

# 2023

# 面向对象程序设计

## 第二讲： C++ 基础

李际军 lijijun@cs.zju.edu.cn



# 学习目标 / GOALS

---

- ( 1 ) 了解 C++ 程序的组成部分;
- ( 2 ) 掌握命名空间、变量的的作用域与可见性及生存期的概念;
- ( 3 ) 掌握引用及函数的引用参数和返回引用的概念和使用;
- ( 4 ) 掌握带有默认参数的函数的使用;
- ( 5 ) 掌握内联函数和重载函数的使用;
- ( 6 ) 掌握动态内存分配和释放的方法;
- ( 7 ) 掌握磁盘文件的输入输出操作方法。

# 目录 / Contents

---

**01**

C++ 程序的组成部分

**02**

命名空间

**03**

C++ 数据的输入输出

**04**

引用

**05**

函数

**06**

变量的作用域与可见性

# 目录 / Contents

---

**07**

对象的生存期

**08**

const 常量

**09**

动态内存分配和释放

**10**

编译预处理

**11**

文件的输入和输出

# 01



## C++ 程序的组成部分

实例 1-1 在屏幕上显示“我们欢迎你”五个字。

该项目由两个文件组成：

头文件： A. h

源文件：  
Project1. cpp

## A. h 文件程序代码

```
class A          // 声明一个类 A
{
    public:
        void Print()          // 类的输出成员函数
    {
        cout<<" 我们欢迎你 !"<<endl;
        // 在屏幕上输出 “我们欢迎你 !”
    }
};
```

## Project1. cpp 文件程序代码

```
#include <iostream>
#include "A. h "

using namespace std;

int main()
{
    A a;
    a.Print();
    return 0;
}
```

由此可见，C++ 程序由**注释**、**编译预处理**和**程序主体**组成。C++ 程序的结构和书写格式归纳如下。



### C++ 程序组织结构

- ◆ C++ 程序可以由一个程序单元或多个程序单元构成。每一个程序单元作为一个文件。
- ◆ 一个 C++ 程序的组织结构一般由三部分组成：类的定义、类成员的实现和主函数。
- ◆ 如果比较的小程序，可以将这三部分写在同一个文件中。
- ◆ 在规模较大的项目中，往往需要多个程序文件，一般将一个类的定义写在头文件中，使用该类的编译单元则包含这个头文件。
- ◆ 因此，通常一个项目可以划分为三个文件：类声明文件（\*.h 文件）、类实现文件（\*.cpp 文件）和类的使用文件（main() 所在的 \*.cpp 文件）。
- ◆ 对于更为复杂的程序，每一个类都有单独的定义和实现文件，采用这样的组织结构可以对不同的文件进行单独的编写、编译，最后再连接，利于程序的调试和修改，实现多人合作开发。

### 编译预处理

- ◆ 在 Project1.cpp 文件中第一个“#”号是预处理标记。每个“#”开头的行称为编译预处理行，`#include` 称为文件包含预处理命令。
- ◆ `<stdafx.h>` 和 `<iostream>` 是两个头文件。
- ◆ `<stdafx.h>` 的英文全称为：Standard Application Framework Extensions（标准应用程序框架的扩展）。是预编译头文件。
- ◆ 所谓预编译头文件，就是把一个工程（Project）中使用的一些标准头文件（如 Windows.H、Afxwin.H）预先编译，以后该工程编译时，不再编译这部分头文件，仅仅使用预编译的结果。这样可以加快编译速度，节省时间。

### 编译预处理

- ◆ `<stdafx.h>` 文件中就包含了这些必要的标准头文件，因此所有的 .cpp 实现文件的第一条语句都是 `#include <stdafx.h>`
- ◆ 在 VC++ 中新建一个 project 时，系统会自动添加头 `stdafx.h` 和 `stdafx.cpp` 文件。
- ◆ `<iostream>` 是输入 / 输出流文件。
- ◆ 头文件有：系统头文件和自定义头文件。
- ◆ 本例中，`<iostream>` 是系统头文件，“A.h” 是自定义头文件。
- ◆ `<iostream>` 文件设置了 C++ 的 I/O 相关环境，定义了标准输入 / 输出流对象 `cout` 与 `cin` 等。例如，在程序中“A.h”文件中定义的 `Print()` 函数里调用 `cout` 对象，所以在程序的开头加入 `#include <iostream>` 语句。
- ◆ 注意： `<iostream>` 和 `<iostream.h>`，`<iostream.h>` 和 “`iostream.h`” 的区别。

### 注释

- 注释是程序员为程序作的说明，是提高程序可读性的一种手段。一般分为两种：

序言注释和解释性注释。

前者用于程序开头，说明程序或文件的名称、用途、编写时间、编写人等；

后者用于解释程序中难懂的地方。

在 C++ 程序中，可以使用 “`//`” 实现单行的注释，称为行注释。

也可以使用 `/*……*/` 表示多行的注释，称为块注释。

### 命名空间

C++ 标准中引入命名空间的概念，是为了避免在大规模程序的设计中，不同模块或者函数库中相同标识符命名冲突的问题。标准 C++ 引入了关键字 `namespace` 定义命名空间，用来控制标识符的作用域。标准 C++ 库（不包括标准 C 库）中的所有标识符（包括常量、变量、结构、类和函数等）都被定义在命名空间 `std`（standard 标准）中了。

在 `Project1.cpp` 文件中第二行 `using namespace std;` 的意思是“使用命名空间 `std`”。程序中如果需要使用 C++ 标准库中的有关内容，就需要使用“`using namespace std;`”语句进行申明，表示后续程序中要用到命名空间 `std` 中的内容。这条语句在使用标准函数库的 C++ 程序中频繁出现，本教程中大部分例子代码中也将用到它。

### 输入和输出

- `cout` 和 `cin` 是 C++ 预定义的流类对象，用来实现输入 / 输出功能。
- 输出操作由 `cout` 和插入流运算符 “<<” 结合，功能是将紧随其后的双引号内的字符原样输出到标准输出设备上（显示器）。
- `endl` 表示输出换行并刷新缓冲区。
- `cin` 和析取流运算符 “>>” 结合表示用户从标准输入设备（键盘）输入数据。
- 当用户输入数据时，所输入的数据类型。必须与对应的变量类型一致，否则将产生错误。
- 当输入多个数据时，用空格键或 `Tab` 键分隔，当全部数据输入完后，按 `Enter` 键表示输入结束。
- 在 C++ 中除了用 `cout` 和 `cin` 进行输出输入，也可以用 C 语言中的 `printf`（）和 `scanf`（）函数进行输出和输入。

### 类的定义

- ◆ 类是 C++ 新增加的重要的数据类型，是 C++ 对 C 的最重要的扩展。
- ◆ 有了类，可以实现面向对象程序设计方法中的封装、信息隐蔽、继承、派生、多态等功能。
- ◆ 在一个类中可以包括数据成员和成员函数，他们可以被指定为私有的 (private) 和公有的 (public) 属性。
- ◆ 公有类型成员定义了类的外部接口，在类外只能访问类的公有成员；私有类型成员只能被本类的成员函数访问，来自类外的任何访问都是非法的；保护类型成员的性质和私有成员的性质相似，差别在于继承过程中对派生类的影响不同。

### 主函数

- 程序中主函数 `main()` 是创建项目时在 `Projet1.cpp` 文件中自动生成的（也可以自己定义），它与 C 语言的 `Main()` 函数功能相同。
- `Main()` 函数是程序的入口，一般在其前面加一个类型声明符 `int`，表示该函数的返回值为一个整型（标准 C++ 规定 `main()` 函数必须申明为 `int` 型）。程序中语句 “`return 0`”，表示返回值为 “0”。



02



## 命名空间

### 1. 什么是命名空间

#### 命名冲突

在 C++ 中，名称（ name ）可以是符号常量、变量、宏、函数、结构、枚举、类、对象等。在大规模程序的设计中，开发过程都是团队合作，多个程序文档以及在程序员使用各种各样的 C++ 库时，在对标识符命名时就有可能发生命名冲突，从而导致程序出错。

## 1. 什么是命名空间

### 【例 2-1】命名冲突

//1. cpp

```
#include <iostream>
using namespace std;
int a=1;
int main()
{
    cout<< a<<endl;
    return 0;
}
```

//2. cpp

```
int a=2;
```

该程序由两个文件 1. cpp 和 2. cpp 组成，这两个文件都定义了全局变量 a，发生命名冲突。

编译时程序无法通过，并返回出错信息：

```
2. obj      :      error
LNK2005:      "int      a"
already      defined      in
1. obj
```

## 1. 什么是命名空间

### 定义

命名空间（namespace）是一种特殊的作用域，可以将不同的标识符集合在一个命名作用域内，这些标识符可以是类、对象、函数、变量、结构体、模板以及其他命名空间等。

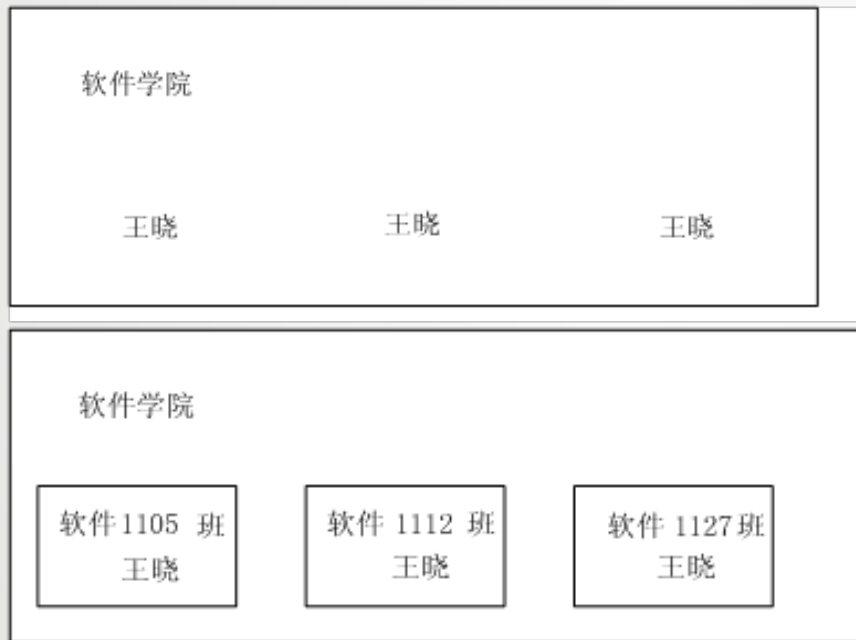
在作用域范围内使用命名空间就可以访问命名空间定义的标识符。

### 作用

- ◆ 命名空间的使用目的是为了将逻辑相关的标识符限定在一起，组成相应的命名空间，可使整个系统更加模块化，最重要的是它可以防止命名冲突。命名空间是用来组织和重用代码的编译单元。
- ◆ 有了命名空间，标识符就被限制在特定的范围内，在不同的命名空间中，即使使用同样的标识符表示不同的事物，也不会引起命名冲突。

