



华中科技大学

面向对象程序设计

第二章 C++基础

许向阳

xuxy@hust.edu.cn





学习内容

输入输出的流式运算

命名空间

const 修饰符

宏

内联函数

函数缺省参数 函数常参数 引用参数

内部(或静态)函数

指针 引用





2.1 C++语言

- C的超集，完全兼容C，代码质量高、速度快。
- 多继承的**强类型**的混合型的OO语言

- 为所有变量指定数据类型

- 先声明后使用

- 不同类型的变量相互操作时应加类型转换

```
int    p1 = 6;
```

```
float  p2 = 10.5;
```

```
p2 = (float)p1;
```

```
p1 = (int)p2;
```

```
p2 = p1; //警告：从int转换到float，可能丢失数据
```

```
p1 = p2; //警告：从float转换到int，可能丢失数据
```





2.1 C++语言

- 支持面向对象的运算符重载
(至少一个操作数的类型代表对象的类型)
 - + : 两个整型数相加
 - + : 两个复数相加 (复数类中的加法运算)
 - + : 两个矩阵相加 (矩阵类中的加法运算)
- 提供函数模板和类模板等高级抽象机制
- 支持面向对象的异常处理
- 支持名字空间 namespace





2.1 C++语言

- ◆ 支持C的注解 `/*...*/`
- ◆ 引入C++新注解 `//`到本行结束。
- ◆ 扩展名为.h的文件只应包含变量、函数的说明，其定义应出现在扩展名为.cpp的程序文件里。
 - 变量说明: `extern int x;` //说明只含类型和名称
 - 变量定义: `int x; int y=3;`
 - `extern int z=4;` //定义还包括初值
 - 函数说明: 没有函数体的原型声明。
`double sin(double);`
`extern "C" int fc(int)` //说明".c"文件中的函数 `fc`





2.1 C++语言

变量申明、变量定义、变量初始化
函数申明、函数定义

申明 VS 定义

分配空间 VS 不分配空间
给相应的存储空间赋值

推荐：自己给变量赋初值，

别用编译器赋予默认的值。

不同的编译器(不同版本)的做法可能不同





2.2 输入输出的流式运算

◆ 支持stdio.h:

`int scanf(const char *, ...);` // 返回成功输入变量的个数。
`int printf(const char *, ...);` // 返回成功输出字符的个数。

◆ 流iostream类对象cin通过重载>>运算符函数完成输入。

◆ >>可以自左至右连续运算。

`cin>>y>>z;`

等于 `cin>>y; cin>>z`

◆ cin和cout关于>>和<<运算的结果分别为cin和cout。

`cout << "my name is " << name << endl;`





2.2 输入输出的流式运算

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    char y[10];
    cin >> x;
    cin >> y;
    cout << "your input x=" << x << "; y is " << y << endl;
    return 0;
}
```

```
// 若不使用 using namespace std;
// 则在语句中 std::cout << x<<std::endl;
```





2.2 输入输出的流式运算

输出的格式控制

```
int x=12;  
cout << "8进制 x=" << oct << x << endl;  
cout << "16进制 x=" << hex << x << endl;
```

输出格式控制：

```
#include <iomanip>    // manipulators 操纵器  
// setw    设置输出数据的宽度  
// setprecision 设置小数点后的精度位数  
cout<<" 16 coded x=" << hex <<setw(10)<< x <<endl;
```





2.2 输入输出的流式运算

要求输入一个整数，而用户输入的不是整数，结果如何？

```
int x = 123;  
char s[20];  
cout << "input a integer :";  
cin >> x;  
cin >> s;  
// 若程序运行时，输入的不是数值  
// 虽然会报错，但是后面的数据不能输入  
// 会导致 x 的值为 0， 并且 s 串也没有输入
```





2.2 输入输出的流式运算

```
cout << cin.rdstate() << endl;    // 输入非数字，显示 2
while (cin.rdstate() != ios::goodbit) { // goodbit 是 0
    cout << "input error" << endl;
    cin.clear();    // 清除错误状态
    cout << cin.rdstate() << endl;
    // 此时，cin.rdstate 已变成 0 了。
    cin.ignore(numeric_limits<int>::max(), '\n');
    // ignore 设得足够大，实际起作用的是第二个参数。
    // 例如；    cin.ignore(20, '\n'); 同样，在输入不超过
    20个字符时起作用；忽略输入
    cin >> x;
}
```





2.3 const 修饰符

```
#define PI 3.14
```

```
const float PI=3.14; // PI 是一个常量，运行中不得修改
```

```
float const PI=3.14; // const float 等价于 float const
```

```
float x;    const float PI = x;    // 若x未初始化，则有警告
```

PI **float**

3.14 不可变

const float 是一种新的数据类型

x **float**

..... 可变





2.3 const 修饰符

