C++编程基础

Hello, world!

```
#include <iostream>
#include <stdlib.h>  //使用atexit()
using namespace std;

void show1(void) {
    cout << "first exit main()" << endl;
}

void show2(void) {
    cout << "second exit main()" << endl;
}

int main(int argc, char* argv[]) {
    atexit(show1);
    atexit(show2);
    cout << "Hello, world!" << endl;

    system("pause");
    return 0;
}
/*打印
Hello, world!
second exit main()
first exit main()
*/
```

main 函数前面的数据类型 int 与 void

main 函数的返回值是返回给主调进程，使主调进程得知被调用程序的运行结果。

标准规范中规定 main 函数的返回值为 int，一般约定返回 0 值时代表程序运行无错误，其它值均为错误号，但该约定并非强制。

如果程序的运行结果不需要返回给主调进程，或程序开发人员确认该状态并不重要，比如所有出错信息均在程序中有明确提示的情况下，可以不写 main 函数的返回值。在一些检查不是很严格的编译器中，比如 VC, VS 等，void 类型的 main 是允许的。不过在一些检查严格的编译器下，比如 g++, 则要求 main 函数的返回值必须

为 int 型。

所以在编程时，区分程序运行结果并以 int 型返回，是一个良好的编程习惯。

main 函数执行完后执行其他语句

有时候需要在程序退出时做一些诸如释放资源的操作。我们可以用 atexit() 函数来注册程序正常终止时要被调用的函数。

atexit() 在一个程序中最多可以注册32个处理函数，这些处理函数的调用顺序与注册顺序相反，即后注册的函数先被调用。

"\n" 与 endl 的区别

在 C++ 中，终端输出换行时，用 cout<<.....<<endl 与 "\n" 都可以，这是初级的认识。但二者有小小的区别，用 endl 时会刷新缓冲区，使得栈中的东西刷新一次，但用 "\n" 不会刷新，它只会换行，栈内数据没有变化。但一般情况，二者的区别是很小的，建议用 endl 来换行。

```
1  std::cout << std::endl;
```

相当于:

```
1  std::cout << '\n' << std::flush;
```

或者

```
1  std::cout << '\n'; std::fflush(stdout);
```

endl 除了写 '\n' 之外，还调用 flush 函数，刷新缓冲区，把缓冲区里的数据写入文件或屏幕。考虑效率就用 '\n'。

一般情况下，不加endl大多数情况下，也能正常输出，是因为在系统较为空闲时候，会查看缓存区的内容，如果发现新的内容，便进行输出。但是你并不清楚，系统什么时候输出，什么时候不输出，与系统自身的运行状况有关。而刷新缓存区，是强制性的，绝对性的输出，不取决于系统运行状况。所以正如《C++ Primer》书中所写，为了避免出现没有刷新输出流的情况发生，在使用打印语句来调试程序时，一定要加入 endl或 flush 操纵符。

这里可能会想到，以后遇到这类问题，干脆直接都使用 endl，不用 \n 不就好了吗?

也不是，要知道，endl会不停地刷新输出流，频繁的操作会降低程序的运行效率，这也是C++标准库对流的输入/输出操作使用缓冲区的原因。没有必要刷新输出流的时候应尽量使用 \n，比如对于无缓冲的流 cerr，就可以直接使用 \n。

头文件里的 " " 与 < >

<> 默认去系统目录中找头文件。像标准的 C 头文件 stdio.h、stdlib.h 和 C++ 头文件 iostream、string 等用这个方法。
" " 首先在当前目录下寻找，如果找不到，再到系统目录中寻找。 这个用于 include 自定义的头文件，让系统优先使用当前目录中定义的。

命名空间 std

所谓名称空间它是一种将库函数封装起来的方法。通过这种方法，可以避免和应用程序发生命名冲突的问题。

如果想要使用 `cin`， `cout` 这两个 `iostream` 对象，不仅要包含 `<iostream>` 头文件，还得让命名空间 `std` 内的名称曝光，即 `using namespace std;`

真正的开发过程中， 尽量避免使用 `using namespace std;` 等直接引入整个命名空间，否则会因为命名空间污染导致很多不必要的问题， 比如自己写的某个函数，名称正好和 std 中的一样， 编译器会不知道使用哪一个， 引起编译报错， 建议使用由命名空间组合起来的全称：
`std::cout << "Hello World" << std::endl;`

system("pause")

包含头文件 stdlib.h，并在主程序中加入 `system("pause");` 可以在程序运行完以后使黑框暂停显示，等待输入，而不是闪退。

cout 与 printf()

cout 流速度较慢，如果速度过慢可以用 <stdio.h> 库中的 printf() 格式化输出函数，不需要 `using namespace std;`。但注意 printf() 中不能使用 endl。printf 是函数。cout是ostream对象，和 << 配合使用。如果 printf 碰到不认识的类型就没办法了，而cout可以重载进行扩展。

main函数与命令行参数

一个程序的main()函数可以包括两个参数

- 第一个参数的类型为int型；
- 第二个参数为字符串数组。

通常情况下，将第一个参数命名为argc，第二个参数为argv（当然参数名字可以换）。由于字符串数组有两种表达方式，因此，main函数存在两种书写方法：

```
1  int main(int argc, char* argv[])//这里使用char* argv[]
2
3  int main(int argc, char** argv)//这里使用char **argv
```

int argc：表示字符串的数量，操作系统会自动根据第二个参数传入数字，程序员不用管，只需要正确使用即可。若用户输入N个字符串，那么argc= N + 1；因为 argv[0] 为程序的路径。

char* argv[]：字符串数组，即多个字符串。为用户输入的一系列字符串，字符串之间以空格间隔，形式为：str1 str2 str3
在linux下，若存在可执行文件a.out，则运行该程序命令为：

```
./a.out str1 str2 str3
```

预处理、编译与链接

- 预处理： 预处理也称为预编译，它为编译做准备工作，主要进行代码文本的替换，用于处理#开头的指令 。
- 编译：编译是将编译好的源程序*.cpp文件翻译成二进制目标代码的过程。编译过程是使用C++提供的编译程序完成的，该过程会检查程序语法错误、函数变量的声明是否正确。正确的源程序文件经过编译在磁盘上生成目标文件（windows上是*.obj，linux上是*.o）。
- 链接：链接将编译生成的各个目标程序模块（一个或者多个）及系统或者第三方提供的库函数*.lib链接在一起，生成可以在操作系统上直接运行的可执行文件（windows上的*.exe）

安装 g++

```
sudo apt-get install g++ build-essential
```

将源文件hello.cpp编译成可执行文件hello

```
g++ hello.cpp -o hello
```

运行可执行文件

```
./hello
```

如果省略 `-o hello` 也是没问题的。由于命令行中未指定可执行程序的文件名，编译器采用默认的 a.out。程序可以这样来运行：

```
./a.out
```

如果是多个 C++代码文件，如f1.cpp，f2.cpp，编译命令如下：

```
g++ f1.cpp f2.cpp -o myexec
```

则会生成一个名为 `myexec` 的可执行文件。

静态链接和动态链接

要生成可执行文件，必须经历两个阶段，即编译、链接。

在链接过程中，静态链接和动态链接就出现了区别。静态链接的过程就已经把要链接的内容已经链接到了生成的可执行文件中，就算你在去把静态库删除也不会影响可执行程序的执行；而动态链接这个过程却没有把内容链接进去，而是在执行的过程中，再去找要链接的内容，生成的可执行文件中并没有要链接的内容，所以当你删除动态库时，可执行程序就不能运行。所以动态链接生成的可执行文件要比静态链接生成的文件要小一些。

各自的优缺点：

- 静态链接库执行速度比动态链接库快。（执行过程不需要找链接的内容）
- 动态链接库更节省内存。（未写入要链接的内容）

深入浅出静态链接和动态链接

iostream

头文件	函数和描述
<iostream>	该文件定义了 cin、cout、cerr 和 clog 对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。
<iomanip>	该文件通过所谓的参数化的流操纵器（比如 setw 和 setprecision），来声明对执行标准化 I/O 有用的服务。
<fstream>	该文件为用户控制的文件处理声明服务。

标准输入流 cin

```
1  char name[50];
2  short age;
3  cout << "请输入您的名称与年龄： ";
4  cin >> name >> age;
5  //C++ 编译器根据要输入值的数据类型，选择合适的流提取运算符来提取值，并把它存储在给定的变量中。
6  //流提取运算符 >> 在一个语句中可以多次使用
```

标准输出流 cout

```
1  char str[] = "Hello C++";
2  cout << "Value of str is : " << str << endl;
```

标准错误流 cerr

cerr 对象附属到标准错误设备，通常也是显示屏，但是 cerr 对象是非缓冲的，且每个流插入到 cerr 都会立即输出。

```
char str[] = "Unable to read....";
cerr << "Error message : " << str << endl;
```

标准日志流 clog

clog 对象附属到标准错误设备，通常也是显示屏，但是 clog 对象是缓冲的。这意味着每个流插入到 clog 都会先存储在缓冲在，直到缓冲填满或者缓冲区刷新时才会输出。

```
char str[] = "Unable to read....";
clog << "Error message : " << str << endl;
```

```
#include <iostream>
using namespace std;
int main()
{
   cout << "cout" << endl;
   cerr << "cerr" << endl;
   return 0;
}
```

linux下命令行输入：

```
g++ main.cpp -o a
./a >> test.log
```

终端输出“cerr”，
打开test.log，里面只有一行字符串“cout”。

- cout默认情况下是在终端显示器输出，cout流在内存中开辟了一个缓冲区，用来存放流中的数据，当向cout流插入一个endl，不论缓冲区是否满了，都立即输出流中所有数据，然后插入一个换行符。cout可以被重定向到文件。
- cerr不经过缓冲而直接输出，一般用于迅速输出出错信息，是标准错误，默认情况下被关联到标准输出流，但它不被缓冲，也就说错误消息可以直接发送到显示器，而无需等到缓冲区或者新的换行符时，才被显示。不被重定向。
    - 有时程序调用导致栈被用完了，此时如果使用cout 会导致无内存输出。使用cerr 会在任何情况下输出错误信息。
    - 不被缓冲就是你打一个字符就马上在显示器显示，而不是等到endl才打印。
- clog流也是标准错误流，作用和cerr一样，区别在于cerr不经过缓冲区，直接向显示器输出信息，而clog中的信息存放在缓冲区，缓冲区满或者遇到endl时才输出。clog用的少。

