第一章 C++与C

本章主要讲解C++相较于C一些独有的比较重要的知识点。

C++源文件后缀名.cc/.cpp，头文件后缀名.hh/.hpp

安装g++命令：**sudo apt install g++**

编译命令 g++ 文件名.cc/.cpp [-o name]

设置代码预设片段

```
ray@ubuntu:~$ cd .vim/plugged/prepare-code/snippet/
ray@ubuntu:~/.vim/plugged/prepare-code/snippet$ ls
snippet.c  snippet.cc  snippet.cpp  snippet.go  snippet.h  snippet.py  snippet.sh
ray@ubuntu:~/.vim/plugged/prepare-code/snippet$ vim snippet.cc
```

首先从C++的hello,world程序入手，来认识一下C++语言

```
1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char * argv[]){
5      cout << "hello,world" << endl;
6      return 0;
7  }
```

（1）iostream是C++的头文件，为什么没有后缀？—— 模板阶段再作讲解

（2）using namespace std是什么含义？—— 命名空间的使用

（3） cout << "hello,world" << endl; 实现了输出hello,world的功能，如何理解这行代码？—— cout的使用

命名空间

为什么要使用命名空间

一个大型的工程往往是由若干个人独立完成的，不同的人分别完成不同的部分，最后再组合成一个完整的程序。由于各个头文件是由不同的人设计的，有可能在不同的头文件中用了相同的名字来命名所定义的类或函数，这样在程序中就会出现名字冲突。不仅如此，有可能我们自己定义的名字会与C++库中的名字发生冲突。

名字冲突就是在同一个作用域中有两个或多个同名的实体，为了解决命名冲突，C++中引入了**命名空间**，所谓命名空间就是一个可以由用户自己定义的作用域，在不同的作用域中可以定义相同名字的变量，互不干扰，系统能够区分它们。

C语言中避免名字冲突，只能进行起名约定

```
int hw_cpp_tom_num = 100;
int wd_cpp_bob_num = 200;
```

什么是命名空间

命名空间又称为名字空间，是程序员命名的内存区域，程序员根据需要指定一些有名字的空间域，把一些全局实体分别存放到各个命名空间中，从而与其他全局实体分隔开。通俗的说，每个名字空间都是一个名字空间域，存放在名字空间域中的全局实体只在本空间域内有效。名字空间对全局实体加以域的限制，从而合理的解决命名冲突。

C++中定义命名空间的基本格式如下：

```
namespace wd
{
int val1 = 0;
char val2;
}// end of namespace wd
```

在声明一个命名空间时，大括号内不仅可以存放变量，还可以存放以下类型：

变量、常量、函数、结构体、引用、类、对象、模板、命名空间等，它们都称为实体

（1）请尝试定义命名空间，并在命名空间中定义实体。

（2）命名空间中的实体如何使用呢?

命名空间的使用方式

命名空间一共有三种使用方式，分别是using编译指令、作用域限定符、using声明机制。

1. 作用域限定符

   每次要使用某个命名空间中的实体时，都直接加上**作用域限定符::**，例如：

```
namespace wd
{
int number = 10;
void display()
{
    //cout,endl都是std空间中的实体，所以都加上'std::'命名空间
```

```
    std::cout << "wd::display()" << std::endl;
}
}//end of namespace wd

int main(void)
{
    std::cout << "wd::number = " << wd::number << endl;
    wd::display();
    return 0;
}
```

好处：准确，只要命名空间中确实有这个实体，就能够准确调用（访问）

坏处：繁琐

2. using编译指令

我们接触的第一个C++程序基本上都是这样的，其中std代表的是标准命名空间。

```
#include <iostream>
using namespace std; //using编译指令

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

其中第二行就使用了using编译指令。如果一个名称空间中有多个实体，使用using编译指令，就会把该空间中的所有实体一次性引入到程序之中；对于初学者来说，如果对一个命名空间中的实体并不熟悉时，直接使用这种方式，有可能还是会造成名字冲突的问题，而且出现错误之后，还不好查找错误的原因，比如下面的程序就会报错，当然该错误是人为造成的。

```
#include <iostream>
using namespace std;
double cout()
{
return 1.1;
}
int main(void)
{
cout();
return 0;
}
```

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

void xout(){ //发生冲突，不能用cout
    int num = 1;
    printf("%d\n",num);
}

void test0(){
    cout << "hello" << endl;
}
```

3. using声明机制

using声明机制的作用域是从using语句开始，到using所在的作用域结束。要注意，在同一作用域内用using声明的不同的命名空间的成员不能有同名的成员，否则会发生重定义。

```cpp
#include <iostream>
using std::cout;
using std::endl;
namespace wd
{
int number = 10;
void display()
{
cout << "wd::display()" << endl;
}
}//end of namespace wd
using wd::number;
using wd::display;
int main(void)
{
    cout << "wd::number = " << number << endl;
    wd::display();
    return 0;
}
```

在这三种方式之中，我们推荐使用的就是第三种，需要哪个实体的时候就引入到程序中，不需要的实体就不引入，尽可能减小犯错误的概率。

```
//不是一次性引入所有实体
//用什么声明什么
using wd::num;
using wd::func;
void test0(){
    cout << num << endl;
    func();
}
```

命名空间的嵌套使用

类似于文件夹下还可以建立文件夹，命名空间中还可以定义命名空间。那么内层命名空间中的实体如何访问呢？尝试一下

```
namespace wd
{
int num = 100;

void func(){
    cout << "func" << endl;
}


namespace cpp
{
int num = 200;

void func(){
    cout << "cpp::func" << endl;
}
}//end of namespace cpp

}//end of namespace wd
```

```
void test0(){
    cout << wd::cpp::num << endl;
    wd::cpp::func();
}

void test1(){
    using namespace wd::cpp;
    cout << num << endl;
    func();
}

void test2()
{
    using wd::cpp::num;
    using wd::cpp::func;
    cout << num << endl;
    func();
}
```

匿名命名空间

命名空间还可以不定义名字，不定义名字的命名空间称为匿名命名空间（简称匿名空间），其定义方式如下：

```
namespace {
int val1 = 10;
void func();
}//end of anonymous namespace
```

```cpp
int num = 100;

//可以理解为，在单一的源文件中
//匿名空间中定义的实体类似于定义在全局位置的实体
namespace {
int num = 10;

void func(){
    cout << "func()" << endl;
}
}

void test0(){
    cout << ::num << endl;
    ::func();
}
```

```
ray@ubuntu:~/HaiBao/54th/day01$ ./a.out
100
func()
```

使用匿名空间中实体时，可以直接使用，也可以加上作用域限定符（没有空间名），但是如果匿名空间中定义了和全局位置中同名的实体，会有冲突，即使使用::作用域限定符也无法访问到匿名空间中重名的实体，只能访问到全局的实体。