```
const float PI=3.14;
```

```
float      x,  *q;
```

```
PI =4.2;
```

// Error : 不能给常量赋值

```
PI =x;
```

// Error : 不能给常量赋值

```
const float cf;
```

// Error: 常量变量 cf 需要初始值设定

```
x =PI;
```

```
q= &PI;
```

// 无法从const float * 转换为 float *

```
*q=5.6;
```

q  3.14 不可变 PI 与 *q=5.6;矛盾





2.3 const 修饰符

`const float PI=3.14;` // PI 是一个常量，运行中不得修改

`PI =4.2;`

`float *q = &PI;`

// 无法从 `const float *` 转换为 `float *`

// 防止出现 `*q=5.6;` 就是修改PI中的值

`const float *q = &PI;` // 因为不能修改*q,
// 也就不能修改q指向的PI;

`float u = PI;` // u是一个独立于PI的变量;
// 之后，修改u 的值，与 PI 无关

`float *q = (float *)&PI;` // 强制类型转换
// &PI 的类型是 `const float *`
// 转换为 `float *`

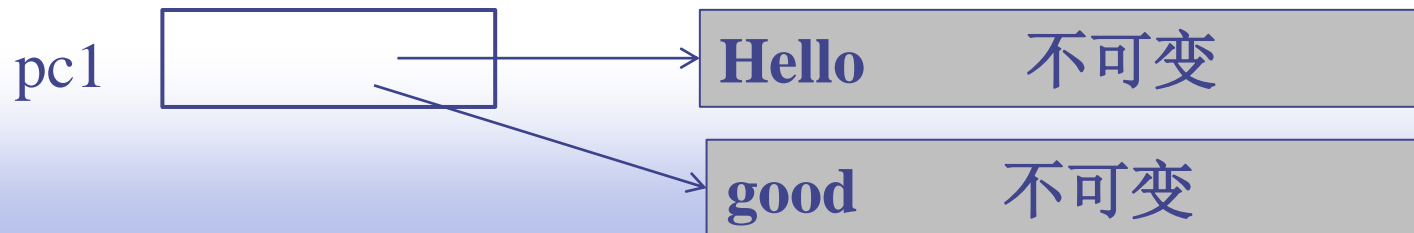


2.3 const 修饰符

```
char const * pc1; // pc1是一个指针，指向常量字符串
const char * pc2; // const char 于 char const 等价
const char * pc3 = "const string" ;
pc1="Hello";
pc1="good";
```

从右向左读，将 * 读成 pointer to
pc1 is a pointer to const char;

- 字符串中的内容不能修改
- 但指针中的值（即一个单元的地址）可以修改





2.3 const 修饰符

```
char *p1;    p1=new char[10];
```

```
char * const cp=p1;
```

cp 是一个常指针，指向字符串

cp 中的内容不可变，必须在定义时初始化

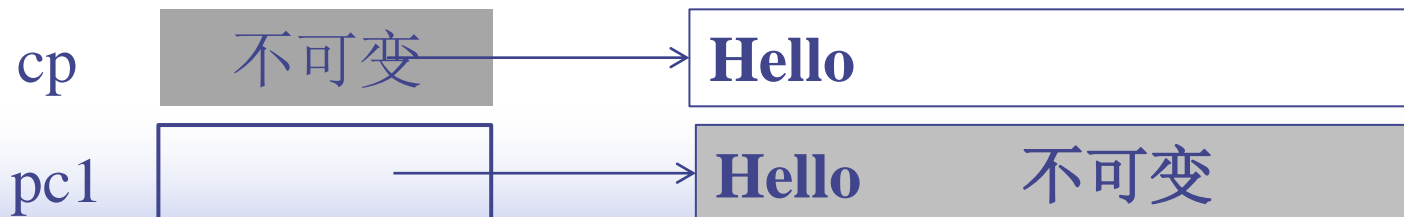
```
char * const cp1=new char[10];
```

```
char * const cp2 = "Hello"; // 无法从const char[6] 转换为char *
```

将 * 读成 pointer to, 从右向左读 char * const cp=p1;

cp is a const pointer to char;

```
char const * pc1;    pc1 is a pointer to const char;
```





2.3 const 修饰符

```
char s[]="good";  
char *p = new char[10];
```

```
const char *pc = s;  
pc[3] = 'g';  
pc = p;  
p=pc;  
p=(char *)pc;
```

```
char * const cp = s;  
cp[3] = 'g';  
cp = p;  
p = cp;
```

测验：判断语句的对错

const 放在类型名前
或放在变量名前

常变量：

在定义时，必须赋初值。
不能出现在赋值号左边。





2.3 const 修飾符

```
const char * const cccp = s;  
const char * const cccp = "good";
```

cccp

0x00123456



good

0x00123456



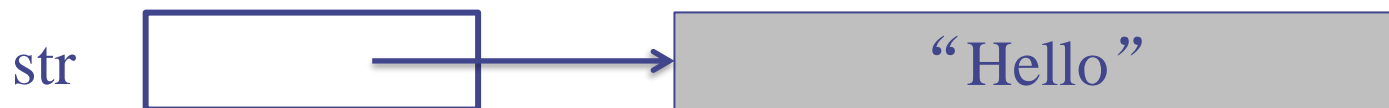
2.3 const 修饰符

const 修饰函数参数，防止在函数中修改参数数据。

例：C标准库中有：