初始化

```
int a(5); // 构造函数语法，比如复数需要初始化两个值
complex<double> purei(0, 7);
int b = 6; // C 风格的运算符（=）初始化

//关于指针
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr; // 合法
cout << p[1] << endl;

vector<int> vec(arr, arr + 5);
vector<int> *pv = &vec; // 合法
//而 vector<int> *pv = vec; 不合法，因为 vec 是 vector 型对象，它的名字不是首地址
cout << vec[1] << endl; // 注意与数组的差别
```

new 与 vector 定义多维变长数组并初始化可以访问该目录下的cpp/init.cpp

逗号表达式

```
// 【例 1】
cout << a_string << (cnt % line_size ? ' ' : '\n');
// 若某一行计数不超过 line_size，就打印空格，否则换行。用来限制某一行字符打印的个数
//【ESS 11】

// 【例 2】
fun(f2(v1,v2),(v3,v4,v5),(v6,max(v7,v8)));
// fun 函数的实参个数有几个?
// 答：三个。
// 第一个是 f2 的返回值，第二个是逗号表达式(v3,v4,v5)的值，第三个是逗号表达式(v6,max(v7,v8))的值。
```

指针

如果指针不指向任何对象，则【提领】操作（也可称为【解引用】，即取指针指向的内容）会导致未知的执行结果。这意味着在使用指针时，必须在提领前确定它的确指向某对象。

一个未指向任何对象的指针，其地址为0，也被称为 nullptr 指针。我们可以在定义阶段便初始化指针，令其值为 0.

```
int *p = 0; // 初始化指针地址为0
if (p && *p != 1024) // 如果 p 地址不为零，才能有 *p 操作
    *p = 1024;
// 当 p 的地址为 0，直接提领，会报错
// 【ESS 28】
```

野指针：

野指针指向一个已删除的对象或未申请就访问受限内存区域。

与空指针不同，野指针无法通过简单地判断是否为 nullptr 避免。

成因：

- 指针变量未初始化时不会自动成为 nullptr，而是一个随机值；
- 指针指向的内存被释放后，指针未赋值为 nullptr；
- 指针操作超越变量的作用域，比如函数返回栈内存的指针或引用。

指针与数组名的区别

- 修改内容上的差别：

```
char a[] = "hello";
a[0] = 'H';
char *p = "world"; // p 指向常量字符串，该字符串存储在文字常量区，不可更改
// p[0] = "W"     // 所以这个赋值有问题
```

```
int main() {
    char a[] = "hello";
    char *p = a; // 这样让指针指向数组 a，而非常量字符串，就可以修改了
    p[0] = 'H';
    a[1] = 'E';
    printf("%d", sizeof(p)); // 64 位机器
    std::cout << p << std::endl;
    return 0;
}
//输出：8HEllo
```

- sizeof

```
sizeof(a); // 输出6，包含 '\0'
sizeof(p); // 64 位机器输出 8
/*
指针类型对象占用内存为一个机器字的长度：
这意味着在16位cpu上是16位 = 2字节
32位cpu上是32位 = 4字节
64位cpu上就是64位也就是8字节了
*/
```

数组指针与指针数组

```
type (*p)[]; // 数组指针
/*
type a[]，a 是一个数组，数组内元素都是 type 类型，将 a 换成 (*p)，
可以理解为 （*p）是一个数组，数组内元素都是 type 类型，那么 p 就是指向这样的数组的指针，即数组指针。
*/

type *p[]; // 指针数组
/*
type *p[]即(type *)p[]，则 p 是一个 type* 类型的数组(类比int a[10]，a 是 int 型数组)
即 p 是一个数组，数组内元素都是指针（ type *）
*/

// 在后文的介绍中使用右左法则，更容易理解
```

稍微复杂一些的指针数组：

```
const int seq_cnt = 6;
vector<int> *seq_addrs[seq_cnt] = {
  &fibonacci, &lucas, &pell,
  &triangular, &square, &pentagonal
};
// seq_addrs 是一个数组，存储 vector<int> * 型的指针，
// seq_addrs[0] 的内容为指针，该指针指向 fibonacci，
// 而 fibonacci 的类型为 vector<int>。
// 【ESS 29】
```

指针函数与函数指针

指针函数：返回类型是指针的函数，如

```
char *StrCat(char *ptr1, char *ptr2)
{
    char *p;
    do something;
    return p;
}
```

函数指针：指向函数的指针，如

```
#include <stdio.h>
int max(int x, int y)
{
    return x > y? x: y;
}
int main()
{
    // int max(int x, int y);
    int (*ptr)(int, int);
    ptr = max;
    int max_num = (*ptr)(12, 18);  // int max_num = ptr(12, 18);也是对的
    printf("%d", max_num);
    return 0;
}
// 打印 18。
```

```
int (*ptr)(); // “()” 表明该指针指向函数，函数返回值为 int
int (*ptr)[3]; // “[]” 表明该指针指向一维数组，该数组里包含三个元素，每一个元素都是int类型。
```

数组名指向了内存中一段连续的存储区域，可以通过数组名的指针形式去访问，也可以定义一个相同类型的指针变量指向这段内存的起始地址，从而通过指针变量去引用数组元素。

每一个函数占用一段内存区域，而函数名就是指向函数所占内存区的起始地址的函数指针（地址常量），故 函数指针解引用与不解引用没有区别 。通过引用函数名这个函数指针让正在运行的程序转向该入口地址执行函数的函数体，也可以把函数的入口地址赋给一个指针变量，使该指针变量指向该函数。

复杂指针声明

函数指针数组

```
const vector<int>* (*seq_array[])(int) = {
    fibon_seq, lucas_seq, pell_seq,
    triang_seq, square_seq, pent_seq
};
```

seq_array 是一个可以持有六个函数指针的指针数组，第一个元素指向函数 fibon_seq()，该函数原型为 `const vector<int> *fibon_seq(int);`。

【ESS 62】

解读复杂指针使用右左法则：首先从未定义的标识符所在的圆括号看起，然后往右看，再往左看。每当遇到圆括号就调转阅读方向。一旦解析完圆括号里的东西，就跳出圆括号。重复这个过程，直到整个声明解析完毕。

```
int (*func)(int *p, int (*f)(int*));
// func左边有一个*表明func是一个指针，跳出圆括号看右边，右边有括号，说明func是一个函数指针，
// 指向的函数接收两个形参，分别是整型指针 int * 和函数指针 int (*f)(int*)，返回值为 int。

int (*func[5])(int *p);
// func 是一个数组，含5个元素，左边*号，表明 func 是一个指针数组（由于[]优先级高于*，func先跟[]结合，然后*修饰func[5]，故该数组的元素都是指针）。
//再往右看，是括号，说明 func 里的指针是函数指针，函数指针所指向的函数接收 int * 型参数并返回 int。

int (*(*func)[5])(int *p);
// func 是一个指针，指向含有 5 个元素的数组，数组里的元素都是指针，而且都是函数指针，
// 函数指针指向的函数接收 int * 型形参并返回 int。

int (*(*func)(int *p))[5];
// func 是一个指针，该指针指向一个函数，该函数接收 int * 型参数返回一个指针，
// 返回的指针指向一个数组，该数组含有 5 个元素，每一个元素都是 int 型。
```

指针(pointer)与引用(reference)

```
int ival = 1024;
int *pi = &ival;  // pointer
int &rval = ival;  // reference
```

- 引用是别名，而指针是地址。指针可以被赋值，以指向另一个不同的对象，而引用只能在定义时被初始化一次，以后不能修改，但引用的那个对象内容却可以改变。可以把引用理解为指针常量，而普通指针为指针变量。

- 引用不能为空，指针可以为空。故在使用上，指针可能（也可能不）指向一个对象，提领时一定要先确定其值非 0。而引用，则必定会代表某个对象，所以不需要作此检查。
- 从内存分配上来看，程序为指针变量分配内存区域，而不为引用分配内存区域。
- 引用使用时无需解引用(*)，指针需要解引用；
- 引用没有 const，指针有 const；
- “sizeof 引用” 得到的是所指向的变量(对象)的大小，而 “sizeof 指针” 得到的是指针本身的大小；
- 指针可以有多级，但是引用只能是一级（int **p；合法 而 int &&a是不合法的）
- 指针和引用的自增(++)运算意义不一样；

```
int a = 0;
int &b = a;
int *p = &a;
b++; // 相当于 a++; b 只是 a 的一个别名，和 a 一样使用。
p++; // p 指向 a 后面的内存
(*p)++; // 相当于 a++
```

（在二进制层面，引用一般是通过指针来实现的，只不过编译器帮我们完成了转换。总的来说，引用既具有指针的效率，又具有变量使用的方便性和直观性。）

【ESS 46，47】

引用作为函数参数

- 传递引用给函数与传递指针给函数的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。
- 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配临时存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。
- 使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用 "*指针变量名" 的形式进行运算，这很容易产生错误且程序的可读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

常引用作为函数参数

如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用

```
void foo(const string &s) {
    cout << s << endl;
}

// 如果形参里的 const 去掉，程序就报错，
// 因为 ss 与 "world!" 都是常量，你不能把一个 const 类型转换成非 const 类型。
// 所以 foo() 形参必定要用 const 修饰。

int main() {
    const string ss("hello ");
    foo(ss);
    foo("world!");
}
```

函数返回引用与返回值

返回引用的好处：在内存中不产生被返回值的副本。

同时注意，正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生 runtime error!

【注意】

- 最好不要返回局部变量的引用；
- 最好不要返回函数内部 new 分配的内存的引用；
- 流操作符重载返回值应为引用；
- 全局变量和局部静态变量的返回值可以是引用；
- 可以返回类成员的引用；

指针运算

*&p和&*p

- &：取出变量的存储地址。对指针变量 p，&p 取指针变量 p 所占用内存的地址，可以说是二级指针。
- *：引用指针所指向单元的内容。

```
1、*&p 等价于*(&p)
2、&*p 等价于&(*p)
p 是 int 变量，那么 *&p = p，而 &*p 是非法的。因为 *p 非法
p 是 int* 变量，那么 *&p = p，&*p = p，都是 p
```

p+(或-)n

指针加减一个数，是指该指针上移或下移n个数据之后的内存地址。即：

```
p +(或-) n * sizeof(type);
```

p++，(p++)， (*p)++，*++p，++*p

- *p++ 和 *(p++) 没有区别，应该理解为，由于后++优先级高于*，应该先 p++，后解引用，但因为 ++ 在变量之后，运算得先用再自增，所以先执行*p，p再自增。这又与 (*p)++ 有区别，这里面先 *p，*p 再自增而不是 p 自增。

```
// 一个 ++ 在变量后，先用再自增的例子。
int a = 3;
int b = 3;
printf("%d",a++); //打印3,但a值已经变成4了。先用a，a再自增，所以就是先打印3,打印完再自增。
printf("%d",++b); //先自增，自增完，再用。
```

