C++高阶编程技术学习报告

第一章 C++ Primer

1.2 C++ 基础

1.2.1 内置类型

1. **类型转换**

```
w_char      // 宽字符
char16_t   // Unicode 字符
char32_t   // Unicode 字符
```

计算机存储的是二进制补码，正数补码不变，-正数的补码：正数的反码加 1

切勿混用 signed 和 unsigned 类型，signed 和 unsigned 运算时，signed 会转换成 unsigned来算2.

```
unsigned int a=10;
unsigned int b=42;
cout<<a-b;    // 出错，结果还是正值
// 所以，for 循环中 int i 是 singned int
```

2. **字面值常量**

```
L'a'       // char32_t 'a'
u8"hi"    // UTF-8( 仅用于字符串字面常量 )
10L          // long int 10
42ULL    // unsigned long long
3.14L    // long double
1E-3F    // float 1*10^-3
int *ptr    // nullptr 是指针的字面值
```

1.2.2 变量

初始化：创建变量时获得了一个特定的值

```
std::string book("holly poter");   // 使用构造函数初始化
```

```
int i = 0;
int i = {0};
int i{0};
int i(0);   // 全部都是初始化

double pi = 3.14;
int i = {pi};   //  == i{pi}     // 报错
int i = pi;   //  == i(pi)        // OK 强制类型转换
```

标准库中，会出现 __两个下划线开头，一个下划线加大写，来命名

```
__Register
_Handler
// 常规使用
index;
class Sales_item;
studentLoan;
studentLoan;
```

```
int i =1;
int main()
{
    int i =0;
    cout<<i;    // 0
    cout<<::i;   // 1
}
```

1.2.3 复合类型

C++11 新增了右值引用，但是 左值引用是最常用的。

定义引用时必须要初始化，且引用不能重新绑定变量

```
double dval =3.14;
int &refVal = dval;   // Error
int *ptr = &dval;    // Error
double *ptrD = &dval;      //OK
ptr = ptrD;       // Error
```

引用不是对象，而指针是对象，因此会有指针的指针，指针可以赋值和拷贝，没有指向引用的指针

空指针，以下三种方式定义

```
int *ptr = nullptr;
int *ptr = 0;
int *ptr = NULL;   // #include <cstdlib>  因为 NULL 是预处理，在C中常用
```

对于空指针 if(ptr) 为 false

void * 可以存放任意对象的地址，最常用于函数的返回值（也说明该函数可以操作多种数据类型）

```
int *&ptr;   // 是一个对指针的引用
```

1.2.4 const 限定符

1. const修饰的变量只在该文件内可见，如果需要在多个文件中可见

```
extern const int month =12;   // 1.cpp
extern const int month;         // 2.cpp
```

---

2. 常量的引用： const + 引用

```
const int ci =1024;
const int &ri = ci;   // OK, ri 是 ci的引用，不允许通过 ri 来修改 ci 的值
ri =42;   // ERROR
int &r2 = ci;    // ERROR
```

常量的引用，之前常规的引用的类型需要和 被引用的对象一致，但常量的引用不一样

常量的引用，初始化时允许用任意表达式作为初始值，只要该表达式的结果能够转发为引用的类型即可。

```
int i =42;
int &r0 = i;              // 常规的引用
const int &r1 = i;     // 常规的引用，i的值可以变，但是不能通过 r1来改变
const int &r2 = 42;       // 常量的引用 OK
const int &r3 = r1*2;    // 常量的引用 OK
int &r4 = r1*2;     // ERROR
```

常量的引用

```
double dval = 3.14;
const int &ri = &dval;   // OK, ri 是  3 引用

// 编译器内部会生成临时的变量 const int temp = dval; const int &ri = temp;
```

---

3. 常量指针：指向常量的指针，不运行通过指针来修改指向对象的值

```
int i =10;
const int *pt = *i; // 常量的指针，不允许通过 pt 来改变 i
```

指针常量：指针指向的位置不能改变

```
int *const pt = *i;
```

---

4. 顶层 const 与 底层 const

顶层const表示指针本身是个常量（不能改变指向的位置）

```
int *cont ptr;
int const ci;
```

底层const表示指针指向的对象是一个常量

```
const int *ptr;
cont int &r =ci;
```

顶层const也可以用于所有的类型变量，表示任意类型的对象是常量

而底层const与指针和引用等复合类型有关

5.constexpr 和常量表达式

常量表达式：是编译时就具有具体的值

带const不一定是常量表达式

```
int t =1;   // 是变量，不是常量表达式
const int i =20;   // 是常量表达式
const int j=i+1;   // 是常量表达式
const int sz=getsize();   //  不是常量表达式，只有在执行时候才有值
```

第二种方法定义常量表达式

```
constexpr int mf = 20;   // 是常量表达式，但是必须放在函数体外
constexpr int j=i+1;      // 是常量表达式
constexpr int sz =getsize();   // 因为不是常量表达式，会报错
```

如果需要定义常量表达式，使用 constexpr

constexpr 与指针

```
const int *p = nullptr;   // 指向常量的指针
constexpr int *p2 = nullptr;   // 指针常量，指向的地址不能变

//
int j =0;
constexpr int *p3 = &j;   // OK  但要求很严格，要求 int j =10 定义在所有函数
之外
//

//
constexpr const int *p4 = &i;   //  == const int * const p4
```

1.2.5 类型别名

两种方式

- 传统的 typedef

```
typedef unsigned long pid_t;

//
typedef doube wages;
typedef wages base, *p;   // double == wages == base, p == double *
```

- 别名声明 using

```
using SI = Sales_item;   // SI 是 Sales_item 的同义词
```

指针、常量和类型别名
指针这样的复合类型用到类型别名里，就会出现意想不到的结果
不能直接简单的替换

```
char *pstring;
const pstring cstr = 0;   //  ??  == const char * cstr  ?? 错误
//  == char * const cstr = 0;
```

---

auto 类型说明符

使用 auto 必须要给初始值

```
auto i=0, *ptr = &i;   // OK , auto == int
auto sz = 0, pi = 3.14;    // ERROR
```

---

decltype 指示符

decltype 分析表达式并得到他的类型，但是不会计算表达式的值

```
decltype( func() ) sum = x; // sum 的类型就是函数 func() 的返回值类型
// 实际上不会调用运行 func()
```

```
const int ci =0, &cj=ci;
decltype(ci) x =0;    // const int
decltype(cj) y = x;   //const int &
decltype(cj) z;         // ERROR ,为初始化
```

decltype 和引用

引用是作为其所指对象的同义词，但是在decltype 处是例外

```
int *ptr;
decltype (*ptr) x;     / x 是 int & 类型
decltype (ptr) y;       // y是 int *
```

decltype((variable)) z; // z 永远是引用类型

自定义数据类型：结构体、类

```
struct Sales_item
{
    std::string name[20];
      int sales;
      double price;
};
```

1.3 字符串、向量和数组

1.3.1 命名空间的 using 声明

.h 头文件中不要使用 using 声明

1.4 表达式

1.4.1 基础

- 一元运算符：作用于一个运算对象，如 取地址& 和取值 *
- 二元运算符：作用于两个运算对象，如 + - * 、 ==
  只有 && || ， ?: 明确是左边先运行
- 三元运算符：如条件表达式 ？ ，

> 重载运算符：当运算符作用域类类型的运算对象时，用户可以自定义其含义

- 左值：
- 右值：

左值可以位于赋值运算符的左侧，右值则不能

当一个对象被用作右值时，使用的是对象的值（内容）；当被用作右值时，用的是对象的身份（在内存中的位置）

```
cout<<i<<" "<<++i;    // 结果不可预料，视编译器而异
int i = f()+g();      // f() 与 g() 的调用顺序不确定
&&    ||    ,      ?:    // 明确是左边先运行
```

1.4.2 算术运算符

(-m)/n = -(m/n)

m/(-n) = -(m/n)

(-m)%n = -(m%n) // 如 -21 % -8 = -5

m%(-n) = m%n // 如 21% -5 = 1

1.4.3 逻辑和关系运算符

s被声明成了 对常量的引用：`for (const auto &s : text)` 因为 text 的元素是 string 对象，可能非常大，所以 将s声明成 引用类型 可以避免对元素的拷贝；因为不需要对 string 对象做写操作，所以s被声明成 对常量的引用

```
if(a == b<c)    // if a==1 or a==0
```

1.4.4 赋值运算符

```
int k = 3.13;    //OK
int k = {3.14}; // ERROR
```

1.4.5 递增和递减运算符

除非必要，否则不用递增和递减的后置版本。因为后置版本效率低

1.4.6 成员访问运算符

迭代器 vector::iterator

```
vector<string>::iterator iter;
*iter++;     // OK
(*iter)++;  //ERROR 没有字符串++
*iter.empty(); // ERROR
iter->empty(); // OK
++*iter;          // ERROR ++没有字符串
iter++->empty();  // OK  == iter->empty(); iter++;
```

1.4.7 条件运算符

:？ 三元运算符

```
(grade>90)?"great":(grade>60)?"good":"bad";
== (grade>90)?"great": ((grade>60)?"good":"bad") ;
```

```
#include <vector>
{
    vector <int> ivect ={1,2,3,4,5,6,78,9,10};
    for (auto &val : ivect)
        val = (val%2==0)?val:(2*val);
    for (const auto &val : ivect)
        cout<<val<<'\t';
}
```

1.4.8 位运算符

1.4.9 sizeof 运算符

```
sizeof(type)
sizeof expression

Sales_data *p;
sizeof *p;   // OK 因为 *p 并不执行
sizeof data.revenue;   //  == sizeof Sales_data::revenue;
```

```
#include <vector>
{
    vector<int> ivect = {1,2,3,4,5};   //sizeof ivect 返回固定值（视编译器），无论元素多少个
}
```

1.4.10 逗号运算符

从左向右依次求值，逗号表达式的结果是最右边表达式的值

1.4.11 类型转换

```
int ia[10];
int *ptr = ia;   // ia转换成指向数组首元素的指针
```

当数组被用作 decltype 关键字的参数，或者作为取地址符 &，sizeof 以及 typeid 等运算符的运算对象时，上述转换不会发生。

```
sizeof(ia);   //  == 数组占用内存大小
decltype(ia) ib;    // ib 也是数组
typeid       //
```

```
int *ptr = 0;   // ptr 指针初始化为空指针
if(ptr)      // 判断是否为空指针

int i;
const int &j = i;    // OK
const int *p = &i;   // OK
int &r =j; int *q = p;   // ERROR
```

强制类型转换 static_cast, dynamic_cast,const_cast 和 reinterpret_cast

```
cast-name <type> (expression);
cast-name : static_cast, dynamic_cast,const_cast 和 reinterpret_cast
```

1. static_cast: 任何具有明确定义了的类型转换，只要不包含底层const，都可以使用

```
double d;
int i = static_cast<int>(d);
void *p =&d;
double *dp = static_cast<double *>(p);   // 将 p 强制转换回去 double
*，前提是必须保证正确
```

2. dynamic_cast: 危险不使用
3. **const_cast: 只能改变运算对象的底层 const，即把常量转换为非常量，只有 指针和引用 可用**

```
const char *pc;
const char a;
char *q = pc;   // ERROR
char *p = const_cast<char *>(pc);   // OK,可以通过 p来修改pc的值，不推荐使用
char &m = const_cast<char &>(a);
char c;
char&m = const<const char &>(c);   // const_cast 也可以添加 const属性
```

   const_cast 常常用于函数重载的上下文中

4. reinterpret_cast: 危险不使用

1.5 语句

1.5.1 简单语句

switch 内部的变量定义

```
switch(flag)
{
    case 1:
        int i;      // OK
        int j =1; // ERROR
        break;
    case 0:
        cout<<&i;
}
switch(flag)
{
    case 1:
    {
        int i;      // OK
```

```
        int j =1; // OK
        break;
     }
    case 0:
        cout<<&i;   // ERROR undefined
}
```

```
#include <vector>
int main(void)
{
    vector <int >v1 ={1,2,3,4};
    vector <int >v2 ={1,2,3,4,5,6};
    auto it1 = v1.cbegin();
    auto it2 = v2.cbegin();
    while(it1!=vi.cend() && it2!= v2.cend() )
    {
        if(*it1 != *it2)
            {
                cout <<"no";
                break;
            }
            ++it1;
            ++it2;
    }
    if(it1 == v1.cend() )
        cout<<"v1 is pre of v2";
    if(it2 == v2.cend() )
        cout<<"v2 is pre of v1";

return 0;
}
```

范围 for 语句

```
for (declaration:expression)   // expression 必须为一个序列，如数组，
vector，string等类型的类，他们的特
   // 定是拥有能返回迭代器的 begin 和 end 成员
    statement;
// declaration 定义一个变量，序列中的每个元素都得能转换成该变量的类型，使用
auto 确保类型相容
```

```
vector <int> v ={0,1,2,3,4,5,6,7};
for (auto &r : v)   // 对于 v 中每个元素
    r *= 2;
// 每次迭代，都会重新定义 r，定义 r 为当前元素的引用

for(auto beg = v.begin,end = v.end; beg!=end; ++beg)
{
      auto &r = beg;
      beg *= 2;
}
```

在范围 for 中，不能增加 vector对象的元素，因为预存了 end() 的值。

```
#include <string>
using namespace std;
int main(void)
{
    string str1,str2;
    do{
    cout<<"enter two string:"<<end;
    cin>>str1>>str2;
    if(str1.size()>str2.size)
        cout<<str1;
    else if (str1.size()<str2.size)
        cout<<str2;
    else cout<<"equal";
    }
    while(cin);
}
```

1.5.5 跳转

goto，return，break，continue

1.5.6 try 语句块和异常处理

异常是指纯在于运行时的反常行为，这些行为超出了函数正常功能的范围。典型的异常包括失去数据库连接以及遇到意外输入等。

异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持，异常处理包括

- **throw 表达式**：异常检测部分使用throw表达式来表示它遇到了无法处理的问题，即throw引发raise了异常

- **try 语句块**：异常处理部分使用 try，以一个或多个catch子句结束，catch子句处理异常，也成为异常处理代码
- 一套异常类，用于在throw表达式和相关的catch子句之间传递异常信息

```
int main(void)
{
    cout<<"enter two integer";
    cin>>n1>>n2;
    if(n2!=0)
        cout<<n1/n2;
      else return -1;
    return 0;
}
```

throw 抛出异常

标准库异常类型：runtime_error 运行时检测出的问题

exception 最常见的问题

overflow_error 运行时错误：计算上溢

underflow_error 运行时错误：计算下溢

```
int main(void)
{
    cout<<"enter two integer";
    cin>>n1>>n2;
    if(n2!=0)
        cout<<n1/n2;
      else throw runtime_error("cannot be zero");   // 创建一个匿名的
runtime_error 对象
    return 0;
}
```

try 处理异常

```
try {
    program-statement
} catch (exception-declaration){
    handler-statement
} catch (exceptiion-declaration){
    hadler-statement
}   // ...
```

```
int main(void)
{
      int n2,n1;
    cout<<"enter two integer";
    while(cin>>n1>>n2)
     {
           try{    if(n2!=0)
                       cout<<n1<<"/"<<n2<<"="<<n1/n2<<endl;
                       else throw runtime_error("cannot be zero");   // 创
建一个匿名的 runtime_error 对象
               } catch (runtime_error err)
          {
                 cout<<"请确认错误信息："<<err.what()<<endl;   //  .what()
返回异常的文本信息（字符串）
                 cout<<"是否重新输入"<<endl;
                 char ch;
                 cin>>ch;
                 if( ch!='y' && ch != 'Y')
                       break;
          }
           cout<<"enter two integer";
     }
    return 0;
}
```

1.6 函数

函数的返回值不能是数组和函数，但是可以返回指向数组和函数的指针

自动对象：只存在于块执行期间的对象

局部静态对象：在程序的执行路径第一次经过对象定义语句时初始化，生命周期贯穿函数调用及之后的时间如 static

```
siez_t count_calls()   // 记录函数被调用次数
{
    static size_t ctr = 0;   // 调用结束后，变量仍有效
    return ++ctr;
}
```

一个函数不会被用到，那么它可以只有声明没有定义。

必须为每一个虚函数都提供定义，不管它是否被用到了。

```
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end);
```

分离式编译

1.6.1 参数传递

```
void reset(int *ip)
{
    *ip = 0;   // 实参指向对象的值改变了
    ip = 0;    // 只改变了ip的局部拷贝，实参未被改变
}
int i =42;
reset(&i);   // 只改变了i的值，i的地址没变
```