在C++代码中可以直接使用一些C语言的函数，就是通过匿名空间实现（体现了C++对C的兼容性），在本文件使用匿名命名空间的实体时不必用命名空间限定。

```cpp
namespace {

//在C++中可以直接使用的C语言的函数
//都已经定义在匿名空间中了
//如果人为地在匿名空间重新定义，
//再调用时会有新的逻辑
//不建议这样改写，会造成混乱
void printf(const char * str,int a){
    cout << str << endl;
    cout << a << endl;
}

int num = 10;

void func(){
    cout << "func()" << endl;
}
}
```

```
void test0(){
    int num = 11;
    cout << num << endl;
    func();
    printf("%d\n",num);
}
```

printf本身可以直接用，和C语言中的效果一致。但是经过匿名空间改写后，效果不一样了——**不要随意改写**

匿名空间注意事项：

（1）匿名空间不要定义与全局空间中同名的实体；

（2）匿名空间中支持改写兼容C语言的函数，但是最好不要改写；

（3）匿名空间中的实体不能跨模块调用。

补充：匿名空间和有名空间（具名空间）统称为命名空间（名称空间）。

跨模块调用问题

一个.c/.cc/*.cpp的文件可以称为一个模块。

（1）全局变量和函数是可以跨模块调用的

externA.cc

```
int num = 100;

void print(){
    cout << "externA print()" << endl;
}
```

externB.cc

```cpp
#include <iostream>
using std::cout;
using std::endl;

extern int num;//外部引入声明
extern void print();

void test0(){
    cout << num << endl;
    print();
}

int main(void){
    test0();
    return 0;
}
```

对externA.cc和externB.cc联合编译，实现跨模块调用

（2）有名命名空间中的实体可以跨模块调用

```cpp
int num = 100;

void print(){
    cout << "externA print()" << endl;
}

namespace wd2
{
int num = 200;

void print(){
    cout << "externA wd print()" << endl;
}
```

```cpp
extern int num;//外部引入声明
extern void print();


namespace wd2
{
extern int num;

extern void print();
```

```cpp
void test0(){
    using wd2::num;
    using wd2::print;
    cout << num << endl;
    print();
}
```

命名空间中的实体跨模块调用时，要在新的源文件中再次定义同名的命名空间，进行联合编译时，这两次定义被认为是同一个命名空间。

使用规则：如果要同时从全局位置和命名空间中外部引入实体，要么让它们不要重名，要么在使用时采取作用域限定的方式。

（3）静态变量和函数只能在本模块内部使用

externNoA.cc externNoB.cc

```cpp
#include <iostream>
using std::cout;
using std::endl;

static int num = 100;
static void print(){
    cout << "static print" << endl;
}
```

externNoA.cc externNoB.cc

```cpp
#include <iostream>
using std::cout;
using std::endl;

extern int num;

void test0(){
    cout << num << endl;
}
```

```
ray@ubuntu:~/HaiBao/54th/day01$ g++ externNoA.cc externNoB.cc
/tmp/cctdaRR5.o: In function `test0()':
externNoB.cc:(.text+0x6): undefined reference to `num'
collect2: error: ld returned 1 exit status
```

（4）匿名空间的实体只能在本模块内部使用

匿名空间中的实体只能在本文件的作用域内有效，它的作用域是从匿名命名空间声明开始到本文件结束。

```cpp
namespace
{
int num2 = 300;

void print(){
    cout << "Anonymous print" << endl;
}

}
```

```cpp
namespace {

extern int num2;
}

void test0(){
    cout << num2 << endl;
}
```

```
ray@ubuntu:~/HaiBao/54th/day01$ g++ externNoA.cc externNoB.cc
/tmp/ccTeAVxP.o: In function `test0()':
externNoB.cc:(.text+0x6): undefined reference to `(anonymous namesp
ace)::num2'
collect2: error: ld returned 1 exit status
```

补充：extern外部引入的方式适合管理较小的代码组织，用什么就引入什么，但是如果跨模块调用的关系不清晰，很容易出错；

include头文件的方式在代码组织上更清晰，但是会一次引入全部内容，相较而言效率比较低。

补充：vim多窗口操作

```
:e 创建或者打开另一个文件
: bn 切换到下一个窗口
: bp 切换到上一个窗口
: bd 关闭当前窗口
```

命名空间可以多次定义

函数可以声明多次，但是只能定义一次；命名空间可以多次定义。

```
namespace wd
{
int num = 200;
}

namespace wd
{
int num2 = 300;
//int num = 300;//error
}
```

在同一个源文件中可以多次定义同名的命名空间，被认为是同一个命名空间，所以不能进行重复定义。

```
namespace wd
{
int num ;
//num = 400; //error

void print(){
    cout << "print()" << endl;
}

//print();//error
}

namespace wd
{
int num2 = 300;
//int num = 300;//error
}

using namespace wd;
void test0(){
    print();
    num = 500;
    cout << num << endl;
}
```

在命名空间中可以声明实体、定义实体，但是不能使用实体。使用命名空间中的实体一定在命名空间之外，可以理解为命名空间只是用来存放实体。

总结

命名空间的作用：

1. 避免命名冲突：命名空间提供了一种将全局作用域划分成更小的作用域的机制，用于避免不同的代码中可能发生的命名冲突问题；
2. 组织代码：将相关的实体放到同一个命名空间；
3. 版本控制：不同版本的代码放到不同的命名空间中；

   总之，需要用到代码分隔的情况就可以考虑使用命名空间。

还有一个隐藏的好处：声明主权。

下面引用当前流行的命名空间使用指导原则：

1. 提倡在已命名的名称空间中定义变量，而不是直接定义外部全局变量或者静态全局变量。
2. 如果开发了一个函数库或者类库，提倡将其放在一个命名空间中。
3. 对于using 声明，首先将其作用域设置为局部而不是全局。
4. 不要在头文件中使用using编译指令，这样，使得可用名称变得模糊，容易出现二义性。
5. 包含头文件的顺序可能会影响程序的行为，如果非要使用using编译指令，建议放在所有#include预编译指令后。

const关键字

修饰内置类型*

```
const int number1 = 10;
int const number2 = 20;

const int val;//error 常量必须要进行初始化
```

const修饰的变量称为常量，之后不能修改其值

char/short/int/long/float/double 整型、浮点型数据都可以修饰——const常量

```
#define MAX 100

void test0(){
    cout << MAX - 10 << endl;

    //const int max = "hello";//error, 因为const常量有类型检查
    const int max = 100;
    cout << max - 10 << endl;

    //max = 120;//error, const常量不能修改

    int num;
    num = 1000;
    //const int num2; //error, const常量必须初始化
    //num2 = 1200;

    int const min = 0;
    //min = 20;//error
    cout << min << endl;
}
```

