



中國人民大學  
RENMIN UNIVERSITY OF CHINA

程序设计荣誉课程

## 2 函数

授课教师：孙亚辉

# 本章内容

1. 函数基本概念
2. 函数重载
3. 操作符重载
4. 头等函数

# 1. 函数基本概念

- 函数声明：负责指定函数的名字，返回值类型，以及调用该函数所需参数数量和类型。函数声明可帮助编译器正确验证函数返回值、参数数目和参数类型是否正确。
- 函数定义：函数定义是特殊的函数声明，它给出了函数体的内容。

```
void swap(int*, int*);  
void swap(int *p, int *q) {  
    int t = *p;  
    *p = *q;  
    *q = t;  
}
```

# 参数传递

- 参数传递：当调用一个函数时，参数值将传给该函数。
  - 形参：在函数定义中出现的参数可以看做是一个占位符，它没有数据，只能等到函数被调用时接收传递进来的数据，所以称为形式参数，简称形参。
  - 实参：函数被调用时给出的参数包含了实实在在的数据，会被函数内部的代码使用，所以称为实际参数，简称实参。

```
1. int g(int x) {  
2.     //x是函数g的形参  
3.     return 2*x;  
4. }  
5. void f() {  
6.     int y;  
7.     scanf("%d", &y);  
8.     //y是调用函数g时给定的实参  
9.     g(y);  
10. }
```

# 参数传递的三种形式

- 值传递：实参变量的值被复制到对应的形参变量中去，成为形参变量的初始值，**函数中语句对形参变量的操作与实参变量无关。**
- 指针传递：函数的形参是指针类型，实参是目标变量的地址，调用时实参地址值赋给形参指针变量，**被调函数可以通过形参指针间接访问实参地址所指向的变量。**
- 引用传递：函数的形参是引用类型，实参是目标对象，形参作为引用绑定在实参标识的对象上，**对形参的操作就是对实参对象的操作**，函数执行结束后撤销引用绑定。

# 实例：参数传递的三种形式

test.cpp

```
1  #include<iostream>
2  using namespace std;
3
4  void swap1(int a,int b){int t; t=a; a=b; b=t;}
5  void swap2(int* a,int* b){int t; t=*a; *a=*b; *b=t;}
6  void swap3(int &a,int &b){int t; t=a; a=b; b=t;}
7
8  int main()
9  {
10     int x=10,y=20;
11     swap1(x,y);
12     cout<<x<<' '<<y<<endl;
13     swap2(&x,&y);
14     cout<<x<<' '<<y<<endl;
15     swap3(x,y);
16     cout<<x<<' '<<y<<endl;
17     return 0;
18 }
19
```

Yahui [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Activities

Terminal

Mar 20



yahui@Yah

```
yahui@Yahui: /media/sf_VM$ g++ test.cpp
```

```
yahui@Yahui: /media/sf_VM$ ./a.out
```

```
10 20
```

```
20 10
```

```
10 20
```

# 参数传递： 指针的引用

```
1  #include <iostream>
2  using namespace std;
3
4  void func(int& x, int y, int *p)
5  {
6      p = &x;
7      x = 3;
8  }
9
10 int main()
11 {
12     int x = 1, y = 2;
13     int *p = &y;
14     func(x, y, p);
15     cout << *p << endl;
16
17     return 0;
18 }
```

```
1  #include <iostream>
2  using namespace std;
3
4  void func(int& x, int y, int *&p)
5  {
6      p = &x;
7      x = 3;
8  }
9
10 int main()
11 {
12     int x = 1, y = 2;
13     int *p = &y;
14     func(x, y, p);
15     cout << *p << endl;
16
17     return 0;
18 }
```


# const约束形参的访问特性

为避免被调函数修改实参所指的变量，可用const约束形参指针的访问特性

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

void swap1(int a,int b){int t; t=a; a=b; b=t;}
void swap2(const int* a, const int* b){int t; t=*a; *a=*b; *b=t;}
void swap3(const int &a, const int &b){int t; t=a; a=b; b=t;}

int main()
{
    int x=10,y=20;
    swap1(x,y);
    cout<<x<<' '<<y<<endl;
    swap2(&x,&y);
    cout<<x<<' '<<y<<endl;
    swap3(x,y);
    cout<<x<<' '<<y<<endl;
    return 0;
}
```



