

类成员的可访问范围

郭 炜 刘家瑛

北京大学



类成员的可访问范围

- 关键字 -- 类成员可被访问的范围
 - private:** 指定私有成员, 只能在成员函数内被访问
 - public :** 指定公有成员, 可以在任何地方被访问
 - protected:** 指定保护成员
- 三种关键字出现的次数和先后次序都没有限制



对象成员的访问权限

定义一个类

```
class className {
```

private:

私有属性和函数

public:

公有属性和函数

protected:

保护属性和函数

```
};
```

说明类成员
的可见性



对象成员的访问权限

- 缺省为私有成员

```
class Man {  
    int nAge;           //私有成员  
    char szName[20]; // 私有成员  
  
public:  
    void SetName(char * Name){  
        strcpy(szName, Name);  
    }  
};
```



对象成员的访问权限

▀ 类的成员函数内部, 可以访问:

- 当前对象的全部属性, 函数
- 同类其它对象的全部属性, 函数

▀ 类的成员函数以外的地方,

- 只能够访问该类对象的公有成员



```
class CEmployee {
    private:
        char szName[30]; //名字
    public :
        int salary;      //工资
        void setName(char * name);
        void getName(char * name);
        void averageSalary(CEmployee e1,CEmployee e2);
};

void CEmployee::setName( char * name) {
    strcpy( szName, name); //ok
}

void CEmployee::getName( char * name) {
    strcpy( name, szName); //ok
}
```



```
void CEmployee::averageSalary(CEmployee e1,CEmployee e2){  
    salary = (e1.salary + e2.salary )/2;  
}  
  
int main(){  
    CEmployee e;  
    strcpy(e.szName,"Tom1234567889"); //编译错, 不能访问私有成员  
    e.setName( "Tom"); // ok  
    e.salary = 5000;    //ok  
    return 0;  
}
```

- 设置私有成员的目的
 - 强制对成员变量的访问一定要通过成员函数进行
- 设置私有成员的机制 -- 隐藏



szName → char szName[5]

- 如果szName不是私有, 需要修改全部:

```
strcpy(man1.szName, "Tom1234567889");
```

- 如果将szName变为私有,

所有对 szName的访问都是通过成员函数来进行,

```
man1.setName( "Tom12345678909887");
```




程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>



北京大学
PEKING UNIVERSITY

信息科学技术学院《程序设计实习》 郭炜 刘家瑛

函数指针

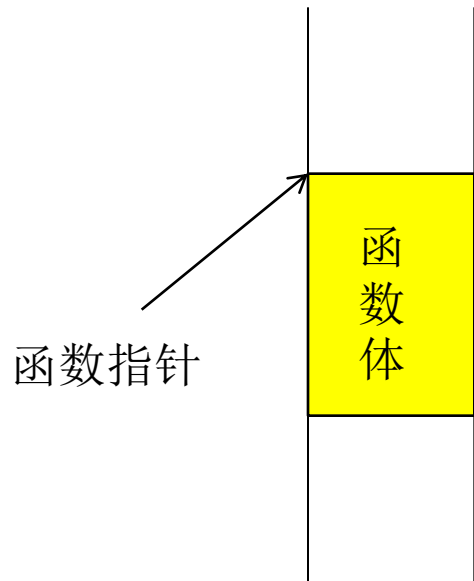
(教材P107)

基本概念

程序运行期间，每个函数都会占用一段连续的内存空间。而函数名就是该函数所占内存区域的起始地址(也称“入口地址”)。我们可以将函数的入口地址赋给一个指针变量，使该指针变量指向该函数。然后通过指针变量就可以调用这个函数。这种指向函数的指针变量称为“**函数指针**”。

基本概念

程序运行期间，每个函数都会占用一段连续的内存空间。而函数名就是该函数所占内存区域的起始地址(也称“入口地址”)。我们可以将函数的入口地址赋给一个指针变量，使该指针变量指向该函数。然后通过指针变量就可以调用这个函数。这种指向函数的指针变量称为“**函数指针**”。



定义形式

类型名 (* 指针变量名)(参数类型1, 参数类型2,...);

定义形式

类型名 (* 指针变量名)(参数类型1, 参数类型2,...);

例如:

```
int (*pf)(int ,char);
```

定义形式

类型名 (* 指针变量名)(参数类型1, 参数类型2,...);

例如:

int (*pf)(int ,char);

表示pf是一个函数指针，它所指向的函数，返回值类型应是int，该函数应有两个参数，第一个是int 类型，第二个是char类型。

使用方法

可以用一个原型匹配的函数的名字给一个函数指针赋值。

要通过函数指针调用它所指向的函数，写法为：

函数指针名(实参表);

使用方法

```
#include <stdio.h>

void PrintMin(int a,int b) {
    if( a<b )
        printf("%d",a);
    else
        printf("%d",b);
}

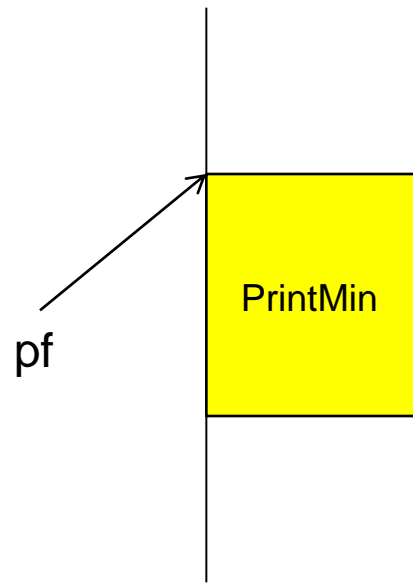
int main() {
    void (* pf)(int ,int);
    int x = 4, y = 5;
    pf = PrintMin;
    pf(x,y);
    return 0;
}
```

使用方法

```
#include <stdio.h>

void PrintMin(int a,int b) {
    if( a<b )
        printf("%d",a);
    else
        printf("%d",b);
}

int main() {
    void (* pf)(int ,int);
    int x = 4, y = 5;
    pf = PrintMin;
    pf(x,y);
    return 0;
}
```