除了这种方式可以创建常量外，还可以使用宏定义的方式创建常量

```
1  #define NUMBER 1024
```

由此引出一个**面试常考题**：

const常量和宏定义常量的区别

1. **发生的时机不同**：C语言中的宏定义发生时机在预处理时，做字符串的替换；

   const常量是在编译时（const常量本质还是一个变量，只是用const关键字限定之后，赋予只读属性，使用时依然是以变量的形式去使用）

2. **类型和安全检查不同**：宏定义没有类型，不做任何类型检查；const常量有具体的类型，在编译期会执行类型检查。

   在使用中，应尽量以const替换宏定义，可以减小犯错误的概率。

修饰指针类型*

三种形式：const int * p　　int const * p1　　int * const p2

```
int number1 = 10;
int number2 = 20;
const int * p1 = &number1;//常量指针
*p1 = 100;//error 通过p1指针无法修改其所指内容的值
p1 = &numbers;//ok 可以改变p1指针的指向
int const * p2 = &number1; //常量指针的第二种写法
int * const p3 = &number1;//指针常量
*p3 = 100;//ok 通过p3指针可以修改其所指内容的值
p3 = &number2;//error 不可以改变p1指针的指向
const int * const p4 = &number1;//两者皆不能进行修改
```

理解常量指针和指针常量的区别（重点）

```
void test1(){
    int num = 10, num2 = 11;
    //const在星号前面，表示常量指针
    //指向一个常量的指针
    const int * p = &num;
    cout << *p << endl;
    //*p = 5; //不能通过解引用修改所指向的值
    p = &num2;//可以修改指向
    cout << *p << endl;
}
```

```
void test3(){
    int num = 10, num2 = 11;
    //const在星号后面，表示指针常量
    //是一个常量，一个指针特性的常量
    //只读属性施加在指针上
    int * const p = &num;
    cout << *p << endl;
    *p = 5; //能够通过解引用修改所指向的值
    //p = &num2;//不能修改指向
    cout << *p << endl;

}
```

与这组概念相似的，再补充两组对比，也应该理解其含义，尝试写代码，分辨一下：

数组指针

```
//数组指针,指向一个数组的指针
void test4(){
    int arr[5] = {1,2,3,4,5};
    int (*p)[5] = &arr;
    for(int i = 0; i < 5; ++i){
        cout << (*p)[i] << endl;
    }
}
```

指针数组

```
//指针数组，是一个数组，元素都是指针
void test5(){
    int num = 5, num2 = 6, num3 = 7;
    int * p1 = &num;
    int * p2 = &num2;
    int * p3 = &num3;
    int* arr[3] = {p1,p2,p3};
    for(int i = 0; i < 3; ++i){
        cout << *arr[i] << endl;
    }

}
```

函数指针

```
//函数指针
int add(int x, int y){
    return x + y;
}

void test6(){
    //函数指针定义和使用的完整形式
    int (*p)(int,int) = &add;
    cout << (*p)(7,8) << endl;

    //省略形式
    int (*p2)(int,int) = add;
    cout << p2(7,8) << endl;
}
```

指针函数

```
//指针函数——返回类型为指针的函数
int Number = 600;

int * f(){
    int * p = &Number;
    return p;
}

void test7(){
    cout << *f() << endl;
}
```

new/delete表达式

C/C++申请、释放堆空间的方式对比

C语言中使用malloc/free函数，C++使用new/delete表达式

new语句中可以不加参数，初始化为各类型默认值；也可加参数，参数代表要初始化的值

```
int * p = new int(1);
cout << *p << endl;
```

valgrind工具集*

valgrind是一种开源工具集，它提供了一系列用于调试和分析程序的工具。其中最为常用和强大的工具就是memcheck。它是valgrind中的一个内存错误检查器，它能够对C/C++程序进行内存泄漏检测、非法内存访问检测等工作。

- **sudo apt install valgrind**

安装完成后即可通过memcheck工具查看内存泄漏情况，编译后输入如下指令

```
valgrind --tool=memcheck ./a.out
```

如果想要更详细的泄漏情况，如造成泄漏的代码定位，编译时加上-g

```
valgrind --tool=memcheck --leak-check=full ./a.out
```

但是这么长的指令使用起来不方便，每查一次就得输入一次，如果需要查看静态区的情况，还需要

```
valgrind --tool=memcheck --leak-check=full --show-reachable=yes
./a.out
```

- 在home目录下编辑.bashrc文件，改别名

```
alias memcheck='valgrind --tool=memcheck --leak-check=full --show-reachable=yes'
```

- 重新加载 source .bashrc

```
115 # some more ls aliases
116 alias ll='ls -alF'
117 alias la='ls -A'
118 alias l='ls -CF'
119 alias memcheck='valgrind --tool=memcheck --leak-check=full --show-reachable=yes'
```

改写之后，就可以直接使用memcheck指令查看内存泄漏情况 —— memcheck ./a.out

```
==36201==    at 0x4C3217F: operator new(unsigned long) (in /usr/lib/v
algrind/vgpreload_memcheck-amd64-linux.so)
==36201==    by 0x10899F: test0() (new.cc:10)
==36201==    by 0x108A1B: main (new.cc:19)
==36201==
==36201== 4 bytes in 1 blocks are definitely lost in loss record 2 of
 2
==36201==    at 0x4C3217F: operator new(unsigned long) (in /usr/lib/v
algrind/vgpreload_memcheck-amd64-linux.so)
==36201==    by 0x1089DC: test0() (new.cc:13)
==36201==    by 0x108A1B: main (new.cc:19)
==36201==
==36201== LEAK SUMMARY:
==36201==    definitely lost: 8 bytes in 2 blocks
==36201==    indirectly lost: 0 bytes in 0 blocks
==36201==      possibly lost: 0 bytes in 0 blocks
==36201==    still reachable: 0 bytes in 0 blocks
==36201==         suppressed: 0 bytes in 0 blocks
```

（1）绝对泄漏了；（2）间接泄漏了；（3）可能泄漏了，基本不会出现；（4）没有被回收，但是不确定要不要回收；（5）被编译器自动回收了，不用管

```
int * p = (int*)malloc(sizeof(int));
*p = 10;
free(p);

int * p1 = new int();//初始化为该类型的默认值
cout << *p1 << endl;

int * p2 = new int(10);//初始化为传入的参数值
cout << *p2 << endl;
```

需要对new表达式申请的空间进行回收

```
int * p1 = new int();//初始化为该类型的默认值
cout << *p1 << endl;
delete p1;

int * p2 = new int(10);//初始化为传入的参数值
cout << *p2 << endl;
delete p2;
```

通过new表达式的使用，引申出**常考面试题**

malloc/free 和 new/delete 的区别

1. malloc/free是库函数；new/delete是表达式，后两者使用时不是函数的写法；
2. new表达式的返回值是相应类型的指针，malloc返回值是void*；

3. malloc申请的空间不会进行初始化，获取到的空间是有脏数据的，但new表达式申请空间时可以直接初始化；
4. malloc的参数是字节数，new表达式不需要传递字节数，会根据相应类型自动获取空间大小。

new表达式申请数组空间

new表达式还可以申请数组空间

```
int * p = new int[10]();
for(int idx = 0; idx != 10; ++idx)
{
    p[idx] = idx;
}
```

```
//10代表数组要存放的元素个数
//写上小括号，确保了对申请的这片空间进行了初始化
int * p3 = new int[10]();
for(int idx = 0; idx < 10; ++idx){
    cout << p3[idx] << endl;
}
cout << endl;

//new申请数组空间，如果确定好要存放的元素，
//可以采用初始化列表
//大括号包含要初始化的元素
//如果写了小括号，不能往里面传参数
int * p4 = new int[3]{1,2};
for(int idx = 0; idx < 3; ++idx){
    cout << p4[idx] << endl;
}
```

