§2.5 运算符：

ANSI C++标准保留了ANSI C中丰富的运算符，并增加了三个新的运算符，即new、delete、::(作用域运算符)。运算量、功能（操作内容）、运算顺序等为运算符的三大要素。如果运算符所操作的运算量个数为一个、两个、三个，可将其分为单目运算符、双目运算符、三目运算符。例如，取负数运算符（-），为单目运算符，它只对一个运算量进行操作；加法运算符（+）、比较运算符（<，<=，.....），它对两个运算量进行操作。C++中仅有一个三目运算符，即条件运算符（？ ：），它可对三个运算量进行操作。

1．表达式中的类型转换：C++与C语言一样具有两种类型转换。

(1)隐式类型转换：这种类型转换是在执行运算符的操作时，自动进行的一种转换，有时也称自动类型转换。

①当不同类型的变量同时出现在一个表达式中时，实现“隐式类型转换”。它是在数据运算过程中自动进行的类型转换。由编译系统把它们转变成同一类型的变量再进行运算。转换的原则是参加运算的各种变量都转换为它们之中数据最长的数据类型，并且转换总是逐个运算符进行的。这样转换处理，使数据的精度不会降低。其转换的方向如下：

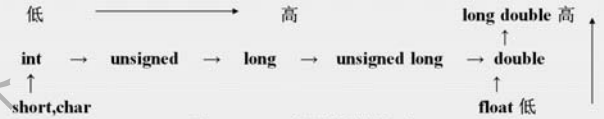
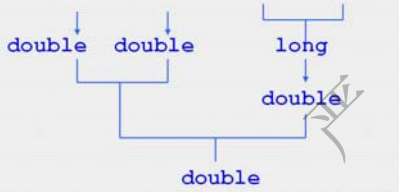


图 2 - 6 类型转换的方向

例如：
```
float f = 3.6;
int n = 6;
long k = 21;
double ss = f * n + k / 2;
```



计算ss时，首先将f（float型）和n（int型）转换成double型数，算得它们的积为21.6。然后计算k/2得整除运算结果10（long int型），再将long int型的数字10转换成double型数10.0。21.6(double型)和10.0(double型)两个数相加，得到最后结果31.6(double型)。

②在赋值语句中，当左值(赋值运算符左边的值)和右值(赋值运算符右边的值)的类型不同时，一律将右值类型自动转换为左值的类型(向左看齐)。

③在调用有返回值的函数时，总是将return后面的表达式类型自动转换为该函数的类型（当两者类型不一致时）。

```
#include <iostream>
using namespace std;
int add(double x , double y)
{   cout << "(2)x + y = "
         << x + y << endl;
      return x + y;
}
void main( )
{  int a , d;
   double b(3.56) , c(4.6);
   a = b;
   cout << "(1)a = " << a
        << " , b = " << b << endl;
   d = add(b , c);
   cout << "(3)d = " << d << endl;
}
```


该程序的输出结果：

```
(1)a = 3 , b = 3.56
(2)x + y = 8.16
(3)d = 8
```

在表达式a = b;中，将右值b(double型)转换成为int型值3，然后赋给左值a。而在调用add( )函数时，将return后面的表达式(x+y)，其结果double型值8.16转换成函数的类型，即int型数8赋给d。

(2)强制类型转换：

此类转换因有明确的程序书写格式，由编程者按格式书写程序来控制，又称“显式类型转换”。ANSI C++有两种书写格式：

①强制转换表示法：与ANSI C标准相同，其格式：

（类型名）表达式

编译时，将表达式强制转换成（类型名）所指定的类型。例如：

```
double s ;
int n = 2;
s = sqrt((double) n);
```


②函数表示法：在ANSI C++中，基本数据类型名可当成一个函数名使用，其格式：

基本数据类型名（表达式）

其作用是将（表达式）强制转换成指定的基本数据类型。

例如：

```
double f = 6.86;
int h = int(f);//或int h =(int)f;
```

函数表示法不能用于非基本数据类型，例如，把一个正整数值转换为指针类型，必须使用强制转换表示法：

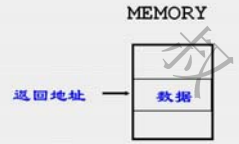
```
char *p = (char *)0x00650666;
```

C++ 程序设计

2.指针函数：一个函数可以带回一个整型值、字符值、实型值等。也可带回指针型的数据(即地址值)，把返回值为地址值的函数称为指针函数。其定义格式为：

```
<存储类><类型> * 函数名(形参表)
{
    ...
    return  地址表达式;
}
```

