0、C++综述

Bjarne Stroustrup 对C语言进行了扩展和创新，取名为 C With Class 。到了1983年正式改名为C++. 既支持面向过程的编程模式，又新增了面向对象编程模式和泛型编程模式。

嵌入式初级阶段： C++核心编程 + QT编程就够了。

1.面向过程编程

1.1 第一个C++程序

- 编写程序

```
vim 01helloworld.cpp
```

```
#include <iostream>

using namespace std;

int main(void){

    cout << "helloworld!" << endl;

    return 0;
}
```

- 编译运行

```
g++ 01helloworld.cpp  -o helloworld   #或者

gcc 01helloworld.cpp -o helloworld  -lstdc++

./helloworld
```

- 简单分析
  - 文件命名

    C++文件一般命名为.cpp

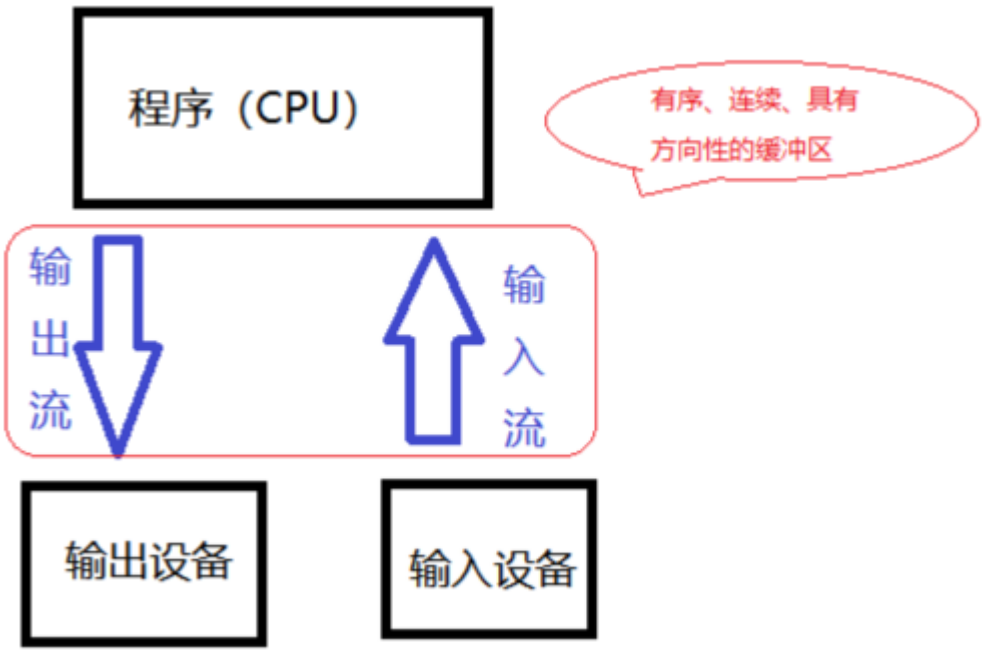
  - 代码

```
/*
  C++ 使用IO相关的函数时的标准头文件  类似于 stdio.h
  C++风格的很多头文件没有.h后缀
  C++兼容C, C++中可以使用stdio.h   也提供了C++风格的头文件  cstdio
  该头文件一般位于 /usr/include/c++/编译器版本/
*/
#include <iostream>
/*名字空间*/
using namespace std;

int main(void){
    /*类似于  printf("helloworld\n")
      cout 输出对象
      <<, 输出插入运算符
      endl, 相当于  '\n'
    */
    cout << "helloworld!" << endl;

    return 0;
}
```

1.2 数据的输入输出

1.2.1 流的概念

C++ 中的输入与输出可以看做是一连串的数据流，输入即可视为从文件或键盘中输入程序中的一串数据流，而输出则可以视为从程序中输出一连串的数据流到显示屏或文件中。



输入流： 从输入设备流向内存的字节序列

输出流： 从内存流向输出设备的字节序列

1.2.2 cout和插入运算符<<

当程序执行都cout语句时，遇到<<运算符就会将要是输出的信息插入到输出流中去，最终将输出流中的数据会被输出到标准输出设备（通常为屏幕）上去。

```
cout<<x;
```

输出时自动判断基本数据类型的类型。

```
#include <iostream>

using namespace std;

int main(void){
    int x = 10;
    float y = 1.1;
    char z = 'c';

    /*
     *printf("%d %f %c\n", x, y,z);
     * */
    cout << x <<" "<< y <<" "<< z << endl;

    return 0;
}
```

cout的优势在于自动解析这些基本数据类型。当然cout也可以格式化输出

1.2.3 cin和析取运算符>>

```
cin >>x;
```

当程序执行到cin语句时，就会停下来等待键盘数据的输入。输入数据被插入到输入流中，数据输完后按Enter键结束。当遇到运算符>>时，就从输入流中提取一个数据，存入变量x中。

需要说明的几点内容：

- 在一条cin语句中可以同时为多个变量输入数据。 各输入数据之间用一个或多个空白作为间隔符

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

int main(void){

    int x, y, z;
#if 0
    scanf("%d %d %d", &x, &y, &z);
    printf("%d %d %d\n", x, y, z);
#endif
    cin >> x >> y >> z;
    cout << x << " " <<  y <<  " " << z << endl;
    return 0;
}
```

- cin具有自动识别数据类型的能力，析取运算符>>根据它后面的变量类型从输入流中为他们提取对应的数据。

  比如： cin >>x;

  假设输入数据2， 析取运算符>>将根据其后x的类型决定输入的2到底是数字还是字符。若x是char类型，则2就是字符；若x是int，float之类的类型，则2就是一个数字。

  假设输入34，且x是char类型，则只有字符3被存储到x中，4将继续保存在流中。

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

int main(void){
    int a;
    double b;
    char c;

    cin >> a >> b >> c; //12.34a
    cout << "a: " << a << " b: " << b << " c: " << c << endl;

    return 0;
}
```

1.3名字空间（了解）

1.3.1名字空间的由来

最初C++标准中并没有名字空间，要求程序中全局作用域中声明的变量、函数、类型等必须具有唯一的名字。如果在同一个程序中有两个名字相同的全局变量将产生命名冲突（和C语言一样）。

如果程序中引入第三方库就必须保证程序中定义的全局名都不能与所用库中的名字相同，否则就会产生冲突，这就是所谓的全局名字空间污染问题，该问题在大型程序中处理起来非常困难。为此引入了名字空间。

在一个名字空间中，可以定义许多不同对象，并将这些对象的有效范围局限在名字空间内。不同名字空间中，可以定义相同名称的对象，只要两个同名对象不在同一名字空间中，就不会引起冲突。

1.3.2 名字空间的定义

- 语法格式

```
namespace  xxx_name{
    members;
}
```

- 举例

```
namespace  ABC{
    int num;
    struct stu{
        int age;
        char *name;
    };
    double add(int a, int b) {
        return (double)a+b;
    }
    int Min(int a, int b){
        return  a>b?a:b;
    }
}
```

1.3.3 名字空间成员的访问

名字空间成员的访问有主要有三种方法：

- 通过作用域限定符(::)

```
#include <iostream>

using namespace std;

namespace ns1{
    void func(void){
        cout << "ns1 func" << endl;
    }
}

namespace ns2{
    void func(void){
        cout << "ns2 func" << endl;
    }
}
int main(void){
    ns2::func();
    ns1::func();
    return 0;
}
```

- using引用名字空间单个成员

```
#include <iostream>

using namespace std;
```

```
namespace ns1{
    void func(void){
        cout << "ns1 func" << endl;
    }
}

namespace ns2{
    void func(void){
        cout << "ns2 func" << endl;
    }
}
int main(void){
    using ns1::func;

    ns2::func();
    //ns1::func();
    func();
    return 0;
}
```

- using引用名字空间全部成员

```
#include <iostream>

using namespace std;

namespace ns1{
    void func(void){
        cout << "ns1 func" << endl;
    }

    int a = 100;
}

namespace ns2{
    int a = 200;
    void func(void){
        cout << "ns2 func" << endl;
    }
}
int main(void){
    using namespace ns1;

    ns2::func();
    //ns1::func();
    func();
    cout << a << endl;


    return 0;
}
```

1.3.4 无名名字空间

未命名的名字空间称作无名名字空间。

```
#include <iostream>

//using namespace std;
```

```
namespace ns1{
    void func(void){
        std::cout << "ns1 func" << std::endl;
    }

    int a = 100;
}

namespace ns2{
    int a = 200;
    void func(void){
        std::cout << "ns2 func" << std::endl;
    }
}
namespace{  //无名名字空间
    int a = 300;
}


int main(void){
    //using namespace ns1;

    ns2::func();
    ns1::func();
    std::cout << ns1::a << std::endl;
    std::cout << ::a<< std::endl;  //无名名字空间成员的引用

    return 0;
}
```

1.3.5名字空间嵌套

指定义在其他名字空中的名字空间。

```
#include <iostream>

using namespace std;

namespace ns1{
    void func(void){
        std::cout << "ns1 func" << std::endl;
    }

    int a = 100;
    namespace ns2{
        int b = 111;
    }
}
int main(void){
    cout << ns1::ns2::b << endl;

    return 0;
}
```

1.4 C++对C语言数据类型的扩展

基本数据类型 char、 unsigned char 、int、 short、unsigned short 、 long 、unsigned long 、float double、long double与C语言相同。扩展了bool类型，对结构体、联合、枚举做了改进。

1.4.1 结构体

- C++中定义结构型变量，可以省略struct关键字
- C++结构体中可以直接定义函数，谓之成员函数（方法）

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

int main(void){

    struct stu{
        int age;
        char name[20];
        void who(void){
            cout <<"我是："  << name << " 我今年："  << age <<endl;
        }
    };

    stu  s1;
    s1.age = 21;
    strcpy(s1.name, "张飞");
    s1.who();

    return 0;
}
```

1.4.2 联合

- C++中定义联合体变量，可以省略union关键字

```
union XX{……};
XX x;//定义联合体变量直接省略union
```

- 支持匿名联合

```
union { //没有名字
    ……
};
```

```
#include <iostream>

using namespace std;

int main(void){

    union{       //匿名联合
        int num;
```

```
        char c[4];
    };

    num = 0x12345678;
    cout << hex << (int)c[0] <<" " << (int)c[1] << endl;
    return 0;
}
```

1.4.3 枚举

- C++中定义枚举变量，可以省略enum关键字
- C++中枚举是独立的数据类型，不能当做整型数使用

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

int main(void){
    enum COLOR{RED, GREEN, BLUE};

    COLOR c = GREEN;
    //c = 2; //error

    cout << c << endl;

    return 0;

}
```

1.4.4 布尔

C++中布尔(bool)是基本数据类型，专门表示逻辑值

布尔类型的字面值常量：

true 表示逻辑真
false表示逻辑假

布尔类型的本质： 单字节的整数，使用1表示真，0表示假

任何基本类型都可以被隐式转换为布尔类型

```
#include <iostream>

using namespace std;

int main(void){

    bool b = true;

    cout << b <<endl;
    cout <<boolalpha <<  b <<endl;

    b = 3 + 2;
    cout <<boolalpha <<  b <<endl;
```

```
    return 0;
}
```

1.4.5 字符串

- C++兼容C中的字符串表示方法和操作函数
- C++专门设计了string类型表示字符串
  - string类型字符串定义

```
string s; //定义空字符串
string s("hello");
string s = "hello";
string s = string("hello");
```

  - 字符串拷贝

```
string s1 = “hello”;
string s2 = s1;
```

  - 字符串连接

```
string s1 = “hello”, s2 = “ world”;
string s3 = s1 + s2;//s3:hello world
s1 += s2;//s1:hello world
```

  - 字符串比较

```
string s1 = “hello”, s2 = “ world”;
if(s1 == s2){ cout << “false”<< endl; }
if(s1 != s2){ cout << “true”<< endl; }
```

  - 随机访问

```
string s = “hello”;
s[0] =“H”; //Hello
```

  - 获取字符串长度

```
size_t  size();
size_t length();
```

  - 转换为C风格的字符串

```
const char* c_str();
```

  - 字符串交换

```
void swap(string s1,string s2)
```

- 实例代码

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

int main(){
    /*定义*/
    string s1;  //定义空字符串
    string s2("aaa");
    string s3 = string("bbb");
    string s4 = "cccc";

    /*字符串的拷贝*/
    string s5 = s2;  // char *p5 = p2;
    cout << "s5 = " << s5 << endl;

    /*拼接*/
    s5 += s3;
    cout << "s5 = " << s5 << endl;

    /*字符串比较*/
    if(s2 == s3){    //strcmp(.....)
        cout << "true" << endl;
    }
    else
        cout << "false" << endl;
    /*取字符串长度*/
    cout << "s5 length = "<< s5.length() << endl;
    /*转换为C风格字符串*/
    const char *p = s5.c_str();
    printf("%s\n", p);

    /*交换*/
    swap(s2, s3);
    cout << "s2= "<< s2 << "   s3= "<< s3<< endl;
    return 0;
}
```

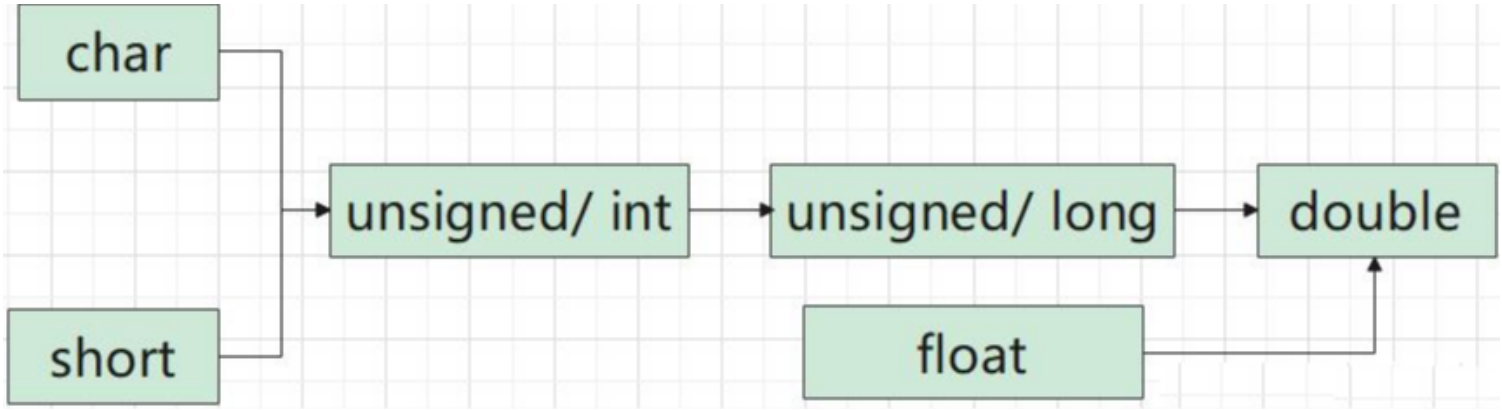
1.5 类型转换

类型转换分为隐式转换和显示转换。

写C/C++代码的时候，有时候不可避免的会使用类型转换，良好的编码风格中应该避免隐式转换，隐式转换有时候会产生不易察觉的问题。

1.5.1 隐式类型转换

C++定义了一套标准数据类型转换的规则，在必要时C++会用这套转换规则进行数据类型的转换。这种转换是在程序员不参与的情况下自动进行的，所以成为隐式类型转换。转换原则：



以下四种常见类型会发生隐式转换：

- 多种数据类型的算术表达式中

```
int a = 2;
float b = 3.4;
double d = 2.2;
a+b+c;
```

- 将一种数据类型赋值给另外一种数据类型变量

```
int a = 2;
float b = 3.4;
long double d = 2.2;
b = a;
d = a;
```

- 函数调用时，若实参表达式与形参的类型不相符

```
int Min(int a, int b){
    return a<b?a:b;
}
int a = 2;
float b= 3.4;
int x = Min(b, a+3.5);
```

- 函数返回时，如果返回表达式的值与函数返回类型不同

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

1.5.2 显示类型转换(了解)

显示类型转换也称为强制类型转换，是指把一种数据类型强制转换为指定的另一种数据类型

```
int a = 4;
float c = (float) a; //C风格  C++也支持
float d = float(a); //C++风格 C不支持
```

C++ 提供了更严格的类型转换，可以提供更好的控制转换过程，C++增加了四个强制转换运算符： static_cast, dynamic_cast, const_cast和reinterpret_cast.

- 静态类型转换 static_cast

    - 目标类型变量 = static_cast<目标类型> (源类型变量)
    - 用于隐式转换的逆转换，常用于基本数据类型之间的转换、void* 转换为其它类型的指针
- 不能用于整型和指针之间的互相转换， 不能用于不同类型的指针、引用之间的转换 （风险高）

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(void){
    int a = 100;
    double a1 = (double)a; //c风格
    double a2 = double(a);//C++风格

    double b = static_cast<double>(a);

    void *p = malloc(100);
    int *pi = static_cast<int *>(p);
    char *pc = static_cast<char *>(p);

//int num = static_cast<int>(p); //error
    //pi = static_cast<int *>(pc); //error

    return 0;
}
```

    - 用于自定义类型的转换(向上造型，后面讲)
- 重解释类型转换 reinterpret_cast
    - 目标类型变量 = reinterpret_cast<目标类型> (源类型变量);
    - 用于任意类型指针或引用之间的转换
    - 指针和整型数之间的转换

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(void){
    int a = 100;
    double a1 = (double)a; //c风格
    double a2 = double(a);//C++风格

    double b = static_cast<double>(a);

    void *p = malloc(100);
    int *pi = static_cast<int *>(p);
    char *pc = static_cast<char *>(p);

    //int num = static_cast<int>(p); //error
    //pi = static_cast<int *>(pc); //error

    int num = reinterpret_cast<int>(p);
    pi = reinterpret_cast<int *>(pc);


    int n = 0x00414243;

    char *ps = reinterpret_cast<char *>(&n);
    cout << ps << endl;
    cout << hex << n << endl ;
```

```
    return 0;
}
```

- 常类型转换 const_cast
  - 目标类型变量 = const_cast<目标类型> (源类型变量);
  - 用于去除指针或引用的常属性

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

int main(void){

    int tmp = 123;

    const int *p1 = &tmp; //不能通过指针修改tmp对应的值
    //(*p1)++; //error

    int *p2 = const_cast<int *>(p1);//去除常属性

    *p2 = 10000;
    cout << tmp << endl;
    return 0;
}
```

- 动态类型转换 dynamic_cast
  - 目标类型变量 = dynamic_cast<目标类型> (源类型变量);
  - 主要用于多态中类指针的向下转型，可以检测是否可以转型成功 (后面讲)

1.6 引用

c++出现了新的概念：引用。引用是某个对象的别名。

语法格式如下：

类型 &引用名 = 变量名;

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

int main(void){
    int i = 10;
    int &ir  = i; // 给变量i起了被别名 叫ir

    cout << "i = " << i << endl;
    cout << "ir = " << ir << endl;


    cout << "i的地址: " << &i << endl;
    cout << "ir的地址: " << &ir << endl;
    return 0;
}
```

可以看出来ir和i其实是同一块内存。

用途：

1）简化编程， 用指针的场景可以用引用替换 （尽量减少指针的使用）

2）系统开销更小

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

void swap(int *a, int *b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void swap2(int &a, int &b){
    int tmp = a;
    a = b;
    b = tmp;
}
struct stu{
    char name[20];
    int age;
    int id;

};
void func(struct stu s1){

}
void func2(struct stu &s1){
}
int main(void){
    int x = 10;
    int y = 20;

    //swap(&x, &y);
    swap2(x,y);

    cout << "x= "<< x << "\t" << "y= " << y << endl;

    stu zs;
    func(zs);
    func2(zs);

    return 0;
}
```

在使用引用时需要注意以下几个问题：

- &的位置是灵活的，以下几种定义完全相同

```
int&  ir = i;
int & ir = i;
int &ir = i;
```

- 在变量声明时出现&才是引用运算符（包括函数参数声明和函数返回类型的声明）

```
int &ir =i;
int &f(int &i1, int &);
```

- 引用必须定义时初始化

```
float f;
float &r1 = f;
float &r2;
r2 = f; //错误
```

- const引用（常引用）。在定义引用时一般为左值（变量）。

  左值，是指变量对应的那块内存区域，是可以放在赋值符号左边的值；

  右值，是指变量对应的内存区域中存储的数据值，是可以“放在赋值符号右边的值”。

  常量、表达式都是右值，例如：

  ```
  int i = 1;
  i = i+10;
  i+10 = i;//错误
  i=10;
  10=i; //错误
  ```

  可以使用const进行限制，使他成为不允许被修改的常量引用。

  ```
  int &i = 3; //错误
  const int &i = 3;// 正确
  int a = 100;
  char &c = a; //错误 将a转换char类型，转换结果保存到临时变量中 实际引用临时变量，而临时变量是右值
  ```

- 引用的本质

  引用的本质就是指针常量。

  ```
  //int *const x = &m; int *const y= &n
  void swap(int &x, int &y){
        int tmp = 0;
        tmp = x;  //tmp = *x;
        x  = y;     // *x = * y;
        y = tmp; // *y = tmp
  }
  int  main(void){
      int  m = 10, n = 20;
      swap(m, n);
      return 0;
  }
  ```

1.7 函数

1.7.1函数缺省参数

在C++中，函数的形参列表中的形参是可以有默认值的。有默认值的参数即为默认参数 。

在函数调用时，有默认参数可以缺省。

语法：返回值类型 函数名 （参数= 默认值）{函数体}

```
```

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

int add(int x, int y, int z=100){
    return x + y + z;
}

int main(void){

    cout << add(1,2,3) << endl;
    cout << add(1,2) << endl;

    return 0;
}
```

注意事项：

- 靠右原则：如果某个位置参数有默认值，那么从这个位置往后，从左向右，必须都要有默认值。

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

int add(int x, int y=10, int z){ //error
    return x + y + z;
}

int main(void){

    cout << add(1,2,3) << endl;
    cout << add(1,2) << endl;

    return 0;
}
```

- 函数声明和函数实现（即函数定义），只允许其中一个有默认值，即如果函数声明有默认值，则函数实现的时候就不能有缺省参数。

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

int add(int x, int y, int z=200); //error
int main(void){

    cout << add(1,2,3) << endl;
    cout << add(1,2) << endl;

    return 0;
}
int add(int x, int y, int z = 200){ //error
    return x + y + z;
}
```

1.7.2 哑元

只有类型而没有变量名的参数称为“哑元”。

```
int func(int a, int ){
    ...
}
```

需要使用哑元的场景：

- 兼容旧代码,保证函数的向下兼容性

```
void func(int i,int j){…}==升级==>void func(int i){…}
void func(int i,int j){…}==升级==>void func(int i, int/*哑元*/){…}
```

- 操作符重载中，区分前后++/--

1.7.3 引用参数

- 可以将引用用于函数的参数，这时形参就是实参的别名
- 引用型函数参数作用
    - 在函数中修改实参的值

```
#include <iostream>
using namespace std;
void swap(int & ,int&);
int main()
{

      int i =3,j = 5;
      swap(i,j);
      cout<<"i="<<i<<endl<<"j="<<j<<endl;
      getchar();
      return 0;
}

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

    - 避免实参到形参数值复制的开销，提高传参效率

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

struct Teacher{
    char name[100];
    int age;
};
void print( Teacher& t){
    cout << t.name << ',' << t.age/*++*/ << endl;
}
int main(void)
{
    const Teacher t = {"张飞",40};
    print(t);
```

```
    print(t);
    print(t);
    return 0;
}
```

- 引用型函数参数可能意外修改实参，如果不希望通过引用修改实参本身，可以将其声明为常引用，在提高传参效率的同时还可以接收常量型实参

```
int add(const int &a, const int &b){
    return a+b;
}

int main(void){
    add(3, 5);
    return 0;
}
```

1.7.4 返回引用

- 可以将函数的返回类型声明为引用型，这时函数的返回结果就是return后面数据的别名，避免了函数返回值的开销
- 函数中返回引用，一定要保证在函数返回以后，该引用的目标依然有效
    - 可以返回全局变量、静态变量和成员变量的引用
    - 可以返回引用型参数的本身
    - 可以返回调用对象自身的引用
    - 可以返回堆中动态创建对象的引用
    - 不能返回局部变量的引用//非常危险

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

int& add(const int &a, const int &b)
{
    int tmp  = 0;
    tmp = a + b;
    return tmp;  //风险   函数结束对应内存空间回收
}

int main(void){

    int x = 100;
    int y = 200;

    int &t = add(x,y);
    t++;//通过别名操作 已经被系统回收的内存
    cout << t <<endl;

    return 0;
}
```

1.7.5 函数的重载

- 基本使用

同一作用域，函数名相同，但是的参数表必须有所区分（类型、个数、顺序），将构成重载关系

```
```

```cpp
#include <iostream>

using namespace std;

void func(int a, char *p){

}
void func(char *p, int a){

}

void swap(int *a, int *b){
    int tmp = *a;

    *a = *b;
    *b = tmp;
}
void swap(char *a, char *b){
    char tmp = *a;

    *a = *b;
    *b = tmp;
}
void swap(bool *a, bool *b){
    bool  tmp = *a;

    *a = *b;
    *b = tmp;
}

int main(){

    int x = 10;
    int y = 20;
    swap(&x, &y);

    cout << x <<" " << y << endl;

    char a = 'a';
    char b = 'b';
    swap(&a, &b);
    cout << a << " " << b << endl;

    bool f1 = true;
    bool f2 = false;
    swap(&f1, &f2);
    cout << f1 << " " << f2 << endl;

    return 0;
}
```

- 重载的实现

  编译器将形参变量的类型作为最终函数名的一部分。

  ```
  nm  a.out
  ```

  注意:

  形参变量名不同 不构成重载的要素

  函数返回类型不同 不构成重载的要素

- 函数匹配的优先级

  当前g++编译器匹配的一般规则：
  1）完全匹配 //最高
  2）常量转换 //较好
  3）升级转换 //一般
  4）降级转换 //较差
  5）省略号匹配 //最差

```
#include <iostream>

using namespace std;

void bar(int i){
    cout <<"bar(int)" << endl;
}
void bar(const char c){
    cout << "bar(const char)" <<endl;
}

void foo(char c){
    cout << "foo(char)" << endl;
}
void foo(int i){
    cout << "foo(int)" << endl;
}

void hum(int i, ...){
    cout <<"hum(int, ...)" << endl;
}
void hum(int i, int j){
    cout << "hum(int, int)" << endl;
}
int main(){

    int m = 100;
    bar(m);
    char c = 'A';
    bar(c); //常量转换  char -----> const char

    short s = 10;
    foo(s); // short ----> int

    hum(10, 1.23);  // double -----> int 降级转换
    return 0;
}
```

注意二义性问题：

- 默认类型转换带来的二义性

```
int func(unsigned int x){
    return x;
}
double func(double x){
    return x;
}

int n = 12;
func(n); // error
```

- 缺省参数带来的二义性

```
void  f(int a, int b , int c=100){

}
void f(int a, int b){

}
f(1,2); // ambigous
```

1.8 内联函数

在函数声明或定义时，将inline关键字加在函数返回类型前面就是内联函数。

```
#include <iostream>

using namespace std;

inline int add(int x, int y){ //内联函数
    return x+y;
}

int main(void){

    int ret = add(3,5); // int ret = 3+5;
    return 0;
}
```

不需要建立函数调用时的运行环境，不需要进行参数传递， 不需要跳转,效率更高。

注意：

- inline关键字只是建议编译器做内联优化，编译器不一定做内联
- 内联函数的声明或定义必须在函数调用之前完成
- 一般几行代码适合作为内联函数， 像递归函数 包含循环 switch goto复杂逻辑的函数不适合内联

1.9 new和delete

在C语言中，如果需要使用堆内存，程序员可以用函数malloc()从堆中分配指定大小存储区域，用完之后必须用free()将之归还系统。如果用完之后没用free()释放，就会造成内存泄漏。

malloc()函数的使用比较麻烦，除了需要计算需求内存的大小之外，还必须对获得的内存区域进行类型转换。为此C++提供了new 和delete两个运算符。

- new的用法
    - p = new type
    - p = new type(x)
    - p = new type[n]

    其中p是指针变量，type是数据类型。用法1 只分配内存， 用法2将分配的堆内存初始化为x,用法3分配具有n个元素的数组。分配不成功返回空指针（NULL）。
- delete的用法
    - delete p;

- delete [] p;

其中p是用new分配到的堆空间指针变量。用法1用于释放动态分配的单个内存，用法2用于释放动态分配的数组存储区。

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


int main(){

    int *p1, *p2, *p3;

    //p1 =(int *) malloc(sizeof(int));
    p1 = new int ; //在heap(堆)中分配一个int类型内存空间
    p2 = new int(123); //在heap(堆)中分配一个int类型内存空间并初始化为123
    p3 = new int[123]; //在heap(堆)中分配123个int类型内存空间

    *p1 = 100;
    p3[3] = 0x55;


    //free(p1);
    delete p1;
    delete p2;
    /*注释数组内存空间的释放
     *如果写成 delete p3; 只释放了一个int 类型内存空间p3[0] 其余都未释放 造成内存泄漏
     * */
    delete [] p3;
    return 0;
}
```

面向对象编程

1 类与对象

1.1结构体

前面讲到C++中的结构体不仅可以包含不同类型的数据，而且还可以包含操作这些数据的函数。

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

struct Complex{

    double r ; //实部
    double i; //虚部

    void init(double rr, double ii){
        r = rr;
        i = ii;
    }

    double real(){
```

```
        return r;
    }
    double image(){
        return i;
    }

};

int main(void){
    Complex a;
    a.init(2,3);

    cout << a.real() << " + " << a.image() << "i" << endl;

    return 0;
}
```

将数据和操作数据的函数包装在一起的主要目的就是实现的封装和隐藏。隐藏就是不让结构体外的函数直接修改数据结构中的数据，只能通过结构的成员函数对数据进行修改。但上面的代码显然没能做到这一点。为此C++中新增了3个访问权限限定符，用于设置结构体中数据成员和函数成员的访问权限：

- public

  公有成员（函数、数据）， 可被任何函数访问（结构体内和结构体外）

- protected

  保护成员， 与继承相关，后续介绍

- private

  私有成员（函数、数据），只能被结构体内部函数访问

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

struct Complex{

private:
    double r ; //实部
    double i; //虚部
public:
    void init(double rr, double ii){
        r = rr;
        i = ii;
    }

    double real(){
        return r;
    }
    double image(){
        return i;
    }
    void set_real(double data){
        r = data;
    }
    void set_image(double data){
        i = data;
    }

};

int main(void){
```

```
    Complex a;
    a.init(2,3);

    //a.r = 8;
    a.set_real(8);

    cout << a.real() << " + " << a.image() << "i" << endl;

    return 0;
}
```

1.2类

struct还是容易和传统C语言中的结构混淆，C++中引进了功能与struct相同，但更安全的数据类型：类。

更安全是指结构体将所有成员都默认为publice，不够安全；类中成员默认为private权限。

```
//语法格式
class 类名{
private:
    成员数据;
    成员函数;
public:
    成员数据;
    成员函数;
protected:
    成员数据;
    成员函数;
};  //特别注意;不要忘了
```

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

//struct Complex{
class Complex{
private:
    double r ; //实部
    double i; //虚部
public:
    void init(double rr, double ii){
        r = rr;
        i = ii;
    }

    double real(){
        return r;
    }
    double image(){
        return i;
    }
    void set_real(double data){
        r = data;
    }
    void set_image(double data){
        i = data;
    }

};

int main(void){
    Complex a;
```

```
    a.init(2,3);

    //a.r = 8;
    a.set_real(8);

    cout << a.real() << " + " << a.image() << "i" << endl;

    return 0;
}
```

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

class Student{
private:
    string m_name;
    int m_age;
    int m_no;
public:
    void setName(const string& name){
        m_name = name;
    }
    void setAge(int age){
        if(age < 0)
            cout << "无效年龄" << endl;
        else
            m_age = age;
    }
    void setNo(int no){
        m_no = no;
    }
    void sleep(int hour){
        cout << "我睡了"<< hour <<"小时"<< endl;
    }
    void eat(const string &food){
        cout << "我正在吃" <<food << endl;
    }
    void learn(const string &course){
        cout << "我正在学习" << course << endl;
    }
    void who(){
        cout << "我叫: "<<m_name << ",  我今年" << m_age << endl;
    }

};

int main(void){
    Student s1;
    s1.setName("张飞");
    s1.setAge(21);
    s1.setNo(10003);
    s1.who();
    s1.eat("烧烤");

    Student s2;
    s2.setName("刘备");
    s2.setAge(28);
    s2.setNo(10000);
    s2.who();
    s2.learn("C++");

    return 0;
}
```

1.3构造函数

构造函数(constructor)是与类同名的特殊成员函数，主要用来初始化对象的数据成员。

```
class X{
    ...
    X（...）{ //构造函数
        ...
    }

};
```

构造函数的特点：

- 与类同名
- 没有返回类型
- 可以被重载
- 由系统自动调用，不允许在程序中显示调用

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

class Student{
private:
    string m_name;
    int m_age;
    int m_no;
public:
    Student(const string& name, int age, int no){
        cout << "Student constructor" <<endl;

        m_name = name;
        m_age = age;
        m_no = no;
    }
    void setName(const string& name){
        m_name = name;
    }
    void setAge(int age){
        if(age < 0)
            cout << "无效年龄" << endl;
        else
            m_age = age;
    }
    void setNo(int no){
        m_no = no;
    }
    void sleep(int hour){
        cout << "我睡了"<< hour <<"小时"<< endl;
    }
    void eat(const string &food){
        cout << "我正在吃" <<food << endl;
    }
    void learn(const string &course){
        cout << "我正在学习" << course << endl;
    }
    void who(){
        cout << "我叫: "<<m_name << ",  我今年" << m_age << endl;
    }
```

```
};

int main(void){
    Student s1("张飞", 21, 10003);
    s1.who();
    s1.eat("烧烤");

    Student s2("刘备", 28, 10000);
    s2.who();
    s2.learn("C++");

    return 0;
}
```

练习

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

class Desk{
private:
    int length, width, height, weight;
public:
    Desk(int l, int w, int h, int ww);
};

Desk::Desk(int l, int w, int h, int ww){
    cout << "Desk constructor" << endl;

    length = l;
    width = w;
    height = h;
    weight = ww;
}

int main(void){
    Desk d1(1,2,3,4);


    return 0;
}
```

1.3.1 缺省构造函数

缺省构造函数也称无参构造函数，但其未必真的没有任何参数，为一个有参构造函数的每个参数都提供一个缺省值，同样可以达到无参构造函数的效果

注意：

- 如果一个类没有定义任何构造函数，那么编译器会为其提供一个缺省构造函数
    - 对基本类型的成员变量，不做初始化
    - 对类类型的成员变量(成员子对象)，将自动调用相应类的无参构造函数来初始化

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

```
class A{

public:
    A(void){
        cout << "A的无参构造" << endl;
        m_i = 0;
    }
public:
    int m_i;

};
class B{
public:
    int m_j; //基本类型成员变量
    A m_a; //类类型成员变量（成员子对象）
};


int main(void){
    B b; //调用成员对象m_a的无参构造函数  调用B的缺省构造函数

    cout << b.m_j << endl; //未知
    cout << b.m_a.m_i <<endl; //0
    return 0;
}
```

- 如果一个类定义了构造函数，无论是否有参数，那么编译器都不会再提供缺省构造函数

1.3.2 构造函数的重载

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

struct param{
    int l;
    int w;
    int h;
    int ww;
};

class Desk{
private:
    int length, width, height, weight;
public:
    Desk(int l, int w, int h, int ww);
#if 1
    Desk(void){
        cout << "Desk(void)" << endl;
        length = 0;
        width = 0;
        height = 0;
        weight = 0;
    }
#endif
    Desk(param & p){
        cout << "Desk(parm &)" << endl;
        length = p.l;
        width = p.w;
        height = p.h;
        weight = p.ww;
```

```
    }

};
#if 1
Desk::Desk(int l, int w, int h, int ww){
    cout << "Desk(int, int, int, int)" << endl;

    length = l;
    width = w;
    height = h;
    weight = ww;
}
#endif

int main(void){
    Desk d1(1,2,3,4);

    Desk d2;  //调用无参构造函数

    param pm;
    pm.l = 1;
    pm.w = 2;
    pm.h = 3;
    pm.ww =4;
    Desk d3(pm);

    return 0;
}
```

某些重载的构造函数具有特殊的含义:

- 缺省构造函数：按缺省方式构造
- 类型转换构造函数：从不同类型的对象构造
- 拷贝构造函数：从相同类型的对象构造

1.3.3 类型转换构造函数

将其它类型转换为当前类类型需要借助**转换构造函数（Conversion constructor）** ， 转换构造函数只有一个参数 。

```
#include <iostream>
#include <cstring>

using namespace std;

class Integer{
private:
    int m_i ;
public:
    Integer(void){
        cout << "Integer(void)" << endl;
        m_i = 0;
    }
    explicit Integer(int n){ //类型转换构造函数
        cout << "Integer(int)" <<endl;
        m_i = n;
    }
    explicit Integer(const char *str){
        cout << "Integer(const char *)" << endl;
        m_i = strlen(str);
    }
```

```
    void print(void){
        cout << m_i << endl;
    }
};


int main(void){

    Integer a;
    a.print();

    //Integer b = 1; //编译器会找参数为int类型的构造函数
    Integer b = Integer(1); //编译器会找参数为int类型的构造函数
    b.print();

    //Integer c = "abc"; //编译器会找参数为const char *类型的构造函数
    Integer c = Integer("abc"); //编译器会找参数为const char *类型的构造函数
    c.print();
    return 0;
}
```

explicit关键字，就是告诉编译需要类型转换时，强制要求写成如下形式

```
Integer b = Integer(1);
//Integer b = 1; //error
```

1.3.4 拷贝构造函数

- 用一个已定义的对象构造同类型的副本对象，将调用该类的拷贝构造构造函数

```
class A{
    A(const A &that){ //拷贝构造函数 注意参数必须是常引用
        ...
    }
};
A a;
A b(a); //调用拷贝构造
A c = a; //调用拷贝构造
```

实例:

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

class A{
public:
    int m_data;

    A(int data = 0){
        cout << "A(int)" << endl;
        m_data = data;
    }
    A(const A& that){  //拷贝构造函数
        cout << "A(const A&)" << endl;
        m_data = that.m_data;
    }
};

int main(void){
```

```
    A a1;
    A a2(a1); //编译器会调用拷贝构造函数
    A a3 = a1; //调用拷贝构造

    return 0;
}
```

- 如果一个类没有显式定义拷贝构造函数，那么编译器会为其提供一个缺省拷贝构造函数
  - 对基本类型成员变量，按字节复制
  - 对类类型成员变量(成员子对象)，调用相应类的拷贝构造函数

```
class User {
    string m_name; // 调用string类的拷贝构造函数
    int m_age; // 按字节复制
};
```

实例:

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

class A{
public:
    int m_data;

    A(int data = 0){
        cout << "A(int)" << endl;
        m_data = data;
    }
#if 1
    A(const A& that){  //拷贝构造函数
        cout << "A(const A&)" << endl;
        m_data = that.m_data;
    }
#endif
};

class B{
public:
    A m_a;
};

int main(void){
#if 0
    A a1(123);
    A a2(a1); //编译器会调用拷贝构造函数
    A a3 = a1; //调用拷贝构造
    cout << a2.m_data <<endl;
    cout << a3.m_data <<endl;
#endif
    B b1; //调用A的构造函数
    b1.m_a.m_data = 98;
    B b2 = b1; //调用B的缺省拷贝构造函数，由于有成员对象m_a，所以会调用A的拷贝构造函数
    cout << b2.m_a.m_data << endl;
    return 0;
}
```

- 注意事项
  - 拷贝函数的调用时机
    - 用已定义对象作为同类型对象的构造实参
    - 以对象的形式向函数传递参数
    - 从函数中返回对象
  - 拷贝构造过程风险高而且效率低，设计时应尽可能避免
    - 避免或减少对象的拷贝
    - 传递对象形式的参数时，使用引用型参数
    - 从函数中返回对象时，使用引用函数返回值

1.4 初始化列表

1.4.1 构造函数的初始化列表

构造函数对数据成员进行初始化还可以通过成员初始化列表的方式完成。语法格式：

```
构造函数名(参数表): 成员1(初始值参数),成员2(初始值参数){

}
```

实例：

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

class Student{
private:
    string m_name;
    int m_age;
    int m_no;
public:
    Student(const string& name, int age, int no):m_name(name), m_age(age), m_no(no){
        cout << "Student constructor" <<endl;
    }
    void setName(const string& name){
        m_name = name;
    }
    void setAge(int age){
        if(age < 0)
            cout << "无效年龄" << endl;
        else
            m_age = age;
    }
    void setNo(int no){
        m_no = no;
    }
    void sleep(int hour){
        cout << "我睡了"<< hour <<"小时"<< endl;
    }
    void eat(const string &food){
        cout << "我正在吃" <<food << endl;
    }
    void learn(const string &course){
        cout << "我正在学习" << course << endl;
    }
    void who(){
        cout << "我叫: "<<m_name << ",  我今年" << m_age << endl;
```

```
    }

};

int main(void){
    Student s1("张飞", 21, 10003);
    s1.who();
    s1.eat("烧烤");

    Student s2("刘备", 28, 10000);
    s2.who();
    s2.learn("C++");

    return 0;
}
```

1.4.2 需要显式初始化列表的场景

一般而言，使用初始化列表和在构造函数体对成员变量进行赋初值，两者区别不大，可以任选一种，但是下面几种场景必须要使用初始化列表:

- 如果有类类型的成员变量(成员子对象)，而该类又没有无参构造函数，则必须要通过初始化列表显式指明其初始化方式

```
#include <iostream>
using namespace std;
class A{
private:
    int m_data;
public:
    A(int data){
        cout <<"A(int)" << endl;
        m_data = data;
    }
};
class B{
private:
    A m_a;
public:
    B(void):m_a(123){
        cout <<"B(void)" << endl;
    }
};

int main(void){

    B b; //一定会去构造成员对象m_a , 未指定如何构造，系统去调用m_a的无参构造函数
    return 0;
}
```

- “const”修饰的成员变量(常成员变量)必须要在初始化列表中初始化
- “引用型”成员变量必须要在初始化列表中初始化

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

int num = 12;

class A{
public:
    int& m_r;
    const int m_c;
```

```
/*
 * error
    A(void){
        m_r = num;
        m_c = 100;
    }
*/
    A(void):m_r(num), m_c(100){
    }
};

int main(void){

    A a;

    cout << a.m_r << " " <<a.m_c <<endl;

    return 0;
}
```

1.4.3 初始化顺序

类中成员变量按声明顺序依次被初始化，而与初始化表中的顺序无关

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


class A{
public:
    A(int a){
        cout <<"A constuctor" << endl;
    }
};

class B{
public:
    B(int b){
        cout <<"B constuctor" << endl;
    }
};

class C{
private:
    A m_a;
    B m_b;
public:
    C(int a, int b):m_b(b), m_a(a){

    }

};
int main(void){
    C c(1,2);

    return 0;
}
```

1.5this指针

1.5.1什么是this指针

不同的对象各自拥有独立的成员变量，但它们共享同一份成员函数代码，那么在成员函数中如何区分所访问的成员变量隶属于哪个对象？

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

class Student{
public:
    Student(int age, const string &name){
        m_age = age;
        m_name = name;
    }
    void print(){
        cout << m_name << ":" << m_age << endl;
    }
private:
    int m_age;
    string m_name;
};
int main(void){
    Student zs(12, "zhangsan");
    Student ls = Student(13, "lisi");
    zs.print();
    ls.print();
    return 0;
}
```

答案就是this指针。

this是一个用于标识对象自身的隐式指针，代表对象自身的地址。

在编译类成员函数时，C++编译器会自动将this指针添加到成员函数的参数表中。在用类的成员函数时，调用对象会把自己的地址通过this指针传递给成员函数。

以上程序编译器编译后的样子，大致如下：

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

class Student{
public:
    Student(Student *this, int age, const string &name){
        this->m_age = age;
        this->m_name = name;
    }
    void print(Student *this){
        cout << this->m_name << ":" << this->m_age << endl;
    }
private:
    int m_age;
    string m_name;
};
int main(void){
    Student zs(12, "zhangsan");  // (&zs, 12, "zhangsan")
    Student ls = Student(13, "lisi");
    zs.print(); //print(&zs)
    ls.print();//print(&ls)
    return 0;
}
```

1.5.3 this指针的应用

需要显示使用this指针的常见场景：

- 类中的成员变量和参数变量名字一样，可以通过this指针区分
- 从成员函数中返回调用对象自身(返回自引用)，支持链式调用
- 在成员函数中销毁对象自身(对象自销毁)

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

class Counter{

private:
    int count;
public:
    Counter(int count = 0){
        this->count = count;
    }
    Counter &add(void){
        ++count;
        return *this;
    }
    void print(void){
        cout << count << endl;
    }
    void destroy(void){
        cout <<"this : " << this << endl;
        delete this; //销毁对象本身
    }
};

int main(void){
    Counter cnt;
    cnt.print();
    cnt.add().add().add();
    cnt.print();


    Counter *pcn = new Counter;
    pcn->add();
    pcn->print();
    pcn->destroy();
    cout<<"pcn: " <<pcn <<endl;

    return 0;
}
```

1.6常成员函数

在C++中，为了禁止成员函数修改成员数据的值，可以将它设置为常成员函数。设置方法就是在函数体之前加上const关键字。

```
class X{
    void func(参数1， 参数2， ...) const{

    }
};
```

```
#include <iostream>
```

```
using namespace std;

class Student{
private:
    int age;
    string name;
public:
    Student(int age, const string &name){
        this->age = age;
        this->name = name;
    }
    void whoami(void) const{
        age++;
        cout << "我是: " << name << " 我今年: " << age << endl;
    }

};
int main(void){

    Student s1(22, "张三");
    s1.whoami();

    return 0;
}
```

常函数的实现本质：

常函数中的this指针是常指针，所以不能在常函数中修改成员变量。

```
class A{
public:
    void print (void)  const { ... }//编译前
    void print (const A* this)  { ... }//编译后
};
```

常函数的使用注意事项：

- 常对象只能调用常函数，非常对象既可以调用非常函数 也可以调用常函数
- 函数名和形参表相同的成员函数，常版本和非常版本可以构成重载
    - 常对象只能选择常版本
    - 非常对象优先选择非常版本
- 被mutable修饰的成员可以常函数中修改（了解）

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

class A{
public:
    A(int mm=0, int nn=0):m(mm),n(nn){}

    void fun(void){   //void func(A * this)
        cout <<__func__<< endl;
    }
    void bar(void) const{
        cout << __func__ << endl;
    }
    void fun(void) const {  //void func(const A *this)
        cout <<__func__<< "const" <<endl;
```

```
        m++;
        //n++;//语法错误
    }

private:
    mutable int m;
    int n;
};

int main(void){
    A a;
    a.fun();
    a.bar();
    const A b;
    //b.fun(); //语法错误
    b.bar();
    return 0;
}
```

1.7 析构函数

析构函数是与类同名的另外一个特殊成员函数，作用与构造函数相反，用于对象生存期结束时，完成对象的清理工作。析构函数的名字是：~类名

```
class X{
public:
    X(){}
    ~X(){} //析构函数
};
```

析构函数的特点：

- 无参无返回值
- 不能重载
- 只能由系统调用，不能显示调用
    - 栈对象，离开其作用域时析构函数自动调用
    - 堆对象，执行delete操作时析构函数自动调用

```
#include <iostream>

using namespace std;

class Integer{

public:
    Integer(int i=0){
        m_pi = new int(i);
    }
    ~Integer(void){
        cout << "析构函数" << endl;
        delete m_pi;
    }
    void print(void) const{
        cout <<*m_pi << endl;
    }
private:
```

```
    int *m_pi;
};

int main(void){
    if(1){
        Integer i(100);
        i.print();

        cout << "test1" << endl;
        Integer *pi = new Integer(200);
        pi->print();
        delete pi;  //析构函数被调用
        cout << "test2 " << endl;
    }// i 的生命周期结束  对应的析构函数被调用
    cout << "test3" << endl;
    return 0;
}
```

缺省虚构函数

如果类没有显式定义析构函数，那么编译器会为其提供一个缺省析构函数，缺省析构的函数体为空，在空析构函数体执行完毕后，类中的成员会被自动销毁

- 对基本类型成员变量，什么也不做
- 对类类型成员变量(成员子对象)，将会自动调用相应类的析构函数

```
#include <iostream>

using namespace std;

class A{
public:
    A(void){
        cout << "A(void)" << endl;
    }
    ~A(void){
        cout <<"~A(void)" << endl;
    }
};

class B{
public:
    B(void){
        cout << "B(void)" << endl;
    }
    ~B(void){
        cout << "~B(void)" << endl;
    }
    A m_a; // 成员子对象
};
int main(void){
    B b;
    return 0;
}
```

对象的创建和销毁的过程:

- 对象创建
    - 分配内存
    - 构造成员对象

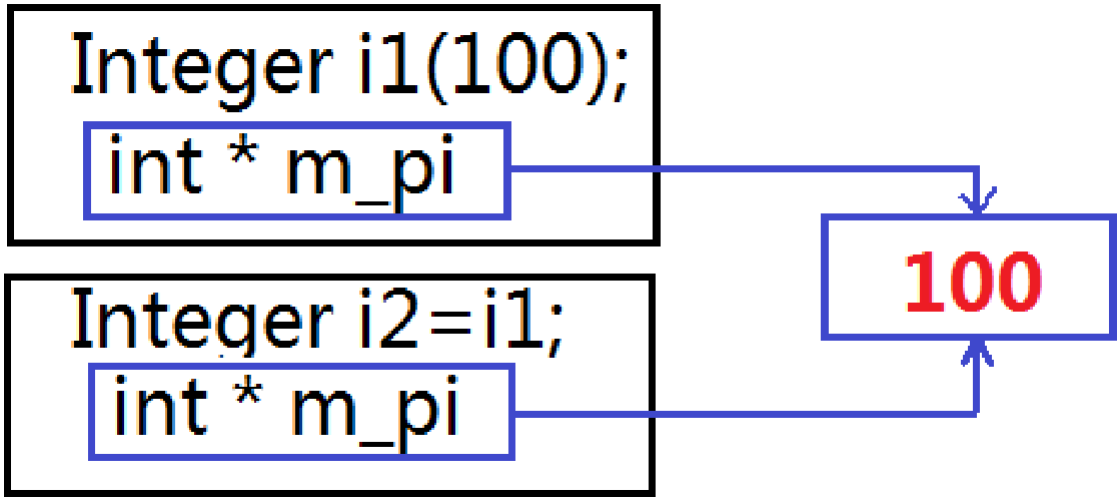
    - 调用构造函数
- 对象销毁
    - 调用析构函数
    - 析构成员对象
    - 释放内存

1.8 拷贝构造与拷贝赋值

1.8.1深拷贝与浅拷贝

```
#include <iostream>

using namespace std;

class Integer{
private:
    int *m_pi ;
public:
    Integer(int i = 0){
        m_pi = new int(i);
    }
    void print(void){
        cout << *m_pi << endl;
    }
    ~Integer(){
        delete m_pi;
    }

};
int main(void){
    Integer i1(100);