使用new语句申请数组空间需要使用delete [ ] p的形式回收堆空间

回收空间时的注意事项

（1）三组申请空间和回收空间的匹配组合

```
malloc              free

new                 delete

new int[5]()        delete[]
```

如果没有匹配，memcheck会报出错误匹配的信息，实际开发中有可能回收掉了有用的信息。

（2）安全回收

delete只是回收了指针指向的空间，但这个指针变量依然还在，指向了不确定的内容（野指针），容易造成错误。所以需要进行安全回收，将这个指针设为空指针。C++11之后使用**nullptr**表示空指针。

```
int * p1 = new int();//初始化为该类型的默认值
cout << *p1 << endl;
delete p1;
//这个写法存在隐患
//因为其他的程序可能将数据存到了这片空间
//但是p1还是指向这片空间，有可能造成篡改
//cout << *p1 << endl;
p1 = nullptr;//安全回收
cout << *p1 << endl;//error
```

引用（重点）

引用的概念

在理解引用概念前，先回顾一下变量名。 变量名实质就是一段连续内存空间的别名。那一段连续的内存空间只能取一个别名吗？ 显然不是，引用的概念油然而生。在C++中，**引用是一个已定义变量的别名**。

其语法是：

```
//定义方式：     类型 & ref = 变量;
int number = 2;
int & ref = number;
```

在使用引用的过程中，要注意以下几点：

1. &在这里不再是取地址符号，而是引用符号
2. 引用的类型需要和其绑定的变量的类型相同（目前这样使用，学习继承后这一条有所不同）
3. **声明引用的同时，必须对引用进行初始化，否则编译时报错**
4. **引用一经绑定，无法更改绑定**

```
void test0(){
    int num = 100;
    int & ref = num;//声明ref时进行了初始化（绑定）
    //int & ref2; //error
    cout << num << endl;
    cout << ref << endl;
    cout << &num << endl;
    cout << &ref << endl;

    int num2 = 200;
    ref = num2; //这不是更改绑定，而是赋值操作
    //对引用操作就是对变量本身操作
    cout << ref << endl;//ref被改成了200
    cout << num << endl; //num也被改成了200
    cout << num2 << endl;
    cout << &num << endl;
    cout << &ref << endl;
    cout << &num2 << endl;
}
```

引用的本质

C++中的引用本质上是一种被限制的指针。类似于线性表和栈的关系，栈是被限制的线性表，底层实现相同，只不过逻辑上的用法不同而已。

由于**引用是被限制的指针**，所以引用是占据内存的，占据的大小就是一个指针的大小。有很多的说法，都说引用不会占据存储空间，其只是一个变量的别名，但这种说法并不准确。引用变量会占据存储空间，存放的是一个地址，但是编译器阻止对它本身的任何访问，从一而终总是指向初始的目标单元。在汇编里，引用的本质就是“间接寻址”。

可以尝试对引用取址，发现获取到的地址就是引用所绑定变量的地址。

引用与指针的联系与区别*

这是一道非常经典的面试题，请尝试着回答一下：

联系：

1. 引用和指针都有地址的概念，都是用来间接访问变量；
2. 引用的底层还是指针来完成，可以把引用视为一个受限制的指针。

区别：

1. 引用必须初始化，指针可以不初始化；

2. 引用不能修改绑定，但是指针可以修改指向；
3. 在代码层面对引用本身取址取到的是变量的地址，但是对指针取址取到的是指针变量本身的地址

引用的使用场景

引用作为函数的参数（重点）

在没有引用之前，如果我们想通过形参改变实参的值，只有使用指针才能到达目的。但使用指针的过程中，不好操作，很容易犯错。 而引用既然可以作为其他变量的别人而存在，那在很多场合下就可以用引用代替指针，因而也具有更好的可读性和实用性。这就是引用存在的意义。

一个经典的例子就是交换两个变量的值，请实现一个函数，能够交换两个int型变量的值：

```
void swap(int x, int y){//值传递，发生复制
    int temp = x;
    x = y;
    y = temp;
}

void swap2(int * px, int * py){//地址传递，不复制
    int temp = *px;
    *px = *py;
    *py = temp;
}

//在实参传给swap3时，
//其实就是发生了初始化int & x = a;
//int & y = b;
void swap3(int & x, int & y){//引用传递，不复制
    int temp = x;
    x = y;
    y = temp;
}
```

```
void test0(){
    int a = 1, b = 2;
    swap2(&a,&b);
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
}

void test1(){
    int a = 1, b = 2;
    swap3(a,b);
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
}
```

参数传递的方式包括值传递、指针传递和引用传递。采用值传递时，系统会在内存中开辟空间用来存储形参变量，并将实参变量的值拷贝给形参变量，即形参变量只是实参变量的副本而已；如果函数传递的是类对象，而该对象占据的存储空间比较大，那发生复制就会造成较大的不必要开销。这种情况下，强烈建议使用引用作为函数的形参，这样会大大提高函数的时空效率。

当用引用作为函数的参数时，其效果和用指针作为函数参数的效果相当。当调用函数时，函数中的形参就会被当成实参变量或对象的一个别名来使用，也就是说此时函数中对形参的各种操作实际上是对实参本身进行操作，而非简单的将实参变量或对象的值拷贝给形参。

使用指针作为函数的形参虽然达到的效果和使用引用一样，但当调用函数时仍需要为形参指针变量在内存中分配空间，也由于指针的灵活更可能导致问题的产生，故在C++中推荐使用引用而非指针作为函数的参数。

- 不希望函数体中通过引用改变传入的变量，那么可以使用**常引用作为函数参数**

（1）不会修改值　　（2）不会复制（不会造成不必要的开销）

```
void func(const int & x,int & y){
    //x = 100;//error 常引用
    y = 200;
}
```

引用作为函数的返回值

要求：当以引用作为函数的返回值时，**返回的变量其生命周期一定是要大于函数的生命周期**的，即当函数执行完毕时，返回的变量还存在。

目的： 避免复制，节省开销

```
int  func(){
    //...
    return a;    //在函数内部，当执行return语句时，会发生复制
}

int & func2(){
     //...
    return b;    //在函数内部，当执行return语句时，不会发生复制
}
```

```
int gNumber = 100;

int func(){
    cout << "gNumber:" << gNumber << endl;
    return gNumber; //返回类型是int(非引用)，return时复制
}

int & func2(){
    cout << "gNumber:" << gNumber << endl;
    //int & ref = gNumber;
    //return ref;
    return gNumber; //返回类型是引用，return时不复制
}

void test0(){
    //cout << &func() << endl;//返回的是临时变量，不能取地址
    cout << &func2() << endl;
    cout << &gNumber << endl;
}
```

注意事项

1. 不要返回局部变量的引用。因为局部变量会在函数返回后被销毁，被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

```
int & func()
{
    int number = 1;
    return number;
}
```