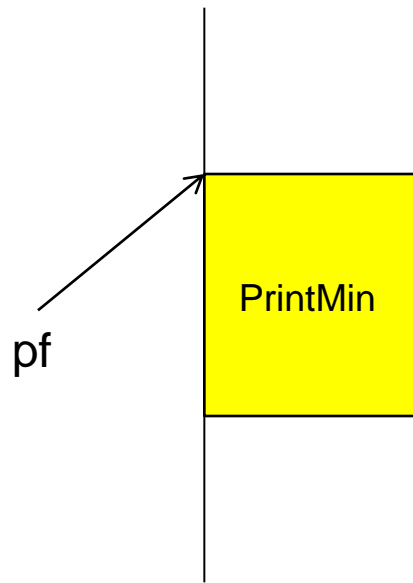


使用方法

```
#include <stdio.h>

void PrintMin(int a,int b) {
    if( a<b )
        printf("%d",a);
    else
        printf("%d",b);
}

int main() {
    void (* pf)(int ,int);
    int x = 4, y = 5;
    pf = PrintMin;
    pf(x,y);
    return 0;
}
```



输出结果：
4

函数指针和qsort库函数

C语言快速排序库函数：

```
void qsort(void *base, int nelem, unsigned int width,  
    int ( * pfCompare)( const void *, const void *));
```

可以对任意类型的数组进行排序

函数指针和qsort库函数

a[0]	a[1]	a[i]	a[n-1]
------	------	-------	------	-------	--------

对数组排序，需要知道：

函数指针和qsort库函数

a[0]	a[1]	a[i]	a[n-1]
------	------	-------	------	-------	--------

对数组排序，要知道：

1) 数组起始地址

函数指针和qsort库函数

a[0]	a[1]	a[i]	a[n-1]
------	------	-------	------	-------	--------

对数组排序，需要知道：

- 1) 数组起始地址
- 2) 数组元素的个数

函数指针和qsort库函数

a[0]	a[1]	a[i]	a[n-1]
------	------	-------	------	-------	--------

对数组排序，需要知道：

- 1) 数组起始地址
- 2) 数组元素的个数
- 3) 每个元素的大小（由此可以算出每个元素的地址）

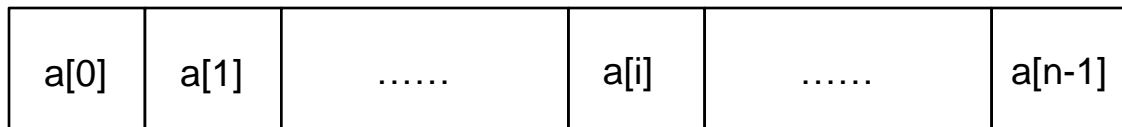
函数指针和qsort库函数

a[0]	a[1]	a[i]	a[n-1]
------	------	-------	------	-------	--------

对数组排序，需要知道：

- 1) 数组起始地址
- 2) 数组元素的个数
- 3) 每个元素的大小（由此可以算出每个元素的地址）
- 4) 元素谁在前谁在后的规则

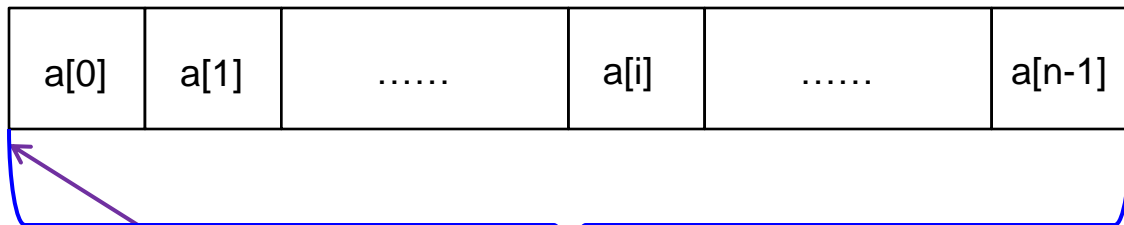
函数指针和qsort库函数



```
void qsort(void *base, int nelem, unsigned int width,  
int ( * pfCompare)( const void *, const void *));
```

base: 待排序数组的起始地址,

函数指针和qsort库函数

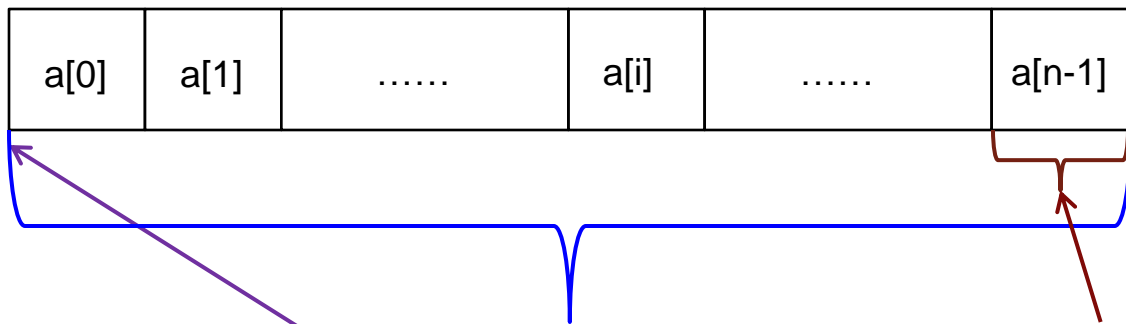


```
void qsort(void *base, int nelem, unsigned int width,  
int ( * pfCompare)( const void *, const void *));
```

base: 待排序数组的起始地址,

nelem: 待排序数组的元素个数,

函数指针和qsort库函数



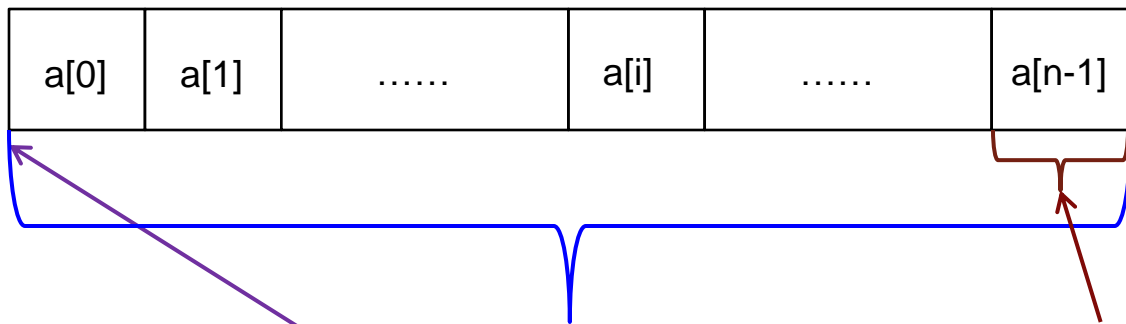
```
void qsort(void *base, int nelem, unsigned int width,  
int ( * pfCompare)( const void *, const void *));
```

base: 待排序数组的起始地址,

nelem: 待排序数组的元素个数,

width: 待排序数组的每个元素的大小 (以字节为单位)