- *++p 等价于 *(++p)，p 先自增，自增完后再取 *p 的值，即取下一个元素的值，而不是当前元素。
  ++*p，*p自增。

  【单目运算* 和前++是同等优先级，后缀运算后++优先级大于单目运算*】

i++ 与 ++i 的效率

- 对于内置数据类型，两者差别不大；
- 对于自定义数据类型（如类），++i 返回对象的引用，而 i++ 返回对象的值，导致较大的复制开销，因此效率低。

sizeof

sizeof 计算普通变量与指针所占空间的大小

```
// 用 sizeof 计算下列变量所占空间的大小，环境在 win32 操作系统上
// win32上，char:1，short:2, int:4，long:4
char str[] = "hello";      // 6
char *p = str;             // 4
int n = 10;                // 4
void fun(char str[100])
{
    sizeof(str);           // 4，函数形参对传入的数组是按照指针处理的，在函数体内可以通过修改 str 修改函数外的数组
}
void *p = malloc(100);     // 4, p 指向 100 个字节的堆内存，但本质上还是指针。
```

sizeof 计算空类的大小

```
class A {};
cout << sizeof A << endl;  // 1
// 空类不包含任何信息，本来求 sizeof 的时候应该是 0，但是当我们声明该类型的实例的时候，它必须在内存中占用空间，否则无法使用。至于占多少空间由编译器来决定，在 vs 中占 1 个字节
```

```
class A
{
public:
    A() {};
    ~A() {};
    // virtual void f() {}; //【1】
};
cout << sizeof A << endl;
// 无虚函数，依旧占用空间 1 字节
// 如果去点【1】处注释，含虚函数，得占用内存 4。有虚函数时，默认有一个指针指向虚函数表。该指针占 4 字节。
// 普通成员函数不占用类的空间。类的函数是该类所有实例共享的，调用时通过隐藏的 this 指针和类的实例相关联，
// 普通成员函数代码编译后存储在程序代码区，根本就不在类实例中，所以不占实例空间。
```

sizeof 计算类对象与结构体、联合体所占空间的大小

```
class A          // 1
{
public:
    char ch;
};

```

```
class B        // 4 + 2 + 填充2 = 8
{
public:
    int i;
    short j;
};

class C        // 4 + 2 + 1 + 1 = 8
{
public:
    int i;
    short j;
    char c1;
    char c2;
};

class D        // 1 + 填充3 + 4 + 1 + 填充1 + 2 = 12
{
public:
    char c1;
    int i;
    char c2;
    short j;
};
// 【注意】类 C，D 的区别在于变量定义的顺序不一样
D d;
printf("%x, %x, %x, %x, %x\n", &d, &d.c1, &d.i, &d.c2, &d.j);
// 输出：12ffed4, 12ffed4, 12ffed8, 12ffedc, 12ffede
```

```
// 注意 S1 与 S2 也就是成员变量定义时顺序不一样而已
struct S1       // 4 + 1 + 1 + 填充2 = 8
{
    int a1;
    char c1;
    char c2;
};

struct S2       // 1 + 填充3 + 4 + 1 + 填充3 = 12
{
    char c1;
    int a1;
    char c2;
};
```

```
15
16  S2 s2;
17  printf("%x, %x, %x, %x\n", &s2, &s2.c1, &s2.a1, &s2.c2);
18  // 输出：10ff794, 10ff794, 10ff798, 10ff79c
```

```
1   union u1        // 8
2   {
3     double a;
4     int b;
5   };
6
7   union u2        // 13，对齐方式是1，13 是 1 的整数倍
8   {
9     char a[13];
10    char b;
11  };
12
13  union u3        // 16，对齐方式是 4，16 是 4 的整数倍
14  {
15    char a[13];
16    int b;
17  };
```

这一切都与字节对齐有关：

- 结构体变量的首地址能够被其最宽基本类型成员的大小所整除；
- 结构体每一个成员相对于结构体首地址的偏移量都是成员大小的整数倍，如有需要，编译器会在成员之间加上填充字节；
- 结构体的总大小为结构体最宽基本数据类型成员大小的整数倍，如有需要，编译器会在最末一个成员之后加上填充字节。

联合体（共用体），类同理。

sizeof 与 strlen 区别

- sizeof 是操作符，strlen 是函数；
- 数组做 sizeof 参数不退化，传递给 strlen 就退化成指针（函数的形参将数组按指针处理）；

```
char *s = "hello";
char str[20] = "hello";
cout << sizeof(s) << ' ' << sizeof(str) << endl;  // sizeof 对字符指针和字符数组的处理不同
// cout << sizeof s << ' ' << sizeof str << endl;
// sizeof 后面不加括号也正确，因为 sizeof 是运算符不是函数，但用 sizeof 求类型大小，必须加括号，比如 sizeof(int)
cout << strlen(s) << ' ' << strlen(str) << endl;
/* 打印
4 20
5 5
*/
```

- sizeof 在编译阶段就计算出来了，所以可以定义数组的维度，而strlen是在运行时候才计算出来；

strcpy 与 memcpy 区别

- 复制的内容不同。strcpy 只能复制字符串，而 memcpy 可以复制任意内容，例如字符数组、整型、结构体、类等。
- 复制的方法不同。strcpy 不需要指定长度，它遇到被复制字符的结束符 "\0" 才结束，所以容易溢出。memcpy 则是根据其第 3 个参数决定复制的长度。
- 用途不同。通常在复制字符串时用 strcpy，而需要复制其他类型数据时则一般用 memcpy 。

作用域与生存周期

c++ 变量有两个属性非常重要：作用域和生存周期。

花括号作用域

```
// 花括号可以看作是作用域标识符，除了在写函数时候用到，它还有一个作用是表示变量的作用域：
{           // 外花括号
  int a = 0;
  {         // 内花括号
    int b = 0;
    a = 1;   //【1】正确，对外花括号内的 a 重新赋值
  }
  // b = 1;    // 错误，b 已经被销毁了
  cout << a << endl;
  /*
  输出为 1。如果将【1】处 a = 1； 改成 int a = 1；此时输出为 0。
```

```
    因为前者 a = 1 表明修改的是外花括号内的 a，而后者表明定义一个局部变量 a 并初始化，
    内花括号的 a 屏蔽了外花括号的 a，而 cout 打印的是外花括号的 a
    */
}

// 对于 if, while, for 在花括号里定义的变量，出了花括号就被释放了，
    int a = 1;
    for (int a = 2; a == 2; ++a)
    {
        cout << a << endl; //for 中定义的 a 作用域在花括号内，屏蔽 for 外定义的 a
    }
    cout << a << endl;
/* 打印
2
1
*/
```

内存分配

一个程序将操作系统分配给其运行的内存块分为 5 个区域：

- 静态区（全局区）：存放程序的全局变量和静态变量。初始化的全局变量和静态变量在一个区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。
- 堆区：存放程序的动态数据。
- 栈区：存放程序的局部数据，即各个函数的参数和局部变量等。函数结束，自动释放。
- （文字）常量区：常量字符串存放的区域，程序结束后由系统释放。
- 代码区：存放程序的代码，即程序中的各个函数代码块。

```
int a = 0; // 全局初始化区
char *p1; // 全局未初始化区
int main() {
    int b;  // 栈
    char s[] = "abc"; // 栈
    char *p2; // 栈
    char *p3 = "123456"; // 字符常量 “123456\0” 在常量区，p3 在栈上。
    static int c = 0; // 全局（静态）初始化区
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20); // 分配得来的 10 和 20 字节的区域在堆区。
    strcpy(p1, "123456"); // “123456\0” 放在文字常量区，编译器可能会将它与 p3 所指向的 "123456\0" 优化成一个地方
}
```

局部变量和全局变量

局部变量也称为内部变量，它是在函数内定义的。其作用域仅限于函数内，离开该函数后再使用这种变量是非法的。

全局变量也称为外部变量，它是在函数外部定义的变量。它不属于哪一个函数，它属于一个源程序文件。其作用域是整个源程序。在函数内部，局部变量可以屏蔽全局变量。如果一个全局变量用 static 修饰，它就是静态全局变量，它的作用域是该文件范围（称为文件作用域，即其它文件不能使用它）

静态存储与动态存储

变量的生存周期只与变量的存储位置（存储类别）有关。可以分为：

- 静态存储方式（在程序运行期间，系统对变量分配固定的存储空间）
- 动态存储方式（在程序运行期间，系统对变量动态（不固定）的分配存储空间）

而变量的存储类别可以分为静态存储和动态存储

- auto 自动变量(动态存储方式)
- static 静态变量(静态存储方式)
- register 寄存器变量(动态存储方式)
- extern 外部变量(静态存储方式)

C++变量保存在堆还是栈?

- 如果对象是函数内的非静态局部变量，则对象，对象的成员变量保存在栈区。
- 如果对象是全局变量，则对象，对象的成员变量保存在静态区。
- 如果对象是函数内的静态局部变量，则对象，对象的成员变量保存在静态区。
- 如果对象是 new 出来的，则对象，对象的成员变量保存在堆区。

```
1  void foo() {
2     int* p = new int[5];
3  }
4  // 在栈内存中存放了一个指向一块堆内存的指针 p
5  // 程序会先确定在堆中分配内存的大小，然后调用 operator new 分配内存
6  // 然后返回这块内存的首地址，存入栈中
```

堆与栈的效率

栈是程序启动的时候，系统分好了给你的，你自己用，系统不干预。

堆是用的时候才向系统申请的，用完了还回去，这个申请和交还的过程开销相对就比较大了。

堆相对于栈，效率低，多次分配（malloc/new）容易产生碎片，使用时最好结合相关操作系统（Linux、Windows、RTOS ）使用，因为系统针对内存管理有专门的优化算法，减少内存碎片。堆虽然有一定的缺点，但其最大的优点是使用灵活，而且堆容量大，一般需要申请比较大的内存块时，都会从堆中申请。

new 与 delete

堆内存（空闲空间）里的内存分配通过 new 表达式来完成，释放通过 delete 表达式来完成。堆内存由程序员自行管理。

```
int *p1 = new int; // 未初始化
int *p2 = new int(1024); // 指定初值, p2 指向一个 int 对象，其值为 1024
int *p3 = new int[1024]; // 从 heap 中分配一个数组，含有1024个元素，p3 指向数组第一个元素
delete p1;
delete p2; // 对于普通数据类型， delete p 与 delete [] p 作用效果一样。
delete [] p3 // 复杂对象，必须用 delete [] p 来释放内存
// 如果不使用 delete，由 heap 分配而来的对象就永远不会被释放，这被称之为内存泄漏。
```

【ESS 49，50】

new/delete 与 malloc/free 关系

malloc/free 是 C/C++ 的标准库函数，new/delete 是 C++ 的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 malloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free（malloc 只能申请内存，不能在申请内存的时候对所申请的内存进行初始化工作，而构造函数可以。free 只能释放内存，而如果析构函数设计得好的话，在释放内存的同时还可以完成额外的其他工作）。因此 C++ 语言需要一个能完成动态内存分配和初始化工作的运算符 new ，以及一个能完成清理与释放内存工作的运算符 delete 。