## 1. 什么是命名空间

如下图所示：假如软件学院 2021 级的同学有三个名字叫王晓的同学。



### 2. C++ 中命名空间的定义

命名空间可以由程序员自己来创建，可以将不同的标识符集合在一个命名作用域内，包括类、对象、函数、变量及结构体等。在作用域范围内使用命名空间就可以访问命名空间定义的标识符。

## 2. C++ 中命名空间的定义

在 C++ 语言中，命名空间使用关键字 `namespace` 来声明，并使用 `{ }` 来界定命名空间的作用域，命名空间定义格式如下：

- `namespace` 命名空间标识符名
- `{`
- 成员的声明； // 类、对象、函数、变量及结构体等
- `}`

命名空间标识符名在所定义的域中必须是唯一的，和类，结构体类似，但不能实例化，只可以引用。

## 2. C++ 中命名空间的定义

```
namespace ABC
{
    int count;

    typedef float book_price;

    struct student {
        char *name;
        int age;
    };
    int add(int x, int y){return x+y;}

    int min(int x, int y);
}
```

```
int ABC::min(int x, int y)
{
    return x>y?x:y;
}
```



## 2. C++ 中命名空间的定义

### 【例 2-2】命名空间的定义

```
#include <iostream>
using namespace std;
namespace A
{
    int x=2
};
```

```
namespace B
{
    int x=5;
    void Print();
};
int main()
{
    cout<<A::x<<B::x<<endl;
    return 0;
}
```

### 2. C++ 中命名空间的定义

说明：

1. 命名空间标识符名在所定义的域中必须是唯一的；
2. 命名空间作用域不能以分号结束；
3. 命名空间可以在全局作用域或其他作用域（另一个命名空间）内部定义，但不能在函数或类内部定义；
4. 命名空间和类、结构体类似，但不能实例化，只能引用；
5. 命名空间的成员都是公有的，不能对它们私有化；

## 2. C++ 中命名空间的定义

说明:

6. 一般在命名空间中声明函数，而在命名空间之外定义函数；

7. 命名空间可以嵌套，例如：

```
namespace AA
{
    namespace BB
    {
        int x=2;
    }
}

int main()
{
    cout<<
    AA::BB::x<<endl;
    return 0;
}
```

## 2. C++ 中命名空间的定义

可以定义未命名的命名空间，每个文件可以有自己的未命名的命名空间，但未命名的命名空间是不能够跨越多文件。例如：

```
namespace  
{  
    int x=2;  
}
```

此程序定义一个未命名的命名空间，未命名的命名空间的成员可直接使用。用于声明局部于文件的实体，相当于全局变量。

### 3. C++ 中命名空间的使用

C++ 中，使用命名空间的标识符时，可以有三种访问方法：

- ① 使用 `using namespace`
- ② 直接指定标识符
- ③ 使用 `using` 关键词

## 3. C++ 中命名空间的使用

### ① 使用 using namespace

使用 using namespace 可以释放命名空间中的所有名字，如变量名或对象名等。命名空间成员的访问方式为：

```
namespace num
{
    int x=20;
    int y=10;
}

int main()
{
    using namespace num;
    cout<<"x:"<<x<<" y:"<<y<<endl;
    return 0;
}
```

## 3. C++ 中命名空间的使用

【例 2-3】命名空间的使用。

```
#include <iostream>
using namespace std;
namespace A{
    int x;
    void f() {cout<<"namespace A::f() "<<endl;}
    void g() {cout<<"namespace A::g() "<<endl;}
}
namespace B{
    int x;
    void f() {cout<<"namespace B::f() "<<endl;}
    void t() {cout<<"namespace B::t() "<<endl;}
}
```

## 3. C++ 中命名空间的使用

```
int main()
{   using namespace A;
    using namespace B;
    A::x=4; // 不能写成 x=4
    A::f(); // 不能写成 f();
    B::f();
    g();
    t();
    return 0;
}
```



### 3. C++ 中命名空间的使用

#### ② 直接指定标识符

假如不想将它们全部从命名空间中释放出来，而是只使用该空间中某个特定的变量，可以使用命名空间加作用域解析符 `::`，然后指定该变量名。

命名空间成员的访问方式为：

命名空间标识符名`::`成员名

## 3. C++ 中命名空间的使用

```
namespace A
{
    int x=2;
}
namespace B
{
    int x=5;
    void Print();
}
```

```
int main()
{
    int x=8;
    cout<<x<<endl;
    cout<<A::x<<endl;
    cout<<B::x<<endl;
    return 0;
}
```

## 3. C++ 中命名空间的使用

### ③ 使用 using 关键词

例如：

```
int main()
{
    using B::x;
    cout<<x<<endl;
    using B::Print;
    Print();
    return 0;
}
```

但是在编程中使用这三种命名空间的方法时，要注意当两个以上的名字空间有相同的标识符时，使用不当时容易出错，最好使用第（2）种方法。

### 4. std 命名空间

C++ 经过一个较长的发展和标准化的过程，形成了两个版本的 C++：

传统的 C++，

标准的 C++，

这两个版本的核心内容基本形同，但标准 C++ 增加了传统 C++ 中没有的一些特征。

两种版本的 C++ 有大量相同的库和函数，其区分方法是头文件和命名空间。

传统的 C++ 采用与 C 语言相同风格的头文件，扩展名有（如 .h，.hpp，.hxx 等）；

标准的 C++ 头文件没有扩展名。

### 4. std 命名空间

例如:

传统 C++ 的头文件写成:

```
#include <iostream.h>
```

```
#include <string.h>
```

标准 C++ 对应头文件为:

```
#include <iostream>
```

```
#include <string>
```

### 4. std 命名空间

#### 函数库

标准 C++ 包含了所有 C 函数库，支持在 C++ 中引用 C 函数库。但标准 C++ 也提供了与之对应的新式函数库，标准 C++ 中与 C 的函数库对应的头文件的命名方式是：在原来 C 函数库头文件名的前面加上 “c” 前缀，并去掉 .h，例如：

C 语言的头文件为：

```
#include <stdlib.h>、 #include <math.h>
```

标准 C++ 头文件为：

```
#include <cstdlib>、 #include <cmath>
```

### 4. std 命名空间

#### std 命名空间

- 标准 C++ 将新格式头文件中的内容全部放到了 std 命名空间中。
- 如果程序中要引用标准 C++ 新格式头文件中的函数，就需要在程序中使用 `using namespace std;` 语句将 std 命名空间中的标识符引入到全局命名空间。
- 虽然 C++ 编译器提供了对新老格式头文件的的同时支持，但标准的 C++ 具有更多的新特性和功能，在程序设计中建议使用新标准 C++ 。

## 4. std 命名空间

### 【例 2-3】标准 C++ 的简单程序设计

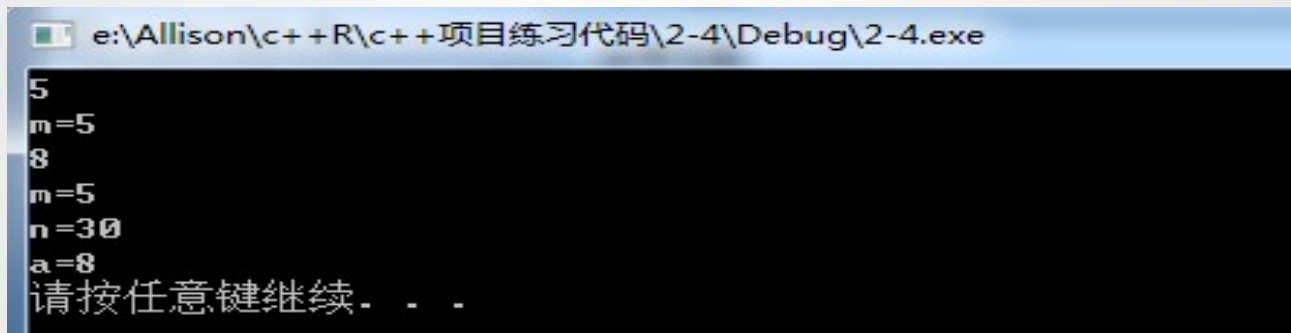
```
#include <iostream>
using namespace std;
#include <cstdio>
#include <cmath>
int main()
{
    int m, a;
    int n=abs(-30);
    // 调用 cmath 库中的 abs 绝对值函数
```

```
scanf("%d", &m);
printf("m=%d\n", m); // 调用 scanf
和 printf 来源于 cstdio 库
    cin>>a;
    cout<<"m="<<m<<endl;
    cout<<"n="<<n<<endl;
    cout<<"a="<<a<<endl;
    // 调用 cin、cout 来源于 iostream
    return 0;
}
```



### 4. std 命名空间

【例 2-3】标准 C++ 的简单程序设计



```
e:\Allison\c++R\c++项目练习代码\2-4\Debug\2-4.exe
5
m=5
8
m=5
n=30
a=8
请按任意键继续. . .
```