```
int strlen(const char *str);
```

```
char *strcpy(char *dest, const char *source);
```



不能通过str来修改串中的内容， 函数的常参数

2.3 const 修飾符

const 修飾函數參數，防止在函數中修改參數數據。

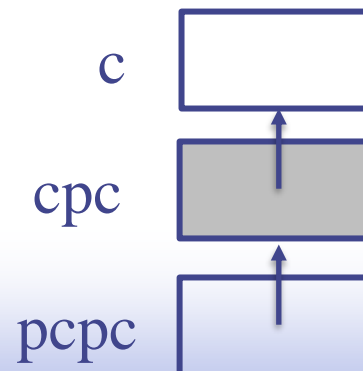
```
char * const * pcpc;
```

```
char ** q;
```



不能修改 ***pcpc** 中的內容；可修改 **pcpc**、**** pcpc**。

```
char c;  
char * const cpc=&c;  
pcpc = &cpc;
```





2.3 const 修饰符

讨论:

- 对于程序（或函数中），不需要（或不允许）变化的变量，加上const，有什么好处？
- 能不能通过其他手段，修改const 变量的值呢？
- 如何记忆有关规则？



2.3 const 修饰符

讨论:

- 能不能通过其他手段, 修改const 变量的值呢?

```
int xx;
```

```
cin >> xx;
```

// 输入 xx 为 100

```
const int yy = xx;
```

```
cout << "yy = " << yy << endl; // 显示 yy = 100
```

```
*(int*)&yy = 123;
```

```
cout << "yy = " << yy << endl; // 显示 yy = 123
```

原理:

- yy是 const int 类型, &yy 是 const int * 类型
- 采用强制地址类型转换 (int *),
将 const int *, 转换为 int *
- *(int *) 访问 yy





2.3 const 修饰符

讨论:

- 能不能通过其他手段, 修改const 变量的值呢?

```
*(int*)& yy = 123;  
    mov     dword ptr [yy], 7Bh
```

等价写法:

```
*const_cast<int*>(& yy) = 123;  
    mov     dword ptr [yy], 7Bh
```



2.3 const 修饰符

讨论:

- 能不能通过其他手段, 修改const 变量的值呢?

```
const int yy=xx;  
int *p;
```

p= &yy; // 无法从 const int * 转换为 int *

```
p = (int *)&yy;  
p= const_cast<int*>(&yy); // 与上一行等价  
// (&yy)这的括号不能省略  
*p=123;
```

此时, 显示 yy=123





2.3 const 修饰符

讨论:

- 地址类型的强制转换与数据类型转换的差别

```
int    xx=10;  
const int yy=xx;  
const int *p = &yy;
```

```
*p = 123;    // error
```

```
*(int *)p=123; // 强制将 p 的类型 (const int *)  
               // 转换为 int *  
               // 等价 *(int *)&yy=123;
```

此时，显示 yy=123





2.3 const 修饰符

讨论:

- 能不能通过其他手段, 修改const 变量的值呢?

```
const int yy = 100;  
cout << "yy = " << yy << endl; // 显示 yy = 100  
*(int*)&yy = 123;  
cout << "yy = " << yy << endl;
```

显示的 yy =?

显示的 yy =100

原理:

编译器看到 yy 是一个const int, 又给定了值100;
就认为 yy不再会改变, 后面直接用 100来代换了 yy。
*(int *)&yy语句是执行了的, 调试时看得到yy=123;
但 cout 的结果是 yy =100. 定义 const int yy=xx; 就无法给 yy 一个常量值。





2.3 const 修饰符

讨论:

- 能不能通过其他手段，修改const 变量的值呢？

```
const int yy = 100;    VS  int xx=100;  
                           const int yy = xx;
```

下面同样一段程序的执行结果不同

```
cout <<“yy = ”<< yy << endl;  // 显示 yy = 100
```

```
*(int*)& yy = 123;
```

```
cout <<“yy = ”<< yy << endl;  // 显示100 VS 显示 123
```

volatile const int yy = 100; // 后面访问 yy时，都要直接
// 访问对应的内存单元

上面程序 显示 yy =100
 yy = 123





2.3 const 修饰符

讨论:

- 能不能通过其他手段，修改const 变量的值呢？

```
int xx;  
cin >>xx;  
const int yy = xx;  
cout << "yy=" << yy<<endl;    // 显示输入xx的值  
*(&xx - 3) = 30;  
cout << "yy=" << yy<<endl;    // 显示30
```

// 注意，变量xx的地址与yy地址之间的关系
// 与编译器的设置有关





2.3 const 修饰符

思考题:

```
char * const p = new char[10];  
char q[20];
```

如何 让 p 能指向q，或者另一个新申请的空间？

```
*(char **)&p = q ;
```

```
*(char **)&p = new char[20];
```

```
*const_cast<char**>(&p) = q;
```

```
*const_cast<char**>(&p) = new char[20];
```





2.3 const 修饰符

思考题:

```
char* pc = "hello"; // 编译时报错  
// 无法从 const char[6] 转换为 char *
```

如何修改，使之无语法错误？

```
const char * pc="hello"; // 方法 1
```

```
char * pc=(char *)"hello"; // 方法 2
```

虽然语法上，方法 1和2都正确，但方法2 存在潜在危险






2.3 const 修饰符

思考题:

```
char * pc=(char *)“hello”; // 方法 2
```

```
pc[0]='H'; // 在执行时出现问题
```



```
char* pc =(char *) "hello";  
pc[0] = 'H' ;  
return 0;
```

已引发异常

引发了异常: 写入访问权限冲突。
pc 是 0x529BEC。

原理:

“Hello”在只读数据存储区，其中的内容是不能修改的。pc 指向了一个只读数据存储单元，不能修改pc指向的单元。





2.3 const 修饰符

思考题:

```
char * pc=(char *)“hello”; // 有风险
```

// 直接通过 pc 修改只读数据区

```
const char *pc = “hello”; // 也有风险
```

// 通过 pc数据类型的转换， 修改只读数据区

```
pc[0]='H'; // 编译报错
```

```
*(char *)&(pc[0])='H'; // 运行报错
```

```
char pa[10]=“hello”; // 保险做法
```





2.3 const 修饰符

讨论: **const** 与 **#define** 相比, 有何优点?

- **const** 常量有数据类型, 编译器可对其进行类型安全检查
- 宏常量无数据类型, 没有类型安全检查
- 宏替换时, 可能出现预想不到的错误

```
#define doubled(x) x*2
```

```
doubled(1+2) = ?
```

- 在调试时, 可对**const** 常量进行调试





2.3 const 修饰符

讨论：为什么 `char a[] = "hello";` 正确

`char *p="hello";` 编译时错误？（VS2019）

❑ `const char *p = "hello";`

❑ `char *p = (char *)"hello";`

`p[0]='g';`

虽然此种形式在编译时未报错，但执行时异常，为什么？





2.4 宏

```
#define doubled(x) x*2
```

```
cout<<"doubled(3) = "<<doubled(3)<<endl;
```

```
cout<<"doubled(1+2) = "<<doubled(1+2)<<endl;
```

```
doubled(3) = 6
```

```
doubled(1+2) = ?      5
```





2.5 内联函数

```
inline int doubled(int x) {  
    return x*2;  
}
```

```
cout<<"doubled(3) = "<<doubled(3)<<endl;  
cout<<"doubled(1+2) = "<<doubled(1+2)<<endl;
```

doubled(3) = 6

doubled(1+2) = ? 6

希望生成的代码更优化一些;

有些编译器不理睬内联的请求。 VS2019 不理睬 inline





2.6 引用类型

引用变量的定义、初始化、使用

- 引用变量
- 引用参数
- 返回引用值

2.6 引用类型

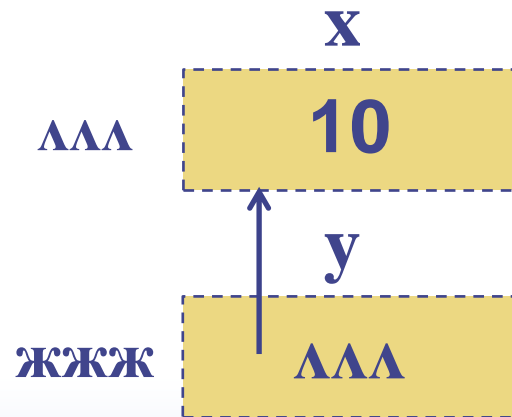
引用变量

```
int x=10, t;
```

```
int &y = x;    (1)
```

```
y=20;        (2)
```

```
t = y;
```



执行(1)后

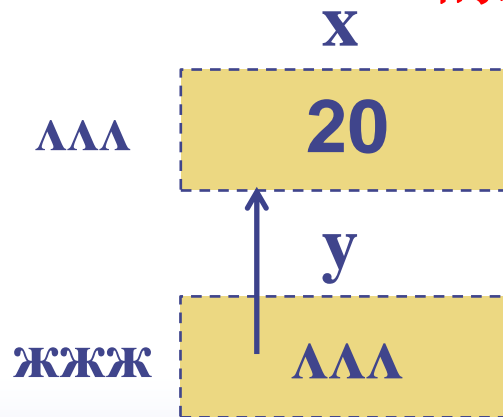
与指针的对比

```
int *y;
```

```
y = &x;
```

```
*y = 20;
```

```
t = *y;
```



执行(2)后

为什么定义引用变量时就一定要初始化?

定义时的 =
与使用时的 =
的差别?



2.6 引用类型

定义引用变量时就一定要正确初始化

```
int x=10;
```

```
int &y;    // 错误语句，必须初始化引用
```

```
int &y=20; // 错误语句，无法从“int”转换为“int &”
```

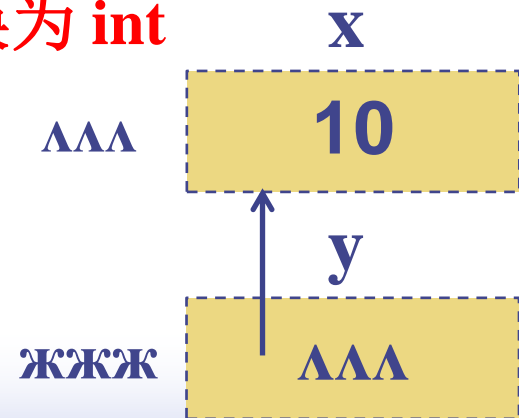
```
int &y=(int *)malloc(sizeof(int));
```

// 无法从“int*”转换为“int &”

