本 课 程 初 步安 排

- 1. 限于时间，只讲Nav2010代码中用到的C++特性。
- 2. 从2月17日开始，共7天时间，每天上午(9:00~12:00)讲课3小时，下午练习。
- 3. 2月25日下午考试。
- (1) 选择题 (60%)
- (2) 编程题 (40%)

第一课 C++与C的关系

第一部分 C++的发展历史简介

第二部分 C++对C常规性能的扩充

第一部分　C++发展历史简介

随着面向对象程序设计思想的日益普及，很多支持面向对象程序设计方法语言也相继出现了，C++就是这样一种语言。C++是Bjarne Stroustrup于1980年在AT&T开发的一种语言，最初这种扩展后的语言称为带类的C，1983年才被正式称为C++语言。C++语言由C语言扩展而来，同时它又对C语言的发展产生了很大的影响，ANSI C语言在后来的标准化过程中吸收了C++语言中某些语言成分。

What is C++?

C++ is a general-purpose programming language with a bias towards system programming that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

C++的标准化进程

- C语言的标准化(ISO/IEC C9899)

  C89, C94, C99

- C++语言的标准化(ISO/IEC 14482)

  C++98, C++03, C++0x

参考读物

初级—— Programming -- Principles and Practice Using C++, C++ Primer Plus, …

中级—— C++ Primer , Effective C++, More Effective C++, Generic programming and the STL, Professional C++, …

高级—— The C++ Programming Language , Inside The C++ Object Model, ISO/IEC 14482, …

第二部分 C++对C常规性能的扩充

1. 基本输入流和输出流类

除了C语言的标准输入输出外，C++语言又提供了类层次结构的输入输出流类库。完整的C++输入输出流类库在很多书籍中介绍，这里不再详述。

为方便讲解中的程序举例，我们简单介绍C++中最常用的基本输入流和输出流。

流库中iostream类是最常用的基本输入输出流类。iostream类是基本输入类istream和ostream多重继承派生出的。iostream类中包括了键盘输入类、屏幕输出类和错误信息输出类。cin、cout和cerr分别为键盘输入类、屏幕输出类和错误信息输出类的系统默认对象。cin对象键盘输入的运算符为“>>”，叫“提取”；cout对象和cerr对象屏幕输出的运算符为“<<”，叫“插入”。

例如：

```
#include <iostream>        // 包含iostream头文件
int main()                 // 返回类型必须是int
{
  char name[30];
  std::cout << “name: ”;  // 输出到终端设备
  std::cin >> name;        // 从终端设备输入
 std::cout << name << ‘\n’;
 return 0;                 // 可以省略
}
```

上例中，第一行用include语句包含了iostream头文件。cin是键盘输入类的系统默认对象，它的输入操作的运算符为>>，它的参数为变量name。运算符可看作函数的另一种形式的表示，所以运算符也可以有参数。cout是屏幕输出类的系统默认对象，它的输出运算符为<<。

输入运算符和输出运算符都允许一个对象连续多次使用。转义字符\n的功能是换一行。endl也是换行，每执行一次endl操作换一行。

与C语言的输入函数scanf( )和输出函数printf()相比，C++语言的cin对象的输入运算符(>>)和cout对象的输出运算符(<<)，能对系统的基本数据类型自动进行匹配，并能自动进行格式转换。另外，当要输入输出数据的类型不是系统的基本数据类型，而是用户自定义的数据类型时，C++语言的输入输出方法允许用户通过把该自定义数据类型作为参数，重载输入运算符>>和输出运算符<<来方便地输入输出用户自定义数据类型的数据。C++语言的这些性能极大地方便了用户的程序设计。

2. 变 量

C++语言中的变量和C语言中的变量相比，功能扩充之处主要体现在：变量的定义方法、作用域限定运算符、枚举类型、结构体类型、const类型限定符、变量引用等。

2.1 变量的定义方法

变量类型 变量名;