拷贝大的类类型对象或者容器对象时，一般通过引用形参访问该类型的对象。

```
void func(const int i);
void func(int i);   // ERROR ,重复定义，const int i的const会被忽略
```

形参中使用引用时，尽量使用 const 类型的引用。

```
void find(const char ch); // 传递参数时，可以传 char j; 也可以 const char j;
```

vecter类型的迭代器

```
vector <int>::iterator 是一种类型

vector <int>::iterator a;
vector<int> vec(10);
a = vec.begin();
a = vec.end();
```

数组形参

```
void print(const int *);    // 三种方法一样的
void print(const int[]);
void print(const int[10]);   // 10没有作用，不推荐
```

```
void print(const char *cp)
{
    if(cp)   // 判断是否为空指针
        while(*cp)   // 判断是否为空字符
            cout<<*(cp++)
}
```

数组的引用作为形参（很少用）

```
void print(itn (&arr)[10])   // arr 是具有10个整数数组的引用
{
    for(auto elem:arr)
        cout<<elem;
}
// void print(const int &arr[10]);  // false，arr是数组名，里面有10个元素，每个元素都是引用
```

多维数组作为形参

```
void print(int (*matrix)[10], int rowSize);    //  == void print(int matrix[][10], int rowSize)
// matrix 是指向含有10个整数的数组的指针，指向数组的首元素，该数组的元素是由10个整数构成的数组

int *matrix[10]    // matrix 是一个数组，数组中含有10个指针作为元素
```

如何交换指针的地址？？

```
void swapAddress(int *pt1,int *pt2)   // false,针对形参做的操作，并没有改变实参
{
    int *temp = pt1;
    pt1 = pt2; pt2 = temp;
}

void swapAddress2(int *(&pt1), int*(&pt2))   // pt1，pt2是指针的引用
{
    int *temp = pt1;
    pt1 = pt2; pt2 = temp;
}
```

1.6.2 main 处理命令行选项

main 函数的参数

```
int main(int argc,char *argv[]){}   // argv 是一个数组，元素是指向字符的指针，即字符串数组

./a.out hi world
// 此时，argc =3，argv[0] = "./a.out"
                  argv[1] = "hi'"
                  argv[2] = "world"
atoi() 函数可以把字符串转换成 int
```

1.6.3 含有可变形参的函数

函数处理不同数量参数的函数，C++ 11 有两种方法

1. **intializer_list：全部实参类型相同，个数未知**
2. **编写一种特殊函数，即可变参数模板**
   intializer_list的用法，参数都是常量，不可在调用函数中修改

```
#include <initializer_list>
void error_msg(initializer_list <string> i1)
{
    for(auto beg = i1.begin(); beg!=i1.end(); ++beg)
        cout<<*beg<<' ';
}
error_msg( {"one", "two", "three"} );
```

```
void error_msg(initializer_list <string> i1)   // 等效于上面
{
    for(const auto &elem:i1)
        cout<<elem<<' ';
}
```

计算参数列表中整数的总和

```
int sum(initialzer_list<int> ilist)
{
    int sum = 0;
    for(const auto &elem:ilist)   // 使用 const &的目的是，避免赋值拷贝占用时间和内存，同时不修改实参
        sum += elem;
      return sum;
}
```

省略符作为形参，只用于与C函数交互的接口程序。很少用

```
void foo(...);
void foo(parm,...);
```

1.6.4 返回类型和 return 语句

无返回值的函数可以加 return; 程序运行到此直接结束。

不要返回局部对象的指针和引用，因为函数终止局部变量的引用将指向不再可用的空间

```
const string &func()
{
    string temp;
    return temp;   // ERROR
    return "hi";   // ERROR  "hi"是局部临时量
}
```

左值：可以被赋值，可以取地址操作

调用一个返回引用的函数得到左值，其他返回类型是右值

列表初始化返回值

```
vector<string> process()
{
    return {};   // 返回空 vector 对象
    return {"1"};   // OK
    return {"3"," ","2"};   // OK,因为函数返回和初始化赋值是一样的操作
}
int integer = {1};   // OK  return {1};  OK
```

main 函数不能调用自己

```
vector<int> v1 = {1,2,3,4,5};
print(v1,0);
int print(vector<int> vint,int index)
{
    if(index < vint.size())
    {
        cout<<vint[index];
        print(vector<int> vint,index+1);
    }
}
```

返回数组的指针

typedef int arrT[10]; // arrT 是一个类型别名，表示的类型是含有10个整数的数组

using arrT = int [10]; // 与上面等效

arrT *func(); // 函数返回一个指向含有10个整形数组的指针

```
int arr[10];
int *ptr1[10];   // ptr1 是一个数组
int (*ptr2)[10] = &arr;   // ptr2 是数组的指针
int *ptr3 = arr;
```

方法0：使用类型别名

```
typedef int arr[10];
arr *func(int i);
```

方法一：定义返回数组的指针的函数

```
int (*func (int i) ) [10];
```

方法二：尾置返回类型

```
auto func(int i) -> int(*)[10];
```

方法三：使用 decltype

```
int odd[] = {1,2,3,4,5};
decltype(odd) * func(int i)
{}
```

返回数组的引用，即将 * 改为 &

1.6.5 函数重载

函数的形参个数不同或者形参类型不同的同名函数，称为函数重载。与函数的返回值无关

main 函数不能重载

```
Record lookup(const Account&); // 函数重载  == Record lookup(const Account
&acct); // 函数重载
Record lookup(const Phone&);    // 函数重载    因为函数声明时候可以不用写形参
名称
Record lookup(const Name&);      // 函数重载
```

```
Record lookup(const Account&);
bool lookup(const Account&);   // 错误，函数重复定义

typedef   Account T;
Record lookup(const Account&);
Record lookup(const T&);    // 错误，函数重复定义
```

```
Record lookup(Phone);
Record lookup(const Phone);    // 错误，函数重复定义

Record lookup(Phone *);
Record lookup(Phone * const);   // 错误，函数重复定义
```

如果形参是某种类型的指针或引用，则可以通过 const 来实现函数重载

```
Record lookup(Phone &);
Record lookup(const Phone &8);    // 新函数，作用于常量引用

Record lookup(Phone *);
Record lookup(const Phone *);    // 新函数，作用于指向常量的引用
```

---

const_cast 和 重载

const_cast 可以去 const 化

```
const string &shortStr(const string &str1,const string &str2)
{
    return (str1.size()<=str2.size())?str1:str2;
}
// 函数可以接收 非常量的string，但是返回的结果仍然是 const string 类型
// 解决方法
string &shortStr( string &str1, string &str2)
{
      // 首先，添加 const,使得可以调用上面函数
      auto &r=shortStr( const_cast<const string &>(str1), const_cast<const
string &>(str2));
    return const_cast<string &>(r);   // 去 const 化
}
```

重载与作用域

```
void print(const string&);
void print(double);
void print(int);
void func()
{
    print("hi");   // 调用 void print(const string&)
    print(3);       // 调用 void print(double)
    print(3.14);   // 调用 void print(int)
}
```

```
void print(const string&);
void print(double);

void func()
{
    void print(int);   // 因此，不要在函数内放函数声明
    print("hi");   // 错误 void print(int) 把前面函数声明覆盖掉
    print(3);       // 调用 void print(int)
    print(3.14);   // 调用 void print(int)
}
```

1.6.6 特殊用途

默认参数：函数声明中提供默认值（函数定义中不用加），从右往左提供，因此，默认参数一般放在形参右侧

内联函数和 constexpr 函数

- 内联函数可避免函数调用的开销，一般用于规模较小，流程直接，频繁调用的函数
- constexpr 函数：**能用于常量表达式的函数，函数形参和返回值都是字面值类型，且函数体中有且只有一个return**

```
oncstexpr int new_sz(){return 42;}   // OK
constexpr int foo = new_sz();   // OK,foo是一个常量表达式  ==constexpr
int foo = 42;
```

  constexpr函数是隐式的内联函数

内联函数 和 constexpr 函数通常定义在头文件 .h 中

assert 预处理宏断言

预处理宏其实是一个预处理变量，行为类似于内联函数

```
#include <cassert>
assert (expr);   // 如果 expr 为假，则终止程序；否则什么都不做
```

NDEBUG 预处理变量

如果定义了 NDEBUG，则 assert 不起作用

```
g++ -D NDEBUG assert.cpp   // 与在程序中 #define NDEBUG 一样效果
./a.out
```

使用 NDEBUG 在程序中添加调试代码

```
void func()
{
    ...
    #ifndef NDEBUG
        cout<<"输出调试信息";
        cerr<<__func__;   // 输出存放函数的名字
    #endif
}
```

1.6.7 函数匹配

```
void func(int);
void func(int,int);
void func(double,double i =2.2);
func(2.2);       // 调用第三个
func(3,2.2);   // 第二个月与第三个一样匹配，二异性-，故报错
```

1.6.8 函数指针

函数指针：指向函数的指针

```
bool lengthComp (const string &,const string &);   // 函数声明
bool *pf (const string &,const string &);            // 函数声明，pf 函数返回 bool 类型的指针
bool (*pf) (const string &,const string &);          // 函数指针，定义函数指针
// 函数指针赋值
pf = 0;   // OK,不指向任何函数
pf = lengthComp;
pf = &lengthComp;    // 与上面等效
// 使用函数指针调用函数
bool b1 = pf("hi","world");   // == lengthComp ("hi","world");
bool b1 = (*pf)("hi","world");   // 与上面等效
```

重载函数的指针

```
void ff(int *);
void ff(unsigned int);

void (*pf)(unsigned int) = ff;   // 指向的函数是 void ff(unsigned int)
```

函数指针作为形参

形参看起来是函数类型，实际上是函数的指针

```
void useBigger( bool (*pf)(const string &,const string &));
void useBigger( bool pf(const string &,const string &));    // 与上面等效
// 使用
useBigger( lengthComp);
```

使用 typedef 为函数指针起别名

```
// 函数类型,Func,Func1
typedef bool Func(const string &, const string &);
typedef decltype(lengthComp) Func1;   // 与上面等效

// 函数的指针,Func2,Func3
typedef bool (*Func2)(const string &, const string &);
typedef decltype(lengthComp) *Func3;   // 与上面等效


// 使用类型别名
void useBigger( Func );
void useBigger( Func2 );   // 与上面等效，因为形参看起来是函数类型，实际上是函数的指针
```

返回指向函数的指针

最简单的方法是使用类型别名

```
using F = int(int*,int);   // 函数类型
using PF = int(*)(int*,int); // 函数指针类型
// 使用
PF func1(int);   // func 返回指向函数的指针
F *func2(int);   // 与上面等效
F func2(int);    // 错误，不能返回函数
```

更复杂的方法

```
int (*fun3(int))(int *,int);
int func4(int) -> int (*)(int *,int); // 与上面等效
// 返回函数的指针，函数类似于 int (*)（int *,int）
```

将 auto 和 decltype 用于函数指针类型

decltype 作用于一个函数时，返回的是函数的类型

```
int func5(int);
decltype (func5) *func6(const string &); // func6 函数返回一个函数指针，指向类似于int func5(int)的函数
```

练习：编写函数声明，接受int参数，返回类型是int；然后声明一个 vector 对象，元素是指向该函数的指针

```
int func(int, int);   int func2(int, int);
using pf = int (*)(int, int);
vector<pf> a;
// 其他方法：
vector<int (*)(int, int)> a = {func,func2};
vector<decltype(func) *> a;
// 通过 vector 使用里面的对象
for (auto funcPtr:a)
      funcPtr(22,11);
```

1.7 类

类的基本思想是：数据抽象(data abstraction)和封装(encapsulation)，数据抽象是一种依赖于接口(interface) 和实现 (implementation)分离的编程

class 与 struct 几乎一样，唯一区别：struct 的访问权限默认的 public，而 class 则相反。

友元函数：其他类或者其他函数访问它的private成员

```
class Sales_date
{
    public:
    friend void func();   // 定义时写 void func()
};
```

构造函数：类通过一个或几个特殊的成员函数来控制器对象的初始化过程

构造函数名字和类名相同，没有返回类型。构造函数不能声明为const

当创建一个const对象时，直到构造函数完成初始化过程，对象才取得const属性。因此，构造函数在const对象的构造过程可以向其写值。

默认构造函数：不带任何参数或者所有参数都具有默认值

```
class Sales_date
{
    string bookNo;
    unsigned int units_sold =0;
    double price;
    double revenue = 0;
    public:
      Sales_data(){}   // == Sales_data()=defaule;
    Sales_data(const string &s):bookNo(s){}
    Sales_data(const string &s,unsigned int n,double
p):bookNo(s),units_sold(n),price(p){}
};
```

编译器会默认生成以下函数：构造，拷贝，赋值和析构

内置基本类型可以使用默认的拷贝赋值，但是当成员数据里面有指针时（动态内存分配），默认的就不行

---

class 内定义类型别名，类型别名出现在类开始的地方，因为要先定义后使用。

```
class  Screen
{
    public:
        typedef std::string::size_type pos;
    private:
        pos cursor = 0;
};
```

内联函数的定义一般写在 .h 文件中，内联函数的定义可以直接跟在声明后面，也可以在 .c 和 .h 中定义

可变数据成员 mutable

```
private:
    mutable size_t access_ctr;   // 即使在一个 const 对象内也能被修改
    std::vector<Screen> screens{Screen(24,80,' ')};
public:
    void some_member() const;

void Screen::some_member() const   // const 成员函数不能修改成员变量
{
    ++access_ctr;
}
```

返回对象的引用

```
public:
    void some_member() const;
    Screen &display() const;   //ERROR
    const Screen &display() const;   // OK, const成员函数要是返回引用，必须返回const引用
```

一个类的成员变量不能是类自己的类型，然而类出现即认为是声明过了，因此类允许包含定义指向 该类的指针或引用作为成员变量

友元类：一个类中想要操作另一个类中的数据

```
class Screen
{
    friend class Window_mgr;   //Window_mgr 的成员函数可以访问 Screen 类的私有部分
};
```

友元成员函数：

```
void Window_mgr::clear();
class Screen
{
    friend void Window_mgr::clear();   // Window_mgr::clear() 函数可以访问 Screen 类的私有部分
};                  // // Window_mgr::clear() 函数需要在前面先有声明
```

类的作用域：使用作用域访问运算符 ： ：

```
class  Screen
{
    public:
        typedef std::string::size_type pos;
        pos func();
    private:
        pos cursor = 0;
};

// .c 中
Screen::pos Screen::func()
{}
```

编译器编译完全部声明后才会处理成员函数的定义

```
class  Screen
{   public:
        void func1(){ func2() };   // OK
        void func2(){ height = 0; };   // OK
    private:
        int height;
};
```

1.7.1 构造函数

列表初始化：效率高，给const和&引用的数据变量赋值（必须使用列表初始化），或者成员是某种未提供默认构造函数的类的类型

委托构造函数

没有创建任何构造函数时，编译器才会合成默认构造函数

默认构造函数：不带参数或所有参数都带有默认值的构造函数

隐式的类类型转换

```
class Sales_data
{
private:
    string bookNo;
public:
    Sales_date(const string &s):bookNo(s){}
    combine(const Sales_data);
};
```

```
int main()
{
    string null_book = ;
    Sales_date item;
    item.combine(null_book);   // OK,会先创建一个无名类的对象
Sales_date(const string &s):bookNo(s){}
                               // string 隐式转换为 Sales_date 类型
      item = null_book;   // string 隐式转换为 Sales_date 类型
      item = string("9-99");   // OK, string 隐式转换为 Sales_date 类型
      item = "9-99";   // FALSE, "9-99" ->string->Sales_data  转换了两次
}
```

只有一个形参的构造函数才会发生隐式转换

因为有了构造函数 Sales_date(const string &s):bookNo(s){}，所以才允许隐式转换

抑制构造函数定义的隐式转换 explicit

```
explicit Sales_date(const string &s):bookNo(s){}   // 类外定义时不用加
explicit
// 但是依然可以显式的转换
item.combine(null_book);   // FALSE 隐式
item.combine(Sales_data(null_book));   // OK 显式
item.combine(static_cast<Sales_data>(null_book)); // OK 显式
```

聚合类使得用户可以直接访问其成员，并且具有特殊的初始化语法形式。

集合类满足以下条件：

- 所有成员都是public
- 没有定义任何构造函数
- 没有类内初始值（类内初始值，如 class a{ int num = 0}; ）
- 没有基类，也没有virtual 函数