2. **不要轻易**返回一个堆空间变量的引用，非常容易造成内存泄漏。

```
int & func()
{
    int * pint = new int(1);
    return *pint;
}

void test()
{
    int a = 2, b = 4;
    int c = a + func() + b;//内存泄漏
}
```

```
int & func4(){
    int * hNumber = new int(1);
    cout << *hNumber << endl;
    return *hNumber;
}
```

```
int & ref = func4();
ref = 100;
delete &ref;
```

如果函数返回的是一个堆空间变量的引用，那么这个函数调用一次就会new一次，非常容易造成内存泄露。所以谨慎使用这种写法，并且要有完善的回收机制。

总结

引用总结：

1. 在引用的使用中，单纯给某个变量取个别名没有什么意义，引用的目的主要用于在函数参数传递中，解决大块数据或对象的传递效率和空间不理想的问题。
2. 用引用传递函数的参数，能保证参数传递中不产生副本，提高传递的效率，还可以通过const的使用，保证了引用传递的安全性。
3. 引用与指针的区别是，指针通过某个指针变量指向一个变量后，对它所指向的变量间接操作。程序中使用指针，程序的可读性差；而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作。**可以用指针或引用解决的问题，更推荐使用引用**。

强制转换

C语言中的强制转换在C++代码中依然可以使用，这种C风格的转换格式非常简单

```
TYPE a = (TYPE) EXPRESSION;
```

但是c风格的类型转换有不少的缺点，有的时候用c风格的转换是不合适的，因为它可以在任意类型之间转换，比如你可以把一个指向const对象的指针转换成指向非const对象的指针，把一个指向基类对象的指针转换成指向一个派生类对象的指针，这两种转换之间的差别是巨大的，但是传统的c语言风格的类型转换没有区分这些。

另一个缺点就是，c风格的转换不容易查找，它由一个括号加上一个标识符组成，而这样的东西在c++程序里一大堆。c++为了克服这些缺点，引进了4个新的类型转换操作符，他们是static_cast，const_cast，dynamic_cast，reinterpret_cast。

static_cast

最常用的类型转换符，在正常状况下的类型转换，用于将一种数据类型转换成另一种数据类型，如把int转换为float

使用形式

```
目标类型 转换后的变量 = static_cast<目标类型>(要转换的变量)
```

好处：不允许非法的转换发生；方便查找

```
int iNumber = 100;
float fNumber = 0;
fNumber = (float) iNumber; //C风格
fNumber = static_cast<float>(iNumber);
```

也可以完成指针之间的转换，例如可以将void*指针转换成其他类型的指针

```
void * pVoid = malloc(sizeof(int));
int * pInt = static_cast<int*>(pVoid);
*pInt = 1;
```

但**不能完成任意两个指针类型间的转换**

```
int iNumber = 1;
int * pInt = &iNumber;
float * pFloat = static_cast<float *>(pInt);//error
```

```
const char * pstr = "hello";
//int * p = static_cast<int*>(pstr);//非法的转换，error
```

总结，static_cast的用法主要有以下几种：

1）用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性需要开发人员来保证；

2）把void指针转换成目标类型的指针，但不安全；

3）把任何类型的表达式转换成void类型；

4）用于类层次结构中基类和子类之间指针或引用的转换（后面学）。

const_cast

该运算符用来修改类型的const属性，**基本不用。**

常量指针被转化成非常量指针，并且仍然指向原来的对象；

常量引用被转换成非常量引用，并且仍然指向原来的对象；

常量对象被转换成非常量对象。

```
const int number = 100;
int * pInt = &number;//error
int * pInt2 = const_cast<int *>(&number);
```

```
void test1(){
    const int number = 100;
    //int * pInt = &number;//error
    const int * pInt = &number;//ok
    int * pInt2 = const_cast<int *>(&number);
    *pInt2 = 1000;//这里修改的数据没有写回内存，在寄存器
    cout << *pInt2 << endl;
    cout << number << endl;
    cout << pInt2 << endl;
    cout << &number << endl;
}
```

```
ray@ubuntu:~/HaiBao/54th/day02$ ./a.out
1000
100
0x7fff54c29524
0x7fff54c29524
```

dynamic_cast：该运算符主要用于基类和派生类间的转换，尤其是向下转型的用法中（后面讲）

reinterpret_cast：功能强大，慎用（也称为万能转换）

该运算符可以用来处理无关类型之间的转换，即用在任意指针（或引用）类型之间的转换，以及指针与足够大的整数类型之间的转换。由此可以看出，reinterpret_cast的效果很强大，但错误的使用reinterpret_cast很容易导致程序的不安全，**只有将转换后的类型值转换回到其原始类型，这样才是正确使用reinterpret_cast方式**。

函数重载

在实际开发中，有时候需要实现几个功能类似的函数，只是细节有所不同。 如交换两个变量的值，但这两种变量可以有多种类型，short，int，float等。在C语言中，必须要设计出不同名的函数，其原型类似于：

```
void swap1(short *, short *);
void swap2(int *, int *);
void swap3(float *, float *);
```

但在C++中，这完全没有必要。C++ 允许多个函数拥有相同的名字，只要它们的参数列表不同就可以，这就是函数重载（Function Overloading）。借助重载，一个函数名可以有多种用途。

在同一作用域内，可以有一组具有相同函数名，不同参数列表的函数，这组函数被称为重载函数。重载函数通常用来命名一组功能相似的函数，这样做减少了函数名的数量，对于程序的可读性有很大的好处。

注意：C 语言中不支持函数重载，C++才支持函数重载。

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

/* int add(int y, int x){ */
/*     return y + x; */
/* } */

int add(int x, int y, int z){
    return x + y + z;
}

float add(float x, int y){
    return x + y;
}

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

实现函数重载的条件

函数参数的数量、类型、顺序任一不同则可以构成重载。

只有返回类型不同，参数完全相同，是不能构成重载的

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

```cpp
void add(int x,int y){
    cout << x + y << endl;
}
```

函数重载的实现原理

实现原理： 名字改编(name mangling)——当函数名称相同时，会根据参数的类型、顺序、个数进行改编

- g++ -c Overload.cc
- nm Overload.o

查看目标文件，可以发现原本的函数名都被改编成与参数相关的函数名。

```
ray@ubuntu:~/HaiBao/54th/day02$ nm overload.o
                 U __cxa_atexit
                 U __dso_handle
                 U _GLOBAL_OFFSET_TABLE_
00000000000001b9 t _GLOBAL__sub_I__Z3addii
0000000000000160 T main
0000000000000030 T _Z3addfi
0000000000000048 T _Z3addif
0000000000000000 T _Z3addii
0000000000000014 T _Z3addiii
```

C语言没有名字改编

overload.c overload.cc

```c
#include <stdio.h>
#include <stdlib.h>

float add(float x, int y){
    return x + y;
}

float add2(int x, float y){
    return x + y;
}
```

```
ray@ubuntu:~/HaiBao/54th/day02$ gcc -c overload.c
ray@ubuntu:~/HaiBao/54th/day02$ nm overload.o
0000000000000000 T add
0000000000000018 T add2
0000000000000030 T main
```

extern "C"