```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'void swap2(const int*, const int*)':
test.cpp:5:54: error: assignment of read-only location '* a'
      5 | void swap2(const int* a,const int* b){int t; t=*a; *a=*b; *b=t;}
        |                                                     ~~~~
test.cpp:5:61: error: assignment of read-only location '* b'
      5 | void swap2(const int* a,const int* b){int t; t=*a; *a=*b; *b=t;}
        |                                                     ~~~~
test.cpp: In function 'void swap3(const int&, const int&)':
test.cpp:6:52: error: assignment of read-only reference 'a'
      6 | void swap3(const int &a,const int &b){int t; t=a; a=b; b=t;}
        |                                                     ~~~
test.cpp:6:57: error: assignment of read-only reference 'b'
      6 | void swap3(const int &a,const int &b){int t; t=a; a=b; b=t;}
        |                                                     ~~~
```



# 默认参数值

- 默认参数：在定义参数时同时给它一个初始值。在调用函数时，可省略含有默认值的参数。
- 如果用户指定了参数值，则使用用户指定的值，否则使用默认参数的值。

```
1. int def(int a, int b = 10) { return a + b; }  
2. void f() { def(1); }  
3. void g() { def(1, 3); }
```

①

```
1. void f() {  
2. ....  
3.     def(1);  
4.     80483eb: push    0xa // 0xa=10  
5.     80483ed: push    0x1  
6.     80483ef: call   80483db <_Z3defii>  
7.     80483f4: add    esp,0x8  
8. }
```

②

```
1. void g() {  
2. ....  
3.     def(1, 3); }  
4.     80483fd: push    0x3  
5.     80483ff: push    0x1  
6.     8048401: call   80483db <_Z3defii>  
7.     8048406: add    esp,0x8  
8. }
```

③

# 本章内容

1. 函数基本概念
2. 函数重载
3. 操作符重载
4. 头等函数

## 2. 函数重载

- 函数重载：同一个函数名对应着多个函数实现的情况。
- 函数重载的关键：函数参数列表（函数特征标）。特征标不同的同名函数存在重载关系。

# 函数重载注意事项


- 若两个函数的参数数目和类型相同，同时参数排序也相同，则它们的函数特征相同。
- 函数的返回值类型不影响函数特征。

例: `void f(int a, int b) {};` `int f(int a, int b) {};` //not overloaded

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

void f(int a, int b) {}
int f(int a, int b) {}

int main(){
    f(1, 1);
}
```



```
test.cpp:6:5: error: ambiguating new declaration of 'int f(int, int)'
    6 | int f(int a, int b) {}
      |     ^
```

# 函数重载注意事项


- 类型引用和类型本身视为相同的特征，但类型指针和类型本身视为不同的特征。

例: `void f(int a){}; void f(int &a) {}; //not overloaded`  
`void f(int a) {}; void f(int *a) {}; //overloaded`

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

```
int f(int a) { return 1; }
int f(int &a) { return 2; }
```

```
int main()
{
    int x = 1;
    cout << f(x) << endl;
}
```




```
test.cpp:10:14: error: call of overloaded 'f(int&)' is ambiguous
10 |     cout << f(x) << endl;
    |               ^
```

从编译器角度思考：对于{ int x;  
f(x); }, 编译器无法区分该使用f(int  
a)还是f(int &a)

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

```
int f(int a) { return 1; }
int f(int *a) { return 2; }
```

```
int main()
{
    cout << f(1) << endl;
}
```



打印结果：1

# 函数重载注意事项

问题： 以下两对函数是否形成重构

- void f(int \*a) {};      v.s.      void f(int &a) {};

# 编译器如何进行重载解析

- **单实参解析**：按如下顺序尝试一系列评判准则，如果某次函数调用在能找到匹配的最高层级发现了不止一个可用匹配，则因产生二义性被拒绝。
  - 精确匹配：无须类型转换或仅需简单的类型转换（如数组名/函数名转换成指针，类型T转换成const T）
  - 执行提升后匹配：整数提升（bool/char/short转换成int，上述转换的unsigned版本），浮点数提升（float转换成double）
  - 执行标准类型转换后实现匹配：除整数提升和浮点数提升以外的类型转换（例如之后讲的nullptr到const char\*的类型转换）

# 编译器如何进行重载解析

- **多实参解析**：对于一组重载函数以及一次调用来说，如果该调用对于各函数的参数类型在计算的效率和精度上差别明显，则可应用重载解析规则从中选出最合适的。
  - 将单实参解析规则作用于每一个实参，选出基于该实参的最佳匹配规则
  - 若某个函数是其中一个实参的最佳匹配，同时基于其它实参也是最优的匹配或至少不弱于别的函数，则该函数被最终确定为是最佳匹配函数
  - 若找不到符合上述条件的函数，则本次调用因二义性被拒绝