字面值常量类

1100， true，2.3， 类（100，200） 都是字面值

字面值常量类：（1）数据成员都是字面值的聚合类，

（2）或者非聚合类，但满足以下条件

- 数据成员必须是字面值类型

- 类至少有一个constexpr 构造函数
- 如果数据成员中含有类内初始值，则内置构造函数的初始值必须是一条常量表达式；或者如果成员属于某种类类型，则初始值必须使用成员自己的 constexpr 构造函数
- 类必须使用析构函数的默认定义，该成员负责销毁类的对象。

第一种字面值常量类

```
class Point
{
public:
    int x;
    int y;
};

// 使用
const Point start(100,200);
```

第二种：

```
class Point
{
private :
int x; int y;
public:
constexpr Point(int a,int b):x(a),y(b){}
};

// 使用
constexpr Point start(100,200);
```

1.7.2 类的静态变量

静态数据成员不属于任何一个类的对象，只和类关联，他们不是由构造函数初始化，静态数据成员只能初始化一次，一般在类的外部初始化

静态成员函数不能声明为 const，不包含 this 指针

```
class Account
{
      static double interestRate; // 要在类的外部初始化，const类型是例外

public:
```

```
    static void rate(double);   //外部定义函数时不用加 static
}

double Account::interestRate =9.9;
void Account::rate(double n)   //外部定义函数时不用加 static
{
      interestRate=n;
}
// 使用静态成员函数
Account::rate(3.14);
Account ac1;
ac2.rate(2.2);
Account *ac2;
ac2->rate(2.2);   // 通过指针或引用调用静态成员函数
```

在类内定义常量

1. 类内使用枚举
2. 使用 static const

```
class Test
{
enum{Months = 12};
static const int MONTH = 12;
}
```

静态数据成员的类型可以是该类的类型，而普通成员变量（除指针外）不行

```
class Bar
{
    private:
        static Bar mem1;   // OK，静态成员可以是不完全类型
        Bar *mem2;   // OK，指针成员可以是不完全类型
        Bar mem3;    // FALSE,数据成员必须是完全类型
}
```

1.8 I/O库

I/O库包括 iostream，fstream，sstream

```
istream   // 输入流类型，提供输入操作
ostream   // 输出流类型
cin        // 一个 istream 对象，从标准输入读取数据
cout       // 一个ostream 对象
cerr       // 一个ostream对象，通常用于输出程序错误信息，写入到标准错误
```

头文件	类型
iostream	istream从流读取数据,ostream，iostream读写流
fstream	ifstream从文件读取数据,ofstream，fstream文件流
sstream	istringstream从string读取数据，ostringstream

通常可以将一个派生类对象当作其基类对象来使用。

IO 对象没有拷贝或赋值操作，但是可以以引用方式传递和返回流

```
ofstream out1,out2;
out1 = out2;   // error
out2 = print(out2); // error
```

IO库中有一些条件状态

```
int ival;
cin>>ival;   // 当输入的不是int类型时，cin会进入错误的状态，一旦发生错误，后续的IO操作都失败

// 通常使用 while一直判断cin 的状态
while(cin>>word)
```

查询流的状态

```
cin.clear();   // 使 cin 有效
cin.clear(cin,rdstate() & ~cin.failbit & ~cin.badbit);   // 清除多个错误位
```

1.8.2 文件输入输出

1.8.3 string 流

第二章 C++进阶

2.1 C基础

编译器如何编译程序的

广义编译器包含：预处理器、编译器、汇编器、链接器

点击 build，处理过程

> file.c file.h → 预处理器 → file.i → 编译器 → file.s → 汇编器 → file.o
>
> 源代码 → 预处理文件 → 汇编代码 → 机器指令

1.预编译

预处理指令示例：gcc -E file.c -o file.i

- 处理所有的注释，以空格替代
- 将所有的**#define**删除，并**展开所有宏定义(文本替换)**
- 处理条件编译指令**#if**、**#ifdef**、**#elif**、**#else**、**#endif**
- 处理**#include**，展开被包含的文件
- 保留编译器需要使用的**#progma**指令

2.编译：将源程序翻译成**汇编代码**

编译指令示例：gcc -S file.i -o file.s

- 对预处理文件进行**词法分析**、**语法分析**、**语义分析**
  词法分析：分析关键字、标识符、立即数等是否合法
  语法分析：分析表达式是否遵循语法
  语义分析：在语法分析基础上进一步分析表达式是否合法
- 分析结束后进行**代码优化生成相应的汇编代码文件**

3.汇编：将汇编代码翻译为**机器指令**（目标文件，是二进制文件）

汇编指令示例：gcc -c file.s -o file.o

- 汇编器将汇编代码转变为机器的可执行指令
- 每条汇编语句几乎都对应一条机器指令

4.链接：将目标文件链接，生成最终的可执行程序

链接示例：gcc file.o

每一个.c被编译后都会生成一个.o文件，链接器把各个模块之间的相互引用的部分处理好，使得各个模块之间能够正确的衔接。

1. 静态链接：目标文件**直接链接进入可执行文件**
2. 动态链接：在程序**启动后才加载**目标文件

（1）静态链接：右链接器在链接时将库的内容直接加入到可执行程序中

> file1.o file2.o libc.c → 链接器 → a.out

a.out不依赖三个文件，因为链接时候已经包含进去了

Linux下静态库的创建和使用

```
示例：将 lib.c 变成静态库
编译静态库源码：gcc -c lib.c -o lib.o
生成静态库文件：ar -q lib.a lib.o   // 本质就是打包，生成 lib.a
使用静态库编译：gcc main.c lib.a -o main.out

// lib.c
char* name(){ return "zzh"; }

// main.c
#include <stdlib.h>
extern char* name();
int main(){ printf("%s",name()); }

// 删除掉 lib.c lib.a lib.o 之后，main.out 仍然可以运行
```

（2）动态链接

- 可执行程序在**运行时才动态地加载库**进行链接
- **库的内容不会进入可执行程序中**

lib1.so stub1

lib2.so stub2

stub1 stub2 → 链接器 linker → main.out

Linux下动态库的创建和使用

```
示例：源程序 dlib.c 编译成动态库 dlib.so 并使用该动态库
编译动态库源码：gcc -shared dlib.c -o dlib.so
使用动态库源码：gcc main.c -ldl -o main.out    // -ldl 编译选项，说明会使用动态链接的方式
关键系统调用：
dlopen:   打开动态库文件
dlsym:   查找动态库中的函数并返回调用地址
dlclose:   关闭动态库文件

// dlib.c
char* name(){ return "dynamic lib"; }

// main.c
#include <stdlib.h>
#include <dlfcn.h>
extern char* name();
int main(){
      void* pdlib = dlopen("./dlib.so",RTLD_LAZY);
      char* (*pname)();
      if(pdlib != NULL)
     {
            pname = dlsym(pdlib,"name");
            if(pname != NULL)
           {
                 printf("%s",pname());
           }
            dlclose(pdlib);
     }
      else
            printf("Cannot open lib...");
      return 0;
}

// 删除掉 dlib.s 之后，main.out 会 Cannot open lib...
```

小结：静态链接和动态链接各有利弊

- 静态链接适用于小程序，就一个可执行文件，在哪一台机器都可运行
- 动态链接可以使用动态库部分更新应用程序，如大型网络游戏

```
typedef int INT32;
typedef unsigned char BYTE;
typedef struct _tag_str
{
    BYTE b1;
    BYTE b2;
    short s;
    INT32 i
}STR;
STR i32;
printf("%d, %d",sizeof(STR),sizeof(i32));   // 8, 8
```

变量本质是一段实际**连续存储空间的别名**

char 1字节，short 2字节，int 4字节

> %s 字符串　%d 整数　%f 浮点数

数据类型和变量

数值（字面量）的默认类型

2为int，0.2为double，‘c’为char，0.2f为float

大类型赋值给小类型，可能发生溢出（1）但数值在小类型范围内，赋值成功（2）超过小类型范围，赋值失败

小类型可以安全赋值给大类型

浮点类型赋值给整型，会发生截断

整数赋值给浮点类型，赋值成功

无符号数以原码形式保存，有符号数以补码（正数的原码取反+1）

无符号数与有符号数相加时，有符号数将被转化为无符号数再进行计算，结果为无符号数

浮点数

存储方式：**符号位**，**指数**，**尾数**

float和double类型的数据在计算机内部表示法相同，但是由于所占存储空间不同，且分别能够表示的数值范围和精度不同

8.25的小数表示：1000.01 → 1.00001*（2^3）

符号位　　指数（127+3）　　尾数（00001）

0　　10000010　　00001000000000000000000（23位）

```
float f = 8.25;
unsigned int* p = (unsigned int*)&f;
printf("0x%08X",*p);
```

int 范围：[-2^31, 2^31 - 1]

float 范围：[-3.4*10^38, 3.4*10^38]

- 它们都占4个字节内存，说明**能表示的具体数字的个数是一样**的，
- float可表示的数字之间**不是连续**的，存在间隙，有的浮点数无法精确表示
- float只是一种**近似的表示法**，不能作为精确数使用
- float的运算速度比int慢得多
- double与float在内存中表示法相同，也是不精确的，但是double所占内存较多，所能表示的精度比float高

类型转换

强制类型转换，隐式类型转换

```
short s = 0x1122;
char c = (char)s;   // 0x22
int i = (int)s;    //  0x1122
unsigned int p = (unsigned int)&s;   // 64位机器会发生截断，只取后面32位，因为 unsigned int 是4个字节
printf("\n p = %x,&s = %p, c = %x,ii = %x",p,&s,c,i);
```

高类型到低类型的隐式转换是不安全的，将导致不正确的结果

隐式类型转换的发生点

- 算术运算中，低类型转换为高类型
- 赋值表达式中，表达式的值转换为左边变量的类型
- 函数调用时，实参转换为形参的类型

- 函数返回值，return表达式返回为返回值类型

> 安全的隐式转换：
>
> - char → int → unsigned int → long → unsigned long → float → double
> - short → short
> - short → int

变量属性

在定义变量的时候可以加上"属性"关键字，指明变量的特有意义

```
property type var_name
// 如
auto char i;
register int j;
static long k;
extern double d;
```

> - auto 即C语言中**局部变量的默认属性**，表明该变量存储在栈上，编译器默认所有的局部变量都是auto的
> - register 指明将**局部变量存储于CPU寄存器中而不是内存中**，只是请求，不一定成功，不能用&获取register变量的地址（&只能取内存中的变量）
> - static 指明变量的“静态”属性，**static修饰的局部变量存储在静态区**，同时具有“作用域限定符”的作用（1）static全局变量作用域只是声明的文件中（2）static函数作用域只是声明文件中
> - extern 用于声明"外部"定义的变量和函数,extern变量和函数在文件**其他地方分配空间和定义**

```
// C++和变种C编译器默认会按“自己”的方式编译函数和变量，通过extern可以命令编译器，以标准C方式编译
// extern 指定编译方式
extern "C"
{
    int f(int a, int b)
        return a+b;
}
```

---

case语句中的只能是整型或字符型，按字母或数字顺序排放各条语句

C的goto禁用

void 修饰函数返回值和参数是为了表示无

void* （1）作为左值时，接收任意类型的指针；（2）作为右值时，需要进行强制类型转换

```
f()   // 可以接受任意参数，默认返回int
{}
int i = f(1,2,3);
```

ANSIC：标准C语言规范

扩展C：在ANSIC的基础上进行了扩充，如gcc

sizeof(void) 在扩展C是合法的，gcc中是1

```
int* pi = (int*)malloc(sizeof(int));
void* p = NULL;
p = pi;   // OK
pi = p;   // ERROR
pi = (int*)p; // OK
```

---

const

const变量是只读的（不能出现在=左边），本质还是变量。const局部变量在栈上分配空间，全局变量在全局数据区分配。const只在编译器有用，运行期无用。

const全局生命周期变量（包括static const局部变量）存储在全局只读区还是普通全局区（能修改），视编译器而异

const 修饰函数参数表示在函数体内不希望改变参数的值

const 修饰返回值表示返回值不可改变，多用于**返回指针的情形**

C的字符串字面量存储于自读存储区，要用 const char* s = "perfectDiary"

---

volatile

编译器警告提示字

- volatile告诉编译器必须每次去**内存中取变量值**
- 主要修饰可能被**多个线程访问的变量**
- 也可以修饰可能被未知因数更改的变量

```
int obj = 10
a = obj;
sleep(100);
b = obj;   // 编译器会优化，把10直接替换obj
```

---

struct

struct空结构体的大小

```
struct STRC{}
struct STRC s1,s2;
sizeof(STRC);sizeof(s1);
// 0,0，s1和s2的地址相同
```

柔性数组即大小待定的数组

```
struct SoftArray{
int len;
int array[];   // OK，不占用空间，只是待使用的标识符
}
sizeof(struct SoftArray);   // == sizeof(int)
// 使用柔性数组
struct SoftArray* sa = NULL;
sa = (struct SoftArray*)malloc(sizeof(struct SoftArray*)+sizeof(int)*5);
sa->len = 5;
//
struct SoftArray* create_soft_array(int size)
{
      struct SoftArray* ret = NULL;
      if(size>0)
     {
            ret= (struct SoftArray*)malloc(sizeof(struct
SoftArray*)+sizeof(int)*5);
            ret->len = size;
     }
            retnru ret;
}
void init_soft_array(struct SoftArray* sa)
{
      int i = 0;
      if(NULL != sa)
     {
            for(i=0;i<sa->len;++i)
           {
```

```
                    sa->array[i] = i;
                }
            }
        }
```

union

语法上与struct类型，只分配最大成员的空间，所有成员共享该空间

union受系统大小端(小端模式：低地址在低位)的影响

```
union C{int i; char c;};   // 原理：c取地址最后一个字节的值
union C uc;
ui.i = 1;
printf("%d\n",uc.c);   // 1:小端  0:大端
```

---

enum

C中真正意义上的常量

用来**定义常量**，或**离散的整型值**

sizeof是编译器的内置提示符，是关键字而不是函数，sizeof的**值在编译期就已确定**，编译后会用值替换掉sizeof

```
int var = 0;
int size = sizeof(var++);   // sizeof(func());在运行时func()不会被调用
printf("%d,%d",var,size);
// 0, 4
```

---

typedef的意义：给一个已经存在的数据类型重命名，本质上不能产生新的类型

\ 接续符

位运算

&	**按位与**
|	按位或
~	取反
^	**按位异或**
<<	高位丢弃，低位补0，相当于*2的n次方
>>	高位补符号位，低位丢弃

左移操作数必须为整数类型（char和short被隐式转换为int）

右操作数范围 [0,31]

运算优先级：四则运算 > 位运算 > 逻辑运算

```
#define SWAP(a,b)     \      // 宏只能占一行
{                    \
    a = a + b;        \
    b = a - b;        \
    a = a - b;        \
}
#define SWAP2(a,b)     \
{                    \
    a = a ^ b;        \
    b = a ^ b;        \    // == a^b^b = a
    a = a ^ b;        \    // == a^b^a = b
}
```

++和--参与混合运算的结果是不确定的

```
int i = 1
int j = ++i+++i+++i;   // C采用贪心法读取，   →  ++i++  → 1++ 报错
int c = i+++j;   // C采用贪心法读取，   →  i+++j  → 1+j
```

空格是休止符，贪心法读取到空格会立马停止读取（而对已读到的内容进行处理），尽量使用空格

```
int j = ++i + ++i + ++i;    // == (++i) + (++i) + (++i)
```

三目运算符

三目运算符返回的是值，返回值会通过隐式转换到a和b中较高类型

```
((a<b) ? a : b) = 3;   // ERROR,三目运算符返回的是值，而不是变量

/* 用地址实现 */
*((a<b) ? &a  :&b) = 3;   // OK
```

```
char c = 0; int i = 0; char* p = "str";
printf("%d",sizeof(c ? c : i));   // 4
printf("%d",sizeof(c ? c : p));   // ERROR
```

逗号表达式

```
exp1, exp2, exp3, exp4
```

- “**粘帖剂**”，将多个子表达式连接为一个表达式
- **最终结果是最后一个子表达式的值**
- 前N-1个子表达式可以没有返回值
- 按照**从左向右顺序**计算每个子表达式的值

```
#include <assert>
int strlen(const char* s)
{
    assert(s);
    return ((*s) ? strlen(s+1)+1 : 0)
}
/* 利用逗号表达式，一行实现 */
int strlen(const char* s)
{
    return assert(s), ((*s) ? strlen(s+1)+1 : 0)
}
```

2.2 C与C++的区别

区别1：register

C++中的register只是一个兼容的作用，因为C++有自己的优化方式，不需要register

C如果对register变量取地址，会报错

C++中对register变量取地址，那么编译器会直接把register去掉，让变量存储在内容中，并取出地址。因为取地址是针对内存，而不是寄存器

区别2.1：const

C语言中：

- const 修饰的变量是只读的，本质还是变量
- const修饰的**局部变量在栈上分配内存**（可以在运行时修改其值）
- 修饰的**全局变量在只读存储区分配空间**
- const只在编译器有用，在运行期无效