# 03



## C++ 数据的输入输出

### 简介

数据的输入 / 输出是一个比较重要的操作

C++ 的输入 / 输出由 `iostream` 库提供支持。它利用多继承和虚拟继承实现了面向对象类层次结构。

C++ 的输入 / 输出机制包括：

（ 1 ） 内置数据的输入 / 输出 ； （ 2 ） 文件的输入 / 输出。

### C++ 的输入输出数据流

在 C++ 中，I/O（input/output，输入 / 输出）数据是一系列从源设备到目的设备的字节序列，称为**字节流**。

有两种类型的数据流：

**输入数据流：**从输入设备到计算机的序列字符。

**输出数据流：**从计算机到输出设备的序列字符。

### C++ 的输入输出数据流

在 C++ 中，标准的输入设备通常是指键盘，标准的输出设备是指显示器。

为了从键盘中输入数据，或为了将数据输出到显示器上，程序中必须包含头文件 `iostream`。

`iostream` 文件包含：输入流 `istream` 和输出流 `ostream` 两种数据类型。

这两种数据类型定义了如下变量：

```
istream cin;
```

```
ostream cout;
```

其中，`cin` 用于从键盘中输入数据；`cout` 用于将内存数据输出到显示器。

### cin 和析取运算符 >>

在 C++ 程序中，常用 cin 从键盘中输入数据，其输入格式如下：

```
cin>> 变量名 ;
```

当程序执行到 cin 语句时，就会停下来等待键盘数据的输入，数据输入被插入到输入流中，数据输入完后按 Enter 键结束。

例如：

```
int x;
```

```
double y;
```

```
char z;
```

```
cin>>x>>y>>z;
```

### cin 和析取运算符 >>

说明:

- (1) 使用 cin 从键盘中输入数据，原则上是系统内置的简单数据类型，如 int、double、char、float 等。
- (2) 在输入数据时，如果有多个数据，各个数据之间用空格（回车或 Tab）分隔，输入 Enter 键结束。
- (3) 在析取运算符 >> 后面只能出现变量名，这些变量应该是系统预定义的简单类型，否则将出现错误。如下面的语句是错误的。

```
cin>>8>>x;    // 错误，>> 后面有常数 8
```

```
cin>>' a' >>x;  // 错误，>> 后面有字符 ' a'
```

- (4) cin 具有自动识别数据类型的能力，析取运算符 >> 将根据它后面的变量类型从输入流中为它们提取对应的数据。

### cout 和插入运算符 <<

在 C++ 程序中，使用 cout 输出数据流可以在屏幕上显示字符和数字等数据，其输出格式如下：

`cout<< 变量名或常量 ;`

例如：

```
#include <iostream>
int x=10;
double y=20.5 ;
cout<<" x=" <<x<< " " <<" y=" <<y<<endl;
```



### cout 和插入运算符 <<

说明:

(1) 使用 `cout` 从显示器上输出数据, 数据可以是系统预定义的简单数据类型, 也可以是用户自定义的数据类型, 如对象等。

(2) 当输出多个数据时, 可以使用 `cout` 进行连续输出。输出数据即可以是变量也可以是常量。

(3) `cout` 输出语句中, 如果有带双引号的字符串, 将字符串原样输出。

(4) 在 `cout` 输出语句中, 还可以设置数据输出控制符, 如字宽、左对齐、右对齐等格式, 详细请查看本书第八章文件和流。

### C++ 字符串变量

除了可以使用字符数组处理字符串外，C++ 还提供了一种更方便的方法——用字符串类型（`string` 类型）定义字符串变量。

对于 C 与 C++ 来说是没有字符串型的数据类型的，它是在 C++[std 命名空间](#)中的标准库定义了一个 `string` 字符串类，用这个类定义字符串变量（对象）。

### 定义字符串变量

和其他类型变量一样，字符串变量必须先定义后使用，字符串变量要用类名 `string`。如：

```
string string1;
```

```
string string2= "china" ;
```

**注意：**要使用 `string` 类功能时，必须在本文件的开头将 `C++` 标准库中的 “`string`” 头文件包含进来，即：

```
#include<string>// 注意不是头文件名 <string.h>
```

### 对字符串变量赋值

```
string str1 = "hello world!"
```

```
string str2 = "thank you!"
```

```
string str[] = {"zxw", "qwe"};
```

### 字符串变量的输入和输出

```
cin>>str1;  
cout<< str2;
```

### 对字符串变量赋值

( 1 ) 用赋值运算符实现字符串赋值

```
str1 = str2;    // 不要求长度相同
```

( 2 ) 用加法实现字符串连接

```
string str1 = "hello world!" ;  
string str2 = "thank you!";  
str1= str1+ str2;
```

( 3 ) 用关系运算符实现字符串的比较

可以直接使用 == , < , > 等。

# 04



## 引用

- ◆ 引用是一个对象（即变量）的别名。
- ◆ 在 C 语言中没有引用这个概念，它是 C++ 引入的新概念。
- ◆ 引用由符号 & 来定义，格式如下：

类型 & 引用名 = 变量名

例如：

```
int x=5;
```

```
int &ix=x;
```

### 【例 2-5】引用的简单实例

```
#include <iostream>
using namespace std;
int main()
{   int m;
    int &n=m;      // 变量 n 为 m 的引用别名
    m=30;
    cout<<"m="<<m<<"n="<<n<<"\n";
    n=80;
    cout<<"m="<<m<<"n="<<n<<"\n";
    cout<<"m 地址是 : "<<&m<<endl;
    cout<<"n 地址是 : "<<&n<<endl;
    return 0;
}
```

## 【例 2-5】引用的简单实例



```
C:\ D:\C++编写\第二章\例2-4\Debug\2.1.exe
m=30n=30
m=80n=80
m地址是:0012FF60
n地址是:0012FF60

搜狗拼音 半:
```



## 说明:

( 1 ) 在变量声明时出现的 & 才是引用运算符，其他地方出现的 & 都是地址运算符。例如：

```
int m;  
  
int &n=m;           // 引用运算符  
  
cout<<"m 地址是 : "<<&m<<endl;    // 地址运算符
```

( 2 ) 引用是变量的别名，必须在定义时进行初始化，不能在定义完后再赋值，下面的定义是错误的。

```
int m;  
  
int &n;    // 错误，定义为引用，但没有初始化  
  
m=n;
```

### 说明:

( 3 ) 可以为一个变量指定多个引用，为引用提供的初始值，可以是一个变量，也可以是另一个引用名。

例如:

```
int m;  
int &n=m;  
int &i=m;  
int &j=i;
```

### 说明:

( 4 ) 一个引用名只能是一个变量的别名，不能再次将它指定为其他变量的别名。例如：

```
int m, a;
```

```
int &n=m;
```

```
n=&a;    // 错误，一个引用为 2 个变量的别名
```

( 5 ) 建立引用时，需要注意以下 3 个限制：

- a. 不能建立引用的引用。
- b. 不能建立引用数组，也不能建立数组的引用。
- c. 可以建立指针的引用，但不能创建指向引用的指针。

例如:

```
int a, b[8];  
int &&aa=a;           // 错误, aa 是引用的引用  
int &ib[6];           // 错误, ib 是引用数组  
int &bb=b;            // 错误, bb 是数组的引用  
int &*ap=a;           // 错误, ap 是指向引用的指针  
int *pi=&a;  
int *&pr=pi;          // 正确, pr 是指针的引用
```

在 C++ 中, 引用主要用于定义函数参数和返回值类型。因为引用只须传递一个对象的地址, 在传递大型对象的函数参数或从函数返回大型对象时, 可以提高效率。

05



# 函数

函数是 C 和 C++ 程序的基本构件，在 C++ 中，定义函数的方法和规则与 C 语言基本相同。

C++ 中关于函数增加了新的内容，如：

函数原型

带有默认参数的函数

内联函数

重载函数

函数的参数是引用以及返回值为引用

## 1. 函数原型

- ◆ C 语言中没有强调必须使用函数原型，但在 C++ 中要求定义函数原型。
- ◆ C++ 是一种强制类型检查语言，每个函数的实参在编译期间都要经过类型检查。
- ◆ 如果实参类型与对应的形参类型不匹配，C++ 就会尝试可能的类型转换，若转换失败，或实参个数与函数的参数个数不相符，就会产生一个编译错误。要实现这样的检查，就要求所有的函数必须在调用之前进行声明或定义。
- ◆ 为了能使函数在定义之前就能被调用，C++ 规定可以先说明函数原型，然后就可以调用函数，函数定义可放在程序后面。

### 函数原型声明格式

- ◆ 函数原型类似函数定义时的函数头，又称函数声明，只有一条语句，由函数返回类型、函数名和形式参数表三部分组成。
- ◆ 函数原型声明格式为：

```
返回类型 函数名 (数据类型 参数名, 数据类型 参数名 ....);
```



## 【例 2-6】返回两个数相加的结果

```
#include <iostream>
using namespace std;
int add(int x, int y);    // 函数原型的声明
int main()
{
    int a=10, b=20;
    add(a, b);
    return 0;
}
int add(int x, int y)
{
    return x+y;
}
```

(1) 参数表包含所有参数的数据类型，参数之间用逗号分开。在 C++ 中，函数声明就是函数原型。

(2) 函数原型和函数定义在返回类型、函数名和参数表上必须完全一致。如果它们不一致，就会发生编译错误。

(3) 函数原型不必包含参数的名字，而只要包含参数的类型。下面的函数原型声明是合法的。

```
int add(int ,int );
```

等价于: `int add(int x,int y);`

(4) 如果函数的定义出现在程序中第一次调用此函数之前，就不需要函数原型。

(5) 由于函数原型是一条语句，因此函数原型必须以分号结束。

(6) C++ 与 C 语言的函数参数声明存在区别。C 语言可以将参数的类型说明放在函数头和函数体之间，C++ 不支持这种传统的函数声明方式。

## 2. 重载函数

- 在 C 语言中，函数名必须唯一，不允许同名的两个函数出现在同一程序中。如果要对不同类型的数据进行相同的操作，必须编写不同名字的函数。例如，要打印三种不同类型的数据：整型、字符型和实型，则必须用三个不同的函数名，例如：
  - `Print_int()`、
  - `Print_char()`、
  - `Print_float()`。
- C++ 提供了函数重载功能。函数重载是指两个或两个以上的函数具有相同的函数名，但参数类型不一致或参数个数不同。
- 编译时编译器将根据实参和形参的类型及个数进行相应地匹配，自动确定调用哪一个函数。使得重载的函数虽然函数名相同，但功能却不完全相同。
- 函数重载，方便使用，便于记忆。

【例 2-7】求两个或三个整数的和，求两个或三个双精度浮点数的和

```
#include <iostream>
using namespace std;

int add(int x, int y);
int add(int x, int y, int z);
double add(double x, double y);
double add(double x, double y, double z);
```

