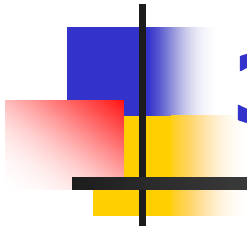
第三章 C++中的C

面向对象程序设计(C++)



3.1 C++的词法及词法规则

3.1 C++的词法及词法规则

1.标识符: 由字母和数字组成，首字符必须是字母("_"也看作字母)的函数名字、类名、变量名、常量名、对象名、标号名、类型名等。

合法的标识符：hello97，a_dog，_uname, CLASS
非法的标识符：97hello, a#123, class

2.关键字: 系统已预定义的单词，属于保留字，用户不可再重新定义。

asm auto break case catch char class const continue default delete do double else enum extern float for friend goto if inline int long new operator private protected public register return short signed sizeof static struct switch template this throw try typedef union unsigned virtual void volatile while bool true false

3.1 C++的词法及词法规则(cont.)

3.运算符:

- 系统预定义的函数名字 eg: +,-,*,/,sizeof
- 作用域分辨符： ::

  eg class_name::member
- 全局作用域分辨符： ::

  eg ::name
- 堆分配/回收运算符： new , delete

  eg int * pint; pint=new int(10)

  int * parr; parr=new int[10];

  delete pint;

  delete[] parr;
- 成员节： .* object.*pointer_to_member
- 成员节： ->* pointer->*pointer_to_member

3.1 C++的词法及词法规则(cont.)

4.分隔符（标点符号）

- 空格：单词与单词之间　int　i;
- 逗号：如说明多个变量，函数的多个参数分隔　int i,j,k;
- 分号：语句之间，如for循环中, for(int i;i<10;i++)
- 冒号：语句标号与语句之间的分隔符
- 括号:　()
- 大括号：{ }

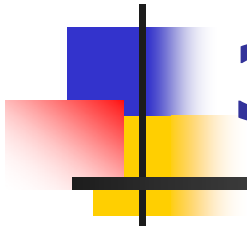
5. 常量: 在程序中直接使用符号表示的数据。

如：3.1415, “hello world!”

6. 注释符

/* … */　　/* This program writes two messages to the screen*/

//　　// This is also a comment.



3.2 数据类型简介

数据类型定义**使用存储空间(内存)的方式**。通过定义数据类型，告诉编译器怎样创建一片特定的存储空间，以及怎样操纵这片存储空间。

```
int  i, j, k ;     // 内定义类型
struct student  stu[40];     // 结构体
Cstudent   cstu[10];         // 类
```

3.2.1 基本内部类型

- char/ int/ float/ double/ void

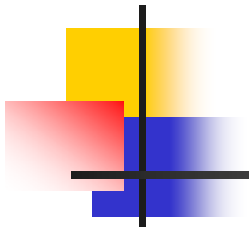
```
int main() {
    /* Definition without     initialization:   */
    char protein;
    int carbohydrates;
    float fiber;
    double fat;
    /* Simultaneous definition & initialization: */
    char pizza = 'A', pop = 'Z';
    int dongdings = 100, twinkles = 150,
    heehos = 200;
    float chocolate = 3.14159;
     // Exponential notation:
     double fudge_ripple = 6e-4;
} ///:
```

```
char *p="abc";     /* 字符串指针(const)    warning?

char a[10]="Hello";

char a[10];
a="Hello";                ?     error

char a[10]="Hello";               ?
```



3.2.1 基本数据类型-修饰符

修饰符

- signed
- unsigned
- long
- short
- const
- volatile（见下页）

3.2.1 基本数据类型-修饰符(续)

- volatile：告诉编译程序，该变量值可能按程序中没有显示说明的方式改变，防止编译器作不正确的优化。

eg: 用户程序中定义的GlobalVar被OS的时钟子程序用来存放时间。

```
volatile int GlobalVar;
int i,k;
k=GlobalVar;
for(i=1;i<10000;i++){   cout<<i;   }
k=GlobalVar;
```