```
y=&x;    // 错误语句，无法从 int* 转换为 int
```

```
int &y=x;
```

```
int &y=*(int *)malloc(sizeof(int));
```





2.6 引用类型

引用参数

交换两个整型变量的值的函数

```
void swap (int &x, int &y)
{
    int t;
    t=x;
    x=y;
    y=t;
}
```

```
int a, b;
swap(a,b);
```

看汇编代码：观察引用参数传递的是什么？





2.6 引用类型

引用参数

```
swap(int *x, int *y)
{
    int t=*x;
    *x=*y;
    *y=t;
}
```

```
int a=10;
int b=20;
swap(&a, &b);
```

```
swap(int &x, int &y)
{
    int t=x;
    x=y;
    y=t;
}
```

```
int a=10;
int b=20;
swap(a, b);
```





2.6 引用类型

引用参数

```
struct student
{
    char name[20];
    int age;
    int weight;
};
```

```
struct student xu;
print_info1(xu);
print_info2(xu);
```

```
void print_info1(struct student &s)
{
    cout<< s.name<<endl;
}
```

```
void print_info2(struct student s)
{
    cout<<s.name<<endl;
}
```

看汇编代码：观察引用参数传递的是什么？





2.6 引用类型

返回结果为引用

```
int & f( ) {  
    int t=25;  
    return t;  
}
```

```
lea    eax,[t] // 返回地址
```

```
int  a=f();  
call f  
mov  eax,dword ptr [eax]  
mov  dword ptr [ebp-0Ch],eax
```

```
int f( ) {  
    int t=25;  
    return t;  
}
```

```
mov    eax,dword ptr [t]  
// 返回值
```

```
int  a=f();  
call f  
mov  dword ptr [ebp-0Ch],eax
```

两者都显示 a=25

warning C4172: 返回局部变量或临时变量的地址



2.6 引用类型

返回结果为引用

```
int & f() {  
    int t=25;  
    return t;  
}
```

```
int a=f(); // 执行后 a=25;  
int &b = f(); // 执行后b 中的  
              内容为 t 的地址
```

类比:

```
int x=10;  
int &y = x; // y中内容是x的地址  
int u = y; // u中内容是x中的内容, 即 10  
int &v=y; // v中内容是y中的内容, 即 x的地址
```

与传统指针相比: `int a; int &y=a : int *p=&a;`
`int u = y : int u=*p; int &v=y : int &v=(*p)`



2.6 引用类型

返回结果为引用

```
int & f() {  
    int t=25;  
    return t;  
}
```

```
int g() {  
    int t=25;  
    return t;  
}
```

```
int x = f();
```

```
int y = g();
```

// x, y 的值都是25, 但实现的方法不同
// f 函数编译有警告

```
int &u= f();
```

```
int &v = g();
```

//无法从int 转换为 int &





2.6 引用类型

返回结果为引用

```
int & f(int t) {  
    t=t+10;  
    return t;  
}
```

```
a=f(10)+f(20);  
    // 显示 a = 60  
b = f(20)+f(10);  
    // 显示 b = 40
```

```
int f(int t) {  
    t=t+10;  
    return t;  
}
```

```
a=f(10)+f(20);  
    // 显示 a = 50  
b = f(20)+f(10);  
    // 显示 b = 50
```

warning C4172: 返回局部变量或临时变量的地址





2.6 引用类型

返回结果为引用

```
int & f(int t) {  
    t=t+10;  
    return t;  
}
```

```
a=f(10)+f(20);  
    // 显示 a = 60  
b = f(20)+f(10);  
    // 显示 b = 40
```

$a=f(10)+f(20)$

先执行 $f(10)$,
返回 函数 f 中变量 t 的地址

再执行 $f(20)$
返回 函数 f 中变量 t 的地址

根据第1个返回地址, 取相应单元的内容, 为 30

根据第2个返回地址, 取相应单元的内容, 为 30

故 $a=60$

warning C4172: 返回局部变量或临时变量的地址





2.6 引用类型

返回结果为引用

```
int & f(int t) {  
    t=t+10;  
    return t;  
}
```

```
    t = t + 10;  
mov     eax,dword ptr [t]  
add     eax,0Ah  
mov     dword ptr [t],eax  
    return t;  
lea     eax,[t]
```

```
a=f(10)+f(20) ;  
        // 显示 a = 60  
  
push    0Ah  
call    f (08D1190h)  
add     esp,4  
mov     esi,eax  
push    14h  
call    f (08D1190h)  
add     esp,4  
mov     ecx,dword ptr [esi]  
add     ecx,dword ptr [eax]  
mov     dword ptr [a],ecx
```





2.6 引用类型

返回结果为引用

```
int & f(int t) {  
    int *p=(int *)malloc(sizeof(int));  
    *p=t+10;  
    return *p;  
}
```

```
a=f(10)+f(20);  
    // 显示 a = 50  
b = f(20)+f(10);  
    // 显示 b = 50
```

删除引用& 后，结果同样正确。





2.6 引用类型

返回结果为引用