在C/C++混合编程的场景下，如果在C++代码中想要按照C的方式编译函数应该怎么办?

```c
extern "C" void func() //用 extern"C"修饰单个函数
{

}

//如果是多个函数都希望用C的方式编译
//或是需要使用C语言的库文件
//都可以放到如下{}中
extern "C"
{
//……
}
```

```
extern "C"{

int add(int x, int y, int z){
    return x + y + z;
}

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

```
overload.cc: In function 'float add(float, int)':
overload.cc:19:7: error: conflicting declaration of C function 'float add(flo
, int)'
 float add(float x, int y){
       ^~~
overload.cc:15:5: note: previous declaration 'int add(int, int, int)'
 int add(int x, int y, int z){
```

假如这段代码用C的编译器进行编译，extern "C"{}是不能被识别的，会出现问题，所以可以用如下的宏包裹起来。

```
#ifdef __cplusplus
extern "C"{
#endif

int add(int x, int y, int z){
    return x + y + z;
}

float add(float x, int y){
    return x + y;
}
#ifdef __cplusplus
}
#endif
```

默认参数

默认参数的目的

C++可以给函数定义默认参数值。通常，调用函数时，要为函数的每个参数给定对应的实参。

```
void func1(int x, int y)
{
cout << "x = " << x << endl;
cout << "y = " << y << endl;
}
```

无论何时调用func1函数，都必须要给其传递两个参数。但C++可以给参数定义默认值，如果将func1函

数参数中的x定义成默认值0， y定义成默认值0，只需简单的将函数声明改成

```
void func1(int x = 0, int y = 0);
```

这样调用时，若不给参数传递实参，则func1函数会按指定的默认值进行工作。允许函数设置默认参数值，是为了让编程简单，让编译器做更多的检查错误工作。

```
int add(int x = 12, int y = 1){
    return x + y;
}

void test0(){
    cout << add(24,30) << endl;
    //缺省调用
    cout << add(100) << endl;
    cout << add() << endl;
}
```

函数给了默认参数之后就可以进行缺省调用，但是传入的参数优先级高于默认参数。

默认参数的声明

一般默认参数在函数声明中提供。当一个函数既有声明又有定义时，只需要在其中一个中设置默认值即可。若在定义时而不是在声明时置默认值，那么函数定义一定要在函数的调用之前。因为声明时已经给编译器一个该函数的向导，在定义时设默认值时，编译器只有检查到定义时才知道函数使用了默认值。若先调用后定义，在调用时编译器并不知道哪个参数设了默认值。

```
//这样可以编译通过
void func(int x,int y);

void test0(){
   func(1,2);
}

void func(int x,int y){
    cout << x + y << endl;

}
```

```
//这样无法缺省调用
void func(int x,int y);

void test0(){
   func();//error
}

void func(int x = 0,int y = 0){
    cout << x + y << endl;

}
```

所以我们**通常是将默认值的设置放在声明中而不是定义**中。

```
int multiply(int x = 100, int y = 50);


int multiply(int x, int y){
    return x * y;
}

void test1(){
    cout << multiply() << endl;
}
```

如果在声明中和定义中都传了默认值，会报错

默认参数的顺序规定

如果一个函数中有多个默认参数，则形参分布中，默认参数应从右至左逐渐定义。当调用函数时，只能从右向左匹配参数。如：

```
void func2(int a = 1, int b, int c = 0, int d);//error
void func2(int a, int b, int c = 0, int d = 0);//ok
```

若给某一参数设置了默认值，那么在参数表中其后所有的参数都必须也设置默认值，否则，由于函数调用时可不列出已设置默认值的参数，编译器无法判断在调用时是否有参数遗漏。

完成函数默认参数的设置后，该函数就可以按照相应的缺省形式进行调用。

```
//当后面的参数没有默认值时，前面的参数也不能有默认值
int add(int x, int y = 1){
    return x + y;
}
```

总结：函数参数赋默认值从右向左（严格）

默认参数与函数重载

默认参数可将一系列简单的重载函数合成为一个。例如：

```
1  void func3();
2  void func3(int x);
3  void func3(int x, int y);
4  //上面三个函数可以合成下面这一个
5  void func3(int x = 0, int y = 0);
```

如果一组重载函数（可能带有默认参数）都允许相同实参个数的调用，将会引起调用的二义性。

```
1  void func4(int x);
2  void func4(int x, int y = 0);
3
4  func4(1);//error,无法确定调用的是哪种形式的func4
```

所以在函数重载时，要谨慎使用默认参数。

```
int add(int x, int y = 1){
    return x + y;
}

int add(int x){
    return x;
}

void test0(){
    cout << add(24,30) << endl;
    //缺省调用
    /* cout << add(100) << endl; */
```

重载是允许的，但是缺省调用时会产生冲突。

bool类型

bool类型是在C++中一种基本类型，用来表示true和false。true和false是字面值，可以通过转换变为int类型，**true为1，false为0.**

```
int x = true;// 1
int y = false;// 0
```

任何数字或指针值都可以隐式转换为bool值。

任何非零值都将转换为true，而零值转换为false（**注意：-1也是代表true**）

```
bool b1 = -100;
bool b2 = 100;
bool b3 = 0;
bool b4 = 1;
bool b5 = true;
bool b6 = false;
int x = sizeof(bool);//x = 1
```

bool变量占1个字节的空间。

```
bool b1 = 0;
bool b2 = 100;
bool b3 = 1;
bool b4 = -1;

cout << "b1:" << b1 << endl;
cout << "b2:" << b2 << endl;
cout << "b3:" << b3 << endl;
cout << "b4:" << b4 << endl;

//true
if(b4){
    cout << "hello" << endl;
}

//false
if(b1){
    cout << "hello" << endl;
}

cout << "sizeof(bool):" << sizeof(bool) << endl;
```

inline函数

在C++中，通常定义以下函数来求取两个整数的最大值

```
int max(int x, int y)
{
    return x > y ? x : y;
}
```

为这么一个小的操作定义一个函数的好处有：

(1)阅读和理解函数 max 的调用，要比读一条等价的条件表达式并解释它的含义要容易得多；

(2)如果需要做任何修改，修改函数要比找出并修改每一处等价表达式容易得多；

(3)使用函数可以确保统一的行为，每个测试都保证以相同的方式实现；

(4)函数可以重用，不必为其他应用程序重写代码。

虽然有这么多好处，但是写成函数有一个潜在的缺点：调用函数比求解等价表达式要慢得多。在大多数的机器上，调用函数都要做很多工作：调用前要先保存寄存器，并在返回时恢复，复制实参，程序还必须转向一个新位置执行。即对于这种简短的语句使用函数开销太大。

在C语言中，我们使用带参数的宏定义这种借助编译器的优化技术来减少程序的执行时间，请定义一个宏完成以上的max函数的功能

参考1.9.2

那么在C++中有没有相同的技术或者更好的实现方法呢？答案是有的，那就是内联(inline)函数。内联函数作为编译器优化手段的一种技术，在降低运行时间上非常有用。

什么是内联函数

内联函数是C++的增强特性之一，用来降低程序的运行时间。当内联函数收到编译器的指示时，即可发生内联：编译器将使用函数的定义体来替代函数调用语句，这种替代行为发生在编译阶段而非程序运行阶段。

定义函数时，在函数的最前面以关键字“inline”声明函数，该函数即可称为内联函数（内联声明函数）。

