
                o << ", ";
            o << v;
        }
    o << "]";
    return o;
}

int main(int argc, char * argv[]) {
    MyRange r1(3, 6);  // 生成 3、4、5
    MyRange r2(1, 10, 3);  // 生成 1、4、7
    MyRange r3(10, 0, -2);  // 生成 10、8、6、4、2

    cout << r1 << endl;
    cout << r2 << endl;
    cout << r3 << endl;

    cout << r2[0] << endl; // 打印 1
    cout << r2[1] << endl; // 打印 4
    cout << r2[100] << endl; // 打印 0, 索引越界返回 0

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o op_index op_index.cpp
weimz@mzstudio:~$ ./op_index
[3,4,5]
[1,4,7]
[10, 8, 6, 4, 2]
```

```
1
4
0
```

8. 函数调用运算重载

函数对象

函数对象是指重载了 `()` 运算符的对象，此类对象能像函数一样调用，因此叫函数对象。

如：

```
MyClass a;
a(666); // a 必须重载 () 运算符
```

示例

创建一个类 `Translater`，此类的对象会保存一个中间词。然后调用此对象并给出两个参数。结果打印 `第一个参数 中间此 第二个参数` 这样的内容。

```
// filename: op_call.cpp
#include <iostream>

using namespace std;

// 翻译类
class Translater {
    public:
        Translater(const char * s) : middle(s){}
        void operator()(const char *who,const char *what) {
            cout << who << " " << middle << " " << what << endl;
        }
    private:
        string middle;
};

int main(int argc, char * argv[]) {
    Translater chinese("说");
    Translater english("said");
    chinese("张三", "你好");  // 需要重载 () 运算符
    english("lisa", "bye");
    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o op_call op_call.cpp
weimz@mzstudio:~$ ./op_call
张三 说 你好
lisa said bye
```

9. new 和 delete 运算符重载

C++ 语言允许重载 `new` 和 `delete` 运算符，通过重载这些运算符，可以实现内存分配的自主控制。

`new`、`delete` 运算符重载的方法

```
// 动态分配单个对象
void *operator new(size_t sz) { ... }
// 释放单个对象
void operator delete(void *ptr) noexcept { ... }

// 动态分配对象数组
void *operator new[](size_t sz) { ... }
// 释放对象数组
void operator delete[](void *ptr, size_t sz) noexcept { ... }
```

> 关于 `noexcept` 关键字，后面才讲。

示例

重载 `new` 和 `delete` 运算符，将所有使用 `new` 动态创建的对象头使用同一块内存（对象内存共享）。

```
// filename: op_new_delete.cpp
#include <iostream>

using namespace std;

class Complex {
    public:
        Complex(double r=0, double i=0): real(r), image(i){}
    private:
        double real;
        double image;

    // 友元全局函数声明
    friend ostream & operator << (ostream & o, const Complex & c);
};

ostream & operator << (ostream & o, const Complex & c) {
```

```
    o << "(" << c.real << "+" << c.image << "i)";
    return o;
}

// 用于存放动态创建对象的缓冲区
static char mem_buf[4096];

// 重载 new 和 delete 运算符
void * operator new(size_t sz) {
    cout << "operator new 被调用, sz:" << sz << endl;
    return mem_buf;
}
void operator delete(void *ptr) noexcept {
    cout << "operator delete: ptr:" << ptr << endl;
}

// 重载 new[] 和 delete[] 运算符
void * operator new[](size_t sz) {
    cout << "operator new[] 被调用, sz:" << sz << endl;
    return mem_buf;
}
void operator delete[](void *ptr,size_t sz) noexcept {
    cout << "operator delete[]: ptr:" << ptr << "sz:" << sz << endl;
}
int main(int argc, char * argv[]) {
    Complex *pc1, *pc2, *pc3;
    pc1 = new Complex(1, 2);

    cout << "mem_buf:" << &mem_buf << endl;
    cout << "pc1:" << pc1 << " *pc1:" << *pc1 << endl;
    cout << "动态创建 Complex(3, 4);" << endl;
    pc2 = new Complex(3, 4);
    cout << "pc1:" << pc1 << " *pc1:" << *pc1 << endl;
    cout << "pc2:" << pc2 << " *pc2:" << *pc2 << endl;
    cout << "动态创建 new Complex[2];" << endl;
    pc3 = new Complex[2]{Complex(5, 6), Complex(7, 8)};
    cout << "pc1:" << pc1 << " *pc1:" << *pc1 << endl;
    cout << "pc2:" << pc2 << " *pc2:" << *pc2 << endl;
    cout << "pc3:" << pc3 << "pc3[0]:" << pc3[0] << endl;
    delete pc1;
    delete pc2;
    delete[] pc3;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o op_new_delete op_new_delete.cpp
weimz@mzstudio:~$ ./op_new_delete
operator new 被调用, sz:16
mem_buf:0x58ffb9499160
pc1:0x58ffb9499160 *pc1:(1+2i)
动态创建 Complex(3, 4);
```

```
operator new 被调用, sz:16
pc1:0x58ffb9499160 *pc1:(3+4i)
pc2:0x58ffb9499160 *pc2:(3+4i)
动态创建 new Complex[2];
operator new[] 被调用, sz:32
pc1:0x58ffb9499160 *pc1:(5+6i)
pc2:0x58ffb9499160 *pc2:(5+6i)
pc3:0x58ffb9499160pc3[0]:(5+6i)
operator delete: ptr:0x58ffb9499160
operator delete: ptr:0x58ffb9499160
operator delete: ptr:0x58ffb9499160
```

第十三章、异常

异常（Exception）是 C++ 提供的一种处理程序运行时错误的机制，它运行程序时检测错误，并快速返回到上层调用的地方进行错误处理，同时会调用栈上创建对象的析构函数。

C++ 程序在执行时通常用两个流程：**正常流程**和**异常流程**

正常流程的程序在函数调用是通常使用 `return` 语句一层一层的向上返回到函数调用者。而异常流程则可以进入异常状态，在异常状态下略过程序的正常执行的代码快速返回到上层，直至找到能够处理此异常状态的 `try` 语句，程序才重新回到正常状态。

正常状态的程序执行流程

```
fa() ---> fb() ---> fc() ---> fd()
   ^       | ^       |  ^       |
   |       | |       |  |       |
   +-------+ +-------+  +-------+
```

异常状态的程序执行流程

```
fa() ---> fb() ---> fc() ---> fd()
   ^                            |
   |                            |
   +-----------------——---------+
```

异常相关的关键字

```
try    catch    throw
```

1. throw 语句

throw 语句用于抛出错误，让程序进入异常状态（走异常路径），返回到上层调用处等待处理。

语法

```
throw 错误对象;
```

说明

- 一旦执行 throw 语句， throw 语句之后的语句将不再执行，程序进入异常流程并快速返回。
- 错误对象是进入异常流程后传递的异常信息，此异常信息将传递给函数上层的 `try` 语句进行类型匹配和捕获。

示例

```
// filename: throw.cpp
#include <iostream>

using namespace std;

class MyRange{
    public:
        MyRange(int begin, int end, int stp=1): start(begin), stop(end), step(st
p) {}
        // 重载 [] 运算符
        int operator [] (int index) {
            int value = start + index * step;
            if (index < 0)
                throw "索引不能为负数值";
            if (step > 0 && value > stop)
                throw "索引越界";
            else if (step < 0 && value < stop)
                throw "索引越界";

            return value;
        }
    private:
        int start;
        int stop;
        int step;
};

int main(int argc, char * argv[]) {
    MyRange r2(1, 10, 3);  // 生成 1、4、7

    cout << r2[0] << endl;  // 打印 1
    cout << r2[1] << endl;  // 打印 4
    cout << r2[-1] << endl;  // 抛处 const char * 类型的错误
    cout << r2[100] << endl;  // 抛处 const char * 类型的错误

    printf("程序正常退出!\n");
    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o throw throw.cpp
weimz@mzstudio:~$ ./throw
```

```
1
4
terminate called after throwing an instance of 'char const*'
已中止                    （核心已转储） ./throw
weimz@mzstudio:~$
```

从上述运行结果可知，当执行表达式 `r2[-1]` 时，调用成员函数 `operator[]`，此处使用 `throw "索引不能为负数值";` 语句抛出 `const char *` 类型的错误并进入异常状态（进入异常返回流程），由于上层调用者没有使用 `try` 语句接收此异常。此异常一直穿透 `main` 函数。最终程序被异常终止。如果需要让程序接收此错误信息并转为正常状态，则需要使用 `try` 语句进行接收并处理错误。

2. try语句

try语句（也叫 try-catch 语句）用于异常流程的处理，它将尝试捕获异常，并将程序有异常状态转成正常状态（重新进入正常流程）。

语法

```
try {
    可能触发错误的语句。
}
catch (错误类型1 错误对象1) {  // catch 子句可以有多个。
    处理代码（可以为空）。
}
catch (错误类型2 错误对象2) {
    处理代码（可以为空）。
}
catch (...) {  // catch... 子句只能有一个且只能放在最后。
}
```

说明

- `try { }` 中用于执行可能出现异常的语句。
- `catch (错误类型 错误对象)` 根据错误类型依次匹配，如果成功，则被异常被处理，程序恢复正常状态。
- `catch(...)` 用于捕获任何类型的异常，
- `catch (错误类型 错误对象)` 子句可以有0个、一个或多个，但只能有一个最先匹配的类型进行捕获异常并处理。
- `catch(...)` 子句可以有0个或一个且只能放在最后。
- 一旦 `catch` 捕获错误，则异常流程中经过的函数的局部变量的析构函数都将被执行。

示例