    Integer i2 = i1;  //拷贝构造函数
    Integer i3(i1); //拷贝构造函数
    i2.print();
    i3.print();


    return 0;
}
```

以上程序执行会出现dobule free的情况，为什么?



浅拷贝：类中的缺省拷贝构造函数，对指针形式的成员变量按字节复制，而不会复制指针所指向的内容，这种拷贝方式被称为浅拷贝。

深拷贝：为了避免浅拷贝的问题，获得完整意义上的对象副本，必须自己定义拷贝构造函数，针对指针形式的成员变量，实现对指针指向内容的复制，即深拷贝

```
#include <iostream>

using namespace std;

class Integer{
private:
    int *m_pi ;
public:
    Integer(int i = 0){
        m_pi = new int(i);
    }
    void print(void){
        cout << *m_pi << endl;
    }
    ~Integer(){
        delete m_pi;
    }
    Integer(const Integer &that){
        m_pi = new int(*that.m_pi);
    }

};
int main(void){
    Integer i1(100);

    Integer i2 = i1;  //拷贝构造函数
    i2.print();

    return 0;
}
```

1.8.2拷贝赋值

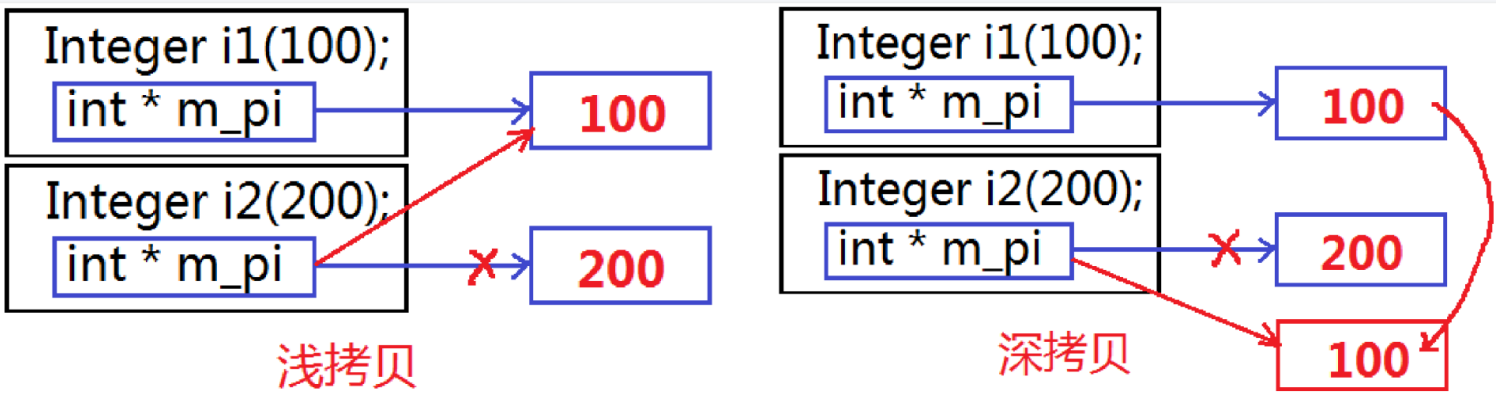
```
#include <iostream>

using namespace std;

class Integer{
private:
    int *m_pi ;
public:
```

```
    Integer(int i = 0){
        m_pi = new int(i);
    }
    void print(void){
        cout << *m_pi << endl;
    }
    ~Integer(){
        delete m_pi;
    }
    Integer(const Integer &that){
        m_pi = new int(*that.m_pi);
    }

};
int main(void){
    Integer i1(100);

    Integer i2 = i1;  //拷贝构造函数
    i2.print();

    i2 = i1;

    return 0;
}
```

当两个对象进行赋值操作时，比如“i2=i1”编译器会将其处理为“i2.operator=(i1) 的成员函数调用形式”,其中“operator=”称为拷贝赋值操作符函数，由该函数完成其赋值运算，其返回结果就是表达式的结果。

如果没有自己定义拷贝赋值操作符函数，编译器会为该类提供缺省的拷贝赋值操作符函数，用于完成两个对象的赋值操作。

但是编译器提供的缺省拷贝赋值函数，和缺省拷贝构造函数类似，也是浅拷贝，有“double free”和“内存泄漏”的问题，这时需要自定义深拷贝赋值函数




```
#include <iostream>

using namespace std;

class Integer{
private:
    int *m_pi ;
public:
    Integer(int i = 0){
        m_pi = new int(i);
    }
    void print(void){
        cout << *m_pi << endl;
    }
    ~Integer(){
        delete m_pi;
    }
    Integer(const Integer &that){
```

```
        m_pi = new int(*that.m_pi);
    }
    Integer &operator=(const Integer &that){
        cout << "operator=" << endl;
        if(this != &that){
            delete m_pi;
            m_pi = new int(*that.m_pi);
        }

        return *this;
    }

};
int main(void){
    Integer i1(100);

    Integer i2 = i1;  //拷贝构造函数
    i2.print();

    i2 = i1;
    return 0;
}
```

1.9 string类编程实例

实现一个字符串类String，为其提供可接受C风格字符串的构造函数、析构函数、拷贝构造函数和拷贝赋值函数

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

class String{
public:
    String(const char *str){
        this->str = new char[strlen(str) + 1];
        strcpy(this->str, str);
    }
    String(const String &that){
        str = new char[strlen(that.str) + 1];
        strcpy(str, that.str);
    }

    void print(void){
        cout << str << endl;
    }
    String &operator=(const String &that){
        if(this != &that){
            /*释放掉this对象原来的堆空间*/
            delete [] str;
            str = new char[strlen(that.str) + 1];
            strcpy(str, that.str);
        }

        return *this;
    }
    ~String(){
        delete []str;
    }

    const char *c_str(){
```

```
        return str;
    }

private:
    char *str;
};

int main(void){

    String s1 = "hello"; //类型转换构造函数
    s1.print();

    String s2 = s1; //拷贝构造函数
    s2.print();

    String s3 = "world"; //类型转换构造函数
    s2 = s3; //拷贝赋值
    s2.print();

    cout << s3.c_str() << endl; //string:: c_str
    return 0;
}
```

1.10 静态成员

- C++为什么需要静态成员

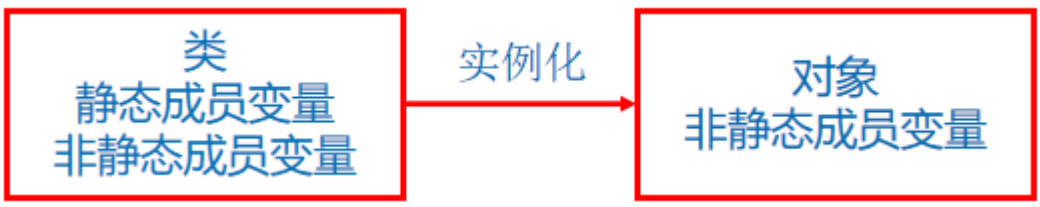
  C语言中可以通过全局变量实现数据共享，在程序的任何位置都可以访问

  C++中希望某个类的多个对象之间实现数据共享，可以通过static建立一个被局限在类中使用的全局资源，该类型资源被称为静态成员。

- 静态成员变量（可以理解为局限在类中使用的全局变量）

  被static修饰的成员变量即为静态成员变量

```
class 类名{
   static 数据类型 变量名;//声明
 };
 数据类型 类名::变量名 = 初值;//定义和初始化
```



  实例化对象时只实现非静态成员变量

  访问方式：

  - 类名::静态成员变量;
  - 对象.静态成员变量;

```
#include <iostream>

using namespace std;

class A{
```

```
public:
    int m_data;
    static int s_data;

    A(int data=0):m_data(data){}
};
int A::s_data = 100; //定义 并 初始化

int main(void){

    cout << A::s_data << endl;

    A a1(123);
    cout << "a1 size =  " << sizeof(a1) << endl;
    cout << a1.s_data << endl;


    A a2(1);
    a2.s_data  = 999;
    cout << a1.s_data << endl;

    return 0;

}
```

- 静态成员函数

被static修饰的成员函数即为静态成员函数。

```
class 类名{
访问控制限定符:
    static 返回类型 函数名(形参表) { ... }
};
```

注意:

- 静态成员函数可以直接定义在类的内部，也可以定义在类的外部，这一点和普通的成员函数没有区别。
- 静态成员函数没有this指针，没有const属性，可以把静态函数理解为被限制在类中使用的全局函数.
- 静态成员函数中只能访问静态成员，但是在非静态成员函数中既可以访问静态成员也可以访问非静态成员
- 静态成员函数和静态成员变量一样，也要受到类的访问控制限定符的约束

在类的外部访问静态成员函数

类名::静态成员函数(实参表);
对象.静态成员函数(实参表);

```
#include <iostream>

using namespace std;

class A{
public:
    int m_data;
    static int s_data;

    A(int data=0):m_data(data){}

    void func1(void){ // func1(A *this)
        cout << m_data << endl;
        cout << s_data << endl;
    }
```

```
    static void func2(){ //静态成员函数
        cout << "静态成员函数" << endl;
        cout << s_data << endl;
        //cout << m_data << endl; //error
    }

};
int A::s_data = 100; //定义 并 初始化

int main(void){

    cout << A::s_data << endl;
    A::func2();
    //A::func1(); //func1(&对象的地址) error
    return 0;
}
```

1.11 友元

类的封装具有信息隐藏能力，但也带来了访问效率的问题.c++通过友元给某些函数一项特权，可以访问类中的私有成员，使用的关键字是friend。

1.11.1友元函数

友元函数可以直接访问类的私有成员

```
class X{
    friend T f(...) ;// 声明f为X类的友元
    ...
};
T f(...){} //友元不是类的成员函数
```

```

```

1.11.2友元类

一个类可以是另一个类的友元，友元类的所有成员函数都是另一个类的友元函数，能够直接访问另一个类的所有成员。

```
#include <iostream>

using namespace std;

class A{
private:
    int x, y;
public:
    A(int i, int j){

        x = i;
        y = j;
    }
    int getx(void){
        return x;
    }
    int gety(void){
        return y;
```

```
    }
    friend class B; //声明B是A的友元类
};
class B{
private:
    int z;
public:
    B(int i = 0){
        z = i;
    }
    int add(const A& a){
        return a.x + a.y + z;
    }
    int sub(const A& a){
        return a.x - a.y - z;
    }
};

int main(void){

    A a(2,3);
    B b(4);

    cout << b.add(a) << endl;
    cout << b.sub(a) << endl;

    return 0;
}
```

友元类不是双向的：B是A的友元类， 不意味着A也是B的友元类

1.11.3友元成员函数

对一个类，可以指定它的某个成员函数是另一个类的友元，也就是友元成员函数。

```
#include <iostream>

using namespace std;

class A;
class B{
private:
    int z;
public:
    B(int i = 0){
        z = i;
    }
    int add(const A&);
    int sub(const A&);
};
class A{
private:
    int x, y;
public:
    A(int i, int j){

        x = i;
        y = j;
    }
    int getx(void){
```

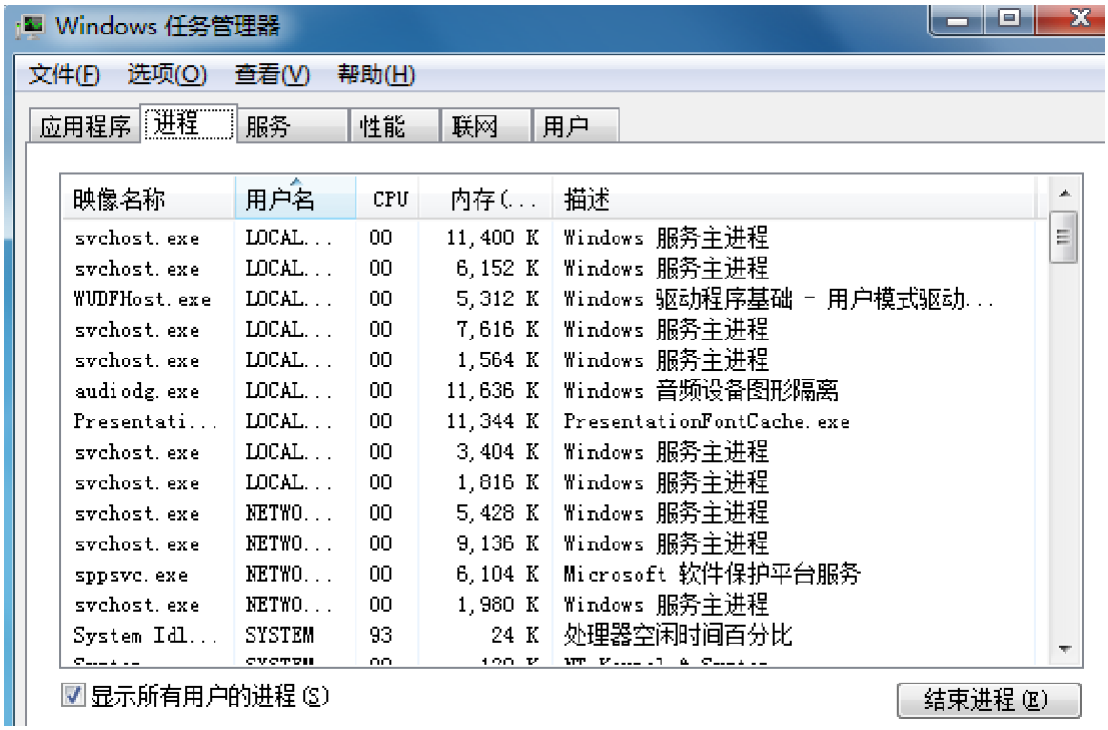
```
        return x;
    }
    int gety(void){
        return y;
    }
    //friend class B; //声明B是A的友元类
    friend int B::add(const A& a);
};

int B::add(const A& a){
    return a.x + a.y + z;
}

int B::sub(const A& a){
//  return a.x - a.y - z;
}
int main(void){

    A a(2,3);
    B b(4);

    cout << b.add(a) << endl;
    cout << b.sub(a) << endl;

    return 0;
}
```

1.12 单例模式

单例模式(Singleton Pattern，也称为单件模式)，使用最广泛的设计模式之一。其意图是保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。

面向对象编程中，每个对象都应该抽象代表一个设备，并通过对象完成对某个具体设备的管理和维护。

对于有些类只能有一个实例很重要，例如打印机管理器、设备管理器、任务管理器等。



实现单例模式的三个主要步骤：

- 私有化构造函数

```
class Singleton{
private:
    Singleton(void){...}
    Singleton(const Signleton &that){}
    ...
};
```

- 使用静态成员变量维护唯一的单例对象

```
class Singleton{
private:
    Singleton(void){...}
    Singleton(const Signleton &that){}
    ...

    static Singleton s_instance;
};

Singleton Singleton::s_instance;
```

- 定义静态成员函数用于获取单例对象

```
class Singleton{
private:
    Singleton(void){...}
    Singleton(const Signleton &that){}
    ...

    static Singleton s_instance;
public:
    static Singleton& getInstance(void){
        return s_instance;
    }
};
Singleton Singleton::s_instance;
```

1.12.1饿汉式

加载进程时即完成创建（饿），用不用都创建。

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

class Signleton{
private:
    int m_i;
    Signleton(int i=0){
        m_i = i;
    }
    Signleton(const Signleton& that){}

    static Signleton m_instance;
public:
    static Signleton& getInstance(){
```

```
        return m_instance;
    }
    void print(){
        cout << m_i << endl;
    }
};
Signleton Signleton::m_instance = 111;

int main(void){

    Signleton& s1 = Signleton::getInstance();
    Signleton& s2 = Signleton::getInstance();
    Signleton& s3 = Signleton::getInstance();


    //Signleton s4 = 12; //error

    cout << &s1 << endl;
    cout << &s2 << endl;

    s1.print();
    s2.print();

    return 0;
}
```

1.12.2 懒汉式

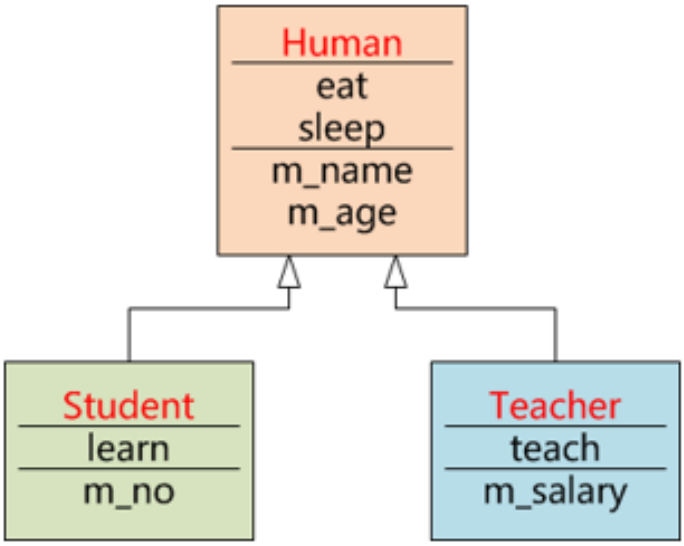
用时再创建（懒），不用再销毁

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


class Signleton{
private:
    int m_i;
    static int m_count ; //记录对象的引用次数
    static Signleton *m_instance;
    Signleton(int i = 0){
        m_i = i;
        cout << "constructor " << endl;
    }
    Signleton(const Signleton& that){}
public:
    static Signleton& getInstance(void){
        if(m_instance == NULL){
            m_instance = new Signleton(123);
        }
        m_count++;
        return *m_instance;
    }
    void release(){
        m_count--;
        if(m_count == 0){
            delete m_instance;
            m_instance = NULL;
        }
    }
    ~Signleton(){
```

```
        cout << "destroy" << endl;
    }
};
Signleton * Signleton::m_instance = NULL;
int Signleton::m_count = 0;

int main(void){
    //Signleton s1; //error

    Signleton& s1 = Signleton::getInstance();
    Signleton& s2 = Signleton::getInstance();
    Signleton& s3 = Signleton::getInstance();

    cout << &s1 << "  " << &s2 << " " << &s3 << endl;
    s1.release();
    s2.release();
    s3.release();

    return 0;
}
```

2 继承

继承，基于一个已有类创建新类，使新类与已有类具有同样的功能，即新类具有已有类相同的数据成员和成员函数。继承是代码重用的基本工具。已有类称为基类（父类 /超类），新类称为派生类（子类）。



注意：

- 基类的构造函数和析构函数不能继承
- 基类的友元函数不能继承
- 静态数据成员和静态成员函数不能继承

2.1 继承的方式

C++的继承可以分为公有继承、保护继承和私有继承。不同继承方式会不同程度影响基类成员在派生类的访问权限。

语法格式：

```
class 派生类名：继承方式 基类名{
    派生类成员声明与定义
};
```

访问控制限定符	访问控制属性	内部	子类	外部	友元
public	公有成员	OK	OK	OK	OK
protected	保护成员	OK	OK	NO	OK
private	私有成员	OK	NO	NO	OK

基类中的	在公有子类中变成	在保护子类中变成	在私有子类中变成
公有成员	公有成员	保护成员	私有成员
保护成员	保护成员	保护成员	私有成员
私有成员	私有成员	私有成员	私有成员

2.1.1公有继承

继承方式为public的继承称为公有继承，在这种继承中，基类成员的访问权限在派生类中保持不变。

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

class Base{
private:
    int m_a;
protected:
    int m_b;
public:
    int m_c;
    Base(int a=1, int b=2, int c=3){
        m_a = a;
        m_b = b;
        m_c = c;
    }
    int geta(){
        return m_a;
    }
};
```

```
class Derived: public Base{
public:
    void print(){
        //cout << m_a << endl; //error
        cout << geta() << endl;
        cout << m_b << endl;
        cout << m_c << endl;
    }
};

int main(void){
    Derived test;

    //cout << test.m_a << endl; //  error private
    //cout << test.m_b << endl;  // error protected
    cout << test.m_c << endl;
    return 0;
}
```

2.1.2 保护继承

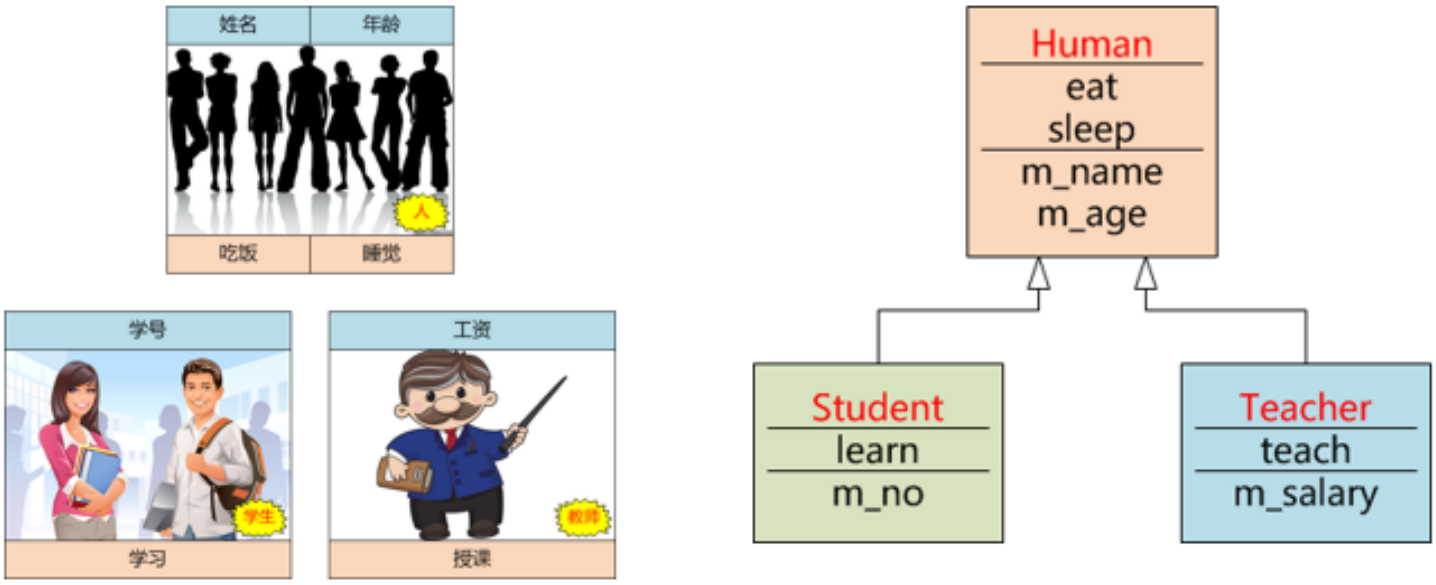
```
#include <iostream>
using namespace std;

class Base{
private:
    int m_a;
protected:
    int m_b;
public:
    int m_c;
    Base(int a=1, int b=2, int c=3){
        m_a = a;
        m_b = b;
        m_c = c;
    }
    int geta(){
        return m_a;
    }
};

class Derived: protected Base{
public:
    void print(){
        //cout << m_a << endl; //error
        cout << geta() << endl;
        cout << m_b << endl;
        cout << m_c << endl;
    }
};

int main(void){
    Derived test;

    //cout << test.m_a << endl; //  error private
    //cout << test.m_b << endl;  // error protected
    cout << test.m_c << endl; //protected继承 子类中该变量为protected
    return 0;
}
```

2.1.3 私有继承

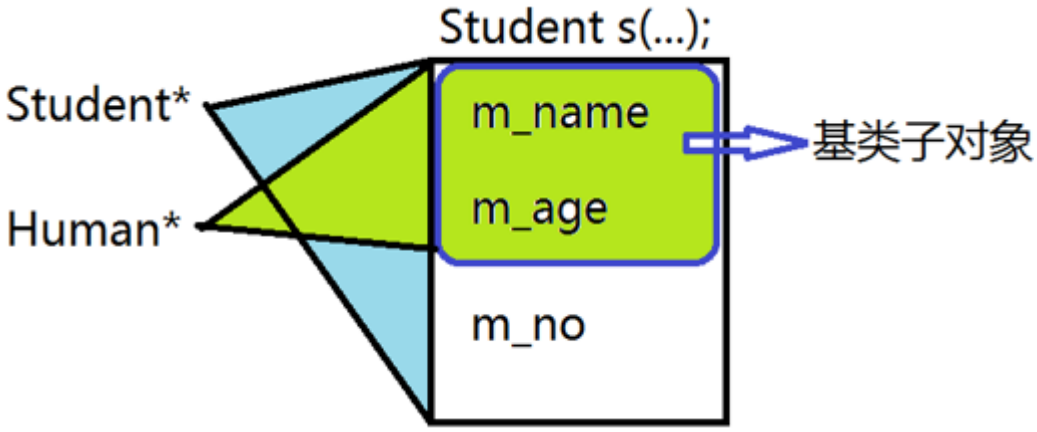
```
#include <iostream>
using namespace std;

class Base{
private:
    int m_a;
protected:
    int m_b;
public:
    int m_c;
    Base(int a=1, int b=2, int c=3){
        m_a = a;
        m_b = b;
        m_c = c;
    }
    int geta(){
        return m_a;
    }
};

class Derived: private Base{
public:
    void print(){
        //cout << m_a << endl; //error
        cout << geta() << endl;
        cout << m_b << endl;
        cout << m_c << endl;
    }
};

int main(void){
    Derived test;