（尽管在用户程序中未显式修改其值，但GlobalVar的值仍可能变化。）

3.2.1 基本数据类型-字长

• 字长

```
#include<iostream.h>
void main()
{
        cout << sizeof(char) << endl;
        cout << sizeof(int) << endl;
        cout << sizeof(float) << endl;
        cout << sizeof(double) << endl;
        cout << sizeof(long double) << endl;
}
```

输出:
(硬件相关)

```
1    (1)
2    (4)
4    (4)
8    (8)
10   (8)
```

注： C的规范(C++继承)并不说明每一个内部类型必须有多少位，而是规定内部类型必须能存储的最大值和最小值。因此，内部类型的字长跟硬件相关。

3.2.1 基本数据类型-bool类型

C中：非0值表示“真”，0表示“假”

C++: 内部常量true表示“真”，false表示“假”

```
bool   tag;

tag = true ;
if (…)   {
      tag = false ;
}

if(tag)   cout<<“TRUE”;
else     cout << “FALSE” ;
```

3.2.2 构造类型--数组

- **数组：**数目固定、类型相同的若干个变量的有序集合；在内存中顺序存储。

  - **定义：** <类型>　<数组名>[N1] [N2]…[Nk] ;

```
int a[3];            // 一维数组
char b[3][5];        // 二维数组
float c[3][5][7];    // 三维数组
```

思考：

- ✓ a+1 vs b+1 vs c+1?
- ✓ c[2][3][4] 相对与数组起始地址的偏移?

3.2.2 构造类型--数组(续)

- **String:字符串数组**

char s[5] = “abcd”;
char s[3][4] = {“abc”, “ mnp”, “ xyz”};
char s[ ][4] = {“12”, “ er”, “ a”}; ……对/错???

char s[4][ ] = {“12”,”er”,”a”};……对/错???

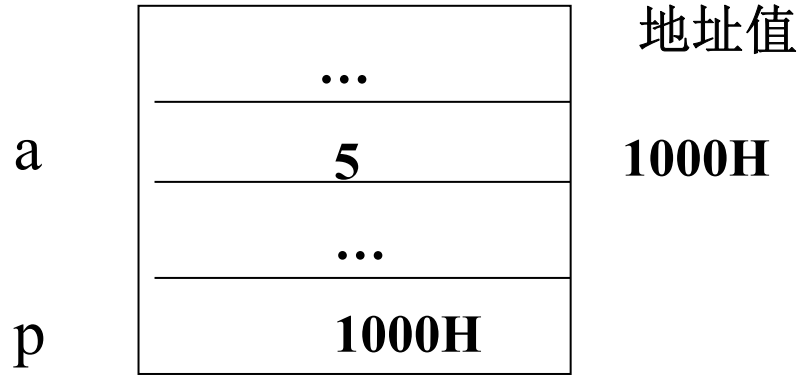
(char s[4][ ]=… 错误定义)

问：跟普通数组的区别?
答：字符串数组可以整体参与运算。

3.2.3 构造类型：指针

- **指针：是一种特殊的**变量，**具有一般变量的三个基本要素：**名字、类型、值；
- **指针的**类型**是它所指向变量的类型；**
- **指针的**值**是该指针“指向”的变量的地址。**

```
int a=5;      // a的值是5，类型是 int ，存储在 DS:1000H中
int *p = &a;  // p的值是DS:1000H, p的类型是 int ,存储在
```



想一想：

问：下列语句

```
int a=5;
int *p;
*p=a;
```

错在哪里?

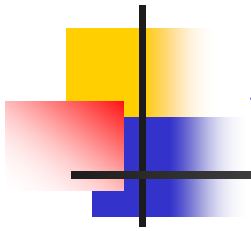
想一想：

问：下列语句

```
int a=5;
int *p;
*p=a;
```

错在哪里?