```
// filename: try_stat.cpp
#include <iostream>

using namespace std;

class MyRange{
    public:
        MyRange(int begin, int end, int stp=1): start(begin), stop(end), step(st
p) {}
        // 重载 [] 运算符
        int operator [] (int index) {
            int value = start + index * step;
            if (index < 0)
                throw "索引不能为负数值";
            if (step > 0 && value > stop)
                throw "索引越界";
            else if (step < 0 && value < stop)
                throw "索引越界";

            return value;
        }
    private:
        int start;
        int stop;
        int step;
};

int main(int argc, char * argv[]) {
    MyRange r2(1, 10, 3);  // 生成 1、4、7

    try{
        cout << r2[0] << endl;  // 打印 1
        cout << r2[1] << endl;  // 打印 4
        cout << r2[-1] << endl;  // 抛处 const char * 类型的错误
        cout << r2[100] << endl;  // 抛处 const char * 类型的错误
    }
    catch (const char * err) {
        cout << "捕获 const char * 类型的错误: " << err << endl;
        cout << "错误已经处理，程序进入正常状态!" << endl;
    }
    catch (...) {
        cout << "捕获其他类型的错误并处理" << endl;
    }
    printf("程序正常退出!\n");

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o try_stat try_stat.cpp
weimz@mzstudio:~$ ./try_stat
1
4
捕获 const char * 类型的错误: 索引不能为负数值
错误已经处理，程序进入正常状态!
程序正常退出!
```

可见 `r2[-1]` 内抛出的 `const char *` 类型的错误被 `try` 语句收到，并将程序转为正常状态（正常流程）继续向 try 语句后执行。

3. 自定义异常类型

上述示例中我们使用 `const char *` 类型的错误信息来表示 索引超出范围这样的一种错误，这种方法虽然可行，当容易产生混乱，因此 C++ 建议为每一种错误定义一种专门表式此种错误的类型。这样便于区分和处理。

以下自定义 `RangeError` 类型的错误来描述 `MyRange` 类索引超出范围这样类型的错误如下：

```
class RangeError {
    public:
        RangeError(const string &msg) : message(msg) {}
        const string & get_message() const {return message;}
    private:
        string message;
};
```

示例

以下是完整的代码

```
/ filename: custom_excep_type.cpp
#include <iostream>

using namespace std;

class RangeError {
    public:
        RangeError(const string &msg) : message(msg) {}
        const string & get_message() const {return message;}
    private:
        string message;
};

class MyRange{
    public:
        MyRange(int begin, int end, int stp=1): start(begin), stop(end), step(st
```

```
p) {}
        // 重载 [] 运算符
        int operator [] (int index) {
            int value = start + index * step;
            if (index < 0)
                throw RangeError("索引不能为负数值");
            if (step > 0 && value > stop)
                throw RangeError("索引越界");
            else if (step < 0 && value < stop)
                throw RangeError("索引越界");

            return value;
        }
    private:
        int start;
        int stop;
        int step;
};

int main(int argc, char * argv[]) {
    MyRange r2(1, 10, 3);  // 生成 1、4、7

    try{
        cout << r2[0] << endl;  // 打印 1
        cout << r2[1] << endl;  // 打印 4
        cout << r2[-1] << endl;  // 抛处 RangeError 类型的错误
        cout << r2[100] << endl;  // 抛处 RangeError 类型的错误
    }
    catch (const RangeError &err) {
        cout << "捕获 RangeError 类型的错误: " << err.get_message() << endl;
        cout << "错误已经处理，程序进入正常状态!" << endl;
    }
    catch (...) {
        cout << "捕获其他类型的错误并处理" << endl;
    }
    printf("程序正常退出!\n");

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o custom_excep_type custom_excep_type.cpp
weimz@mzstudio:~$ ./custom_excep_type
1
4
捕获 RangeError 类型的错误: 索引不能为负数值
错误已经处理，程序进入正常状态!
程序正常退出!
```

4. 标准异常类型

C++ 标准库中已经定义好了一些异常类型供我们使用。这些异常类型都包含在 `std` 名字空间下。

以下列出这些异常类型以及它们的头文件如下：

- std::logic_error：所有标准异常的基类，提供 what() 虚函数获取错误描述。头文件 `<stdexcept>`。
  - std::invalid_argument：函数接收到无效参数。头文件 `<stdexcept>`。
  - std::domain_error：数学函数的定义域错误。头文件 `<stdexcept>`。
  - std::length_error：试图创建超过最大允许长度的对象（如 std::string）。头文件 `<stdexcept>`。
  - std::out_of_range：访问超出有效范围（如 vector::at() 越界）。头文件 `<stdexcept>`。
- std::runtime_error：表示运行时错误，通常无法在编译期检测。头文件 `<stdexcept>`。
  - std::range_error：内部计算中的范围错误。头文件 `<stdexcept>`。
  - std::overflow_error：算术运算上溢（结果过大）。头文件 `<stdexcept>`。
  - std::underflow_error：算术运算下溢（结果过小）。头文件 `<stdexcept>`。
  - std::regex_error (since C++11)：正则表达式库（）在解析或匹配过程中出错。头文件 `<regex>`。
  - std::system_error (since C++11)：底层系统调用失败时抛出。 头文件`<system_error>`。
    - std::ios_base::failure (since C++11)：输入/输出流操作失败时抛出。 头文件 `<ios>` 或 `<fstream>`。
- std::bad_typeid：对空指针使用 typeid 操作符时抛出。 头文件`<typeinfo>`。
- std::bad_cast：dynamic_cast 操作失败时抛出（针对引用类型）。 头文件 `<typeinfo>`。
- std::bad_weak_ptr (since C++11)：使用无效的 std::weak_ptr 构造 std::shared_ptr 时抛出。头文件`<memory>`。
- std::bad_function_call (since C++11)：调用空的 std::function 对象时抛出。头文件 `<functional>`。
- std::bad_alloc：内存分配失败时抛出（如 new 操作失败）。头文件 `<new>`。
  - std::bad_array_new_length (since C++11)： 数组 new 操作的长度无效（如负数或超大）。 头文件 `<new>`。
- std::bad_exception：用于处理 unexpected_handler 抛出的意外异常。头文件 `<exception>`。

5. noexcept 说明符

C++11 中提供了 `noexcept` 关键字用于异常相关的声明和运算。

`noexcept` 关键字有两个作用：

1. 作为说明符，声明函数（或成员函数）一定不会抛出异常（有利于编译器优化）。
2. 作为运算符，在编译期检查表达式是否可能抛出异常，返回 bool 值。

1. 作为说明符的语法：

```
void 函数名() noexcept {
    // 如果这里真的抛出了异常，程序会直接调用 std::terminate() 终止
}
```

如：

```
void may_throw() { throw 1; }  // 可能抛出异常的函数
void no_throw() noexcept {}  // 一定不抛出异常的函数
```

2. 作为运算符的语法：

```
noexcept(表达式)
```

如：

```
cout << noexcept(may_throw()) << endl;  // 打印：false
cout << noexcept(no_throw()) << endl;  // 打印：true
cout << noexcept(1 + 2) << endl;  // true（基本运算不会抛异常）
```

第十四章、模板

模板（Template）是C++中实现泛型编程的工具。简单的说模板就是使用一套代码编写出与类型无关的函数模板或者类模板。在使用时在根据操作数据的类型用这个函数模板或类模板来生成相应的函数或者类。 模板可以实现代码和类型的脱离，在使用时根据类型来生成和类型相关的代码，实现代码的通用（简称：泛型编程）。

1. 模板原理和 typename

举个例子，如果需要计算两个整数相加返回两个整数和，则我们编写函数如下：

```
int myadd(int x, int y) {
    return x + y;
}
```

如果要实现两个双精度浮点类型的两个数字相加则需要编写函数如下：

```
double myadd(double x, double y) {
    return x + y;
}
```

如果要实现 `Complex` 类型的两个对象相加则需要编写函数如下：

```
Complex myadd(Complex x, Complex y) {
    return x + y;
}
```

实际上我们可以把类型提取出来，我们取名称为 `T`，这样我们的代码就可以写成如下：

```
T myadd(T x, T y) {
    return x + y;
}
```

然后我们在告诉编译器， `T` 是类型参数名，在需要的时候将 T 替换成相关的类型，然后再用替换后的类型生成相应的函数。这样我们就随时可以根据不同的类型来生成不同的函数了。

typename 关键字

typename 关键字 的主要作用是声明一个类型参数，当编译器编译到某个标识符时当做一个类型，稍后在进行替换成相应的类型。

如改写上面的代码如下:

```
template <typename T>
T myadd(T x, T y) {
    return x + y;
}
```

这样 `myadd` 将不再是一个函数，而是一个函数模版，`typename T` 是告诉编译器， `T` 是类型参数。在需要时要进行替换。

这样函数模版将 `T` 替换成 `float` 则生成的函数就是参数和返回值都是 `float` 类型的函数了。

2. 函数模板

函数模板是用来生成函数的模版。它的主要作用是用来编写与类型无关的通用函数模板，在需要时在根据类型生成相应的函数。

函数模板定义的语法