跟C中变量定义方法完全相同，不同之处在于引用。

```
const int a = 1;

double b = 1.0 * a;

int c = a;

char* cstr1 = “1234567890”;     // C风格字符串

const char* cstr2 = “abcdefg”; // 跟上面一样
```

```
int main()
{
   int ii, jj, tt, v(6);
   for(int i = 0; i < 10; i++)
   {
     for(int j = 0; j < i ; j++) {}
     int t = i << 1;
   }
        ii = i;          // error!
   //   jj = j;          // error!
   //   tt = t;          // error!
}
```

变量ii，jj，tt的作用域是整个主函数；变量i的作用域是从外层循环的for语句处到外循环末尾；变量j的作用域是从内循环的for语句处到内循环结束处；变量t的作用域是从定义处到外循环结束处。

声明与定义

“声明”是向系统报个到，仅仅是通知；

“定义”是要分配空间的。可以伴随着初始化动作。

```
int  a;  //在标准C中这叫声明，在C++这是定义
int  * p;   // 同上
int  A[10];   // 同上
extern int b;   //在标准C和C++中这都叫外部变量声明
struct s;   //在标准C和C++中这都叫前向声明
void fun (int , float);  //在标准C和C++中这都叫函数原型声明
```

变量的初始化

全局变量的初始化、表达式中的隐含类型转换、类中隐含成员的初始化等都是编译器的责任。

局部变量的初始化、表达式中的强制类型转换、类中非静态成员的初始化等都是程序员的责任。

请留意类型

在C中，NULL的类型可能是void *。

如果你想在C++中使用NULL，可以这样定义：

```
const int NULL = 0;
```

类型占用字节的大小依赖于编译器和系统，例如：

bool类型在32位系统上通产占用一个字节。

在C89之前，未声明的类型默认int型，C89之后则不再允许，C++不支持默认类型，必须显式指定。

void 是其大小无法确定的类型，不可以用sizeof 来测量，只能用于指针和函数。

标准C允许非void *与void * 之间相互转换；

如可以将int * 转换成void * ,再将void *转换成double *。

在C++中可以把任意类型的指针指派给void类型的指针——只寄存了地址；但不可以把void类型的指针指派给非void类型的指针，除非强转转换；这样限制的一个很重要的原因是考虑到delete会强制级联调用析构函数。

2.2 作用域限定运算符

C++语言的运算符 :: 称作作用域运算符，用于解决变量的名字冲突问题，主要用于访问一个在当前作用域内被当前局部变量隐藏的外层全局变量。下例中变量count有两个定义，定义在函数外的是整型，定义在函数内的是浮点型。定义在函数外的变量count作用域是整个文件，定义在主函数内的变量count作用域是整个主函数。变量 ::count就表示出了在当前作用域内被局部变量count隐藏了的全局变量count。

```
int sum = 2;
int main()
{
  double sum = 0.1F;
  sum = 5.5;  // 局部count
  ::sum = 3;  // 全局count
}
```

C++语言的运算符 :: 还可用于限定数据成员或成员函数所属的类，这将在以后的章节中介绍。

2.3 关于等号运算符（=）：

在C语言中 = 运算符称作赋值运算符，但也用于对变量或数组以及指针的初始化。一身兼二职，使许多人搞不清其准确含义。

在C++中给予清晰地规范：只作为运算符，不再承担初始化的职能，初始化工作一律由构造函数完成。但由于还要全面兼容C语言，所以又把C的不清晰带了进来。

int a = 30; //此时的等号是完成了 a 的初始化工作，应称为定义符。 C++则提倡写成int a （30）；以示与对象一致。

x = y; //这才是赋值 。明显的标志是：前面没有类型名

2.4 枚举类型

和C语言不同的是，C++语言的枚举类型是一个真正的类型名，即在C++语言中枚举类型可像其它类型一样使用。系统对待枚举变量将像对待其它变量一样。

C++的枚举类型一般形式是：

enum 枚举类型名 {枚举列表} [枚举变量表] ;