    //cout << test.m_a << endl; //  error private
    //cout << test.m_b << endl;  // private继承   error private
    //cout << test.m_c << endl; //private继承 子类中该变量为private
    return 0;
}
```

2.2 基类与派生类的关系

2.2.1 向上造型和向下造型



子类对象会继承基类的属性的行为，任何时候子类对象都可以被当做基类类型的对象，通过子类对象可以直接访问基类中的成员，如同是基类对象在访问它们一样



向上造型(upcast)：将子类类型的指针或引用转换为基类类型的指针或引用；这种操作性缩小的类型转换，在编译器看来是安全的，可以隐式转换。

向下造型(downcast)：将基类类型的指针或引用转换为子类类型的指针或引用；这种操作性放大的类型转换，在编译器看来是危险的，不能隐式转化，但是可以显式转换

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

class Human{
private:
    int m_private;
protected:
    string m_name;
    int m_age;
    const int& get(void){
        return m_private;
    }
public:
    Human(const string &name, int age){
        m_name = name;
        m_age = age;
        m_private = 1234;
    }
    void eat(const string& food){
        cout << "我在吃: " << food << endl;
    }
    void sleep(int hour){
        cout << "我睡了" << hour << "小时" <<endl;
```

```
    }
};
class Student: public Human{
private:
    int m_no; //学号
public:
    Student(const string& name, int age, int no):Human(name, age){
        m_no = no;
    }
    void who(void){
        cout << "我叫: " << m_name << ", 今年" <<m_age<<"岁，学号是: "<<m_no << endl;
        //cout << m_private << endl; //error
        cout << get() << endl;
    }
    void learn(const string& course){
        cout << "我在学" << course << endl;
    }
};
class Teacher: public Human{
private:
    int m_salary;
public:
    Teacher(const string& name, int age, int salary):Human(name, age),m_salary(salary){
    }
    void teach(const string& course){
        cout << "我正在讲 " << course << endl;
    }
    void who(void){
        cout << "我叫 "<<m_name << ",今年" << m_age << "岁, 工资是" << m_salary << endl;
    }
};

int main(void){
    Student s("张飞", 28, 100011);
    cout << "sizeof(s) = "<<sizeof(s) << endl;
    s.who();
    s.eat("宫保鸡丁");
    s.sleep(8);
    s.learn("C++编程");

    Teacher t("诸葛亮", 34, 200000);
    t.who();
    t.teach("嵌入式");
    t.sleep(7);
    t.eat("汉堡");

    // Student * -----> Human *:向上造型
    Human *ph = &s;
    ph->eat("香蕉");
    ph->sleep(10);
    //ph->who();  //error

    // Human * --------> Student *: 向下造型（合理）
    Student *ps = static_cast<Student *>(ph);
    ps->who();

    Human h("赵云", 22);
    //Human * -------> Student *: 向下造型 (不合理)
    Student *ps2 = static_cast<Student *>(&h);
    ps2->who();

    return 0;
}
```

2.2.2 成员函数的重定义（名字隐藏）

重定义: 简单的说就是子类中定义了和父类的同名函数，对父类的成员函数造成了隐藏（讲多态时会做进一步的限定）

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

class Base{
private:
    int x;
public:
    void set(int i){
        x = i;
    }
    void print(){
        cout << "Base class " << "x= " << x << endl;
    }
};
class Derived: public Base{
private:
    int m, n;
public:
    void set(int p, int k){
        m = p;
        n = k;
    }
    void print(){
        Base::print();
        cout << "Derived class "<< "m = "<< m <<", n=" << n << endl;
    }

};
int main(void){

    Derived d;
    d.set(10,20);
    //d.set(100); // error  名字隐藏
    d.Base::set(100);
    d.print();
    return 0;
}
```

2.3派生类的构造与析构

2.3.1构造

- 如果子类构造函数没有显式指明基类部分(基类子对象)的初始化方式，那么编译器将会自动调用基类的无参构造函数来初始化基类子对象。
- 如果希望以有参的方式来初始化基类部分，那么必须使用初始化列表来显式指明
- 子对象构造顺序
    - 分配内存
    - 构造基类子对象(按继承表顺序)
    - 构造成员子对象(按声明顺序)

- 执行子类构造函数代码

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

class Member{
private:
    int m_k;
public:
    Member(){
        cout << "Member()" << endl;
        m_k = 0;
    }
    Member(int k){
        cout << "Member(int)" << endl;
        m_k = k;
    }
};

class Base{
private:
    int m_i;
public:
    Base(){
        cout << "Base()" << endl;
        m_i = 0;
    }
    Base(int i){
        cout << "Base(int)" << endl;
        m_i = i;
    }
};

class Derived:public Base{
private:
    int m_j;
    Member m_m;
public:
    Derived():Base(100),m_m(200){
        cout << "Derived()" << endl;
    }
    Derived(int i, int j):Base(i){
        cout << "Derived(int, int)" << endl;
        m_j = j;
    }
};
int main(void){
    Derived d1;
    Derived d2(1,2);
    return 0;
}
```

2.3.2 析构

- 子类的析构函数，无论自己定义的，还是编译器缺省提供，都会自动调用基类的析构函数，完成基类子对象的销毁。
- 子类对象销毁过程
    - 执行子类析构函数代码
    - 析构成员子对象(按声明逆序)

- 析构基类子对象(按继承表逆序)
- 释放内存

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

class Member{
private:
    int m_k;
public:
    Member(){
        cout << "Member()" << endl;
        m_k = 0;
    }
    Member(int k){
        cout << "Member(int)" << endl;
        m_k = k;
    }
    ~Member(){
        cout << "~Member()" << endl;
    }
};

class Base{
private:
    int m_i;
public:
    Base(){
        cout << "Base()" << endl;
        m_i = 0;
    }
    Base(int i){
        cout << "Base(int)" << endl;
        m_i = i;
    }
    ~Base(){
        cout << "~Base()" << endl;
    }
};

class Derived:public Base{
private:
    int m_j;
    Member m_m;
public:
    Derived():Base(100),m_m(200){
        cout << "Derived()" << endl;
    }
    Derived(int i, int j):Base(i){
        cout << "Derived(int, int)" << endl;
        m_j = j;
    }
    ~Derived(){
        cout << "~Derived()" << endl;
    }
};
int main(void){
    Derived d1;
    //Derived d2(1,2);
    return 0;
}
```

2.4 多重继承

C++允许一个类从一个或多个基类派生。如果一个类只有一个基类，称为单一继承。如果一个类具有两个或两个以上的基类，就称为多重继承。

```
class 派生类名：继承方式 基类名1， 继承方式 基类名2，...{};
```

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

class Phone{
private:
    string m_number;
public:
    Phone(const string& number){
        m_number = number;
    }
    void call(const string& number){
        cout << m_number << "打给：  " << number << endl;
    }
};
class Player{
public:
    Player(const string& media){
        m_media = media;
    }
    void play(const string& music){
        cout << m_media << "正在播放: " << music << endl;
    }
private:
    string m_media ; //播放器的名称
};
class Computer{
private:
    string m_os; //使用的操作系统
public:
    Computer(const string& os):m_os(os){
    }
    void run(const string& app){
        cout << "在" << m_os << "正在运行: " << app << endl;
    }
```

```
};
/*典型的多重继承*/
class SmartPhone:public Phone, public Player, public Computer{
public:
    SmartPhone(const string& number, const string& media, const string& os):Phone(number), Player(media),
Computer(os){

    }
};


int main(void){

    SmartPhone huawei("13988888888", "MP4", "鸿蒙");

    huawei.call("010-12345");
    huawei.play("我和我的祖国");
    huawei.run("王者荣耀");

    return 0;
}
```

2.4.1名字冲突

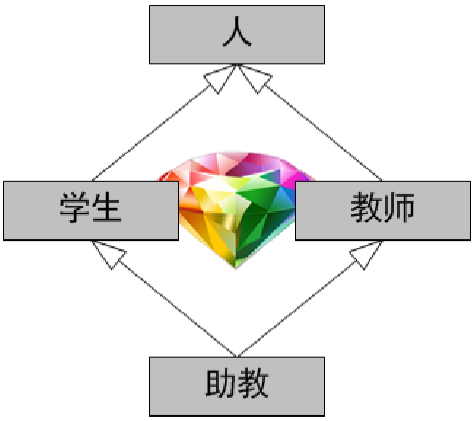
当两个不同基类拥有同名成员时，容易产生名字冲突问题。

使用域限定符解决。

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

class A{
public:
    void func(void){
        cout << "A::func()" << endl;
    }
};
class B{
public:
    void func(void){
        cout << "B::func()" << endl;
    }
};
class C: public A, public B{
};
int main(void){
    C test;
    test.A::func();
    return 0;
}
```

2.4.2 钻石继承与虚继承

钻石继承，一个派生类继承的多个基类又源自一个公共的祖先（公共基类）。



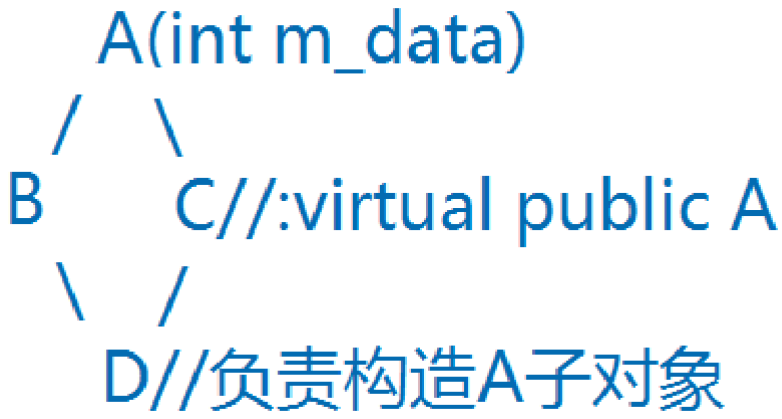

```
#include <iostream>
using namespace std;

class A{
protected:
    int m_data;
public:
    A(int data){
        m_data = data;
        cout << "A(int)" << endl;
    }
};
class B: public A{
public:
    B(int data):A(data){
        cout << "B(int)" << endl;
    }
    void set(int data){
        m_data = data;
    }

};
class C: public A{
public:
    C(int data):A(data){
        cout << "C(int)" << endl;
    }
    int get(void){
        return m_data;
    }
};

class D: public B, public C{
public:
    D(int data): B(data), C(data){
        cout << "D(int)" << endl;
    }

};
int main(void){

    D d(100);

    cout << sizeof(d) << endl;

    cout << d.get() << endl; // 100
    d.set(200);
    cout << d.get() << endl; //100
```

```
    return 0;
}
```

解决方式：虚继承

虚继承语法：

- 在继承表使用virtual关键字修饰
- 位于继承链末端子类负责构造公共基类子对象




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

class A{
protected:
    int m_data;
public:
    A(int data){
        m_data = data;
        cout << "A(int)" << endl;
    }
};
class B: virtual public A{
public:
    B(int data):A(data){
        cout << "B(int)" << endl;
    }
    void set(int data){
        m_data = data;
    }

};
class C: virtual public A{
public:
    C(int data):A(data){
        cout << "C(int)" << endl;
    }
    int get(void){
        return m_data;
    }
};

class D: public B, public C{
public:
    D(int data): B(data), C(data), A(data){
        cout << "D(int)" << endl;
    }
```

```
};
int main(void){
    D d(100);

    cout << sizeof(d) << endl;

    cout << d.get() << endl; // 100
    d.set(200);
    cout << d.get() << endl; //200
    return 0;
}
```

2.5 继承与组合

继承与组合是C++实现代码重用的两种主要方法。

继承是Is-a的关系，比如水果和梨

组合是Has-a的关系，图书馆有图书。

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

class vehicles{ //交通工具
public:
    void load(const string& goods){
        cout << "装载"  << goods << endl;
    }
};

class tyre{ //轮胎
public:
    void run(const string& dest){
        cout << "转动方向: " << dest << endl;
    }
};
class car: public vehicles{

public:
    tyre wheel;

};

int main(void){
    car c1;

    c1.load("电脑");
    c1.wheel.run("北京");
    return 0;
}
```

2.6 多文件编程实例

一般将类的声明放在.h文件中，类中成员函数的定义放在.cpp文件中。

```
/*person.h*/
#ifndef __PERSON_H__
#define __PERSON_H__
```

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

class person{
private:
    int age;
    string name;
public:
    person(int age, const string& name);
    void whoami(void);
};


#endif
```

```cpp
/*person.cpp*/
#include "person.h"

person::person(int age, const string& name){
    this->age = age;
    this->name = name;
}

void person::whoami(void){
    cout << "我是: " << name << endl;
}
```

```cpp
/*student.h*/
#ifndef __STUDENT_H__
#define __STUDENT_H__

#include "person.h"

class student: public person{
private:
    float score;
public:
    student(int age, const string& name, float score);
    void whoami(void);
};

#endif
```

```cpp
/*student.cpp*/
#include "student.h"

student::student(int age, const string& name, float score):person(age,name){
    this->score = score;
}
void student::whoami(void){
    person::whoami();
    cout << "我的成绩是: " << score << endl;
}
```

```
/*main.cpp*/
#include "student.h"

int main(void){
    student s1(22, "刘备", 81.5);
    s1.whoami();

    return 0;
}
```

```
g++ person.cpp student.cpp main.cpp
./a.out
```

3 多态

多态性是面向对象程序设计语言的又一重要特征，多态（polymorphism）通俗地讲，就是用一个相同的名字定义许多不同的函数，这些函数可以针对不同数据类型实现相同或相似的功能，即所谓的“一个接口，多种实现”。

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

class Shape{
public:
    virtual void draw(void){
        cout << "draw shape" << endl;
    }
};
class Rect:public Shape{
public:
    void draw(void){
        cout << "draw Rect" << endl;
    }

};
class Circle: public Shape{
public:
    void draw(void){
        cout << "draw Circle" << endl;
    }

};
class Ellipse: public Shape{
public:
    void draw(void){
        cout << "draw Ellipse" << endl;
    }

};
int main(void){
/*
    Ellipse e;
    e.draw();
    e.Shape::draw();
*/
    Shape *buf[128] = {0};

    buf[0] = new Rect;
    buf[1] = new Circle;
```

```
    buf[2] = new Ellipse;

    for(int i=0; buf[i] != NULL;  i++){
        buf[i]->draw();
    }

    return 0;
}
```

3.1虚函数

被**virtual**关键字修饰的成员函数称为**虚函数。**

如果将基类中的某个成员函数声明为虚函数，那么子类中与该函数具有相同原型的成员函数就也是虚函数，并且对基类中版本形成**覆盖**，即**函数重写**.

如果子类提供了对基类虚函数有效的覆盖，那么通过指向子类对象的基类指针，或者通过引用子类对象基类引用，调用该虚函数，实际被执行将是子类中的覆盖版本，而不再是基类中原始版本，这种语法现象被称为**多态**.

多态的意义在于，一般情况下，调用哪个类的成员函数由调用者指针或者引用本身类型决定的， 而有了多态，调用哪个类的成员函数由 调用者指针或者引用实际目标对象的类型决定。

这样一来，源自同一种类型的同一种激励，竟然可以产生多种不同的响应，也就是对于同一个函数调用，能够表达出不同的形态，即为多态。

虚函数覆盖的条件：

- 只有类中的成员函数才能声明为虚函数，而全局函数、静态成员函数、构造函数都不能被声明为虚函数
- 只有在基类中以virtual关键字声明的虚函数，才能作为虚函数被子类覆盖，而与子类中的virtual关键字无关
- 虚函数在子类中的版本和基类中版本要具有相同的函数签名，即函数名、参数表、常属性一致
- 如果基类虚函数返回基本类型的数据，那么子类中的版本必须返回相同类型的数据；如果基类虚函数返回类类型指针(A)*或引用(A&)，那么允许子类中的版本返回其子类类型指针(B)*或引用(B&)

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

class A{};
class B:public A{};

class Base{
public:
    virtual void func(void){
        cout << "Base func" << endl;
    }
    virtual A* foo(void){
        cout << "Base foo" << endl;
    }
};
class Derived: public Base{

    void func(void) {
        cout <<"Derived func" << endl;
    }
    B* foo(void){
        cout << "Derived foo" << endl;
    }
};

int main(void){
    Derived d1;
    Base *pd1 = &d1;
    pd1->func();
```

```
    Base& pd2 = d1;
    pd2.foo();
    return 0;
}
```

产生多态的条件：

- 除了要满足函数重写的语法要求，还必须是通过指针或引用调用虚函数，才能表现出来

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

class A{};
class B:public A{};

class Base{
public:
    virtual void func(void){
        cout << "Base func" << endl;
    }
    virtual A* foo(void){
        cout << "Base foo" << endl;
    }
};
class Derived: public Base{

    void func(void) {
        cout <<"Derived func" << endl;
    }
    B* foo(void){
        cout << "Derived foo" << endl;
    }
};

int main(void){
    Derived d1;

    Base b = d1;
    b.func(); //调用 base中的func

    return 0;
}
```

- 调用虚函数的指针也可以是this指针，当使用子类对象调用基类中的成员函数时，该函数里面this指针将是一个指向子类对象的基类指针，再通过this去调用满足重写要求的虚函数同样可以表现多态的语法特性

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

class A{};
class B:public A{};

class Base{
public:
    virtual void func(void){
        cout << "Base func" << endl;
    }
    A* foo(void){
        //this->func();
```

```
        func();
    }
};
class Derived: public Base{

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

int main(void){
    Derived d1;
    d1.foo();
    return 0;
}
```

3.2 纯虚函数和抽象类

3.2.1 纯虚函数

如果一个虚函数仅表达抽象的行为，没有具体的功能，即只有声明没有定义，这样的虚函数被称为**纯虚函数**或**抽象方法**

```
 class 类名 {
    public:
        virtual 返回类型 函数名 (形参表) = 0;
    };
```

假设有图形类Figure, 设计计算面积的成员函数area()。Figure只是一个纯抽象意义上得概念，不存在计算面积或体积的具体方法，所以只能将成员函数area()设计为纯虚函数。

```
#include <iostream>
using namespace std;
class Figure{
protected:
    double x, y;
public:
    void set(double i, double j){
        x = i;
        y = j;
    }
    virtual void area()=0;
};
```

3.2.2 抽象类

如果类中包含了纯虚函数，那么这个类就是**抽象类**,抽象类只能最为其它类的基类，不能用来建立对象。

如果类中的所有成员函数都是纯虚函数则可以称为**纯抽象类**

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

class Shape{
public:
    virtual void draw(void) = 0;
};
```

```
class Rect:public Shape{
public:
    void draw(void){
        cout << "draw Rect" << endl;
    }

};
class Circle: public Shape{
public:
    void draw(void){
        cout << "draw Circle" << endl;
    }

};
class Ellipse: public Shape{
public:
    void draw(void){
        cout << "draw Ellipse" << endl;
    }

};
int main(void){
/*
    Ellipse e;
    e.draw();
    e.Shape::draw();
*/
    //Shape  s1; //error
    Shape *buf[128] = {0};

    buf[0] = new Rect;
    buf[1] = new Circle;
    buf[2] = new Ellipse;

    for(int i=0; buf[i] != NULL;  i++){
        buf[i]->draw();
    }

    return 0;
}
```

3.3 虚析构函数

C++允许将析构函数定义为虚函数，为什么?

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


class Base{
public:
    Base(){
        cout << "Base 中通过 new 申请100个字节内存空间" << endl;
    }
    ~Base(){
        cout << "~Base 中通过 delete释放100个字节内存空间" << endl;
    }

};
class Derived:public Base{
```

```
public:
    Derived(){
        cout << "Derived 中通过new 申请200个字节内存空间" << endl;
    }
    ~Derived(){
        cout << "~Derived 中通过delete 释放200个字节内存空间" << endl;
    }
};

int main(void){

    Base *pb = new Derived;

    delete pb; //只调用了基类的析构函数  造成内存泄露
    return 0;
}
```

如何解决该问题？将基类析构函数定义为虚函数

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


class Base{
public:
    Base(){
        cout << "Base 中通过 new 申请100个字节内存空间" << endl;
    }
    virtual ~Base(){
        cout << "~Base 中通过 delete释放100个字节内存空间" << endl;
    }

};
class Derived:public Base{
public:
    Derived(){
        cout << "Derived 中通过new 申请200个字节内存空间" << endl;
    }
     ~Derived(){
        cout << "~Derived 中通过delete 释放200个字节内存空间" << endl;
    }
};

int main(void){

    Base *pb = new Derived;

    delete pb; //只调用了基类的析构函数  造成内存泄露
    return 0;
}
```

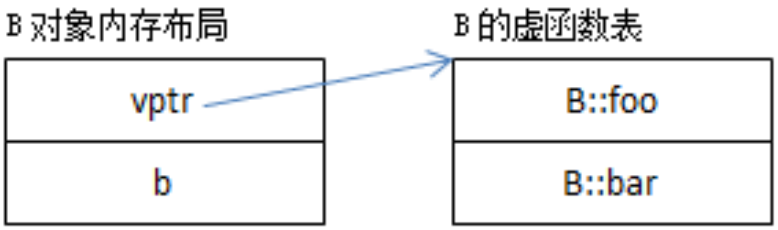
3.4 虚函数的实现技术

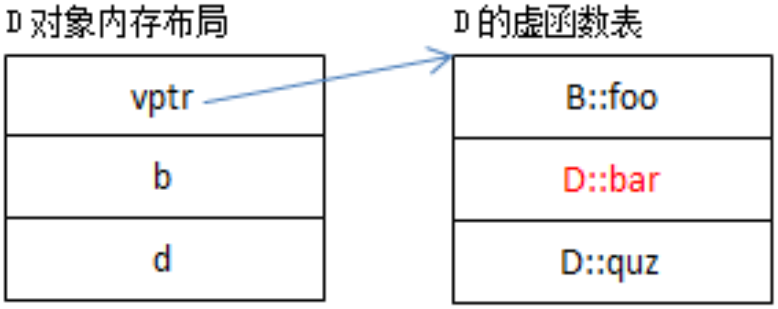
```
#include <iostream>
using namespace std;

class A{
public:
```

```
    int m;
};
class B{
public:
    int m;
    virtual void foo(void){
    }
};

int main(void){
    A a;
    B b;
    cout << "a size : " << sizeof(a) << endl;
    cout << "b size : " << sizeof(b) << endl;
    return 0;
}
```

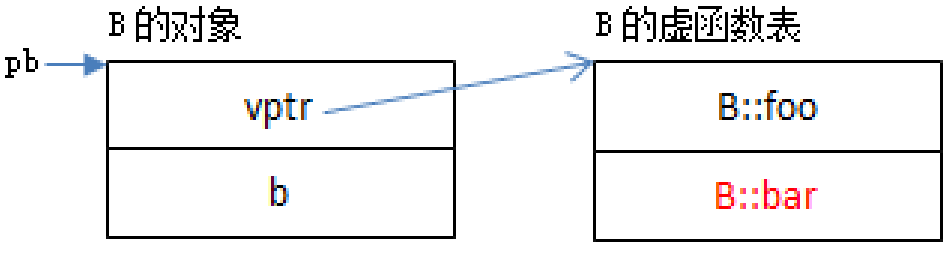
每一个含有虚函数（无论是其本身的，还是继承而来的）的类都至少有一个与之对应的虚函数表，其中存放着该类所有的虚函数对应的函数指针。

```
class B {
    long b;
    virtual void foo() {}
    virtual void bar() {}
};
class D : public B {
    long d;
    virtual void bar() {}
    virtual void quz() {}
};
```

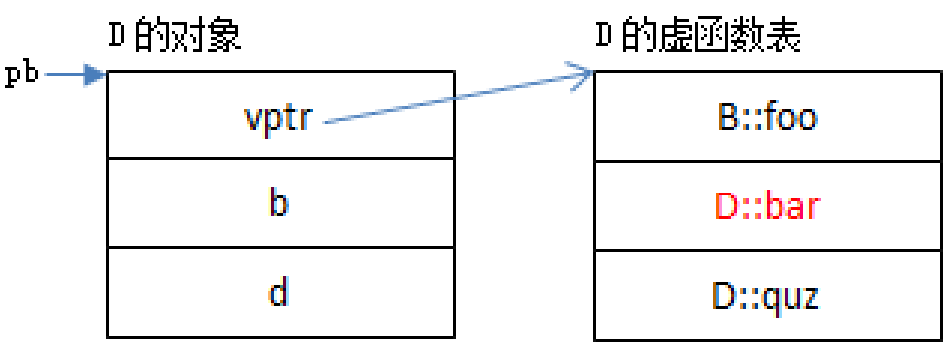




当编译器编译以下test函数时只知道pb是B*类型的指针，并不知道它指向的具体对象类型： pb可能指向的是B的对象，也可能指向的是D的对象。 只有当程序执行过程中给test函数传递了具体参数才能确定pb指向了哪个对象，从而确定访问哪个虚表，从而实现了多态。




```
void test(B* pb) {
    pb->bar();
}
```



3.5 运行时类型信息

运行时类型信息（Run-time Type Information,RTTI）提供了在程序运行时刻确定对象类型的方法，是面向对象程序语言为解决多态问题而引入的一种语言特性。由于多态的要求，C++指针或引用可能与他们实际代表的类型不一致（如基类指针可以指向派生类对象），当将一个多态指针转换为其实际指向类型对象时，就需要知道对象类型信息。

在C++中，用于支持RTTI的运算符有：dynamic_cast, typeid, type_info.

3.5.1 typeid和type_info

- typeid操作符既可用于类型也可用于对象，返回typeinfo对象的常引用，用于表示类型信息
- typeinfo类的成员函数name()，可以获取字符串形式的类型信息

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

int  main(void){

    int i = 123;

    cout << typeid(i).name() << endl;
    cout << typeid(int).name() << endl;


    int * a1[10];
    int (*a2)[10];

    cout << typeid(a1).name() << endl;
    cout << typeid(a2).name() << endl;


    return 0;
}
```

- typeinfo类支持“==”和“!=”操作符，可直接用于类型相同与否的判断，如果类型之间存在多态的继承关系，typeid还可以利用多态的特性确定实际对象的类型

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

class A{
public:
    virtual void foo(void){}
};
class B: public A{
    void foo(void){}
};

class C: public A{
    void foo(void){}
};

void func(const A& a){
    if(typeid(a) == typeid(B)){
        cout << "B" << endl;
    }
    else if(typeid(a) == typeid(C)){
        cout << "C" << endl;
```

```
    }

}
int  main(void){
    B b;
    C c;
    func(b);
    func(c);


    return 0;
}
```

3.5.2 dynamic_cast

强制类型转换运算符，主要用于具有多态继承关系父子类指针或引用之间的显式转换。

语法格式：

```
dynamic_cast <目标类型> (表达式)
```

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

class Base{
public:
    virtual ~Base(){}
};
class Derived: public Base{
public:
    void f(){
        cout << "Derived f()" << endl;
    }
};
int  main(void){
    Base *pb, b;
    Derived *pd, d;

    pd = &d;
    pb = pd; //Base * <---------- Derived * 向上造型 隐式转换 编译阶段完成

    pb = dynamic_cast<Base *> (&d); //显示转换  向上造型 运行时完成的

    class C{
    public:
        virtual void func(){}
    };

    C c;
    //pb = &c; //error
    pb = dynamic_cast<Base *> (&c);
    if(pb){
        cout << "dynamic_cast ok" << endl;
    }
    else{
        cout << "dynamic_cast failed" << endl;
    }

    return 0;
}
```

向下造型时，动态类型转换会对所需转换的基类指针或引用做检查，如果其目标确实为期望得到的子类类型的对象，则转换成功，否则转换失败

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

class Base{
public:
    virtual ~Base(){}
};
class Derived: public Base{
public:
    void f(){
        cout << "Derived f()" << endl;
    }
};
int  main(void){
    Base *pb, b;
    Derived *pd, d;

    pb = &b;

    /*
     *语法上不报错：  具有父子关系 具有多态特性
     *该转换不合理  在运行期间完成转换  转换失败  C++认为不安全
     * */
    pd = dynamic_cast<Derived *>(pb);
    if(pd) cout << "ok" << endl;
    else cout << "error" << endl;


    //pd = reinterpret_cast<Derived *>(pb);
    //pd = (Derived *)pb; //C风格强转  可以编译通过
    //pd->f(); //有问题


    pb = &d;  //向上造型
    pd = dynamic_cast<Derived *>(pb); //编译通过    也是合理转换
    if(pd) cout << "ok" << endl;
    else cout << "error" << endl;

    return 0;
}
```

4 运算符重载

4.1 什么是运算符重载

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

class Complex{
private:
    double r;
    double i;
public:
```

```
    Complex(double r, double i){
        this->r = r;
        this->i = i;
    }
    void print(void){
        cout << r << " + " << i << "i" << endl;
    }
};

int main(void){
    Complex a(1,2);
    Complex b(3,4);
    a.print();
    b.print();

    int x = 10;
    int y = 20;
    int z = x + y;

    a+b; //想像x+y 实现Complex对象加运算  +运算符新的逻辑功能  要实现+运算符的重载

    return 0;
}
```

4.2 双目运算符重载

双目运算符：有左右两个操作数的操作符 L#R

- 算术运算：*、/、%、+、-
- 关系运算：>、>=、<、<=、==、!=
- 逻辑运算：&&、||
- 位运算：&、|、^、<< 、>>
- 赋值与复合赋值：=、+=、-=、*=、/=、%=、&=、|=、^=、<<= 、>>=
- ... ...