```
template <typename 类型参数名1 [=默认类型1], typename 类型参数名2 [=默认类型2], ...>
返回值  函数模板名(形参列表) { ... }
```

说明

- 类型参数名常用`T` 开头,如:`T` 、 `T1` 、 `T2`
- 类型参数的缺省参数 **默认类型** 可以省略。
- 模板是声明不占用存储空间，一般写在头文件中。
- 在使用函数模板时，C++会在编译阶段自动生成对应的函数代码。
- 关键字 `typename` 可以使用 关键字 `class` 替换（`class` 是早期版本C++使用的关键字，新版本的建议使用 `typename`）。
- 在使用函数模板时，如果调用时不指定模板类型，编译器会根据实参类型自动尝试推导模板的类型。
- 一个 `template <...>` 类型声明的有效范围是一个函数模板（或类模板）的定义语句。

函数模板使用的方法

```
函数模板名<实参类型1, 实参类型2, ...>(实参列表)
```

说明

- 在使用函数模板时，如果直接给出函数模板名代替函数名，不给出**<实参类型列表>**，则编译器会根据实参列表来自动推导模版的实参类型，如果推导失败则会报错处理。

示例

```cpp
// filename: func_template.cpp
#include <iostream>

using namespace std;

template <typename T>
T myadd(T x, T y) {
    return x + y;
}

int main(int argc, char * argv[]) {
    cout << myadd<double>(1.2, 3.4) << endl;
    cout << myadd<int>(5, 6) << endl;
    cout << myadd(7, 8) << endl; // 自动推倒模板 T 的类型为 int

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o func_template func_template.cpp
weimz@mzstudio:~$ ./func_template
4.6
11
15
```

练习

尝试自己编写一个能对任何类型的数据进行冒泡排序的函数模版 `bubble_sort`。

3. 类模板

类模板是用来生成类的模版。它的主要作用是用来编写与类型无关的通用类模板，在需要时在根据类型生成相应的类。

类模板定义的语法

```
template <typename 类型参数名1 [=默认类型1], typename 类型参数名2 [=默认类型2], ...>
class 类模板名 {
    ...
};
```

说明

- 类型参数名常用`T`开头,如:`T`、 `T1`、`T2`
- **类型参数**的缺省参数 **默认类型** 可以省略。
- 模板是声明不占用存储空间，一般写在头文件中。
- 在使用类模板时，C++会在编译阶段自动生成对应的类代码。
- 关键字 `typename` 可以使用 关键字 `class` 替换。
- 一个 `template <...>` 类型声明的有效范围是一个函数模板（或类模板）的定义语句。

函数模板使用的方法

```
类模板名<实参类型1, 实参类型2, ...> 对象名;
```

说明

- 在使用类模板时，必须给出 **<实参类型列表>**，如果不给出实参类型名则使用 **默认类型** 替换。。

以下类是只能存储 100 个整型数据的静态数组封装的类。代码如下：

```
class StaticArray {
    public:
        StaticArray() : data_cnt(0) {}
        void push_back(const int &value) { data[data_cnt] = value; data_cnt++; }
        int size() { return data_cnt; }
        int & operator[] (int index) { return data[index];}
    private:
        int data_cnt;
        int data[100];
};
```

下面我们将 `StaticArray` 改写成 类模板，其中存储的类型用 `T` 代替，改写后的代码如下：

```
template <typename T=int>
class StaticArray {
    public:
        StaticArray() : data_cnt(0) {}
```

```
        void push_back(const T &value) { data[data_cnt] = value; data_cnt++; }
        int size() { return data_cnt; }
        T & operator[] (int index) { return data[index];}
    private:
        int data_cnt;
        T data[100];
};
```

示例

完整的类模板的示例代码如下：

```
// filename: class_template.cpp
#include <iostream>

using namespace std;

template <typename T=int>
class StaticArray {
    public:
        StaticArray() : data_cnt(0) {}
        void push_back(const T &value) { data[data_cnt] = value; data_cnt++; }
        int size() { return data_cnt; }
        T & operator[] (int index) { return data[index];}
    private:
        int data_cnt;
        T data[100];
};

int main(int argc, char * argv[]) {
    // 创建存储 int 类型的静态数组 int_arr
    StaticArray<> int_arr; // 使用 T 的默认值int
    // 创建存储 string 类型的静态数组 str_arr
    StaticArray<string> str_arr;

    int_arr.push_back(11);
    int_arr.push_back(22);
    cout << "int_arr.size(): " << int_arr.size() << endl;
    cout << "int_arr[0]: " << int_arr[0] << endl;
    cout << "int_arr[1]: " << int_arr[1] << endl;

    str_arr.push_back("hello");
    str_arr.push_back("laowei");
    str_arr.push_back("weimingze");
    cout << "str_arr.size(): " << str_arr.size() << endl;
    cout << "str_arr[0]: " << str_arr[0] << endl;
    cout << "str_arr[1]: " << str_arr[1] << endl;
    cout << "str_arr[2]: " << str_arr[2] << endl;

    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~$ g++ -o class_template class_template.cpp
weimz@mzstudio:~$ ./class_template
int_arr.size(): 2
int_arr[0]: 11
int_arr[1]: 22
str_arr.size(): 3
str_arr[0]: hello
str_arr[1]: laowei
str_arr[2]: weimingze
```

实验

尝试编写一个动态数组的类模板。

4. 模板中的非类型参数

在 函数模板 或 类模板 的类型参数声明列表（`template<...>`）中，类型参数列表中不但可以放置 `typename` 定义的类型参数，还可以方式 非类型参数，此类型参数通常是基础的数据类型。

template 的语法

```
template <typename 类型参数名1 [=默认类型1],
          typename 类型参数名2 [=默认类型2],
          ...,
          基础数据类型 变量名1 [=默认值1],
          基础数据类型 变量名2 [=默认值2],
          ...>
```

例如上节我们定义的类模板 `StaticArray` 只能静态存储 100 个某种类型的数据，代码如下：

```
template <typename T=int>
class StaticArray {
    public:
        StaticArray() : data_cnt(0) {}
        void push_back(const T &value) { data[data_cnt] = value; data_cnt++; }
        int size() { return data_cnt; }
        T & operator[] (int index) { return data[index];}
    private:
        int data_cnt;
        T data[100];
};
```

现在我们在模版的 类型参数声明列表（`template <typename T = int>`）中 添加一个非类型参数 `SIZE`，修改后的内容如下

```
template <typename T=int, int SIZE=100>
```

然后我们将静态数组的变量 `data` 的声明常数 `100` 改成 `T data[SIZE];`

这样我们就可以在使用 类模板 `StaticArray` 创建类时动态的改写 非类型参数 `SIZE` 的值来变化该类创建的对象保存数据的个数了。

示例

添加非类型参数的模板示例。

```
// filename: template3.cpp
#include <iostream>

using namespace std;

template <typename T=int, int SIZE=100>
class StaticArray {
    public:
        StaticArray() : data_cnt(0) {}
        void push_back(const T &value) { data[data_cnt] = value; data_cnt++; }
        int size() { return data_cnt; }
        T & operator[] (int index) { return data[index];}
    private:
        int data_cnt;
        T data[SIZE];
};

int main(int argc, char * argv[]) {
    // 创建存储 int 类型的静态数组 int_arr
    StaticArray<> int_arr; // 使用 T 的默认值int,SIZE默认为100
    // 创建存储 string 类型的静态数组 str_arr
    StaticArray<string, 3> str_arr;

    int_arr.push_back(11);
    int_arr.push_back(22);
    cout << "int_arr.size(): " << int_arr.size() << endl;
    cout << "int_arr[0]: " << int_arr[0] << endl;
    cout << "int_arr[1]: " << int_arr[1] << endl;

    str_arr.push_back("hello");
    str_arr.push_back("laowei");
    str_arr.push_back("weimingze");
    cout << "str_arr.size(): " << str_arr.size() << endl;
    cout << "str_arr[0]: " << str_arr[0] << endl;
    cout << "str_arr[1]: " << str_arr[1] << endl;
    cout << "str_arr[2]: " << str_arr[2] << endl;