其中，enum是枚举类型标识关键字，枚举列表中定义的枚举值对应着符号数字常量，其编号从0开始。枚举变量可以放在以后需要变量时再定义。枚举类型允许变量使用的操作只有一个，即赋值操作。一个使用枚举类型定义变量的例子如下：

```
enum Color { Red, Gree, Yellow };
#include <iostream>

int main()
{
    Color  MyColor;          // Color是枚举类型名
    MyColor = Yellow;        //对枚举变量赋值操作
    std::cout << MyColor; // 默认转换成整型常量2
    MyColor = 5;             // 不能对枚举变量赋整数值
}
```

2.5 布尔类型

C++语言新增了bool类型。

bool的值只有true和false，可以默认准换成整数1和0，所有非整数值都可以默认为是true

但这两个值既不是1和0,也不是字符串。因此bool类型的变量只能在程序的语句中对其赋值，不可以从键盘输入。

2.6 结构体类型

C++语言定义结构体类型的方法和C语言的定义方法基本类同，差别主要有下面两点。

(1) C++语言的结构体类型定义和使用更简单一些。结构体一旦定义，就可以直接使用该结构体名定义变量，而不用处处在结构体名前加关键字struct

例如：

```
struct MyStrut { };
MyStruct s;          // 不必struct MyStrut s;
```

(2) C++语言把结构体看作类的特殊形式，即结构体是将成员默认为public的类。换句话说，在C++语言的结构体中也是类，也可以像类的定义那样，定义结构体的私有的、公有的成员函数和数据成员。关于结构体中成员函数的定义和使用方法可对照类的成员函数的定义和使用方法。

```
struct A
{
  // …
};
```

⇔

```
class A
{
public:
  // …
};
```

2.7 const类型限定符

C++语言中，const类型限定符用于限定某标识符在定义域范围内为常量，所限定的常量、函数参数等成为只读的，不能被修改。在C语言编程序时，通常用宏来定义常量，如

```
#define MAX 100;
```

用C++语言编程序时，通常用const来定义常量，如

```
const int MAX = 100;
```

虽然两种方法均可定义程序中使用的常量，编译时编译器处理方式不同。前者在预编译或预处理阶段直接代换，后者则发生在编译时。

归纳const的用途主要有六点：

(1) 当const类型说明符用于说明变量时，其作用是冻结所定义的变量在定义域范围内为常量。C++要求用const说明的变量必须在定义时初始化赋值，以后不允许再有赋值操作。

const说明的常量和同类型的变量具有单向兼容性。即，const说明的常量可赋给变量，但反之则不允许。如下例中，变量y定义为整型常量，若要把变量赋给该常量将出错。

```
int main()
{
    int x = 4;              // 定义为变量
    const int y = 5;        // 定义为常量

    x = y;                  // ok!
    y = x;                  // error, y is read-only.
}
```

const说明的形参可被同类型的非const实参初始化。

```
#include <iostream>
void F1(const int x)
{
   std::cout << x << std::endl;
}
void F2( int x)
{
   std::cout << x << std::endl;
}
int main()
{
    const int  y = 20;
    int x = 5;
    F1(x);      // 合法
    F2(y);      // 亦合法
}
```

例中，函数F1中可使用常整型和非常整型的实参。函数F2中的形式参数定义为非常整型，主函数中的y定义为常整型，函数调用时意味着x是可以修改的。 当然改的不是y，而是x.

(2) 当const类型说明符用于说明引用时，则称为常引用。即不能通过引用名来修改原变量的值。

```
int  i = 100;   const int & ri = i ;
```

(3) 当const类型说明符用于说明指针类型变量时，有三种情况：冻结所定义的指针变量所指向的数据；冻结指针的地址值；或者冻结指针变量所指向的数据和指针的地址值。