总结：const修饰的变量不是真的常量，它只是告诉编译器，该变量不能出现在赋值符号的左边。const使得变量具有只读属性，const将具有全局生命周期的变量存储与只读存储区。**const不能定义真正意义上的常量，C语言中真正意义上的常量只有enum**

```
int main()
{
    const int C= 0;
    int *p = (int *)&C;
    *p = 5;
    printf("C = %d\n",C);
    printf("*p = %d\n",*p);
    return 0;
}
/*
gcc编译：输出 C = 5，*p = 5
g++编译：输出 C = 0，*p = 5，&C与p的相同
*/
```

C++中对const进行进化，const int C = 0;是真正意义上的常量，完全兼容C中const的语法特性

- 当碰到const声明时**在符号表中加入常量**
- 编译过程如果发现使用常量则**直接以符号表中的值替换**
- 编译过程如果发现下述情况则给对应的常量分配存储空间：（1）对const常量使用extern（2）对const常量使用&操作符

注意：C++编译器虽然可能为const常量分配空间，但不会使用其存储空间的值，只会使用符号表中的值

总结：

C语言中的const变量是只读变量，会分配存储空间

C++中的const变量可能会分配存储空间，（1）当const常量为全局，并且需要在其他文件中使用，（2）当使用&对const 常量取地址

区别2.2：const与宏

const常量用编译器处理（类型检查和作用域检查），而宏由预处理器处理，单词的文本替换

```
void f(){
    #define a 3
    const int b = 4;
}
void g(){
    printf("a = %d\n",a); // 编译之前已经做了文本替换
    //printf("b = %d\n",b);  // 加上会报错，因为b是局部作用域
}

int main(){
    const int A = 1;
    const int B =2;
    int array[A+B] = {0};
    int i = 0;
    for(i=0;i<(A+B);i++)
    {
        printf("array[%d] = %d\n",i,array[i])
    }
    f();
    g();
    return 0;
}
/*
gcc:报错 int array[A+B] = {0}; // A B是变量
g++:OK
*/
```

区别3：bool

C++新引入 bool 类型（C语言中用 int 替代）和引用 &

- bool 是C++中的基本数据类型，可取的值只有 true 和 false，理论上只占一个字节
- 可以定义 bool 全局变量，常量，指针，数组

区别4：三目运算符

C++对三目运算符进行了升级

C语言中，三目运算符返回的是变量值，不能将结果作为左值

C++中，的三木运算符直接返回变量本身，既可以作为左值，又可以作右值。特例：如果三目运算符返回的值中有一个是常量，则不能作为左值使用

```
int a =1,b =2;
(a < b ? a : b) = 3;
printf("a = %d, b = %d",a,b);
/*
gcc:报错 (a < b ? a : b) 是一个值，不能作为左值
g++:OK a = 3, b = 2
修改为：(a < b ? a : 2) = 3; 则会报错
*/
```

1. **当三目运算符的可能返回都是变量时，则返回的是变量的引用**
2. **当三目运算符的可能返回中有常量时，返回的是值。（C返回的都是值）**

区别5：& 引用

变量是一段实际连续存储空间的别名，引用可以看作一个已定义变量的别名，普通引用在定义时**必须用同类型**的变量进行初始化

```
int a=4;
float& b = a; // ERROR，C++是强类型的语言
int& c;   // ERROR
int& d = 1;   // ERROR
```

引用的本质：

引用作为变量的别名而存在，在一些场合可以替代指针

引用相当于指针来说具有更好的可读性和实用性

```
void swap(int& a,int& b)   // 函数中的引用形参不需要进行初始化
{
    int t = a;
    a = b;
    b = t;
}
void swap(int* a,int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

const 引用让变量拥有只读属性

```
int a =4;
const int& b =a;
int* p = (int*) &b;
b = 5;   // ERROR
*p   = 5; // OK
```

使用常量对const引用初始化后将生成一个只读变量

```
const int& a =4;

int* p = (int*) &a;
a = 5;   // ERROR
*p   = 5; // OK，a修改为5
```

引用的特殊意义：要让一个变量具有只读属性，就定义const引用

引用有自己的存储空间吗？

```
struct TRef{
char &r;
};

char c = 'c';
char& rc = c;
TRef ref = {c};
printf("sizeof(char&) = %d\n",sizeof(char&));   // 1
printf("sizeof(rc) = %d\n",sizeof(rc));   // 1
printf("sizeof(TRef) = %d\n",sizeof(TRef));   // 4
printf("sizeof(ref) = %d\n",sizeof(ref.r));   // 1  == sizrof(c)
```

引用本质上是指针，占8个字节，与指针一样

引用在编译器的内部实现：指针常量

```
int &a;     ==    int* const a;
a=5 ;       ==    *a = 5;
```

C++的引用旨在大多数情况下替代指针：

优点：

- 简单易用，又功能强大
- 功能可以满足多数使用指针的场合，可以避开由于指针操作不当而带来的内存错误

缺点：不要返回局部变量引用

```
int& demo()   // ERROR, 相当于返回野指针
{
    int d = 0;
    return d;
}
int& func()   // OK
{
    static int s = 0;
    return s;
}
```

区别6：内联函数

宏是预处理器进行文本替换，没有任何编译过程，因此可能出现副作用

C++中，const常量可以替代宏参数

```
const int A =3;    ==   #define A 3
```

inline 内联函数用来代替宏代码块

编译器直接将函数体扩展到函数调用的地方，内联函数没有普通函数调用的开销（压栈，跳转，返回）

inline只是请求，编译器不一定答应函数的内联请求

```
#define FUNC(a,b) ( (a)<(b)?(a):(b) )
__attribut__((always_inline))   // 强制内联
int func(int a,int b);   // 声明
inline int func(int a,int b)
{
    return a<b?a:b;
}
int a = 1; int b =3;
int c = FUNC(++a,b);
printf("a=%d,b=%d,c=%d",a,b,c);   // a=3,b=3,c=3
```

强制编译器进行内联编译

```
g++:__attribut__((always_inline))
MSVC:__forceinline
     或 inline 标准C++
```

内联编译的限制：（最先进的编译器没有这些限制）

- 不能存在任何形式的循环
- 不能存在过多的条件判断语句
- 函数体不能过于庞大
- 不能对函数进行取地址操作
- 函数内联声明必须在调用语句之前

区别7.1：函数参数的扩展

C++中可以在函数声明时为函数提供一个默认值

```
int mul(int x = 0);   // 函数声明，C中只能写int mul(int x);
int main()
{
    mul();   // == mul(0)
}
int mul(int x)   // 函数定义时不必写出参数默认值
{
    return x * x;
}
```

提供默认参数值只能写在声明中，不然会报错

```
int mul(int x = 5)
{
    return x * x;
}
int main()
{
    mul();   // == mul(0)
}
// OK，因为函数定义int mul(int x = 5) 包含了声明
```

```
int mul(int x = 0);
int main()
{
    mul();   // == mul(0)
}
int mul(int x = 5)
{
    return x * x;
}
// 报错
```

```
int mul(int x);
int main()
{
    mul();   // == mul(0)
}
int mul(int x = 5)
{
    return x * x;
}
// 报错
```

函数默认参数的规则

- 默认值必须从右往左提供
- 函数调用时使用了默认值，则后续参数必须使用默认值

区别7.2：占位参数

C++中可以为函数提供占位参数

- 占位参数只有个函数类型声明，而没有参数名声明
- 一般情况下，在函数内部无法使用占位参数。**占位参数没什么用，只是为了兼容C，因为C没有参数时，表示接受任意参数**

```
int func(int x,int){
    return x;
}
func(1,2);   // OK
```

```
int func()
{

}
func();   // C中OK，C++中OK
func(1,2);   // C中OK，C++中报错
```

void func() 与 void func(void)在C++中等价，而在C中不等价，void func() 接受任意参数

```
int func(int ,int)
{

}
func();   // C++中报错
func(1,2);   // C++中OK
```

占位参数与默认参数结合

```
int func(int =0,int =0)
{

}
func();   // C++中OK
func(1,2);   // C++中OK
```

区别8：函数重载 overload

函数重载至少满足下面的一个条件

- 参数个数不同
- 参数类型不同
- 参数顺序不同

```
int func(int a, const char* s);
int func(const char* s, int a);   // 构成重载
```

当函数重载遇到函数默认参数，C++的第一个冲突

```
int func(int a, int b, int c = 1);
int func(int a, int b, int c)
{
    return a+b+c;
}
int func(int a, int b)
{
    return a+b;
}
int c = func(1,2);   // 报错，因为 ambiguous 两个函数都满足，有二义性，无法匹配
```

编译器调用重载函数的准则：

- 将所有同名的函数作为候选者
- 尝试寻找可行的候选函数
  法1：精确匹配参数
  法2：通过默认参数能够匹配实参
  法3：通过默认类型转换匹配实参
- 匹配失败
  情况1：找到的候选函数不唯一，出现二义性，报错
  情况2：无法匹配所有候选者，报错，函数未定义

函数重载注意：

- 重载函数在**本质上是相互独立的不同函数**
- 重载函数的**函数类型不同**
- 函数重载是**由函数名和参数列表决定**，函数返回值不能作为函数重载的依据

```
int add(int a, int b);   // 函数类型 int(int, int)
int add(int a, int b, int c);   // 函数类型 int(int, int, int)

printf("%p",(int(*)(int, int))add );   // 打印函数入口地址
printf("%p",(int(*)(int, int, int))add );    // 两个函数入口地址不同
```

函数重载8.2

函数指针遇到重载函数

```
int func(int x);
int func(int x,int y);
int func(const char* s);
typedef int(*PFUNC)(int x);
int c = 0;
PFUNC p = func;   // p指向的函数为第一个 int func(int x);
c= p(1);

typedef void(*PFUNC2)(int x);
PFUNC2 p = func;   // ERROR，无法匹配，不仅要参数列表匹配，返回值也要匹配

typedef double(*PFUNC3)(int x);
PFUNC3 p = func;   // ERROR，无法匹配，严格匹配参数列表和返回值
```

将重载函数名赋值给函数指针时：

1. 根据重载规则挑选与**函数指针参数列表一致**的候选者
2. 严格匹配候选者的函数类型和**函数指针的函数类型**（即不仅要参数列表匹配，返回值也要匹配）

注意：

- 函数重载**必须发生在同一个作用域中**
- 重载函数的调用时，编译器需要用**参数列表（一般情况）**或**函数类型（函数指针）**进行函数选择
- **无法直接通过函数名得到重载函数的入口地址**

```
int add(int a, int b);   // 函数类型 int(int, int)
int add(int a, int b, int c);   // 函数类型 int(int, int, int)

printf("%p",( int(*)(int, int))add );   // 打印函数入口地址
printf("%p",add );    // ERROR
```

C++与C的相互调用

- C++编译器完全兼容C的编译方式
- C++编译器会优先使用C++的编译方式
- extern关键字能强制让C++编译器进行C方式的编译

```
extern "C"
{
    // do C-style compilation here
}

// 假设 add.c add.h 是C代码
// 第一步：编译为目标文件 add.o
// gcc -c add.c -o add.o
// 第二步：使用
#include "add.h"
int main()
{
      int c = add(1,2);
      printf("c = %d\n",c);
      return 0;
}
// ERROR ，因为默认使用C++编译器，正确方法如下
extern "C"
{
    #include "add.h"
}
```

```c
int main()
{
        int c = add(1,2);
        printf("c = %d\n",c);
        return 0;
}
```

如何保证一段C代码只会以C的方式被编译

```c
extern "C"   // 这一段代码只在C++中有效，用 gcc 编译会报错
{
    #include "add.h"
}
```

__cplusplus 是C++编译器内置的标准宏定义

意义：确保C代码以统一的C方式被编译成目标文件

```c
#ifdef __cplusplus
extern "C"{    // 如果定义了 __cplusplus  则保留extern "C"{
#endif

// C-style compilation

#ifdef __cplusplus   // 如果定义了 __cplusplus  则保留 }
}
#endif
```

使用gcc和g++都可以编译上述代码

注意：

- C++编译器**不能以C的方式编译重载函数**
- **编译方式**决定函数名被编译后的**目标名**
  C++编译方式将**函数名**和**参数列表**编译成**目标名**，如addii，addiii
  C编译方式只将**函数名**作为目标名进行编译，如add

总结：

函数重载通过函数参数列表区分不同的同名函数

extern关键字能够实现C和C++的相互调用，编译方式决定符号表中的函数名和最终目标名

区别 9：C++中的新成员

9.1 C++中的动态内存分配

- C++通过new 和 delete 或 new[] delete[] 来动态申请和释放内存
- C++动态内存申请是基于类型进行的，C的malloc是基于字节数，free释放（变量和数组都一样的形式）

```
// 变量申请
Type* pointer = new Type;
delete pointer;
// 数组申请
Type* pointer = new Type[N];
delete[] pointer;
delete pointer;   // 只释放了数组第一个元素的空间
```

```
int* pointer = new int[10];   // pointer指向的内存空间至少是 40 个字节
```

new 和 malloc 的区别

- new关键字是C++的一部分，malloc是由**C库提供的函数**
- new以**具体类型为单位**进行内存分配，malloc以**字节为单位**进行内存分配
- new在申请单个类型变量时可进行初始化，malloc不具备内存初始化的特性

new在申请单个类型变量时进行初始化

```
int* pi = new int(1);    // pi 申请成功后立即初始化为 1
float* pf = new int(2.0f);
char* pc = new int('c');

int* pi = new int[1]; // 申请数组空间，大小为1个 int 类型
```

9.2 C++的命名空间

在C中只有一个全局作用域

- C中所有全局标识符共享同一个作用域
- **标识符之间可能发生冲突**

C++中提出了命名空间的概念

- 命名空间**将全局作用域分成不同的部分**

- **不同命名空间中的标识符可以同名，而不会发生冲突**
- **命名空间可以相互嵌套**
- 全局作用域也叫默认命名空间

```
namespace Name   // 命名空间的定义
{
    namespace Internal
    {}
}
// 命名空间的使用
using namespace Name;   // 使用整个命名空间 Name
using Name::variable; // 使用命名空间 Name 中的 variable
::varibale   // 使用默认命名空间中的 variable
```

示例

```
namespace First{ int i = 0; }
namespace Second{ int i = 1;   namespace Internal
{
    struct P{ int x; int y;};
}
}

int main(){
    using namespace First;
    using Second::Internal::P;
    printf("First::i = %d",i);
    printf("Second::i = %d",Second::i);
    P p(2,3);
}
```

总结：

- C++中内置动态内存分配的关键字，可在动态内存分配同时进行初始化，基于类型进行，用于解决名称冲突问题

区别10：新型的类型转换

C强制类型转换，

- 过于粗暴，任何类型之间都可以进行转换，编译器不知道其正确性
- 难以定位，无法快速定位所有使用强制类型转换的语句

```
typedef void(PF)(int);
struct Point{int x; int y;};
int main(){
    int v = 0x12345;
    PF* pf = (PF*)v;   // 0x12345存放的是函数吗？？
    pf(5);
    char c = char(v);
    Point *p = (Point*)v;
}
// 编译可以通过，但是运行会报错
```

强制类型转换是 bug 的重要来源，C++将强制类型转换分为4只能不同类型

static_cast	const_cast
dynamic_cast	reinterpret_cast

用法

```
xxx_cast< Type >(Expression)
```

static_cast 使用范围

- 用于**基本类型**间的转换
- **不能用于基本类型指针**间的转换
- 用于有**继承关系类对象**之间的转换和**类指针**之间的转换

```
int i = 0x12345;
char c = 'c';
int* pi = &i;
char* pc = &c;
c = static_cast<char>(i);   // OK, int→char
pc = static_cast<char*>(pi);   // ERROR , int*→char*
```

const_cast 使用范围

- 用于**去除变量的只读属性**
- 强制转换的**目标类型**必须是**指针或引用**

```
const int& j =1;   // 产生只读变量,可以通过指针修改其值
int& k = const_cast<int&>(j);   // ok

const int x =2;   // 产生真正意义上的常量
int& y = const_cast<int&>(x); // OK,x会在栈中开辟空间，y是这个空间的别名。但是使用x的时候，是从表中取出其值
int z = const_cast<int>(x);   // ERROR
 k = 5;
printf("k=%d, j=%d",k,j);
// k=5, j=5
 y = 8;
printf("x=%d, y=%d, &x = %d, &y = %d",x,y,&x,&y);
// x=2, y=8, &x与&y的地址相同
```

dynamic_cast 使用范围（C中没有）

- 用于**有继承关系的类指针**间的转换
- 用于**有交叉关系的类指针**间的转换
- 具有**类型检查**的功能
- **需要虚函数的支持**