答：p没有被初始化，容易造成内存误用



例：

```
int *pi;          // int型指针
char *pi;         //  char 型指针
char (*pi)[3]  // 指向数组的指针
int (*pi)( );     //  函数指针
int  *test();     //  指针函数
int **pi;         //  指向指针的指针
int *pi[3];       //  指针数组
```

3.2.3.3 指针与数组的等价性

```
char  buf[ ];
char *pbuf;
```

// &buf[0] 与pbuf 具有相同的意义。
// 数组名不能赋值运算： ++buf 是错误的。

```
pbuf----------------&buf[0]
pbuf + 1-----------&buf[1]
*pbuf--------------buf[0]
*(pbuf + 1)--------buf[1]
*(pbuf + index)---buf[index]
```

- **二维数组：**

  a[3][2]　　=　　*(*(a+3)+2)

- **数组名指向整个对象**

  char a[3];　　//　&a类型为char(*)[3]

  char (*p)[3]=&a;　　//　指向数组的指针
  char *p=&a; //error
  char *p=&a[0]; // right

例1：

```
#include <iostream.h>
void main( )
{
        static int a[5] = {5,4,3,2,4};
        int i,j;
        i = a[2] + a[4];
        j = *(a+2) + *(a+4)
        cout << i << “\n”<< j ;
}
```

例1：

```
#include <iostream.h>
void main( )
{
        static int a[5] = {5,4,3,2,4};
        int i,j;
        i = a[2] + a[4];
        j = *(a+2) + *(a+4)
        cout << i << “\n”<< j ;
}
```

输出：7 7

例2：

```
#include <iostream.h>
void main( )
{
        static int b[ ][3] = {{1,2,3},{4},{5,6}};
        b[0][2] = 7;
        b[1][2] = 8;
        cout << **b <<“\t”<<**(b+1)
             <<“\t”<<*(*(b+1)+2) << “\n”;
        cout << b[0][2] + b[1][2] + b[2][2];
}
```

例2：

```
#include <iostream.h>
void main( )
{
        static int b[ ][3] = {{1,2,3},{4},{5,6}};
        b[0][2] = 7;
        b[1][2] = 8;
        cout << **b <<“\t”<<**(b+1)
             <<“\t”<<*(*(b+1)+2) << “\n”;
        cout << b[0][2] + b[1][2] + b[2][2];
}
```

输出： 1 4 8 15

例3：

遍历数组：

```
void f(char v[])
{
      for(int i=0; v[i]!=0;++i)
            use(v[i])
}

void f(char v[])
{
      for(char *p=v;*p!=0;++p)  //字符数组最后一位为'\0'
            use(*p)
}
```

例4：

下列哪些语句有错？

```
void f(int* pi)
{
        void* pv=pi;
        *pv;
        ++pv;

        int* pi2=static_cast<int*>(pv);

        double*  pd1=pv;
        double*  pd2=pi;
        double*  pd3=static_cast<double*>(pv);
}
```

例4：

下列哪些语句有错？

```
void f(int* pi)
{
        void* pv=pi;
        *pv;
        ++pv;

        int* pi2=static_cast<int*>(pv);

        double*  pd1=pv;
        double*  pd2=pi;
        double*  pd3=static_cast<double*>(pv);
}
```

3.2.4 引用(Reference)

- 指针是通过地址间接访问某个变量，而引用是通过**别名**直接访问某个变量，对引用的操作就是对被引用的变量的操作。

```
int a=10;
int &b=a;  // b是a的别名;
b=b+10;    // 对b的改变，实际上改变的是a;
cout << a ;
```

3.2.4 引用(Reference)

指针是通过地址间接访问某个变量，而引用是通过**别名**直接访问某个变量，对引用的操作就是对被引用的变量的操作。

```
int a=10;
int &b=a;  // b是a的别名;
b=b+10;    // 对b的改变，实际上改变的是a;
cout << a ;
```