对于指针变量的这三种情况，const类型说明符的用法将不同。例如：

```
int main()
{
    const int x = 5, y = 6;
    int m = 10;
    const int *p1 = &x;          // 定义指针所指向的量为常量
    int * const p2 = &m;         // 定义指针自身为常量
    const int * const p3 = &x; //指针所指的量和指针自身为常量
     p1 = &y;  // 修改指针地址操作合法
      //*p1 = 4; //非法操作, 修改指针地址内的值

     //p2 = &y; //非法操作，修改指针地址
     *p2 = 3;   // 修改指针地址内的值操作合法

    //p3 = &y; //非法操作，修改指针地址
    //*p3 = 3;  //非法操作，修改指针地址内的值
}
```

(4) 当const类型用于函数的形式参数时，其作用是冻结实际参数在该函数内为常量，即该参数不允许在函数内被修改形参可以是const int x或const int& rx。

(5) 当const说明符用于函数的返回类型时，其作用是限定该函数返回的是个常量:　const int func();

该函数的调用定是 const int a = func();

(6) 当const类型说明符用于类的成员函数时，其作用是不允许该成员函数修改对象的数据成员。

int classname::mem_func() const ;

这样的例子将在类与对象的章节中介绍。

使用const的好处：

它允许指定一种语意上的约束——某对象不能被修改。编译器将严守这一约束，对函数进行检查。这实际上是函数对调用者所作的承诺。

常量必须在定义时就初始化。

当变量不能为const时, 可以使用mutable，其意思是永远不为常量，就算加上const，也能修改。

3. 引 用 类 型

引用类型是C++语言新增加的一个类型，引用类型用标识符 & 表示。引用是给已存在的变量或对象起一个别名，即引用引入了一个变量或对象的同义词。当建立引用时，程序用另一个变量或者对象的名字初始化它。从那时起，引用就作为目标的别名而使用，对引用的改动实际上是对目标本身的改动。

C++语言增加引用类型主要用在三个方面：

(1) 定义变量或对象的别名；

(2) 定义函数的参数为引用型；

(3) 定义函数的返回值类型是引用型。

引用变量必须在定义时初始化，此后就再也不能改变了，这个意义上讲它已经是常量；

给引用变量赋值，修改的是它所依附的变量本身的内容，而不是让引用改换门庭。这正是引用与指针的区别所在。

```
int  ii = 0; jj = 10;
const int&  ci = ii;     // ci 成为ii的另一个名字
ci = jj;  //错误。这不是让ci成为jj的别名，而是要把ii的值变成10, 因为ci是const型,不可改，但ii = 10; 却是可以的。
int& test = 1;// 不行，因为1是常量，类型不符
const int& index = 1;// 却可以，Why?
```

原因有些微妙，需要作些解释：

引用在内部存放的是一个对象的地址，它是该对象的别名。对于不可寻址的值，如文字常量以及不同类型的对象，编译器为了实现，必须生成一个临时对象，引用实际上指向该对象，但用户不能访问它。例如当我们写：**double dval = 1024; const int &ri = dval;**

编译器将其转换成：

int temp = dval; const int &ri = temp;

若允许非**const** 引用指向需要临时对象的对象或值，则意味着可以通过引用修改常量，显然不合逻辑。而**const** 引用是只读的，用它无后顾之忧。

引用的初始化方式：

```
#include<iostream>
int main()
{
  using std::cout;
  using std::endl;
  int i=110;
  int* p = &i;
  int& r = * p;
  cout << i <<"   "<< *p <<"   "<< r << endl;
  int* q = new int ( 220);
  int& rq = *q;
  cout << *q <<"   "<< rq << endl;
  delete &rq;
}
```

3.1 变量或对象的别名

被引用的对象可以无名，但不可是临时对象。

下边的例子中可以看出，可以定义一个引用，它的初始化是由另一个引用完成的；

另外，可以定义指针类型的引用变量，此时要注意，求地址运算符&和引用标识符&完全一样，但使用的场合是不同的。

引用的操作：对引用的操作就是对其所代表的变量的操作，如果程序寻找引用的地址，则为引用所依附的目标的地址。

例：

```
int  i=5;
int& ri1 = i, & ri2 = ri1;
cout<<i <<’ ’ <<ri1 <<’ ’ <<ri2 <<endl;
i *=3;
cout<<i<<’ ’ <<ri1 <<’ ’ <<ri2 <<endl;
ri1 +=5;
cout<<i<<’ ’ <<ri1 <<’ ’ <<ri2 <<endl;
```