```
int i = 0;
char* pi = &i;
char* pc = dynamic_cast<char*>(pi);   // ERROR,用于类指针，并且需要虚函数
```

reinterpret_cast 使用范围

- 用于**指针类型间**的强制转换
- 用于**整数和指针类型间**的强制转换（嵌入式中）

```
int i = 0;
char c = 'c';
int* pi = &i;
char* pc = &c;
pc = reinterpret_cast<char*>(pi);   // OK  char* → int*
pi = reinterpret_cast<int*>(pc);    // OK  int* → char*
pi = reinterpret_cast<int*>(i);    //  OK  int → int*
c = reinterpret_cast<char>(i);    // ERROR
```

总结：

- **C中的强制类型转换过于粗暴，不易于定位，很难发现潜在的问题**

- **C++新型类型转换以4种关键字的方式出现，编译器能够检查潜在的问题，方便定位，支持动态类型识别（dynamic_cast）**

案例1：const的疑问，何时为只读变量，何时是常量

const常量判别标准

在编译期间不能直接确定初始值的const标识符，都被作为只读变量处理

- 只有用字面量初始化的const常量才会进入符号表
- 使用其它变量初始化的const常量仍然是只读变量
- 被volatile修饰的const常量不会进入符号表

```
const int x= 1;
const int& rx = x;
int& nrx = const_cast<int&>(rx);
nrx = 5;
printf("x= %d,rx =%d, nrx = %d",x,rx,nrx);
// x=1,rx=5,nrx=5，三者地址相同（该地址是）
```

```
volatile const int y =2;    // 只读变量
int* p = const_cast<int*>(&y);
*p = 6;
printf("y = %d, p = %d",y,*p);
// y=6,p=6

const int z = y;   // 只读变量
p = const_cast<int*>(&z);
*p= 7;
printf("z = %d",z);
// z = 7
```

---

const引用的类型与初始化变量的类型

1. 相同：初始化变量成为只读变量
2. 不同：**生成一个新的只读变量**

```
char c = 'c';
char & rc = c;
const int & trc = c;   // 情况2，会生成一个新的只读变量
rc = 'a';
printf("\n c = %c, rc = %c,trc = %c",c,rc,trc);
//  c = a, rc = a,trc = cz
```

指针与引用

在编译器内部，使用指针常量来实现引用

指针是一个变量，

- 值为一个内存地址，不需要初始化，可以保存不同的地址
- 通过指针可以**访问对应内存地址中的值**
- 指针可以被**const修饰成为常量或只读变量**

引用只是一个变量的新名字

- 对引用的操作（赋值，取地址等）都会**传递到代表的变量上**
- const引用使其代表的变**量具有只读属性**
- 引用**必须在定义时初始化**，之后无法代表其他变量

从使用者角度来看：引用和指针没有任何关系

从编译器角度：使用指针常量来实现引用，因此必须在定义时初始化

在过程项目开发中

- 当进行编程时，直接站在使用者角度看待引用和指针
- 进行调试分析时，一些特殊情况，可以站在编译器角度看待引用

```
int a = 1;
struct SV{
int &x;
int &y;
int &z;};
int main()
{
    int b = 2;
    int* pc = new int(3);
    SV sc = {a,b,*pc};   // OK
    int& array[] = {a,b,*pc};   // ERROR，C++中不支持引用数组，因为数组的元素地址需要是类型递增的（C中）
```

```
    printf("&sc.x = %p, &sc.y = %p, &sc.z = %p\n",&sc.x,&sc.y,&sc.z);    //
地址不连续
    delete pc;
    return 0;

}
```

2.3 面向对象

面向对象的意义，是当今软件开发中的重要方法

- 将日常生活中**习惯的思维方式引用程序设计中**
- 将需求中的概念**直观地映射**到解决方案中
- 以**模块为中心**构建可复用的软件系统
- 提高软件产品的**可维护性和可扩展性**

软件灾难：用户需求改变，需要改代码，改代码又引入新的bug

类与对象，是面向对象理论中的基本概念

- 类：**一类事物**，是一个**抽象的概念**
- 对象：属于某个类的**具体实体**
- 类是一种模型，这种模型可以创建出不同的对象实体
- 对象实体是类模型的一个具体实例

一个类可以有很多对象或者0个或1个，而一个对象必然属于某个类

> 类用于**抽象的描述一类事物所特有的属性和行为**
>
> 对象是**具体的事物**，拥有所属类中描述的一切属性和行为

类之间的基本关系

1. 继承
    - 从已存在类细分出来的类和原类之间具有继承关系（is-a）
    - 继承的类拥有父类的所有属性和行为
    - 空心三角箭头表示，子类指向父类
2. 组合

- 一些类的存在**必须依赖于其它类**，称组合，组合关系是类之间整体和部分的关系
- 组合的类**在某一个局部上由其它的类组成**
- 实心菱形箭头表示，如电脑有硬盘、cpu、键盘等类组合而成

2.3.1 类和封装

类的内部实现细节，类的使用方式

类的封装机制使得使用方式和内部细节相分离，通过访问级别实现封装机制

使用类时，不需要关心器实现细节；当创建类时，才需要考虑内部实现细节

类的成员：

成员变量：表示类属性的变量

成员函数：表示类行为的函数

类成员的访问属性：

public：在类内部和外界访问和调用

private:：在类内部被访问和调用

类成员的作用域：

- 类成员的作用域都**只在类的内部**，外部无法直接访问
- 成员函数可直接访问成员变量和调用成员函数
- 类外部可以**通过类变量访问public成员**
- 类成员的**作用域与访问级别没有关系**

C++中 struct 和 class 的用法完全相同，唯一区别：默认访问级别不同，用struct定义的类中成员默认为public

::i全局作用域的i

2.3.2 对象的构造

成员变量的初始值

类相当于一种自定义数据类型，可以在以下位置定义对象

1. 局部类的变量初始化，保存在栈中，初始值随机
2. 全局类的变量初始化，存在全局存储区（静态存储区，其中有全局变量和static局部变量），初始值为0
3. new出来的对象，保存在堆中，初始值随机

构造函数：用于对象的初始化

- 构造函数与类名相同
- 构造函数没有返回值
- 在对象定义时自动被调用

```
class Complex{
public:
    double a;
    double b;

    Complex(double aa=0 ,double bb=0):a(aa),b(bb){}
};
// 使用
Complex c1;   // 会调用 Complex(double aa=0 ,double bb=0)，因为全部参数都有默认值的构造函数也是默认构造函数
```

构造函数可以根据需要定义参数

一个类中可以存在多个重载的构造函数

构造函数的重载遵循C++重载函数的规则

对象定义：申请对象的空间并调用构造函数

对象声明：告诉编译器存在这样一个对象

```
Text t;   // 定义
extern Test t;   // 声明

class Test{
      Test();
      Test(int v)
     {}
}
Test t1(2);    //  调用 Test(int v)
Test t2 = 3;   // 调用 Test(int v)
```

初始化：对**正在创建的对象**进行初值设置

赋值：对**已经存在的对象**进行值设置

```
// 创建对象数组
Test ta[3];          //  默认调用 Test()
// 手动调用构造函数
Test ta[3] = { Test(), Test(1), Test(3)};    // 手动调用

/* 区别  */
// 1.初始化
Test t2 = 3;   //    == Test t2 = Test(3)
// 2.定义并赋值
Test t3;      // 先调用 Test()
t3 = Test(3);    // 再手动调用 Test(int v)
```

```
// 数组的定义
m_pointer = new int[len];
delete[] m_pointer;
```

两个特殊的构造函数：

1. **无参数构造函数**，用于定义对象的默认初始状态
   当类中没有定义构造函数时，编译器默认提供一个无参构造函数，并且该函数体为空
2. **拷贝构造函数**：const class_name&的构造函数，用于创建对象时拷贝对象的状态
   当类中没有定义拷贝构造函数时，编译器默认提供一个拷贝构造函数，简单地进行成员变量地值复制
   Test(const Test& t){ value = t.value; }

拷贝构造函数也是构造函数，意义：兼容C的初始化方式

- 浅拷贝

拷贝后对象的物理状态相同，编译器提供的拷贝构造函数只进行浅拷贝

- 深拷贝

拷贝后对象的逻辑状态相同

```
class Test{
int* p;
      Test(int v){ p = new int; *p = v; }
      free(){delete p;}
}

Test t1(3);
Test t2 = t1;   // 会使用编译器提供的默认拷贝构造函数
printf("t1.p = %p, t2.p = %p",t1.p,t2.p);
// 两者的地址是一样的，但是 delete
t1.free();
t2.free();
// ERROR，不能 delete 同一个地址两次

// 修改如下
class Test{
int* p;
      Test(int v){ p = new int; *p = v; }
      Test(const Test& t){   p = new int; *p = *(t.p); }
      free(){delete p;}
}
Test t1(3);
Test t2 = t1;   // == Test t2(t1);
printf("t1.p = %p, t2.p = %p",t1.p,t2.p);
// 两者的地址是不一样的，但是 *t1.p 与 *t2.p 相等
```

什么时候需要深拷贝

- 对象中有成员指代了系统中的资源

1. 成员指向了动态内存空间

- 成员打开了外存中的文件
- 成员使用了系统中的网络端口

> 一般性原则：如果需要自定义拷贝构造函数，必然需要实现深拷贝

2.3.3 初始化列表

```
class Test{
    int a = 1;        // ERROR，禁止类内初始化
    const int ci;     // 只读变量，可以被修改
    int value;
    public:
      Test():value(0),ci(5)   // 初始化顺序：先ci(5)，后value(0)，初始化顺序
与成员声明顺序相同
     {
           // 初始化列表先于构造函数体执行
     }
      int setCI(int v)
     {
           int* p = const_Cast<int*>(&ci);
           *p = v;
     }
};
```

初始化列表：

- 初始化顺序与成员声明顺序相同
- 初始化列表先于构造函数体执行
- const成员必须在初始化列表中指定初始值

类中可以定义const成员

- 类中const成员会被分配空间（与当前对象所在空间一样）
- 本质是只读变量
- 只能在初始化列表中指定初始值

2.3.4 构造顺序

构造时机

1. 对于局部对象：当程序流到达对象的定义语句时进行构造

```
int func()
{
    Test a1 =1;
    Test a2 =2;
goto End;
    Test a3(3);   // 不会执行，不创建a3
End:
      printf("a3.value = %d",a3.value);   // 有的编译器报错，有的不会报错（输出的值随机）
    return 0;
}
// 因此，不要使用 goto
```

2. 对于堆对象
- 当程序流到达new语句时创建对象
- 使用new创建对象并自动触发构造函数的调用
3. 对于全局对象
- 构造顺序是**不确定**的
- 不同编译器使用不同的规则确定规则顺序

因此，不要使用全局对象；即使要使用，程序中要避开全局对象之间的依赖

构造顺序

单个对象构建时构造函数调用顺序

1. 调用父类的构造函数
2. 调用成员变量的构造函数（调用顺序与说明顺序相同）
3. 调用类自身的构造函数

析构函数对应构造函数的调用顺序相反

多个对象析构时，析构顺序与构造顺序相反

2.3.5 对象的销毁

析构函数：在对象销毁是进行清理的特殊函数

析构函数，功能与构造函数相反，没有参数没有返回值类型声明，在对象销毁时自动被调用

当类中自定义了构造函数，并在其中使用了系统资源（如内存申请，文件打开等），则需要自定义析构函数

对于栈对象和全局对象，类似于入栈和出栈的顺序，最后构造的对象最先被析构

堆对象的析构发生在使用delete的时候，与delete的顺序有关

2.3.6 临时对象

```
class Test{
    int value;
    public:
      Test()
     {
            Test(0);    // 手动调用构造函数，生成一个临时对象 Test(0)
     }
      int Test(int v)
     {
          value = v;
     }
};

Test t1;
printf("t1.value = ",t1.value);    // 输出结果随机
```

- 直接调用构造函数将产生一个临时对象
- 临时对象的生命周期只有一条语句的时间
- 临时对象的作用域只在一条语句中
- 临时对象是性能的瓶颈，也是bug来源之一

C++编译器会尽力减少临时对象的产生，实际开发中要人为的避开临时对象

```
Test t = Test(10);
// 个人理解：1.生成临时对象Test(10)；2.用临时对象初始化t对象（调用拷贝构造函数）
// 实际上并非如此，编译器会尽量减少临时对象的产生，编译器会使得 Test t = 10;
```

```
Test func(){ return Test(20); }

Test tt = func();    // 编译器会尽量减少临时对象的产生，编译器会使得 Test tt = 20;
```

2.3.7 const与对象

const修饰对象

　对象为只读对象，其成员变量不允许被改变，只读对象是编译阶段的概念，运行时无效

const修饰成员函数

- **const对象只能调用const的成员函数**
- **const成员函数只能调用const成员函数**
- const成员函数中不能直接改写成员变量的值

2.3.8 静态成员变量

静态成员变量属于整个类

生命期不依赖于任何对象，而是整个程序运行期

可以通过类名和对象名访问公有静态成员变量

所有对象共享类的静态成员变量

静态成员变量需要在类外单独分配空间，位于全局数据区

2.3.9 静态成员函数

属于整个类

可以通过类名和对象名访问公有静态成员函数

静态成员函数与普通成员函数：

	静态成员函数	**普通成员函数**
所有对象共享	y	y
隐含this指针	**n**	y
访问普通成员变量/函数	**n**	y
访问静态成员变量/函数	y	y
通过类名直接调用	y	**n**
通过对象名来调用	y	y

2.3.10 二阶构造模式

构造函数只能决定对象的初始化状态，初始化操作的失败不影响对象的诞生

初始化不完全会产生半成品对象

初始化操作不能按照预期完成而得到的对象

半成品对象是合法的C++

对象，也是bug的重要来源

工程开发的构造过程，一分为二

第一阶段构造：

1. 资源无关的初始化操作

不可能出现异常情况

第二阶段构造：

2. 需要使用系统资源的操作

可能出现异常情况，如内存申请，访问文件

二阶构造能够确保创建的对象都是完整初始化的

```
class TwoPhaseCons
{
private:
    TwoPhaseCons(){}    // 第一阶段构造函数，初始化成员变量（不使用new）
    bool construct()    // 第二阶段构造函数
    {
            bool ret = true;
            m_pointer = new int;
            if(m_pointer)
            {
                  // 初始值
            }
            else ret = false;
            return ret;
    }
public:
    static TwoPhaseCons* NewInstance()
    {
        TwoPhaseCons* ret = new TwoPhaseCons();
```

```
        if(ret && ret->construct())
        {

        }
        else
        {
            delete ret;
            ret = NULL;
        }
        return ret;
    }
};
// 使用
TwoPhaseCons* obj = TwoPhaseCons::NewInstance();
```

2.3.11 友元

friend

友元关系发生在函数与类之间，类与类之间

右元关系是单向的，不能传递

类的友元可以是其它类或者具体函数或其它类的成员函数

友元不是类的一部分

友元不受访问级别的限制，友元可以直接访问具体类的所有成员

```
class Test
{
int value;
friend int func(Test&);
};
int func(Test& t){ return t.value;}   // 直接访问私有变量，如果不是友元的话需要借助类的接口才可以访问
```

友元是为了**兼容C的高效**而诞生的，但**友元直接破坏了面向对象的封装**，因此，友元被遗弃

2.3.12 函数重载

函数重载的本质为相互独立的不同函数

通过**函数名和参数列表**来确定函数的调用

无法直接通过函数名得到重载函数的入口地址

函数重载必然发生在**同一个作用域**中

类的成员函数重载有三种：

构造函数、普通函数、静态成员函数

静态成员函数可以与普通函数重载

重载意义：通过函数名对函数功能进行提示，通过参数列表对函数的用法进行提示，扩展系统中已经存在的函数功能

```
const char* str= "D.T. software"
char* buff[3] = {0};
strcpy(buff,str);     // ERROR
strncpy(buff,str,sizeof(buff)-1); // OK，一样的功能却需要记住两个函数名
```

2.3.13 操作符重载

重载能够扩展操作符的功能，用特殊形式的函数扩展操作符的功能，**操作符重载的本质为函数定义**

全局函数和成员函数都可以实现操作符重载，编译器优先在成员函数中寻找操作符重载函数