函数指针和qsort库函数



```
void qsort(void *base, int nelem, unsigned int width,  
int (* pfCompare)( const void *, const void *));
```

base: 待排序数组的起始地址,

nelem: 待排序数组的元素个数,

width: 待排序数组的每个元素的大小 (以字节为单位)

pfCompare: 比较函数的地址

函数指针和qsort库函数

```
void qsort(void *base, int nelem, unsigned int width,  
    int ( * pfCompare)( const void *, const void *));
```

pfCompare: 函数指针，它指向一个“比较函数”。
该比较函数应为以下形式：

```
int 函数名(const void * elem1, const void * elem2);
```

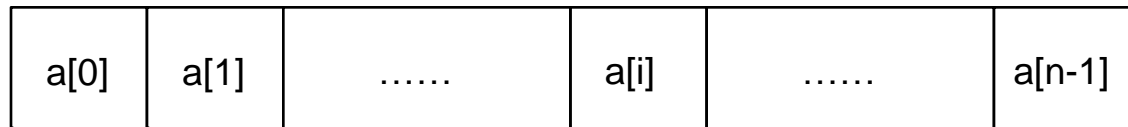
比较函数是程序员自己编写的

函数指针和qsort库函数


排序就是一个不断比较并交换位置的过程。

qsort函数在执行期间，会通过pfCompare指针调用“比较函数”，调用时将要比较的两个元素的地址传给“比较函数”，然后根据“比较函数”返回值判断两个元素哪个更应该排在前面。

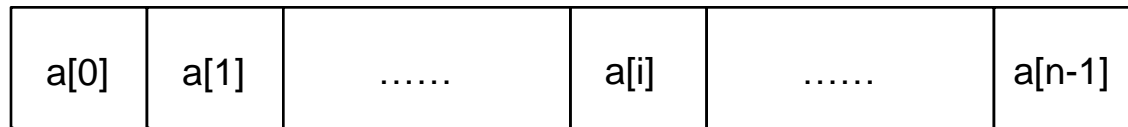
函数指针和qsort库函数



pfCompare(e1, e2);



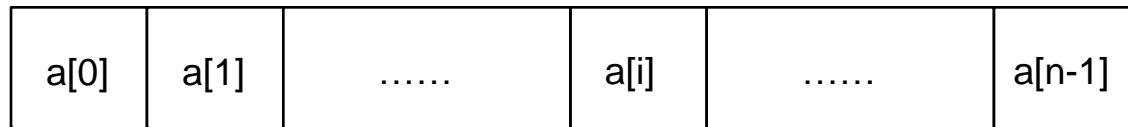
函数指针和qsort库函数



pfCompare(e1, e2);

int 比较函数名(const void * elem1, const void * elem2);

函数指针和qsort库函数



pfCompare(e1, e2);

int 比较函数名(const void * elem1, const void * elem2);

比较函数编写规则:

- 1) 如果 * elem1应该排在 * elem2前面, 则函数返回值是负整数
- 2) 如果 * elem1和* elem2哪个排在前面都行, 那么函数返回0
- 3) 如果 * elem1应该排在 * elem2后面, 则函数返回值是正整数

函数指针和qsort库函数

实例：

下面的程序，功能是调用qsort库函数，将一个unsigned int数组按照个数从小到大进行排序。比如 8，23，15三个数，按个数从小到大排序，就应该是 23，15，8

```
#include <stdio.h>
#include <stdlib.h>
int MyCompare( const void * elem1, const void * elem2 )
{
    unsigned int * p1, * p2;
    p1 = (unsigned int *) elem1; // “* elem1” 非法
    p2 = (unsigned int *) elem2; // “* elem2” 非法
    return (* p1 % 10) - (* p2 % 10 );
}
#define NUM 5
int main()
{
    unsigned int an[NUM] = { 8,123,11,10,4 };
    qsort( an,NUM,sizeof(unsigned int), MyCompare);
    for( int i = 0;i < NUM; i ++ )
        printf("%d ",an[i]);
    return 0;
}
```