    return 0;
}
```

编译也运行结果如下：

```
weimz@mzstudio:~$ g++ -o template3 template3.cpp
weimz@mzstudio:~$ ./template3
int_arr.size(): 2
int_arr[0]: 11
int_arr[1]: 22
str_arr.size(): 3
str_arr[0]: hello
str_arr[1]: laowei
str_arr[2]: weimingze
```

练习

编写一个函数模板 `MyMax` 实现给定两个数字或者可以比较大小的数据类型，然后返回数值（也可以是其他的比较依据）较大的一个。并使用此函数模板进行数据的比较。

第十五章、标准模板库

在介绍标准模板库之前我们先来介绍一下**泛型算法**.

泛型算法是一种编程范式，核心思想是编写与类型无关的通用代码，让算法和数据结构等能够适用于不同的数据类型。

函数模板和**类模板**是泛型算法的基础。

泛型算法的核心思想 是一次编写，多次使用。在计算机领域数据结构和算法基本固定，只是操作的数据类型不同，因此可以把这些算法写成**函数模板**或**类模板**封装在一个库中。这样就可以使用这些泛型算法对任意类型进行操作而无需重写这些算法。

1. 标准模板库

C++ 语言将泛型算法封装成标准库供开发者直接使用，这个库就是标准模板库（Standard Template Library，STL）

标准模板库（STL） 是 C++ 标准库的核心组成部分，它是一套通用的数据结构和算法的框架。这套数据结构和算法已经高度优化，基本可以代替任何手搓的数据结构和算法。

标准模板库都包含在 `std` 名字空间中。

标准模板库核心组成（常用）：

- 容器(Container)
- 算法(Algorithms)
- 迭代器(Iterators)

1. 容器（Containers）

容器是用来存储数据的结构，标准模块库中的常用容器分类如下：

1. 序列容器（按顺序存储）：vector（动态数组）、array（定长数组）、list（双向链表）、deque（双端队列）。
2. 关联容器（自动排序，通常是红黑树实现）： set（集合）、map（键值对映射）。
3. 无序关联容器（C++11起，哈希表实现，查找极快）：unordered_set、unordered_map。

2. 算法（Algorithms）

算法是用来对数据进行处理的一些函数集合。其中包含约 100 个通用函数，如排序（sort）、查找（find）、拷贝（copy）、累加（accumulate）等。

这些函数不直接操作容器，而是通过迭代器来操作容器，因此一套算法可以适用于多种容器。

3. 迭代器（Iterators）

迭代器 是一套连接 **容器** 和 **算法** 的 **"指针"**（实质是重载了 * 号运算符的对象）。

迭代器抽象了指针的行为，用来遍历和操作容器中的数据元素。

示例

使用 `vector` 存储整型数据，然后使用 sort 函数模版对 `vector` 内的数据进行排序，然后输出排序后的结果

```
// filename: stl_demo.cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char * argv[]) {
    // 创建一个用于存放整数的动态数组 numbers
    vector<int> numbers = {11, 3, 7, 5};
    numbers.push_back(9);  // 追加一个整数
    // 使用 迭代器遍历 numbers 中的所有整数
    for (auto it = numbers.begin(); it != numbers.end(); it++)
        cout << " " << *it;
    cout << endl;
    // 使用 sort 函数模板对 numbers进行排序
    sort(numbers.begin(), numbers.end());
    // 再次打印 numbers 中的所有整数
    for (auto it = numbers.begin(); it != numbers.end(); it++)
        cout << " " << *it;
    cout << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o stl_demo stl_demo.cpp
weimz@mzstudio:~$ ./stl_demo
```

```
 11 3 7 5 9
 3 5 7 9 11
```

2. vector 动态数组

动态数组（vector） 是顺序存储的一种数据结构，动态数组的特点是使用 索引随机定位的速度快，尾部插入数据的速度相对较快，中间插入数据的速度较慢，当存储空间不足时会动态申请更大的内存来满足存储需求。

`vector` 模板的定义方法如下:

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

`vector` 的第一个类型参数 `T` 是存储数据的类型。第二个参数 `Allocator` 是内存分配器，用来自定义动态分配内存时使用，一般我们使用默认值。

`vector` 创建的动态数组可以使用**统一初始化列表** `{...}` 进行初始化。如:

```
vector<double> vd = {1.2, 3.4};
```

统一初始化列表 通常用来初始化标准模板库中的容器对象。

统一初始化列表的语法格式

```
{对象1, 对象2, 对象3, ...}
```

常用的成员函数

成员函数	说明
`vector`	构造函数。
`~vector`	析构函数。
`operator=`	赋值。
`assign`	等同于 `operator=`。
元素访问相关成员函数	
`at`	访问单个元素，检查边界。
`operator[]`	访问单个元素。
`front`	访问第一个元素
`back`	访问最后一个元素
`data`	访问存储的内存区
容量相关成员函数	
`empty`	判断是否为空
`size`	返回数据元素个数
`max_size`	返回可能存储的最大数据元素个数。
`reserve`	反转数组顺序
`capacity`	返回当前存储区能存储的数据元素的个数
修改相关成员函数	
`clear`	清空数据。
`insert`	插入数据
`emplace(C++11)`	使用构造对象替换
`erase`	删除数据
`push_back`	向后追加单个数据
`pop_back`	删除尾部单个数据
`resize`	修改数据元素个数量
`swap`	交换两个容器
迭代器相关成员函数	

`begin` 或 `cbegin`(C++11)	返回容器数据开始位置的迭代器
`end`或`cend`	回容器数据结束位置的迭代器（最后一个元素的后面）
`rbegin`或`crbegin`(C++11)	返回反向迭代器的起始位置（最后一个元素)
`rend`或 `crend`(C++11)	返回反向迭代器的结束位置（第一个元素的前一位置）

> 以上函数只给出了函数名，以上函数大多数都有重载，具体请查看官方文档.

非成员函数

操作	说明
`operator==`	比较两个容器是否相同。

参考文档

https://en.cppreference.com/cpp/container/vector

示例

```cpp
// filename: vector.cpp
#include <iostream>
#include <vector>

int main()
{
    // 创建一个整数 vector 容器。
    std::vector<int> v = {8, 4, 5, 9};

    // 向容器内加入两个整数
    v.push_back(6);
    v.push_back(9);

    // 覆盖位置为2的数据
    v[2] = -1;

    // 打印 vector
    for (auto it = v.begin(); it != v.end(); it++)
        std::cout << *it << ' ';
```

```
    std::cout << '\n';
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o vector vector.cpp
weimz@mzstudio:~$ ./vector
8 4 -1 9 6 9
```

3. 迭代器

迭代器是一种模拟指针来对顺序或随机容器进行数据访问的工具。使用迭代器可以用统一的方法对标准模板库中的容器进行访问。

C++ 中不同的容器可以通过如下成员函数返回容器的迭代器。虽然不同容器返回的迭代器的类型可能不同，但迭代器的操作方式都是相同。

容器中迭代器相关的成员函数

成员函数	说明
	迭代器相关成员函数
`begin` 或 `cbegin`(C++11)	返回容器数据开始位置的迭代器
`end`或`cend`	回容器数据结束位置的迭代器（最后一个元素的后面）
`rbegin`或`crbegin`(C++11)	返回反向迭代器的起始位置（最后一个元素)
`rend`或 `crend`(C++11)	返回反向迭代器的结束位置（第一个元素的前一位置）

> 上述 `cbegin`、`cend`、`crbegin`、`crend` 这个四个以 `c` 开头的迭代器相关成员函数返回的迭代器具有常属性，不能使用迭代器对容器内的数据进行修改。

迭代器的基本操作

成员函数	说明
`*it`	解引用，引用当前的数据元素。
`=`	赋值（拷贝赋值）。
`==` 或 `!=`	比较两个迭代器是否指向同一位置。
`++it` 或 `it++`	前置/后置自增，让迭代器向后移动到下一个元素。
`--it` 或 `it--`	前置/后置自减，让迭代器向前移动到前一个元素。
`it + n` 或 `n + it`	返回向后移动 n 个位置的新迭代器。
`it - n`	返回向前移动 n 个位置的新迭代器。
`it += n`	it 向后移动 n 个位置，并返回迭代器自身。
`it -= n`	it 向前移动 n 个位置，并返回迭代器自身。
`it2 - it1`	返回两个迭代器的距离（（需指向同一容器，整数值）。
`<` 或 `<=` 或 `>` 或 `>=`	比较两个迭代器的前后顺序（需指向同一容器）。
`it[n]`	下标访问，等同于 `*(it + n)`。
`it->`	访问结构体类（或类）对象的成员。

示例

```cpp
// filename: iterator.cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v = {11, 3, 7, 5, 9};
    auto it = v.begin();
    cout << "*it: " << *it << endl;  // 11
    cout << "*(it+2): " << *(it+2) << endl; // 7
    cout << "it[3]: " << it[3] << endl; // 5
    auto it2 = it + 4;
    cout << "*it2: " << *it2 << endl;  // 9
    cout << "it < it2: " << (it < it2) << endl; // 1
    it2++;
    cout << "it2 == v.end(): " << (it2 == v.end()) << endl; // 1

}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o iterator iterator.cpp
weimz@mzstudio:~$ ./iterator
*it: 11
*(it+2): 7
it[3]: 5
*it2: 9
it < it2: 1
it2 == v.end(): 1
```

4. 算法

C++ 的 STL 库中提供了算法库，它使用泛型编程方法将经典的算法几乎都编辑成为函数模板并高度优化。他能让你用及其简化的代码完成复杂的数据操作。

STL 的算法库在头文件`<algorithm>`中声明，使用时需要包含这个头文件。

STL中的算法函数模版

函数模板名	说明
查找操作	
`all_of`(C++11)	所有元素都为 `true` 返回 `true`，否则返回 `false`。
`any_of`(C++11)	只要有元素为 `true` 返回 `true`，都为 `false` 返回 `false`。
`find`	顺序查找。
`find_if`	条件查找
`binary_search`	二分查找（需有序）
`count`	计数
`count_if`	条件计数
`mismatch`	找第一个不匹配位置
`equal`	判断两个区间是否相
复制操作	
`copy`	拷贝
`copy_if`	条件拷贝
`move`	移动
交换操作	
`swap`	交换
变换操作	
`transform`	使用函数替换
`replace`	替换
`replace_if`	条件替换
通用操作	
`fill`	填充固定值
`generate`	用函数生成值
移除操作	
`remove`	删除
`unique`	去重复

顺序改变操作	
`reverse`	本地调换顺序
`reverse_copy`	返回调换顺序的复制品
排序操作	
`sort`	排序
`stable_sort`	稳定排序
`partial_sort`	部分排序

具体操作详见官方文档：https://en.cppreference.com/cpp/algorithm

示例：