- new / new[]：完成两件事，先底层调用 malloc 分配内存，然后调用构造函数（创建对象）。
- delete/delete[]：也完成两件事，先调用析构函数（清理资源），然后底层调用 free 释放空间。
- new 在申请内存时会自动计算所需字节数，而 malloc 则需我们自己输入申请内存空间的字节数。

malloc/free 的使用

```cpp
#include <iostream>
using namespace std;
int main() {
    int *score, num;
    cin >> num;
    if ((score = (int *)malloc(sizeof(int) * num)) == nullptr)
        cerr << "fail" << endl;
    else {
        do something;
        free(score); // 释放内存
    }
}
```

宏

```cpp
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define POW(x) ((x) * (x))
// 为什么用括号？因为宏只是简单的字符替换
```

头文件

- 头文件的扩展名习惯上是.h，标准库例外。
- 函数的定义只能有一份，倒是可以有多份声明。我们不能把函数的定义放在头文件，因为一个程序的多个代码文件可能都会包含这个头文件。但只定义一份的规则有一个例外，内联函数。为了能够扩展内联函数的内容，以便在每个调用点上，编译器都取得其定义，必须将内联函数的定义放在头文件中，而不是放在各个不同程序代码文件中（如果两个函数在定义时函数名和参数列表都一样(返回类型可以不一样)，则会出错，因为重载要保证参数列表不一样）。
- 一个对象和变量同函数一样，也只能在程序中定义一次，因此也应该将定义放在程序代码文件中，而不是头文件中。一般地，加上 extern 就可以放在头文件中作为声明了。

```cpp
// 如果想声明一个变量而非定义它，就在变量名前添加 extern 关键字，而且不要显式地初始化变量
extern int x; // 声明 x 而非定义
int y;      // 声明并定义 y
```

- `const int a = 6;` 就可以放入头文件中，因为 const object 就和 inline 函数一样，是”一次定义“规则下的例外。因为 const 定义一出文件外便不可见（文件作用域），这意味着可以在多个不同的文件中加以定义。
- 头文件用 <> 表明此文件被认为是标准的或项目专属的头文件，编译器搜索此文件时，会先在系统默认的磁盘目录下寻找；头文件用 ” “ 表明此文件被认为是用

户提供的头文件，会先在包含此文件的磁盘目录开始寻找，找不到再去系统默认的目录下寻找。

【ESS 63，64】

#ifndef/#define/#endif作用

防止头文件的重复包含和编译

```
#ifndef A_H // 意思是 if not define a.h"  如果不存在a.h
#define A_H // 就引入 a.h
#include <math.h> // 引用标准库的头文件
…
#include  “header.h”  // 引用非标准库的头文件
…
void Function1(…); // 全局函数声明
…
class Box // 类结构声明
{
…
};
#endif // 最后一句应该写 #endif，它的作用相当于 if 的反花括号 '}'
```

extern “C”

作为C语言的扩展，C++ 保留了一部分过程式语言的特点，因而它可以定义不属于任何类的全局变量和函数。但是，C++ 毕竟是一种面向对象的设计语言，为了支持函数的重载，C++ 对全局函数的处理方式有着明显的不同。

首先看一下 C++ 对类似C的函数是怎样编译的：

作为面向对象的语言，C++ 为了支持函数重载，函数在被 C++ 编译后在符号库中的名字与 C 语言的不同。假如某个函数的原型为 void foo(int x, int y);，该函数被 C 编译器编译后在 符号库 中的名字为 _foo，而 C++ 编译器则会产生 _foo_int_int 之类的名字。_foo_int_int 这样的名字是包含了函数名以及形参，C++就是靠这种机制来实现函数重载的。 如果在 C 中连接 C++ 编译的符号时，就会因找不到符号问题而发生连接错误。

被 **extern “C”** 修饰的函数或者变量是按照 **C** 语言方式编译和链接的，所以可以用一句话来概括 **extern “C”** 的真实目的：实现 **C++** 与 **C** 的混合编程。

typedef 声明

可以使用 typedef 为一个已有的类型取一个新的名字。

typedef 可以声明各种类型名，但不能用来定义变量。用 typedef 可以声明数组类型、字符串类型，使用比较方便。

用 typedef 只是对已经存在的类型增加一个类型名，而没有创造新的类型。

```
typedef int feet;     // 告诉编译器，feet 是 int 的另一个名称：可以理解 feet 为 int 的别名
feet distance;       // 它创建了一个整型变量 distance

typedef int A[];     // 定义了数组类型，数组大小由初始化时候决定
typedef int B[9];    // 定义了大小为9的数组类型，用 B 定义并初始化数组时，元素个数不应超过9
A arra = { 0, 1, 2, 3, 4 };      // arra 长度为 5
cout << arra[3] << ' ' << sizeof arra/sizeof arra[0] << endl; // 打印 3 5
B arrb = { 0, 1, 2, 3, 4 };      // 实际是 {0， 1， 2， 3， 4， 0， 0， 0， 0}
cout << arrb[8] << ' ' << sizeof arrb / sizeof arrb[0] << endl; // 打印 0 9

typedef int (*pfun)(int x, int y); // 定义一个 pfun 类型，表明一个函数指针类型
int fun(int x, int y);
pfun p = fun;                // 定义了一个 pfun 类型的函数指针，并指向函数 fun
int ret = p(2, 3);

typedef struct Student          // 定义了一个 ST 类型的结构体，下次定义结构体可直接用 ST 定义，
{                          // ST Lux = { 123, 'F' };
  int id;
  char sex;
}ST;
```

decltype 类型指示符

C++11新增 `decltype` 类型指示符，作用是选择并返回操作数的数据类型，此过程中编译器不实际计算表达式的值。

```
decltype(f()) sum = x; // sum has whatever type f returns
```

`decltype` 处理顶层 `const` 和引用的方式与 `auto` 有些不同，如果 `decltype` 使用的表达式是一个变量，则 `decltype` 返回该变量的类型（包括顶层 `const` 和引用）。

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0;   // x has type const int
decltype(cj) y = x;   // y has type const int& and is bound to x
decltype(cj) z;   // error: z is a reference and must be initialized
```

如果 decltype 使用的表达式不是一个变量，则 decltype 返回表达式结果对应的类型。如果表达式的内容是解引用操作，则 decltype 将得到引用类型。如果 decltype 使用的是一个不加括号的变量，则得到的结果就是该变量的类型；如果给变量加上了一层或多层括号，则 decltype 会得到引用类型，因为变量是一种可以作为赋值语句左值的特殊表达式。

decltype((var)) 的结果永远是引用，而 decltype(var) 的结果只有当 *var* 本身是一个引用时才会是引用。

枚举类型

如果一个变量只有几种可能的值，可以定义为枚举(enumeration)类型。每个枚举元素在声明时被分配一个整型值，默认从 0 开始，逐个加 1。也可以在声明时将枚举元素的值一一列举出来。

注意

- 枚举元素是常量，除了初始化时不可给它赋值。
- 枚举变量的值只可取列举的枚举元素值。

```
enum 枚举名{
    标识符[=整型常数],
    标识符[=整型常数],
...
    标识符[=整型常数]
} 枚举变量;
```

如果枚举没有初始化,，即省掉 "=整型常数" 时，则从第一个标识符开始。

```
enum color { red, green, blue } c;
c = blue; // c 为枚举变量，把枚举元素 blue 赋给 c，此时 c = 2
// 若直接这样赋值 c = 2；就会报错！你只能把{ red, green, blue }赋值给 c。
// blue = 2；报错！blue 是常量而不是变量。
// 未初始化默认 red = 0, green = 1, blue = 2
```

若给某一个标识符赋值如 green = 5

```
enum color { red, green = 5, blue } c;
c = blue;  // 6
color d;  // d 也为枚举变量
d = red;  // d = 0
// 部分值初始化，要满足后面的值比前面值大 1，此时 red 默认为 0
// green 初始化为 5 ，blue 要满足比 green 大 1 即 blue = 6
```

C结构体、C++结构体、C++类的区别

C 结构体与 C++ 结构体

- C 语言中的结构体不能为空，否则会报错
- C 语言中的结构体只涉及到数据结构，而不涉及到算法，也就是说在 C 中数据结构和算法是分离的。换句话说就是 C 语言中的结构体只能定义成员变量，但是不能定义成员函数（虽然可以定义函数指针，但毕竟是指针而不是函数）。然而 C++ 中结构体既可以定义成员变量又可以定义成员函数， C++ 中的结构体和类体现了数据结构和算法的结合。

C++ 中结构体与类

- 相同之处： 结构体中也可以包含函数；也可以定义 public、private、protected 数据成员；定义了结构体之后，可以用结构体名来创建对象。也就是说在 C++ 当中，结构体中可以有成员变量，可以有成员函数，可以从别的类继承，也可以被别的类继承，可以有虚函数。总的一句话：class 和 struct 的语法基本相同，从声明到使用，都很相似，但是 struct 的约束要比 class 多，理论上，struct 能做到的 class 都能做到，但 class 能做到的 stuct 却不一定做的到。
- 区别：对于成员访问权限和继承方式，class 中默认的是 private，而 struct 中则是 public。class 还可以用于表示模板类型，struct 则不行。

结构体与联合体（共用体）的区别

- 结构和联合都是由多个不同的数据类型成员组成, 但在任何同一时刻, 联合中只存放了一个被选中的成员（所有成员共用一块地址空间）, 而结构体的所有成员都存在（不同成员的存放地址不同）。 （在 struct 中，各成员都占有自己的内存空间，它们是同时存在的。一个 struct 变量的总长度等于所有成员长度之和。在 Union 中，所有成员不能同时占用它的内存空间，它们不能同时存在。Union 变量的长度等于最长的成员的长度。）
- 对联合体不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于结构的不同成员赋值是互不影响的。

类 & 对象

类包含对象所需的数据，以及描述用户与数据交互所需的操作。

成员函数可以定义在类内部，或者单独使用 范围解析运算符（域区分符） :: 来定义。在类定义中定义的成员函数把函数声明为 内联 的，即使没有使用 inline 标识符。

类的成员名和方法的参数名不能相同，建议成员名加上 'm' 前缀或者末尾加上 '_'。

【PLUS 353】

类访问修饰符

- public：公有成员在程序中类的外部是可访问的。可以不使用任何成员函数来设置和获取公有变量的值。
- private：私有成员变量或函数在类的外部是不可访问的，甚至是不可查看的。只有类和友元函数可以访问私有成员（派生类也不能访问）。
- protected：保护成员变量或函数与私有成员十分相似，但有一点不同，保护成员在派生类（即子类）中是可访问的。

封装继承多态

封装

封装是实现面向对象程序设计的第一步，封装就是将数据或函数等集合在一个个的单元中（称之为类）。

封装的意义在于保护或者防止代码（数据）被无意中破坏。

继承

继承主要实现代码重用，节省开发时间。子类可以继承父类的一些东西。

有 public，protected，private 三种继承方式，它们相应地改变了派生类的用户以及派生类的派生类的访问权限。

- public 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：public, protected, private
- protected 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：protected, protected, private
- private 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：private, private, private

但无论哪种继承方式，上面两点都没有改变：

- private 成员只能被本类成员（类内）和友元访问，不能被派生类访问；
- protected 成员可以被派生类访问。

多态

同一个方法在派生类和基类中的行为是不同的，即方法的行为取决于调用该方法的对象。有两种重要的机制可以实现多态公有继承：

- 在派生类中重新定义基类的方法
- 使用虚方法

【注意】在派生类中重新定义基类的方法，会导致基类方法被隐藏（函数隐藏），这不是重载，重载是一个类中的方法与另一个方法同名，但是参数表不同，这种方法称之为重载方法。

重载与重写

- 重载（overload）：是指允许存在多个同名函数，而这些函数的参数列表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）返回值类型随意。与多态无关。
- 重写（overried，覆盖、覆写）：是指子类重新定义父类虚函数的方法。与多态有关。

构造函数与析构函数