输出结果:
10 11 123 4 8



程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>

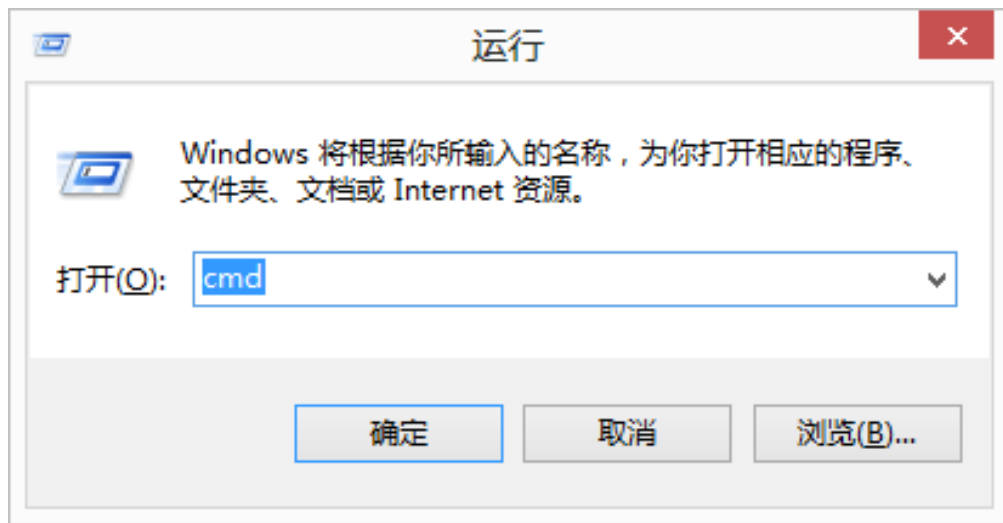


命令行参数

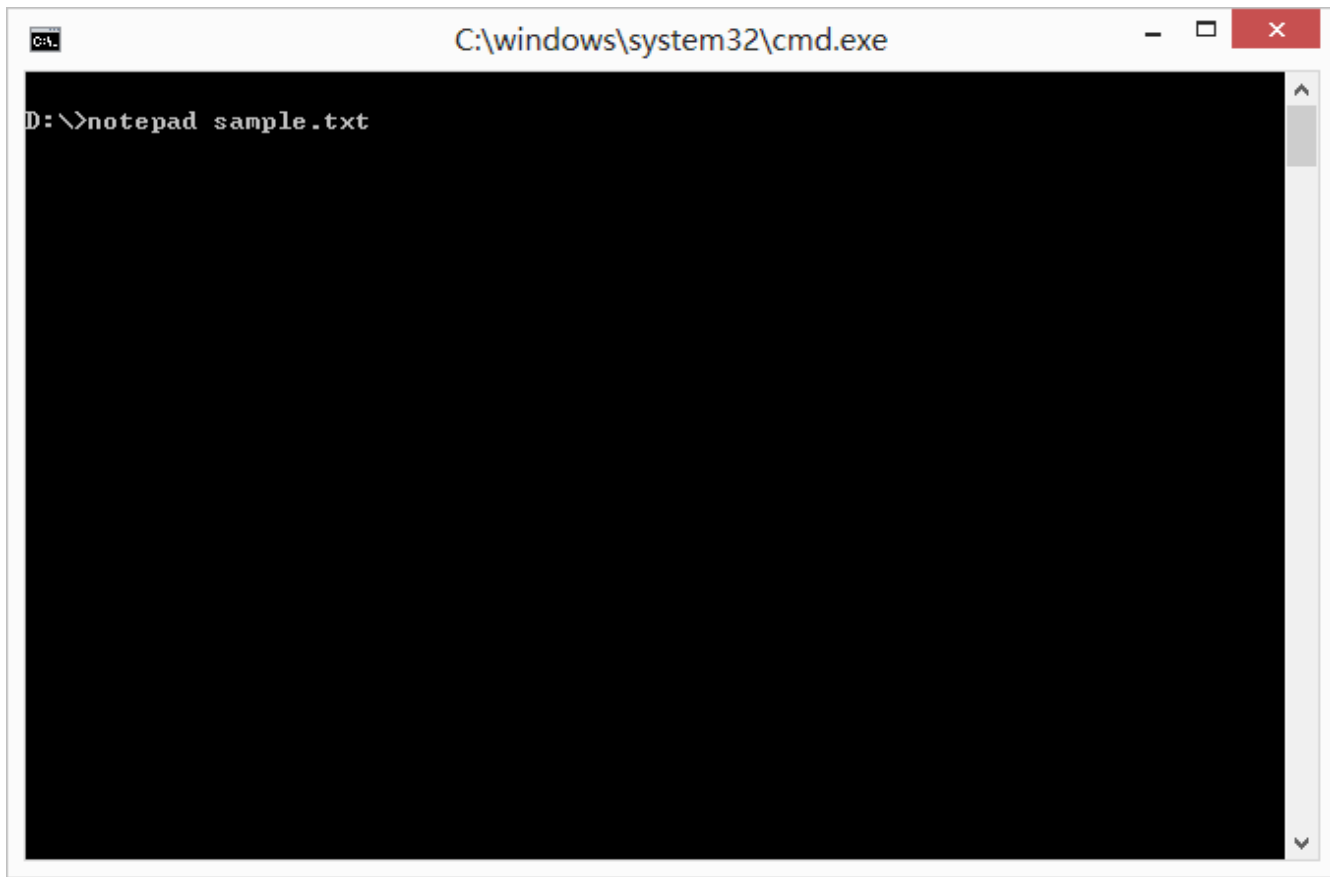
(教材P157)

命令行方式运行程序

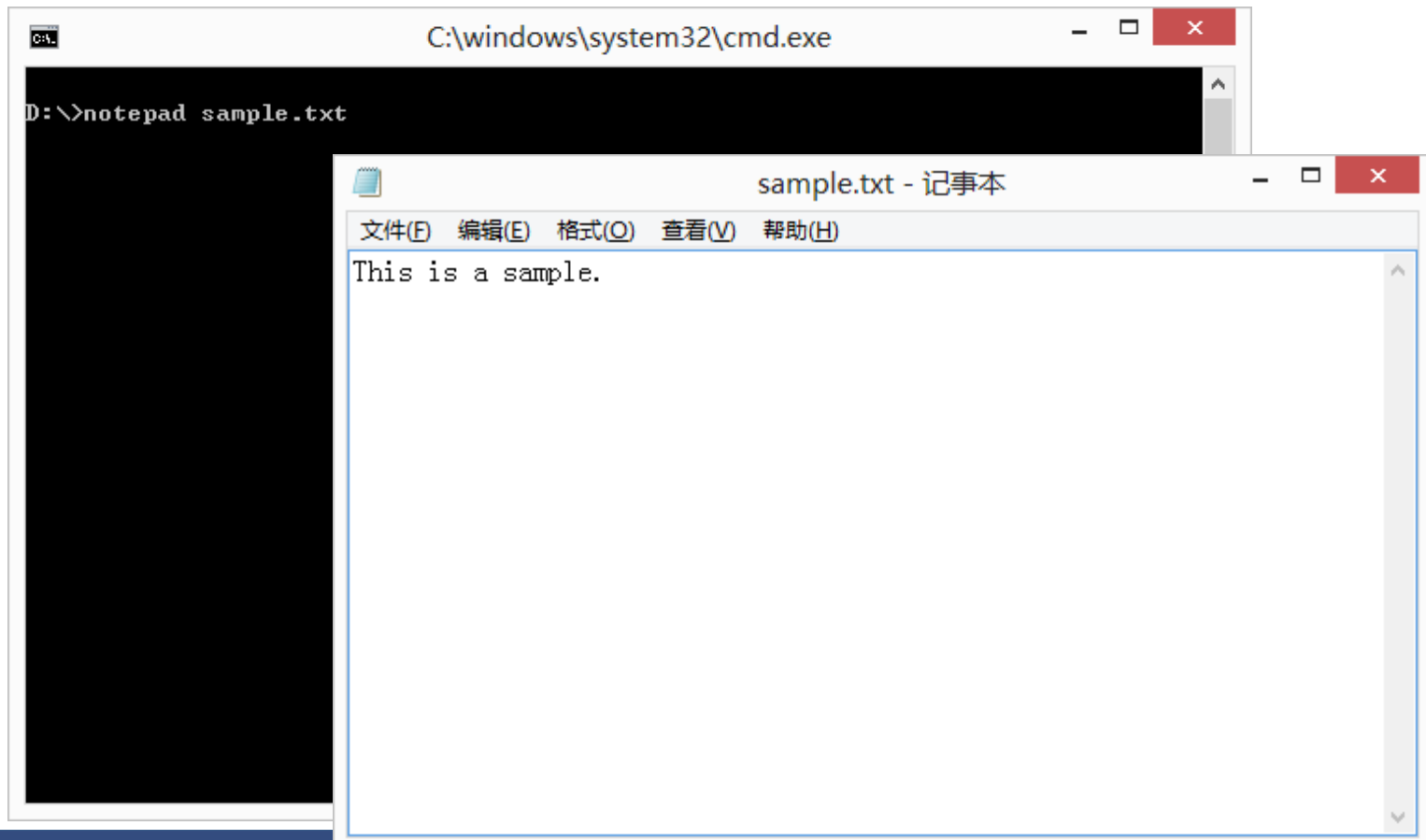
Windows + R 键:



命令行方式运行程序



命令行方式运行程序



命令行方式运行程序

```
notepad sample.txt
```

notepad 程序如何得知，用户在以命令行方式运行它的时候，后面跟着什么参数？

命令行参数

将用户在CMD窗口输入可执行文件名的方式启动程序时，跟在可执行文件名后面的那些字符串，称为“**命令行参数**”。命令行参数可以有多个，以空格分隔。比如，在CMD窗口敲：

```
copy file1.txt file2.txt
```

“copy”，“file1.txt”，“file2.txt”
就是命令行参数

如何在程序中获得命令行参数呢？

命令行参数

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

argc: 代表启动程序时，命令行参数的个数。C/C++语言规定，可执行程序程序本身的文件名，也算一个命令行参数，因此，argc的值至少是1。

命令行参数

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

argc: 代表启动程序时，命令行参数的个数。C/C++语言规定，可执行程序程序本身的文件名，也算一个命令行参数，因此，argc的值至少是1。

argv: 指针数组，其中的每个元素都是一个char* 类型的指针，该指针指向一个字符串，这个字符串里就存放着命令行参数。

例如，argv[0]指向的字符串就是第一个命令行参数，即可执行程序的文件名，argv[1]指向第二个命令行参数，argv[2]指向第三个命令行参数……。

```
#include <stdio.h>
int main(int argc, char * argv[])
{
    for(int i = 0; i < argc; i ++ )
        printf( "%s\n", argv[i]);
    return 0;
}
```

将上面的程序编译成sample.exe，然后在控制台窗口敲：

```
sample para1 para2 s.txt 5 "hello world"
```

```
#include <stdio.h>
int main(int argc, char * argv[])
{
    for(int i = 0; i < argc; i ++ )
        printf( "%s\n", argv[i]);
    return 0;
}
```

将上面的程序编译成sample.exe，然后在控制台窗口敲：

sample para1 para2 s.txt 5 “hello world”

输出结果就是：

sample
para1
para2
s.txt
5
hello world



程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>



位运算

(教材P28)

基本概念

位运算：

用于对整数类型（int, char, long 等）变量中的**某一位**(bit)，或者**若干位**进行操作。比如：

基本概念

位运算：

用于对整数类型（int, char, long 等）变量中的**某一位**(bit)，或者**若干位**进行操作。比如：

1) 判断某一位是否为1

基本概念

位运算：

用于对整数类型（int, char, long 等）变量中的**某一位**(bit)，或者**若干位**进行操作。比如：

- 1) 判断某一位是否为1
- 2) 只改变其中某一位，而保持其他位都不变。

基本概念

位运算:

用于对整数类型 (int, char, long 等) 变量中的
某一位 (bit), 或者若干位进行操作。比如:

- 1) 判断某一位是否为1
- 2) 只改变其中某一位, 而保持其他位都不变。

C/C++语言提供了六种位运算符来进行位运算操作:

&	按位与 (双目)
	按位或 (双目)
^	按位异或 (双目)
~	按位非 (取反) (单目)
<<	左移 (双目)
>>	右移 (双目)

按位与“&”

将参与运算的两操作数各对应的二进制位进行与操作，只有对应的两个二进制位均为1时，结果的对应二进制位才为1，否则为0。

按位与 “&”

例如：表达式 “21 & 18 ” 的计算结果是16
(即二进制数10000)， 因为：

按位与 “&”

例如：表达式 “21 & 18 ” 的计算结果是16
(即二进制数10000)， 因为：

21 用二进制表示就是：

0000 0000 0000 0000 0000 0000 0001 0101

18 用二进制表示就是：

0000 0000 0000 0000 0000 0000 0001 0010

二者按位与所得结果是：

0000 0000 0000 0000 0000 0000 0001 0000

按位与“&”

通常用来将某变量中的某些位清0且同时保留其他位不变。
也可以用来获取某变量中的某一位。

例如，如果需要将int型变量n的低8位全置成0，而其余位不变，则可以执行：

按位与“&”

通常用来将某变量中的某些位清0且同时保留其他位不变。
也可以用来获取某变量中的某一位。

例如，如果需要将int型变量n的低8位全置成0，而其余位不变，则可以执行：

```
n = n & 0xffffffff00;
```

按位与 “&”

通常用来将某变量中的某些位清0且同时保留其他位不变。
也可以用来获取某变量中的某一位。

例如，如果需要将int型变量n的低8位全置成0，而其余位不变，则可以执行：

```
n = n & 0xffffffff00;
```

也可以写成：

```
n &= 0xffffffff00;
```

按位与“&”

通常用来将某变量中的某些位清0且同时保留其他位不变。
也可以用来获取某变量中的某一位。

例如，如果需要将int型变量n的低8位全置成0，而其余位不变，则可以执行：

```
n = n & 0xffffffff00;
```

也可以写成：

```
n &= 0xffffffff00;
```

如果n是short类型的，则只需执行：

```
n &= 0xff00;
```

按位与 “&”

如何判断一个int型变量n的第7位（从右往左，从0开始数）是否是1 ？

按位与 “&”

如何判断一个int型变量n的第7位（从右往左，从0开始数）是否是1？

只需看表达式 “`n & 0x80`” 的值是否等于0x80即可。

0x80: 1000 0000

按位或 “|”