【例 2-7】求两个或三个整数的和，求两个或三个双精度浮点数的和

```
int main()
{
    int a=2, b=3, c=4, i, j;
    double d=1.1, e=2.2, f=3.3, m, n;
    i=add(a, b);
    cout<<"i="<<i<<endl;
    j=add(a, b, c);
    cout<<"j="<<j<<endl;
    m=add(d, e);
    cout<<"m="<<m<<endl;
    n=add(d, e, f);
    cout<<"n="<<n<<endl;
    return 0;
}
```

【例 2-7】求两个或三个整数的和，求两个或三个双精度浮点数的和

```
int add(int x, int y)
{   return x+y;
}

double add(double x, double y)
{   return x+y;
}

int add(int x, int y, int z)
{   return x+y+z;
}

double add(double x, double y, double z)
{   return x+y+z;
}
```

## 说明

(1) 重载函数必须具有不同的参数个数或不同的参数类型，若只是返回值的类型不同或形参名不同是错误的。例如：

```
float add (int x, int y);
```

```
int add (int x, int y);           // 错误，编译器不以返回值来区分函数
```

再如：

```
int add(int x, int y);
```

```
int add(int a, int b);           // 错误，编译器不以形参名来区分函数
```

重载函数应满足：函数名相同，函数的返回值类型可以相同也可以不同，但**参数表必须不同**。

即：各函数的**参数表中的参数个数或类型必须有所不同**。

这样才能进行区分，从而正确地调用函数。

(2) 匹配重载函数的顺序：首先寻找一个精确匹配，如果能找到，调用该函数；其次进行提升匹配，通过内部类型转换（窄类型到宽类型的转换）寻求一个匹配，如 char 到 int、short 到 int 等，如果能找到，调用该函数；最后通过强制类型转换寻求一个匹配，如 int 到 double 等，如果能找到，调用该函数。

(3) 不要将不同功能的函数定义为重载函数，以免产生误解。例如：

```
int f(int a, int b)
{
    return a+b;
}
```

```
double f(double a, double b)
{
    return a*b;
}
```



## 说明

(4) 在定义和调用重载函数时，要注意二义性。例如

```
int f(int &x) {...};
```

```
int f(int x) {...};
```

这两个函数属于重载函数，但当调用时出现下面的情况，编译器就不知道调用哪一个函数，会出现二义性。

```
int a=4;
```

```
f(a);    // 错误，编译器无法确定是调用函数 f(int& x) ，还是调用函数 f(int x) ，产生二义性。
```

## 说明

下面的函数重载同样会产生二义性:

```
int f(unsigned int x) {return x};
```

```
double f(double x) {return x};
```

当调用时如果出现如下情况也会产生二义性:

```
int a=4;
```

```
f(a); // 错误, 产生二义性
```

同时, 当函数重载带有默认参数时, 也容易产生二义性.

### 3. 带有默认参数的函数

- C++ 中允许函数提供默认参数，也就是允许在函数的声明或定义时给一个或多个参数指定默认值。
- 在调用具有默认参数的函数时，如果没有提供实际参数，C++ 将自动把默认参数作为相应参数的值。

### 【例 2-8】求两个整数的和

```
#include <iostream>
using namespace std;
int add(int x=7, int y=2);
int main()
{
    int a=4, b=6, c;
    c=add(a, b);
    cout<<"c="<<c<<endl;
    c=add(a);
    cout<<"c="<<c<<endl;
    c=add();
    cout<<"c="<<c<<endl;
    return 0;
}
```

```
int add(int x, int y)
{
    return x+y;
}
```

(1) 当函数既有原型声明又有定义时，默认参数只能在原型声明中指定，而不能在函数定义中指定。如果一个函数的定义先于其调用，没有函数原型，若要指定参数默认值，需要在定义时指定。

例如：

```
int add(int x=7, int y=2) {return x+y;} // 函数调用前定义，可以指定默认参数。

int main()
{
    int z;
    z=add(5, 6);
}
```

(2) 在函数原型中，所有取默认值的参数都必须出现在不取默认值的参数的右边。也就是一旦某个参数开始指定默认值，其右面的所有参数都必须指定默认值，**遵循从右至左的规则**。

例如：

```
int add(int i, int j=5, int k);
```

```
// 错误，在取默认参数的 int j=5 后，不能再说明非默认参数 int k
```

应改为：

```
int add(int i, int k, int j=5);
```

或

```
int add(int i, int k=5, int j=8);
```

(3) 在调用具有默认参数值的函数时，若某个实参默认而省略，则其右面的所有实参皆应省略而采用默认值。不允许某个参数省略后，再给其右面的参数指定参数值，**遵循从左至右的规则**。例如：

```
int add(int x=7, int y=2, int z=11);
```

在主函数中，针对此函数有如下调用：

```
add( );           // 正确, x=7, y=2, z=11
```

```
add(3);           // 正确, x=3, y=2, z=11
```

```
add(5, 6);        // 正确, x=5, y=6, z=11
```

```
add(5, 6, 5);     // 正确, x=5, y=6, z=5
```

```
add( , 8, 4);     // 错误, x 默认了, 而右面的 y、z 没有默认。
```

( 4 ) 当函数的重载带有默认参数时，要注意避免二义性。例如：定义如下两个重载函数：

```
double add(double x, double y=2.2);
```

```
double add(double x);
```

这是错误的，因为如果有调用函数 `add(2.5)` 时，编译器将无法确定调用哪一个函数。

( 5 ) 函数的带默认参数值的功能可以在一定程度上简化程序的编写。



## 4. 内联函数

- 函数使用有利于代码重用，提高开发效率，增强程序的可靠性，便于分工合作，便于修改维护。
- 函数的调用会降低程序的执行效率，需要保存和恢复现场和地址。需要时间和空间的开销。为解决这一问题，C++ 中对于功能简单、规模小、使用频繁的函数，可以将其设置为内联函数。
- 内联函数 (inline function) 的定义和调用和普通函数相同，但 C++ 对它们的处理方式不一样。如果一个函数被定义为内联函数，在编译时，C++ 将用内联函数代码替换对它每次的调用。

## 4. 内联函数

内联函数声明或定义时，将 `inline` 关键字加在函数的返回类型前面就可以将函数定义为内联函数。

格式如下：

```
inline 返回值类型  函数名（形式参数表）  
{  
    .....    // 函数体  
}
```

## 【例 2-9】求两个数的最大值

```
#include <iostream>
using namespace std;
inline int max(int x,int y)
{
    return x>y?x:y;
}
int main()
{
    int z1=max(9, 34);
    int z2=max(4, 55);
    int z3=max(z1, z2);
    return 0;
}
```

## 【例 2-9】求两个数的最大值

上面的程序中，main() 函数三次调用了内联函数 max()，C++ 编译此程序时会将 main() 函数调用函数替换成如下形式：

```
int main()
{
    int z1=9>34?9:34;
    int z2=4>55?4:55;
    int z3=z1>z2?z1:z2;
    return 0;
}
```

- ◆ **优点：节约时间。**内联函数没有函数调用的开销，即节省参数传递、控制转移的开销，从而提高了程序运行时的效率。
- ◆ **缺点：增大空间。**由于每次调用内联函数时，只是将这个内联函数的所有代码复制到调用函数中，所以会增加程序的代码量，占用更多的存储空间，增大了系统空间方面的开销。
- ◆ 因此，内联函数是一种以空间换时间的方案。

- （ 1 ）内联函数体内不能有循环语句和 switch 语句。递归调用的函数不能定义为内联函数。
- （ 2 ）内联函数的声明必须出现在内联函数第一次被调用之前。
- （ 3 ）内联函数代码不宜太长，一般是 1 ~ 5 行代码的小函数，调用频繁的简单函数可以定义为内联函数。
- （ 4 ）在类内定义的成员函数被默认为内联函数。

## 5. 引用参数和返回引

用

引用只须传递一个对象的地址，可以提高函数的调用和运行效率效率。

C++ 中，引入引用主要用于定义函数参数和返回值类型。

### (1) 引用参数

- 在 C 语言中，函数中参数传递的方式有两种：值传递和地址传递。
- 值传递是单向传递，形参值的变化不影响实参。
- 地址传递是双向传递，形参值的变化影响实参。
- 如果使用引用作为函数的参数，**形参是实参的别名**，在形参与实参结合的过程中，引用参数传递的是实参的地址，因此，这也是一种地址传递，能够达到与指针同样的效果，但它的使用形式比指针参数简单。

## 【例 2-10】使用引用参数完成两个数值的交换

```
#include <iostream>
using namespace std;
void swap(int &x, int &y);
int main()
{
    int a=2;
    int b=9;
    cout<<" 交换前 a 和 b 的值为 : "<<"a="<<a<<"b="<<b<<endl;
    swap(a, b);
    cout<<" 交换后 a 和 b 的值为  a: "<<"a="<<a<<"b="<<b<<endl;
    return 0;
}
```

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



使用引用参数，一般在下面的几种情况下使用：

- （ 1 ） 需要从函数中返回多于一个值；
- （ 2 ） 修改实参值本身；
- （ 3 ） 传递地址没有传值和生成副本的空间和时间消耗。提高函数调用和运行效率。

### (2) 返回引用

C++ 中，函数除了能够返回值或指针外，也可以返回一个引用。

返回引用的函数定义格式如下：

返回值类型 & 函数名（形参表）

当一个函数返回引用时，实际是返回了一个变量的地址，这使函数调用能够出现在赋值语句的左边。

## 【例 2-11】返回引用

```
#include <iostream>
using namespace std;
```

```
int z;
int& add(int x, int y);
```

```
int main()
{
    int a=add(5, 7);
    cout<<a<<endl;
    add(4, 9)++;
    cout<<z<<endl;
```

```
    add(2, 8)=8;
    cout<<z<<endl;
    return 0;
}
int& add(int x, int y)
{
    return z=x+y;
}
```

注意，当函数返回一个引用时，`return` 语句只能返回一个变量，而不能返回一个表达式，但函数是返回值时可以。例如：

```
int add(int x, int y)
{
    return x+y;    // 正确，当函数返回一个值时，可以使用表达式
}

int& add(int x, int y)
{
    return x+y;    // 错误，当函数返回一个引用时，不可以使用表达式
}
```