```
struct AAA {  
    int x;  
    int y;  
};  
AAA & f()  
{  
    AAA t;  
    t.x=40;  
    t.y=50;  
    return t;  
}
```

```
AAA zzz;  
zzz=f();  
cout <<"zzz.x = "<<zzz.x<<" zzz.y  
= "<<zzz.y<<endl;
```

warning C4172: 返回局部变量或临时变量的地址

运行显示: zzz.x=40 zzz.y=50





2.6 引用类型

```
char * fcharp()
{
    char t[20];
    strcpy_s(t, "hello");
    cout<<"t is "<<t<<endl;
    return t;
}
```

```
char *pc;
pc=fcharp(); // 此处观察 pc 指向的串为 hello
cout<<"pc is"<<pc<<endl;
```

warning C4172: 返回局部变量或临时变量的地址
运行显示 并非 hello





2.6 引用类型

讨论:

- 指针和引用有什么差别?
- 传引用比传指针安全, 为什么?
- 引用在创建时, 必须初始化, 即引用到一个有效的对象; 指针在定义时, 可以不初始化
- 指针可以为NULL
引用必须与合法的存储单元关联, 不存在NULL引用
- 引用一旦被初始化为指向一个对象, 就不能再改变为另一个对象的引用; 指针是可变的





2.7 函数参数及返回值

```
void DrawCircle(int x, int y, int r=10)
{
    .....
}
```

缺省参数

`DrawCircle(100, 100, 20);`

`DrawCircle(100, 100);`

如果某个参数给了缺省值，其右边的参数都需要给参数值。

思考：编译器会如何处理？





2.7 函数参数及返回值

```
int printf(const char *format, ...);
```

..... 省略参数

编写n个整数求和，n是未定的

```
long sum(int n, ...)
```

```
{
```

```
    long s=0;
```

```
    int *p=&n+1; //p指向第1个省略参数
```

```
    for (int k=0; k<n; k++) s+=p[k];
```

```
    return s;
```

```
}
```

```
void main( ) {
```

```
    long s=sum(3,4,5,6); //执行完后s=15
```

```
}
```

◆ 参数个数不定

◆ 常参数

在本函数内不变





2.7 函数参数及返回值

函数的返回值

`void f(...)`: 函数无返回值。

看汇编代码：观察函数返回的是什么？



2.8 变量的定义和声明

- ◆ 变量定义：定义性声明 **defining declaration (definition)**
 - ◆ 变量声明：引用性声明 **referencing declaration**, 简称申明
- 定义和申明的差别：是否为变量**分配存储空间**

- ◆ 函数定义须**有函数体**

- ◆ **C++ 规则**

- 变量声明和定义不必在语句之前。
- 所有变量都可以用任意表达式初始化。
- 函数内定义的变量如不初试化，其值不确定。
- 在if、while和for的条件部分定义变量，仅限于其语句访问。
- **由开工函数产生(初始化全局变量)，由收工函数消灭全局变量**





2.8 变量的定义和声明

```
#include <stdio.h>
extern int h;           //C变量定义方式: 常量表达式初始化
extern int i;           //变量声明, C和C++都可先声明后定义
int i=5;
int j=i+printf("ABC\n"); //C++定义方式: 任意表达式初始化, 输出ABC
static int p=j+5;       //C++定义方式: 任意表达式初始化
void main(void){
    static int n=j+5;    //C++定义方式: 任意表达式初始化
    int k, i=20;
    for(int j=k+2; j<9; j++){
        int m=5;
    };
    int q=23;            //C++定义方式, 定义出现在for语句后
    struct{ int k, m;}b={j+3, 5}; //C++定义方式
    int a[4]={scanf_s("%d", &k), 1}; //C++定义方式
} //即使main函数体为空也会输出ABC。程序作为对象(产生、活动和灭亡)。
```





2.8 变量的定义和声明

◆ **左值表达式**：在等号左边出现的值表达式 $++X$

◆ **右值表达式**：只能出现在等号右边的数值表达式。 $X++$

```
int x=10, y=20;
```

```
y = ++x;    等价于：x=x+1; y=x;
```

```
y = x++;    等价于：y= x;    x=x+1;
```

```
++x = y;    等价于：x=x+1; x=y;
```

```
x++ = y;    // 错， = 左操作数必须为 左值
```





2.8 变量的定义和声明

传统左值表达式:

能在等号左边出现的值表达式

➤ 非const类型变量

int x; x=2; // const定义的变量是传统右值

➤ 有址引用非const类型的变量

int &y=x; y=3;

➤ 非const指针和非const内容:

char z[]="abc",*p=z; p=(char*)"ab";*p='c';

➤ 有址引用非const类型的函数

int &f(){..... }; f()=3;

➤ 前置++和--运算、赋值运算

int C=1; (++C)=5; (C+=2)=3;





2.8 变量的定义和声明

传统右值表达式:

只能出现在等号右边的数值表达式

- **const**定义的变量是传统右值

const int x=3; 不可再给 **x**赋值

- **C**语言的函数调用为传统右值,

其调用只在等号右边出现;

- **C++**函数可返回传统左值, 可出现在等号左边

int &f(){.....} 。 F()=3;

- 一个表达式既然能出现在等号左边, 就必然能出现在等号右边; 反之则不一定成立。





2.8 变量的定义和声明