将参与运算的两操作数各对应的二进制位进行或操作，只有对应的两个二进制位都为0时，结果的对应二进制位才是0，否则为1。

按位或 “|”

例如：表达式 “21 | 18 ” 的值是23，因为：

按位或 “|”

例如：表达式 “21 | 18 ” 的值是23， 因为：

21:	0000	0000	0000	0000	0000	0000	0001	0101
18:	0000	0000	0000	0000	0000	0000	0001	0010
21 18:	0000	0000	0000	0000	0000	0000	0001	0111

按位或 “|”

按位或运算通常用来将某变量中的某些位置1且保留其他位不变。

例如，如果需要将int型变量n的低8位全置成1，而其余位不变，则可以执行：

按位或 “|”

按位或运算通常用来将某变量中的某些位置1且保留其他位不变。

例如，如果需要将int型变量n的低8位全置成1，而其余位不变，则可以执行：

```
n |= 0xff;
```

0xff: 1111 1111

按位异或 “^”

将参与运算的两操作数各对应的二进制位进行异或操作，即只有对应的两个二进制位不相同，结果的对应二进制位才是1，否则为0。

例如：表达式 “ $21 \wedge 18$ ” 的值是7(即二进制数111)。

21: 0000 0000 0000 0000 0000 0000 0001 0101

18: 0000 0000 0000 0000 0000 0000 0001 0010

$21 \wedge 18$: 0000 0000 0000 0000 0000 0000 0000 0111

按位异或 “^”

按位异或运算通常用来将某变量中的某些位取反，且保留其他位不变。

例如，如果需要将int型变量n的低8位取反，而其余位不变，则可以执行：

```
n ^= 0xff;
```

0xff: 1111 1111

按位异或 “^”

异或运算的特点是：

如果 $a \oplus b = c$ ，那么就有 $c \oplus b = a$ 以及 $c \oplus a = b$ 。（穷举法可证）

此规律可以用来进行最简单的加密和解密。

按位异或 “^”

另外异或运算还能实现不通过临时变量，就能交换两个变量的值：

```
int a = 5, b = 7;  
a = a ^ b;  
b = b ^ a;  
a = a ^ b;
```

即实现a,b值交换。穷举法可证。

按位非 “~”

按位非运算符 “~” 是单目运算符。

其功能是将操作数中的二进制位0变成1，1变成0。

例如，表达式 “~21” 的值是整型数 0xffffffffea

21:	0000	0000	0000	0000	0000	0000	0001	0101
~21:	1111	1111	1111	1111	1111	1111	1110	1010

左移运算符“<<”

表达式：

$a \ll b$

的值是：将a各二进制位全部左移b位后得到的值。左移时，高位丢弃，低位补0。a 的值不因运算而改变。

左移运算符“<<”

例如：

9 << 4

9的二进制形式：

0000 0000 0000 0000 0000 0000 0000 1001

因此，表达式“9<<4”的值，就是将上面的二进制数左移4位，得：

0000 0000 0000 0000 0000 0000 1001 0000

即为十进制的144。

左移运算符 “<<”

实际上，左移1位，就等于是乘以2，左移n位，就等于是乘以 2^n 。而左移操作比乘法操作快得多。

右移运算符“>>”

表达式：

$a \gg b$

的值是：将a各二进制位全部右移b位后得到的值。右移时，移出最右边的位就被丢弃。 a 的值不因运算而改变。

右移运算符“>>”

表达式：

`a >> b`

的值是：将a各二进制位全部右移b位后得到的值。右移时，移出最右边的位就被丢弃。 a 的值不因运算而改变。

对于有符号数，如long, int, short, char类型变量，在右移时，符号位（即最高位）将一起移动，并且大多数C/C++编译器规定，如果原符号位为1，则右移时高位就补充1，原符号位为0，则右移时高位就补充0。

右移运算符 “>>”

实际上，右移 n 位，就相当于左操作数除以 2^n ，并且将结果往小里取整。

$$-25 \gg 4 = -2$$

$$-2 \gg 4 = -1$$

$$18 \gg 4 = 1$$

```
#include <stdio.h>
int main()
{
    int n1 = 15;
    short n2 = -15;
    unsigned short n3 = 0xffe0;
    char c = 15;
    n1 = n1>>2;
    n2 >>= 3;
    n3 >>= 4;
    c >>= 3;
    printf( "n1=%d,n2=%x,n3=%x,c=%x",n1,n2,n3,c);
}
```

```
#include <stdio.h>
int main()
{
    int n1 = 15;
    short n2 = -15;
    unsigned short n3 = 0xffe0;
    char c = 15;
    n1 = n1>>2;
    n2 >>= 3;
    n3 >>= 4;
    c >>= 3;
    printf( "n1=%d,n2=%x,n3=%x,c=%x",n1,n2,n3,c);
} //输出结果是: n1=3,n2=fffffffe,n3=ffe,c=1
```


n1: 0000 0000 0000 0000 0000 0000 0000 1111

n1 >>= 2: 变成3

0000 0000 0000 0000 0000 0000 0000 0011

n2: 1111 1111 1111 0001

n2 >>= 3: 变成 ffffffff, 即-1

1111 1111 1111 1110

n3: 1111 1111 1110 0000

n3 >>= 4: 变成 ffe

0000 1111 1111 1110

c: 0000 1111

c >>= 3; 变成1

0000 0001

int n1 = 15;

short n2 = -15;

unsigned short n3 = 0xffe0;

char c = 15;