```
// filename: algorithm.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

int main()
{
    vector<int> v = {11, 3, 7, 5, 9};
    // 查找
    auto it = find(v.begin(), v.end(), 5);  // 指向5
    if (it != v.end())
        cout << "finded: " << *it << endl;

    // 排序
    std::sort(v.begin(), v.end());  // v = {3, 5, 7, 9, 11}
    for(auto it = v.begin(); it < v.end(); it++)
        cout << *it << " ";
    cout << endl;

    // 求和（需要 <numeric>）
    int sum = std::accumulate(v.begin(), v.end(), 0);  // 25
    cout << "sum: " << sum << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o algorithm algorithm.cpp
weimz@mzstudio:~$ ./algorithm
```

```
finded: 5
3 5 7 9 11
sum: 35
```

5. 范围 for 语句

范围 for 语句 是 C++11 版本后推出的一种语句，它用于遍历容器、数组或任何可迭代序列中的每个元素。它语法简洁非常容易使用。

范围 for 语句 可以用于遍历任何含有 `begin()` 和 `end()`成员函数并返回迭代器的可迭代的对象。

语法

```
for (类型 变量: 可迭代对象) {
    ... 循环体.
}
```

例如

```
for (int x: myarr) {
    ...
}
// 等同于:
for (auto it = myarr.begin(); it != myarr.end(); ++it) {
    int x = *it;
    ...
}
```

范围 for 语句不但可以用于遍历 `STL` 中的容器，还可以用于遍历数组。

示例

```
// filename: range_for.cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v = {11, 3, 7, 5, 9};
    // 遍历容器对象
    for (int value : v) {
        cout << "value: " << value << endl;
    }
```

```
    // 遍历原生数组
    int arr1[] = {111, 222, 333, 444};
    for (int value: arr1) {
        cout << "value:" << value << endl;
    }

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o range_for range_for.cpp
weimz@mzstudio:~$ ./range_for
value: 11
value: 3
value: 7
value: 5
value: 9
value:111
value:222
value:333
value:444
```

第十六章、类型转换

关于**数据类型转换** 在 C 语言中只有两种类型转换模式：**隐式类型转换**（编译器默认）和**强制类型转换**（使用`()`运算符手动转换）.

在 C++ 语言中，隐式类型转换依旧保留，但不建议使用强制类型转换。

由于 C++ 的结构体和类会存在虚表指针等问题。因此 C 语言的强制类型转换可能会带来不可预测的结果。所以 C++ 语言提供了四种比较安全且语义明确的类型转换方法。

C++ 中的类型转换

1. 静态转换：`static_cast`。
2. 动态转换：`dynamic_cast`。
3. 常量转换：`const_cast`。
4. 重解释转换：`reinterpret_cast`。

1. 静态转换

静态转换（`static_cast`） 是**编译时**对已知类型进行确定性的类型转换。

可以用与静态转换的场景有:

1. 基本数据类型的转换，如：`int` 转 `float`。
2. 具有继承关系的类的指针或引用的转换，通常是向父类方向转换时安全的。也允许向子类方向转换，但开发者自担风险。
3. 将 `void*` 类型的指针转换为具体类型的指针。

静态转换失败（编译报错）的情况有:

1. 无关类型转换（无关指针和无关引用）
2. 移除 const 属性

语法

```
static_cast<新类型>(表达式)
```

示例

```cpp
// filename: static_cast.cpp
#include <iostream>

using namespace std;

class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) { }
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
    public:
        float x;
        float y;
};
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) { }
        void info(void) {  // 子类的此函数也为虚函数
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    int x;
    float y = 3.14;

    x = static_cast<int>(y);
    cout << "x:" << x << endl;
    Point p1(1, 2);
    Circle c1(3, 4, 5);
    Point * pp;
    Circle *pc;

    pp = static_cast<Point*>(&c1);
    pp->info();

    // 可以类型转换，但运行时出错
    pc = static_cast<Circle*>(&p1);
    pc->info(); // 运行结果有错。
    Point & rp = static_cast<Point&>(c1);
    rp.info();
    // 可以类型转换，但运行时出错
    Circle & rc = static_cast<Circle&>(p1);
    rc.info(); // 运行结果有错。
    // x = static_cast<int>(p1); // 报错：无关类型转换
    // const int z = 666;
    // int *pz = static_cast<int*>(&z); // 报错：不能去除const 属性

    return 0;
}
```

编译和运行运行结果如下:

```
weimz@mzstudio:~/old$ g++ -o static_cast static_cast.cpp
weimz@mzstudio:~/old$ ./static_cast
x:3
点(3,4)
圆(1,2,3)
点(3,4)
圆(1,2,3)
```

2. 动态转换

动态转换（`dynamic_cast`）是用于含有虚函数的类创建的对象之间的类型转换，主要是用于按继承链安全向下转型转换或跨层级转换。

语法

```
dynamic_cast<新类型>(表达式)
```

动态转换 的过程是根据虚表进行判断。

`dynamic_cast` 是在运行时进行的类型转换，**动态转换失败的情况有:**

1. 在指针转换时，转换失败返回 `nullptr`。
2. 在引用转换时转换失败会抛出 `std::bad_cast` 类型的异常。

示例

```
// filename: dynamic_cast.cpp
#include <iostream>
#include <exception>

using namespace std;

class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) { }
        // 注意以下函数已经改成虚函数
        virtual void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
    public:
        float x;
        float y;
```

```
};
class Circle : public Point {
    public:
        Circle(float ax, float ay, float radius)
            : Point(ax, ay), r(radius) { }
        void info(void) {  // 子类的此函数也为虚函数
            cout << "圆(" << x << "," << y << "," << r << ")\n";
        }
    public:
        float r; // 半径
};

int main(int argc, char * argv[]) {
    Point p1(1, 2);
    Circle c1(3, 4, 5);
    Point * pp;
    Circle *pc;

    pp = dynamic_cast<Point*>(&c1);
    if (nullptr == pp)
        printf("dynamic_cast<Point*>(&c1): 转换失败!\n");
    else
        pp->info();

    pc = dynamic_cast<Circle*>(&p1);
    if (nullptr == pc)
        printf("dynamic_cast<Circle*>(&p1): 转换失败!\n");
    else
        pc->info();

    try {
        Point & rp = dynamic_cast<Point&>(c1);
        rp.info();
    } catch (std::bad_cast &err) {
        printf("dynamic_cast<Point&>(c1): 转换失败!\n");
    }
    try {
        Circle & rc = dynamic_cast<Circle&>(p1);
        rc.info(); // 运行结果有错。
    } catch (std::bad_cast &err) {
        printf("dynamic_cast<Circle&>(p1): 转换失败!\n");
    }

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~/old$ g++ -o dynamic_cast dynamic_cast.cpp
dynamic_cast.cpp: In function ‘int main(int, char**)’:
dynamic_cast.cpp:41:10: warning: ‘dynamic_cast<class Circle*>(Point p1)’ can nev
er succeed
   41 |     pc = dynamic_cast<Circle*>(&p1);
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
weimz@mzstudio:~/old$ ./dynamic_cast
```

```
圆(3,4,5)
dynamic_cast<Circle*>(&p1): 转换失败!
圆(3,4,5)
dynamic_cast<Circle&>(p1): 转换失败!
```

可见基于有虚函数的类创建的对象进行动态转换时，基类对象转为子类对象的指针或引用时都可能失败，返回空指针（指针）或触发错误（引用）。

3. 常量转换

常量转换（`const_cast`） 是只用于**移除**或**添加** `const`（或`volatile`） 修饰符的转换。它只能改变 `const` 属性。

常量转换不能改变对象的类型本身，只能改变其 `const` 属性。主要用于需要调用不修改对象的非 `const` 常成员函数的场景。

> **注意** 移除 `const` 属性后对原本就是 `const` 定义的对象进行写操作是未定义行为。

语法

```
const_cast<新类型>(表达式)
```

示例

```
// filename: const_cast.cpp
#include <iostream>

using namespace std;

int main(int argc, char * argv[]) {
    const int c = 100;
    int & rc = const_cast<int&>(c);

    cout << "c:" << c << "rc:" << rc << endl;
    rc = 200; // 为定义的行为（结果不确定）。
    cout << "c:" << c << "rc:" << rc << endl;

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~/old$ g++ -o const_cast const_cast.cpp
weimz@mzstudio:~/old$ ./const_cast
c:100rc:100
c:100rc:200
```

4. 重解释转换

重解释转换（`reinterpret_cast`）是将内存二进制位重新解释为另一种类型的转换（如将 `int*` 转为 `char*`，或将 `int` 转为各种指针）。

重解释转换（`reinterpret_cast`）是最危险的转换，他在编译期直接按位拷贝，不进行任何检查。几乎不具备可移植性（依赖硬件字节序等），他是最危险的类型转换。

在使用 reinterpret_cast 转换指针时必须注意内存对齐要求，未对齐的访问会导致未定义行为。C++11 引入了 `alignas` 和 `alignof` 可以尽可能的保证 `reinterpret_cast` 转换时内存的对齐要求。

语法