# 重载解析实例

- 多实参解析实例

```
1  #include <iostream>
2  using namespace std;
3
4  void pow(int a, int b) {}
5  void pow(double a, double b) {}
6
7  int main()
8  {
9      pow(2.0, 2);
10 }
```

- 若某个函数是其中一个实参的最佳匹配，同时基于其它实参也是最优的匹配或至少不弱于别的函数，则该函数被最终确定为是最佳匹配函数
- 若找不到符合上述条件的函数，则本次调用因二义性被拒绝

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:9:13: error: call of overloaded 'pow(double, int)' is ambiguous
   9 |     pow(2.0, 2);
     |           ^
test.cpp:4:6: note: candidate: 'void pow(int, int)'
   4 | void pow(int a, int b) {}
     |           ^~~
test.cpp:5:6: note: candidate: 'void pow(double, double)'
   5 | void pow(double a, double b) {}
     |           ^~~
```

# 本章内容

1. 函数基本概念
2. 函数重载
3. 操作符重载
4. 头等函数

# 3. 运算符重载

- 运算符重载：对C++已有的运算符重新进行定义，赋予其另一种功能。
- C++语言本身就重载了很多运算符，例如<<是位运算中的左移运算，但在输出操作中又与流对象cout配合成为流插入运算符。
  - `int i = 1 << 4;`
  - `cout << "hello world" << 2020;`
- C++允许程序员重载大部分运算符。虽然重载运算符的任务也可通过显式的函数调用来完成，但使用运算符重载往往可让程序更加清晰。
  - `Complex a, b, c; c = add(a, b);`
  - `Complex a, b, c; c = a + b;`

# 运算符重载的形式

形式:

返回类型	<code>operator</code> 运算符符号（形式参数列表）
{	
函数体	
}	

- 本质上，运算符重载是一种特殊的函数重载，其特殊之处
  - 函数的名称：`operator`运算符符号
  - 函数的参数：参数个数由具体运算符的特质决定

```
C: > Users > Yahui > Documents > Drive > VM > test.cpp > A
1  #include <iostream>
2  using namespace std;
3
4  class A
5  {
6  public:
7      A(int x, int y) : a(x), b(y) {}
8      int a, b;
9  };
10
11  A operator+(A m, A n)
12  {
13      A k(m.a + n.a, m.b + n.b);
14      return k;
15  }
16
17  int main()
18  {
19      A m(1, 2), n(3, 4);
20      cout << (m + n).a << " " << (m + n).b << endl;
21  }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
4 6
```

# 运算符重载的规则

- C++绝大部分运算符都可以重载，不能重载的运算符包括：
  - `.` `*` `::` `?:`
  - `sizeof` `typeid`
  - `const_cast` `dynamic_cast` `reinterpret_cast` `static_cast`
- 不改变运算符的优先级、结合性和运算对象数目。
- 运算符重载函数不能使用默认参数。
- 重载运算符必须具有一个类对象（或类对象的引用）的参数，不能全部参数是C++内置数据类型。
- 不能创建新运算符，如`operator**()`表示求幂✗。

## 3.1 运算符重载为类成员函数

- 当运算符重载为类的成员函数时，运算符函数的形参个数比运算符规定的运算对象个数要少一个。
- 原因：类的非静态成员函数都有一个隐含的 **this** 指针（作为第一个参数），运算符函数可以通过 **this** 指针隐式地访问类对象的成员，因此这个对象自身的数据可直接访问，不需要放到形参列表中进行传递。

# 运算符重载为类成员函数

本质上，运算符重载是一种特殊的函数重载。

ebp-0x3c是m的地址，  
ebp-0x34是n的地址

```
class A{
public:
    A(int x, int y) : a(x), b(y) {}
    A operator+(A n)
    {
        A k(a + n.a, b + n.b);
        return k;
    }
    int a, b;
};
```

打印结果: 4 6

```
int main(){
    A m(1, 2), n(3, 4);
    cout << (m + n).a << " " <<
    (m + n).b << endl;
}
```

A::operator+(A)    A::A(int, int)

A m(1, 2), n(3, 4);