```
void f(int u, int v)
{
    cout << "u= " << u << " v = " << v << endl;
}
// 显示 u =3, v=4

int main()
{
    int x = 3, y = 4;
    f(x++, y++);
    cout << "x= " << x << " y = " << y << endl;
    return 0;
}
// 显示 x=4 y =5
```



2.8 变量的定义和声明

f(x++, y++);

```
mov     eax,dword ptr [y]
mov     dword ptr [ebp-0DCh],eax    // y的值拷贝到一个临时空间
mov     ecx,dword ptr [y]          // 实现 y = y+1
add     ecx,1
mov     dword ptr [y],ecx
mov     edx,dword ptr [x]
mov     dword ptr [ebp-0E0h],edx    // x的值拷贝到一个临时空间
mov     eax,dword ptr [x]          // 实现 x = x+1
add     eax,1
mov     dword ptr [x],eax
mov     ecx,dword ptr [ebp-0DCh]    // 从临时空间取数作为参数
push    ecx
mov     edx,dword ptr [ebp-0E0h]    // 从临时空间取数作为参数
push    edx
call    f (0C910E6h)
```





2.8 变量的定义和声明

左值引用

```
void fr(int &u, int &v)
{
    cout << "u= " << u << "  v = " << v << endl;
    u = 20;    v = 30;
}              // 显示 u =3,    v=4

int main()
{
    int  x = 3, y = 4;
    fr(x, y);
    cout << "x= " << x << "  y = " << y << endl;
    return 0;      // 显示 x=20  y =30
}
```





2.8 变量的定义和声明

左值引用

```
void fr(int &u, int &v)
{
    cout << "u= " << u << "  v = " << v << endl;
    u = 20;    v = 30;
}              // 显示 u =4,    v=5

int main()
{
    int  x = 3, y = 4;
    fr(++x, ++y);
    cout << "x= " << x << "  y = " << y << endl;
    return 0;      // 显示 x=20  y =30
}
```





2.8 变量的定义和声明

左值引用

```
void fr(int &u, int &v)
{
    cout << "u= " << u << "  v = " << v << endl;
    u = 20;    v = 30;
}              // 显示 u =4,    v=5

int main()
{
    int  x = 3, y = 4;
    fr(x++, y++); // 语法错误, 无法从int转换为int &
    cout << "x= " << x << "  y = " << y << endl;
    return 0;
}          // 想要传递一个临时对象的地址
```





2.8 变量的定义和声明

右值引用

```
void frr(int &&u, int &&v)
{
    cout << "u= " << u << "  v = " << v << endl;
    u = 20;    v = 30;
}              // 显示 u =3,    v=4

int main()
{
    int  x = 3, y = 4;
    frr(x++, y++);
    cout << "x= " << x << "  y = " << y << endl;
    return 0;
}              // 显示 x =4,    y=5
```





2.8 变量的定义和声明

`frr(x++, y++);`

右值引用

```
mov     eax, dword ptr [y]
mov     dword ptr [ebp-0E0h], eax
mov     ecx, dword ptr [y]
add     ecx, 1
mov     dword ptr [y], ecx
mov     edx, dword ptr [x]
mov     dword ptr [ebp-0ECh], edx
```

.....

```
lea     ecx, [ebp-0E0h]    // 传递临时对象的地址
push    ecx
lea     edx, [ebp-0ECh]
push    edx
call    frr (0CE1168h)
```





2.8 变量的定义和声明

`void *p` 所指向的实体单元字节数不定。

- 可以将任意类型变量的地址赋给 `p`。
- 对指向的实体单元赋值时，类型或字节数必须确定，必须进行强制类型转换

例如： `*(double *)p = 3.2;`



2.8 变量的定义和声明

```
int i=1;
```

```
const int &j=5; //只读引用变量引用常量
```

```
// 5是一个常量，有一个单元中存放的数为5
```

```
// 变量j中存放的是存放5的那个单元的地址，亦称 匿名地址
```

```
// int &j=5; error, 无法从int转换为int &
```

```
// int &j=i+5; error, 无法从int转换为int &
```

```
const int k=10, &m=k; //只读引用变量引用只读变量(右值表达式)
```

```
int &x=++i; //左值引用，++i为左值表达式，x指向i; i=2
```

```
int &y=i=0; //左值引用，i=0为左值表达式，被引用变量为i
```

```
// 先执行 i=0; 后执行 &y=i; y引用i
```

```
int &z=y=3; //左值引用，y=3为左值表达式，被引用变量为i
```

```
// 此时 i=3; y=3 ;(即y指向的单元为i为3, *y=3)
```

```
// z也是指向i的
```

执行后 i,x,y,z 皆为 3; k, m 为 10.





2.8 变量的定义和声明

```
int &f(int &v){  
    return v+=5;    //v+=5为左值，返回v引用的变量  
}  
  
void main(void)  
{  
    int &w=i++; //错误，i++为右值，无法从int转换为int &  
    i=7; //x、y、z共享i的内存，故x=y=z=i=7  
    int w=8; //x=y=z=i=7，w=8共享匿名左值变量的内存  
    ++y=10; //左值++y使i=8并引用i，10赋给i使x=y=z=i=10  
    (z=9)=15; //左值z=9使i=9并引用i，15赋给i使x=y=z=i=15  
    (f(y)=1)=2; //f(r)返回y引用的i，  
                // (f(y)=1)为左值引用i，x=y=z=i=2  
}
```





2.8 变量的定义和声明

位段 Bit field

在结构体或者联合体中以位为单位定义成员变量所占的空间