```
reinterpret_cast<新类型>(表达式)
```

建议应用方式：

1. 日常优先用 `static_cast`。
2. 有继承关系的对象（必须有虚函数）向下转型用 `dynamic_cast`。
3. 迫不得已修改 `const` 属性时用 `const_cast`。
4. 尽量避免 `reinterpret_cast`，它几乎总意味着设计缺陷或极低层级的优化。

第十七章、C++11的新特性

1. Lambda 表达式

lambda表达式（又叫匿名函数），是C++11标准后新增加的用来创建并返回一个函数的一个表达式。它允许程序在运行时创建函数并返回。

lambda表达式用于在需要的地方创建函数并作为表达式返回，避免在其他地方定义在此调用，可以使程序更易于理解。

语法

捕获列表-> 返回类型 { 函数体 }

说明

1. **捕获列表**决定外部变量如何被 lambda 表达式使用（值捕获、引用捕获等）。
2. **形式参数列表** 同普通函数的形式参数列表一样，是用来定义接收数据的形参变量的列表。
3. **->返回类型** 是此函数的返回类型，可省略。如省略则由 return 后的表达式自动推导得到。

捕获列表的写法

写法	说明
`[]`	不使用外部的变量。
`[&]`	引用使用外部的变量（可修改）。
`[=]`	只读使用外部的变量（不可修改）。
`[=,&y]`	混合使用外部的变量。

示例

向 copy_if 模版函数中传递 lambda 表达式创建的函数。

```
// filename: lambda.cpp
#include <iostream>
#include <vector>
#include <algorithm>
```

```cpp
using namespace std;

// 如果是val为负数则返回 true
bool is_negative(const int & val) {
    return val < 0;
}
int main(int argc, char * argv[]) {
    std::vector<int> v{3, -4, 2, -8, 15, 267};
    std::vector<int> v_nega1; // 存放负数的容器
    std::vector<int> v_nega2;

    // 打印 v
    for(auto val: v)
        cout << val << " ";
    cout << endl;

    // 使用 is_negative 函数筛选出所有的负数
    copy_if(v.begin(),
            v.end(),
            back_inserter(v_nega1), // 末尾插入
            is_negative);

    // 打印 v_nega1;
    cout << "筛选出来负数有:" << endl;
    for(auto val: v_nega1)
        cout << val << " ";
    cout << endl;
    // 使用 lambda 表达式 筛选出所有的负数
    copy_if(v.begin(),
            v.end(),
            back_inserter(v_nega2), // 末尾插入
            [](const int &val)->bool{return val < 0;});
    // 打印 v_nega2;
    cout << "筛选出来负数有:" << endl;
    for(auto val: v_nega2)
        cout << val << " ";
    cout << endl;

    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~$ g++ -o lambda lambda.cpp
weimz@mzstudio:~$ ./lambda
3 -4 2 -8 15 267
筛选出来负数有:
-4 -8
筛选出来负数有:
-4 -8
```

示例2

在 Lambda 表达式内使用外部的变量。

```
// filename: lambda2.cpp
#include <iostream>
#include <algorithm>

using namespace std;

int main(int argc, char * argv[]) {
    int x = 100;
    int y = 200;

    auto f1 = [=, &y](int a, int b) {
        cout << x + a + b + y << endl;
        y = 6666;
        return x;
    };

    cout << f1(10, 20) << endl;
    cout << "x:" << x << endl;
    cout << "y:" << y << endl;

  return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o lambda2 lambda2.cpp
weimz@mzstudio:~$ ./lambda2
330
100
x:100
y:6666
```

可见通过捕获列表 `[=,&y]` 在 lambda 表达式内可以使用 `x` 和 `y` 这两个外部的变量，并可以在 lambda 表达式内修改 `y` 这个变量。

练习

将下列数组按绝对值排序。

```
std::vector<int> v{3, -4, 2, -8, 15, 26;
```

按绝对值升序排序结果。

```
2, 3, -4, -8, 15, 26;
```

2. 左值和右值

左值 是指可以放置在赋值运算符左侧的值，**右值**是只能放在赋值运算符右侧的值。左值和右值是表达式（非变量）的基本属性。

左值 是具有内存地址的表达式，它存在于内存中，可以用取地址运算符（`&`）来获取内存地址。如：变量、数组、指针解引用等。如：

```
int x = 100;  // x 是左值
int *px;
px = &x;   // px 是左值
```

右值 是不具有内存地址的表达式，它是临时的、即将销毁的值，它没有持久的内存地址，不能用取地址运算符（`&`）来获取内存地址。如：字面值（`88`），算数表达式的返回值（`a+b`）、函数返回的临时对象等。如：

```
int x = 100;  // 100 是右值（字面值）
int y = x + 1;  // x+1 返回的值（对象）是右值（临时值）
```

现代 C++（C++11 起）的进一步把右值分成了两类（为了更好的优化程序）：

1. 纯右值：就是传统的右值，比如 `100`、`true`、`x + y`、返回的临时对象的函数调用。
2. 将亡值：即将被移动的资源。比如 `std::move(a)`的返回值（后面才讲），它像个右值，但实质是绑定一个即将被窃取资源的左值。

为什么要区分左值和右值?

区分左值和右值是为了优化程序的需要，区分左值和右值可以有效的根据左值和右值调用不同的重载函数，避免不必要的深拷贝（窃取将亡值的资源），以提升程序运行的性能。

3. 右值引用

右值引用是 `C++11` 引入的一种新的引用类型，用 `T &&` 表示，它用于代替使用 `const` 类型的左值引用（`const T &`）来绑定右值，给右值取一个别名，延长右值的生命周期，同时可以**窃取**右值的资源，从而避免复杂的深拷贝来提高执行效率。

语法

```
类型 && 引用变量名 = 右值;
```

示例

```
#include <iostream>

using namespace std;

int main(int argc, char * argv[]) {
    int x = 100;
    int &lr_x = x; // 左值引用
    // int &lr_e = 200; // 错误，左值引用不能绑定左值
    const int &clr = 300; // const 左值引用可以绑定右值（延长生命周期）
    int && rr_1 = 400; // 右值引用绑定右值
    int && rr_2 = x + 1; // 右值引用绑定右值
    // int && rr_3 = x; // 错误右值引用不能绑定左值

    return 0;
}
```

移动语义 `std::move`

`std::move` 是 C++11 中一个纯粹的强制类型转换（`static_cast`），作用是把左值类型无条件转换为右值类型，从而 **告诉** 编译器，此`std::move` 函数返回的类型是右值，在函数重载时最匹配的类型是右值引用。

用法:

```
std::move(obj)  // 可以将一个左值（或右值）将其转化为右值。
```

函数重载决议优先级顺序

1. 精确匹配
2. 左值/右值引用匹配
3. 常量转换
4. 其他转换

示例

```
// filename: override.cpp
#include <iostream>
```

```
using namespace std;


void fx(int &x) {
    cout << "fx(int &x):" << x << endl;
}
void fx(const int &x) {
    cout << "fx(const int &x):" << x << endl;
}
void fx(int &&x) {
    cout << "fx(int &&x):" << x << endl;
}
void fx(double x) {
    cout << "fx(double x):" << x << endl;
}

int main(int argc, char * argv[]) {
    // 右值
    int i = 100; // r是右值
    int &ri = i;  // ri左值引用
    const int & cri = i; //  cri 是带有const 修饰属性的左值引用
    fx(i);  // 调用 void fx(int &x);
    fx(ri);  // 调用 void fx(int &x);
    fx(cri);  // 调用 void fx(const int &x);
    fx(200);  // 调用 void fx(int &&x);
    fx(cri+1);  // 调用 void fx(int &&x);
    fx(1+2);  // 调用 void fx(int &&x);
    fx(std::move(i));  // 调用 void fx(int &&x);

    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -o override override.cpp
weimz@mzstudio:~$ ./override
fx(int &x):100
fx(int &x):100
fx(const int &x):100
fx(int &&x):200
fx(int &&x):101
fx(int &&x):3
fx(int &&x):100
```

4. 移动构造和移动赋值函数

移动构造函数和移动赋值函数 是 C++11 引入的语法，它们的主要作用是高效地转移资源所有权，避免不必要的深度拷贝。

语法

```
class 类名 {
    public:
        类名(类名&& other) noexcept {...}  // 移动构造函数
        类名& operator=(类名&& other) {...}  // 移动赋值函数
};
```

说明:

1. 移动构造函数是在使用一个右值来创建新对象时调用。
2. 移动赋值成员函数是在使用一个右值来对一个已有的对象赋值时调用。

示例

编写一个动态数组的类 DynamicArray 用来存储整数数据。此数组使用push_back 追加数据，并可可以实现数组的拼接等操作。

```
// filename: move.cpp
#include <iostream>

using namespace std;

#define MIN_SIZE (8)
class DynamicArray {
    public:
        // 初始化动态数组
        DynamicArray() : data(NULL), count(0), max_count(MIN_SIZE) {
                cout << "无参数构造函数被调用" << endl;
            }
        DynamicArray(const DynamicArray &src)
            :data(NULL), count(src.count), max_count(src.max_count) {
                cout << "拷贝构造函数被调用" << endl;
                if (src.data) {  // 如果src有数据才分配内存
                    data = new int[max_count];
                }
                // 复制数据
                for (int i = 0; i < count; i++) {
                    data[i] = src.data[i];
                }
            }
        DynamicArray & operator = (const DynamicArray &src) {
            cout << "拷贝赋值函数被调用" << endl;
            if (0 == src.count) { // 没有数据需要复制
                if (data) {
                    delete [] data;
                    data = NULL;
                }
                count = 0;
            } else { // 需要复制
```