```
inline int max(int x, y)
{
    return x > y ? x : y;
}
```

宏函数与内联函数

在C程序中，可以用宏代码提高执行效率。宏代码本身不是函数，但是看起来像函数。编译预处理器用拷贝宏代码的方式取代函数调用，省去了参数压栈、生成汇编语言的CALL调用、返回参数、执行return等过程，从而提高了速度。

使用宏代码最大的缺点是容易出错，预处理器在拷贝宏代码时常常产生意向不到的边际效应。例如：

```
#define MAX(a, b) (a) > (b) ? (a) : (b)

int result = MAX(20,10) + 20//result的值是多少?

int result2 = MAX(10,20) + 20//result2的值是多少?

//result = MAX(i, j) + 20; 将被预处理器扩展为: result = (i) > (j) ?(i): (j)+20
```

可以修改宏代码为

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

可以解决上面的错误了，但也不是万无一失的，例如：

```
int i = 4,j = 3;
result = MAX(i++,j);
cout << result << endl; //result = 5;
cout << i << endl; //i = 6;

//使用MAX的代码段经过预处理器扩展后，result = ((i++) > (j) ? (i++):(j));
```

宏的另一个缺点就是**不可调试**，但内联函数是可以调试的。内联函数不是也像宏一样进行代码展开吗？怎么能够调试呢？其实内联函数的"可调试"不是说展开后还能调试，而是在程序的调试（Debug）版本里它根本就没有真正内联，编译器像普通函数那样为它生成含有调试信息的可执行代码。在程序的发行（Release)版本里，编译器才会实施真正的内联。

那C++的内联函数是如何工作的呢？

对于任何内联函数，编译器在符号表（符号表是编译器用来收集和保存字面常量和某些符号常量的地方）里放入函数的声明，包括名字、参数类型、返回值类型。如果编译器没有发现内联函数存在错误，那么该函数的代码也会被放入符号表里。在调用一个内联函数时，编译器首先检查调用是否正确（进行类型安全检查，或者进行自动类型转换）。如果正确，内联函数的代码就会直接替换函数调用语句，于是省去了函数调用的开销。这个过程与预处理有显著的不同，因为预处理器不能执行类型安全检查和自动类型转换。

—— 内联函数就是在普通函数定义之前加上inline关键字

（1）inline是一个建议，并不是强制性的，后面会学到inline失效的情况

（2）inline的建议如果有效，就会在**编译时**展开，可以理解为是一种更高级的代码替换机制（类似于宏——预处理）

（3）函数体内容如果太长或者有循环之类的结构，不建议inline，以免造成代码膨胀；比较短小的代码适合用inline。

C++的**函数内联机制**既具备宏代码的效率，又增加了安全性，而且可以自由操作类的数据成员，所以在C++中应尽可能的用内联函数取代宏函数。

对比总结：

宏函数 优点：只是进行字符串的替换，并没有函数的开销，对于比较短小的代码适合使用；

缺点：没有类型检查，存在安全隐患，而且比较容易写错。

如果使用普通函数的方式又会增加开销，所以一些时候可以采用内联函数（结合了宏函数和普通函数的优点）。

inline函数本质也是字符串替换（编译时），所以不会增加开销，但是有类型检查，比较安全。

内联函数注意事项

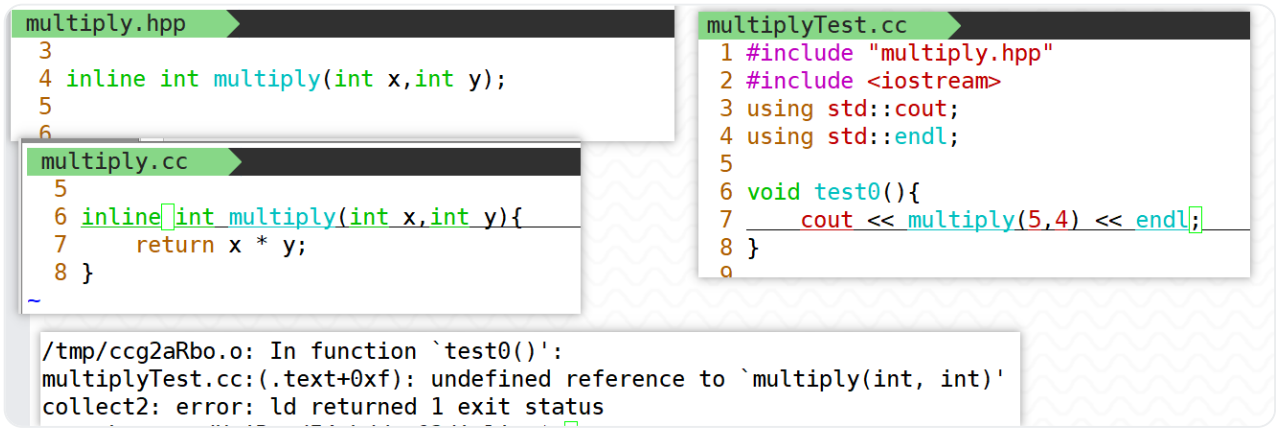
1. **如果要把inline函数声明在头文件中，则必须把函数定义也写在头文件中。**若头文件中只有声明没有实现，被认为是没有定义替换规则。

   如下，foo函数不能成为内联函数：

```
inline void foo(int x, int y);//该语句在头文件中

void foo(int x, int y)//实现在.cpp文件中
{ //... }
```

因为编译器在调用点内联展开函数的代码时，必须能够找到 inline函数的定义才能将调用函数替换为函数代码，而对于在头文件中仅有函数声明是不够的。

当然内联函数定义也可以放在源文件中，但此时只有定义的那个源文件可以用它，而且需要为每个源文件拷贝一份内联函数的定义(每个源文件里的定义必须是完全相同的)。相比之下，放在头文件中既能够确保调用函数是定义是相同的，又能够保证在调用点能够找到函数定义从而完成内联(替换)。



从测试文件出发，找到头文件，发现此函数是inline函数，那么要展开替换，必须要有明确的替换规则，但是在头文件中并没有发现替换规则，所以报错未定义问题。

inline函数在头文件必须有定义。

2．谨慎使用内联

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数？事实上，内联不是万灵丹，它以代码膨胀（拷贝）为代价，仅仅省去了函数调用的开销，从而提高程序的执行效率。（注意：这里的"函数调用开销"是指参数压栈、跳转、退栈和返回等操作）

如果执行函数体内代码的时间比函数调用的开销大得多，那么 inline 的效率收益会很小。另外，每一处内联函数的调用都要拷贝代码，将使程序的总代码量增大，消耗更多的内存空间。以下情况不宜使用内联：

- 如果函数体内的代码比较长，使用内联将导致可执行代码膨胀过大。
- 如果函数体内出现循环或其他复杂的控制结构，那么执行函数体内代码的时间将比函数调用开销大得多，因此内联的意义并不大。

实际上，inline 在实现的时候就是对编译器的一种请求，因此编译器完全有权利取消一个函数的内联请求。一个好的编译器能够根据函数的定义体，自动取消不值得的内联，或自动地内联一些没有inline 请求的函数。因此编译器往往选择那些短小而简单的函数来内联。

异常处理

异常是程序在执行期间产生的问题。C++ 异常是指在程序运行时发生的特殊情况，比如尝试除以零的操作。异常提供了一种转移程序控制权的方式。C++ 异常处理涉及到三个关键字：try、catch、throw。

抛出异常即检测是否产生异常，在 C++ 中，其采用 **throw 语句**来实现，如果检测到产生异常，则抛出异常。该语句的格式为：