思考题：

有两个int型的变量a和n($0 \leq n \leq 31$),
要求写一个表达式, 使该表达式的值和a的第n位相同。

思考题：

有两个int型的变量a和n($0 \leq n \leq 31$),
要求写一个表达式, 使该表达式的值和a的第n位相同。

答案：

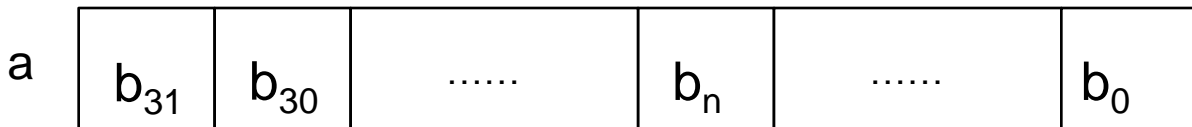
$(a \gg n) \& 1$

思考题：

有两个int型的变量a和n($0 \leq n \leq 31$),
要求写一个表达式, 使该表达式的值和a的第n位相同。

答案：

$(a \gg n) \& 1$

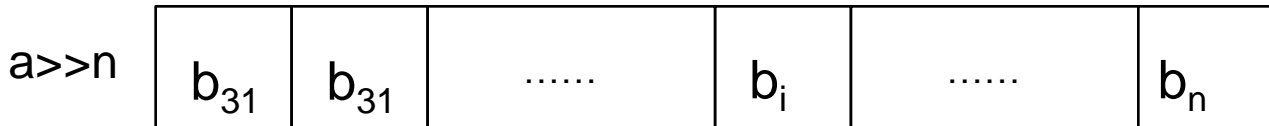
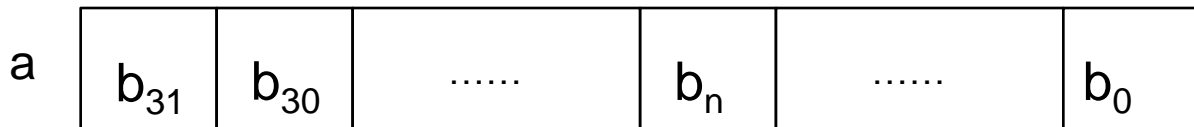


思考题：

有两个int型的变量a和n($0 \leq n \leq 31$),
要求写一个表达式, 使该表达式的值和a的第n位相同。

答案：

$(a \gg n) \& 1$



思考题:

有两个int型的变量a和n($0 \leq n \leq 31$),
要求写一个表达式, 使该表达式的值和a的第n位相同。

答案:

$(a \gg n) \& 1$

a

b_{31}	b_{30}	b_n	b_0
----------	----------	-------	-------	-------	-------

$a \gg n$

b_{31}	b_{31}	b_i	b_n
----------	----------	-------	-------	-------	-------

$(a \gg n) \& 1$

0	0	0	b_n
---	---	-------	---	-------	-------

思考题：

有两个int型的变量a和n($0 \leq n < 31$),
要求写一个表达式, 使该表达式的值和a的第n位相同。

另一答案：

$(a \ \& \ (1 \ll n)) \gg n$



“引用”的概念和应用

引用的概念 （教材第62页）

➤ 下面的写法定义了一个引用，并将其初始化为引用某个变量。

类型名 & 引用名 = 某变量名；

```
int n = 4;
```

```
int & r = n; // r引用了 n, r的类型是
```

引用的概念 (教材第62页)

➤ 下面的写法定义了一个引用，并将其初始化为引用某个变量。

类型名 & 引用名 = 某变量名;

```
int n = 4;
```

```
int & r = n; // r引用了 n, r的类型是 int &
```

引用的概念 (教材第62页)

➤ 下面的写法定义了一个引用，并将其初始化为引用某个变量。

类型名 & 引用名 = 某变量名;

```
int n = 4;
```

```
int & r = n; // r引用了 n, r的类型是 int &
```

➤ 某个变量的引用，等价于这个变量，相当于该变量的一个别名。

引用的概念

```
int n = 4;  
int & r = n;  
r = 4;  
cout << r; //输出 4  
cout << n;  
n = 5;  
cout << r;
```

引用的概念

```
int n = 4;  
int & r = n;  
r = 4;  
cout << r; //输出 4  
cout << n; //输出 4  
n = 5;  
cout << r;
```

引用的概念

```
int n = 4;  
int & r = n;  
r = 4;  
cout << r; //输出 4  
cout << n; //输出 4  
n = 5;  
cout << r; //输出 5
```

引用的概念

➤ 定义引用时一定要将其初始化成引用某个变量。

引用的概念

- 定义引用时一定要将其初始化成引用某个变量。
- 初始化后，它就一直引用该变量，不会再引用别的变量了。

引用的概念

- 定义引用时一定要将其初始化成引用某个变量。
- 初始化后，它就一直引用该变量，不会再引用别的变量了。
- 引用只能引用变量，不能引用常量和表达式。

引用的概念

```
double a = 4, b = 5;  
double & r1 = a;  
double & r2 = r1;  // r2也引用 a  
r2 = 10;  
cout << a << endl;  
r1 = b;  
cout << a << endl;
```

引用的概念

```
double a = 4, b = 5;  
double & r1 = a;  
double & r2 = r1; // r2也引用 a  
r2 = 10;  
cout << a << endl; // 输出 10  
r1 = b;  
cout << a << endl;
```

引用的概念

```
double a = 4, b = 5;  
double & r1 = a;  
double & r2 = r1;    // r2也引用 a  
r2 = 10;  
cout << a << endl;  // 输出 10  
r1 = b;              // r1并没有引用b  
cout << a << endl;
```

引用的概念

```
double a = 4, b = 5;  
double & r1 = a;  
double & r2 = r1;    // r2也引用 a  
r2 = 10;  
cout << a << endl;  // 输出 10  
r1 = b;              // r1并没有引用b  
cout << a << endl;  //输出 5
```

引用应用的简单示例

C语言中，如何编写交换两个整型变量值的函数？

引用应用的简单示例

C语言中，如何编写交换两个整型变量值的函数？

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

int n1, n2;
swap(&n1, &n2); // n1, n2的值被交换
```

引用应用的简单示例

➤有了C++的引用：

```
void swap( int & a, int & b)
```

```
{
```

```
    int tmp;
```

```
    tmp = a; a = b; b = tmp;
```

```
}
```

```
int n1, n2;
```

```
swap(n1,n2) ; // n1,n2的值被交换
```


引用作为函数的返回值(教材第63页)

```
int n = 4;
int & SetValue() { return n; }
int main()
{
    SetValue() = 40;
    cout << n;
    return 0;
}
```

引用作为函数的返回值(教材第63页)

```
int n = 4;
int & SetValue() { return n; }
int main()
{
    SetValue() = 40;
    cout << n;
    return 0;
} //输出: 40
```

常引用 (教材第65页)

定义引用时，前面加`const`关键字，即为“常引用”

```
int n;
```

```
const int & r = n;
```

r 的类型是

常引用

(教材第65页)

定义引用时，前面加`const`关键字，即为“常引用”

