第四章 功能（过程）抽象——函数

主讲人：侯海良

通信与控制工程系

本章内容

- 基于过程抽象的程序设计
- 子程序的概念
- C++的函数
- 变量的局部性和变量的生存期
- 标识符的作用域
- 递归函数
- 内联函数
- 函数名重载
- 条件编译——程序调试与多环境程序编制
- 标准库函数

基于过程抽象的程序设计

- 人们在设计一个复杂的程序时，经常会用到功能分解和复合两种手段：
  - 功能分解：在进行程序设计时，首先把程序的功能分解成若干子功能，每个子功能又可以分解成若干子功能，等等，从而形成了一种自顶向下（top-down）、逐步精化（step-wise）的设计过程。
  - 功能复合：把已有的（子）功能逐步组合成更大的（子）功能，从而形成一种自底向上（bottom-up）的设计过程。
- 过程抽象：一个功能的使用者只需要知道相应功能是什么（what to do），而不必知道它是如何做（how to do）的。

子程序

- 子程序是取了名的一段程序代码，在程序中通过名字来使用（调用）它们。
- 子程序的作用：
  - 减少重复代码，节省劳动力
  - 实现过程抽象（功能抽象）
  - 封装和信息隐藏的作用

子程序之间的数据传递

- 一个子程序所需要的数据往往要从调用者（也是一个子程序）那里获得，计算结果也需要返回给调用者。
- 子程序之间的数据传递方式可以通过：
  - 全局变量：所有子程序都能访问到的变量。（不好）
  - 参数：形式参数（形参）和实在参数（实参）。
    - 值传递：把实参的值复制一份给形参。
    - 地址或引用传递：把实参的地址传给形参。
  - 返回值机制：返回计算结果。

C++函数

- 函数是C++提供的用于实现子程序的语言成分。
- 函数的定义：

  <返回值类型> <函数名> (<形式参数表>) <函数体>

  - <返回值类型>描述了函数返回值的类型，
    - 可以为任意的C++数据类型。
    - 当返回值类型为void时，它表示函数没有返回值。
  - <函数名>用于标识函数的名字，用标识符表示。
  - <形式参数表>描述函数的形式参数，由零个、一个或多个形参说明（用逗号隔开）构成，形参说明的格式为：

    <类型> <形参名>

- <函数体> 为一个<复合语句>，用于实现相应函数的功能。
  - 函数体内可以包含return语句，格式为：
    - **return <表达式>;**
    - **return;**
  - 当函数体执行到return语句时，函数立即返回到调用者。如果有返回值，则把返回值带回给调用者。
  - 如果return中的<表达式>的类型与函数<返回值类型> 不一致，则进行隐式类型转换，基本原则为：把<表达式>转成<返回值类型> 。
  - **注意：在函数体中不能用goto语句转出函数体。**

例1：用函数实现阶乘

```
int factorial(int n) //求n的阶乘
{ int i,f=1;
  for (i=2; i<=n; i++)
     f *= i;
  return f;
}
```

例2：编写求x^n的函数

```
double power(double x, int n) //求x的n次幂
{  if (x == 0) return 0;
   double product=1.0;
   if (n >= 0)
        while (n > 0)
        {       product *= x;
                n--;
        }
   else
        while (n < 0)
        {       product /= x;
                n++;
        }
   return product;
}
```

函数main

- 每个C++程序都要定义一个名字为main的函数，C++程序的执行是从main开始的。对于函数main，其返回值类型为int，例如：

```
int main()
{ ......
   ... return -1;
   ......
   return 0;
}
```

- 一般情况下，返回0表示程序正常结束；返回负数（如－1）表示程序非正常结束。

函数的调用

- 对于定义的一个函数，必须要调用它，它的函数体才会执行。
- 除了函数main外，程序中对其它函数的调用都是从main开始的。main一般是由操作系统来调用。
- 函数调用的格式如下：

  <函数名>(<实在参数表>)

  - <实在参数表>由零个、一个或多个表达式构成（逗号分割）
  - 实参的个数和类型应与相应函数的形参相同。类型如果不同，编译器会试图进行隐式转换，转换规则是把实参类型转换成形参类型 。
- 注意：不能用goto语句从函数外转入函数体