```
1.使用成员函数重载操作符
class Complex{
    int pos;
    int neg;
    public:
    Complex operator + (const Complex& c2)
    {
        Complex temp;
        temp.pos = this.pos + c2.pos;
        temp.neg = this.neg + c2.neg;
        return temp;
    }
};
// 使用
Complex c1 = {1,5};Complex c2 = {2,4};
Complex c3 = c1+c2;   // == Complex c3 = c1.operator+(c2);
```

```
// 2.全局函数也可以实现操作符重载
friend Complex operator+(const Complex& c1,const Complex& c2);
Complex operator+(const Complex& c1,const Complex& c2)
{
    Complex temp;
    temp.pos = c1.pos + c2.pos;
    temp.neg = c1.neg + c2.neg;
    return temp;
}
Complex c3 = c1+c2;   // == Complex c3 = operator+(c1,c2);
// 成员函数Complex operator+(const Complex& c2)和全局函数friend Complex
operator+(const Complex& c1,const Complex& c2)都存在时，优先调用成员函数里面
的
```

C++规定赋值操作符=只能重载为成员函数

操作符重载不能改变原操作符的优先级，不能改变操作数的个数，不能改变操作符的原有语义

运算：+ - * /

比较：== !=

赋值：=

求模：modulus

```
Complex& operator = (const Complex& c2)
{
    if(this != &c2)
     {
           a=c2.a;b=c2.b;
     }
      return *this;
}
bool operator == (const Complex& c2)
{
    return (a==c1.a)&&(b = c2.b);
}
bool operator != (const Complex& c2)
{
    return (*this == c2);
}
// 使用
Complex c1,c2,c3;
```

```
(c3 = c2) = c1;   // OK ==  c3=c1
```

编译器为每个类默认重载了赋值操作符，仅完成浅拷贝

当需要进行深拷贝时必须重载赋值操作符

赋值操作符与拷贝构造函数有相同的存在意义

只要类中有成员使用了系统资源，则需要重载赋值操作符(也需要定义拷贝构造函数)，来实现深拷贝

注意：

```
Complex& operator = (const Complex& obj)
{
    if(this != &obj)
    return *this
}
```

空的类中有什么

```
class A{};
// 实际上类里面有的东西
public:
    A();
    A(cosnt A&);
    A& operator = (cosnt A &);
    ~A();
```

2.4 STL标准库

C++标准库不是C++的一部分，

C++标准库是由**类库**和**函数库**组成的集合

C++标准库中定义的类和对象都位于 **std** 中

C++标准库的头文件都不带.h

C++标准库**涵盖了C库的功能**

标准库中包含**经典算法**和**数据结构**

```
// 重载 <<
class Test
{
public:
    Test& operator<<(int i)
    {
        printf("%d",i);
         return *this;
    }
};
// 使用
Test t1;
t1<<3;   //  == t1.operator<<(3);
t1<<3<<2<<1;   // OK
```

C++编译环境包含：

编译模块：C++标准语法模块、C++扩展语法模块（每个厂商开发的各异）

C++标准库、C语言兼容库（厂商提供）、编译器扩展库（厂商提供）

```
// C++中
#include "stdio.h"   // 是C语言兼容库
#include <cstdio>   // 是C++标准库
#include <cstdlib>
#include <cmath>
using namespace std;
```

C++标准库有 cmath

2.4.1 字符串

C不支持真正意义上字符串，而是用字符数组和一组函数实现字符串操作，不支持自定义类型，因此无法获得字符串类型

C++中可以通过类完成字符串类型的定义，没有原生的字符串类型

C++标准库提供string类型，直接支持字符串连接、字符串大小比较、子串查找和提取、字符串的插入和替换

```
#include <string>
void stringSort(string a[],int len)
{
    for(int i = 0;i<len; ++i)
    {
```

```
            for(int j =i;j<len;++j)
            {
                if(a[i]>a[j])
                {
                    swap(a[i],a[j]);
                }
            }
        }
    }
```

字符串与数字的转换

sstream字符串流类，用于string的转换

```
// string → 数字
istringstream iss("3.14");
double num;
iss>>num;    // 返回值是bool

// 数字 → string
ostringstream oss;
oss<<543.21;   // 返回值是oss本身
string s = oss.str();
```

string → 数字

```
bool to_number(const string&,int& n)
{
    istringstream iss(s);
    return iss>>n;
}
int n;
to_number("234",n);

// 使用宏可以适用于各种类型数据
#define TO_NUMBER(s,n) (istringstream(s)>>n)
```

数字 → string

```
string to_string(int n)
{
    ostringstream oss;
    oss<<n
    rturn oss.str();
}
// 使用宏可以适用于各种类型数据
#define TO_STRING(n) (((ostringstream &)(ostringstream()<<n)).str())
string s = TO_STRING(123);
```

面试题：字符串循环右移

```
// 如 abcdefg 移位3  efgabcd
string rightFunc(const string& s,unsigned int n)
{
    string ret = "";
    unsigned int pos;
    n = n%s.length();
    pos = s.length() - n;
    ret = s.substr(pos);   // 提取出 efg
    ret += s.substr(0,pos);
    return ret;
}
string s = rightFunc(str,3);
//  第二种方法
string operator >> (const string& s,unsigned int n)
{
    string ret = "";
    unsigned int pos;
    n = n%s.length();
    pos = s.length() - n;
    ret = s.substr(pos);   // 提取出 efg
    ret += s.substr(0,pos);
    return ret;
}
string s = str>>3;
```

---

string不要与C风格的char *混用

```
stting s = "123";
const char* p = s.c_str;   // p指向了s的首地址
s.append("abc");   // 执行完之后，原来的存储空间会被删除，"123abc"会放在新开辟的空间中，名称为s
cout<<p;          // p 成为了野指针
```

```
const char* p = "hello";
string s = "";
s.reserve(10);
for(int i =0 ;i <6;++i)
{
    s[i] = p[i];
}
if(!s.empty())   // s为空，但是 s[0]='h',s[1]='e',s[2]='l',s[3]='l',s[4]='o',因为s.length()没有改变，还是0，字符串为空，bug
{
    cout<<s;
}
// 以下OK
string p = "hello";
string s = "";
s = p;
```

2.4.2 []操作符重载

```
string str= "a1b2c3d4e5";
int count = 0;
for(int i=0;i<str.length();++i)
{
    if(isdigit(str[i]))
    {
        ++count;
    }
}
cout<<"cnt = "<<count;
```

str[i]，类的对象怎么支持 [] 访问

[] 数组访问操作符是C/C++种的内置操作符，原生意义是**数组访问和指针运算**，计算数组的偏移量

> a[n] == *(a+n) == *(n+a) == n[a]

```
int a[5] = {0};
for(int i = 0; i < 5; ++i)
{
    i[a] = i+10;   // == a[i] = i+10
}
```

[] 只能通过类的成员函数重载，重载函数能且只能使用一个参数，可以定义不同参数的多个重载函数

```
class Test{
      int a[5];
public:
    int& operator [](int i)
    { return a[i]; }
    int& operator [](const string& s)
    { if(s=="1st") return a[0];
      else return a[1];  }

};
Test t;
t[1] = 5;   // t.operator[1] = 5;  返回引用作为左值
cout<<t["1st"];
```

2.4.3 函数对象

需求：一个函数，每调用一次返回一个值

```
int gib()
{
    static a0 = 0;
    static a1 = 1;
    int ret = a1;
    a0=a1;
    a1 = a0+a1;
    return a0;
}   // 缺陷：函数无法重置
```

函数对象，常用于**取代函数指针**

- **使用具体的类对象取代函数**

- 该类的对象**具备函数调用的行为**
- **构造函数指定具体数列项的起始位置**
- 多个对象相互独立的求解数列项

函数调用操作符 ()，通过类的成员函数重载，可以定义不同参数的多个重载函数

```
class Fib
{
    int a0 ;
    int a1 ;
public:
    Fib(){a0=0; a1=1;}
    Fib(int n)
    {
        a0=0;a1=1;
        for(int i = 2;i<=n;++i)
        {
            int t = a1;
            a0= a1;
            a1 = a0+a1;
        }
    }
    int operator () ()
    {
        int ret = a1;
        a0=a1;
        a1 = a0+a1;
        return a0;
    }
};
Fib f1;
Fib f2(10);   // 从第十项开始
```

2.4.4 智能指针

内存泄漏

- 动态申请堆空间，用完不归还
- C++中没有垃圾回收机制，C#中有
- 指针无法控制所指堆空间的生命周期

指针生命周期结束时主动释放堆空间

一片堆空间最多只能有一个指针标识

杜绝指针运算和指针比较

智能指针的本质是对象：

重载了 -> 和 * 操作符

> 智能指针的使用规则：只能用来指向堆空间中的对象和变量

2.5 特殊的重载表达式

2.5.1 逻辑操作符重载

逻辑操作符可以重载，重载后无法完全实现原生的语义，不满足短路法则

```
class Test{
public:
int value;
Test(int v):value(v){}
int getvalue() const{ return value; }
};
bool operator && (const Test& t1,const Test& t2)
{
     return (t1.getvalue() && t2.getvalue());
}
Test func(Test i)
{
      cout << "Test func(Test i) : i.value() = " << i.getvalue() << endl;

      return i;
}


Test t1(0);
Test t2(2);
cout<< func(t1)&&func(t2);   // func函数会被调用两次
```

原因：操作符重载的本质是函数调用，因此，使用重载操作符，进入函数体之前必须完成所有参数的计算（不会短路），而参数的求值顺序是随机的

实际工程中**避免重载逻辑操作符**

通过**重载比较操作符**替代逻辑操作符重载

直接使**用全局成员函数**代替逻辑操作符重载

使用全局函数对逻辑操作符进行重载

2.5.2 ， 重载

从左向右的顺序计算每个子表达式的值，表达式最终的值为最后一个子表达式的值，前N-1个表达式可以没有返回值

```
(i,j) = 10; // == j =10
```

可以重载，操作符

参数必须有一个是类类型，返回值类型必须为引用

```
Test& operator, (const Test& a, const Test&b)
{
    return const_cast<Test&>(b)
}
Test func(Test &i){ cout<<i.value<<endl; return i;}
Test t3 = (func(t1),func(t2));    // 会先执行 func(t2)，再执行 func(t1)，最终 t3 = t2
// == operator , (func(t1),operator(t2))
```

原因：操作符重载的本质是函数调用，进入函数体前必须完成所有参数的计算，函数参数的计算顺序是随机的，重载后无法保证从左向右计算表达式

在工程中，**绝对不要重载，表达式**

2.5.3 ++重载

全局函数和成员函数都可以对其重载

重载前置++不需要额外的参数

重载后缀++需要一个int类型的占位参数

2.5.4 类型转换函数

标准数据类型之间会进行隐式的类型安全转换

```
小类型 → 大类型进行隐式转换是安全的
char 或 char → int   →   unsigned int →   long   → unsigned long →
  float   → double
```

运算时，char和short会先转换为int

unsigned int 和int运算，int会先转换为unsigned int

普通类型与类类型不能隐式转换，除非有转换构造函数

转换构造函数：

有且仅有一个参数，参数是基本类型，参数是其它类型

```
class Test{
public :
Test(){}
Test(int){}
Test operator +(const Test& p)
{
      Test ret(value+p.value);
      return ret;
}

};
Test t1;
t1 = 5; //  == t1 = Test(5)调用的是Test(int){}
Test t2 = t1+10;   // 可以通过，结果为15，因为编译器进行了隐式转换
```

编译器为了让编译通过，会隐式类型转换，隐式类型转换是工程中bug的重要来源

explicit 杜绝编译器的转换尝试

强制类型转换写法：

```
static_cast<ClassName>(value)    // C++风格
ClassName(value)
(ClassName)value   // C风格，不推荐
```

类类型不能转换为基本类型

```
Test t1 = 1;
int t = (int)t1;   // ERROR
```

类中可以定义类型转换函数

```
class Test{

operator int(){
    return value;
}

};
Test t1 = 1;
int t = t1;   // OK ==  int t = t1.operator int()
```

类转换为另一个类

类型转换函数

```
class Value{};
class Test{

operator int(){
    return value;
}

operator Value(){
    Value ret;
    return ret;
}
};

Test t1 = 1;
Value v1 = t1; // OK  == t1.opertor Value()
```

转换构造函数

```
class Value{
public:
Value(Test&){}
};

class Test{
public:
operator int(){
    return value;
}
```

```
};

Test t1 = 1;
Value v1 = t1; // OK == Value v1 = (Value)t1
```

两个都存在

```
class Value{
public:
Value(Test&){}
};

class Test{
public:
operator int(){
    return value;
}
operator Value(){
    Value ret;
    return ret;
}
};

Test t1 = 1;
Value v1 = t1;   // ERROR，二义性
// 解决方法，使用 explicit Value(Test&){}
```

类型转换函数可能与转换构造函数冲突，工程中一般**不会使用类型转换函数**，而是使用普通成员函数来代替 toValue，如Qt中的toInt

```
class Test{
public:
operator int(){
    return value;
}
operator toValue(){
    Value ret;
    return ret;
}
};
```

```
QString str = "-255";
int i    =str.toInt();    // -255
double d =str.toDouble();    // -255
short s = str.toShort();    // -255 全部正确
```

2.6 继承的概念和意义

组合关系：整体与部分的关系

- 将其它类的对象作为当前类的成员使用
- 当前类的对象与成员对象的**生命周期相同**
- 成员对象在用法上与普通对象完全一致

继承关系：类之间的父子关系

继承是**代码复用**的重要手段，通过继承，可以获得父类的所有功能，并且子类可以**重写已有功能，添加新功能**

子类拥有父类**所有属性和行为**

子类就是一种特殊的父类

子类对象可以当作父类对象使用

子类中可以添加父类没有的方法和属性

子类对象可以直接初始化父类，子类对象可以赋值给父类对象

2.6.1 继承中的访问级别

protected修饰的成员不能被外界直接访问，可以被子类直接访问

是专门为了继承而设计的

2.6.2 不同继承方式

3中不同的继承方式，继承方式影响父类成员在子类中的访问属性

public继承

父类成员在子类中保持原有访问级别

private继承

父类成员在子类中变为私有成员

protected继承

父类中的公有成员变为保护成员，其它成员保存不变

	public	protected	private
public继承	不变	不变	不变
protected继承	protected	不变	不变
private继承	private	private	不变

> 继承成员的访问属性 = Max{ 继承方式， 父类成员访问属性}

C++的默认继承为private

```
class Child : Parent   // 默认为 private
```

工程项目中只使用 public 继承

protected和private继承带来的复杂性远远大于实用性

C++派生语言，D、C# 和 Jave 语言中只有 public继承（Jave中用extends代替public）

2.6.3 继承中的构造和析构

子类构造函数

必须对继承而来的成员进行初始化

- 直接通过初始化列表或者赋值进行
- 调用父类构造函数进行初始化

父类构造函数在子类中的调用方式

1. 默认调用（子类创建对象时自动调用）：适用于无参构造函数和使用默认参数的构造函数
2. 显式调用：通过**初始化列表**进行调用，适用于所有父类构造函数

```
class Child : public Parent
{
    public:
        Child(){}    //  默认调用
        Child(string):Parent("para"){}    // 显式调用
};
```

构造规则

- 子类对象在创建时会先调用父类的构造函数
- **先执行父类构造函数**在执行子类的
- 父类构造函数可以被**隐式调用**或**显式调用**

对象创建时构造顺序：

1. 调用父类的构造函数
2. 调用成员变量的构造函数
3. 调用类自身的构造函数

> 先父母，后客人，再自己

析构函数的调用顺序与构造函数相反

1. 执行本身的析构函数
2. 执行成员变量的析构
3. 执行父类的析构

2.6.4 父子间的冲突

同名成员变量：子类中可以定义父类中的同名成员

- 子类中的成员将隐藏父类中的同名成员
- 父类中的同名成员仍然存在于子类中
- 通过**作用域分辨符 ::** 访问父类中的同名成员

```
Child c;
c.mi;   // 访问子类中的mi
c.Parent::mi;   // 访问父类中的mi
```

同名成员函数：子类中可以定义父类中的同名成员函数（可以完全一样）

- 子类中的同名函数会覆盖父类中的同名函数
- 不会发生函数重载，因为作用域不同
- 通过作用域分辨符访问父类中的同名函数

```
class Parent{
public:
int mi;
void add(int c){mi+=c;};   // 操作的父类的mi
};
class Child : public Parent
{
public:
int mi;
void add(int c){mi+=c;};              // 操作的子类的mi
void add(int a,int b){mi=mi+a+b;};    // 操作的子类的mi
};