输出：20

3.2.4.1 注意

- 引用在定义时必须初始化；初始化后，不能改变引用的“指向”。

```
int a=10; int &b=a;
int &c; // error,未初始化。
```

- 可以用某个引用的地址值赋给一个指针，而指针则指向被引用的变量。

```
int a=10; int &b=a;
int *p;  p=&b;    // 则p指向a
```

- 可以用某个引用初始化另一个引用。

```
int a;
int &r1 = a;
int &r2 = r1;
r2 = 10;      // a的值被改变。
```

3.2.5 创建类型别名：typedef

```
Typedef unsigned long ulong;
ulong  ll;

Typedef int * IntPtr;
Intptr x,y;  // x,y 为int *

Typedef struct{
    int a;
    int b;
    double c;
 }structure2;*structure3;

 structure2 s1, s2;    //s1,s2为结构体变量
 structure3 s3, s4;    //s3,s4为结构体指针变量
```

3.3 运算符

- **优先级**

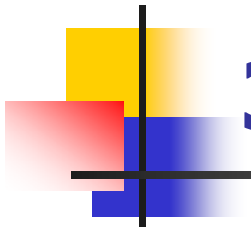
  决定了运算符在表达式中的运算顺序

- **结合性**

  也是决定运算顺序的一种标志。

  结合性：从左到右：（绝大部分）

  从右到左：单目、三目、赋值



3.3.1 C中的运算符

（略）

3.3.2 作用域分辨符：　::

- **类成员分辨**

  Class_name :: member

  例：StackInt::Pop();

- **全局作用域分辨**

  ::name

  例：
```
int x;
void f()
{   int x;
    x=2;        // 局部X
    ::x=10;     // 全局X
 }
```

3.3.3 New和delete

- new: 从“堆”中动态分配存储块。
- delete: 从“堆”中动态删除存储块。

```
int  *pInt;
pInt = new int ;     // 从堆中动态分配内存
if (pInt ==0)
      exit(-1);        // 申请失败
else
      { …;}           // 申请成功
 …
 delete pInt;        //释放空间
```

3.3.3 New和delete(续)

- 为数组分配空间/撤销空间

```
int size ;
cin>>size;
int * parrayInt = new int[size]; //动态申请
char * string = new char[20];
……;   //使用parrayInt

//释放空间, [ ]表示撤销整个数组,
delete [ ] parrayInt;
delete [ ] string;
```

3.3.4 New/delete VS malloc/free

- 编译时，New可以根据对象的类型，自动决定对象的大小；而malloc需要显式指定需要分配空间的大小。
- New返回指向正确类型的指针，不必进行强制类型转换；而malloc返回void *，必须进行强制类型转换，可能带来错误。

```
struct test{
        int a;
        char b;
        test(int ia=0,ch
};
test::test(int ia=0,int
{   a=ia; b=ib;}
```

C:

```
Main()
{   test * objt = (test *)
malloc(sizeof(struct te
    objt->test(86,'A');
    …;
    free(objt);
}
```

C++:

```
Main()
{
    test * objt = new
test(86,'A');
    …;
    delete objt;

}
```

3.3.4 New/delete VS malloc/free(续)

- 用new生成对象时,会调用构造函数, 用malloc则不会;同样,用delete删除对象时,会调用析构函数, 用free则不会.