注意：

- 表达式结果是右值
- 左右操作数既可以是左值也可以是右值

实现方式：

- 成员函数形式：L.operator#(R)
- 友元函数形式：operator#(L,R)

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

class Complex{
private:
    double r;
    double i;
public:
    Complex(double r, double i){
        this->r = r;
        this->i = i;
    }
    void print(void){
        cout << r << " + " << i << "i" << endl;
    }
    const Complex operator+(const Complex& c){
        Complex tmp(r+c.r, i+c.i);
```

```
        return tmp;
    }
    friend const Complex operator-(const Complex& l , const Complex& r);
};
const Complex operator-(const Complex& l, const Complex& r){
    Complex tmp(l.r - r.r, l.i - r.i);
    return tmp;
}

int main(void){
    Complex a(1,2);
    Complex b(3,4);
    a.print();
    b.print();

    Complex c = a+b;// a.operator+(b);
    c.print();

    Complex d = c - a; // operator-(c, a);
    d.print();


    return 0;
}
```

对赋值类双目运算符重载时需要注意的事项：

- 表达式的结果是左值，就是左操作数的自身
- 左操作数必须是左值，右操作数可以是左值也可以是右值

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

class Complex{
private:
    double r;
    double i;
public:
    Complex(double r, double i){
        this->r = r;
        this->i = i;
    }
    void print(void){
        cout << r << " + " << i << "i" << endl;
    }
    const Complex operator+(const Complex& c){
        Complex tmp(r+c.r, i+c.i);
        return tmp;
    }
    Complex & operator+=(const Complex &c){
        r = r + c.r;
        i = i + c.i;

        return *this;
    }

    friend const Complex operator-(const Complex& l , const Complex& r);
    friend Complex & operator-=(Complex &L, const Complex &R);
};
const Complex operator-(const Complex& l, const Complex& r){
    Complex tmp(l.r - r.r, l.i - r.i);
    return tmp;
```

```
}
Complex & operator-=(Complex &L, const Complex &R){
    L.r -= R.r;
    L.i -= R.i;
    return L;
}

int main(void){
    Complex a(1,2);
    Complex b(3,4);
    a.print();
    b.print();

    Complex c = a+b;// a.operator+(b);
    c.print();

    Complex d = c - a; // operator-(c, a);
    d.print();

    a += b; //a.operator+=(b);
    a.print();

    (a -= b).print();

    return 0;
}
```

4.3 单目运算符重载

单目运算符： 只有一个操作数的运算符 #O

- 相反数：-
- 位反：~
- 逻辑非：！
- 自增：++
- 自减：--
- ……

4.3.1 计算类单目运算符

注意：

- 表达式结果是右值
- 操作数可以是左值也可以是右值

实现方式：

- 成员函数形式：O.operator#();
- 友元函数形式：operator#(O);

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


class Integer{
private:
    int i ;
```

```
public:
    Integer(int m=0):i(m){}

    void print(void){
        cout << i << endl;
    }
    const Integer operator-(){
        return Integer(-i);
    }
};
int main(void){
    Integer a(10);

    Integer b = -a;
    b.print();
    return 0;
}
```

4.3.2前缀自增减单目运算符

注意：

- 表达式结果是左值, 而且是自身
- 操作数必须是左值

实现方式：

- 成员函数形式：O.operator#();
- 友元函数形式：operator#(O);

```

```

4.3.3后缀自增减单目运算符

注意：

- 表达式结果是右值，是操作数自增减前的副本
- 操作数必须是左值

实现方式：

- 成员函数形式：O.operator#(哑元);
- 友元函数形式：operator#(O,哑元);

```

```

4.4 其他运算符重载

4.4.1 输入输出运算符重载

```
friend ostream& operato<<(ostream& os,const RIGHT& right){...}
    ostream, 标准库的类
    cout << a; //operator<<(cout,a)
```

```
friend istream& operato>>(istream& is,RIGHT& right){...}
    istream cin;//istream是标准库的类
    cin >> a;//operator>>(cin,a)
```

注：因为无法向标准库类添加成员函数，所以只能使用全局函数的形式

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

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

        return *this;
    }
    friend ostream& operator<<(ostream &os, const Complex& c){
        os << c.r << " + " << c.i << "i" << endl;
        return os;  // cout << a << b << c << endl;
    }
    friend istream& operator>>(istream &is, Complex& c){
        is >> c.r >> c.i ;
        return is;
    }

    friend const Complex operator-(const Complex& l , const Complex& r);
    friend Complex & operator-=(Complex &L, const Complex &R);
};
const Complex operator-(const Complex& l, const Complex& r){
    Complex tmp(l.r - r.r, l.i - r.i);
    return tmp;
}
Complex & operator-=(Complex &L, const Complex &R){
    L.r -= R.r;
    L.i -= R.i;
    return L;
}

int main(void){
```

```
    Complex c1(2,3);
    cout << c1;

    Complex c2(0,0);
    cin >> c2;
    cout << c2;

    return 0;
}
```

4.4.2 new和delete运算符重载

通过new创建该类的对象时，将首先调用该操作符函数分配内存(可重载)，然后再调用该类的构造函数

通过delete销毁该类的对象时，将首先调用该类的析构函数，然后再调用该操作符函数释放内存（可重载）

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

class A{

public:
    A(void){
        cout << "A construct" << endl;
    }
    ~A(void){
        cout << "~A disconstruct" << endl;
    }
    void * operator new(size_t size){
        cout << "A new" << endl;
        return malloc(size);
    }
    void operator delete(void *p){
        cout << "A delete" << endl;
        free(p);
    }
};

int main(void){
    A *p = new A;

    delete p;
    return 0;
}
```

4.5 编程综合实例

实现一个3*3的矩阵类，支持如下操作符：

- 运算类双目操作符：+ - *
- 赋值类双目操作符：+= -= *=
- 单目操作符：-(相反数)
- 输出操作符：<<

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

class Mat33{
private:
    int m_a[3][3];
public:
    Mat33(void){
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                m_a[i][j] = 0;
            }
        }
    }
    Mat33(int a[][3]){
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                m_a[i][j] = a[i][j];
            }
        }
    }
    // +    a+b
    const Mat33 operator+(const Mat33& m) const{
        int a[3][3] = {0};
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                a[i][j] = m_a[i][j] + m.m_a[i][j];
            }
        }

        Mat33 result(a);

        return result;
    }
    // -    a-b
    const Mat33 operator-(const Mat33& m) const{
        int a[3][3] = {0};
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                a[i][j] = m_a[i][j] - m.m_a[i][j];
            }
        }

        Mat33 result(a);

        return result;
    }
    // *    a*b
    const Mat33 operator*(const Mat33& m) const{
        int a[3][3] = {0};
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                for(int k=0; k<3; k++){
                    a[i][j] += m_a[i][k] * m.m_a[k][j] ;
                }
            }
        }

        Mat33 result(a);

        return result;
    }
    // +=  a += b;
```

```
    Mat33& operator+=(const Mat33& m){
        *this = *this + m; // operator+
        return *this;
    }
    // -=  a -= b;
    Mat33& operator-=(const Mat33& m){
        *this = *this - m; // operator-(m)
        return *this;
    }
    // *=  a *= b;
    Mat33& operator*=(const Mat33& m){
        *this = *this * m; // operator*
        return *this;
    }
    //-(取负)  -a;
    const Mat33 operator-(void) const{
        Mat33 m;
        return m - *this; // operator-(m)
    }

    /*
     *  os << a;
     * */
    friend ostream& operator<<(ostream& os, const Mat33& m){
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                os << m.m_a[i][j] << " ";
            }
            cout << endl;
        }
        return os;
    }

};

int main(void){
    int a1[3][3] = {1,2,3,4,5,6,7,8,9};
    int a2[3][3] = {9,8,7,6,5,4,3,2,1};

    Mat33 m1(a1);
    Mat33 m2(a2);

    cout << m1 << endl;
    cout << m2 << endl;

    cout << "m1+m2: "<< endl;
    cout << m1 + m2 << endl;
    cout << "m1-m2: "<< endl;
    cout << m1 - m2 << endl;
    cout << "m1*m2: "<< endl;
    cout << m1 * m2 << endl;

    cout << "m1 += m2: "<< endl;
    cout << (m1 += m2) << endl;
    cout << "m1 -= m2: "<< endl;
    cout << (m1 -= m2) << endl;
    cout << "m1 *= m2: "<< endl;
    cout << (m1 *= m2) << endl;

    cout << "-m2" << endl;
    cout << -m2 << endl;
```

```
    return 0;
}
```

注意：不是所有的操作符都能重载，下面操作符不能重载

```
::   .    ?:    sizeof    typeid
```

5 异常

异常是指程序运行期间发生的不正常情况，如new无法获得所需内存、数组下标越界、运算溢出、除0错误、无效参数以及打开文件不存在等。异常处理就是指对程序执行过程中产生的异常进行适当的处理，避免程序出现丢失数据或破坏系统运行等灾难性后果。

5.1 传统错误处理

5.1.1 通过函数返回值处理异常

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

class A{
public:
    A(void){
        cout << "A(void)" << endl;
    }
    ~A(){
        cout << "~A()" << endl;
    }
};

int func3(){
    A c;

    FILE *fp = fopen("./a.txt", "r");
    if(fp  == NULL){
        return -1;
    }
    cout << "open a.txt successed" << endl;
    fclose(fp);

    return 0;
}
int func2(){
    A b;
    if(func3() == -1)
        return -1;

    return 0;
}
int func1(void){
    A a;

    if(func2() == -1)
        return -1;

    return 0;
}
```

```
int main(void){
    func1();
    return 0;
}
```

优点：函数调用路径中栈对象得到正确析构，不存在内存泄漏

缺点：错误流程处理复杂，代码臃肿

5.1.2 通过远程跳转处理异常

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


jmp_buf env;

class A{
public:
    A(void){
        cout << "A(void)" << endl;
    }
    ~A(){
        cout << "~A()" << endl;
    }
};

int func3(){
    A c;

    FILE *fp = fopen("./a.txt", "r");
    if(fp  == NULL){
        longjmp(env, -1); //跳转到setjmp位置执行
    }
    cout << "open a.txt successed" << endl;
    fclose(fp);

    return 0;
}
int func2(){
    A b;
    func3();
    return 0;
}
int func1(void){
    A a;

    func2();
    return 0;
}

int main(void){

    if(setjmp(env) == 0){ //保存当前栈的快照
        func1();
    }
    else{
        cout << "failed" << endl;

        return -1;
```

```
    }
    return 0;
}
```

优点：不需要逐层判断，一步到位，代码精炼

缺点：函数调用路径中的栈对象失去析构机会，存在内存泄漏风险

5.2 C++的异常处理

对传统错误处理作出了改进：发扬优点，避免缺点。

C++引入了3个用于异常处理的关键字：try, throw,catch。 try用于检测可能发生的异常，throw用于抛出异常，catch用于捕获并处理由throw抛出的异常。try-thow-catch构造了C++异常处理的基本结构，形式如下：

```
try{
    ...
    if err1  throw xx1;
    ...
    if err2  throw xx2;
    ...
    if errn  throw xxn;

}
catch(type1 arg){....}
catch(type2 arg){....}
catch(typen arg){....}
```

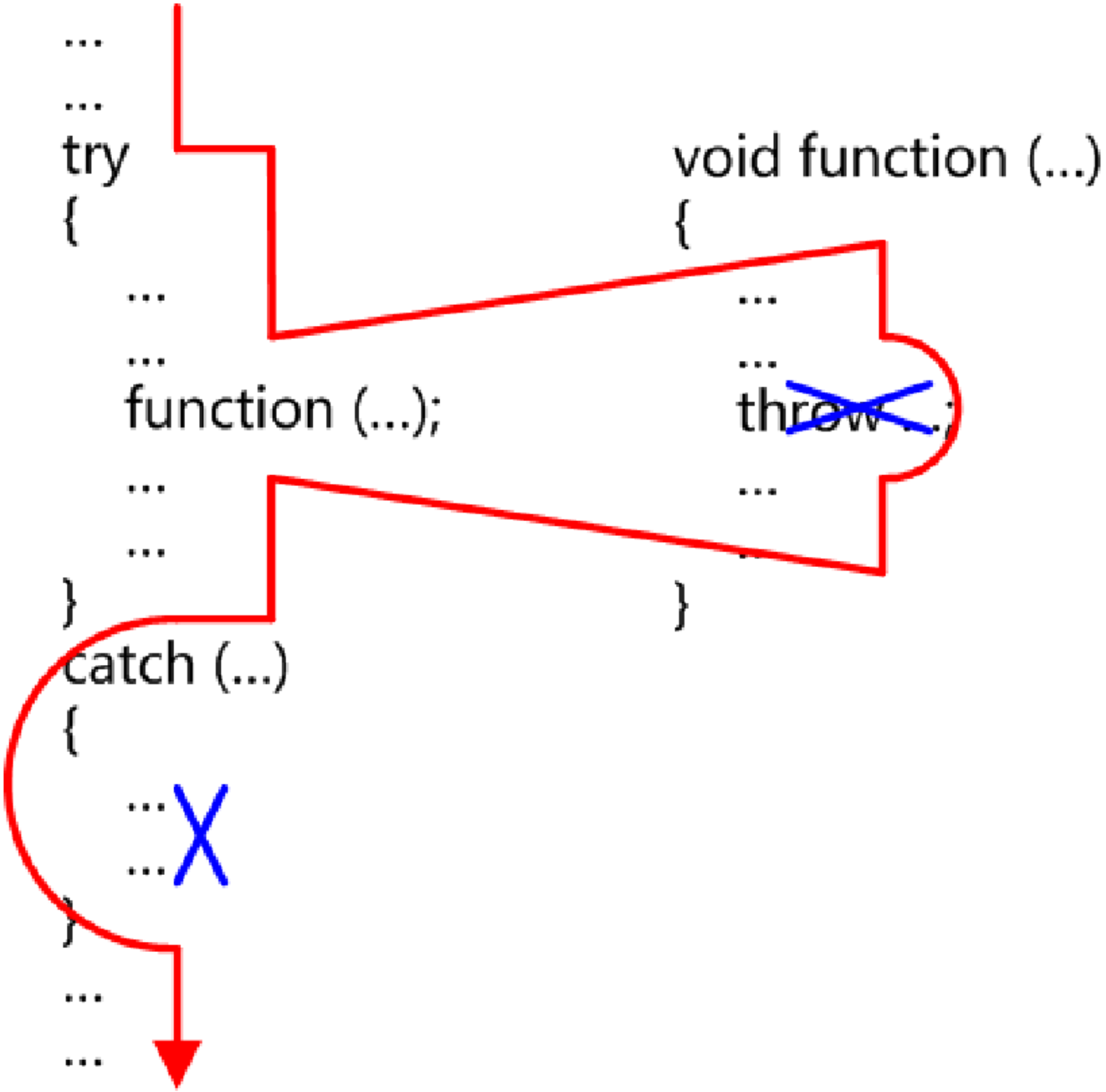
5.2.1 对传统错误处理的改造

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

class A{
public:
    A(void){
        cout << "A(void)" << endl;
    }
    ~A(){
        cout << "~A()" << endl;
    }
};

int func3(){
    A c;

    FILE *fp = fopen("./a.txt", "r");
    if(fp  == NULL){
        throw -1;
    }
    cout << "open a.txt successed" << endl;
    fclose(fp);

    return 0;
}
int func2(){
    A b;
    func3();
```

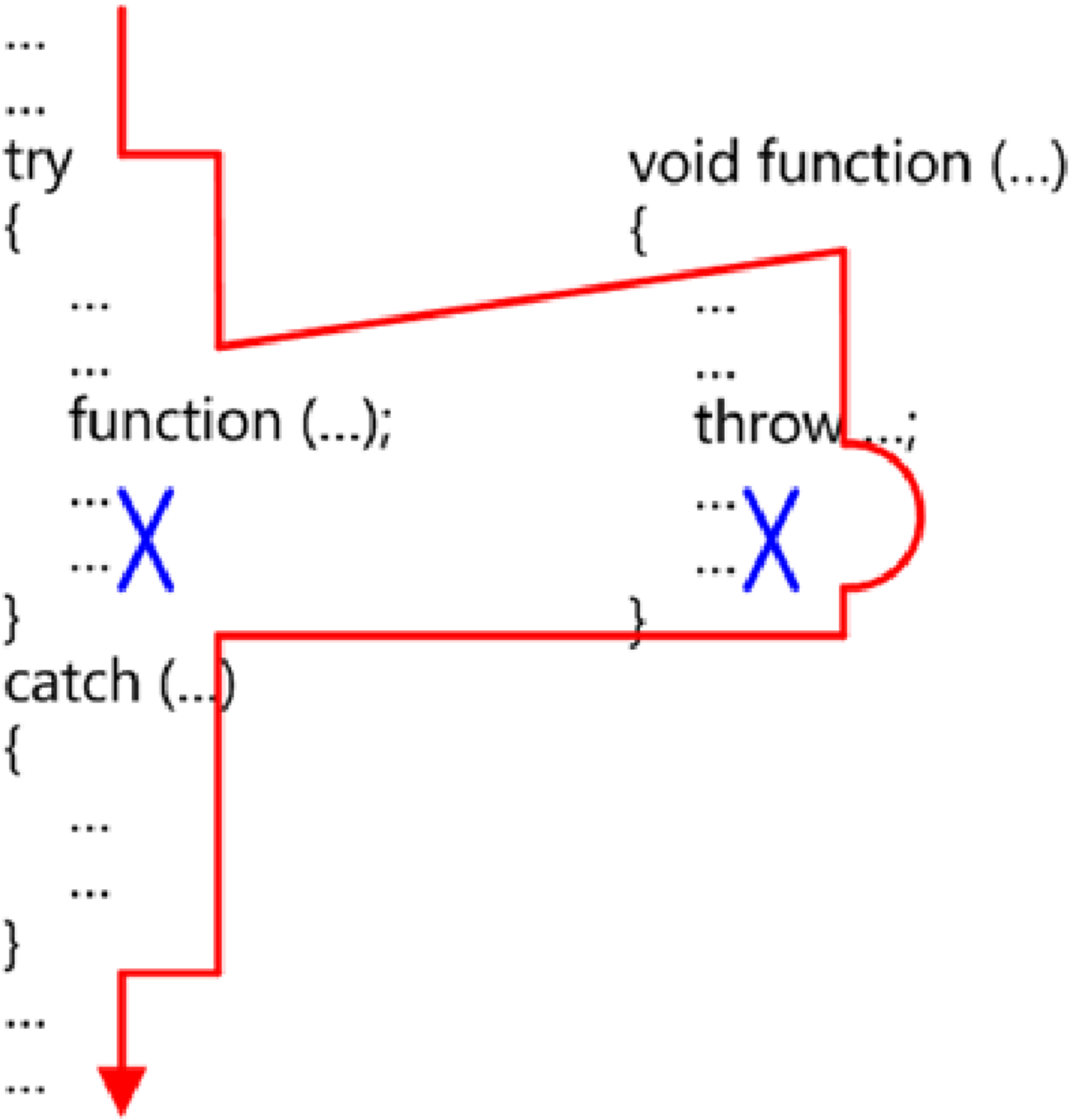
```
    return 0;
}
int func1(void){
    A a;

    func2();
    return 0;
}

int main(void){
    try{
        func1();
    }
    catch(int err){
        if(err == -1){
            cout << "handle file open failed" << endl;
            return -1;
        }
    }
    return 0;
}
```

5.2.2异常处理流程

- 未执行到throw时



- 执行到throw时




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

class A{
public:
    A(void){
        cout << "A(void)" << endl;
    }
    ~A(){
        cout << "~A()" << endl;
    }
};
class FileErr{
private:
    string filename;
    int line;
public:
    FileErr(const string &name, int num){
```

```
        filename = name;
        line = num;
    }
    void print(void){
        cout << filename << ": "<< line << ":" << "file open failed" << endl;
    }
};


int func3(){
    A c;

    FILE *fp = fopen("./a.txt", "r");
    if(fp  == NULL){
        //throw FileErr(__FILE__, __LINE__);
        throw -1;
    }
    cout << "open a.txt successed" << endl;
    fclose(fp);

    return 0;
}
int func2(){
    A b;
    func3();
    return 0;
}
int func1(void){
    A a;

    func2();
    return 0;
}

int main(void){
    try{
        func1();
    }
    catch(int err){
        if(err == -1){
            cout << "handle file open failed" << endl;
            return -1;
        }

    }
    catch(FileErr e){
        e.print();
        return -1;
    }
    return 0;
}
```

注意：catch在进行数据异常类型匹配时，不会进行数据类型的默认转换，只有与异常类型精确匹配的catch块才会被执行。

5.3 函数的异常说明

当一个函数声明中不带任何异常描述时，它可以抛出任何异常。

C++允许限制函数能够抛出的异常类型，限制方法时在函数声明后面添加一个throw参数表，在其中指定函数可以抛出的异常类型。

```
int fun(int, char) throw(int, char);
```