# 06



## 变量的的作用域与可见性

- 作用域讨论的是标识符的有效范围。
- 可见性是讨论标识符是否可以被使用。
- 作用域与可见性二者既相互联系又存在差异。

### 1. 作用域

作用域是一个标识符在程序正文中有效的区域。

C++ 中标识符的作用域有：

- 函数原型作用域
- 块作用域（局部作用域）
- 类作用域
- 文件作用域
- 命名空间作用域

### 1.1 函数原型作用域

- 在函数原型声明时形式参数的作用范围就是函数原型作用域。
- 例如
  - `void fun(int x);` // 变量 `x` 具有函数原型作用域
  - `fun()` 函数中形参 `x` 有效的范围就在左、右两个括号之间，出了这两个括号，在程序的其他地方都无法引用 `x`。
- 函数原型作用域是 C++ 程序中最小的作用域。



### 1.1 函数原型作用域

函数原型形参的定义：

- 函数原型如果有形参，声明时：
- **数据类型**：必须要定义
- **形参名**：可以省略（比如 x）
- 形参名的省略不会对程序有任何影响。
- 一般为了程序可读性，可以写一个容易理解的形参名。

### 1.2 块作用域

所谓块，就是一对大括号括起来的一段程序。

在块中声明的标识符，其作用域从声明处开始，一直到块结束的大括号为止。

例如，块作用域：

```
void fun(int x)
{
    int a;                // a 的作用域开始
    cin>>a;
    {
        int b=2;          // b 的作用域开始
        .....
    }                     // b 的作用域结束
}                          // a 的作用域结束
```

### 1.3 类作用域

- 类的作用域简称**类域**；
- 它是指在类的定义中由一对花括号所括起来的部分。
- 每一个类都具有该类的类域，该类的所有成员属于该类的类作用域中。
- 由类的定义中可知，类成员包括两部分：**数据变量成员和成员函数**。由于类中成员的特殊访问规则，使得类中成员的作用域变得比较复杂。

### 1.3 类作用域

具体地讲，某个类 A 中某个成员 M 在下面情况下具有类 A 的作用域；

- (1) 该成员 (M) 出现在该类的某个成员函数中，并且该成员函数没有定义同名标识符。
- (2) 该类 (A) 的某个对象的该成员 (M) 的表达式中。例如，a 是 A 的对象，即在表达式 a.M 中。
- (3) 在该类 (A) 的某个指向对象指针的该成员 (M) 的表达式中。例如，Pa 是一个指向 A 类对象的指针，即在表达式 Pa->M 中。
- (4) 在使用作用域运算符所限定的该成员中。例如，在表达式 A::M 中。

### 1.4 文件作用域

如果一个标识符没有在前三种作用域中出现，则它具有文件作用域。

这种标识符的作用域从声明处开始，到文件结尾处结束。

具有文件作用域的变量也称为全局变量。

一般说来，文件域中可以包含类域，类域中可包含成员函数的作用域。因此，类域介于文件域和函数域之间，由于类域问题比较复杂，在前面和后面的程序中都会遇到，只能根据具体问题具体分析。

### 1.4 文件作用域

#### 【例 2-12】块作用域和文件作用域

```
#include<iostream>
using namespace std;
int x;                                // 变量 x 具有文件作用域
int main()
{
    x=4;                               // 给 x 赋初值
    {                                  // 子块
        int x; // 在子块中, 定义一个具有块作用域的变量 x
        x=2;
        cout<< "x=" <<x<<endl;      // 输出 2
    }
    cout<< "x=" <<x;                  // 输出 4
    return 0;
}
```

### 1.5 命名空间作用域

- 一个命名空间确定了一个命名空间作用域。
- 凡是在该命名空间之内声明的标识符，都属于该命名空间作用域。
- 在命名空间内部可以直接引用当前命名空间中声明的标识符，否则需要在命名空间之外访问命名空间的标识符，需要使用下面的语法：
- 命名空间名称：：标识符

### 1.5 命名空间作用域

例如:

```
namespace num
{
    int x=5;
    void fun();
}
```

命名空间 num 中的变量 x 和函数 fun() 具有命名空间作用域，如果要引用它们，需要使用下面的方式:

num::x 或 num::fun()



### 1.5 命名空间作用域

有时，在标识符前面使用命名空间限定会显得过于冗长，为了解决这一问题，C++ 又提供了 `using namespace` 命令和 `using` 声明两种形式：

```
using 命名空间名 :: 标识符 ;
```

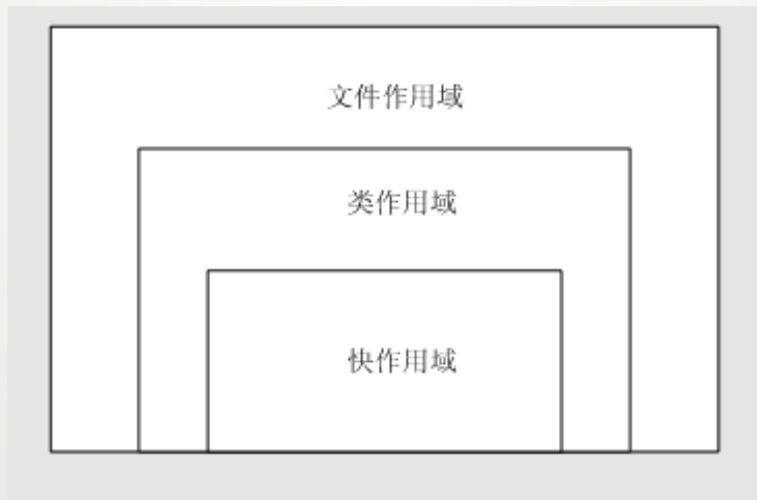
```
using namespace 命名空间名 ;
```

前一种形式将指定的标识符释放在当前的作用域内，使得在当前作用域中可以直接引用该标识符；后一种形式将指定命名空间的所有标识符释放在当前的作用域内，使得在当前作用域中可以直接引用该命名空间内的任何标识符。

### 2. 可见性

标识符的可见性是指在程序的某个地方是否是有效的，是否能够被使用被访问。程序运行到某一处时，能够访问的标识符就是在此处可见的标识符。

上面的四种作用域中，最大的是文件作用域，其次是类作用域，再次是块作用域。它们的包含关系为：



### 2. 可见性

作用域可见性的一般规则是：

- （1）标识符要声明在前，引用在后。
- （2）在同一作用域中，不能声明同名的标识符。
- （3）在没有相互包含关系的不同的作用域中声明的同名标识符，互不影响。
- （4）如果在两个或多个具有包含关系的作用域中声明了同名标识符，则外层标识符在内层不可见。

关于作用域与可见性的规律既适用于简单变量，也适用于自定义数据类型和类的对象。

07



## 对象的生存期

- 所谓对象的生存期是指对象从被创建开始到被释放为止的时间。
- 不同存储的对象生存期不同，在对象生存期内，对象将保持它的值，直到被更新为止。
- 对象生存期有：
  - 静态生存期
  - 动态生存期

### 1. 静态生存

期

- 如果对象的生存期与程序的运行期相同，则称它具有**静态生存期**。
- 静态生存期只要程序开始运行，这种生存期的变量就被分配了内存。
- 在文件作用域中声明的对象具有这种生存期。

## 1. 静态生存

期

【例 2-12】静态生存期

```
#include<iostream>
using namespace std;
int i=5;          // 文件作用域，变量 i 具有静态生存期
int main()
{
    cout<<"i="<<i<<endl;
    return 0;
}
```

### 1. 静态生存

期

- ◆ 如果要在函数内部局部作用域中声明具有静态生存期的对象，则使用关键字 `static`。
- ◆ 例如：下列定义的变量 `i` 便是具有静态生存期的变量，也称为静态变量：

```
int main()
{
    static int i; // 块作用域，变量 i 具有静态生存期
    cout<<"i="<<i<<endl;
    return 0;
}
```



### 2. 动态生存

期

- ◆在块作用域中声明的，没有用 `static` 修饰的对象具有动态生存期（称局部生存期）。
- ◆动态生存期开始于程序执行到声明点时，结束于命名该标识符的作用域结束处。
- ◆这种变量可以随时创建，随时删除。
- ◆创建和删除是程序员用内存操作函数进行的。

### 2. 动态生存

期

【例 2-14】静态生存期和动态生存期

```
#include <iostream>
using namespace std;
int i=1;
void fun()
{
    static int a=2;
    static int b=0;    //a、b 为静态局部变量，具有静态生存期
    int c=10;         //c 为局部变量，具有动态生存期
    a=a+2;
```

### 2. 动态生存

期

【例 2-14】静态生存期和动态生存期

```
i=i+32;  
c=c+5;  
cout<<"fun() 函数:  \n";  
cout<<"i="<<i<<"a="<<a<<"b="<<b<<"c="<<c<<endl;  
b=a;  
}
```

### 2. 动态生存

期

【例 2-14】静态生存期和动态生存期

```
int main()
{
    static int a=0;  //a 为静态局部变量，具有静态生存期
    int b=-10;
    int c=0;  //b、c 为局部变量，具有动态生存期
    cout<<"main() 函数:  \n";
    cout<<"i="<<i<<"a="<<a<<"b="<<b<<"c="<<c<<endl;
```

### 2. 动态生存

期

【例 2-14】静态生存期和动态生存期

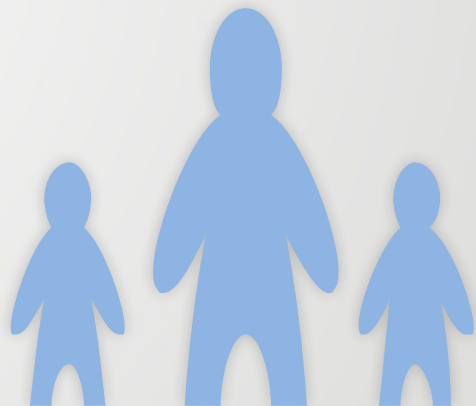
```
c=c+8;  
fun();  
cout<<"main() 函数:  \n";  
cout<<"i="<<i<<"a="<<a<<"b="<<b<<"c="<<c<<endl;  
i=i+10;  
fun();  
return 0;  
}
```

08



const 常量

- ◆ 常量是一种标识符，它的值在运行期间恒定不变。
- ◆ C 语言用 `#define` 来定义常量（称为宏常量）。
- ◆ C++ 语言除了用 `#define` 定义常量外，还可以用 `const` 来定义常量（称为 `const` 常量）。