引用与指针差别很大，指针是个变量，可以把它再变成指向别处的地址，然而，建立引用时必须初始化，并且不能再依附其他不同的变量了。

```
# include <iostream>
int main(void)
{
    int i = 5;
    int& ii = i;
    ii = 6;                          // 对ii的修改就是对i的修改
    cout << i << endl;
    int& iii = ii;                   // 引用变量可传递定义
    iii = 7;                         // 对iii的修改就是对i的修改
    std::cout << i << std::endl;

    int a[10];

    int& rap[10]=a;  //不能建立数组的引用.写成int &rap=a; 更错
```

```
    int* p = &i;                // 这里&i是求地址
    int*& pp = p;               // 定义指针的引用
    *pp = 8;    // 修改了p所指向的内容
     std::cout << *p << std::endl;
}
```

C++中加入引用的主要目的：让代码看上去简洁。

3.2 函数的引用类型参数

在C语言中，系统处理函数参数传递的方法是用函数的实际参数初始化形式参数，即函数处理的仅仅是存放在系统栈中的实际参数的副本。这样，当函数结束，系统把存放在系统栈中的形式参数自动释放，这些局部数据值就不复存在了。若是在函数中修改了形式参数，并需要把这些修改值返回给实际参数，就得使用传址。例如，下边设计的**swap1( )**函数用于交换两个参数的数据值，但它只交换了两个形式参数的数据值，两个实际参数的数据值并未被交换。

```
void swap1(int x, int y)
{
  int tmp = x;
  x = y;
  y = tmp;
}

# include <iostream>

int main(void)
{
  int x = 5, y = 6;
  swap1(x, y);
  std::cout << x << “,  " << y << std::endl;
}
```

```
void swap2(int *px, int *py)
{
   int tmp = *px;
  *px = *py;
  *py = tmp;
}

#include <iostream>
int main()
{
  int x = 5, y = 6;
  swap2(&x, &y);
  std::cout << "x = " << x << std::endl;
  std::cout << "y = " << y << std::endl;
}
```

```
inline void swap3(int &x, int &y)
{
  int tmp = x;
  x = y;
  y = tmp;
}

#include <iostream>
int main()
{
  int x = 5, y = 6;
  swap3(x, y);
  std::cout << "x = " << x << std::endl;
  std::cout << "y = " << y << std::endl;
}
```

分析这段程序的执行结果:

比较引用作形参、静态局部变量、函数返回三者的用法

```
int calculator()
{
int  sum1 = 0, sum2 = 0;
int  aa = 2, ab = 3, ac = 4;
sum1 = Area( aa, &sum2 );
//此时sum1, sum2是多少?
sum1 += Area( ab, &sum2);
//此时sum1, sum2是多少?
sum1 += Area( ac, &sum2);
//此时sum1, sum2是多少?
}
```

```
int  pi = 3;
int  Area(int r, int *s)
{
  int b;
  static int c =0;
  b = pi * r * r;
  c += b;
  *s = c;
  return b ;
}
```

4. 函　数

4.1 函数原型

C++对函数返回值不做类型检查，只要求在函数使用前给出函数返回值的类型的定义即可(讲函数重载时会详细论述这个问题)。例如，下边例子中只要在使用函数max之前定义了函数返回值的类型，则系统不会判错。对函数的实际参数和形式参数不做类型匹配检查，经常会引起系统运行出错。

```
#include <stdio.h>

int max(int, int);  // 写成int max();可以吗?
int main()
{
   int x = 5, y = 6;
   int z = max(x, y);
   printf("z = %d", z);
}
```

```
inline int max(int x, int y)
{
   return y < x ? x : y;
}
```