函数fun被限定只允许抛出int和char类型的异常，当fun函数抛出其他类型的异常时，程序将被异常终止。

如果函数不允许抛出任何异常，只需要指定throw限制表为不包括任何类型的空表。

```
int fun(int, char) throw();
```

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


class FileError{};
class MemoryError{};

void func(void) throw(FileError, MemoryError){
    throw FileError();
    //throw MemoryError();
    //throw -1;
}

int main(void){

    try{
        func();
    }
    catch(FileError& ex){
        cout << "file error" << endl;
        return -1;
    }
    catch(MemoryError& ex){
        cout << "memory error" << endl;
        return -1;
    }
    catch(int errnum){
        cout << "int error" << endl;
    }
    return 0;
}
```

5.4 标准异常类

C++库中专门设计了exception类表示异常。

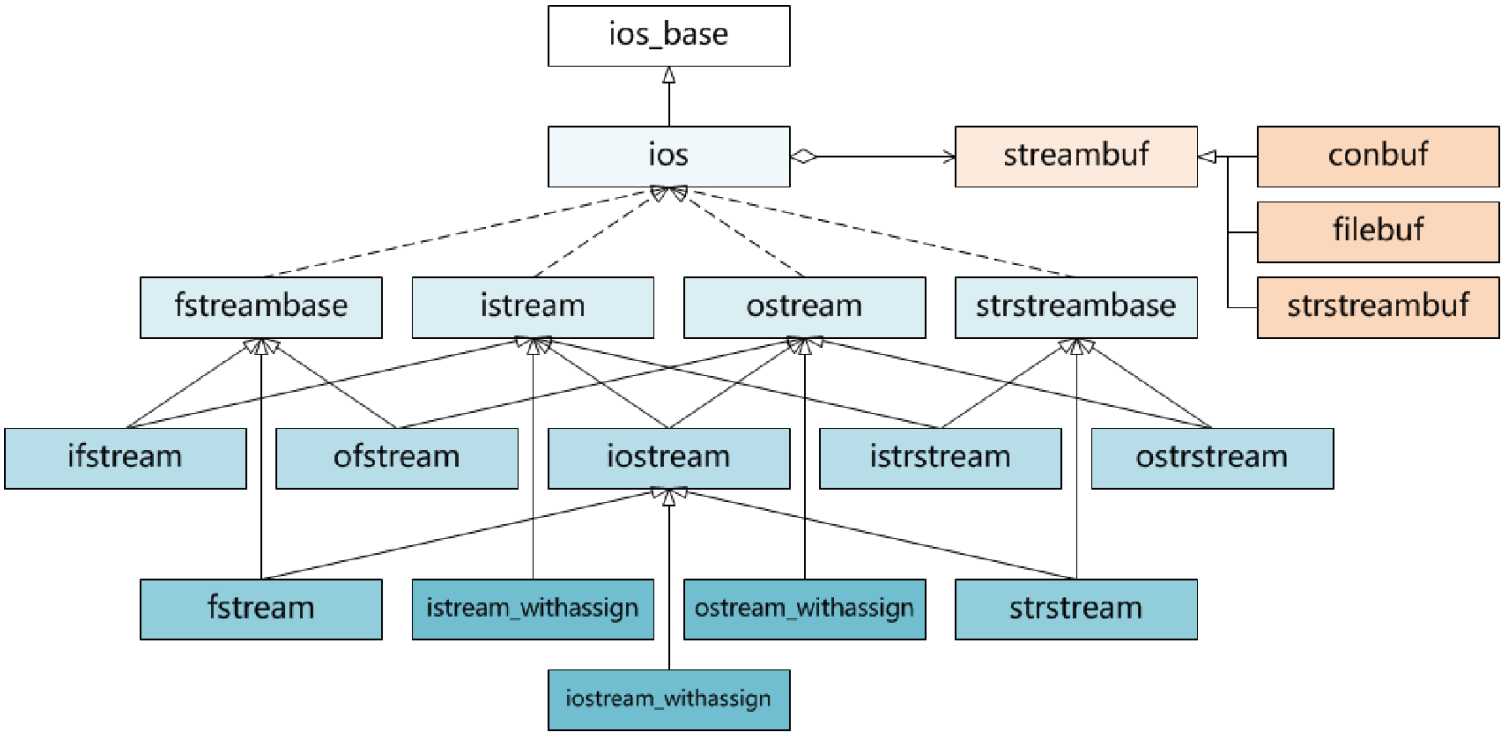
```
60   class exception
61   {
62   public:
63     exception() _GLIBCXX_USE_NOEXCEPT { }
64     virtual ~exception() _GLIBCXX_USE_NOEXCEPT;
65
66     /** Returns a C-style character string describing the general cause
67      *  of the current error.  */
68     virtual const char* what() const _GLIBCXX_USE_NOEXCEPT;
69   };
/usr/include/c++/4.8/exception [RO]
```

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


class FileError:public exception{
public:
    virtual const char *what() const throw(){
        cout << "handle file error" << endl;

        return "FileError";
    }
};
class MemoryError:public exception{
public:
    virtual const char *what() const throw(){
        cout << "handle memory error" << endl;

        return "MemoryErrorError";
    }

};

void func(void) throw(FileError, MemoryError){
    //throw FileError();
    throw MemoryError();
    //throw -1;
}

int main(void){

    try{
        func();
    }
    catch(exception& ex){
        cout << ex.what()<< endl;
        return -1;
    }
    return 0;
}
```

6 文件与流

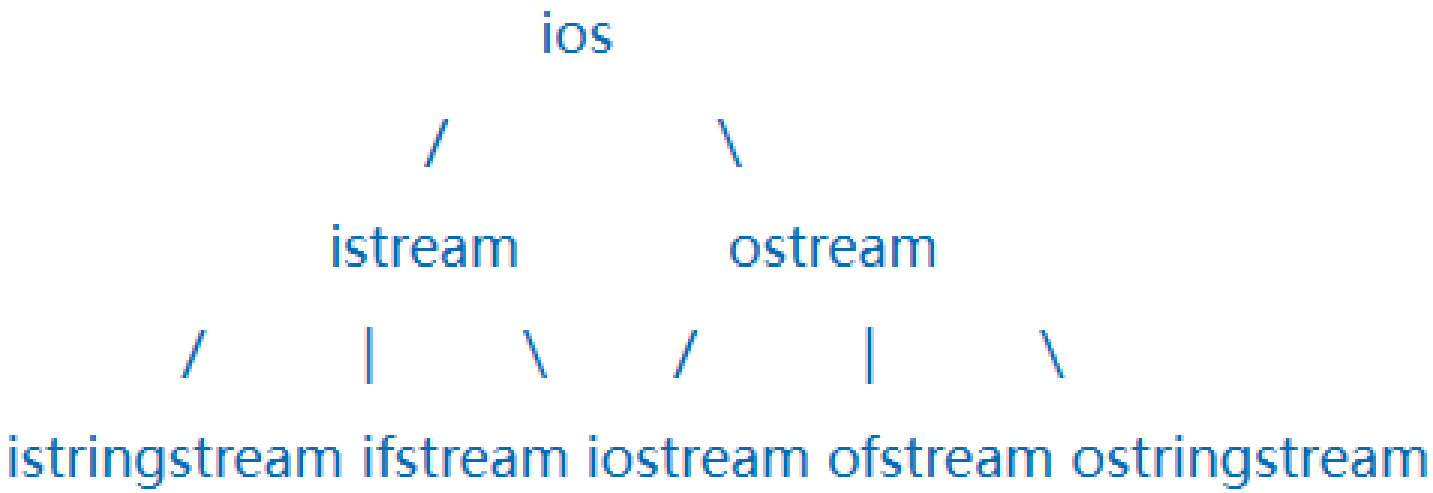
6.1 IO流库概览

C++建立了一个十分庞大的流类库来实现数据的输入输出操作。其中的每个流类实现不同的功能，这些类通过继承组合在一起。



ios类是istream类和ostream类的虚基类，用来提供对流进行格式化I/O操作和错误处理的成员函数。

streambuf主要作为其他类的支持，定义了对缓冲区的通用操作，如设置缓冲区、从缓冲区中读取数据，写入数据等操作。



为了便于程序数据的输入输出，C++预定义了几个标准输入输出流对象：

cout: ostream cout, 与标准输出设备关联

cin: istream cin, 与表示输入设备关联

cerr: ostream cerr, 与标准错误设备关联（非缓冲方式）

clog: ostream clog, 与标准错误设备关联（缓冲方式）

6.2 istream与ostream

6.1.1 istream

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

int main(void){
    char a[100] = {0};
```

```
    cout << "please input a string: ";
    //遇到空格就结束
    cin >> a;  // 123  456

    cout << a << endl;

    return 0;
}
```

istream类定义了许多用于从流中提取数据和操作文件的成员函数，并对流的析取运算符>>进行了重载，实现了对内置数据量的输入功能。其中常用的几个成员函数是get、getline、read 、ignore.

```
get( char_type& ch );  //从输入流中提取一个字符（包括空白符）
//一次性读取一行字符
getline( char_type* s, std::streamsize count, char_type delim );
//从输入流一次性读取count个字符
read( char_type* s, std::streamsize count );
//从输入流中读取字符并丢弃
ignore( std::streamsize count = 1, int_type delim = Traits::eof() )
```

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

int main(void){
    char a[100] = {0};
    char c ;
    cout << "please input a string: ";
    //遇到空格就结束
    //cin >> a;  // 123  456
    cin.getline(a, 100);
    cout << a << endl;

    cout << "use get(), please input char: ";

    while((c = cin.get()) != '\n')
        cout << c;
    cout << endl;


    cout << "use get(a, 10)  input char: ";
    cin.get(a, 10);
    cout << a <<endl;

    return 0;
}
```

6.1.2 ostream

ostream类提供了许多用于数据输出的成员函数，并通过流的输出<<重载，实现了对内置数据类型的输出功能。其中几个常用的成员函数是put、write、flush。

```
put( char_type ch ); //插入一个字符到输出流
write( const char_type* s, std::streamsize count );//插入一个字符序列到输出流中
flush();//刷新输出流
```

```
#include <iostream>
```

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

int main(void){

    char a[100] = "你好";
    cout << "hello" << endl;
    cout.put('w').put('o').put('r').put('l').put('d').put('\n');

    cout.write(a, strlen(a));
    return 0;
}
```

6.1.3 输入输出的格式控制

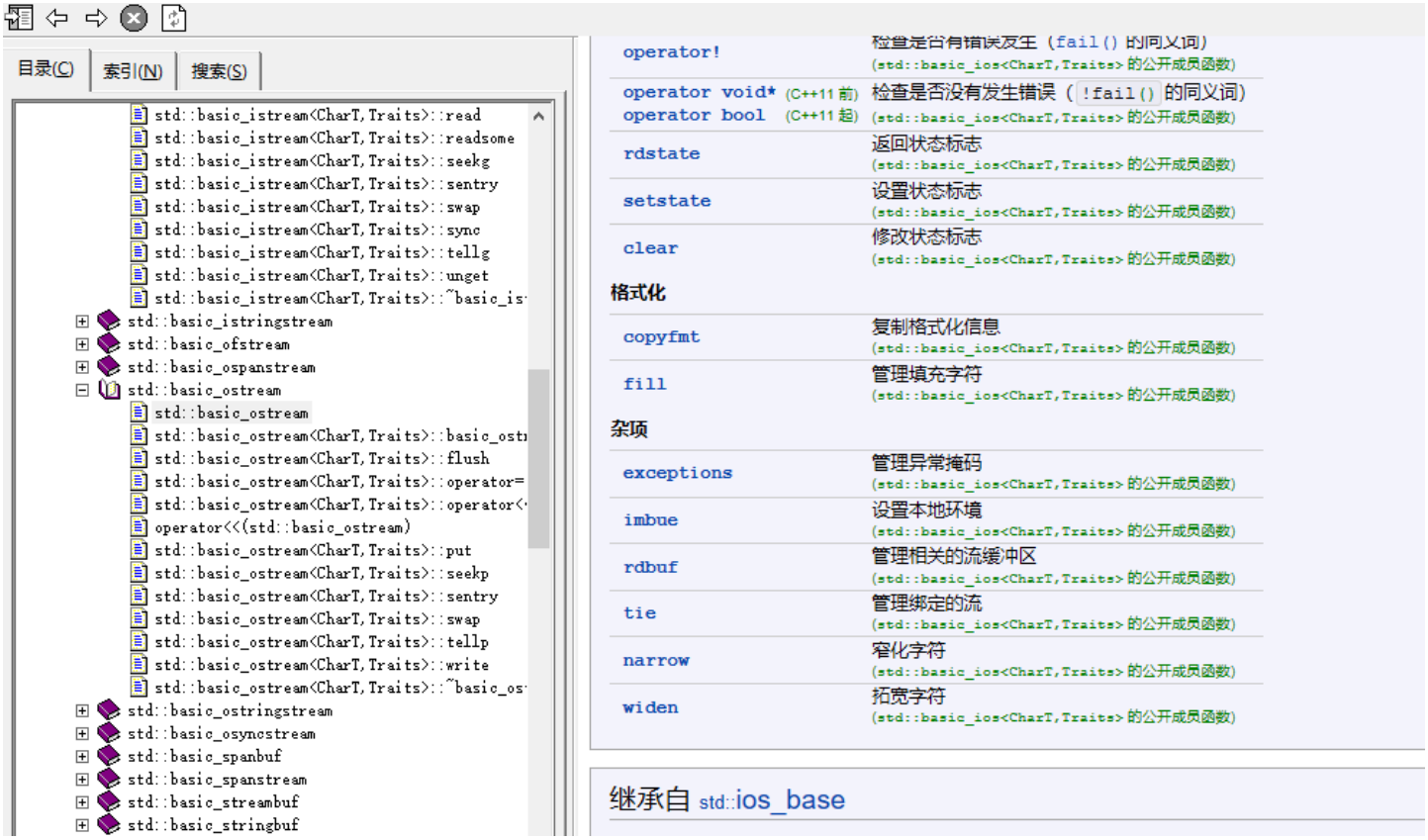
C语言中可以使用scanf()和printf()实现数据的格式化输入输出，C++中利用ios类的格式控制成员函数或者操纵符进行输入、输出数据的格式化。

- 操纵符 （全局函数）


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

int main(void){

    cout << 10/3.0 << endl;

    cout << setprecision(10) << 10/3.0 << endl;


    cout << setw(10) << 1234567890 << endl;
    cout << setw(10) << setfill('0') << 123 << endl;
```

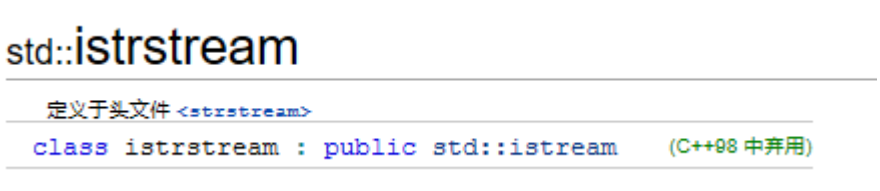
```
    cout << setw(8) << left << setfill(' ') << hex << showbase << 100 << endl;
    return 0;
}
```

- 成员函数




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

int main(void){

    cout << 10/3.0 << endl;

//  cout << setprecision(10) << 10/3.0 << endl;
    cout.precision(10);
    cout << 10/3.0 << endl;


    cout << 1234567890 << endl;
    //cout << setw(10) << setfill('0') << 123 << endl;
    cout.width(10);
    cout.fill('0');
    cout << 123 << endl;

    //cout << setw(8) << left << setfill(' ') << hex << showbase << 100 << endl;
    cout.width(8);
    cout.fill(' ');
    cout.setf(ios::hex, ios::basefield);
    cout.setf(ios::showbase);

    cout << 100 << endl;

    return 0;
}
```

6.3 string流



istrstream和ostrstream在98标准中废弃，取而代之的是istringstream和ostringstream，实现类似于C语言中sprintf和sscanf的效果。

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

int main(void){

    int i = 1234;
    double d = 56.78;
    char s[] = "hello";
#if 0
    char buf[100] = {0};
    sprintf(buf, "%d %lf %s", i, d, s);
    printf("%s\n", buf);

    char str[] = "100 1.23 world";
    sscanf(str, "%d %lf %s", &i, &d, s);
    printf("%d %lf %s\n", i, d, s);
#endif
    ostringstream oss;
    oss << i << ' ' << ' ' << d << ' ' << s;
    cout << oss.str() << endl;

    istringstream iss;
    iss.str("100 1.24 world");
    iss >> i >> d >> s;
    cout << i << ", " << d << ", " << s << endl;

    return 0;
}
```

6.4 文件流

C++将文件看成是一个个字符在磁盘上的有序集合，用流来实现文件的读写操作。C++中用来建立流对象的类有ifstream(输入)、ofstream（输出）、fstream（输入输出）。

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

int main(void){
    int i = 1234;
    double d = 56.78;
    char s[] = "hello";

    ofstream ofs("a.txt");
    ofs << i << d <<s << endl;
    ofs.close();
```

```
    ifstream ifs("a.txt");
    int i2;
    double d2;
    string s2;

    ifs >> i2 >> d2 >> s2;
    cout << i2 << endl;
    cout << d2 << endl;
    cout << s2 << endl;

    return 0;
}
```

泛型编程

1 模板与STL

模板（template）是C++实现代码重用机制的重要工具，是泛型技术（即与数据类型无关的通用程序设计技术）的基础。模板表示的是概念级的通用程序设计方法，它把算法和数据类型区分开来，能够设计出独立于具体数据类型的模板程序，模板程序能以数据类型为参数生成针对于该类型的实际程序代码。模板分为函数模板和类模板两类，ANSI标准C++库就是使用模板技术实现的。

1.1 模板的概念

某些程序除了所处理的数据类型之外，程序代码和程序功能相同，但为了实现他们，却不得不编写多个与具体数据类型紧密结合的程序。例如

```
int Min(int a, int b){
    return a<b?a:b;
}
float Min(float a, float b){
    return a<b?a:b;
}
double Min(double a, double b){
    return a<b?a:b;
}
char Min(char a, char b){
    return a<b?a:b;
}
```

如何简化以上编程呢？C语言中，可以通过宏的方式实现以上想法：

```
#define Min(x,y) ((x)<(y)?(x):(y))
```

C++中，也可以利用宏来进行类似程序设计，但宏避开了C++类型检查机制，在某些情况下可能引发错误，是不安全的。更好的方法就是模板来实现这样的程序设计。

C++中的模板与制作冰糕的模具很相似，是生产函数或类的模具。模板接收数据类型参数，并根据此类型创建相应的函数或类。

对于上面的所有的Min()而已，只需要下面的函数模板就能够生成说有的Min函数。

```
template <typename T>
T Min(T a, T b){
    return a<b?a:b;
}
```

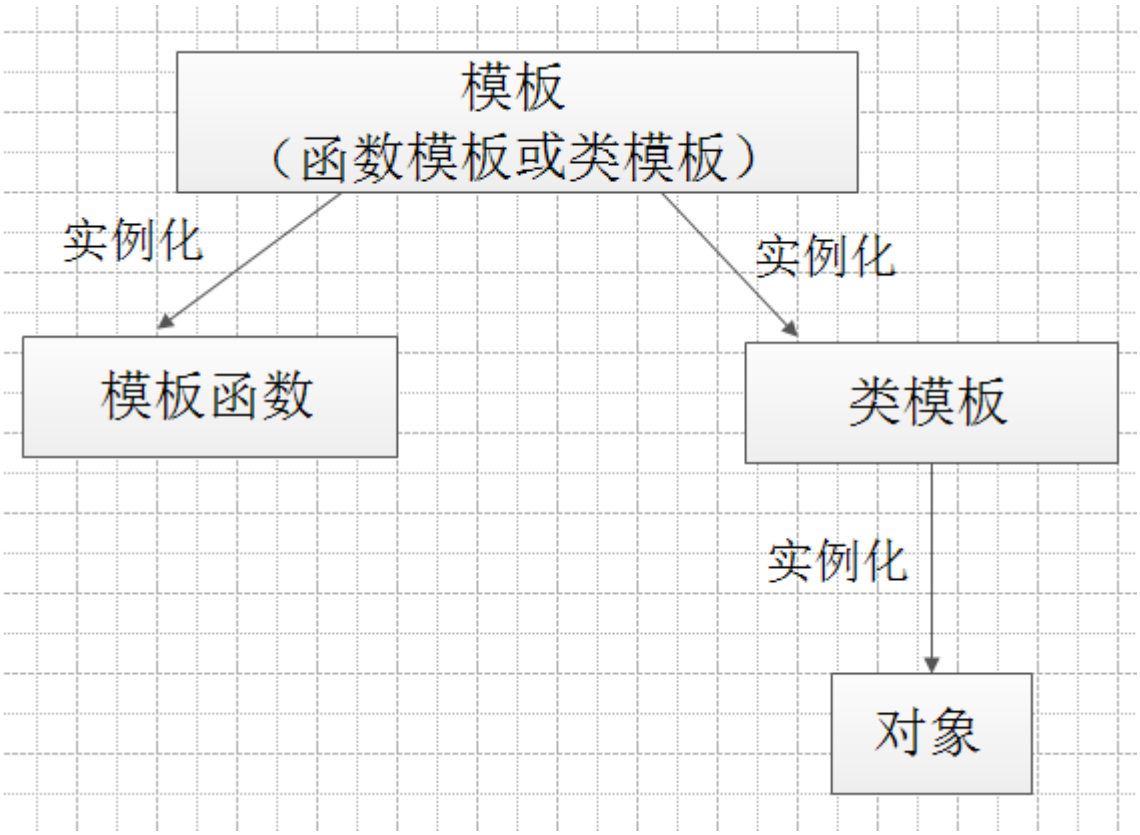
template和typename是用来定义模板的关键字。Min模板不涉及任何具体的数据类型，而是用T代表任意数据类型，称为类型参数。Min模板代表了求两个数值最小值的通用算法，它与具体数据类型无关，但能够生成计算各种具体数据类型的最小值的函数。编译器的做法是用具体的类型替换模板中的类型参数T，生成具体类型的函数Min()。比如用int替换掉模板中所有的T就能生成求两个int类型数据的函数Min().

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

#if 0
int Min(int a, int b){
    return a<b?a:b;
}
float Min(float a, float b){
    return a<b?a:b;
}
double Min(double a, double b){
    return a<b?a:b;
}
char Min(char a, char b){
    return a<b?a:b;
}
#endif
template <typename T>
T Min(T a, T b){
```

```
    return a<b?a:b;
}

int main(void){
    int m=9, n=3;
    double d1=1.8, d2=3.4;

    cout << Min(m,n) << endl;
    cout << Min(d1, d2) << endl;


    return 0;
}
```

从函数模板Min可以看成，C++模板提供了对逻辑结构相同的数据对象通用行为的定义方法，它把通用算法的实现和具体的数据类型区分开来，模板操作的是参数化的数据类型（类型参数）而非实际数据类型。一个带有类型参数的函数称为函数模板，带有类型参数的类称为模板类。

在调用模板时，必须为它的类型参数提供实际数据类型，C++将用该数据类型替换模板中的全部类型参数，由编译器生成与具体的数据类型相关的可以运行的程序代码，这个过程称为**模板的实例化。**由函数模板实例化生成的函数称为**模板函数**， 由类模板实例化生成的类称为**模板类。**



1.2 函数模板

函数模板提供了一种通用的函数行为，该函数行为可以用多种不同的数据类型进行调用，编译器会根据调用类型自动将它实例化为具体数据类型的函数代码，也就是说函数模板代表了一个函数家族。与普通函数相比，函数模板中某些函数元素的数据类型是未确定的，这些元素的类型将在使用时被参数化；与重载函数相比，函数模板不需要程序员重复编写函数代码，它可以自动生成许多功能相同但参数和返回值类型不同的函数。

1.2.1 函数模板的定义

```
template <typename T1, typename T2, ....>
返回类型 函数名(参数表){
    ... ...
}
```

template是模板定义的关键字； 写在<>中的T1 T2,...是模板参数，其中的typename表示其后的参数可以是任意类型的。

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

#if 0
int Min(int a, int b){
    return a<b?a:b;
}
float Min(float a, float b){
    return a<b?a:b;
}
double Min(double a, double b){
    return a<b?a:b;
}
char Min(char a, char b){
    return a<b?a:b;
}
#endif
template <typename T>
T Min(T a, T b){
    return a<b?a:b;
}

int main(void){
    int m=9, n=3;
    double d1=1.8, d2=3.4;
    cout << Min(m,n) << endl;
    cout << Min(d1, d2) << endl;
    return 0;
}
```

1.2.2 函数模板的实例化

当编译器遇到关键字template和跟随其后的参数定义时，它只是简单地知道这个函数模板在后面的程序代码中可能会用到。除此之外，编译器并不会做额外的工作。在这个阶段函数模板本身并不能使编译器产生任何代码，因为编译器此时并不知道函数模板要处理的具体数据类型，根本无法生成任何函数代码。

当编译器遇到程序中对函数模板的调用时，它才会根据调用语句中实参的具体类型，确定模板参数的数据类型，并用此类型替换函数模板的模板参数，生成能够处理该类型的函数代码，即模板函数。

对于1.2.1代码可以执行以下命令观察实验效果。

```
nm a.out | grep a.out
```

1.2.3模板参数

- 模板参数的匹配问题

  C++在实例化函数模板的过程中，只是简单地将模板参数替换为实参的类型，并以此生成模板函数，不会进行参数类型的任何转换。这种方式与普通函数的参数处理有着极大的区别，以前在普通函数的调用过程中，会进行参数的自动类型转换。

  ```
  #include <iostream>
  ```

```
using namespace std;

double Max(double a, double b){
    return a>b?a:b;
}

int main(void){
    double a = 2, b = 3.4;
    float c = 1.1, d = 2.2;

    cout << "2, 3.4 的最大值是: " << Max(a,b) << endl;
    cout << "a, c的最大值是: "  << Max(a,c) << endl;
    cout << "a, 100的最大值是: " << Max(a,100) << endl;
    return 0;
}
```

以上程序 能够正确执行。现在使用函数模板来实现通用的功能，如下所示

```
#include <iostream>

using namespace std;

template <typename T>
T Max(T a, T b){
    return a>b?a:b;
}

int main(void){
    double a = 2, b = 3.4;
    float c = 1.1, d = 2.2;

    cout << "2, 3.4 的最大值是: " << Max(a,b) << endl;
    cout << "a, c的最大值是: "  << Max(a,c) << endl;
    cout << "a, 100的最大值是: " << Max(a,100) << endl;
    return 0;
}
```

编译以上程序，产生模板参数不匹配的错误。产生这个错误的原因是模板实例化过程中不会进行任何的参数类型转换。编译器在翻译Max(a,c)时，由于实参类型为double和float，而Max函数模板只有一个形参类型T，总不能让T同时取double和float两种类型吧？要知道模板实例化过程中，C++不会进行任何形式的隐式类型转换，于是产生了上述编译错误。

这种问题的解决方式有：

- 在模板调用时进行参数类型的强制转换

  ```
  cout << "a, c的最大值是: "  << Max(a,double(c)) << endl;
  ```

- 显示指定函数模板实例化的类型参数

  ```
  #include <iostream>

  using namespace std;

  template <typename T>
  T Max(T a, T b){
      return a>b?a:b;
  }

  int main(void){
  ```

```
    double a = 2, b = 3.4;
    float c = 1.1, d = 2.2;

    cout << "2, 3.4 的最大值是: " << Max(a,b) << endl;
    cout << "a, c的最大值是: "  << Max<double>(a,c) << endl;
    cout << "a, 100的最大值是: " << Max<double>(a,100) << end;
    return 0;
}
```

- 指定多个模板参数

  在模板函数的调用过程中，为了避免出现一个模板参数与多个调用实参的类型冲突问题，可以为函数模板指定多个不同的类型参数。