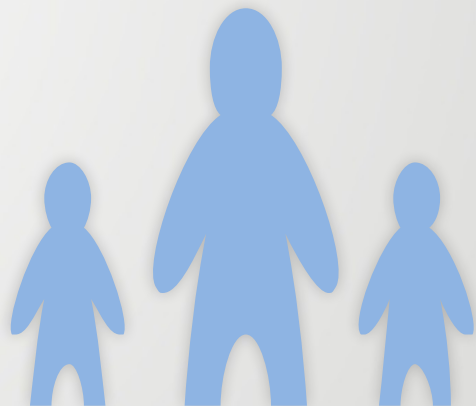
## 1. 常量的定义

在 C++ 中，常用 `const` 修饰符来定义常量，定义常量的方法如下：

`const` 常量类型 常量名 = 常量值

例如：

```
const int i=10;           // 定义整型常量  
const char c=' A' ;      // 定义字符常量  
const char a[]=" C++const! " ; // 定义字符串常量数组
```





## 1. 常量的定

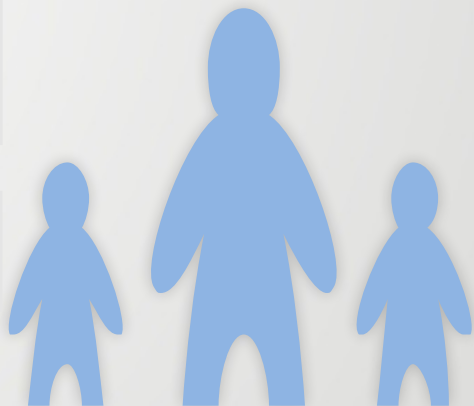
义

(1) 常量一经定义就不能修改，常量名不能出现在赋值符“=”的左边，例如：

```
const int i=6;           // 定义常量 i
i =34;                   // 错误，修改常
    量
i++;                     // 错误，修改常
    量
```

(2) const 常量必须在定义时初始化。 例如：

```
const int i;             // 错误，常量 i 未被初始
    化
```



## 1. 常量的定

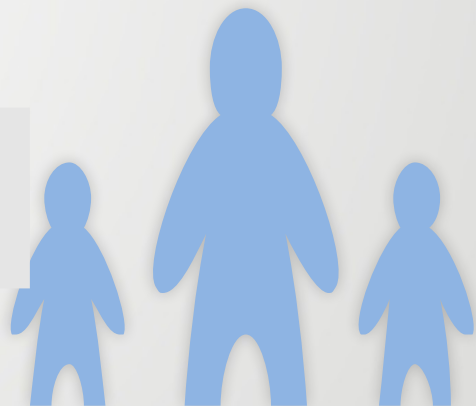
义

(3) 在 C++ 中，表达式可以出现在常量定义语句中。例如：

```
int  a=4, b;  
const int b=a+55;
```

(4) 在另一连接文件中引用 const 常量

```
extern const int i;           // 合法  
extern const int i=10;       // 非法，常量不可以被再次赋值
```



## 1. 常量的定

### 常量定义规则:

- (1) 需要对外公开的常量放在头文件中, 不需要对外公开的常量放在定义文件的头部。为便于管理, 可以把不同模块的常量集中存放在一个公共的头文件中。
- (2) 如果某一常量与其它常量密切相关, 应在定义中包含这种关系, 而不应给出一些孤立的值。例如:

```
const float    RADIUS = 100;  
const float    DIAMETER = RADIUS * 2;
```

const 可以与指针、函数的参数和返回值、类的数据成员和成员函数等结合起来, 定义常量指针, 函数的参数和返回值为常量以及常对象, 常数据成员、常成员函数等。

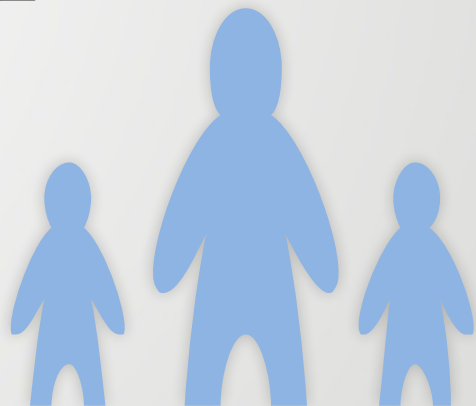


## 2. const 和 #define

在 C++ 中，既可使用 `const` 定义常量，也可以使用 `#define` 定义常量。

例如：

```
#define    MAX    100        /* C 语言的宏常量 */  
const int  MAX = 100; // C++ 语言的 const 常量  
const float PI = 3.14159; // C++ 语言的 const 常量
```



## 2. const 和 #define

但是，`#define` 是 C 语言中用来定义宏常量，`const` 定义常量比 `#define` 定义常量有更多的优点。

（1）`const` 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。

（2）有些集成化的调试工具可以对 `const` 常量进行调试，但是不能对宏常量进行调试。因此，建议在 C++ 程序中用 `const` 取代 `#define` 定义常量。



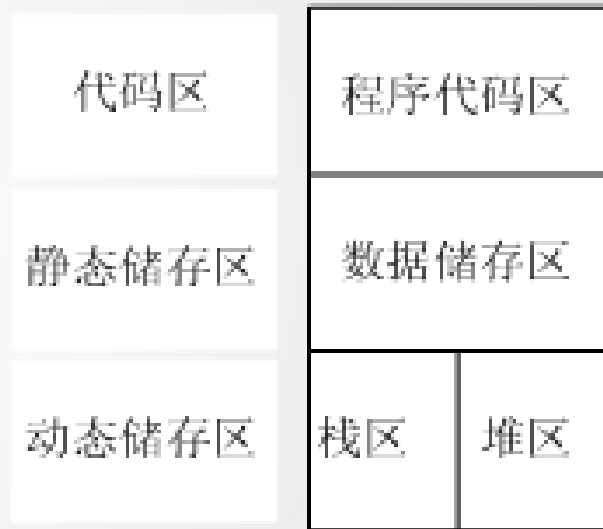
09



# 动态内存分配和释放

程序数据存储所占内存一般分为三部分：

- ① 程序代码区
- ② 静态存储区（数据区）
- ③ 动态存储区（栈区和堆区）



- ① 代码区存放程序代码，程序运行前就分配存储空间。
- ② 数据区存放常量、静态变量、全局变量等。
- ③ 动态存储区分为栈区和堆区。
  - 栈区由编译器自动分配并且释放，用来存放局部变量、函数参数、函数返回值和临时变量等；
  - 堆区是程序空间中存在的一些空闲存储单元，这些空闲存储单元组成堆，堆也称为自由存储单元，由程序员申请分配和释放。





- 在堆中创建的数据对象称为堆对象。
- 当堆对象不再使用时，应予以删除，回收其所占用的动态内存。
- 在 C++ 中建立和删除堆对象使用 new 和 delete 两个运算符。



### 1. new 运算符

◆ 在 C++ 程序中，运算符 new 的功能类似于 malloc()，用于从堆内存中分配指定大小的内存空间，并获得内存区域的首地址。

◆ new 运算符的语法格式包括三种形式：

（1）指针变量  $p = \text{new } T$ ；

（2）指针变量  $p = \text{new } T$ （初始值列表）；

（3）指针变量  $p = \text{new } T$  [元素个数]；

其中，p 是指针变量，用于返回申请的堆内存空间的首地址，T 是数据类型。



## 1. new 运算符

- 形式（1）只分配内存；
- 形式（2）将分配的堆内存进行初始化；
- 形式（3）分配具有 n 个元素的数组空间。

new 能够根据数据类型 T 自动计算分配的内存大小，若分配成功，指针变量 p 返回堆内存空间的首地址，如分配失败，则返回空指针。

例如：

```
int *p;  
p=new int(10);  
if(!p) {  
    cout<<" allocation failure" <<endl;  
    return 0;  
}
```



### 1. new 运算符

(1) T 是一个数据类型名，T 既可以是个系统预定义的数据类型，也可以用户自己定义的数据类型。初始值列表可以省略，例如：

```
int *p;
```

```
float *p1;
```

```
p=new int(10);    //p 指向一个数据类型为整型的堆地址，该地址中存放值 10
```

```
p1=new float;     //p1 指向一个数据类型为实型的堆地址
```



### 1. new 运算符

(2) new 可以为数组动态分配内存空间，这时应该在类型名后面指明数组大小。其中，元素个数是一个整型数值，可以是常数也可以是变量。指针类型应与数组类型一致。例如：

```
int *p=new int[10];    // 系统为指针 p 分配了有 10 个元素整型数组的内存
```

或

```
int n, *p;
```

```
cin>>n;
```

```
p=new int[n]; // 表示 new 为具有 n 个元素的整型数组分配了内存空间，并将首地址赋给了指针 p。
```



### 1. new 运算符

( 3 ) new 不能对动态分配的数组存储区进行初始化。例如：

```
int *p;
```

```
p=new int[10](0); // 错误，不能对动态分配的数组进行初始化
```

( 4 ) 用 new 分配的空间，使用结束后只能用 delete 显式地释放，否则这部分空间将不能回收而造成内存泄露。



### 2. delete 运算符

- ◆ 运算符 delete 的功能类似于 free()，用于释放 new 分配的堆内存空间，以便于被其他程序使用。
- ◆ delete 运算符的语法格式如下：

( 1 ) delete 指针变量名 p ;

( 2 ) delete[] 指针变量名 p ;

其中，p 是用 new 分配的堆空间指针变量，

形式 ( 1 ) 用于释放动态分配的单个对象内存空间；

形式 ( 2 ) 用于释放动态分配的数组存储区。



### 2. delete 运算符

- ◆ 释放动态分配的单个对象内存空间，例如：

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

```
//.....
```

```
delete p;      // 释放指针 p 所指向的动态内存空间
```

- ◆ 释放动态数组所占的内存空间，例如：

```
int *p;
```

```
p=new int[10];
```

```
//.....
```

```
delete []p;    // 释放为数组动态分配的内存
```





### 2. delete 运算符

- ( 1 ) new 和 delete 需要配套使用。
- ( 2 ) 在用 delete 释放指针所指的空间时，必须保证这个指针所指的空间是用 new 申请的，并且只能释放一次。
- ( 3 ) 如果在程序中用 new 申请了空间，就应该在结束程序前释放所有申请的空间，否则将造成内存泄漏。
- ( 4 ) 当 delete 用于释放由 new 创建的数组的连续内存空间时，无论是一维数组还是多维数组，指针变量名前必须使用 [ ]，且 [ ] 内没有数字。