函数调用的例子

```
.....
int main()
{ int x;
  cout << "请输入一个正整数：";
  cin >> x;
  cout << "Factorial of " << x << " is "
       << factorial(x) //调用阶乘函数
       << endl;
  return 0;
}
```

```
.....
int main()
{ double a;
  int b;
  cout << "请输入a和b：";
  cin >> a >> b;
  cout << a << "的" << b << "次方是："
       << power(a,b) << endl;
  return 0;
}
```

函数调用的执行过程

- 计算实参的值（对于多个实参，C++没有规定计算次序）；
- 把实参分别传递给被调用函数的形参；
- 执行函数体；
- 函数体中执行return语句返回函数调用点，调用点获得返回值（如果有返回值）并执行调用之后的操作。
- 可以把有返回值的函数调用作为操作数放在表达式中参加运算：x+power(x,y)*z

函数声明

- 程序中调用的所有函数都要有定义。
- 如果函数定义在其它文件（如：C++的标准库）中或定义在本源文件中使用点之后，则在调用前需要对被调用的函数进行声明。
- 函数声明的格式如下：

  <返回值类型> <函数名>(<形式参数表>); //函数原型

  或

  extern <返回值类型> <函数名>(<形式参数表>);

  - 在函数声明中，<形式参数表>中可以只列出形参的类型而不写形参名

```
//file1.cpp
int x=0; //定义
int main() //定义
{ extern void f(); //声明
   extern int g(int); //声明
   extern int y; //声明
   y = x + 2;
   f(); //调用
   y = g(x); //调用
   return 0;
}
int y=0; //定义
void f() //定义
{x = y + 1;
}
```

```
//file2.cpp
int g(int i) //定义
{ extern int x,y; //声明
   int z; //定义
   z = x + y;
   return z+i;
}
```

函数声明的作用是什么？

例5：用函数实现求小于n的所有素数。

```
#include <iostream>
#include <cmath>
using namespace std;
bool is_prime(int n);//函数声明
void print_prime(int n, int count);//函数声明
int main()
{  int i,n,count=1;
   cout << "请输入一个正整数："
   cin >> n;  //从键盘输入一个正整数
   if (n < 2) return -1;
   cout << 2 << ",";  //输出第一个素数
   for (i=3; i<n; i+=2)
   {    if (is_prime(i))
        {       count++;
                print_prime(i,count);
        }
   }
   cout << endl;
   return 0;
}
```

```
bool is_prime(int n)
{       int i,j,k=sqrt(n);
        for (i=2, j=k; i<=j; i++)
                if (n%i == 0) return false;
        return true;
}

void print_prime(int n, int count)
{       cout << n << ',';
        if (count % 6 == 0) cout << endl;
}
```

函数的参数传递

- C++提供了两种参数传递机制：
  - 值传递
    - 把实参的值赋值给形参。
  - 地址或引用传递
    - 把实参的地址赋值给形参。
- C++默认的参数传递方式是值传递。

值传递

- 在函数调用时，采用类似变量初始化的形式把实参的值传给形参。
- 函数执行过程中，通过形参获得实参的值，
- 函数体中对形参值的改变不会影响相应实参的值。

值参数传递的例子

```
//函数main调用函数power计算a^b
#include <iostream>
using namespace std;
double power(double x, int n);
int main()
{  double a=3.0,c;
   int b=4;
   c = power(a,b);
   cout << a << "," << b << "," << c << endl;
   return 0;
}
```

```
double power(double x, int n)
{  if (x == 0) return 0;
   double product=1.0;
   if (n >= 0)
        while (n > 0)
        {       product *= x;
                        n--;
        }
   else
        while (n < 0)
        {       product /= x;
                        n++;
        }
   return product;
}
```

- 执行main时，产生三个变量（分配内存空间）a、b和c：

  a: 3.0 b: 4 c: ?