```
#include <iostream>

using namespace std;

template <typename T1, typename T2>
T1 Max(T1 a, T2 b){
    return a>b?a:b;
}

int main(void){
    double a = 2, b = 3.4;
    float c = 1.1, d = 2.2;

    cout << "2, 3.4 的最大值是: " << Max(a,b) << endl;
    cout << "a, c的最大值是: "  << Max(a,c) << endl;
    cout << "a, 100的最大值是: " << Max(a,100) << endl;
    return 0;
}
```

- 模板函数的形参表

  不要误以为函数模板中的参数只能是类型形参，它也可以包括普通类型的参数。

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

template <typename T>
void display(T &arr, unsigned int n){
    for(int i=0; i<n; i++){
        cout << arr[i] << "\t" ;
    }
    cout << endl;
}

int main(void){
    int a[] = {12,34,56,78,11,10,999};
    char b[] = {'x', 'y', 'z', 'n'};

    display(a, sizeof(a)/sizeof(a[0]));
    display(b, sizeof(b)/sizeof(b[0]));
    return 0;
}
```

1.2.4 函数模板的特化

在某些情况下，函数模板并不能生成处理特定数据类型的模板函数。上面例子中的Max函数模板可以计算int 或者 char类型数据的最大值，但对于字符串类型却是无能为力的。解决这类问题的方法就是对函数模板进行特化。所谓特化，就是针对模板不能处理的特殊数据类型，编写与模板同名的特殊函数专门处理这些数据类型。语法格式如下所示。

```
template<>
返回类型 函数名<特化的数据类型>(参数表){
    ....
}
```

例如：

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

template <typename T>
T Min(T a, T b){
    return a<b?a:b;
}
template <>
const char *Min(const char *a, const char *b){
    cout << "Min(const char *, const char *)" << endl;
    return (strcmp(a,b)>0)?b:a;
}

int main(void){
    int m=9, n=3;
    double d1=1.8, d2=3.4;

    cout << Min(m,n) << endl;
    cout << Min(d1, d2) << endl;
    cout << Min('a', 'b') << endl;
    cout << Min("aaa", "bbb") << endl;

    return 0;
}
```

1.3 类模板

函数模板用于设计程序代码相同而所处里的数据类型不同的通用函数。与此类似，C++也支持用类模板来设计结构和成员函数完全相同，但所处理的数据类型不同的通用类。比如，对于堆栈类而言，可能存在整数栈、双精度数栈、字符栈等多种不同数据类型的栈，每个栈类除了所处理的数据类型不同之外，类的结构和成员函数完全相同，可为了在非模板的类设计中实现这些栈，不得不重复编写各个栈类的相同代码，例如初始化栈、入栈、出栈等操作。为了解决该问题，C++中用类模板来设计这样的类簇最方便，一个类模板就能够实例化生成所有需要的栈类。

类模板也称为类属类，它可以接收类型作为参数，设计出与具体类型无关的通用类。在设计类模板时，可以使其中的某些数据成员，成员函数的参数或返回值与具体类型无关。

1.3.1 类模板的定义

类模板与函数模板的定义形式相似，如下所示：

```
template <typename T1, typename T2, ....>
class 类名{
    ... ...
};
```

实例：设计一个栈的类模板Stack，在模板中使用类型参数T表示栈中存放的数据， 用非类型参数MAXSIZE代表栈的大小。

```
vim Stack.cpp
```

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

template<typename T, int MAXSIZE>
class Stack{
private:
    T elements[MAXSIZE];
    int top; //栈顶
public:
    Stack():top(0){
    }
    void push(T e);
    T pop();
    bool empty(){
        return top == 0;
    }
    bool full(){
        return top==MAXSIZE;
    }
};
template<typename T, int MAXSIZE>
void Stack<T, MAXSIZE>::push(T e){
    if(full()){
        cout << "stack is full, can not push" << endl;
    }
    elements[top++] = e;
}

template<typename T, int MAXSIZE>
T Stack<T, MAXSIZE>:: pop(){
    if(top == 0){
        cout << "stack is empty, can not pop" << endl;
        return 0;
    }
    top--;
    return elements[top] ;
}
#if 0
int main(void){
    Stack<int, 10>  iStack;

    return 0;
}
#endif
```

1.3.2 类模板实例化

类模板的实例化包括模板实例化和成员函数实例化。当用类模板定义对象时，将引起类模板的实例化。在实例化类模板时，如果模板参数是类型参数，则必须为它指定具体的类型；如果模板参数是非类型参数，则必须为它指定一个常量值。如对前面的Stack类模板而言，下面是它的一条实例化语句：

```
Stack<int, 10>  istack;
```

编译器实例化Stack的方法是：将Stack模板声明中的所有的类型参数T替换为int, 将所有的非类型参数MAXSIZE替换为10， 这样就用Stack模板生成了一个int类型的模板类。为了区别于普通类，暂且将该类记作Stack<int, 10>, 即在类模板名后面的一对<>中写上模板参数。该类的代码如下：

```
class Stack{
private:
    int elements[10];
    int top; //栈顶
public:
    Stack():top(0){
    }
    void push(int e);
    int pop();
    bool empty(){
        return top == 0;
    }
    bool full(){
        return top==MAXSIZE;
    }
};
```

最后c++用这个模板类定义一个对象istack.

注意：在上面的实例化过程中，并不会实例化类模板的成员函数，也就是说，在用类模板定义对象时并不会生成类成员函数的代码。类模板成员函数的实例化发生在该成员函数被调用时，这就意味着只有哪些被调用的成员函数才会被实例化。或者说，只有当成员函数被调用了，编译器才会为它生成真正的的代码。

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

template<typename T, int MAXSIZE>
class Stack{
private:
    T elements[MAXSIZE];
    int top; //栈顶
public:
    Stack():top(0){
    }
    void push(T e);
    T pop();
    bool empty(){
        return top == 0;
    }
    bool full(){
        return top==MAXSIZE;
    }
};
template<typename T, int MAXSIZE>
void Stack<T, MAXSIZE>::push(T e){
    if(full()){
        cout << "stack is full, can not push" << endl;
    }
    elements[top++] = e;
}

template<typename T, int MAXSIZE>
T Stack<T, MAXSIZE>:: pop(){
    if(top == 0){
        cout << "stack is empty, can not pop" << endl;
        return 0;
    }
    top--;
```

```
    return elements[top] ;
}

int main(void){
    Stack<int, 10>  iStack;
    iStack.push(12);
    return 0;
}
```

可以将pop函数的类外定义删除掉，然后再编译运行程序，可以发现程序通用能够正确地执行。

与普通类的对象一样，类模板的对象或引用也可以作为函数的参数，只不过这类函数通常是模板函数，且其调用实参常常是该类模板的模板类对象。

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

template<typename T, int MAXSIZE>
class Stack{
private:
    T elements[MAXSIZE];
    int top; //栈顶
public:
    Stack():top(0){
    }
    void push(T e);
    T pop();
    bool empty(){
        return top == 0;
    }

    bool full(){
        return top == MAXSIZE;
    }
};
template<typename T, int MAXSIZE>
void Stack<T, MAXSIZE>::push(T e){
    if(full()){
        cout << "stack is full, can not push" << endl;
    }
    elements[top++] = e;
}
template<typename T, int MAXSIZE>
T Stack<T, MAXSIZE>::pop(){
    top--;
    return elements[top];
}
template<typename T>
void display(Stack<T, 10> &s){
    while(!s.empty()){
        cout << s.pop() << endl;
    }
}

int main(void){
    Stack<int, 10>  iStack;
    iStack.push(12);
    iStack.push(1);
    iStack.push(2);
    iStack.push(123);
    iStack.push(12111);
    display(iStack);
```

```
    return 0;
}
```

1.3.3 类模板特化

类模板代表了一种通用程序设计的方法，它表示了无限类集合，可以实例化生成基于任何类型的模板类。在通常情况下，由类模板生成的模板类都能够正常地工作，但也有有类模板生成的模板类代码对某些数据类型不适用的情况。

如：设计一个通用数组类，它能够直接存取数组元素，并能够输出数组中的最小值。

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


template<typename T>
class Arr{
private:
    T *arr;
    int size;
public:
    Arr(int size = 10){
        this->size = size;
        //arr = new T(size);
        arr = (T *)malloc(sizeof(T) * size);
    }
    ~Arr(){
        free(arr);
    }
    T &operator[](int i);
    T Min();

};
template<typename T>
T&  Arr<T>::operator[](int i){

    if(i<0 || i > size -1){
        cout << "\n 数组下标越界" << endl;
    }
    return arr[i];

}
template<typename T>
T Arr<T>::Min(){
    T tmp;
    tmp = arr[0];
    for(int i=1; i<size; i++){
        if(tmp > arr[i])
            tmp = arr[i];
    }
    return tmp;
}

int main(void){

    Arr<int> a(5);
    for(int i=0; i<5; i++)
        a[i] = i+100;
    cout << a.Min() << endl;

    Arr<char *> b(5);
    b[0] = (char *)"faa";
```

```
    b[1] = (char *)"bbb";
    b[2] = (char *)"ccc";
    b[3] = (char *)"ddd";
    b[4] = (char *)"eee";
    cout << b.Min() << endl;


    return 0;
}
```

显然Arr类模板并不完全适用于生成char *类型的模板类，因为Arr类模板的Min成员函数并不适用于字符指针类型的计算大小。

解决上述问题的方法就是类模板的特化，即用与该模板相同的名字为某种数据类型专门重新一个模板类。特化类模板时，可以随意增减和改写模板原有的成员，成员函数的改写也不受任何限制，可以与原来的成员函数变得完全不同。

类模板有两种特化方式，一种是特化整个类模板，另一种是特化个别成员函数。前者是为某种类型单独建立一个类，后者则只针对特化的数据类型提供个别成员函数的实现代码，特化后的成员函数不再是一个模板函数，而是针对特点类型的普通函数。

与函数模板特化方式相同，类模板成员函数的特化也以template<>开头。形式如下：

```
template <>
返回类型 类模板名<特化的数据类型>:: 特化成员函数名(参数表){
    ... ...
}
```

Arr类模板对char *类型来说， 除了Min成员函数不适用外，其余成员都可以，则针对char * 类型重新编写Min成员函数，即特例Arr类模板的Min成员函数就能够解决字符串求大小的问题。特化的Min函数入下：

```
template<>
char * Arr<char *>::Min(){
    char *tmp;
    tmp = arr[0];
    for(int i=1; i<size; i++){
        if(strcmp(tmp ,arr[i])>0)
            tmp = arr[i];
    }
    return tmp;
}
```

为了某种数据类型特化整个类模板也要以template<>开头，形式如下所示：

```
template <> class 类模板名<特化数据类型>{
    ... ...
};
```

上例中，也可以为char *提供整个模板的特化，如下所示：

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


template<typename T>
class Arr{
private:
    T *arr;
    int size;
public:
    Arr(int size = 10){
```

```
        this->size = size;
        //arr = new T(size);
        arr = (T *)malloc(sizeof(T) * size);
    }
    ~Arr(){
        free(arr);
    }
    T &operator[](int i);
    T Min();

};
template<typename T>
T&  Arr<T>::operator[](int i){

    if(i<0 || i > size -1){
        cout << "\n 数组下标越界" << endl;
    }
    return arr[i];

}
template<typename T>
T Arr<T>::Min(){
    T tmp;
    tmp = arr[0];
    for(int i=1; i<size; i++){
        if(tmp > arr[i])
            tmp = arr[i];
    }
    return tmp;
}
template<>
class Arr<char *>{
private:
    char **arr;
    int size;
public:
    Arr(int size = 10){
        this->size = size;
        //arr = new T(size);
        arr = (char **)malloc(sizeof(char *) * size);
    }
    ~Arr(){
        free(arr);
    }
    char * &operator[](int i){
        if(i<0 || i > size -1){
            cout << "\n 数组下标越界" << endl;
        }
        return arr[i];
    }

    char * Min(){
        char * tmp;
        tmp = arr[0];
        for(int i=1; i<size; i++){
            if(strcmp(tmp, arr[i]) > 0)
                tmp = arr[i];
        }
        return tmp;
    }
};

int main(void){
```

```
    Arr<int> a(5);
    for(int i=0; i<5; i++)
        a[i] = i+100;
    cout << a.Min() << endl;

    Arr<char *> b(5);
    b[0] = (char *)"faa";
    b[1] = (char *)"bbb";
    b[2] = (char *)"ccc";
    b[3] = (char *)"ddd";
    b[4] = (char *)"eee";
    cout << b.Min() << endl;
    return 0;
}
```

1.4 STL

STL就是标准模板库(Standard Template Library), 它提供了模板化的通用类和通用函数。STL的核心内容包括容器、迭代器、算法三部分内容，三者常常协同工作，为各种编程问题提供有效的解决方案。

STL提供了许多可直接用于程序设计的通用数据结构和能强大的类与算法，是程序设计者的一个巨大宝藏。程序员可以在STL中找到各种常用的数据结构和算法，这些数据结构和算法是准确而有效的，用它们来解决编程中的各种问题，可以减少程序测试时间，写出高质量的代码，提高编程效率。

1.4.1 容器

容器是用来存储其它对象的对象。容器是容器类的实例，而容器类使用类模板实现的，适用于各种数据类型。STL的容器常被分为顺序容器、关联容器和容器适配器三类。顺序容器常被称为序列容器，它是将相同类型对象的有限集按顺序组织在一起的容器，用来表示线性数据解结构，C++提供的顺序类型容器有向量(vector)、链表(list)、双端队列(deque);关联容器是非线性容器，是用来根据键(key)进行快速存储、检索数据的容器。这类容器可以存储值的集合或键值对，C++中的关联容器主要包括集合(set)、多重集合(multiset)、映射(map)、多重映射(multimap);容器适配器主要指堆栈(stack)和队列(queue)，它们实际是受限制访问的顺序容器类型。

STL库中十大容器

STL容器名	头文件	说明
vector		向量，从后面快速插入和删除，直接访问任何元素
list		双向链表
deque		双端队列
set		元素不重复的集合
multiset		元素可重复的集合
stack		堆栈，先进后出
map		一个键只对应一个值得映射
mutimap		一个键可对应对个值得映射
queue		队列，先进先出
priority_queue		优先级队列

STL是经过精心设计的，为了减小操作使用容器的难度，大多数容器都提供了相同的成员函数，如下表所示。

所有容器都具有的成员函数

成员函数	说明
默认构造函数	对容器进行默认初始化的构造函数，常有多个，用于提供不同的容器初始化方法
拷贝构造函数	用于将容器初始化为同类型的现有容器的副本
析构函数	执行容器销毁时的清理工作
empty()	判断容器是否为空，为空返回true
max_size()	返回容器的最大容量
size	返回容器中当前元素的个数
operator=	将一个容器赋给另一个同类容器
operator<	如果第一个容器小于第二个容器，返回true
operator<=	如果第一个容器小于等于第二个容器，返回true
operator>	如果第一个容器大于第二个容器，返回true
operator>=	如果第一个容器大于等于第二个容器，返回true
swap	交换两个容器中的元素

- **vector**

  vector 容器是STL中最常用的容器之一， vector 实现的是一个动态数组，即可以进行元素的插入和删除，在此过程中，vector 会动态调整所占用的内存空间，整个过程无需人工干预。

  vector 容器以类模板 vector（T 表示存储元素的类型）的形式定义在 头文件中，并位于 std 命名空间中。因此，在创建该容器之前，代码中需包含如下内容：

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

  - 创建向量容器的方式

    ```
    //创建空的向量容器
    vector<double> values;
    values.reserve(20);
    //指定初始值以及元素个数
    vector<int> primes {2, 3, 5, 7, 11, 13, 17, 19};
    //创建 vector 容器时，指定元素个数
    vector<double> values(20); //有 20 个元素，它们的默认初始值都为 0。
    vector<double> values(20, 1.0); //有 20 个元素，它们的默认初始值都为 1.0。
    //通过存储元素类型相同的其它 vector 容器，创建新的 vector 容器
    vector<char> value1(5, 'c');
    vector<char> value2(value1);
    ```

  - vector容器包含的成员函数

函数成员	函数功能
begin()	返回指向容器中第一个元素的迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。
rbegin()	返回指向最后一个元素的迭代器。
rend()	返回指向第一个元素所在位置前一个位置的迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
size()	返回实际元素个数。
max_size()	返回元素个数的最大值。这通常是一个很大的值，一般是 232-1，所以我们很少会用到这个函数。
resize()	改变实际元素的个数。
capacity()	返回当前容量。
empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
reserve()	增加容器的容量。
shrink _to_fit()	将内存减少到等于当前元素实际所使用的大小。
operator[ ]	重载了 [ ] 运算符，可以向访问数组中元素那样，通过下标即可访问甚至修改 vector 容器中的元素。
at()	使用经过边界检查的索引访问元素。
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
data()	返回指向容器中第一个元素的指针。
assign()	用新元素替换原有内容。
push_back()	在序列的尾部添加一个元素。
pop_back()	移出序列尾部的元素。
insert()	在指定的位置插入一个或多个元素。
erase()	移出一个元素或一段元素。
clear()	移出所有的元素，容器大小变为 0。
swap()	交换两个容器的所有元素。
emplace()	在指定的位置直接生成一个元素。
emplace_back()	在序列尾部生成一个元素。

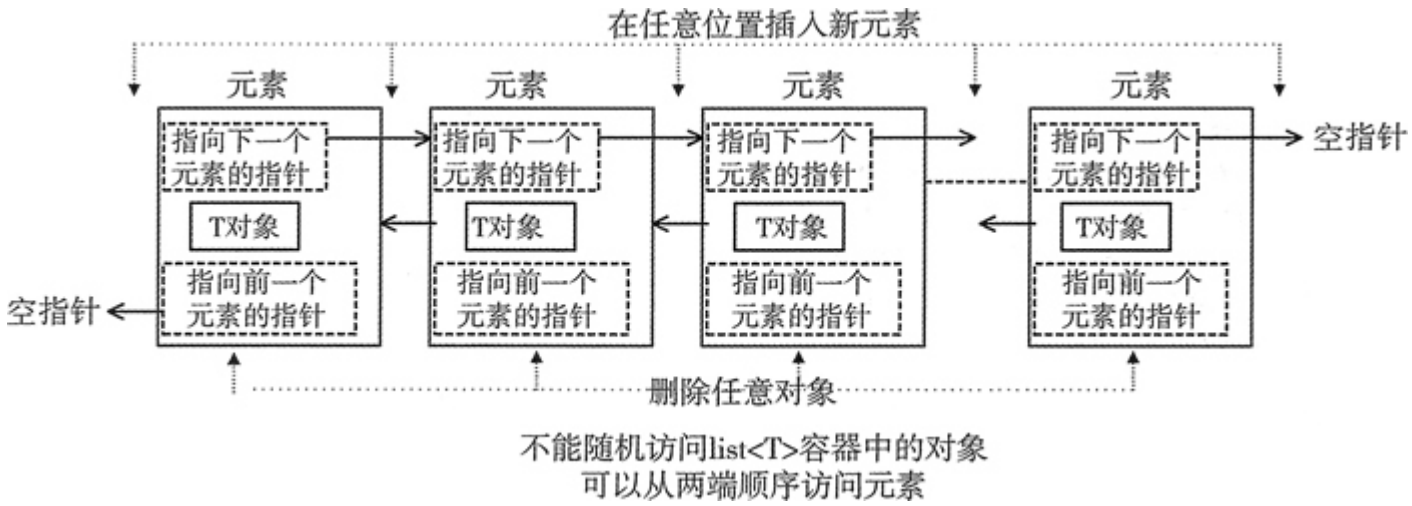
- 编程实例

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

```
void display(vector<int> &v){
    while(!v.empty()){
        cout << v.back() << " ";
        v.pop_back();
    }

    cout << endl;

}
int main(void){

    vector<int> v1;
    v1.reserve(10);
    cout << v1.capacity() << endl;
    display(v1);

    vector<int> v2(100);
    cout << v2.capacity() << endl;


    vector<int> v3(10,3);
    v3.insert(v3.begin(), 1111);
    display(v3);


    return 0;
}
```

- **list**

  list 容器，又称双向链表容器，即该容器的底层是以双向链表的形式实现的。这意味着，list 容器中的元素可以分散存储在内存空间里，而不是必须存储在一整块连续的内存空间中。



  list 容器以模板类 list（T 为存储元素的类型）的形式在``头文件中，并位于 std 命名空间中。因此，在使用该容器之前，代码中需要包含下面两行代码：

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

  - list容器的创建

```
list<int> values;  //没有任何元素
list<int> values(10); //包含 10 个元素每个元素的值都为相应类型默认值（int类型默认值为 0）
list<int> values(10, 5);//包含 10 个元素并且值都为 5 。
list<int> value2(values); //和values元素个数、内容相同
//拷贝普通数组，创建list容器
int a[] = { 1,2,3,4,5 };
std::list<int> values(a, a+5); //values中5个元素，分别为 1  2  3  4 5
```

- 常用成员函数

成员函数	功能
begin()	返回指向容器中第一个元素的双向迭代器。
end()	返回指向容器中最后一个元素所在位置的下一个位置的双向迭代器。
rbegin()	返回指向最后一个元素的反向双向迭代器。
rend()	返回指向第一个元素所在位置前一个位置的反向双向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
size()	返回当前容器实际包含的元素个数。
max_size()	返回容器所能包含元素个数的最大值。这通常是一个很大的值，一般是 232-1，所以我们很少会用到这个函数。
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
assign()	用新元素替换容器中原有内容。
emplace_front()	在容器头部生成一个元素。该函数和 push_front() 的功能相同，但效率更高。
push_front()	在容器头部插入一个元素。
pop_front()	删除容器头部的一个元素。
emplace_back()	在容器尾部直接生成一个元素。该函数和 push_back() 的功能相同，但效率更高。
push_back()	在容器尾部插入一个元素。
pop_back()	删除容器尾部的一个元素。
emplace()	在容器中的指定位置插入元素。该函数和 insert() 功能相同，但效率更高。
insert()	在容器中的指定位置插入元素。
erase()	删除容器中一个或某区域内的元素。
swap()	交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。
resize()	调整容器的大小。
clear()	删除容器存储的所有元素。
splice()	将一个 list 容器中的元素插入到另一个容器的指定位置。
remove(val)	删除容器中所有等于 val 的元素。
remove_if()	删除容器中满足条件的元素。
unique()	删除容器中相邻的重复元素，只保留一个。
merge()	合并两个事先已排好序的 list 容器，并且合并之后的 list 容器依然是有序的。
sort()	通过更改容器中元素的位置，将它们进行排序。

成员函数	功能
reverse()	反转容器中元素的顺序。

- **stack**

  STL 提供了 3 种容器适配器，分别为 stack 栈适配器、queue 队列适配器以及 priority_queue 优先权队列适配器。 容器适配器本质上还是容器，只不过此容器模板类的实现，利用了大量其它基础容器模板类中已经写好的成员函数。

  要使用STL中的堆栈容器，代码中需要包含下面两行代码：

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

  STL中提供堆栈容器的主要操作如下：

  push(), 将一个元素加入stack内，加入的元素放在栈顶

  top(), 返回栈顶元素的值

  pop(), 删除栈顶元素

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


int main(void){

    stack<int> s;

    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.top() << endl; //30

    s.top() = 100;

    s.push(40);
    s.push(50);

    s.pop(); //弹栈  5 0

    while(!s.empty()){
        cout << s.top() << endl; //40 100 20 10
        s.pop();

    }
    cout << endl;

    return 0;
}
```

1.4.2 迭代器

迭代器（iterator）是一个对象，常用它来遍历容器，即在容器中实现“取得下一个元素”的操作。不管容器是否直接提供了访问其对象的方法，通过迭代器都能够访问该容器中的元素，一次访问一个元素。

迭代器是STL的核心，它定义了哪些算法在哪些容器中可以使用，把算法和容器连接起来，使算法、容器和迭代器能够协同工作，实现强大的程序功能。

若某个容器要使用迭代器，它就必须定义迭代器。定义迭代器时，必须指定迭代器所使用的容器类型。比如，若定义了一个保存int类型元素的链表：

```
list<int> L1;
```

则为int类型的list容器指定迭代器的定义如下：

```
list<int>::iterator iter;
```

完成该定义后，STL会自动将此迭代器转换成链表所需要的双向迭代器，该迭代器可以用于int型的list。

迭代器提供的主要操作如下：

```
operator*                       返回当前位置的元素值
operator++                      将迭代器前进到下一个元素位置
operator--                      将迭代器后退到前一个元素位置
operator==或operator!=          判定两个迭代器是否指向同一个位置
operator=                       为迭代器赋值
begin()                         指向容器起点(即第一个元素)位置
end()                           指向容器的结束点，结束点在最后一个元素之后
rbegin()                        指向按反向顺序的第一个元素位置
rend()                          指向按反向顺序的最后一个元素后的位置
```

实例：