C++语言对函数参数做类型检查，它要求任何函数在使用之前，都必须有该函数的定义，并把这种函数定义称作函数原型，系统依据函数原型对实际参数和形式参数做类型匹配检查。函数原型的形式为：

返回类型　函数名(形参列表);

上述例子在C++语言中，定义函数返回值的类型时（第2行）应写成如下形式：

```
int max(int x, int y);                    // 函数原型定义
```

由于函数原型需要定义的是函数参数的类型和函数返回值的类型，所以写成如下只有参数类型、没有参数名的形式也可以：

```
int max(int , int );
```

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
  long fac(int n);
  int n;
  long y;
  cout<<"Enter a positive integer:";
  cin>>n;
  y=fac(n);
  cout<<n<<"!="<<y<<endl;
}
```

```
unsigned long fac(unsigned n)
{
  if(n < 1) return 1;
  else return n*fac(n-1);
}
```

运行结果：

```
Enter a positive integer:8

8!=40320
```

参数传递的本质

在参数中使用指针就一定能完成对值的修改吗？

请看下例：

```
void GetMemory(char *p,  int num)
{ p = (char *)malloc(sizeof(char) * num);  }
void Test(void)
{
    char *str = 0;
    GetMemory(str, 100);      // 尽管p得到了空间，可
                                str 仍然为 NULL
    strcpy(str, “hello”);     // 运行错误
}
```

参数传递的本质

在C/C++中，函数在传递参数时的动作永远是拷贝(在C++里，传递引用是个例外)。

对于变量，拷贝的结果是传值；

对于指针，这拷贝的结果还是传值，只不过是指针的值，也就是在传别人的地址，于是称其为传址。只不过需要注意的是在函数体内要按指针的规律操作它。

在C++里面，传递引用是个例外，它传递变量本身。

函数的返回值也是采用复制的方式将处理结果交给调用者的(引用例外)。

引用的不当使用

```
const string& select(bool cond, const string a, const string b)
{     if(cond)
         return a;
      else
         return b;
}

int main()
{
  string frt, scd;
  bool useFirst = true;
  const string &  name = select(useFirst,frt, scd);
  // …
}
```

4.2 内联函数

函数是模块化软件设计的基本单位，一个软件系统中一般要包括许多函数模块。但系统在进行函数调用时，需要花费额外的空间资源和时间资源，因此，在函数模块太小或函数模块需要被频繁地调用这两种情况下，划分函数模块又是程序的沉重负担，对运行效率不利。

为了解决这种矛盾，C语言程序通常采用宏的方法，而C++语言增加了内联函数也为了解决此矛盾。

内联函数的标识符关键字是inline，使用方法是在函数的开始处加inline。

注意：使用inline只是向编译器建议，最后的决定在于编译器的内联能力。

内联原则：频繁调用的短代码段最好做成内联函数。

C++中加入内联的原因：宏容易导致很多很难检查的bug。况且宏不能重载、不能递归、不能继承。

Bjarne Stroustrup在AT&T担任大规模程序设计的负责人期间，发现大约50%以上的问题由宏引起，这是C++中引入内联的最主要原因之一。此外，内联跟宏相比，不存在效率问题。

C++语言的内联函数和C语言的宏相比，内联函数的主要优点是对参数类型进行一致性检查，可早期发现许多编码错误。

在类设计中，类的成员函数的内联函数可有两种表示方法：一种方法是加标识符inline，另一种方法是把函数体直接写在类定义内。

成员函数内联有两种方式：

在类定义中实现函数

```
class A
{
public:
  void display() const
  {
    // …
  }
};
```

在类外实现时加inline (显式内联)

```
class A
{
public:
void display() const;
};
inline A::dispaly()
{
  //...
}
```

5. 函数重载

将同一个名字用于同一作用域的不同类型的多个函数的情况叫重载。

什么时候发生函数重载？

在同一作用域内，两个或两个以上的函数，其函数名相同，参数类型或参数个数或参数次序或函数属性不同。