```
12ba: 83 ec 04
12bd: 6a 02
12bf: 6a 01
12c1: 8d 45 c4
12c4: 50
12c5: e8 50 01 00 00
12ca: 83 c4 10
12cd: 83 ec 04
12d0: 6a 04
12d2: 6a 03
12d4: 8d 45 cc
12d7: 50
12d8: e8 3d 01 00 00
12dd: 83 c4 10
12e0: 8d 45 d4
12e3: ff 75 d0
12e6: ff 75 cc
12e9: 8d 55 c4
12ec: 52
12ed: 50
12ee: e8 4d 01 00 00
```

```
sub esp,0x4
push 0x2
push 0x1
lea eax,[ebp-0x3c]
push eax
call 141a <_ZN1A1Eii>
add esp,0x10
sub esp,0x4
push 0x4
push 0x3
lea eax,[ebp-0x34]
push eax
call 141a <_ZN1A1Eii>
add esp,0x10
lea eax,[ebp-0x2c]
push DWORD PTR [ebp-0x30]
push DWORD PTR [ebp-0x34]
lea edx,[ebp-0x3c]
push edx
push eax
call 1440 <_ZN1AplES_>
```

编译时实际有两个参数：m的this指针、n对象



# 单目运算符的重载

单目运算符是指运算仅需一个变量的运算符。

- 前置单目运算符：重载为类的成员函数时没有形参
- 后置单目运算符：重载为类的成员函数时需一个int型形参

```
1. class Complex {
2.     private:
3.         int real, imag; //real:实部 imag:虚部
4.     public:
5.         Complex(int r=0, int i=0) { real = r; imag = i; } //构造函数
6.         void print(void) { cout << real << " " << imag << endl; } //输出信息
7.         Complex operator+(Complex &c2); //重载+运算符为类成员函数
8.         Complex operator++(); //用于++i; 重载前置++运算符为类成员函数
9.         Complex operator++(int); //用于i++;重载后置++运算符为类成员函数
10. };
```

# 单目运算符的重载

```
1. Complex Complex::operator++(int) {  
2.     Complex a(*this); ++real; ++imag; return a;  
3. }  
7. int main() {  
8.     Complex a(1, 2);  
9.     Complex b = a++;  
10.    b.print(); //输出值是什么: 1, 2 还是 2, 3?  
11. }
```

- 怎么换成C++标准中的后置++的使用方式呢？

# 双目运算符的重载1

```
1. class Complex2;
2. class Complex1 {
3.     private:
4.         int real, imag; //real:实部 imag:虚部
5.     public:
6.         Complex1(int r=0, int i=0) { real = r; imag = i; }
7.         Complex1 operator+(Complex2 &c2);
8.         Complex1 operator+(int a);
9. };
10. class Complex2 {
11.     private:
12.         int real, imag; //real:实部 imag:虚部
13.     public:
14.         Complex2(int r=0, int i=0) { real = r; imag = i; }
15.         Complex2 operator+(Complex1 &c2);
16. };

17. int main() {
18.     Complex1 a(1, 2);
19.     Complex2 b(2, 3);
20.     Complex1 c = a + b;      //调用Complex1::operator+(Complex2)
21.     Complex2 d = b + a;      //调用Complex2::operator+(Complex1)
22.     a + 1;                   //调用Complex1::operator+(int)
23.     1 + a;                   //错误。怎么办?
24. //解决方法: 在类的外面再定义Complex1 operator+(int a, Complex1 b){...}
25. }
```

# 双目运算符的重载2

**注意：**调用等号运算符重载函数时，等号左右两边的对象必须已经被创建，否则调用的是默认的拷贝构造函数。