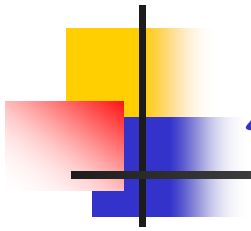
```
Time * pint = new Time;  // 正确地构造
delete pint;       // 正确地析构

Time * pint = new Time; // 正确地构造
free(pint) ; // 可以释放pint,但不调用析构函数,未能完
                // 整释放空间.

Time *pint = (Time *) malloc(sizeof(Time));
                // 仅分配空间, 未调用构造函数
delete pint;     // 正确地析构
```

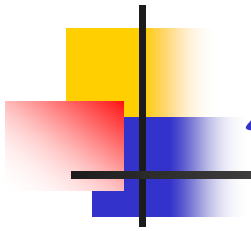
如果用free释放“new创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用delete释放“malloc申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。

因此，new/delete必须配对使用，malloc/free也一样。



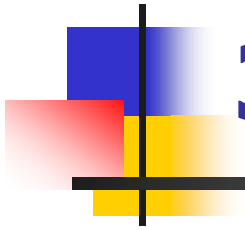
思考

引用和指针分别的适用范围?



思考

- 引用不会造成内存泄漏
- 引用不适合动态内存管理
- 链表等数据结构必须配合指针和动态内存使用
- 尽量只在动态内存管理时使用指针



3.4 表达式

所有表达式都有值！

3.4.1 优先级和结合性

i+++ ++i 或 **i+++(++i)**

等价于

(i++)+(++i)

空格

-1<0<1

等价于

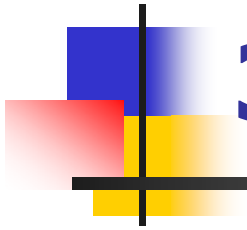
(-1<0) <1

3.4.2 副作用（结果依赖编译器）

```
int V[10],  i = 1;
v[i] = i ++;
```

可能等价于：v[1]=1;

或：v[2]=1;



3.5 语句

3.5.1 C++的语句

- 表达式语句和空语句

```
a+1;
    ;   //空语句
```

- 复合语句

```
a=a+1,b=b+1,c=c+1;
```

- 选择语句
  - 条件语句 if() {… }else{… } else if(){…}
  - 开关语句 switch()

```
{ case:…break;
  case:…break;
  default:…;
}
```

3.5.1 C++的语句(续)

- 循环语句

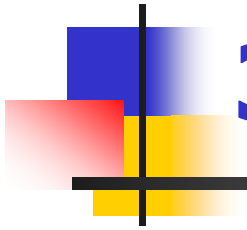
```
while(){…};
do{…} while();
for(;;;){…};
```

- 转向语句

```
label1: 语句1 ;
        … ;
        goto label1;  // 跳转到语句1
break;      // 退出整个循环
continue;   // 退出本次循环，进入下一个循环
```

3.5.2 例

```
#include <iostream.h>
int main()
{
   int i = 1;
   int sum = 0;
loop:
   if( i<=100 )
   {
    sum += i;
    i++;
    goto loop;
   }
   cout<<sum<<endl;
   return 0;
}
```



3.6 C++程序结构和编译环境

由类(class)和函数(function)组成

3.6.1 C++程序实例

```
#include<iostream.h>      // 预处理
#include<cstackint.h>     // 类 stackint  定义体

void main()          // 函数
{
      int  j=3;       // 内部类型变量
      stackint  si(10);   // 类的对象
      si.push(10);    // 类的成员函数
      j=si.pop();
      cout << “Hello, world!” << endl;      // 输入/输出
      cout << “j=“ << j << endl;
}
```

3.6.2 C++程序的基本组成

预处理

输入和输出

函数和类

语句

变量

其他

3.6.3 典型C++环境

- C++系统通常由几个部分组成：
  - 程序开发环境
  - 语言
  - C++标准库

- C++程序通常要经过6个阶段：编辑、预处理、编译、连接、装入和执行(以典型UNIX C++系统为例)

编辑 → a.cpp a.h
预处理 → a.cpp + a.h
编译 → a.obj
连接 → a.obj + Library= a.exe
装入 → a.exe -> memory
执行 → memory -> CPU

补充1：文件I/O

- C++提供以下类用于文件I/O (类比cin,cout)：

ofstream	写入文件
ifstream	读出文件
fstream	读写文件

```
#include <fstream>

int  main(){
      std::ofstream square_file; //建立文件操作流对象
      square_file.open("squares.txt");
      for (int i=0;i<10;++i)
            square_file<<i<<"^2="<<i*i<<std::endl;
      square_file.close();
}
```