```
class A
{
public:
  void f(int ii);                 // overloaded
  void f(int i, int ii);          // overloaded
  void f();                       // overloaded
  void f(int ii) const;           // overloaded
  int f(int i);                   // error, redefined
  virtual void f(int ii);         // error, redefined
private:
  void f();                       // error, redefined
  // …
};
```

6. 带默认形参值的函数

在C++语言中调用函数时，通常要为函数的每个形参给定对应的实参。若没有给出实参，则按指定的默认值进行工作。

当一个函数既有定义又有声明时，形参的默认值必须在声明中指定，而不能放在定义中指定。只有当函数没有声明时，才可以在函数定义中指定形参的默认值。

默认值的定义必须遵守从右到左的顺序，如果某个形参没有默认值，则它左边的参数就不能有默认值。

如：

```
void func1(int a, double b=4.5, int c=3); //合法
void func1(int a=1, double b, int c=3);   //不合法
```

函数实现的设计规则：

经验告诉我们，函数的“入口”和“出口”是最容易出问题的地方。

“入口”是指函数的形参，它理应接受调用者传来的有效数据。但你要多个心眼：传来的数据未必都有效。想想海关商检就明白了。

“出口”是指函数的返回，此处容易效率差甚至出错。

千万不要返回局部对象的引用、指针；

返回局部对象时要注意效率：

```
return  string(s1+s2);  同于
        string  result(s1+s2);
        return result;
```

请比较下面两段函数代码，注意他们的差别：

```
char *  fn1 ()   //此函数错误
{    char str[ ] = “hello world”;    //字符串存在栈上
     cout<< sizeof(str) << endl;   //12
     cout <<strlen(str)<<endl;     //11
     return str;      // 返回野指针(幽灵指针)
}
const  char *  fn2 ( void )   //此函数正确
{
    char *p = “hello world”;       // 字符串存于何处?
    cout<< sizeof(p)<< endl;       // 4
    cout<< strlen(p)<<endl;        // 11
    return p;      // 返回字符串常量的地址
}
```

7 new和delete 运算符

C++语言增加了用于动态存储空间分配和释放的运算符new和delete，不再用函数来完成。所谓动态存储空间是指系统提供的一个称为堆(heap)的内存区域。对于这个区域，系统不再知道其生命期，也不再负责其消亡。当用户动态地需要存储空间时，可以用new运算符向系统动态申请；当用户申请的空间使用完时，应使用delete运算符释放归还给系统。

堆空间的管理需要较大的系统开销。

C语言进行动态存储空间分配和释放的方法是使用malloc()函数（还有其它几个函数）和free( )函数。和C语言的函数方法相比，C++语言的运算符方法有以下三个优点：

(1) C++语言的运算符方法进行类型检查，这可防止很多可能的错误。

(2) C++语言的运算符方法会根据类型自动计算要分配的空间大小，这样使用更简单。

(3) 运算符属于C++语言的一部分，这样使用起来不用像C语言那样用宏包含（#include）语句把这部分函数包含进来。

new运算符的语法格式为：

new 类型名 (初始值)

其中，类型名指定了要分配存储空间的类型。当动态申请单个变量或对象时，可以有初始值，也可以没有初始值；当动态申请数组变量或对象时，不允许有初始值。要注意的是，只有类的构造函数参数为空（void）或全部参数都带有缺省值时，才可以动态申请数组对象。当动态空间申请成功，new运算符按要求分配一块内存，并返回指向该内存起始地址的指针；当动态空间申请不成功时，new运算符返回空指针NULL。

new 包括两个过程：

分配内存，调用执行构造函数构造对象，例如：

```
Date *pDate = new Date(8, 25);  // 调用Date(int, int)
```

delete 包括两个过程：

执行析构函数，释放内存空间

```
delete pDate;                          //调用Date:: ~Date()
```

new[ ]和delete[ ]

new[ ]是创建多个对象的空间；

delete[ ]是销毁多个对象的空间。