```
throw 表达式;
```

- 先定义抛出异常的规则（throw）,异常是一个表达式，它的值可以是基本类型，也可以是类；

```
double division(double x, double y)
{
    if(y == 0)
        throw "Division by zero condition!";
    return x / y;
}
```

try-catch语句块的语法如下：

```
try {
//语句块
} catch(异常类型) {
//具体的异常处理...
} ...
catch(异常类型) {
//具体的异常处理...
}
```

try-catch语句块的catch可以有多个，至少要有一个，否则会报错。

- 执行 try 块中的语句，如果执行的过程中没有异常抛出，那么执行完后就执行最后一个catch块后面的语句，所有 catch 块中的语句都不会被执行；
- 如果 try 块执行的过程中抛出了异常，那么抛出异常后立即跳转到第一个“异常类型”和抛出的异常类型匹配的 catch 块中执行（称作异常被该 catch 块“捕获”），执行完后再跳转到最后一个catch 块后面继续执行。

注意：**catch的是类型，不是具体信息**。

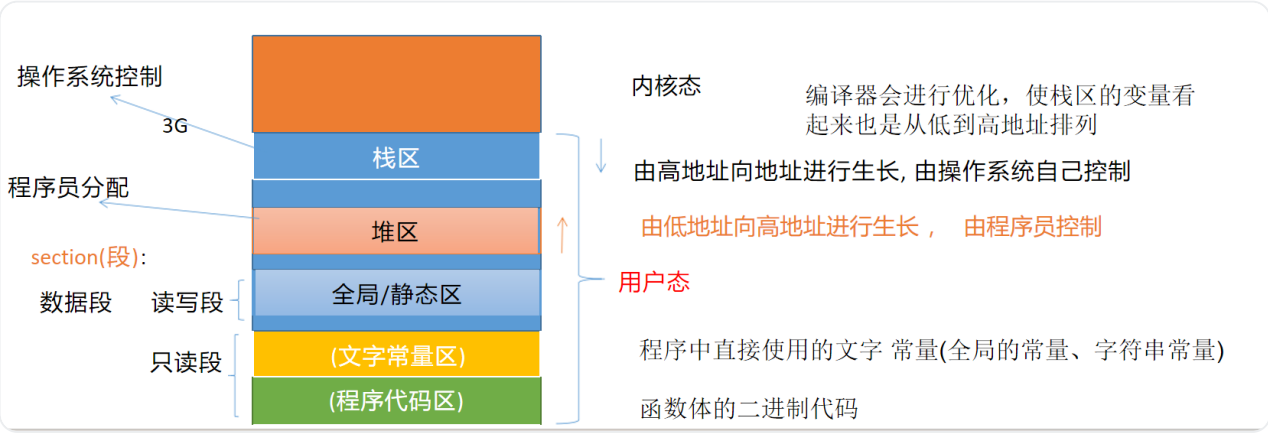
```
double division(double x,double y){
    if(y == 0)
        throw "Deivision by zero";
    return x/y;
}

void test0(){
    double x = 100, y = 0;
    try{
        cout << division(y,x) << endl;
    }catch(const char * msg){ //catch的小括号里是类型
        cout << "hello," << msg << endl;
    }catch(double x){
        cout << "double" << endl;
    }catch(int x){
        cout << "int" << endl;
    }
    cout << "over" << endl;
}
```

内存布局

64位系统，理论空间达到16EB（2^64），但是受硬件限制，并不会达到这么多；

以32位系统为例，一个进程在执行时，能够访问的空间是**虚拟地址空间**。理论上为2^32，即4G，有1G左右的空间是内核态，剩下的3G左右的空间是用户态。从高地址到低地址可以分为五个区域：

- 栈区：操作系统控制，由高地址向低地址生长，编译器做了优化，显示地址时栈区和其他区域保持一致的方向。
- 堆区：程序员分配，由低地址向高地址生长，堆区与栈区没有明确的界限。
- 全局/静态区：读写段（数据段），存放全局变量、静态变量。
- 文字常量区：只读段，存放程序中直接使用的常量，如const char * p = "hello"； hello这个内容就存在文字常量区。
- 程序代码区：只读段，存放函数体的二进制代码。




```cpp
int globalNumber = 8;

void test0(){
    int localNumber = 1;
    int * p = new int(20);
    static int sNumber = 9;
    //编译器优化效果，使得栈区的生长方向看起来和堆区一样
    cout << "&p:" << &p << endl;
    cout << "&localNumber:" << &localNumber << endl;
    cout << "&heapNumber:" << p << endl;
    cout << "&sNumber:" << &sNumber << endl;
    cout << "&globalNumber:" << &globalNumber << endl;

    const char * pstr = "hello";
    //cout << pstr << endl;
    void * p2 = (void*)pstr;
    cout << p2 << endl;
    printf("%p\n",pstr);

    /* cout << "&test0:" << &test0 << endl; */
    printf("%p\n",&test0);

}

int main(void){
    test0();
    printf("%p\n",&main);
    return 0;
```

```
&p:0x7fff4762b860
&localNumber:0x7fff4762b85c
&heapNumber:0x5650a934be70
&sNumber:0x5650a733c014
&globalNumber:0x5650a733c010
0x5650a713ad1d
0x5650a713ad1d
0x5650a713aa0a
0x5650a713abda
```

C风格字符串

如果用数组形式，注意留出一位给终止符；

如果用指针形式，直接定义为const char * ，C++代码中标准C风格字符串的写法。

输出流运算符默认重载，cout利用输出流运算符接char型数组名、指针名时，输出的是内容，而不是地址。

```
void test0(){
    char a[] = {'a','b','c','\0'};
    char b[] = "abcd";
    cout << a << endl;
    cout << b << endl;

    const char * p = "hello";
    cout << p << endl;

    char * p2 = new char[10]();
    strcpy(p2,p);
    cout << p2 << endl;

    char * p3 = new char[4]{"abc"};

    /* int * pint = new int[3]{1,2,3}; */
}
```