<存储类>是函数本身的存储类型，与一般函数相同分extern型和static型。而<类型>是返回的地址值所指的内存空间中存储数据的数据类型。



C++ 程序设计

显然，与一般函数的定义相比较，指针函数的定义语句只是在函数名前面加上一个'*'号，表示该函数的返回值是指针型的。

试有下述程序，它将main( )函数内的一个int型自动变量复制到一个安全地方保存起来。存储空间通过调用标准函数库中的指针函数void * malloc( )来得到。所谓"安全的地方"，不因main( )函数内还有其他函数调用语句，而破坏了该变量的值。

C++ 程序设计

```
#include <stdio.h>
#include <stdlib.h>
void main( )
{  int x = 60;
   int * p;
   if((p =(int *)malloc(sizeof(int)))
        == NULL) {
        printf("\n Heap error !");
        exit(1);
   }
   else
       * p = x;  //将x保存到"安全的地方"
       printf("Save a integer : %d\n", * p);
          ...          //有其他函数调用语句
   free(p);            //在该程序段内x被可靠保存
   printf("Memory freed !\n);
}
```

异常终止路径

C++ 程序设计

该程序的输出结果：

```
Save a integer : 60
Memory freed !
```

C++ 程序设计

(1)动态数据结构：C++中经常使用的各种数组象变量数组、指针数组、结构数组等，它们在内存中占用连续的内存空间。所占内存空间的位置和大小是在定义时由系统分配的。在程序运行期间，其内存空间的位置和大小是固定不变的。这些数组的数据结构称为静态数据结构，静态数据结构中各元素的位置相对固定，便于访问它们的任一元素。但在数组中删除或插入一个元素则比较困难。因此C++的数组与C语言一样通常定义成static型，由编译系统分配固定的内存空间。

另一种数据结构是动态数据结构，它们所占用的内存空间在程序运行期间可以动态地变化。即采用动态存储技术：在程序运行期间，根据需要为某种数据结构动态地分配它们所需要的存储空间，并在使用完后释放这些内存空间以备他用。例如上述程序中的int型自动变量x的值为60。编程者希望可靠地保存它，这可以通过编写程序来控制。而不是象自动变量那样，离开了它的作用域(main( )函数体内)就消失了，它的值不再保留，再次进入该作用域时又要重新给它赋初值。

C++ 程序设计

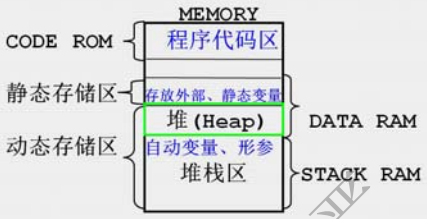
(2)malloc( )和free( )函数：为了实现动态存储技术，C语言的标准函数库特设置了一对标准函数，它们的原型在stdlib.h中。因此在使用它们的程序开头处，必须写有：

```
#include <stdlib.h>

void * malloc(unsigned size);
void free(void * ptr);
```

很多动态分配存储器的标准函数被说明为void *型的指针(无类型指针)，而由编程者根据需要用强制转换指定它的类型。因此void有两个用途①用于定义无返回值的函数。②用void *定义一个”无类型的指针”，它是一种便于灵活使用的指针类型。

C++ 程序设计



(3)由malloc( )函数所分配的内存空间放在数据RAM区的堆(heap)中。堆(heap)是一个自由存储区域，它由编程者设计程序来控制，象经常使用的链表、树、有向图等动态数据结构的存储问题，在ANSI C标准中都是调用malloc( )函数来动态分配内存空间的。

C++ 程序设计

(4)当malloc( )函数执行成功时，其返回值是大小为size的内存空间首地址，否则为空指针。因此在使用malloc( )函数时，必须测试其返回值不为空指针，不然将导致系统毁坏。同时由于它是无类型指针(void *型)，还必须用强制类型转换指明它的类型，以便函数返回的指针转换成正确的类型。