```
class Line
{
  private:
  double length;

  public:
    void setLength(double len);
    Line(double len);  // 构造函数的名称与类的名称完全相同。不会返回任何类型，也不会返回 void，常用于赋初值
    ~Line();        // 析构函数，函数名与类完全相同，只是在前面加了一个波浪号（~）作为前缀，它不会返回任何值，也不会返回void，也不能带有任何参数。析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源。
    double Line::getLength(void) //方法可以定义在类中
    {
      return length;
    }
};

Line::Line(double len) // 方法也可以通过范围解析运算符定义在类外，这是构造函数的具体实现，注意函数前无类型
{
  cout << "Object is being created, length = " << len << endl;
  length = len; //初始化属性
}

Line::~Line(void)
{
  cout << "Object is being deleted" << endl;
}
```

也可以使用初始化列表来初始属性

```
Line::Line( double len): length(len)
{
   cout << "Object is being created, length = " << len << endl;
}
```

```
// 同时初始化多个值
Person::Person( double name, double age, double job): Name(name), Age(age), Job(job)
// 将参数 name, age, job 初始化给属性 Name, Age, Job
{
 ....
}
```

构造函数不同于类方法，因为它创建新的对象，而其他类对象只是被现有的类调用。这是构造函数不能被继承的原因之一。 继承意味着派生类继承基类的成员函数和成员变量，然而，在构造函数完成其工作之前，对象并不存在。

一定要使用显式析构函数来释放类构造函数使用 new 分配的所有内存，并完成类对象所需的任何特殊的清理工作。对于基类即使它不需要析构函数，也应提供一个虚析构函数。

初始化派生类把基类中所有的成员继承过来，除了构造函数和析构函数。友元函数不属于类，它只是给类开了一个后门，自然不能被继承。子类继承父类，那么默认的，就是继承了父类的成员函数和成员变量。

初始化子类时，会先自动调用父类的构造函数，然后才调用子类的构造函数。

析构时，按相反顺序进行。

构造从类层次的最根处开始，在每一层中，首先调用基类的构造函数，然后调用成员对象的构造函数。析构则严格按照与构造相反的次序执行，该次序是唯一的，否则编译器将无法自动执行析构过程。

【PLUS 524，525，527】

拷贝（复制）构造函数

```
class Line
{
   public:
      Line(int len);  //构造函数
      Line(const Line &obj);  //拷贝构造函数
      ~Line();  //  析构函数

   private:
      int *ptr;
```

```
};

// 成员函数定义，包括构造函数
Line::Line(int len)
{
    cout << "调用构造函数" << endl;
    // 为指针分配内存
    ptr = new int;
    *ptr = len;
}

Line::Line(const Line &obj)
{
    cout << "调用拷贝构造函数并为指针 ptr 分配内存" << endl;
    ptr = new int;
    *ptr = *obj.ptr; // 拷贝值
}

Line::~Line(void)
{
    cout << "释放内存" << endl;
    delete ptr;
}
```

什么情况使用拷贝构造函数

```
class Test
{
public:
    int a;
    Test(int x) : a(x) { cout << "Test(int x)" << endl; }
    Test(const Test& t)
    {
        cout << "copy constructor" << endl;
        a = t.a;
    }
};

void fun1(Test test)
{
    cout << "fun1()" << endl;
}

```

```
18  Test fun2()
19  {
20      Test t(2);
21      cout << "fun2()" << endl;
22      return t;
23  }
24
25  int main(int argc, char* argv[]) {
26      Test t1(1);
27      cout << "-------\n";
28      Test t2(t1);              // 通过使用另一个同类型的对象来初始化新创建的对象。
29      cout << "-------\n";
30      fun1(t1);                 // 复制对象把它作为参数传递给函数。
31      cout << "-------\n";
32      fun2();                   // 复制对象，并从函数返回这个对象，返回对象时调用拷贝构
    造函数。
33      //Test t3 = fun2();
34
35      getchar();
36      return 0;
37  }
38  /* 打印
39  Test(int x)
40  -------
41  copy constructor
42  -------
43  copy constructor
44  fun1()
45  -------
46  Test(int x)
47  fun2()
48  copy constructor
49  */
```

类的对象需要拷贝时，拷贝构造函数将会被调用。 以下情况都会调用拷贝构造函数：

- 通过使用另一个同类型的对象来初始化新创建的对象。
- 复制对象，把它作为参数传递给函数。
- 复制对象，并从函数返回这个对象。

浅复制与深复制

如果在类中没有显式地声明一个复制构造函数，那么，编译器将会自动生成一个默认的复制构造函数，该构造函数完成对象之间的浅复制。

自定义复制构造函数是一种良好的编程风格，它可以阻止编译器形成默认的复制构造函数，提高源码效率。

所谓浅复制，直接为数据成员赋值即可。比如一个实类 c1 里的数据成员里有一个指针 c1.p 指向字符数组 str[] = “hello”，使用浅复制来初始化类 c2，则赋值为 c2.p = c1.p，那么通过 c1.p 改变了 str，c2.p 指向的内容也会改变。

而深复制要创建新的对象，要为对象的数据成员分配存储空间，直接赋值就将值保存在相应的空间中。比如

```
1  // c1, c2 均是类
2  c2.p = new char[strlen(c1.p) + 1];
3  strcpy(c2.p, c1.p);
```

让 c2.p 指向 new 出来的空间，这时 c1.p 里的内容改变了也不会影响 c2.p。

初始化列表

使用初始化列表的原因

初始化类的成员有两种方式，一是使用初始化列表，二是在构造函数体内进行赋值操作。

主要是性能问题，对于内置类型，如 int, float 等，使用初始化列表和在构造函数体内初始化差别不是很大，但是对于类类型来说，最好使用初始化列表，因为使用初始化列表少了一次调用默认构造函数的过程，这对于数据密集型的类来说，是非常高效的。

必须使用初始化列表的情况

（只能使用初始化而不能赋值）【PRIMER 259】

初始化和赋值的区别事关底层效率：前者直接初始化数据成员，后者则先初始化再赋值。

除了效率问题之外，有些时候初始化列表是不可或缺的，以下几种情况时必须使用初始化列表：

- 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面；
- 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面；

- 没有默认构造函数的类类型（比如构造函数为私有），因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化。（有一点迷惑）

  如果在子类的构造函数中需要初始化父类的 private 成员。直接对其赋值是不行的，只有调用父类的构造函数才能完成对它的初始化。

【总结】当类中含有 const 常量、reference 成员变量；基类的构造函数都需要初始化列表。

```
class A
{
private:
    int a;
public:
    A(int x) : a(x) {}
};

class B : public A
{
private:
    int b;
public:
    B(int x, int y) : A(x)
    {
        // a = x;       // a 为 private，无法在子类中被访问
        // A(x);        // 在函数体内调用父类的构造函数不合法
        b = y;
    }
};
```

初始化顺序

成员是按照他们在类中出现的顺序进行初始化的，而不是按照他们在初始化列表出现的顺序初始化的。

```
class foo {
    public:
    int i;
    int j;
    foo(int x): i(x), j(i) {} // ok, 先初始化 i，后初始化 j
};
```

再看下面的代码：

```
class foo {
  public:
  int i;
  int j;
  foo(int x): j(x), i(j) {} // i 值未定义
  /*
  这里 i 的值是未定义。因为虽然 j 在初始化列表里面出现在 i 前面，
  但是 i 先于 j 定义，所以先初始化 i，而 i 由 j 初始化，
  此时 j 尚未初始化，所以导致 i 的值未定义。
  一个好的习惯是，按照成员定义的顺序进行初始化。
  */
};
```

友元函数

类的友元函数是定义在类外部，但与成员函数有相同的权限，所以可以访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

友元可以是一个函数，该函数被称为友元函数。友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。

```
class Box
{
  double width;
  public:
    friend void printWidth( Box box );  // printwidth 能够调用 Box 类的所有私有（private）成员和保护（protected）成员
    void setWidth( double wid );
};
void printWidth( Box box )  // printWidth() 不是任何类的成员函数
{
  /* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 */
  cout << "Width of box : " << box.width << endl;
}
```

内联函数

C++ 内联函数是通常与类一起使用。如果一个函数是内联的，那么在编译时，编译器会把该函数的代码副本放置在每个调用该函数的地方。在类中定义的函数都是内联函数，即使不用 inline 说明符。

引入内联函数的目的是为了解决程序中函数调用的效率问题，因为编译器使用相同的函数代码代替函数调用，对于内联代码，程序无需跳转到另一个位置执行代码，再跳回来。因此内联函数的运行速度比常规函数稍快，但代价是需要占用更多的内存。如果在程序的多个不同的地方调用内联函数，该程序将包含该内联函数的多个副本。总的来说就是用空间换时间。所以内联函数一般都是1-5行的小函数。关于内联函数可以总结为：

- 相当于把内联函数里面的内容写在调用内联函数处；
- 相当于不用执行进入函数的步骤，直接执行函数体；
- 相当于宏，却比宏多了类型检查，真正具有函数特性；
- 不能包含循环、递归、switch 等复杂操作；
- 在类声明中定义的函数，除了虚函数的其他函数都会自动隐式地当成内联函数。在类外定义需要显式内联；
  使用内联函数不过是向编译器提出一种申请，编译器可以拒绝你的申请。

内联函数与宏的区别

- 内联函数在编译时展开，宏在预编译（预处理）时展开；
- 在编译的时候，内联函数可以直接被镶嵌到目标代码里，而宏只是一个简单的文本替换；
- 内联函数可以完成诸如类型检测、语句是否正确等编译功能，宏就不具有这样的功能；
- 内联函数是函数，宏不是函数；
- 宏在定义时要小心处理宏参数（用括号括起来），否则会出现二义性，而内联函数定义时不会出现二义性

volatile

volatile 关键字是一种类型修饰符，用它声明的类型变量表示这个变量可能被意想不到的修改（比如：操作系统、硬件或者其它线程等）。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。

explicit

普通函数是能够被隐式调用（如使用=），而 explicit 构造函数只能被显式调用。

explicit 是用来防止隐式转换的，它只对一个实参的构造函数有效，且只允许出现在类内的构造函数声明处。