```
1  #include <iostream>
2  #include <cstring>
3  #include <vector>
4  using namespace std;
5
6  class X
7  {
8  public:
9      X() { cout << "constructor." << endl; }
10     X(const X &tmp) { cout << "copy constructor." << endl; }
11     X &operator=(X &tmp)
12     {
13         cout << "assign operator." << endl;
14         return *this;
15     }
16 };
17 int main()
18 {
19     X a;
20     X b = a;
21     a = b;
22 }
23
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
constructor.
copy constructor.
assign operator.
```

## 3.2 运算符重载为友元函数

类的友元函数是定义在类外部、但有权访问类的所有私有`private`成员和保护`protected`成员的函数。

- 友元函数的用途：
  - C++控制对类对象私有成员的访问。通常，公有类方法提供唯一的访问途径，但是这种限制有时过于严格。
  - 通过让函数成为类的友元，可赋予该函数与类的成员函数相同的访问权限。
  - 在进行双目运算符重载时，参与运算的两个对象类型可能不同，彼此之间不能互相访问私有成员变量，需借助友元函数突破访问限制。
- 当运算符重载为友元函数时，运算符函数的形参个数与运算符规定的运算对象个数相同（因为类的友元函数并不是类的成员函数）。

C: > Users > Yahui > Documents > Drive > VM > test.cpp > Complex2

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  class Complex2; // this is needed to claim early, sin
6
7  class Complex1
8  {
9  private:
10     int real, imag; //real:实部 imag:虚部
11 public:
12     Complex1(int r = 0, int i = 0)
13     {
14         real = r;
15         imag = i;
16     }
17     friend Complex1 operator+(Complex1 &c1, Complex2 &c2);
18     friend Complex1 operator+(int, Complex1 &c);
19 };
20 class Complex2
21 {
22 private:
23     int real, imag; //real:实部 imag:虚部
24 public:
25     Complex2(int r = 0, int i = 0)
26     {
27         real = r;
28         imag = i;
29     }
30     Complex2 operator+(Complex1 &c2);
31     friend Complex1 operator+(Complex1 &c1, Complex2 &c2);
32 };
```

```
34 Complex1 operator+(Complex1 &c1, Complex2 &c2)
35 {
36     Complex1 ret(c1.real + c2.real, c1.imag + c2.imag);
37     return ret;
38 }
39
40 Complex1 operator+(int val, Complex1 &c)
41 {
42     Complex1 ret(c.real + val, c.imag);
43     return ret;
44 }
45
46 int main()
47 {
48     Complex1 a(1, 2);
49     Complex2 b(2, 3);
50     Complex1 c = a + b; //调用operator+(Complex1, Complex2)
51     1 + a;              //调用operator+(int, Complex1)
52 }
```

# 冲突与二义性

- 以类成员函数方式重载的运算符，其冲突空间是类作用域，即：一个类中，不能存在同一个运算符的两个函数特征标相同的重载版本。
- 以友元函数方式重载的运算符，其冲突空间是其所在命名空间，即一个命名空间中不能存在同一个运算符的两个函数特征标相同的重载版本。
- 如果对于同一个运算符，同时存在以类成员函数方式重载的版本和以友元函数方式重载的版本，标准里没有明确定义，但是g++在编译时会报错。

```
class Complex2; // this is needed to claim early, since Complex2 is used in
Complex1
class Complex1 {
private:
    int real, imag; //real:实部 imag:虚部
public:
    Complex1(int r=0, int i=0) { real = r; imag = i; }
    Complex1 operator+(Complex2 &c2);
    friend Complex1 operator+(Complex1 &c1, Complex2 &c2);
    friend Complex1 operator+(int, Complex1 &c);
};
```

**error: ambiguous overload for 'operator+' (operand types are 'Complex1' and 'Complex2')**

# 本章内容

1. 函数基本概念
2. 函数重载
3. 操作符重载
4. 头等函数



## 4. 头等函数

- 如果编程语言允许将函数当作其它任何变量，则称该语言提供头等函数。
  - 将函数作为一个值赋给变量
  - 将一个函数作为实参传递给其他函数
  - 从一个函数中返回另一个函数作为结果
- C++中头等函数的实现方式：
  - 函数指针
  - 函数对象

## 4.1 函数指针

- 函数是实现特定功能的程序代码的集合。函数的代码在内存中也要占据一段存储空间，这段存储空间的起始地址称为函数的入口地址。
- 函数指针是一个变量，存储函数的入口地址，在程序执行的不同时期可指向不同函数。仅使用入口地址不足以调用一个函数。为正确工作，**函数指针的定义还必须存储每个参数类型以及返回值类型**。
- 在给函数指针变量赋值时，只能赋予类型匹配的函数的地址；在使用函数指针进行函数调用时，给定的参数和接收的返回值类型要与定义相匹配。

# 函数指针的定义与使用

- 定义：

返回类型 （\*函数指针变量名）（形式参数列表）；

- 使用方法（在使用函数指针进行函数调用时，给定的参数和接收的返回值类型要与定义相匹配）：

1. 函数指针变量名 = 函数名；
2. 函数指针变量名（实际参数列表）；

```
int test(int value){return value + 1;}  
int main(){  
    int (*f)(int);  
    f = test; // f成为了test函数的指针  
    cout << f(10) << endl;} // f(10)相当于test(10)
```