句中，必须使用地址表达式，该表达式的值就是函数的返回值，其后跟随的表达式结果值可以是变量的地址、数组首地址或已经定向的指针变量，还有结构变量地址、结构数组的首地址等，这些变量和数组是全局型或静态型都可以，但不能把自动变量的地址和自动型数组的首地址作为指针函数的返回值，因为在该函数结束时，它们将自动消失，其存放空间将被释放。

```
#include <stdio.h>//comp.c
typedef struct {
   double real;   double imag;
} COMPLEX;
//COMPLEX result;   //为extern型结构变量
COMPLEX * cadd(COMPLEX * op1,
               COMPLEX * op2)
{  COMPLEX result;   //为自动型结构变量
   result.real = op1->real +op2->real;
   result.imag=op1->imag +op2 -> imag;
   return &result;
}
```



```
void main( )
{ COMPLEX a = {6.0, 8.0};
  COMPLEX b = {2.0, 8.0}, * rp;
  printf("A(%.2lf , %.2lf)\n",
          a.real, a.imag);
  rp = cadd(&a, &b);
  printf("Result(%.2lf,%.2lf)\n",
          rp ->real, rp -> imag);
}
```


3.new和delete运算符：

C++为便于用户应用动态存储技术，直接提供new(动态分配存储空间，类似于malloc( )函数)和delete(释放存储空间，类似于free( )函数)运算符来实现动态存储管理功能。

(1)运算符new的用法：

new运算符的使用格式如下：

```
    new <类型>(初值表)          ①
或  new <类型>                  ②
```

它表明在堆(heap)中动态建立了一个由<类型>所指定的变量(或对象)。第①种形式由圆括号包围的“初值表”给出被创建变量(或对象)的初始值。也可采用不赋初始值的第②种形式，但必须给它赋值后才能参加运算和操作。这里所说的<类型>包含所有的基本数据类型和派生类型(复杂的数据类型、构造类型)，以及用户定义的class类型。

①new和delete运算符的功能与ANSI C中的malloc( )和free( )标准函数基本相同。new为任意类型的对象在堆(Heap)中分配一块所需的存储空间，并返回该存储空间的首地址。如果堆中没有足够的存储空间或分配出错时返回空(NULL)指针。因此与使用malloc( )函数一样，必须检测其返回值不为空指针，例如：

```
#include <iostream>
#include <stdlib.h>
using namespace std;
void main( )
{   int x = 60;
    int * p;
    if((p =(int *)malloc(sizeof(int)))
            == NULL) {
              printf("\n Heap error !");
               exit(1);                    异常终止路径
    }
    if((p = new int) == NULL) {
              cout << "Heap error !\n";
              exit(1);                    异常终止路径
    }
    else    * p = x; //赋值操作，将x安全地保存在堆中。
         cout << "Save a integer : "
          << x << endl;
          . . .   //调用其他函数的语句
    delete p; //在该程序段内x被可靠保存
         cout << "Memory freed !\n";
}
```


②正如上例中，可采用赋值操作创建新的动态变量或动态对象，其格式为：

指针名 = new 类型; 例中：p = new int;

必须预先定义好同类型的指针，这种形式便于应用if语句，检测其返回值是否为空指针，可写为：

if((指针名 = new 类型) == NULL){ 。。。}

例中：if((p = new int) == NULL){ 。。。}

顺便指出，该指针名就是所创建的动态对象名。

由于：

简化

if(p == NULL)（即if(p == 0)）→ if(!p)

简化

if(p != NULL)（即if(p != 0)）→ if(p)

一般格式可写为：

一般格式可写为：

if(! (指针名 = new 类型)){出错处理操作；}

if(指针名 = new 类型){创建成功的处理操作；}

有时，也可直接采用初始化操作创建新的动态对象，其格式为：

类型 * 指针名 = new 类型 (初值表) ;

③new和delete优于malloc( )和free( )，运算符new自动计算<类型>的大小而不需使用sizeof运算符，而且返回的指针能正确指向该类型，而不需强制类型转换处理。更重要的是new和delete可用于用户定义的class类型。

④用new也可动态地创建一个数组并为它的元素在堆中分配存储空间，new的返回值是数组第1个元素的地址，其格式为：

<类型>*指针变量名 = new <类型>[元素个数];

例如：int * pa = new int[10];

即创建了具有10个元素的int型(动态)数组pa，对单个变量也可以采用这种格式，只不过元素个数为1。

例如：

```
float *px = new float[1];
float *py = new float[1];
```

用这种格式创建对象时，只是给对象分配了内存空间，并没有对它赋初值。因此在使用前，还必须用赋值操作对其赋值。例如：

```
*px = 12.6;
*py = 16.8;
```

⑤采用地址赋值，把两个指针指向同一内存位置，将产生”别名”(Aliasing)问题，这对于用new创建的对象最容易发生。.