//
Child c;
c.Parent::add(2);   // 调用父类的
c.add(1);          // 调用子类的
c.add(1,3);         // 调用子类的
```

2.6.5 同名覆盖引发的问题

子类对象可以当作父类对象来使用（兼容性）

- 子类对象可以直接赋值给父类对象
- 子类对象可以直接初始化父类对象
- **父类指针**可以直接**指向子类对象**
- **父类引用**可以直接**引用子类对象**

```
class Parent{
public:
int mi;
```

```
void add(int i){mi+=i;}
void add(int a,int b){mi = mi+a+b;}
};
class Child : public Parent
{
public:
int mv;
void add(int a,int b,int c){mv=mv+a+b+c;}
};

// 第一种兼容性
Child c;
Parent p;
Parent p1(c);   // OK
p = c;   // OK
// 第二种兼容性
Parent& rp = c;
Parent* pp = &c;
rp.mi = 100;
rp.add(5);      // 没有发生同名覆盖
rp.add(1,3);   // 没有发生同名覆盖

pp->mv = 1000;    // ERROR
pp->add(5,5,5);   // ERROR
```

当使用父类指针（引用）指向子类对象时

- 子类对象**退化为父类对象**
- **只能访问父类中定义的成员**
- 可以**直接访问被子类覆盖的同名成员**

特殊的同名函数：

子类中可以重定义父类中已有的成员函数

这种重写发生在继承中，叫**函数重写**

函数重写是同名覆盖的一种特殊情况

```
class Parent{
public:
int mi;
void print(){cout<<"I am parent.";}
};
class Child : public Parent
```

```
{
public:
int mv;
void print(){cout<<"I am child.";}
};

void howToPrint(Parent* p){p->print();}

Child c;
Parent p;
howToPrint(&c);    // "I am parent."
howToPrint(&p);   // "I am parent."
```

void howToPrint(Parent* p){p->print();} 编译期间，编译器只能根据指针类型来判断所指对象

为了安全起见，编译器会调用 Parent中的 print 函数

2.6.6 多态

子类中重写的函数将覆盖父类中的函数，通过 :: 可以访问到父类中的函数

父类的指针指向父类对象，该指针会调用父类中的函数

父类的引用是引用了子类对象，该引用会调用父类中的函数

这与期望不符

实际期望的是，根据实际对象类型判断如何调用重写函数

父类的指针指向父类对象，该指针会调用父类中的函数

父类的引用是引用了子类对象，该引用会调用子类中定义的重写函数

面向对象的多态

- 根据实际的**对象类型决定函数调用**的具体目标
- 同样的调用语句在实际运行时有**多种不同的表现形态**

```
p->print();   // 一条语句展现多种不同的表现形态
// 情况一：p指向父类对象，则调用父类中定义的 print
// 情况二：p指向子类对象，则调用子类中重写的 print
```

virtual 虚函数，是C++支持多态的唯一方式，**virtual** 申明的函数被重写后具有多态特性

```
class Parent{
public:
int mi;
virtual void print(){cout<<"I am parent.";}
};
class Child : public Parent
{
public:
int mv;
void print(){cout<<"I am child.";}   // 没有写virtual,但是他就是虚函数，因为这里virtual可以省略
};

void howToPrint(Parent* p){p->print();}

Child c;
Parent p;
howToPrint(&c);    // "I am child."
howToPrint(&p);   // "I am parent."
```

多态的意义

- 在**程序运行过程中**展现出**多态的特性**
- **函数重写必须多态实现**，否则没有意义
- 多态是**面向对象组件化程序设计**的基础特性

静态联编：在程序的**编译期间就能确定具体的函数调用**，如函数重载

动态联编：在程序**实际运行后才能确定**具体的函数调用，如函数重写（即多态）

即使使用了虚函数，在子类中重写了父类的函数，但是仍然可以通过 c.Parent::print(); 来访问父类中的成员函数

2.6.7 C++对象模型分析

访问权限关键字在运行时失效

在内存中，class仍可看成变量的集合

class 与 struct 遵循相同的内存对齐规则

class中的成员函数和成员变量是**分开存放**的

- 每个对象有独立的成员变量

- 所有对象共享类中的成员函数

```
struct structA{    // sizeof(structA) = 24
int i;   // 4
int j;   // 4
char c; // 8
double d;   // 8
};

class classA{   // sizeof(classA) = 24
int i;   // 4
int j;   // 4
char c; // 8
double d;   // 8
void print(){cout<<"i="<<i<<"j="<<j;}   // 定义成员函数不影响类的size
};


classA ca;
structA* sa = reinterpret_cast<structA*>(&ca);
sa->i = 1;
sa->j = 2;
ca.print();   // i=1, j=2    // 访问权限在运行时失效
```

类的成员函数位于代码段中

- 调用成员函数时，**对象地址作为参数隐式传递**
- 成员函数**通过对象地址访问成员变量**
- **C++隐藏了对象地址的传递过程**

子类由父类成员叠加子类新成员得到，子类中成员变量的存放方式

```
class Parent{
int a;
int b;
};
class Child:public Parent
{
    int c;
};

struct Test{    // Child 的变量存放与 Test的变量一样
int a;
int b;
```

```
int c;
};

Child d;
Test * p = reinterpret_cast<Test*>(&p);
p->a = 10;
```

C++多态的实现原理：虚函数表

- 类中声明虚函数时，编译器会在类中生成一个虚函数表
- **虚函数表是一个存储成员函数地址的数据结构**
- 虚函数表是由编译器自动生成与维护的
- **virtual**成员函数会被编译器**放入虚函数表中**
- **存在虚函数时**，每个对象都有一个**指向虚函数表的指针**

虚函数的效率 < 普通成员函数

```
class Parent{      // size = 8
    int a;
    int b;
};
class Parent   // size = 16，最开始位置是一个指针（8字节），后面依次为 a，b
{
    int a;
    int b;
    virtual void print(){}
};
```

原理演示：

```
// 1. 虚函数表结构
struct VTable{
    int (*pPrint)();   // print 函数的指针

};


class Parent{
    // 2. 虚函数表指针
    struct VTable* vptr = &g_vtal;    // 5.初始化成员指针

    int a;
    int b;
```

```
    virtual void print(){}
};

// 3. 真正意义的虚函数的具体实现
static void g_print(){
      return this->vptr->pPrint();
}
// 4.初始化指针
static struct VTable g_vtal = { g_print };
```

2.6.8 C++的接口类和抽象

抽象类

- 可用于表示现实世界中的**抽象概念**
- 是一种只能**定义类型**，而不能产生对象的类
- **只能被继承**并重写相关函数
- 直接特征是**相关函数没有完整的实现**

C++中没有抽象类的概念

- **通过纯虚函数实现抽象类**
- 纯虚函数是指**只定义原型**的成员函数
- 一个类中**存在纯虚函数**就成为了抽象类
- 一个类中**只存在纯虚函数时成为接口**，接口是一种特殊的抽象类

```
class  Shape{
public:
    virtual double area() = 0;   //  = 0; 表示声明纯虚函数，因此不需要定义函数体
}

class Rect:public Shape{
public:
    int ma,mb;
    double area(){
        return ma*mb;
     }
};

class Circle:public Shape{
public:
```

```
    int r;
    double area(){
        return 3.14*r*r;
     }
};

void area(Shape*)   // OK，虽然没有Shape抽象类的对象，但是可以有指针，因为该指针只能指向抽象类的子类
{
      p->area();   // 可以实现多态
}
```

抽象类**只能作父类被继承**

子类必须**实现纯虚函数**的具体功能

纯虚函数被实现后**成为虚函数**（可以实现多态）

如果子类没有实现纯虚函数，则子类成为抽象类

接口是一种**特殊的抽象类**

- 类中没有定义任何的成员变量
- 所有的**成员函数都是公有的**
- 所有的**成员函数都是纯虚函数**

```
class Channel{   // 接口
public :
    virtual bool open() = 0;
    virtual bool close() = 0;
    virtual bool send(char* buf, int len) = 0;
    virtual bool receive(char* buf, int len) = 0;
};
```

2.7 泛型编程

不考虑具体数据类型的编程方式，用 T 泛指任意的数据类型

2.7.1 函数模板

函数模板是泛型编程的应用方式之一，提高代码复用

交换变量，如宏，函数

```
#define SWAP(t,a,b)     \
      do                \
{     t c = a;          \
      a = b;            \
      b = c;            \
}while(0)              \

int aa=1,bb=2;
SWAP(int,aa,bb);    // 只做文本替换，因此要避免重名
```

函数模板：一种特殊的函数可用不同类型进行调用，看起来和普通函数类似，区别是类型可被参数化

template 关键字用于声明开始进行泛型编程

```
template<typename T>
void Swap(T& a, T& b)
{}

/*  使用  */
Swap(a,b);    // 自动推导
Swap<float>(a,b);    // 显式调用
```

编译器从函数模板通过具体类型**产生不同的函数**

编译器会对函数模板进行**两次编译**

1. 对模板代码本身编译
2. 对参数替代后的代码进行编译

函数模板本身不允许隐式类型转换

- 自动推导类型时，必须严格匹配
- 显示类型指定时，能够进行隐式类型转换

```
typedef void(func1)(int&,int&);
typedef void(func2)(double&,double&);

func1* p1 = Swap;   // 自动推导 T 为 int
func2* p2 = Swap;   // 自动推导 T 为 double

cout<<"p1"<<reinterpret_cast<void*>(p1)<<"p2"<<reinterpret_cast<void*>(p2)
<<endl;   // 地址值不一样
```

多参数函数模板：函数模板可定义多个不同的类型参数

```
template <typename T1, typename T2, typename T3>
T1 add(T2 a, T3 b){ return static_cast<T1>(a+b); }

int r = add<int, float, double>(0.2,0.5);
```

对于多参函数模板，返回值是泛指类型时（必须显式指定），**工程中将返回值类型作为第一个类型参数**

- 无法自动推导返回值类型
- 可**从左向右**部分指定类型参数

```
int r1 = add<int>(0.2,0.5);                    // T1 = int, T2 = double,
T3 = double
double r2 = add<double, float>(0.2,0.5);        // T1 = double, T2 =
float, T3 = double
double r3 = add<double, float, float>(0.2,0.5);   // T1 = double, T2 =
float, T3 = float
```

函数重载遇上函数模板

函数模板可以像普通函数一样被重载

- C++编译器**优先考虑普通函数**
- 如果模板函数可产生一个**更好的匹配**，那么选择模板
- 可通过**空模板实参列表**限定编译器只匹配模板

```
int r1 = Max(1,2);
double r2 = Max<>(0.5,0.8);    // 限定编译器只匹配函数模板
```

2.7.2 类模板

概念与意义：**将泛型思想应用于类**（如STL中使用函数模板和类模板）

C++将模板的思想应用于类，使得类的实现不关注数据元素的具体类型，而只关注类所需实现的功能

- 一些类主要用于**存储和组织数据元素**，如数据结构
- 类中数据组织的方式和数据元素的**具体类型无关**
- 如：数组类，链表类，Stack类，Queue类

类模板

- 以相同的方式处理不同的类型
- 在类声明前使用 **template** 标识
- 《template T》 用于说明类中使用的泛指类型 T

类模板的应用

- 只能**显式指定具体类型**，无法自动推导
- 使用具体类型 定义对象

```
template <typename T>
class Operator{ public : T op(T a, T b); };

Operator<int> op1;
Operator<string> op2;
int i = op1.op(1,2);
```

声明的泛指类型T可以出现在类模板的任意地方

编译器对类模板的处理方式和函数模板相同，需要两次编译

- 从类模板通过具体类型产生不同的类
- 在声明的地方对类模板代码本身进行编译
- 在使用的地方对参数替换后的代码进行编译

```
template<typename T>
class Operator
{
public:
      T add(T a, T b){ return a+b; }
```

```
        T minus(T a, T b) { return a-b; }
        T multiply(T a, T b) { return a*b; }
};

string operator-(string& l, string& r)    // 重写 string类的 - 操作
{
        return "hello";
}

Operator<string> Op2;
cout<<"OP2"<<Op2.minus("gggggggab","ab")<<endl;
```

类模板的工程应用

- 类模板必须在**头文件中定义**
- 类模板**不能分开实现在不同文件中**
- 类模板外部定义的成员函数需要加上模板<>说明

```
//  .h中

template<typename T>
class Operator
{
public:
        T add(T a, T b);
        T minus(T a, T b);
        T multiply(T a, T b);
};

template<typename T>
T Operator<T>::add(T a, T b){ return a+b; }
T Operator<T>::minus(T a, T b) { return a-b; }
T Operator<T>::multiply(T a, T b) { return a*b; }

//  使用
Operator<int> Op1;
cout<<"Op1,"<<Op1.add(22,33)<<endl;
```

类模板可以定义任意多个不同的类型参数

```
template <typename T1, typename T2>
class Test{ public: void add(T1 a, T2 b); };

//
Test<int, float> t;
```

类模板可以被特化

- 指定类模板的**特定实现**
- 部分类型参数必须显式指定
- 根据类型参数**分开实现类模板**

```
template <typename T1, typename T2>
class Test{ public: void add(T1 a, T2 b); };
// 部分特化 →
template <typename T>
class Test<T,T>   // 当Test类模板的两个类型参数完全相同时，使用这个实现
{};
```

特化类型

1. 部分特化：用特定规则约束类型参数
2. 完全特化：完全显式指定类型参数

```
template <typename T1, typename T2>
class Test{ public: void add(T1 a, T2 b); };
// 完全特化 →
template < >
class Test<int ,int>{};
```

类模板特化注意事项

- 特化只是**模板的分开实现**，本质上是同一个类模板
- 特化类模板的**使用方式是统一的**，必须显式指定每一个类型参数

类模板的特化 与 重定义的区别，优先使用特化

重定义

- 一个类模板和一个新类（或者两个类模板）
- 使用的时候需要**考虑如何选择的问题**

特化

- 以统一的方式使用类模板和特化类
- 编译器**自动优先选择特化类**

函数模板可以特化吗

函数模板**只支持类型参数完全特化**

```
template < typename T>
bool Equal(T a, T b) {}   // 函数模板定义

template <>
bool Equal<void*>(void* a, void* b)   // 函数模板完全特化
{}
```

总结：当需要重载函数模板时，**优先考虑使用模板特化**

当模板特化无法满足需求，再使用函数重载

- 类模板可以定义**任意多个不同的类型参数**
- 类模板可以被**部分特化**和**完全特化**
- 特化的本质是**模板的分开实现**
- 函数模板只支持完全特化
- 工程中使用**模板特化替代类（函数）重定义**

2.7.3 数组类模板

模板参数可以是数值型参数（非类型参数）

```
template <typename T, int N>
void func()
{
    T a[N];   // 使用模板参数定义局部数组
}

func<double, 10>();
```

数值型模板参数的限制

变量、**浮点数**、**类对象** 不能作为模板参数

模板参数是**在编译阶段**被处理的单元，因此，在编译阶段必须准确无误地唯一确定

```
template <typename T, int N>    // template <typename T, double N> ERROR
void func()
{
      T a[N] = {0};
      for(int i = 0; i<N; ++i)
      {
            a[i] = i;
      }
}


func<int, 10>();
int a =10;
func<int, a>();   // ERROR
```

```
// 最高效求 1+2+...+N
template <int N>
class Sum
{
public:
    static const int Value = Sum<N-1>::Value+N;
};

template <>
class Sum<1>
{
public:
    static const int Value = 1;
};

cout<<"1+2+...+100="<<Sum<100>::Value<<endl;   // 在编译期间就已经计算出来了
```

数组模板类

```
// .h
template <typename T, int N>
class Array
{
      T m_array[N];
      public:
      int length();
    bool set(int index, T value);
```