```
1  class Test1
2  {
3  public:
4     int num;
```

```
    char *str;
    Test1(int n, char *s)
    {
        num = n, str = s;
        cout << "call Test1(int n, char *s)\n";
    }
};

class Test2
{
public:
    int num;
    char *str;
    string type_info;

    explicit Test2(int n, char *s) : num(n), str(s)
    { cout << "call explicit Test2(int n, char *s)\n"; }

    explicit Test2(string) : type_info("call explicit Test2(string)\n")
    { cout << type_info; }

    explicit Test2(char) : type_info("call explicit Test2(char)\n")
    { cout << type_info; }

    Test2(int) : type_info("call explicit Test2(int)\n") { cout << type_info; }

    explicit Test2(short) : type_info("call explicit Test2(short)\n")
    { cout << type_info; }
};

int main(int argc, char* argv[]) {
    Test1 t1 = { 12, "hello..." };       // 隐式调用成功
    //Test2 t2 = { 12, "hello..." };     // 编译错误，不能隐式调用其构造函数
    Test2 t3(12, "hello...");           // 显式调用成功
    short n = 42;
    Test2 t4 = n;                       // 虽然 n 是 short 类型，但因为 explicit Test2(short) 不能隐式调用，故退而求其次把 n 转换成 int 调用 Test2(int)
    system("pause");
}
/* 打印
call Test1(int n, char *s)
call explicit Test2(int n, char *s)
call explicit Test2(int)
```

```
47  */
```

const

```
1  const Stock &Stock::topval(const Stock & s) const
2  {.....}
3  //有两只股票，返回价格高的那一只股票【PLUS 363，365】
```

该函数显式的访问一个对象（参数），又隐式的访问另一个对象（调用的对象），并返回其中一个对象的引用。参数中的 const 表明，该函数不会修改被显式访问的对象（不会修改参数指针指向的内容），而括号后的 const 表明，该函数不会修改被隐式地访问的对象（该类方法 Stock::topval() 不会修改类里的数据），最前面的 const 表明函数的返回值不能被修改。

函数返回引用

返回引用能节省调用拷贝（复制）构造函数生成的副本所需的时间和析构函数删除副本所需的时间。但并不总是可以返回引用，函数不能返回在函数中创建的临时对象的引用，因为当函数结束，临时对象就消失了。
【PLUS 526】

const作用

- 修饰变量，说明该变量不可以被改变；
- 修饰指针，分为指向常量的指针和指针常量；
- 修饰函数引用参数，即避免了拷贝，又避免了函数对引用值的修改；

  如 `void fun(A const &a);` A是用户自定义类型。相比于值传递减少了临时对象的构造、复制、析构过程，用 const 修饰引用，避免函数通过引用修改 a。
- 修饰函数返回值，说明该返回值不能被修改，且该返回值只能赋值给加 const 修饰的同类型变量
- 修饰类的成员函数，说明在该成员函数内不能修改成员变量。

const使用

```
1  // 类
2  class A
3  {
4  private:
5    const int a;          // 常对象成员，只能在初始化列表赋值
6
7  public:
8    // 构造函数
9    A() : a(0) { };
10   A(int x) : a(x) { };     // 初始化列表
```

```
11
12      // 后置const可用于对重载函数的区分
13      int getValue();         // 普通成员函数
14      int getValue() const;     // 常成员函数，不得修改类中的任何数据成员的值
15  };
16
17  void function()
18  {
19      // 对象
20      A b;                // 普通对象，可以调用全部成员函数
21      const A a;              // 常对象，只能调用常成员函数、更新常成员变量
22      const A *p = &a;        // 常指针
23      const A &q = a;         // 常引用
24
25      // 指针
26      char greeting[] = "Hello";
27      char* p1 = greeting;            // 指针变量，指向字符数组变量
28      const char* p2 = greeting;      // 指针变量，指向字符数组常量
29      char* const p3 = greeting;      // 常指针，指向字符数组变量
30      const char* const p4 = greeting;   // 常指针，指向字符数组常量
31  }
32
33  // 函数
34  void function1(const int Var);        // 传递过来的参数在函数内不可变
35  void function2(const char* Var);      // 参数指针所指内容为常量
36  void function3(char* const Var);      // 参数指针为常指针
37  void function4(const int& Var);       // 引用参数在函数内为常量
38
39  // 函数返回值
40  const int function5();    // 返回一个常数
41  const int* function6();   // 返回一个指向常量的指针变量，使用：const int *p =
    function6();
42  int* const function7();   // 返回一个指向变量的常指针，使用：int* const p =
    function7();
```

const修饰指针

指针本身是一个独立的对象，它又可以指向另一个对象。所以指针和 const 同时使用时，有两种情况：

```
1  int i = 0;
2  int *const j = &i;
3  // 指针常量,指向不可变地址的指针，但可以对它指向的内容进行修改。
4  // 指针j指向i，const修饰指针j本身，
```

```
// 所以不允许修改j，但可以通过j修改i的值
const int *k = &i;
// 常量指针，指向常量的指针，该指针指向的地址里的内容不可变。
// 指针k指向i，const修饰k指向的i，
// 所以可以修改k，但不可以通过k修改i的值
int const *p = &i;
// 即 const int *p，同上，为常量指针。
// const 修饰离右边最近的那一个，int const *p 等价于 const int *p
// 都可以理解为 const 修饰（*p）而不是 p，那么 p 可变,p 指向的值不可变
const int * const p = &i;
// p 只能指向 i，且 p 指向的 i 也不可变
```

const 与 #define

```
#define PI 3.1415926
const float pi = 3.1415926;
```

- const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误 。
- 宏定义是直接替换，它的生命周期止于编译期，不会分配内存，存储于程序的代码段中；
  const 常量存在于程序的数据段，并分配了实际的内存。

存储类

auto

C++ 11 以来，auto 关键字用于两种情况：声明变量时根据初始化表达式自动推断该变量的类型、声明函数时函数返回值的占位符。

auto是类型推导，让使用者获得动态语言的使用体验；但是有区别，那就是 auto 声明的变量类型，你可以不知道，但是编译器一定要知道，这样才不会报错。

根据初始化表达式自动推断被声明的变量的类型，如：

```
auto a;                 // 报错！a未初始化，编译器推算不出来 a 是什么类型，就不知道要开辟多大的空间。
auto f = 3.14;          // double
auto s("hello");        // const char*
auto z = new auto(9);   // int*
auto x1 = 5, x2 = 5.0, x3 = 'r';  // 报错！
/*
当在同一行中定义多个变量时，编译器只对第一个类型进行推导，
然后用推导出来的类型定义其它变量
*/
```

注意：

- auto 不可作为函数的参数：参数要被编译成指令，auto 做参数，会不知道要开辟多大的空间。
- auto 不可直接用来声明数组：因为不知道要开辟多大空间。

auto 在实际中最常见的优势用法是 C++11 提供的新式 for 循环，还有 lambda 表达式等进行配合使用。

static作用

- 修饰变量：修改变量的存储区域和生命周期，使变量存储在静态区，变量只初始化一次。
  - 当修饰全局变量的时候，就是静态全局变量。静态全局变量与非静态全局变量都是存储在静态区，但是它们的作用域不一样：静态全局变量的作用域只在该文件里有效（文件作用域），而非静态全局变量的作用域是在整个源程序里有效。
  - 当修饰局部变量的时候，一般用在函数体中，静态局部变量的作用域与普通局部变量一样，只在该函数中有效，但静态局部变量存储在静态区，在函数调用结束会维持其值不变，下次调用该函数不会初始化而是直接使用上一次调用时的值。
- 修饰普通函数：表明函数的作用范围，仅在定义该函数的文件内才能使用。在多人开发项目时，为了防止与他人命名的函数重名，可以将函数定义为static。
- 修饰成员变量：静态成员变量用来表示唯一的、可共享的成员变量。它可以在同一个类的所有对象中被访问。静态成员变量只有唯一的一份实体。不需要生成对象就可以访问该成员。

  static 成员变量必须 在类声明的外部进行初始化，以示与普通数据成员的区别。

  例如：`int Class_name::static_val = 5;`

static 成员变量和普通 static 变量一样，都在内存分区的全局数据区分配内存，到程序结束后释放。这就意味着，static 成员变量不随对象的创建而分配内存，也不随对象的销毁而释放内存。而普通成员变量在对象创建时分配内存，在对象销毁时释放内存。

- 修饰成员函数：静态成员函数使得不需要生成对象就可以访问该函数。静态成员函数和静态成员变量一样，不属于类，所以静态成员函数不含 **this** 指针，也就无法访问类的非静态成员。

【对1，2条的总结】

static 最重要的一条在于修饰普通变量与普通函数时，隐藏普通变量与普通函数。因为未加 static 前缀的全局变量和函数都具有全局可见性。

静态数据成员与全局变量相比的优势

- 静态数据成员没有进入程序的全局命名空间，因此不存在与其他全局变量名字冲突的可能；
- 使用静态数据成员可以隐藏信息。因为静态数据成员可以是 private 成员，而全局变量不能；

extern

extern 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都可见。使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置(定义只有一次，不可重复定义)。

多个文件定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 extern 来得到已定义的变量或函数的引用。可以这么理解，extern 是用来在另一个文件中声明一个全局变量或函数。

类型转换运算符

C++ 中四种类型转换是：`static_cast, dynamic_cast, const_cast, reinterpret_cast`