```
#include <iostream>
#include <list>

using namespace std;

int main(void){

    int i = 0;
    list<int> L1, L2, L3(10);
    list<int>::iterator iter;

    int a1[] = {100,90,80,70,60};
    int a2[] = {30,40,50,60,60,60,80};

    for(i=0; i<5; i++)
        L1.push_back(a1[i]);
    for(i=0; i<7; i++)
        L2.push_back(a2[i]);

    for(iter = L1.begin(); iter!=L1.end(); iter++){
        cout << *iter << "\t";
    }
    cout << endl;

    int sum = 0;
    //for(iter=--L2.end(); iter!=L2.begin(); iter--){
    iter = L2.end();
    do{
        iter--;
        cout << *iter << "\t";
        sum += *iter;
    }while(iter != L2.begin());
    cout << "\nL2: sum=" << sum << endl;

    int data = 0;
    for(iter=L3.begin(); iter!=L3.end(); iter++)
```

```
        *iter = data+=10;
    for(iter = L3.begin(); iter!=L3.end(); iter++){
        cout << *iter << "\t";
    }
    cout << endl;


    return 0;
}
```

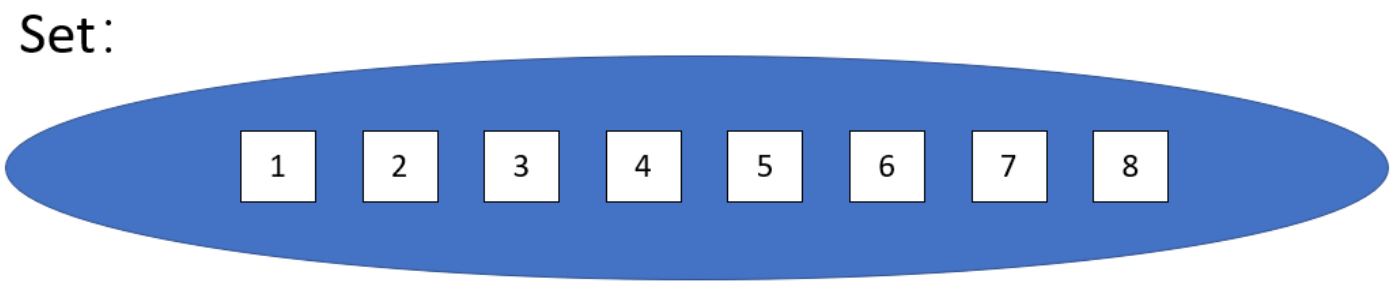
STL中的迭代器可分为双向迭代器、前向迭代器、后向迭代器、输入迭代器和输出迭代器几种类型。前面介绍的迭代器属于双向迭代器，读者若要了解其他几种迭代器的用法，可以参考C++帮助文档。

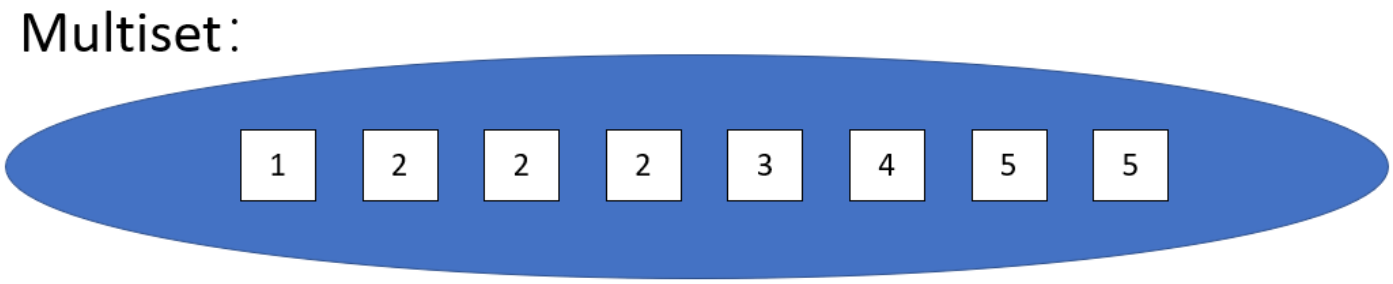
1.4.3 关联式容器

STL关联容器包括集合和映射两大类，集合包括set和multiset,映射包括map和multimap，它们通过关键字存储和查找元素。在每种关联容器中，关键字按顺序排列，容器遍历就可以顺序进行。

- **set和multiset**

  集合类multiset和set提供了控制数字(包括字符及串)集合的操作，集合中的数字称为关键字，不需要有另一个值与关键字相关联。set和multiset会根据特定的排序准则，自动将元素排序，两者提供的操作方法基本相同，只是multiset允许元素重复而set不允许重复。





定义:

操作	描述
set c	Default 构造函数，建立一个空 set / multiset，不含任何元素
set c(op)	建立一个空 set / multiset，以 op 为排序准则
set c(c2)	Copy 构造函数，为相同类型之另一个 set / multiset 建立一份拷贝，所有元素均被复制
set c = c2	Copy 构造函数，为相同类型之另一个 set / multiset 建立一份拷贝，所有元素均被复制
set c(rv)	Move 构造函数，建立一个新的 set / multiset，有相同类型,取 rvalue rv 的内容(**始自C++11**)
set c = rv	Move 构造函数，建立一个新的 set / multiset，有相同类型，取 rvalue rv 的内容(**始自C++11**)
set c(beg, end)	以区间 [beg, end) 内的元素为初值，建立一个 set / multiset
set c(beg, end, op)	以区间 [beg, end) 内的元素为初值，并以 op 为排序准则，建立一个 set / multiset
set c(initlist)	建立一个 set / multiset，以初值列 initlist 的元素为初值(**始自C++11**)
set c = initlist	建立一个 set / multiset，以初值列 initlist 的元素为初值(**始自C++11**)
c.~**set**()	销毁所有元素，释放内存
其中set可为下列形式：	
set	描述
–	–
set< Elem >	一个 set，以 less<> (operator <) 为排序准则
set< Elem, Op >	一个 set，以 Op 为排序准则
multiset< Elem >	一个 multiset，以 less<> (operator <) 为排序准则
multiset< Elem, Op >	一个 multiset，以 Op 为排序准则

其中op可以是less<>或greater<>之一，应用时须在<>中写上类型，如greater。less指定排序方式为从小到大，greater指定排序方式为从大到小，默认排序方式为less。

非更易型操作：

操作	描述
c.key_comp()	返回“比较准则”
c.value_comp()	返回针对 value 的“比较准则”（和 key_comp() 相同）
c.empty()	返回是否容器为空（相当于 size()==0 但也许较快）
c.size()	返回目前的元素个数
c.max_size()	返回元素个数之最大可能量
c1 == c2	返回 c1 是否等于 c2（对每个元素调用==）
c1 != c2	返回 c1 是否不等于 c2（相当于 !(c1 == c2) ）
c1 < c2	返回 c1 是否小于 c2
c1 > c2	返回 c1 是否大于 c2（相当于 c2 < c1）
c1 <= c2	返回 c1 是否小于等于 c2（相当于 !(c2 < c1) ）
c1 >= c2	返回 c1 是否大于等于 c2（相当于 !(c1 < c2) ）

查找操作：

操作	描述
c.count(val)	返回"元素值为 val "的元素个数
c.find(val)	返回"元素值为 val "的第一个元素，如果找不到就返回 end()
c.lower_bound(val)	返回 val 的第一个可安插位置，也就是"元素值 >= val "的第一个元素位置
c.upper_bound(val)	返回 val 的最后一个可安插位置，也就是"元素值 > val "的第一个元素位置
c.equal_range(val)	返回 val 可被安插的第一个位置和最后一个位置，也就是"元素值 == val "的元素区间。将 lower_bound() 和 upper_bound() 的返回值做成一个 pair 返回。 如果 lower_bound() 或" equal_range() 的 first 值"等于" equal_range() 的 second 值"或 upper_bound()，则此 set 或 multiset 内不存在同值元素。

赋值操作:

操作	描述
c = c2	将 c2 的全部元素赋值给 c
c = rv	将 rvalue rv 的所有元素以 move assign 方式给予 c (**始自C++11**)
c = initlist	将初值列 initlist 的所有元素赋值给 c (**始自C++11**)
c1.swap(c2)	置换 c1 和 c2 的数据
swap(c1, c2)	置换 c1 和 c2 的数据

迭代操作:

操作	描述
c.begin()	返回一个 bidirectional iterator 指向第一元素
c.end()	返回一个 bidirectional iterator 指向最末元素的下一位置
c.cbegin()	返回一个 const bidirectional iterator 指向第一元素(**始自C++11**)
c.cend()	返回一个 const bidirectional iterator 指向最末元素的下一位置(**始自C++11**)
c.rbegin()	返回一个反向的 (reverse) iterator 指向反向迭代的第一元素
c.rend()	返回一个反向的 (reverse) iterator 指向反向迭代的最末元素的下一位置
c.crbegin()	返回一个 const reverse iterator 指向反向迭代的第一元素(**始自C++11**)
c.crend()	返回一个 const reverse iterator 指向反向迭代的最末元素的下一位置(**始自C++11**)

插入移除操作:

操作	描述
c.insert(val)	安插一个 val 拷贝，返回新元素位置，不论是否成功——对 set 而言
c.insert(pos, val)	安插一个 val 拷贝，返回新元素位置(pos 是个提示,指出安插动作的查找起点。若提示恰当可加快速度)
c.insert(beg, end)	将区间 [beg, end) 内所有元素的拷贝安插到 c (无返回值)
c.insert(initlist)	安插初值列 initlist 内所有元素的一份拷贝(无返回值；**始自C++11**)
c.emplace(args . . .)	安插一个以 args 为初值的元素，并返回新元素的位置，不论是否成功——对 set 而言(**始自C++11**)
c.emplace_hint(pos , args . . .)	安插一个以 args 为初值的元素，并返回新元素的位置(pos 是个提示，指出安插动作的查找起点。若提示恰当可加快速度)
c.erase(val)	移除"与 val 相等"的所有元素，返回被移除的元素个数
c.erase(pos)	移除 iterator 位置 pos 上的元素，无返回值
c.erase(beg, end)	移除区间 [beg, end) 内的所有元素，无返回值
c.clear()	移除所有元素，将容器清空

实例

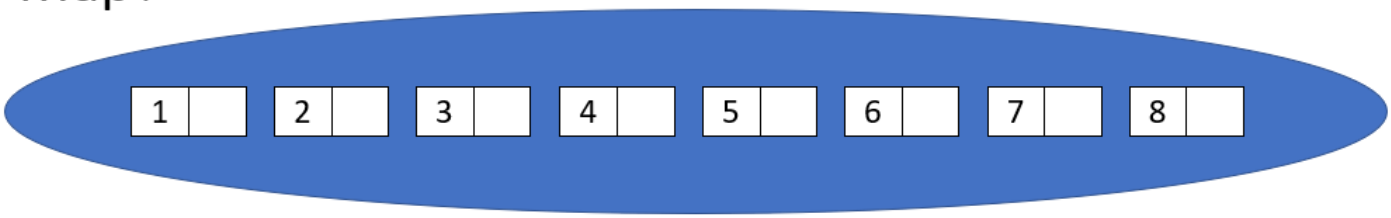
```
#include <iostream>
#include <set>
using namespace std;

int main(void){
    int a1[] = {-2, 0,31, 11,6,7,12,10,9,10};

    set<int, greater<int> > set1(a1, a1+10);

    set<int, greater<int> >::iterator it;

    set1.insert(4);
    for(it=set1.begin(); it!=set1.end(); it++){
        cout << *it << " ";
    }
    cout << endl;

    string a2[] ={"赵云", "张飞", "关羽", "马超", "黄忠", "张辽", "乐进", "于禁", "张合", "徐晃"};
    multiset<string> set2(a2, a2+10);
    multiset<string>::iterator it2;
    set2.insert("赵云");
    for(it2=set2.begin(); it2!=set2.end(); it2++){
        cout << *it2 << " ";
    }
    cout << endl;

    return 0;
}
```

- **map和multimap**

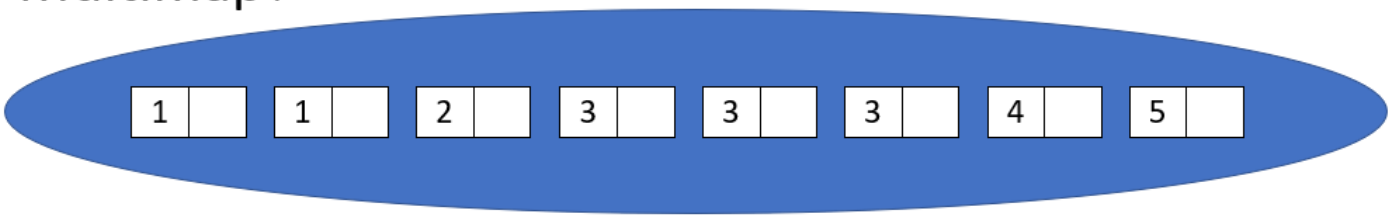
  map和multimap提供了操作<键，值>对的方法，它们存储一对对象，即键对象和值对象，键对象是用于查找过程中的键，值是与键对应的附加数据。例如，若键为单词，对应的值是表示该单词在文档中出现此数的数字，这样的map就成了统计单词在文本中出现次数的频数表；再如，若键为单词，值是单词出现的页号链表，用multimap实现这样的键值对象就是可以构造单词索引表。

  map中的元素不允许重复，而multimap中的元素是可以重复的。

Map：



Multimap：



定义

操作	描述
map c	Default 构造函数,建立一个空 map/multimap，不含任何元素
map c(op)	建立一个空 map/multimap，以 op 为排序准则
map c(c2)	Copy 构造函数，为相同类型之另一个 map/multimap 建立一份拷贝，所有元素均被复制
map c = c2	Copy 构造函数，为相同类型之另一个 map/multimap 建立一份拷贝，所有元素均被复制
map c(rv)	Move 构造函数，建立一个新的 map/multimap，有相同类型，取 rvalue rv 的内容（**始自C++11**）
map c = rv	Move 构造函数，建立一个新的 map/multimap，有相同类型，取 rvalue rv 的内容（**始自C++11**）
map c(beg, end)	以区间 [beg, end) 内的元素为初值，建立一个 map/multimap
map c(beg, end, op)	以区间 [beg, end) 内的元素为初值，并以 op 为排序准则，建立一个 map/multimap
map c(initlist)	建立一个 map/multimap，以初值列 initlist 的元素为初值（**始自C++11**）
map c = initlist	建立一个 map/multimap，以初值列 initlist 的元素为初值（**始自C++11**）
c.~**map**()	销毁所有元素，释放内存
其中，**map** 可为下列形式：	
map	描述
–	–
map< Key, Val >	一个 map，以 less<>（operator <）为排序准则
map< Key, Val, Op >	一个 map，以 Op 为排序准则
multimap< Key, Val >	一个 multimap，以 less<>（operator <）为排序准则
multimap< Key, Val, Op >	一个 multimap，以 Op 为排序准则

非更易型操作

操作	描述
c.key_comp()	返回“比较准则”(comparison criterion)
c.value_comp()	返回针对 value 的“比较准则”（那是个对象，用来在一个 key/value pair 中比较 key）
c.empty()	返回是否容器为空（相当于 size() == 0 但也许较快）
c.size()	返回目前的元素个数
c.max_size()	返回元素个数之最大可能量
c1 == c2	返回 c1 是否等于 c2（对每个元素调用==）
c1 != c2	返回 c1 是否不等于 c2（相当于 !(c1 == c2)）
c1 < c2	返回 c1 是否小于 c2
c1 > c2	返回 c1 是否大于 c2（相当于c2 < c1）
c1 <= c2	返回 c1 是否小于等于 c2（相当于!(c2 < c1)）
c1 >= c2	返回 c1 是否大于等于 c2（相当于!(c1 < c2)）

查找操作

操作	描述
c.count(val)	返回“key 为 val”的元素个数：如果 key 存在于容器中，则返回1，因为映射仅包含唯一 key 。如果键在Map容器中不存在，则返回0。
c.find(val)	返回“key 为 val”的第一个元素，找不到就返回 end()。该函数返回一个迭代器或常量迭代器，该迭代器或常量迭代器引用键在映射中的位置。
c.lower_bound(val)	返回“key 为 val”之元素的第一个可安插位置，也就是“key >= val”的第一个元素位置
c.upper_bound(val)	返回“key 为 val”之元素的最后一个可安插位置，也就是“key > val”的第一个元素位置
c.equal_range(val)	返回“key 为 val”之元素的第一个可安插位置和最后一个可安插位置，也就是“key == val”的元素区间

赋值操作

操作	描述
c = c2	将 c2 的全部元素赋值给 c
c = rv	将 rvalue rv 的所有元素以 move assign 方式给予 c (**始自C++11**)
c = initlist	将初值列 initlist 的所有元素赋值给 c (**始自C++11**)
c1.swap(c2)	置换 c1 和 c2 的数据
swap(c1, c2)	置换 c1 和 c2 的数据

迭代操作

操作	描述
c.begin()	返回一个 bidirectional iterator 指向第一元素
c.end()	返回一个 bidirectional iterator 指向最末元素的下一位置
c.cbegin()	返回一个 const bidirectional iterator 指向第一元素(**始自C++11**)
c.cend()	返回一个 const bidirectional iterator 指向最末元素的下一位置(**始自C++11**)
c.rbegin()	返回一个反向的（reverse) iterator 指向反向迭代的第一元素
c.rend()	返回一个反向的（reverse) iterator 指向反向迭代的最末元素的下一位置
c.crbegin()	返回一个 const reverse iterator 指向反向迭代的第一元素(**始自C++11**)
c.crend()	返回一个 const reverse iterator 指向反向迭代的最末元素的下一位置(**始自C++11**)

插入移除操作'

操作	描述
c.insert(val)	安插一个 val 拷贝，返回新元素位置，不论是否成功——对 map 而言
c.insert(pos , val)	安插一个 val 拷贝，返回新元素位置（pos 是个提示，指出安插动作的查找起点。若提示恰当可加快速度)
c.insert(beg , end)	将区间 [beg, end) 内所有元素的拷贝安插到 c (无返回值)
c.insert(initlist)	安插初值列 initlist 内所有元素的一份拷贝(无返回值；**始自C++11**)
c.emplace(args . . .)	安插一个以 args 为初值的元素，并返回新元素的位置，不论是否成功——对 map 而言(**始自C++11**)
c.emplace_hint(pos , args . . .)	安插一个以 args 为初值的元素，并返回新元素的位置(pos 是个提示，指出安插动作的查找起点。若提示恰当可加快速度)
c.erase(val)	移除"与 val 相等"的所有元素，返回被移除的元素个数
c.erase(pos)	移除 iterator 位置 pos 上的元素，无返回值
c.erase(beg , end)	移除区间 [beg, end) 内的所有元素，无返回值
c.clear()	移除所有元素，将容器清空

前面介绍的set/multiset集合操作的方法同样适用于map/multimap，包括集合的建立方法、成员函数、比较运算、排序规则和方法等，只需要将其中的set更改为map,将multiset更改为multimap就行了。需要说明的只有insert函数和元素访问的不同。

insert成员函数

从形式上看，map/multimap集合的insert成员都具有相同的形式：

```
insert(e)
```

但insert插入到map/multimap和set/multiset的元素是有区别的。插入到set/multiset中的元素是单独的键，而插入到map/multimap中的元素是<键，值>构成的一对数据，这对数据是一个不可分割的整体。

map/multimap的<键，值>可用make_pair函数构造，形式如下：

```
make_pair(e1, e2); //e1代表键  e2代表值
```

元素访问

map/multimap映射的元素是由<键，值>对构成的，且同一个键可以对应多个不同的值，可以通过相关映射的迭代器访问他们的元素。

map/multimap类型的迭代器提供了两个数据成员:一个是first，用于访问键；一个是second用于访问值。

此外map类型的映射可以用键作为数组下标，访问该键所对应的值，但multimap类型的映射不允许用数组下标的方式访问其中的元素。

实例1： 用map查询员工工资

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

int main(void){

    string name[]={"关羽","张飞", "赵云"};
    double salary[]={15000,14000,13000};

    map<string, double> sa1;
    map<string, double>::iterator p; // 迭代器

    for(int i=0; i<3; i++){
        sa1.insert(make_pair(name[i], salary[i]));
    }

    /*通过下标运算加入新元素*/
    sa1["诸葛亮"] = 20000;
    sa1["黄忠"] = 12000;

    for(p=sa1.begin(); p!=sa1.end(); p++){
        cout << p->first <<"\t" << p->second << endl;
    }

    string person;
    cout << "请输入查找人员姓名: ";
    cin >> person;
    for(p=sa1.begin(); p!=sa1.end(); p++){
        if(p->first == person)
            cout << p->second << endl;
    }

    return 0;
}
```

map和multimap的用法基本相同，区别在于map映射中的键不允许重复，而multimap中的键允许重复。此外map允许数组的下标运算访问映射中的值，而multimap是不允许的，multimap在构造一键对多值得查询时非常有用。

实例2：用multimap构造汉英对照词典

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

int main(void){
    multimap<string, string> dict;
    multimap<string, string>::iterator p;

    string eng[]={"plot", "gorge", "cliff", "berg", "precipice", "tract"};
    string che[]={"小块地，地点", "峡谷", "悬崖", "冰山", "悬崖", "一片区域"};

    for(int i=0; i<6; i++){
        dict.insert(make_pair(eng[i], che[i]));  //批量插入
    }
    //插入单个元素
    dict.insert(make_pair(string("tract"),string("地带")));

    for(p=dict.begin(); p!=dict.end(); p++)
```

```
        cout << p->first << "\t" << p->second << endl;

    string word;
    cout << "请输入要查找的英文单词: ";
    cin >> word;
    for(p=dict.begin(); p!=dict.end(); p++){
        if(p->first == word){
            cout <<p->second << endl;
        }
    }

    cout<< "请输入要查找的中文单词: ";
    cin >> word;
    for(p=dict.begin(); p!=dict.end(); p++){
        if(p->second == word){
            cout <<p->first << endl;
        }
    }

    return 0;
}
```

1.4.4 算法

算法(algorithm)是用于模板技术实现的适用于各种容器的通用程序。算法常常通过迭代器间接地操作容器元素，而且通常会返回迭代器作为算法运算的结果。

STL大约提供了70个算法，每个算法都是一个模板函数或者一组模板函数，能够在许多不同类型的容器上进行操作，各个容器则可能包含着不同类型的数据元素。STL中的算法覆盖了在容器上实施的各种常见操作，如遍历、排序、检索、插入及删除元素等操作。STL中许多算法不仅适用于系统提供的容器类，而且适用于普通的C++数组或自定义容器。

下面介绍几个常用算法：

- **find和count算法**

  find用于查找指定数据在某个区间中是否存在，该函数返回等于指定值的第一个元素位置，如果没找到就返回最后元素位置；count用于统计某个值在指定区间出现的次数，其用法如下：

```
find(beg, end, value);
count(beg, end, value);
```

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

int main(void){

    int a1[] = {100,200,300,400,500,600,700,800,900,1000};

    int *ptr = find(a1, a1+10, 800);
    cout << "400出现在数组a1的位置:   " << ptr - a1 << endl;

    list<int> L1;
    int a2[] = {20,30,40,50,60,60,70,60,80};
    for(int i=0; i<sizeof(a2)/sizeof(a2[0]); i++){
        L1.push_back(a2[i]);
    }
    list<int>::iterator it;
    it = find(L1.begin(), L1.end(), 80);
    if(it != L1.end()){
```

```
        cout << "L1链表中存在元素: " << *it << endl;
        cout << "它是链表中第: " << distance(L1.begin(), it) + 1 << "个节点"<< endl;
    }

    int n1 = count(a1, a1+10, 500);
    cout << "a1 数组中 500出现了: "<<n1<< "次"<<endl;
    int n2 = count(L1.begin(), L1.end(), 60);
    cout << "L1 容器中  6 0 出现了: " <<n2 <<"次"<< endl;

    return 0;
}
```

- **search算法**

  find算法从一个容器中查找指定的值，search算法则是从一个容器查找由另一个容器所指定的顺序值。search用法如下所示：

```
search(beg1, end1, beg2, end2);//左闭右开
```

  search将在[beg1, end1)区间查找有无与[beg2, end2)相同的子区间，如果找到就返回[beg1, end1)内第一个相同元素的位置，如果没有找到返回end1;

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
using namespace std;

int main(void){

    int a1[] ={10,20,30,40,50,60,70,80,90};
    int a2[]= {70,80,90};

    int *ptr = search(a1, a1+9, a2, a2+3);
    if(ptr == a1+9)
        cout << "not match" << endl;
    else
        cout << "math: " << ptr - a1 << endl;

    list<int> L;
    vector<int> V;
    for(int i=0; i<9; i++){
        L.push_back(a1[i]);
    }
    for(int i=0; i<3; i++){
        V.push_back(a2[i]);
    }
    list<int>::iterator pos;
    pos = search(L.begin(), L.end(), V.begin(), V.end());

    cout << distance(L.begin(), pos) << endl;

    return 0;
};
```

- **merge**

  merge可对两个容器进行合并，将结果存放在第3个容器中，其用法如下：

```
merge(beg1, end1, beg2, end2, dest)
```

merge将[beg1,end1)与[beg2, end2)区间合并，把结果存放在dest容器中。如果参与合并的两个容器中的元素是有序的，则合并的结果也是有序的。

list链表也提供了一个merge成员函数， 它能够把两个list类型的链表合并在一起。同样地，如果合并前的链表是有序的，则合并后的链表仍然有序。

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
using namespace std;

int main(void){

    int a1[] ={10,20,30,40,50,60,70,80,90};
    int a2[]= {70,80,90, 100};

    int a3[13] = {0};

    merge(a1, a1+9, a2, a2+4, a3);
    for(int i=0; i<13; i++){
        cout << a3[i] << "\t" ;
    }
    cout << endl;

    list<int> L1;
    list<int> L2;
    for(int i=0; i<9; i++){
        L1.push_back(a1[i]);
    }
    for(int i=0; i<4; i++){
        L2.push_back(a2[i]);
    }
    L1.merge(L2);
    list<int>::iterator it;
    for(it=L1.begin(); it!=L1.end(); it++){
        cout << *it << "\t";
    }
    cout << endl;

    return 0;
}
```

- **sort**

sort可对指定容器区间内的元素进行排序，默认排序的方式为从小到大，其用法入下：

```
sort(beg, end);
```

[beg, end)是要排序的区间，sort将按从小到大的顺序对该区间的元素进行排序。

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
using namespace std;
```

```cpp
int main(void){
    int a1[] ={10,2,-30, 840,550,6,120};
    sort(a1, a1+7);
    for(int i=0; i<7; i++){
        cout << a1[i] << "\t";
    }
    cout << endl;

    int a2[] ={10,2,-30, 840,550,6,120};
    vector<int> v;
    vector<int>::iterator it;
    for(int i=0; i<7; i++){
        v.push_back(a2[i]);
    }
    sort(v.begin(), v.end());
    for(it=v.begin(); it!=v.end(); it++){
        cout << *it << "\t";
    }
    cout << endl;

    return 0;
}
```