```
        bool get(int index, T& value);
        T& operator[] (int index);
        T operator[] (int index) const;
        virtual ~Array();
};

template <typename T, int N>
int Array<T, N>::length()
{
        return N;
}

template <typename T, int N>
bool Array<T, N>::set(int index, T value)
{
        bool ret = (0<= index)&& (index<N);
        if(ret)
        {
                m_array[index] = value;
        }
        return ret;
}

template <typename T, int N>
bool Array<T, N>::get(int index, T& value)
{
        bool ret = (0<= index)&& (index<N);
        if(ret)
        {
                value = m_array[index];
        }
        return ret;
}

template <typename T, int N>
T& Array<T, N>::operator[] (int index)
{
        return m_array[index];
}

template <typename T, int N>
T Array<T, N>::operator[] (int index) const
{
        return m_array[index];
}

template <typename T, int N>
```

```
Array<T, N>:: ~Array()
{

}


// .c 中使用

Array<double, 5> ad;
for(int i = 0 ; i<ad.length(); ++i)
{
      ad[i] = i*i;
}
for(int i = 0 ; i<ad.length(); ++i)
{
      cout<<ad[i]<<endl;
}
```

总结

- 模板参数可以是**数值型参数**
- 数值型参数必须在**编译期间唯一确定**
- 数组类模板是基于数值型模板参数实现地
  数组类模板是简易的**线性表数据结构**

2.7.4 智能指针类模板

智能指针的意义

- 现代C++开发库中**最重要的类模板之一**
- C++中**自动内存管理**的主要手段
- 能够在很大程度上避免内存相关的问题

STL中的智能指针auto_ptr

- 生命周期结束时，销毁指向的内存空间
- 不能指向堆数组，只能指向堆对象（变量）
- 一片堆空间只属于一个智能指针对象
- 多个智能指针对象不能指向同一片堆空间

STL中还有的智能指针：

shared_ptr：带有**引用计数**机制，支持多个指针对象指向同一片内存

weak_ptr：配合shared_ptr而引入的一种智能指针

unique _ptr：一个指针对象指向一片内存空间，不能拷贝构造和赋值（即不能转移所有权）

Qt中的智能指针：

1. QPointer：
   当其指向的对象被销毁时，会自动置空
   析构时不会自动销毁所指向的对象
2. QSharedPointer
   引用计数型智能指针
   可以被自由地拷贝和赋值
   当引用计数为0时才删除指向的对象

其它智能指针：

QWeakPointer，QScopedPointer，QScopedArrayPointer，QSharedDataPointer，QExplicitlySharedDataPointer

---

总结：

智能指针C++中自动内存管理的主要手段

指针在各种平台上都有不同的表现形式

智能指针能够尽可能地避开内存相关问题

2.7.5 单例类模板

单例模式某些类最多只能有一个对象纯在（Single Instance），singleton

要控制类的对象的数目，必须对外隐藏构造函数

- 将构造函数的访问属性设置为 private
- 定义 instance 并初始化为 NULL
- 当需要使用对象时，访问 instance的值
  1. 空值：创建对象，并用 instance 标记
  2. 非空：返回 instance 标记的对象

```
class SObject
{
    static SObject* c_instance;
```

```
    SObject(){}
    SObject(const SObject&);
    SObject& operator=(const SObject&);

public:
    void print()
    {
        cout<<"this = "<<this<<endl;
    }
    static SObject* get_instance();

};
SObject* SObject::c_instance = NULL;

SObject* SObject::get_instance()
{
if(c_instance == NULL)
{
     c_instance = new SObject();
}
return c_instance;

}

//  使用
SObject* s1 = SObject::get_instance();
s1->print();

SObject* s2 = SObject::get_instance();
s2->print();

SObject* s3 = s1;   // OK

SObject s4 = *s1;   // ERRO，因为 拷贝构造函数是私有 SObject(const
SObject&);
```

存在的问题：需要使用单例模式时，必须定义静态变量 c_instance，必须定义静态成员函数 get_instance()

解决方法：**将单例模式相关的代码抽取出来**，开发单例类模板

当需要单例类时，直接使用单例类模板

1.单例类模板定义

```
#ifndef SINGLETON_H
```

```
#define SINGLETON_H

template <typename T>
class Singleton
{
	static T* c_instance;
public:
	static T* getInstance();
};


template <typename T>
T* Singleton<T>::c_instance = NULL;


template <typename T>
T* Singleton<T>::getInstance()
{
	if(c_instance == NULL)
	{
		c_instance = new T();
	}
	return c_instance;

}

#endif
```

2.定义单例类，并使用

```
class SObject
{
	friend class Singleton<SObject>;   // 该类使用单例类模板

	SObject(){}
	SObject(const SObject&);
	SObject& operator=(const SObject&);

public:
	void print()
	{ cout<<"this = "<<this<<endl;
	}

	};
```

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

	SObject* s1 = Singleton<SObject>::getInstance();
	s1->print();

	SObject* s2 = Singleton<SObject>::getInstance();
	SObject* s3 = Singleton<SObject>::getInstance();
	s2->print();
	s3->print();

	SObject* s4 = s1;
	s4->print();
	return 0;
}
```

2.8 异常处理

2.8.1 C的异常处理

异常：程序在运行过程中可能产生异常

异常 Exception 与 Bug 区别

- 异常是程序运行时可预料的执行分支
- Bug 是程序中的错误，是不被预期的运行方式

例如：

异常：运行时除0，需要打开的外部文件不存在，数组访问越界

Bug：使用野指针，堆数组使用结束后未释放，选择排序无法处理长度为0的数组

---

c语言中，使用 if else

```
double divide(double a, double b,int* valid)
{
	const double delta = 0.000000001;
	if( !((b>-delta )&&(b<delta)) )		// if 处理正常逻辑
	{
		*valid = 1;
		return a/b;
	}
	else
```

```
        {
                cout<<"devide by zero"<<endl;
                *valid = 0;
                return 0;
        }
}
// 使用
int valid;
double r = divide(1,1,&valid);
if(valid)
{
        cout<<"r = "<<r<<endl;
}
r=divide(1,0,&valid);
if(valid)
{
        cout<<"r = "<<r<<endl;
}
```

缺陷：divide() 需要三个参数，调用后必须判断 valid 代表的结果，当valid为1时，运算结果才是正常

优化：通过 setjmp() 和 longjmp() 进行优化，本质上还是 if else

```
int setjmp(jmp_buf env)      // 将当前上下文保存在 jmp_buf 结构体中
void longjmlp(jmp_buf env, int val)   // 从 jmp_buf 结构体中恢复 setjmp() 保存的上下文，最终从 setjmp() 调用点返回，返回值为 val
```

必然涉及到全局变量，这两个函数简单粗暴，会产生跳转，破坏了 顺序、选择、循环的代码逻辑，可读性降低

C语言中，正常逻辑代码和异常处理代码混合在一起，导致代码迅速膨胀，难以维护

2.8.2 C++的异常处理

try catch 用于分隔**正常功能代码**与**异常处理代码**

try catch 可以直接将函数实现分隔为 两部分

函数声明和定义时可以**直接指定可能抛出的异常类型**

异常声明成为函数的一部分可以提高代码的可读性

如果在main中抛出异常

```
class Test{

public:
      Test(){
            cout<<"Test"<<endl;
      }
      ~Test(){
            cout<<"~Test()"<<endl;
      }
};

int main(int argc, char *argv[])
{
      static Test t;
      throw 1;
      return 0;
}
// 结果不会执行析构函数，打印：terminate called after throwing an instance
of 'int'
```

如果异常无法被处理，terminate() 会自动被调用

默认情况下，terminate()调用库函数 abort() 终止程序

abort() 是的程序执行异常而**立即退出**

C++支持替换默认的 terminate() 实现

---

terminate() 的替换

步骤一：自定义一个无返回无参数的函数

不能抛出任何异常

必须以某种方式结束当前程序

步骤二：调用 set_terminate() 设置自定义的结束函数

参数类型为 void(*)()

返回值为默认的 terminate() 入口地址

```
void my_terminate()
{
      cout<<"my_terminate()"<<endl;
```

```
//      exit(1);  // 结果会打印 析构函数，因为exit会确保程序结束时所有析构函数被调用
        abort();   // 结果不会打印析构函数
}


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

        static Test t;
        throw 1;
        return 0;
}
```

如果异常没有被处理，最后 **terminate() 结束整个程序**

terminate() 是整个程序**释放系统资源的最后机会**

terminate() 结束函数可以自定义，当不能抛出异常

析构函数中不能抛出异常，可能导致 terminate() 多次调用

2.8.3 函数的异常规格说明

函数异常声明的注意事项

- 函数异常声明是一种与编译器之间的契约
- 函数声明异常后就只能抛出声明的异常
  抛出其它异常将导致程序运行终止
  可以直接通过异常声明定义无异常函数

```
int func(int i ) try
{
    return i;
}
catch(...)
{
    return -1;
}

int func(int i, int j)
throw(int)
{
        return i+j;
```

```
}
```

```
int func(int i , int j )
throw(int)    // 若可能抛出多种类型异常，则 throw(int, char)
{
    if( (j>0) && (j<10) )
        return i+j;
    else
        throw 0;   // throw 'a';  将导致程序异常终止
}


void test(int i)
try
{
    cout<<"func"<<func(i,i)<<endl;
 }
catch(int i)
{
      cout<<"exception: "<<i<<endl;
}
catch(...)
{
      cout<<"exception  "<<endl;
}
```

如何判断一个函数是否会抛出异常，抛出哪些异常

C++提供语法用于声明函数所抛出的异常

异常声明作为函数声明的修饰符，写在参数列表后面

```
/* 函数声明 */
void func1();   // 可能抛出异常
void func2() throw(char, int);   // 只能抛出异常类型： char 和 int
void func3() throw();   // 不抛出异常
```

异常规格说明的意义：

- 提示函数调用者必须**做好异常处理的准备**
- 提示函数的维护者**不要抛出其它异常**
- 异常规格说明是**函数接口的一部分**

如果抛出的异常不在声明列表中，会发生两个可能情况

（1）程序异常结束

（2）正常捕获

函数抛出的异常不在规格说明中，全局 `unexpected()` 被调用

默认的 unexpected() 会调用全局的 terminate()

可以自定义函数替代默认的 unexpected() 实现

注意：不是所有的C++ 编译器都支持这个标准行为

```
void func() throw(int)
{
    cout<<"func"<<endl;
    throw 'c';
}
int main()
{
    try{
        func();
    }
    catch(int)
    {
        cout<<"catch int"<<endl;
    }
    catch(char)
    {
        cout<<"catch char"<<endl;
    }
}
```

unexpected() 的替换

步骤一：自定义一个无返回值无参数的函数

能够再次抛出异常：当异常符合触发的异常规格说明时，恢复程序执行

否则，调用全局 terminate() 结束程序

步骤二：调用 set_unexpected() 设置自定义的异常函数

参数类型为 void(*)()

返回值为默认的 unexpected() 入口地址

```
void func5() throw(int)
{
      cout<<"func"<<endl;
      throw 'c';
}

void my_unexpected()
{
      cout<<"my_ expected()"<<endl;
//     exit(1);  // 调用全局 terminate() 结束程序
      throw 1;   // 程序恢复运行
}


// 使用
set_unexpected(my_unexpected);

try{
       func5();
     }
catch(int)
     {
         cout<<"catch int"<<endl;
     }
    catch(char)
    {
         cout<<"catch char"<<endl;
    }
```

总结

- C++中的函数可以声明异常规格说明
- 异常规格说明可看作**接口的一部分**
- 函数抛出的异常不在规格说明中，unexpected() 被调用
  unexpected() 中能够再次抛出异常，再次抛出的异常能够匹配时，恢复程序的执行；否则，调用 terminate() 结束程序

案例：构造函数和析构中如果抛出异常

（1）如果构造函数中抛出异常

构造过程立即停止

当前对象无法生成

析构函数不会被调用

对象所占用的空间立即收回

```
class TestConstract
{
public:
    TestConstract(){
        cout<<"TestConstract"<<endl;
        throw 0 ;
    }
};


TestConstract* p = reinterpret_cast<TestConstract*> (1);
try
{
     p = new TestConstract();
}
catch(...)
{
     cout<<"Exception..."<<endl;
}
```

// 使用内存检测工具 valgrind 发现上述代码不会产生内存泄漏

工程中的建议：不要在构造函数中抛出异常，当构造函数中可能产生异常时，使用二阶构造模式

（2）析构函数中抛出异常

导致 对象使用的资源无法完全释放，或者delete两次

```
class Test{
public:
     Test(){
          cout<<"Test"<<endl;
    }
     ~Test(){
          cout<<"~Test()"<<endl;
          throw 2;
    }
};


void my_terminate()
{
```

```
        cout<<"my_terminate()"<<endl;
        exit(1);
//      abort();
}

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

        static Test t;
        return 0;
}
```

因此，**在构造函数和析构函数中，不要抛出异常**

第三章 C++11 新特性

---

auto

原始意义：与static和extern地位对等

```
static int s_var;   // 变量在全局数据区分配空间
extern int g_var;   // 外部全局变量（不再分配空间）
auto int i;    // 变量在栈上分配空间
```

现代意义：自动变量类型推断

```
auto p = new int(3);
QSharedPointer<auto> pointer(new int(2));   // ERROR, auto不能推断模板参数
```

---

decltype

编译时自动推断类型，同auto

```
int x = 0;
decltype(x)y = x;   // OK

QSharedPointer<IPCMessage> p1;
decltype(p1) p2;   // OK
```

---

constexpr 只能用来定义真正意义的常量

```
constexpr int i = 1; // OK
int j = 2;
constexpr int k = j;// ERROR，必须要在编译时就知道constexpr的值
```

constexpr与const

- constexpr可看作经典c++中const的子集
- constexpr仅仅针对“只读变量可进符号表”的场景
- constexpr可修饰函数的返回值，此时，函数返回值在编译时可确定(即返回值必须都是常量)

```
conexpr int f(int i)
{   return i? 1: 2; }
```

右值引用 &&

Lambda表达式

- 本质上是一个可调用的代码单元
- 可看作一个匿名函数
- 能够直接使用同作用域中的变量
- 可用于实现回调函数

```
[capture list](parameter list)->return_type{code snippet}
捕获列表：需要使用的相同作用域变量列表
参数列表
返回值类型
代码片段
[valist]   对以，分隔的变量，以值方式捕获，如 [i,j,k] {}不能出现i=1; 因为是只读的
[=]   捕获所有同作用域的变量，以值方式捕获
[&]   捕获所有同作用域的变量，以引用方式捕获
[&,valist]   捕获所有除 valist 外中的变量同作用域的变量(以引用方式)，valist 中的变量以值方式捕获，如 [&,&i,&j,&k]
[&valist]   使用引用捕获valist中的变量，如 [&i,&j,&k]
```

Lambda表达式与mutable

- 按值捕获的变量是不可改变的（**只读外部变量**）

- **如果需要改变按值捕获的外部变量，必须对Lambda表达式进行mutable声明**
- mutable只能改变外部变量的拷贝，而无法改变外部变量本身
- 按值捕获的外部变量，无法被Lambda表达式改变

```
[i,j]()->void{i=1;}    // ERROR
[i,j]()mutable->void{i=1;}     // OK，实际上的i没改变
[&i,j]()mutable->void{i=1;}    // OK，实际上的i改变了
```

Lambda一般用来做短小的回调函数

```
vector<int> v ={5,4,3,2,1};
for_each( v.begin(),v.end(),[](int v){cout<<v<<endl;} );
```

Lambda的灵魂是类和对象

枚举enum

传统枚举的本质是定义整型常量，因此

- **同作用域同名枚举会编译报错**
- **枚举类型与整型可进行隐式类型转换**
- 枚举值具体类型依赖编译器（灰色地带）

```
enum{max = 5};     // 真正意义的常量
#define max 5     // 文本替换
const max = 5;     // 只读变量
```

```
enum AE{A =1,B=2,C=3};
enum BE{C=1,B=2,A=3};    // 重定义报错
```

现代C++强类型枚举

- 强作用域：**枚举成员属于不同类型**
- 类型转换：**枚举成员的值不可与整型隐式类型转换**
- 类型指定：强类型枚举的“底层类型”可指定

```
enum class AE{A =1,B=2,C=3};
enum class BE{C=1,B=2,A=3};
AE a = B;   // ERROR，未定义
AE a = AE::B;   // OK
cout<<a;       // 以前的C++可以，现代的报错
cout<<(int)a;   // OK
AE a = BE:B;   // 以前的C++可以，现代的报错
AE a = 1  ;   // 以前的C++可以，现代的报错
```

```
enum CE{CE_V = 0xFFFFFFFF};
enum DE{DE_V = 0xFFFFFFFFFful};
cout<<sizeof(CE)<<"=? "<<sizeof(DE)<<endl;   // 对于以前的C++，打印出来的结果是相等，或者不相等，依赖于编译器
// 现代C++定义底层类型
enum class CE:char{A = '1',B = '2'};   // 定义一个枚举CE，底层类型是 char
cout<<sizeof(CE);   // 1（现代C++），以前的C++打印出来可能是1，4，8
```

没有现代C++），以前的C++使用枚举

```
class Const{
Const();
~Const();
Const& operator=(const Const&);
public:
enum{
    A = 1,B=2,
};
};
```

现代C++：当类对象被用于条件语句中（if，while，for），仍可进行隐式类型转换（为了兼容）

其他情况，类对象不允许进行隐式类型转换

```
MyInt i =1;
if(i)   // OK
```