### 2. delete 运算符

【例 2-15】使用 new 和 delete 申请内存和释放内存

```
#include <iostream>
using namespace std;
int main()
{
    int *p1, *p2, *p3;
    p1=new int;      // 分配一个 int 类型数据的内存区域
    p2=new int(10);   // 分配一个 int 类型的内存区域, 并将 10 存入其
    p3=new int[6];    // 分配能够存放 6 个整数的数组区域
    *p1=8;
    *p2=3;
    p3[0]=5;
    p3[1]=4;
```

中



### 2. delete 运算符

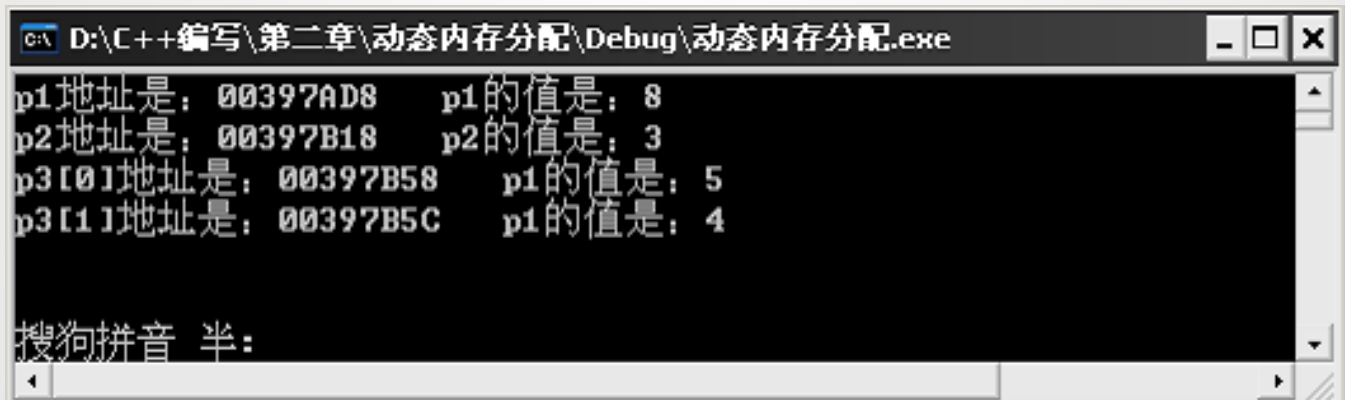
【例 2-15】使用 new 和 delete 申请内存和释放内存

```
cout<<"p1 地址是: "<<p1<<"    "<<"p1 的值是: "<<*p1<<endl;
cout<<"p2 地址是: "<<p2<<"    "<<"p2 的值是: "<<*p2<<endl;
cout<<"p3[0] 地址是: "<<p3<<"    "<<" p3[0] 的值是: "<<*p3<<endl;
cout<<"p3[1] 地址是: "<<&p3[1]<<"    "<<" p3[1] 的值是: "<<p3[1]<<endl;
delete p1;
delete p2;
//delete p3; // 错误, 只释放 p3 指向数组的第一个元素
delete []p3;
return 0;
}
```



### 2. delete 运算符

【例 2-15】使用 new 和 delete 申请内存和释放内存



```
C:\D:\C++编写\第二章\动态内存分配\Debug\动态内存分配.exe
p1地址是: 00397AD8    p1的值是: 8
p2地址是: 00397B18    p2的值是: 3
p3[0]地址是: 00397B58    p1的值是: 5
p3[1]地址是: 00397B5C    p1的值是: 4

搜狗拼音 半:
```



# 10



## 编译预处理

- ◆ 编译预处理是 C++ 编译系统的一个重要组成部分，它负责分析处理几种特殊的指令，这些指令被称为预处理命令。
- ◆ 编译预处理命令，可以改进程序设计环境，提高编程效率。
- ◆ 但它们不是 C++ 语言的组成部分，不能直接对它们进行编译。
- ◆ 编译系统在对源程序进行正式的编译之前，必须先对这些命令进行预处理，经过预处理后的程序不再包括预处理命令，然后由编译系统对预处理后的源程序进行通常的编译处理，得到可供执行的目标代码。

- ◆ C++ 提供的预处理命令主要有以下三种：
  - ( 1 ) 宏定义
  - ( 2 ) 文件包含
  - ( 3 ) 条件编译
- ◆ 这些命令均以“#”开头，每行一条命令，因为它们不是 C++ 的语句，所以命令后无分号。

### 1. 宏定义

- ◆ 可以利用预处理指令 `#define` 来定义宏，而使用 `#undef` 删除由 `#define` 定义的宏，使之不再起作用。
- ◆ 使用 `#define` 预处理指令可以把一个名称指定成任何文字，例如，常量值或者语句。
- ◆ 定义宏后，当此宏的名称出现在源代码中，预处理器就会把它替换掉。
- ◆ `#define` 可以定义符号常量、函数功能、重新命名、字符串的拼接等各种功能。例如：

```
#define PI 3.1415925    // 定义符号 PI 为 3.1415925
```

```
#undef PI              // 取消 PI 的值
```



### 1. 宏定义

说明：

- (1) 宏名一般用大写字母表示，以便与变量名相区别。
- (2) 使用宏名代替一个字符串，可以减少程序中重复书写某些字符串的工作量，当需要改变某一个常量时，可以只改变 `#define` 命令行，做到一改全改，不容易出错。
- (3) 宏定义是用宏名代替一个字符串，在宏展开时只是作简单的字符串替换，并不对语法是否正确进行检查。

### 1. 宏定义

说明:

- ( 4 ) 宏定义不是 C++ 语句，一定不要在行末加分号，如果加了分号，会将分号当成字符串的一部分进行替换。
- ( 5 ) 通常把 `#define` 命令放在一个文件的开头，使其定义在本文件内全部有效，即作用范围从其定义位置起到文件结束。
- ( 6 ) 可以使用 `#undef` 命令来取消宏定义的作用域。

### 2. 文件包 含

- ◆ 文件包含是将另一个源文件中的内容包含到当前文件中。
- ◆ 文件包含可以减少程序员的重复劳动。
- ◆ C++ 中使用 `#include` 预处理指令实现文件包含操作。
- ◆ 使用 `#include` 包含指令有两种格式：

`#include < 文件名 >`

`#include " 文件名 "`

- ◆ 前者 `<>` 用来引用标准库头文件，后者 `" "` 常用来引用自定义的头文件。

### 2. 文件包 含

◆前者 <> 编译器只搜索包含标准库头文件的默认目录，后者首先搜索正在编译的源文件所在的目录，找不到时再搜索包含标准库头文件的默认目录。如果把头文件放在其他目录下，为了查找到它，必须在双引号中指定从源文件到头文件的完整路径。

### 2. 文件包 含

- ( 1 ) 一条 `#include` 指令只能包含一个文件，如果想包含多个文件，需要用多条 `#include` 指令一一指定。
- ( 2 ) 在标准 C++ 中，`#include` 后面的文件名不再有 `.h` 扩展名。为了在 C++ 中使用 C 语言的库函数，标准 C++ 将 C 语言中的头文件前面加上 “`c`” 变为 C++ 头文件。
- ( 3 ) 包含可以是多重的，也就是说一个被包含的文件中还可以包含其他文件。预处理器至多支持 15 层嵌套包含。
- ( 4 ) 在 C++ 中，头文件是不允许相互包含。所谓相互包含是指 `a.h` 中包含 `b.h`，而 `b.h` 包含 `a.h`。

### 3. 条件编译

译

- ◆ 使用条件编译指令，可以限定程序中的某些内容在满足一定的条件下的情况下才参与编译。
- ◆ 条件编译指令可以使同一个源程序在不同的编译条件下产生不同的目标代码。
- ◆ 常用的条件编译指令有：

#if：如果

#ifndef：如果没有定义一个符号，就执行操作

#ifdef：如果定义了一个符号，就执行操作

#elif：否则如果

#endif：结束条件，

#undef：删除一个符号等，也是比较常见的预处理

### 3. 条件编译

译 常用的条件编译语句有 5 种形式:

(1) 指令 `#if` 和 `#endif`

`#if` 常量表达式

程序段 // 当 “常量表达式” 为真时, 编译本程序段

`#endif`

(2) 指令 `#if` 和 `#else`

`#if` 常量表达式

程序段 1 // 当 “常量表达式” 为真时, 编译本程序段

`#else`

程序段 2 // 当 “常量表达式” 为假时, 编译本程序段

`#endif`

### 3. 条件编译

常用的条件编译语句有 5 种形式:

( 3 ) 指令 #elif

```
#if 常量表达式 1
    程序段 1    // 当 “ 常量表达式 1 ” 为真时
编译
#elif 常量表达式 2
    程序段 2    // 当 “ 常量表达式 2 ” 为真
时编译
#else
    程序段 3    // 其他情况下编译
#endif
```



### 3. 条件编译

译 常用的条件编译语句有 5 种形式:

```
( 4 ) 指令 #ifdef 和 #else
#ifdef 标识符
    程序段 1 // 如果“标识符”定义过, 则编译程序
段 1
#else
    程序段 2 // 否则编译程序段 2
#endif
```

### 3. 条件编

译 常用的条件编译语句有 5 种形式:

```
( 5 ) 指令 #ifndef 和 #else
#ifndef 标识符
    程序段 1 // 如果“标识符”未定义过, 则编译程
序段 1
#else
    程序段 2      // 否则编译程序段 2
#endif
```

### 3. 条件编

译【例 2-16】 #ifndef 条件编译的应用例子

```
#include <iostream>
using namespace std;
#define GH
#ifndef GH
    void f() {cout<<"GH not defined!"<<endl;}
#else
    void f() {cout<<"GH is defined!"<<endl;}
#endif
int main()
{
    f();
    return 0;
}
```

# 11



## 文件的输入和输出

- ◆ 前面使用的输入输出是以系统指定的标准设备（输入设备为键盘，输出设备为显示器）为对象的。
- ◆ 在实际应用中，为了能够长期保留数据信息，常常以磁盘文件作为对象，即从磁盘文件中读取数据，或将数据输出到磁盘文件。
- ◆ 磁盘是计算机的外部存储器，能读能写，方便携带，因而得到广泛的使用。