- 调用power函数时，又产生三个个变量x、n和product，然后分别用a、b以及1.0对它们初始化：

  a: 3.0 b: 4 c: ?

  x: 3.0 n: 4 product: 1.0

- 函数power中的循环结束后（函数返回前）：

  a: 3.0 b: 4 c: ?

  x: 3.0 n: 0 product: 81.0

- 函数power返回后：

  a: 3.0 b: 4 c: 81.0

变量的局部性

- 在C++中，根据变量的定义位置，把变量分成：局部变量和全局变量。
  - 局部变量是指在复合语句中定义的变量，它们只能在定义它们的复合语句（包括内层的复合语句）中使用。
  - 全局变量是指在函数外部定义的变量，它们一般能被程序中的所有函数使用（静态的全局变量除外）。

局部变量和全局变量的例子

```
int x=0;  //全局变量
void f()
{  int y=0; //局部变量
   x++; //OK
   y++; //OK
   a++; //Error
}
```

```
int main()
{  int a=0; //局部变量
   f();
   a++; //OK
   x++; //OK
   y++; //Error
   while (x<10)
   { int b=0; //局部变量
      a++; //OK
      b++; //OK
      x++; //OK
   }
   b++; //Error
   return 0;
}
```

变量的生存期（存储分配）

- 把程序运行时一个变量占有内存空间的时间段称为该变量的生存期。
  - 静态：从程序开始执行时就进行内存空间分配，直到程序结束才收回它们的空间。全局变量具有静态生存期 。
  - 自动：内存空间在程序执行到定义它们的复合语句（包括函数体）时才分配，当定义它们的复合语句执行结束时，它们的空间将被收回。局部变量和函数的参数一般具有自动生存期。
  - 动态 ：内存空间在程序中显式地用new操作或malloc库函数分配、用delete操作或free库函数收回。动态变量具有动态生存期。
- 具有静态生存期的变量，如果没有显式初始化，系统将把它们初始化成0。

存储类修饰符

- 在定义局部变量时，可以为它们加上存储类修饰符来显式地指出它们的生存期。
  - auto：使局部变量具有自动生存期。局部变量的默认存储类为auto。
  - static：使局部变量具有静态生存期。它只在函数第一次调用时进行初始化，以后调用中不再进行初始化，它的值为上一次函数调用结束时的值。
  - register：使局部变量也具有自动生存期，由编译程序根据CPU寄存器的使用情况来决定是否存放在寄存器中。

```
void f()
{ auto int x=0; //auto一般不写
   static int y=1;
   register int z=0;
   x++; y++; z++;
   cout << x << y << z << endl;
}
 int main()
{ f();      =>调用f时，输出：x=1, y=2, z=1
   z++;
   f();     =>调用f时，输出：x=1,y=3,z=1
   return 0;
}
```

Static 实例

程序实体在内存中的安排

- 例7：编写一个能够产生随机数的函数

```
usingned int random（）
{  static usigned int seed=1；
   seed=(25173*seed+13849)%65536；
   return seed；
}
```

程序运行时各种数据在内存中的分配：

- 静态数据区用于全局变量、static存会储类的局部变量以及常量的内存分配 。（该区变量未初始化时会默认为0）
- 代码区用于存放程序的指令，对C++程序而言，代码区存放的是所有函数代码；
- 栈区用于auto存储类的局部变量、函数的形式参数以及函数调用时有关信息（如：函数返回地址等）的内存分配；
- 堆区用于动态变量的内存分配。

C++程序的多模块结构

- 逻辑上，一个C++程序由一些全局函数（区别于类定义中的成员函数）、全局常量、全局变量/对象以及类的定义构成，其中必须有且仅有一个名字为main的全局函数。函数内部可以包含形参、局部常量、局部变量/对象的定义以及语句。
- 物理上，可以按某种规则对构成C++程序的各个逻辑单位（全局函数、全局常量、全局变量/对象、类等）的定义进行分组，分别把它们放在若干个源文件中，构成程序模块。
- 程序模块是为了便于从物理上对程序进行组织、管理和理解，便于多人合作开发一个程序。
- 程序模块是可单独编译的程序单位。

C++模块的构成

- 一个C++模块一般包含两个部分：
  - 接口（.h文件 ）：
    - 给出在本模块中定义的、提供给其它模块使用的一些程序实体（如：函数、全局变量等）的声明；
  - 实现（.cpp文件）：
    - 模块的实现给出了模块中的程序实体的定义。

- 在模块A中要用到模块B中定义的程序实体时，可以在A的.cpp文件中用文件包含命令（#include）把B的.h文件包含进来。
- 文件包含命令是一种编译预处理命令，其格式为：

  #include <文件名> 或 #include "文件名"