```
            if (NULL == data)
                data = new int[src.max_count];
            count = src.count;
            max_count = src.max_count;
            for (int i = 0; i < count; i++) {
                data[i] = src.data[i];
            }
        }
        return *this;
    }
    DynamicArray(DynamicArray &&src)
        :data(src.data), count(src.count), max_count(src.max_count) {
            cout << "移动构造函数被调用" << endl;
            src.data = NULL;
            src.count = 0;
        }
    DynamicArray & operator = (DynamicArray &&src) {
        cout << "移动赋值函数被调用" << endl;
        if (data)
            delete [] data;
        data = src.data;
        src.data = NULL;
        count = src.count;
        max_count = src.count;
        return *this;
    }
    ~DynamicArray() {
        if (data) {
            delete [] data;
        }
    }
    // 2.  添加数据
    void push_back(const int & x){
        // 需要重新分配内存
        if (data == NULL)  {  // 一定要分配内存
            data = new int[max_count];
        } else if (count >= max_count) { // 需要扩容
            int * temp = data;
            data = new int[max_count * 2];
            for (int i = 0; i < count; i++)
                data[i] = temp[i];
            delete [] temp;
        }
        data[count] = x;
        count++;
    }

    DynamicArray operator+(const DynamicArray &r) {
        DynamicArray tmp;
        tmp.count = count + r.count; // 返回对象的元素个数
        // 计算 最大容量（最大容量是8的n次方）
        tmp.max_count = MIN_SIZE;
        while(tmp.max_count < tmp.count)
            tmp.max_count *= 2;
        tmp.data = new int[tmp.max_count];
        // 复制每一个整数值
```

```
            int i;
            for (i = 0; i < count; i++) {//复制this数据
                tmp.data[i] = data[i];
            }
            //复制r的数据
            for (int j = 0; j < r.count; j++) {
                tmp.data[i+j] = r.data[j];
            }
            return tmp;
        }
        int size(void) { // 返回大小
            return count;
        }
    private:
        int * data;  // 用来保存数据的指针
        int count; // 用来记录数据的个数。
        int max_count;  // 用来记录当前内存能存储的最大数据
    private:
        friend ostream & operator<<(ostream &o, const DynamicArray & da);
};

ostream & operator<<(ostream &o, const DynamicArray & da) {
    o << "[";
    for (int i = 0; i < da.count; i++) {
        o << da.data[i];
        if ( i < da.count -1)
            o << ", ";
    }
    o << "]";
    return o;
}
int main(int argc, char * argv[]) {
    DynamicArray d1;
    for (int i = 11; i <= 33; i+= 11)
        d1.push_back(i);
    cout << d1 << endl;
    // 调用拷贝构造函数
    DynamicArray d2(d1);
    cout << d2 << endl;
    // 调用移动构造函数
    DynamicArray d3(d1 + d2);
    cout << d3 << endl;

    DynamicArray d4;
    // 调用移动赋值函数
    d4 = d1 + d2;
    cout << d4 << endl;

    return 0;
}
```

为了能看到效果，避免移动构造函数调用被优化，编译时可以使用如下的 g++ 选项

```
g++ -fno-elide-constructors -o xxx xxx.cpp
```

编译和运行结果如下:

```
weimz@mzstudio:~$ g++ -fno-elide-constructors -o move move.cpp
weimz@mzstudio:~$ ./move
无参数构造函数被调用
[11, 22, 33]
拷贝构造函数被调用
[11, 22, 33]
无参数构造函数被调用
移动构造函数被调用
移动构造函数被调用
[11, 22, 33, 11, 22, 33]
无参数构造函数被调用
无参数构造函数被调用
移动构造函数被调用
移动赋值函数被调用
[11, 22, 33, 11, 22, 33]
```

一般，一个类内如果有指针成员变量，则此类在编写时一般多要准守**三五法则**。

三五法则

- 三: 拷贝构造函数、拷贝赋值函数、析构函数，在需要深拷贝时需要编写。
- 五：在三的基础上加上移动构造函数和移动赋值函数，可以提高程序的运行效率。

5. 智能指针

智能指针（Smart pointers） 是 C++11 的新特性，他使用类模板封装了传统的指针并重载了 `->` 和 `*` 运算符，让其可以向普通指针一样进行操作。但他可以在智能指针被销毁时自动调用析构函数来对所指向的对象进行自动的 `delete` 操作来释放**堆**上创建的对象来实现自动的内存管理。

智能指针与传统指针相比有两个显著的作用：

1. 不会因忘记 `delete` 导致的内存（和资源）泄漏。
2. 不会因异常发生时跳过 `delete` 导致的内存（和资源）泄漏。

C++11 的智能指针共有三种

1. 独占指针（`unique_ptr`）。
2. 共享指针（`shared_ptr`）。
    - 使用引用计数的方式来管理指向的对象。

3. 弱指针（`weak_ptr`）。
    - 弱指针智能用和 shared_ptr 配合使用，不参与引用计数，避免形成指针环链。

智能指针在 `<memory>` 头文件定义，使用是需要使用 `#include <memory>` 进行包含。

5.1 独占指针

独占指针（`unique_ptr`）是独占一个动态分配对象的所有权的智能指针。此指针不能再赋值给其他智能指针，即统一时间，只有一个 `unique_ptr` 指向这个对象。

独占指针（`unique_ptr`） 不允许拷贝，即独占指针没有拷贝构造函数和拷贝赋值函数，但独占指针支持移动语义（`std::move`）。移动后，原指针变为空独占指针。

独占指针的运行效率非常高，几乎同普通指针一样快。

语法

```
// 创建独占指针并直接管理单个对象
std::unique_ptr<类> ptr(new 类(构造函数实参1, 构造函数实参2, ...));

// 创建独占指针并直接管理多个对象的数组
std::unique_ptr<类[]> ptr(new 类[元素个数]{...});

// 创建空独占指针，后面需要使用 reset 成员函数为其赋值
std::unique_ptr<类> ptr;
```

常用成员函数

成员函数	说明
`reset`	重新设定管理对象指针，旧的管理对象会被释放(`delete`).
`release`	释放管理对象，返回管理对象的指针。同时将此独占指针置空。
`swap`	交换两个同类型智能指针管理的对象。
`get`	返回管理对象的普通指针。
`operator bool`	判断是否为空指针。

示例

创建独占指针 `ptr1` 让其指向 Point 类型的对象，当 `ptr1` 销毁时会自动释放 `ptr1` 指向的对象。

创建空独占指针 `ptr2` 后面再使用 `reset` 成员函数指向 新的对象。

创建空独占指针 `ptr3` 管理对象的数组。

```
// filename: unique_ptr.cpp
#include <iostream>
#include <memory>

using namespace std;

class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        ~Point() {
            cout << "~Point(" << x << "," << y << ")\n";
        }
        // 定义为虚函数
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
    public:
        float x;
        float y;
};


int main(int argc, char * argv[]) {
    std::cout << "-- 进入 main 函数 --" << std::endl;
    {
        std::cout << "- 进入内部作用域 -" << std::endl;

        unique_ptr<Point> ptr1(new Point(1.1, 1.2));
        unique_ptr<Point> ptr2;
        // ptr2 = ptr1; // 报错，不允许
        ptr2.reset(new Point(2.1, 2.2));
        unique_ptr<Point[]> ptr3(new Point[2]{Point(3.1, 3.2), Point(4.1, 4.2)})
;

        // 使用 -> 访问成员
        ptr1->info();
        ptr2->info();
        // 使用 * 解引用
        (*ptr1).info();
        (*ptr2).info();

        cout << "- 即将离开内部作用域 -" << endl;
    } // 离开作用域，ptr1 自动调用析构函数，内存被释放
    cout << "-- 已离开内部作用域 --" << endl;
```

```
    return 0;
}
```

编译和运行结果如下

```
weimz@mzstudio:~/old$ g++ -o unique_ptr unique_ptr.cpp
weimz@mzstudio:~/old$ ./unique_ptr
-- 进入 main 函数 --
- 进入内部作用域 -
Point(1.1,1.2)
Point(2.1,2.2)
Point(3.1,3.2)
Point(4.1,4.2)
点(1.1,1.2)
点(2.1,2.2)
点(1.1,1.2)
点(2.1,2.2)
- 即将离开内部作用域 -
~Point(4.1,4.2)
~Point(3.1,3.2)
~Point(2.1,2.2)
~Point(1.1,1.2)
-- 已离开内部作用域 --
```

注意

独占指针 一旦绑定管理对象，则不要使用原指针释放对象，否则可能导致不可预期的结果。如：

```
Point * p = new Point(1.2, 3.4);
unique_ptr<Point> ptr1(p);
delete p;  // 释放对象
// 之后销毁 ptr1 时就会操作野指针造成重复释放。
```

5.2 共享指针

共享指针（`shared_ptr`）是使用引用计数的方式来管理指向的对象的智能指针，它允许多个智能指针来管理同一个动态创建的对象。只有多个指针都销毁的情况下才会销毁（`delete`）被管理的对象。

共享指针原理

当用第一次用**共享指针**来管理一个对象时，此共享指针会动态创建一个整数的内存区域并将其初始化为`1`，这个整数对象用来记录指向当前被管理对象的智能指针的个数，此整数由所有管理此对

象的智能指针所共享，并由最后管理此对象的智能指针所释放。这个整数用来**引用计数**，即记录指向此被管理对象的智能指针的个数。

语法