- C++ 的流库中包含了三个专门处理文件输入输出的类：
  - ◆ ofstream 类：输出文件类（写操作），从 ostream 类派生而来。
  - ◆ ifstream 类：输入文件类（读操作），从 istream 类派生而来。
  - ◆ fstream 类：可同时输入输出的文件类（读写操作），从 iostream 类派生而来。
- C++ 的文件操作是首先通过将 ifstream、ofstream、fstream 流类的对象与某个磁盘文件联系起来，创建一个文件流，然后调用这些类的成员函数实现文件的打开、读写和关闭操作。



### 1. 文件的打开和关闭

#### 1.1 打开磁盘文件

文件被打开后，才能进行读写操作。分两步完成：

（1）打开磁盘文件时，首先定义流对象建立输入流、输出流或输入输出流，具体定义格式如下：

```
ifstream 输入流变量名；  
ofstream 输出流变量名；  
fstream  输入输出流变量名
```

例如：

```
ifstream inData;    // 定义输入文件流变量  
ofstream outData;   // 定义输出文件流变量
```



### 1. 文件的打开和关闭

#### 1.1 打开磁盘文件

(2) 建立输入输出流后, 则可用流对象调用 `open()` 成员函数将文件打开, 即将文件与刚建立的流联系起来。

调用 `open()` 函数的一般形式为:

文件流对象 . `open`( 磁盘文件名, 文件的打开模式 );





## 1. 文件的打开和关闭

### 1.1 打开磁盘文件

文件的打开模式可以是：

`ios::in` 打开一个输入文件（默认方式）。

`ios::out` 建立一个输出文件（默认方式），如果此文件已存在，则将原有内容删除。

`ios::app` 若文件存在，将数据被追加到文件的末尾，若不存在，就建立文件。

`ios::ate` 打开文件时，文件指针位于文件尾。

`ios::trunc` 删除文件原来已存在的内容（清空文件）。

`ios::nocreate` 若文件并不存在，打开操作失败。

`ios::noreplace` 若文件已存在，打开操作失败。

`ios::binary` 以二进制的形式打开一个文件，缺省时按文本文件打开。



### 1. 文件的打开和关闭

#### 1.1 打开磁盘文件

- 假如打算设置不止一个的打开模式标志，只须使用“OR”操作符或者是“|”，例如：
  - `ios::app OR ios::binary`
- 假如要打开目录 C:\EF 下的 aa.txt 文件，若文件存在就打开，若不存在就建立该文件，可以用以下命令建立：
  - `ofstream outData;`
  - `outData.open( "C:\\EF\\aa.txt" , ios::app);`



### 1. 文件的打开和关闭

#### 1.1 打开磁盘文件

说明:

(1) 由于“\”被 C++ 用于转义符,所以在指定文件路径时用“\\”作为文件路径中目录之间的间隔符,与回车换行符“\n”中的“\”意义相同;

(2) 打开一个文件时,也可以不使用 open() 成员函数,而是调用流类对象的构造函数(后续章节学习)来打开,这些构造函数的参数与 open() 函数完全相同,例如:

```
ofstream outData("C:\\EF\\aa.txt", ios::app);
```

此语句等价于上面的两条语句,实现打开或建立目录 C:\EF 下的 aa.txt 文件。



## 1. 文件的打开和关闭

### 1.2 关闭磁盘文件

对已经打开的磁盘文件读写操作完成后，应关闭该文件。

关闭文件的成员函数为 `close()`，解除磁盘文件与文件流的关联。

调用 `close()` 函数的一般形式为：

文件流对象.`close()`

例如：

`outData.close()` // 关闭流 `outData` 与文件

`C:\\EF\\aa.txt` 的连接



### 2. 文件的输入和输出

- ◆ 当文件打开后，即建立文件与流对象关联后，就可以进行输入输出（读写）操作了。
- ◆ 输入输出操作与 `cout`、`cin` 用法相同，可以使用插入运算符“<<”或析取运算符“>>”从文件中读写数据。将输入文件流变量与“>>”连接能够从文件中读入数据，将输出文件流变量与“<<”连接能够将数据输出到文件中。

例如：

```
outData<<x;      // 将变量 x 的值输出到文件中  
inData>>x;       // 从文件中读入变量 x 的值
```



### 2. 文件的输入和输出

C++ 文件操作过程的 5 个步骤:

(1) 首先在程序包含头文件 `fstream`。

```
#include <fstream>
```

(2) 定义文件流变量。

```
ifstream inData;          // 定义输入文件流变量
```

```
ofstream outData;         // 定义输出文件流变量
```

(3) 使用 `open()` 函数将文件流变量与磁盘文件关联起来。

```
outData.open("C:\\EF\\aa.txt", ios::app);
```

第 (2) 步、第 (3) 步也可以合并为一步，下面的命令与上面的两条命令等价:

(4) 用文件流变量和 “<<” 或 “>>” 结合读写文件数据

(5) 关闭文件



### 2. 文件的输入和输出

【例 2-17】建立一个磁盘文件 C:\\d.txt。

- (1) 从键盘中输入字符串“床前明月光，疑是地上霜，举头望明月，低台头思故乡。”到文件中。
- (2) 从该磁盘文件中读出该字符串并在屏幕上显示。



### 2. 文件的输入和输出

【例 2-17】建立一个磁盘文件 C:\\d.txt。

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    string str1, str2;
    ofstream outstr;
    ifstream instr;
    outstr.open("C:\\d.txt", ios::out)
    if(!outstr)
```





### 2. 文件的输入和输出

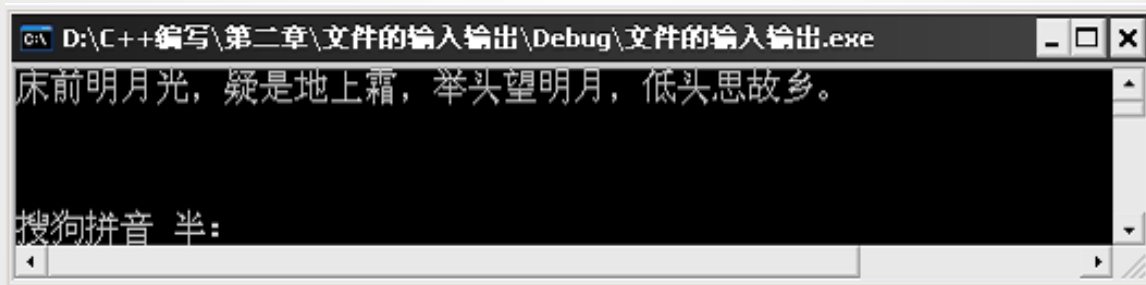
【例 2-17】建立一个磁盘文件 C:\\d.txt。

```
{ cerr<<" 打开失败! ";  
    return -1; }  
str1=" 床前明月光, 疑是地上霜, 举头望明月, 低台  
头思故乡。";  
outstr<<str1;    // 对文件写操作  
    ostr.close();  
instr.open("C:\\d.txt");  
instr>>str2;    // 从文件读操作  
cout<<str2;  
instr.close();  
return 0;}
```



### 2. 文件的输入和输出

【例 2-17】建立一个磁盘文件 C:\\d.txt。



### 1. 命名空间

为了解决不同模块或者函数库中标识符命名冲突的问题，C++ 引入命名空间的概念。命名空间可以由程序员自己来创建，可以将不同的标识符集合在一个命名作用域内，包括类、对象、函数、变量及结构体等。std 命名空间是 C++ 提供的标准命名空间，标准 C++ 将新格式头文件的内容全部放到了 std 命名空间中。

## 2. 数据的输入和输出

在 C++ 程序中，常用 `cin` 从键盘中输入数据，使用 `cout` 在屏幕上显示字符和数字等数据。在实际应用中，为了能够长期保留数据信息，常常以磁盘文件作为对象，即从磁盘文件中读取数据，或将数据输出到磁盘文件。C++ 的流库中包含了 3 个专门处理文件输入输出的类，分别是输出文件类（写操作）`ofstream`、输入文件类（读操作）`ifstream`、可同时输入输出的文件类（读写操作）`fstream`。C++ 的文件操作是首先通过将 `ifstream`、`ofstream`、`fstream` 流类的对象与某个磁盘文件联系起来，创建一个文件流，然后调用这些类的成员函数实现文件的打开、读写和关闭操作。

### 3. 函数重载

函数重载是指两个或两个以上的函数具有相同的函数名，但参数类型不一致或参数个数不同。编译时编译器将根据实参和形参的类型及个数进行相应地匹配，自动确定调用哪一个函数。使得重载的函数虽然函数名相同，但功能却不完全相同。函数重载，方便使用，便于记忆。

### 4. 带有默认参数的函数

C++ 中，允许函数提供默认参数，即在函数的声明或定义时给一个或多个参数指定默认值。在调用具有默认参数的函数时，如果没有提供实际参数，C++ 将自动把默认参数作为相应参数的值。

### 5. 内联函数

函数的调用，需要保存和恢复现场和地址，需要时间和空间的开销，会降低程序的执行效率。为解决这一问题，C++ 中对于功能简单、规模小、使用频繁的函数，可以将其设置为内联函数。内联函数在编译时，C++ 将用内联函数代码替换对它每次的调用。节省参数传递、控制转移的开销，从而提高了程序运行时的效率。内联函数是一种空间换时间的方案。

## 6. 引用

引用为一个变量的别名，可以将函数的参数和返回值定义为引用。引用只须传递一个对象的地址，从而提高函数的调用和运行效率效率。可以将函数的参数和返回值设置为引用。



### 7. 常量的定义

C++ 语言除了 `#define` 外，还可以用 `const` 来定义常量。但一般使用 `const` 来定义常量。

## 8. 变量的生存期和作用域

作用域是一个标识符在程序正文中有效的区域。C++ 中标识符的作用域有函数原型作用域、局部作用域（块作用域）、类作用域、文件作用域和命名空间作用域。

变量的生存期是指变量从被创建分配内存开始到被释放内存为止的时间，不同存储的变量生存期不同。变量生存期有：静态生存期和动态生存期。

## 9. 动态内存申请和释放

在 C++ 中，使用 `new` 和 `delete` 在堆中申请和释放动态空间。