```
Date *pDate = new Date[100];
// 以上语句分配了可容纳100个Data对象的内存空间，
// 但不会调用构造函数创建对象
delete[ ] pDate;      //销毁这100个对象占用的内存空间
```

注意：

new和delete需要配对。

当使用new运算符定义一个一维数组变量或数组对象时，它产生一个指向数组第一个元素的指针；当使用new运算符定义一个多维数组变量或数组对象时，它产生一个指向数组第一个元素的指针，返回的类型保持了除最左边维数外的所有维数。

下边是几个引自Visual C++联机用户帮助手册的数组变量或对象的例子。

(1) char* pstr = new char[10];

// 申请一个一维字符类型数组空间，由字符类型指针pstr指示。一维数组的元素个数为10。

(2)char (*pchar)[10] ;

//定义pchar为指向字符类型二维数组的指针变量

pchar = new char[10][10];

// 申请一个二维字符类型数组空间由指针变量pchar //指示，二维数组为10 * 10 。

当申请二维或二维以上数组空间时，除第一维可为变量外，其余维必须为常数

(3)float (*cp)[25][10];

// 定义cp为浮点二维数组类型的指针变量，指向的是三维数组： cp = new float[10][25][10];

// 申请一个三维浮点类型数组空间由指针cp指示

delete运算符用于释放由new运算符分配的动态存储空间。delete运算符的语法格式为：

delete 指针

或 delete [ ]指针

其中，指针指向的必须是先前用new运算符分配的动态存储空间。当释放动态分配的数组空间时，用第二种格式。

8 行 注 释

C++语言中可以使用两种注释，一种是从C语言继承来的/* */，此外增加了一种单行注释方法，即在要注释的语句行前加标识符//。

C99标准中规定也可以用//注释。

9 namespace的用途

namespace 的设计目的：为了尽可能解决名字冲突。

定义一个namespace的方法：

namespace 命名空间名字 { //… }

例如：namespace mySpace

```
{
  // 我定义的各种全局函数、类等。
}
```

假如两个命名空间同名怎么办？系统自动把它们拼接成一个。

namespace的使用方法

例如：

```
#include <iostream>
using std::cout;
using std::endl;
```

或std::cout << “1234567890” << std::endl;

using namespace std; // 这种用法很常见啊;-)

这种用法很方便，但严重违背了OOP的精神，存在隐患。

练　习

1. 编写程序，判断一个正整数是否是偶数。(提示：用位运算)
2. 编写程序，把一个整数从10进制转换成16进制。(提示：用位运算)
3. 编写程序，比较两个数的大小返回较小者。
4. 编写程序，判断一个正整数是否是质数。
5. 编写程序，判断一个正整数是否是2的整数次幂？例如1, 2, 4, 8, 16, 32, …是2的整数次幂，3, 5, 6, 7, 9,10, 11, 12, …则不是。(提示：用位运算)

6. 运行如下程序，结果正确吗，这个swap被调用了吗？为什么？

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

#include <iostream>
using namespace std;
int main()
{
  double a = 0.111111F, b = 0.222222F;
  swap(a, b);
  std::cout << a << ", " << b << '\n';
}
```

7. 找出下列程序中的错误。

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

int main()
{
  const char* str = “0123456789”;          // 定义一个C风格的字符串
  char* pstr = new char[std::strlen(str)]; //分配适量堆内存空间
  std::strcpy(pstr, str);                  // 复制字符串内容
  std::cout << pstr << '\n';               // 显示拷贝的字符串
  if(str == pstr)                          // 判断两个字符串是否相等
      std::cout << "string copied ok!";
  delete pstr;                             // 释放堆内存
  return 0;
}
```

8. 编码实现C库函数strcpy，strcmp, strlen, strcat

char *strcpy(char *to, const char *from);

size_t strlen(const char *str); // typedef unsigned long size_t;

int strcmp(const char *str1, const char *str2);

char* strcat(char *str1, const char *str2);

9. 找一本适合自己的C++参考读物阅读。

Thanks!