```
cast-name<type>(expression)
// type是转换的目标类型，expression 是被转换的值。
```

- const_cast(常量转换)

  用于将 const 变量转为非 const ，也可以去除 volatile，除此之外不允许任何类型转换。即常量指针被转换成非常量指针，并且仍然指向原来的对象；常量引用被转换成非常量引用，并且仍然引用原来的对象。

- static_cast(静态转换)

  任何编写程序时能够明确的类型转换都可以使用 static_cast（static_cast 不能转换掉底层 const，volatile 和 __unaligned 属性）。由于不提供运行时的检查，所以叫 static_cast，因此，需要在编写程序时确认转换的安全性。

主要在以下几种场合中使用：

  - 用于类层次结构中，父类和子类之间指针和引用的转换；进行上行转换，把子类对象的指针/引用转换为父类指针/引用，这种转换是安全的；进行下行转换，把父类对象的指针/引用转换成子类指针/引用，这种转换是不安全的，需要编写程序时来确认；
  - 用于基本数据类型之间的转换，例如把 int 转 char，int 转 enum 等，需要编写程序时来确认安全性；
  - 把 void 指针转换成目标类型的指针（这是极其不安全的）；

- dynamic_cast(动态转换)

  用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化，只能转指针或引用。向下转化时，如果是非法的对于指针返回 nullptr，对于引用抛异常。要深入了解内部转换的原理。

  向上转换：指的是子类向基类的转换。此时与 static_cast 和隐式转换一样，都是非常安全的。 注意菱形继承中的向上转换要指明路径。

  向下转换：指的是基类向子类的转换；

  它通过变量运行时的类型和要转换的类型是否相同，来判断是否能够进行向下转换。

  为什么只能用于含有虚函数的类?

  因为类中存在虚函数，说明它可能有子类，这样才有类型转换的情况发生，由于运行时类型检查需要运行时类型信息，而这个信息存储在类的虚函数表中，只有定义了虚函数的类才有虚函数表。

- reinterpret_cast(重解释)

  几乎什么都可以转，比如将 int 转指针，可能会出问题，尽量少用；

为什么不使用 C 的强制转换

C 的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

this指针

每个非静态成员函数（包含构造函数和析构函数）都有一个 `this` 指针， `this` 指针指向调用对象， `this` 是地址，如果要引用调用对象本身，用 `*this` 。

（只有成员函数才有 this 指针，静态成员函数和友元函数都不含 this 指针）

当一个成员函数被调用时，自动向它传递一个隐含的参数，该参数是一个指向这个成员函数所在的对象的指针。this 指针被隐含地声明为: `ClassName *const this` ，这意味着不能给 `this` 指针赋值；在 `ClassName` 类的 `const` 成员函数中， `this` 指针的类型为： `const ClassName* const` ，这说明 `this` 指针所指向的这种对象是不可修改的（即不能对这种对象的数据成员进行赋值操作）。

一个类的成员函数只有一份，并不是每一个对象对应一个单独的成员函数体，而成员函数之所以能把属于此类的各个对象的数据区分开，就在于每次执行类成员函数时，都会把当前的 `this` 指针（对象首地址）传入成员函数，函数体内所有对类数据成员的访问都会转化为 `this->数据成员` 的方式。

【PLUS 364】

【注意】静态成员函数里，不能使用 this 指针：

静态成员函数并不是针对某个类的实例对象，而是属于整个类的，为所有的对象实例所共有。他在作用域的范围内是全局的，独立于类的对象之外的。他只对类内部的静态成员变量做操作。当实例化一个类的对象时候，里面不存在静态成员的。this 指针是相当于一个类的实例的指针，this 是用来操作对象实例的内容的，既然静态成员函数和变量都是独立于类的实例对象之外的，它就不能用 this 指针，也不能操作非静态成员。

虚函数

```
1  /*BrassPlus 是 Brass 的子类， ViewAcct() 是两个类中都有的方法。
2  由于 bp是父类指针，如果基类不用虚方法那么就会调用基类的 ViewAcct() 方法
3  若如果在基类中将 ViewAcct() 声明为虚，则 bp->ViewAcct() 根据对象类型
     （BrassPlus）调用 BrassPlue::ViewAcct()方法*/
4  BrassPlus ophelia;
5  Brass *bp;
6  bp = &ophelia;
7  bp->ViewAcct(); // 是调用子类还是父类的 ViewAcct() 方法?
```

【PLUS 503】

虚函数是怎么实现的

虚函数是通过虚函数表实现的。如果一个类中有一个虚函数，则系统会为这个类分配一个指针成员指向一张虚函数表（vtbl），表中每一项指向一个虚函数地址，虚函数表实际上就是一个函数指针数组。

C++虚函数原理

虚函数是否可以内联

- 虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。
- 内联是在编译期间编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。
- `inline virtual` 唯一可以内联的时候是：编译器知道所调用的对象是哪个类，这只有在编译器具有实际对象而不是对象的指针或引用时才会发生。

虚析构函数

```
// Employee 是基类，Singer 是派生类
Employee *pe = new Singer;
...
delete pe; // call ~Employee() or ~Singer()?
```

如果基类中的析构函数不是虚的，就只调用对应于指针类型(Employee)的析构函数，但实际中，是想调用派生类的析构函数。如果基类的析构函数是虚的，将调用相应对象类型(Singer)的析构函数，然后自动调用基类的析构函数。因此，使用虚析构函数可以保证正确的析构函数序列被调用。

【PLUS 501，505】

```
// Employee 是基类，Singer 是派生类
Singer *pe = new Singer;
...
delete pe; // 先调用 ~Singer() 再调用 ~Employee()，
        // 因为 pe 是 Singer* 类型指针，而不是 Employee* 类型指针。
```

静态联编和动态联编

将源程序中的函数调用解释为执行特定的函数代码块被称为函数名联编。在 C 语言中由于不支持函数重载，这很容易，在 C++ 中，由于支持函数重载，编译器必须查看函数的参数才知道调用的是哪一个函数。C++ 在编译过程中就可以完成这种联编，故称为静态联编，又称为早期联编。编译器总是对非虚方法使用静态联编。

然而虚函数使编译器不知道在编译时到底用哪一个函数，因为编译器不知道用户将选择哪种类型的对象，所以编译器必须生成能够在程序运行时选择正确的虚函数的代码，这被称之为动态联编，也被称为晚期联编。

【PLUS 501】

有关虚函数的注意事项

- 构造函数不能是虚函数。先构造父类对象，然后才能是子类对象，如果构造函数设为虚函数，那么当你在构造父类的构造函数时就不得不显式的调用构造，还有一个原因就是为了防错，试想如果你在子类中一不小心重写了个跟父类构造函数一样的函数，那么你的父类的构造函数将被覆盖，即不能完成父类的构造，就会出错。
- 析构函数应当是虚函数，除非类不用做基类。即使类不用作基类，通常应给基类提供一个虚析构函数
- 友元不能是虚函数，因为友元不是类成员，只有类成员才能是虚函数
- 如果派生类没有重新定义函数，将使用该函数的基类版本。如果派生类位于派生链中，则将使用最新的虚函数版本
- 如果派生类重新定义函数，将隐藏同名基类方法，这不同于重载

【PLUS 503，504】

纯虚函数

纯虚函数是一种特殊的虚函数，纯虚函数只是一个接口，是让派生类实现细节的，在纯虚函数中也可以定义具体实现，但没意义。包含纯虚函数的类是抽象基类（ABC，abstract base class），它只能作为基类，不能创建对象。可以从抽象基类派生出具体类（普通类），这些类可以创建对象。

```
class C
public:
    virtual int f1() = 0; //函数原型中的 =0 使虚函数成为纯虚函数
    virtual double area() const = 0;
    virtual ~C() {}
```

【PLUS 509，510】

虚函数、纯虚函数

- 类里如果声明了虚函数，这个函数是实现的，哪怕是空实现，它的作用就是为了能让这个函数在它的子类里面可以被覆盖，这样的话，这样编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。
- 虚函数在子类里面也可以不重载的；但纯虚函数必须在子类去实现。
- 虚函数的类用于“实作继承”，继承接口的同时也继承了父类的实现。当然大家也可以完成自己的实现。纯虚函数关注的是接口的统一性，实现由子类完成。
- 带纯虚函数的类叫抽象类（虚基类），这种类不能直接生成对象，而只有被继承，并重写其虚函数后，才能使用。抽象类和大家口头常说的虚基类还是有区别的，在 C# 中用 abstract 定义抽象类，而在 C++ 中有抽象类的概念，但是没有这个关键字。抽象类被继承后，子类可以继续是抽象类，也可以是普通类，而虚基类，是含有纯虚函数的类，它如果被继承，那么子类就必须实现虚基类里面的所有纯虚函数，其子类不能是抽象类。

多重继承

```
class Person
{
    string s = "Person";
public:
    void sleep() { cout << s + " sleep" << endl; }
    void eat() { cout << s + " eat" << endl; }
};

class Author : public Person      // 标记【1】Author puplic 继承自 Person
{
```

```
11    string s = "Author";
12  public:
13    void writeBook() { cout << s + " wirte book" << endl; }
14  };
15
16  class Programmer : public Person    // 标记【2】
17  {
18    string s = "Programmer";
19  public:
20    void writeCode() { cout << s + " write Code" << endl; }
21  };
22
23  class Programmer_Author : public Programmer, public Author  // 多重继承
24  {
25    string s = "Programmer_Author";
26  };
27
28
29  int main(int argc, char* argv[]) {
30    Programmer_Author pa;
31    pa.writeBook();
32    pa.writeCode();
33    //pa.eat();              // 标记【3】编译错误
34    pa.Person::eat();
35    pa.Programmer::sleep();
36    system("pause");
37    return 0;
38  }
39  /* 打印
40  Author wirte book
41  Programmer write Code
42  Person eat
43  Person sleep
44  */
```

标记【3】处编译错误，是因为通过多重继承 Programmer_Author 类拥有 Programmer 类和 Author 类的一份拷贝，而 Programmer 类和 Author 类都分别拥有 Person 类的一份拷贝，所以 Programmer_Author 类拥有 Person 的两份拷贝，在调用 Person 类的两份接口时，编译器不清楚需要调用哪一份拷贝，从而产生错误。

对于这个问题通常有两个解决方案：

- 加上范围解析运算符（域区分符）确定调用哪一份拷贝。比如 `pa.Programmer::sleep();`
- 使用虚拟继承，使得多重继承类 Programmer_Author 只有 Person 类的一份拷贝。