```
int n;
```

```
const int & r = n;
```

r 的类型是 `const int &`

常引用

不能通过常引用去修改其引用的内容：

```
int n = 100;  
const int & r = n;  
r = 200; //编译错  
n = 300; //没问题
```

常引用和非常引用的转换

`const T &` 和 `T &` 是不同的类型!!!

`T &` 类型的引用或`T`类型的变量可以用来初始化
`const T &` 类型的引用。

`const T` 类型的常变量和`const T &` 类型的引用则
不能用来初始化`T &`类型的引用，除非进行强制类型
转换。

QUIZ 1

下面程序片段哪个没错？

A) `int n = 4;`
 `int & r = n * 5;`

B) `int n = 6;`
 `const int & r = n;`
 `r = 7;`

C) `int n = 8;`
 `const int & r1 = n;`
 `int & r2 = r1;`

D) `int n = 8;`
 `int & r1 = n;`
 `const int r2 = r1;`

QUIZ 2

下面程序片段输出结果是什么？

```
int a = 1,b = 2;  
int & r = a;  
r = b;  
r = 7;  
cout << a << endl;
```

A) 1 B) 2 C) 7



下一小节：“const” 的用法



“const” 关键字的用法

1) 定义常量

```
const int MAX_VAL = 23;
```

```
const string SCHOOL_NAME = "Peking University"  
;
```

2) 定义常量指针

□ 不可通过常量指针修改其指向的内容

```
int n,m;  
const int * p = & n;  
* p = 5;  
n = 4;  
p = &m;
```

2) 定义常量指针

□ 不可通过常量指针修改其指向的内容

```
int n,m;  
const int * p = & n;  
* p = 5; //编译出错  
n = 4;  
p = &m;
```

2) 定义常量指针

□ 不可通过常量指针修改其指向的内容

```
int n,m;  
const int * p = & n;  
* p = 5; //编译出错  
n = 4;   //ok  
p = &m;
```

2) 定义常量指针

□ 不可通过常量指针修改其指向的内容

```
int n,m;
```

```
const int * p = & n;
```

```
* p = 5; //编译出错
```

```
n = 4; //ok
```

```
p = &m; //ok, 常量指针的指向可以变化
```

2) 定义常量指针

❑ 不能把常量指针赋值给非常量指针，反过来可以

```
const int * p1; int * p2;  
p1 = p2;    //ok  
p2 = p1;    //error  
p2 = (int * ) p1; //ok,强制类型转换
```


2) 定义常量指针

- 函数参数为常量指针时，可避免函数内部不小心改变参数指针所指地方的内容

```
void MyPrintf( const char * p )  
{  
    strcpy( p,"this"); //编译出错  
    printf("%s",p);    //ok  
}
```

3) 定义常引用

□ 不能通过常引用修改其引用的变量

```
int n;  
const int & r = n;  
r = 5; //error  
n = 4; //ok
```



下一小节：动态内存分配



动态内存分配

用new 运算符实现动态内存分配 (教材P109)

□ 第一种用法，分配一个变量：

P = new T;

T是任意类型名，P是类型为T * 的指针。

动态分配出一片大小为 sizeof(T) 字节的内存空间，并且将该内存空间的起始地址赋值给P。比如：

```
int * pn;  
pn = new int;  
* pn = 5;
```

用new 运算符实现动态内存分配 (教材P109)

□ 第二种用法, 分配一个数组:

P = new T[N];

T : 任意类型名

P : 类型为T * 的指针

N : 要分配的数组元素的个数, 可以是整型表达式

动态分配出一片大小为 `sizeof(T)` 字节的内存空间, 并且将该内存空间的起始地址赋值给P。

用new 运算符实现动态内存分配 (教材P109)

□ 动态分配数组示例:

```
int * pn;  
int i = 5;  
pn = new int[i * 20];  
pn[0] = 20;  
pn[100] = 30; //编译没问题。运行时导致数组越界
```

用delete运算符释放动态分配的内存

- 用“new”动态分配的内存空间，一定要用“delete”运算符进行释放

delete 指针; //该指针必须指向new出来的空间

```
int * p = new int;
```

```
* p = 5;
```

```
delete p;
```

```
delete p; //导致异常，一片空间不能被delete多次
```


用delete运算符释放动态分配的数组

□ 用“delete”释放动态分配的数组，要加“[]”

delete [] 指针; //该指针必须指向new出来的数组

```
int * p = new int[20];  
p[0] = 1;  
delete [] p;
```



下一小节： 内联函数、函数重载和函数缺省参数



内联函数 函数重载 函数缺省参数

内联函数 (教材P66)

- 函数调用是有时间开销的。如果函数本身只有几条语句，执行非常快，而且函数被反复执行很多次，相比之下调用函数所产生的这个开销就会显得比较大。
- 为了减少函数调用的开销，引入了内联函数机制。编译器处理对内联函数的调用语句时，是将整个函数的代码插入到调用语句处，而不会产生调用函数的语句。

内联函数 (教材P66)

```
inline int Max(int a,int b)
{
    if( a > b) return a;
    return b;
}
```

函数重载 (教材P67)

- 一个或多个函数，名字相同，然而参数个数或参数类型不相同，这叫做函数的重载。

- 以下三个函数是重载关系：

```
int Max(double f1,double f2) { }
```

```
int Max(int n1,int n2) { }
```

```
int Max(int n1,int n2,int n3) { }
```

- 函数重载使得函数命名变得简单。
- 编译器根据调用语句的中的实参的个数和类型判断应该调用哪个函数。

函数重载 (教材P67)

(1) int Max(double f1,double f2) { }

(2) int Max(int n1,int n2) { }

(3) int Max(int n1,int n2,int n3) { }

Max(3.4,2.5); //调用 (1)

Max(2,4); //调用 (2)

Max(1,2,3); //调用 (3)

Max(3,2.4); //error,二义性

函数的缺省参数(教材P61):

- C++中，定义函数的时候可以让最右边的连续若干个参数有缺省值，那么调用函数的时候，若相应位置不写参数，参数就是缺省值。

```
void func( int x1, int x2 = 2, int x3 = 3) { }
```

```
func(10 ) ; //等效于 func(10,2,3)
```

```
func(10,8) ; //等效于 func(10,8,3)
```

```
func(10, , 8) ; //不行,只能最右边的连续若干个参数缺省
```


函数的缺省参数(教材P61):

- 函数参数可缺省的目的在于提高程序的可扩充性。
- 即如果某个写好的函数要添加新的参数，而原先那些调用该函数的语句，未必需要使用新增的参数，那么为了避免对原先那些函数调用语句的修改，就可以使用缺省参数。