```
// 创建共享指针并直接管理单个对象
std::shared_ptr<类> ptr(new 类(构造函数实参1, 构造函数实参2, ...));

// 创建共享指针并直接管理多个对象的数组
std::shared_ptr<类[]> ptr(new 类[元素个数]{...});

// 创建空共享指针
std::shared_ptr<类> ptr;
```

常用成员函数

成员函数	说明
`reset`	重新设定管理对象指针，旧的管理对象的引用计数减一，新的管理对象的引用计数加一。
`swap`	交换两个同类型智能指针管理的对象。
`get`	返回管理对象的普通指针。
`use_count`	返回管理对象的引用计数。
`operator bool`	判断是否为空指针。
`operator =`	修改共享指针的指向。

示例

```
// filename: shared_ptr.cpp
#include <iostream>
#include <memory>

using namespace std;

class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        ~Point() {
            cout << "~Point(" << x << "," << y << ")\n";
```

```
        }
        // 定义为虚函数
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
    public:
        float x;
        float y;
};

shared_ptr<Point> get_point() {
    shared_ptr<Point> ptr(new Point(5.1, 5.2)); // 引用计数为1
    return ptr;// 此 ptr 会作为调用的表达式返回。
}

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

    std::cout << "-- 进入 main 函数 --" << std::endl;
    {
        std::cout << "- 进入内部作用域 -" << std::endl;

        shared_ptr<Point> ptr1(new Point(1.1, 1.2));  // Point(1.1, 1.2)引用计数为
1
        shared_ptr<Point> ptr2(new Point(2.1, 2.2));
        ptr2 = ptr1; // Point(2.1, 2.2)被释放Point(1.1, 1.2) 引用计数为2
        shared_ptr<Point[]> ptr3(new Point[2]{Point(3.1, 3.2), Point(4.1, 4.2)})
;
        shared_ptr<Point> ptr4;

        ptr4 = get_point();
        // 使用 -> 访问成员
        ptr1->info();
        ptr2->info();
        ptr4->info();
        // 使用 * 解引用
        (*ptr1).info();
        (*ptr2).info();
        (*ptr4).info();
        cout << "ptr1的引用计数:" << ptr1.use_count() << endl;
        cout << "ptr2的引用计数:" << ptr2.use_count() << endl;
        cout << "ptr3的引用计数:" << ptr3.use_count() << endl;
        cout << "ptr4的引用计数:" << ptr4.use_count() << endl;

        cout << "- 即将离开内部作用域 -" << endl;
    } // 离开作用域，ptr1 自动调用析构函数，内存被释放
    cout << "-- 已离开内部作用域 --" << endl;
    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~/old$ g++ -o shared_ptr shared_ptr.cpp
weimz@mzstudio:~/old$ ./shared_ptr
```

```
-- 进入 main 函数 --
- 进入内部作用域 -
Point(1.1,1.2)
Point(2.1,2.2)
~Point(2.1,2.2)
Point(3.1,3.2)
Point(4.1,4.2)
Point(5.1,5.2)
点(1.1,1.2)
点(1.1,1.2)
点(5.1,5.2)
点(1.1,1.2)
点(1.1,1.2)
点(5.1,5.2)
ptr1的引用计数:2
ptr2的引用计数:2
ptr3的引用计数:1
ptr4的引用计数:1
- 即将离开内部作用域 -
~Point(5.1,5.2)
~Point(4.1,4.2)
~Point(3.1,3.2)
~Point(1.1,1.2)
-- 已离开内部作用域 --
```

5.3 弱指针

弱指针（`weak_ptr`） 是用来同**共享指针**（`shared_ptr`）配合使用的智能指针，它不参与引用计数，目的是避免形成指针环链。

我们先来看一个使用共享指针导致形成指针环链，最终被管理对象无法释放的例子。

```cpp
// filename: ptr_circle.cpp
#include <iostream>
#include <vector>
#include <memory>

using namespace std;

class Point {
    public:
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        ~Point() {
            cout << "~Point(" << x << "," << y << ")\n";
        }
        // 定义为虚函数
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
```

```
    public:
        // 使用共享指针指向下一个点
        shared_ptr<Point> next_point; // 指向图形的下一个点
    public:
        float x;
        float y;
};

vector<shared_ptr<Point>> get_triangle() {
    vector<shared_ptr<Point>> points;
    shared_ptr<Point> ptr1(new Point(0.0, 1.0)); // 引用计数为1
    shared_ptr<Point> ptr2(new Point(1.0, 0.0)); // 引用计数为1
    shared_ptr<Point> ptr3(new Point(1.0, 1.0)); // 引用计数为1
    ptr1->next_point = ptr2;  // 共享指针引用计数变为2
    ptr2->next_point = ptr3;
    ptr3->next_point = ptr1;
    points.push_back(ptr1);
    points.push_back(ptr2);
    points.push_back(ptr3);
    return points;
}

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

    std::cout << "-- 进入 main 函数 --" << std::endl;
    {
        std::cout << "- 进入内部作用域 -" << std::endl;

        vector<shared_ptr<Point>>  ptr_shape = get_triangle();

        cout << "- 即将离开内部作用域 -" << endl;
    }
    // 此时所有的 被管理的对象的引用计数为 1,被管理对象不会被释放
    cout << "-- 已离开内部作用域 --" << endl;
    return 0;
}
```

编译和运行结果如下：

```
weimz@mzstudio:~/old$ g++ -o ptr_circle ptr_circle.cpp
weimz@mzstudio:~/old$ ./ptr_circle
-- 进入 main 函数 --
- 进入内部作用域 -
Point(0,1)
Point(1,0)
Point(1,1)
- 即将离开内部作用域 -
-- 已离开内部作用域 --
```

可见上述 `Point` 类虽然使用了共享指针，但 `Point` 类创建的三个对象形成了环链。对象没有释放。内存丢失。解决这一问题的方式是对可能形成环链的对象使用弱指针，避免增加引用计数。即上述 `Point`类的的成员变量 `next_point` 使用弱指针可以解决这一问题。

语法

```
// 创建弱指针并直接管理单个对象
std::weak_ptr<类> ptr(对象);

// 创建空弱指针
std::weak_ptr<类> ptr;
```

常用成员函数

成员函数	说明
`reset`	重新设定管理对象指针。
`swap`	交换两个同类型智能指针管理的对象。
`operator =`	修改弱指针的指向。
`use_count`	返回管理对象的共享指针的引用计数（弱指针没有引用计数）。
`operator bool`	判断是否为空指针。
`expired`	检查引用计数是否已经是 `0`，即此弱指针已经无效，无效返回 `true`。
`lock`	返回一个共享指针（需要用变量绑定），引用计数增加1，确保对象不会被销毁。

示例

修改 `Point` 的成员变量 `next_point` 为 弱指针

```
// filename: weak_ptr.cpp
#include <iostream>
#include <vector>
#include <memory>

using namespace std;

class Point {
    public:
```

```
        Point(float ax=0, float ay=0):x(ax), y(ay) {
            cout << "Point(" << x << "," << y << ")\n";
        }
        ~Point() {
            cout << "~Point(" << x << "," << y << ")\n";
        }
        // 定义为虚函数
        void info(void) {
            cout << "点(" << x << "," << y << ")\n";
        }
    public:
        // 使用弱指针指向下一个点
        weak_ptr<Point> next_point; // 指向图形的下一个点
    public:
        float x;
        float y;
};

vector<shared_ptr<Point>> get_triangle() {
    vector<shared_ptr<Point>> points;
    shared_ptr<Point> ptr1(new Point(0.0, 1.0)); // 引用计数为1
    shared_ptr<Point> ptr2(new Point(1.0, 0.0)); // 引用计数为1
    shared_ptr<Point> ptr3(new Point(1.0, 1.0)); // 引用计数为1
    ptr1->next_point = ptr2;  // 弱指针引用计数不变
    ptr2->next_point = ptr3;
    ptr3->next_point = ptr1;
    points.push_back(ptr1);
    points.push_back(ptr2);
    points.push_back(ptr3);
    return points;
}

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

    std::cout << "-- 进入 main 函数 --" << std::endl;
    {
        std::cout << "- 进入内部作用域 -" << std::endl;

        vector<shared_ptr<Point>>  ptr_shape = get_triangle();

        cout << "- 即将离开内部作用域 -" << endl;
    }
    // 此时所有的 被管理的对象的引用计数为 1,被管理对象不会被释放
    cout << "-- 已离开内部作用域 --" << endl;
    return 0;
}
```

编译和运行结果如下:

```
weimz@mzstudio:~/old$ g++ -o weak_ptr weak_ptr.cpp
weimz@mzstudio:~/old$ ./weak_ptr
-- 进入 main 函数 --
- 进入内部作用域 -
Point(0,1)
```

```
Point(1,0)
Point(1,1)
- 即将离开内部作用域 -
~Point(0,1)
~Point(1,0)
~Point(1,1)
-- 已离开内部作用域 --
```

可见弱指针没有对管理对象的引用计数做加一操作，致使共享指针已经可以将引用计数减为零，达到成功释放被管理对象的目的。