  注意：（1） #include <文件名>表示在系统指定的目录下寻找指定文件

  （2）#include "文件名"表示在源文件所在的目录下寻找指定文件

  - include命令的含义是：在编译前，用命令中的文件名所指定的文件内容替换该命令。

```
//file1.h
extern int x;      //全局变量x的声明
extern double y;  //全局变量y的声明
int f();           //全局函数f的声明

//file1.cpp
int x=1;      //全局变量x的定义
double y=2.0;     //全局变量y的定义
int f()           //全局函数f的定义
{ int m;      //局部变量m的定义
   ……
   m += x;       //语句
   ……
   return m;
}
```

```
//file2.h
void g();    //全局函数g的声明

//file2.cpp
#include "file1.h"  //把文件file1.h中的内容包含进来
void g()    //全局函数g的定义
{ double z;    //局部变量z的定义
   ……
   z = y+10;    //语句
   ……
}
```

```
//main.cpp
#include "file1.h"  //把文件file1.h中的内容包含进来
#include "file2.h"  //把文件file2.h中的内容包含进来
int main()  //全局函数main的定义
{ double r; //局部变量r的定义
   .......
   r = x+y*f();  //语句
   .......
   g();  //语句
   .......
}
```

标识符的作用域

- 在程序常量、变量、函数、类等的命名中，要区分它们需要命不同的名字，这样程序才能区分它们。但这会带来不便。如两个程序员写函数的局部变量，可能会出现命名相同。
- 为了对程序中的实体的名字进行管理，引进了标识符的作用域的概念。
  - 一个定义了的标识符的有效范围（能被访问的程序段）称为该标识符的作用域。
- 在不同的作用域中，可以用相同的标识符来标识不同的程序实体。

C++标识符的作用域

- C++把标识符的作用域分成若干类，其中包括：
  - 局部作用域
  - 全局作用域
  - 文件作用域
  - 函数作用域
  - 函数原型作用域
  - 类作用域
  - 名空间作用域

局部作用域

- 在函数定义或复合语句中、从标识符的定义点开始到函数定义或复合语句结束之间的程序段。
- C++中的局部常量名、局部变量名/对象名以及函数的形参名具有局部作用域。

- 例：

```
void f(int y)
{ …y…;          //OK
  …x…;          //Error
  int x;
  …x…;          // OK
  double x;     //Error,重名
  char y;       //Error，重名
}
```

```
void g(double y)    // OK
{
  double x;   //OK
  f(1);       // OK
  ……
}
```

```
void f(int n)
{ x++;     //Error
   int x=0;
    x++;
     n++;
      .......
}
void g()
{  x++;   //Error
   n++;   //Error
}
int main()
{ int x=0;
   int n;
   cin >> n;
   f(n);
     ......
}
```

- 如果在一个标识符的局部作用域中包含内层复合语句，并且在该内层复合语句中定义了一个同名的不同实体，则外层定义的标识符的作用域应该是从其潜在作用域中扣除内层同名标识符的作用域之后所得到的作用域。

```
void f()
{ int x;  //外层x的定义
   ... x ... //外层的x
   while  ( ... x ...)  //外层的x
   { ... x ... //外层的x，
     double x;  //内层x的定义
     ... x ...  //内层的x
   }
   ... x ...  //外层的x
}
```

全局作用域

- 在函数级定义的标识符具有全局作用域。
- 全局变量名/对象名、全局函数名和全局类名的作用域一般具有全局作用域，它们在整个程序中可用。
- 如果在某个局部作用域中定义了与某个全局标识符同名的标识符，则该全局标识符的作用域应扣掉与之同名的局部标识符的作用域。
- 在局部标识符的作用域中若要使用与其同名的全局标识符，则需要用全局域选择符（::）对全局标识符进行修饰（受限）。

```
double x;  //外层x的定义
void f()
{ int x;  //内层x的定义
   ... x ... //内层的x
   ::x ...    //外层的x
}
```

作用域的综合实例

文件作用域

- 在全局标识符的定义中加上static修饰符，则该全局标识符就成了具有文件作用域的标识符，它们只能在定义它们的源文件（模块）中使用。
- C++中的关键词static有两个不同的含义。
  - 在局部变量的定义中，static修饰符用于指定局部变量采用静态存储分配；
  - 而在全局标识符的定义中，static修饰符用于把全局标识符的作用域改变为文件作用域。
- 一般情况下，具有全局作用域的标识符主要用于标识被程序各个模块共享的程序实体，而具有文件作用域的标识符用于标识在一个模块内部共享的程序实体。