# 函数指针要求参数和返回值类型均严格匹配

```
1  #include <iostream>
2  using namespace std;
3
4  int test(int value)
5  {
6      return value + 1;
7  }
8
9  double test2(double value)
10 {
11     return value + 1;
12 }
13
14 int main()
15 {
16     int (*f)(double);
17     f = test;
18     f = test2;
19     cout << f(10) << endl;
20 }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function 'int main()':
test.cpp:17:9: error: invalid conversion from 'int (*)(int)' to 'int (*)(double)'
' [-fpermissive]
   17 |     f = test;
      |         ^~~~
      |         |
      |     int (*)(int)
test.cpp:18:9: error: invalid conversion from 'double (*)(double)' to 'int (*)(double)'
' [-fpermissive]
   18 |     f = test2;
      |         ^~~~~
      |         |
      |     double (*)(double)
```

# 函数指针的意义

- 指向函数的指针多用于指向不同函数，从而可以利用指针变量调用不同函数，相当于将函数调用由静态方式（固定调用指定函数）变为动态方式（调用哪个函数由指针值来确定）。
- 应用函数指针，有利于程序的模块化设计，提高程序的可扩展性。
- 一些用C语言编写的大型程序（如Linux内核），多采用结构体+函数指针的方式实现类似于面向对象语言中的多态特性。
- Linux内核虚拟文件系统代码：

```
struct file{
    struct dentry *f_dentry;
    struct file_operations *f_op;
    ...
};

struct file_operations {
    size_t (*read) (struct file *, char __user *, size_t, loff_t *);
    size_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    ...
};
```

## 4.2 函数对象

- 函数对象可以理解成是函数指针面向对象的改进（在C++中，我们把所有能当做函数使用的对象统称为函数对象）。
  - 相似：函数对象的行为完全类似于函数
  - 不同：函数对象是完整的类对象，有自己的成员变量，可能还有其它成员函数
- 定义

```
class 函数对象类名 {  
    public: 返回值类型 operator() (形式参数列表)  
    { 函数体 }  
};
```

- 使用：

1. 函数对象类名 函数对象实例；
2. 函数对象实例（实际参数列表）

```
class Test{  
public:  
    int operator()(int value) { return  
value + 1; } };  
int main(){  
    Test test_function; //函数对象  
    int value;  
    value = test_function(10); }
```

# 函数对象的本质

- 函数对象的本质：对操作符()的重载

```
1.  class Test { public: int operator()(int value) { return value+1; } };
2.  int main() {
3.      Test test;
4.      int value;
5.      value = test(10);
6.  }
```

```
1.  int main() {
2.      .....
3.      Test test;
4.      int value;
5.      value = test(10);
6.      8048457:    sub     esp,0x8
7.      804845a:    push   0xa
8.      804845c:    lea     eax,[ebp-0x11]
9.      804845f:    push   eax
10.     8048460:    call    804848a <_ZN4TestclEi>
11.     8048465:    add     esp,0x10
12.     8048468:    mov     DWORD PTR [ebp-0x10],eax
13. }
```


Test::operator()(int)



# 实例：C++标准库的排序算法

- 对自定义类型数据的排序

```
1. #include <algorithm>
2. struct Student { int id; char name[20]; };
3. struct Compare { // C++标准库的排序算法里面的compare就是一个函数对象
4.     bool operator() (const Student &s1, const Student &s2) {
5.         return s1.id < s2.id;
6.     }
7. };
8. Student students[] = { {2019161, "Jack"}, {2019162, "Mike"}, {2019163, "Rose"}};
9. int main() {
10.     n = sizeof(students) / sizeof(Student);
11.     std::sort(students, students+n, Compare());
12. }
```



```
1. template <class RandomAccessIterator, class Compare> inline
2. void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
3.
4. template <class RandomAccessIterator, class T, class Compare> inline
5. void __linear_insert(RandomAccessIterator first, RandomAccessIterator last, T*,
6.     Compare comp) {
7.     T value = *last;
8.     if (comp(value, *first)) {
9.         copy_backward(first, last, last + 1);
10.         *first = value;
11.     }
12.     .....
13. }
```



# 作业

- YOJ-338. 重载函数getpower
- YOJ-346. 设计复数类
- YOJ-421. C++复数运算符重载（+与<<）（尝试将运算符重载为类成员函数和友元函数两种方法）