```
// 在上面代码标记【1】标记【2】处加入 virtual 即可，这样 pa.eat(); 就不会产生错误了。
class Author : virtual public Person      // 标记【1】
class Programmer : virtual public Person   // 标记【2】
```

【总结】多重继承的优点是对象可以调用多个基类中的接口，但是容易出现继承向上的二义性。

类型安全

类型安全是指同一段内存在不同的地方，会被强制要求使用相同的办法来解释(内存中的数据是用类型来解释的)。

Java 语言是类型安全的，除非强制类型转换。

C 语言不是类型安全的，因为同一段内存可以用不同的数据类型来解释，比如 1 用 int 来解释就是 1，用 boolean来解释就是 true。

C++ 也不是绝对类型安全的，但使用得当，它将远比 C 更有类型安全性。

C++提供了一些新的机制保障类型安全：

- 操作符 new 返回的指针类型严格与对象匹配，而不是 void
- C 中很多以 void* 为参数的函数可以改写为 C++ 模板函数，而模板是支持类型检查的；
- 引入 const 关键字代替 #define constants，它是有类型、有作用域的，而 #define constants 只是简单的文本替换；
- 一些 #define 宏可被改写为 inline 函数，结合函数的重载，可在类型安全的前提下支持多种类型，当然改写为模板也能保证类型安全；
- C++ 提供了 dynamic_cast 关键字，使得转换过程更加安全，因为 dynamic_cast 比 static_cast 涉及更多具体的类型检查。

STL

STL包括两部分内容：容器和算法。重要的还有融合这二者的迭代器。

迭代器是**STL**的精髓，它提供了一种方法，使它能够按照顺序访问某个容器所含的各个元素，但无需暴露该容器的内部结构。它将容器和算法分开，好让这二者独立设计。

容器，即存放数据的地方。比如array等。

在STL中，容器分为两类：序列式容器和关联式容器。

序列式容器，其中的元素不一定有序，但都可以被排序。如：vector、list、deque、stack、queue、heap、priority_queue。

关联式容器，内部结构基本上是一颗平衡二叉树。所谓关联，元素按照一定的规则存放（元素位置取决于特定的排序准则）。如：RB-tree、set、map、multiset、multimap、hashtable、hash_set、hash_map、hash_multiset、hash_multimap。

下面各选取一个作为说明。

vector：它是一个动态分配存储空间的容器。区别于c++中的array，array分配的空间是静态的，分配之后不能被改变，而vector会自动重分配（扩展）空间。

使用 **vector** 时需要注意：每次插入和移除元素的时候，都会使作用点之后的各元素 **references、pointers、iterators** 失效，如果插入操作引发内存重新分配，那么该容器上所有的 **references、pointers、iterators** 都会失效。

set：其内部元素会根据元素的键值自动被排序。区别于map，它的键值就是实值，而map可以同时拥有不同的键值和实值。

算法，如排序，复制……以及一些容器特定的算法。这点不用过多介绍，主要看下迭代器的内容。

vector扩容原理

空间分配的多，平摊时间复杂度低，但浪费空间也多。

使用 k = 2 增长因子的问题在于，每次扩展的新尺寸必然刚好大于之前分配的总和。也就是说，之前分配的内存空间不可能被使用。这样对于缓存并不友好。最好把增长因子设为 $1 < k < 2$，例如 k = 1.5这样，在几次扩容后，就可以重用之前的内存空间。

其实 C++ 标准也没有规定要用哪一个增长因子， VS2015中以1.5倍扩容，GCC以2倍扩容。

总结

1. vector在push_back以成倍增长可以在均摊后达到O(1)的事件复杂度，相对于增长指定大小的O(n)时间复杂度更好。
2. 为了防止申请内存的浪费，现在使用较多的有2倍与1.5倍的增长方式，而1.5倍的增长方式可以更好的实现对内存的重复利用，因为更好。

C++ STL中vector内存用尽后，为啥每次是两倍的增长，而不是3倍或其他数值?

map 与 hash_map 区别

- 底层数据结构不同：map 是红黑树（查找时间复杂度log(n)），hash_map 是哈希表（查找时间复杂度O(1)）
- map 的优点在于元素可以自动按照键值排序，而 hash map 的优点在于它的各项操作的平均时间复杂度接近常数
- map 属于标准的一部分，而 hash_map 则不是

什么时候用map，什么时候用hash_map?

这个要看具体的应用，不一定常数级别的 hash_map 一定比 log(n) 级别的 map 要好， hash_map 的 hash 函数以及解决地址冲突等都要耗时间，而且众所周知 hash 表是以空间换时间的，因而 hash_map 的内存消耗肯定要大，一般情况下，如果记录非常大，考虑 hash_map，查找效率会高很多，如果要考虑内存消耗，则要谨慎使用hash_map。

【扩展】

map, set, multimap, and multiset 采用红黑树实现，红黑树是平衡二叉树的一种。不同操作的时间复杂度近似为:

插入: O(logN)

查看:O(logN)

删除:O(logN)

hash_map, hash_set, hash_multimap, and hash_multiset 采用哈希表实现，不同操作的时间复杂度为：

插入:O(1)，最坏情况O(N)。

查看:O(1)，最坏情况O(N)。

删除:O(1)，最坏情况O(N)。

红黑树

红黑树是每个节点都带有颜色属性的二叉查找树，颜色或红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:

性质1. 节点是红色或黑色。

性质2. 根节点是黑色。

性质3 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

性质4. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

这些约束强制了红黑树的关键性质：从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

要知道为什么这些特性确保了这个结果，注意到性质3导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质4所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

map 底层为什么用红黑树而不是平衡二叉树（AVL）

AVL 是有下列性质的二叉树：

它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过 1。

将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF，那么平衡二叉树上的所有结点的平衡因子只可能是 -1、0 和 1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

红黑树是在AVL树的基础上提出来的。

红黑树较AVL树的优点：

AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的 rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

所以红黑树在查找，插入删除的性能都是 O(logn)，且性能稳定，所以 STL 里面很多结构包括 map 底层实现都是使用的红黑树。

STL 关联容器底层实现总结

有序关联容器 底层实现为红黑树，增删改查时间复杂度O(n)。

set，multiset，map，multimap。

无序关联容器 底层实现链式哈希表 ，增删改查时间复杂度O(1)。

unordered_set，unordered_multiset，unordered_map，unordered_multimap。

map 和 unordered_map 的性能对比

map内部是红黑树，在插入元素时会自动排序，而无序容器 unordered_map 内部是哈希表，通过哈希而不是排序来快速操作元素，使得效率更高。当你不需要排序时选择 unordered_map 的效率更高。

STL中的 remove 和 erase 区别

- erase 一般作为一个 container 的成员函数，是真正删除的元素，是物理上的删除（迭代器访问不到了）
- algorithm 中的 remove 只是简单的把要 remove 的元素移到了容器最后面，然后其余元素前移，是逻辑上的删除，此时容器的 size 不变化。因为 algorithm 通过迭代器操作，不知道容器的内部结构，所以无法做到真正删除。

智能指针

智能指针主要用于管理在堆上分配的内存，它将普通的指针封装为一个栈对象。当栈对象的生存周期结束后，会在析构函数中释放掉申请的内存，从而防止内存泄漏。C++ 11中最常用的智能指针类型为shared_ptr，它采用引用计数的方法，记录当前内存资源被多少个智能指针引用。该引用计数的内存在堆上分配。当新增一个时引用计数加 1 ，当过期时引用计数减一。只有引用计数为 0 时，智能指针才会自动释放引用的内存资源。对shared_ptr进行初始化时不能将一个普通指针直接赋值给智能指针，因为一个是指针，一个是类，可以通过构造函数传入普通指针。

`std::auto_ptr<string> ptr(new string);`

```
auto_ptr<char*> ap(new char*);
*ap = "ap1";
*ap = "ap2";
char **bp = new char*;
*bp = "bp1";
cout << *ap << endl;   // "ap2"
cout << *bp << endl;   // "bp1"
```

```
#include <memory>
double *p = new double;
shared_ptr<double> pshared(p);    // 合法，显示转换，explicit conversion
//shared_ptr<double> pshared = p; // 不合法，隐式转换，implicit conversion
```

```
int main(int argc, char* argv[]) {
    string str("hello world!");
    // 程序能运行，但是在要释放 pshared 指向的内存时会出错
    // 因为 str 不是存在堆中，当 pshared 过期时，delete 运算符会用于非堆内存，造成错误
    shared_ptr<string> pshared(&str);
    cout << *pshared << endl;
    getchar();
}
```

- auto_ptr 是C++98提供的解决方案，C++11已经摒弃，并提供了以下几种方案

- shared_ptr 被称为共享指针，用于管理多个智能指针共同拥有的动态分配对象，
- unique_ptr 唯一拥有指定的对象，相比普通指针，拥有 RAII 的特性使得程序出现异常时，动态资源可以得到释放。

  RAII，Resource Acquisition Is Initialization，资源获取即初始化： 其核心是把资源和对象的生命周期绑定，对象创建获取资源，对象销毁释放资源
- weak_ptr 是为了配合shared_ptr而引入的一种智能指针，因为它不具有普通指针的行为，没有重载operator*和->,它的最大作用在于协助shared_ptr工作，像旁观者那样观测资源的使用情况。

智能指针的内存泄露以及解决方法

当两个对象相互使用一个 shared_ptr 成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄漏。

为了解决循环引用导致的内存泄漏，引入了 weak_ptr 弱指针，weak_ptr 的构造函数不会修改引用计数的值，从而不会对对象的内存进行管理，其类似一个普通指针，但不指向引用计数的共享内存，但是其可以检测到所管理的对象是否已经被释放，从而避免非法访问。

为什么摒弃 auto_ptr

```
1  auto_ptr<string> p1(new string("hello"));
2  auto_ptr<string> p2;
3  p2 = p1;
4  // 当 p1, p2 过期时，将删除同一个对象两次
```

解决之道：

- 定义复制运算符，使之执行深复制
- 建立所有权概念，使同时只有一个智能指针可拥有它。这样，只有拥有对象的智能指针有权析构该对象，这是auto_ptr 的策略，unique_ptr 的策略更严格。
- 创建智能更高的指针，跟踪引用特定对象的智能指针数。这称为引用计数。

```
int main(int argc, char* argv[]) {
    auto_ptr<string> p1(new string("hello"));
    auto_ptr<string> p2;
    cout << *p1 << endl; // 正常打印
    p2 = p1;          // p1 丧失了对 string 对象的所有权，p1 此时是空指针
    cout << *p1 << endl; // 编译通过，但运行时报错，因为试图提领空指针
    getchar();
}
// 将 auto_ptr 换成 unique_ptr，编译器认为 p2 = p1; 非法，在编译阶段报错（因为 p1 不是临时右值）。
// 将 auto_ptr 换成 shared_ptr，编译运行阶段都没问题，正常打印。
// shared_ptr 采用的策略是引用计数，赋值时，计数加一，过期时，计数减一。仅当最后一个指针过期时，才调用 delete。
```

```
shared_ptr<string> p1(new string("hello"));
shared_ptr<string> p2(p1);    // 合法，将右值 p1 赋给 p2
```

```
unique_ptr<string> p1(new string("hello"));
//shared_ptr<string> p2(p1);     // 不合法，右值 p1 是 unique_ptr，若能赋给 p2，则 p1，p2 指向同一个对象，导致 p1 不合法，此语句编译不通过
//unique_ptr<string> p2(p1);     // 不合法，p1 不是临时右值，注意临时。
```

```
unique_ptr<string> foo() {
    unique_ptr<string> p1(new string("hello"));
    return p1;
}
// 函数返回的 unique_ptr<string> 为临时右值，此时可赋给另一个 unique_ptr 型指针
int main(int argc, char* argv[]) {
    unique_ptr<string> p2(foo());
}
```

左值，右值

基于范围的 for 循环

```cpp
int main(int argc, char* argv[]) {
    int arr[10] = {1, 2, 3, 4, 5, 6};
    for (int &x : arr)    // 只有使用引用 & 才能通过 x 修改 arr 里的值
    {
        cout << (x % 2 ? "奇" : "偶") << ' ';
        x = x + 1;
    }
    cout << endl;
    for (int x : arr) cout << x << ' ';
    getchar();
}
// 奇 偶 奇 偶 奇 偶 偶 偶 偶 偶
// 2 3 4 5 6 7 1 1 1 1
```