补充2：错误处理

- 断言（assert）的用法：

```
#include <cassert>

double square_root(double x)
{
      check(x>=0);
      ……
      assert(result>=0); //如果不成立，程序立刻终止
      return result;
}
```

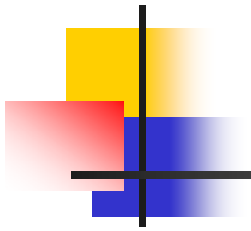
- 用于对程序的各种细节进行充分的测试（debug）
- 然后在正式版本中无效化debug代码，可以提高程序速度

```
#define NDEBUG
#include <cassert>
```

补充2：错误处理

- 异常处理：有一些错误在程序设计阶段很难提前避免，例如待读取文件已经被其它程序删除，或者程序需要内存时正好计算机内存已耗尽。

```
int read_matrix_file(const char* fname,…)
{
    fstream f(fname);
    if (!f.is_open())
       return 1;
    return 0;
}
```

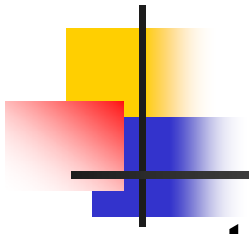


- 更标准的异常处理遵循try-catch模式

```
struct cannot_open_file {};  //自定义异常类型

void read_matrix_file(const char* fname,…)
{
      fstream f(fname);
      if (!f.is_open())
        throw cannot_open_file{};
}
```

- C++几乎允许程序员抛出任何类型的异常
- 大型程序开发通常需要建立自己的异常类型架构，一般从标准库std::exception派生而来
- 异常抛出后，可以选择即时处理或者延后处理



•try-catch模式的一个例子：

```
bool keep_trying=true;
do  {
  char fname[80];
  cin>>fname;
  try{
        A=read_matrix_file(fname);
        …
        keep_trying=false; //没有异常抛出，修改flag
  } catch (cannot_open_file& e){
        cout<<“Could not open the file. Try again!\n”;
  } catch (…) {  // 捕获所有其它异常
        …
  }
} while (keep_trying);
```

补充3：智能指针

- C++的智能指针主要方便内存管理，定义在头文件<memory>

Unique Pointer (对内存独占的控制权)

```
#include <memory>

int  main(){
      unique_ptr<double> dp{new double};
      *dp=7;
}
```

- 智能指针在生命周期结束时自动释放对象内存（因此智能指针指向的内存地址必须是动态分配的！）
- unique_ptr指针不能改变类型，也不能赋值给另一个unique_ptr

```
double* raw_dp=dp.get();  // okay

unique_ptr<double> dp2{dp}; // error

unique_ptr<double>  dp2{move(dp)}, dp3; // okay

dp3=move(dp2); // okay
```

```
std::unique_ptr<double> f()
{   return std::unique_ptr<double> {new double};}

int main(){
    unique_ptr<double> dp3;
    dp3=f(); // f()动态分配内存的所有权直接转移给dp3
}
```

Shared Pointer (对内存共享的控制权)

- shared_ptr 利用引用计数（额外的内存占用）实现内存的自动管理。每当复制一个shared_ptr，引用计数+1；一个shared_ptr离开作用域时，引用计数-1。引用计数清零时，自动delete内存

- 适用领域：并发和多线程

```
shared_ptr<double> p1=make_shared<double>();

// 数据内存地址和指针管理内存地址连续，提高缓存效率
```

```
shared_ptr<double> f()
{
   shared_ptr<double> p2{new double}, p3=p2; //copy?
   cout<<p3.use_count()<<endl;
   return p3;
}

int main()
{
   shared_ptr<double> p=f();
   cout<<p.use_count()<<endl;
}
```

运行这段代码，结果是什么，思考为什么?