C++ 程序设计

```
#include <iostream>
using namespace std;
void main( )
{  int a,* pa = &a;
   int b,* pb = &b;

   * pa = 28;
   * pb = 64;
   cout << " (1) a = " << * pa
        << " , b = " << * pb << endl;
   pa = pb;
   cout << " (2) a = " << * pa
        << " , b = " << * pb << endl;
  * pa = 206;
  cout << " (3) a = " << * pa
       << " , b = " << * pb << endl;
}
```

C++ 程序设计

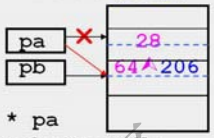
该程序的输出结果：

```
(1) a = 28 , b = 64
(2) a = 64 , b = 64
(3) a = 206 , b = 206
```

指针变量pa和pb分别指向两个int 型变量a和b，给a和b赋值分别为28和64。当执行了pa = pb;语句后，pa和pb指向了同一内存位置。因此再通过pa和pb访问的变量都是变量b，原来存放变量a的内存空间将被封锁，再也不可能访问这一内存空间了，从而导致”内存挂起”。特别对于用new创建的对象，”内存挂起”会造成积累。对于大型软件，在堆(Heap)中”内存挂起”积累太多，可能引起堆溢出，所以用new创建对象时，必须用前述方法测试new的返回值。

C++ 程序设计

```
#include <iostream>
using namespace std;
void main( )
{  int *pa = new int;
   int *pb = new int;
   * pa = 28;
   * pb = 64;
   cout << " (1) a = " << * pa
        << " , b = " << * pb << endl;
  pa = pb; //pa和pb指向了同一内存位置
   cout << " (2) a = " << * pa
        << " , b = " << * pb << endl;
  * pa = 206;
  cout << " (3) a = " << * pa
       << " , b = " << * pb << endl;
  delete []pa;
  delete []pb;//对同一内存空间执行了两次delete操作
}
```



C++ 程序设计

⑥用new创建的对象，其生存期是整个程序运行期间，直到显式地用delete运算符撤消它。

(2) delete运算符的用法：

①delete运算符只能作用于new返回的指针，对于其他指针则没有意义。其格式为：

```
delete 指针名;
```

//该指针保存new分配的内存空间首地址。

②用delete运算符撤消由new创建的数组时采用如下格式：

```
delete [ ] 指向数组的指针名;
```

这里所说的数组可以是基本数据类型的数组，或者是用户定义的class类型的对象数组。

③对一个(用new创建的)对象只能使用一次delete操作。例如：alias.cpp中，由于pa和pb指向了同一内存位置，实质上是已变成了一个对象，因此第2次用delete [ ] pb;再次撤消它时出错。

课堂讲述：Delete 和delete[]的对普通数据类型和对象的使用事项

4.C++中的运算符

表2.3列出了C++运算符集的优先级和结合规则。说明如下：

(1)C++保留了C语言丰富的运算符集，并增加了三个新的运算符，即new、delete、::作用域运算符。优先级和结合规则与C语言基本相同。

(2)C++可以重新定义(或称”重载”)这些运算符，这将在以后章节中讲述。

(3)在ANSI C中，++和--只能用于整型量，而ANSI C++中对整型和浮点型均可使用。

表2.4 C++常用运算符的功能、优先级和结合规则

优先级	运 算 符	名 称	结合规则
1	( )	函数参数表，改变优先级	从左至右
	::	作用域运算符	
	[ ]	数组下标	
	. , ->	成员选择(访问)符	
	.* , ->*	成员指针选择符	
2	++ , --	增1，减1运算符	从右至左
	&	取地址	
	*	取内容	
	!	逻辑求反	
	~	按位求反	
	+ , -	取正数，取负数	
	(数据类型)	强制类型转换	
	sizeof	计算数据类型长度	
	new , delete	动态存储分配	
3	* , / , %	乘法，除法，取余	
4	+ , -	加法 , 减法	
5	<< , >>	左移位 右移位	从左至右

6	<, <=, > , >=	小于，小于等于，大于，大于等于	从左至右
7	== , ! =	判断相等与不等	
8	&	按位求逻辑与	
9	^	按位求异或	
10	|	按位求逻辑或	
11	&&	逻辑与	
12	||	逻辑或	
13	? :	条件运算符	从右至左
14	=, +=, -=, *=, /=, %=, &=, ^= |=, <<=, >>=	赋值运算符	
15	,	顺序运算符	从左至右