```
struct A{  
    int i:3;           //i为位段成员  
    int j:4;           //j为位段成员  
    int k;  
} a;                  // sizeof(a) =8;  
a.i = 6;  
a.j = 9;              // 一个字节中存放的信息
```





2.8 变量的定义和声明

```
struct A{  
    int j:4;           //j为位段成员  
    int k;  
}  
a;  
void main(void){  
    register int i=0, &j=i; //正确, i、j都编译为(基于栈的)自动  
    变量  
    int &&m=j;           //无法从 int 转换为 int &&  
    int &*m;              //错, 不能指向逻辑上不分配内存的引用  
    int &s[4];            //错误, 数组元素不能为引用变量  
    int t[6], (&u)[6]=t; //正确, 引用变量u可以引用数组t  
    int &v=t[0];         //正确, 引用变量v可以引用数组元素  
    int &w=a.j;          //错误, 位段不能被引用  
    int &x=a.k;          //正确, a.k不是位段  
}
```





2.8 变量的定义和声明

静态变量

```
void f ( ....) {  
  
    static int x=0;  
    x++;  
    cout <<x<<endl;  
}
```

第1次调用函数 f 时, 显示 1
第2次调用函数 f 时, 显示 2
第3次调用函数 f 时, 显示 3

变量

```
void f ( ....) {  
  
    int x=0;  
    x++;  
    cout <<x<<endl;  
}
```

显示 1
显示 1
显示 1



2.8 变量的定义和声明

静态变量

```
void f ( ....) {
```

```
    static int x=0;  
    x++;  
    cout <<x<<endl;  
}
```

变量

```
void f ( ....) {
```

```
    int x=0;  
    x++;  
    cout <<x<<endl;  
}
```

- X 分配的空间静态存储器，属于静态存储方式
- X 不是全局变量，它的作用域只在函数内
- X 的生命周期是整个程序，在函数返回时并不消亡
- 全局变量也是静态存储，它的作用域是整个程序
- 静态局部变量在定义时，若未赋初值，会赋予 0





2.8 变量的定义和声明

静态局部变量 VS 局部变量 作用域

静态全局变量 VS 全局变量 生命周期

- 静态全局变量的作用域 是定义该变量的文件
- 其他文件中可以有同名的变量
- 全局变量的作用域是整个程序



2.9 命名空间

定义： namespace 名称 { }

namespace mine

```
{ // 变量，函数，类， .....  
    char username[20]="Xu Xiangyang";  
}
```

namespace yours

```
{  
    char username[20]="zhang";  
    char *p= "hello"; //无法从const char[6] 转换为 char *  
    const char *p= "hello";  
}
```





2.9 命名空间

使用: `using namespace 名称;`
`名称 :: 成员`

`using namespace yours;`

```
void display()
{
    cout<<username<<endl;
    cout<<"Hello "<<yours::username<<endl;
    cout<<"My name is "<<mine::username<<endl;
}
```

标准C++ 将整个库定义在 命名空间 `std` 下。
`using namespace std;`





总结

常

const 修饰符

const int x =; 常变量

char * const p =; 指针常量

const char *p; 常量指针（是指针，指向常量）

const char * const p =; 指向常量的指针常量

Myfun(const int a); 常参数

Myfun(int a) const; 常成员函数 / 类 / 第3章

const Date National_day(1, 10, 1949); 常对象 / 类

常变量：在定义时要赋值，
在程序中不能直接或者间接“修改”



总结

引用

引用

`int &x =`; 引用变量

`void f(int &x);` 引用参数

`int &f(.....);` 返回结果引用 类比 `int &x=...;`

`y=f(...)` `y=x;`

引用变量:

- 相当于指针，变量中存放被引用对象的地址；
- 在定义时赋初值
- 在使用引用变量时等价于指针；`*x`
这也就要求必须在定义时赋初值



总结

静态

static 修饰符

静态全局变量	作用域：文件内	生命周期：整个程序
静态局部变量	作用域：函数内	生命周期：整个程序

静态函数 作用域：文件内 生命周期：整个程序
在其他文件中可以有同名函数

静态数据成员； 类 /第5 章
静态成员函数； 类 /第5 章





总结

函数

函数缺省参数

```
int myfun(int a, int b=5);    // 申明  
x=mufun(10);                 // 调用
```

函数常参数

```
int myfun(int a, const int b);    // 申明  
x=mufun(10, 20);                // 调用  
char * strcpy(char *dst, const char *src);
```

函数引用参数

```
void myfun(int &a, int &b);    // 申明  
Int x,y;  mufun(x, y);        // 调用
```





总结

const 修饰符

static

宏

内联函数

指针

引用

输入输出的流式运算

命名空间

