



第四章 过程抽象--函数



本章内容

- 基于过程抽象的程序设计
- C++函数
- 标识符的作用域和变量的生存期
- 递归函数
- C++标准库函数
- 函数的进一步讨论
- C++编译预处理命令



本章内容

- 基于过程抽象的程序设计
- C++函数
- 标识符的作用域和变量的生存期
- 递归函数
- C++标准库函数
- 函数的进一步讨论
- C++编译预处理命令



过程式程序设计

○ 设计复杂程序的两种手段：

- **功能分解**：把程序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，形成了一种自顶向下、逐步精化的设计过程。
- **功能复合**：把已有的（子）功能组合成更大的（子）功能，形成自底向上的设计过程。

○ 各个子功能体现为一个子程序。子程序为基于功能分解和复合的过程式程序设计提供了基础。



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调用它们。
- 子程序的作用：
 - 减少重复代码，提高代码复用率；
 - 实现过程抽象
 - 实现封装和信息隐藏



子程序之间的数据传递

○ 三种数据传递方式：

- 全局变量（不好）：

- 破坏子程序独立性
- 破坏数据安全
- 带来额外系统开销

- 参数传递：

- 值传递：复制实参的值给形参。
- 地址或引用传递：把实参的地址传给形参。

- 返回值机制。



本章内容

- 基于过程抽象的程序设计
- C++函数
- 标识符的作用域和变量的生存期
- 递归函数
- C++标准库函数
- 函数的进一步讨论
- C++编译预处理命令



函数

- 函数是C++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>`



函数体

- 函数体为一个复合语句，用于实现函数功能。
 - 函数体内可以包含return语句，格式为：
 - return <表达式>;
 - return;
 - 如果return中的<表达式>的类型与函数<返回值类型> 不一致，则把<表达式>隐式转换为<返回值类型>。
 - 在函数体中不能用goto语句转出和转入函数体。



```
int factorial(int n) //求n的阶乘
```

```
{ int i,f=1;
```

```
    for (i=2; i<=n; i++) f *= i;
```

```
    return f;
```

```
}
```

```
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;
```

```
}
```



函数调用

- 函数调用：

- <函数名>(<实参表>)

- 执行过程：

- 计算实参值（C++没有规定实参之间的计算次序）；
 - 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。



函数声明

- 函数定义在其它文件中或定义在本源文件中使用点之后，则在调用前需要对函数进行声明。
- 函数声明的作用给编译程序提供信息，使得编译程序能够对函数调用的合法性进行检查，并产生函数调用的正确代码。



函数声明

○ 函数声明格式:

- `<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;
```

```
int main()
```

```
{ extern void f(); //声明
```

```
  extern void g(); //声明
```

```
  extern int y;
```

```
  y = x + 2;
```

```
  f(); //调用
```

```
  g(); //调用
```

```
  return 0;
```

```
}
```

```
int y=0;
```

```
void f() //定义
```

```
{x = y + 1;
```

```
}
```

```
//file2.cpp
```

```
void g() //定义
```

```
{ extern int x,y; //声明
```

```
  int z; //定义
```

```
  z = x + y;
```

```
}
```



函数的参数传递

○ C++的参数传递机制：

- 值传递：把实参的值赋值给形参，不影响实参值。
- 地址或引用传递：把实参的地址赋值给形参，改变实参值。



//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;
```

```
}
```


- 
- 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



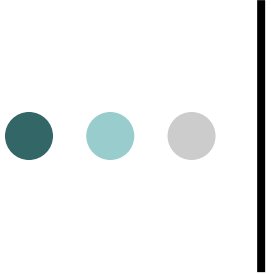
本章内容

- 基于过程抽象的程序设计
- C++函数
- 标识符的作用域和变量的生存期
- 递归函数
- C++标准库函数
- 函数的进一步讨论
- C++编译预处理命令



变量的局部性

- C++根据变量的定义位置，把变量分成：
 - 全局变量：在函数外部定义的变量，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：在复合语句中定义的变量，只能在定义它们的复合语句中使用。



```
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;
}
```

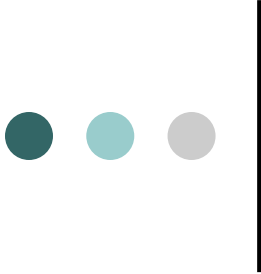
变量的生存期（存储分配）

- 变量占有内存空间的时间段称为变量的生存期。
- C++把变量的生存期分为：
 - 静态**：内存空间从程序开始执行时就进行分配，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。



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可以不写
  static int y=1;
  register int z=0;
  x++; y++; z++;
  cout << x << y << z << endl;
}
```

第一次调用f时，输出：1 2 1

第二次调用f时，输出：1 3 1

程序实体在内存中的安排

程序的内存分配

- **静态数据区**：用于全局变量、static存储类的局部变量以及常量的内存分配。
- **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 P98

- 逻辑上

- 物理上

- 对逻辑单位的分组体现了程序设计中的模块概念
- 设计模块原则：内聚性最大，耦合度最小



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程序实体（如：函数、全局变量等）的**声明**；
 - 实现（.cpp文件）：给出了模块中的程序实体的**定义**。
- 文件包含命令
 - `#include <文件名>` **或** `#include "文件名"`
 - 编译**预处理**命令，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" //包含文件file1.h
```

```
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" //包含文件file1.h
```

```
#include "file2.h" //包含文件file2.h
```

```
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

- 标识符的有效范围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。



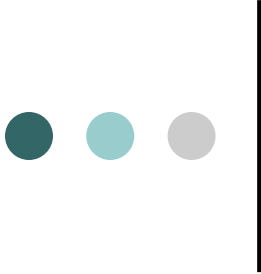
C++标识符的作用域

- 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。
- 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得到的作用域。

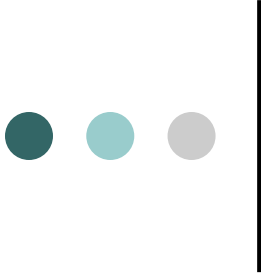


```
void f()
{ ... x ... //Error
  int x; //外层x的定义
  ... x ... //外层的x
  while ( ... x ... ) //外层的x
  {   ... x ... //外层的x
      double x; //内层x的定义
      ... x ... //内层的x
  }
  ... x ... //外层的x
}
```



全局作用域

- 全局作用域指构成C++程序的所有源文件。
- 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。
- 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。在此局部标识符的作用域中，可以用全局域解析符 (::) 来使用全局标识符。
- 把全局标识符的声明放在某个.h文件中，在需要使用#include编译预处理命令把声明文件包含进来。



```
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  
}
```



文件作用域 (1)

- 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中使用**。
- 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。



```
//file1.cpp
```

```
static int y; //文件作用域
```

```
static void f() //文件作用域
```

```
{ .....
```

```
}
```

```
//file2.cpp
```

```
extern int y;
```

```
extern void f();
```

```
void g()
```

```
{ ... y ... //Error
```

```
    f(); //Error
```

```
}
```



文件作用域 (2)

- 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修饰符指定局部变量采用静态存储分配；
 - 在全局标识符的定义中，static修饰符把全局标识符的作用域改变为文件作用域。



函数作用域

- **语句标号**是唯一具有函数作用域的标识符，它们在定义它们的函数体中的任何地方都可以访问。
- 函数作用域与局部作用域的区别：
 - 函数作用域包括**整个函数**，而局部作用域是**从定义点开始到函数定义或复合语句结束**。
 - 在函数体中，**一个语句标号只能定义一次**，即使是在内层的复合语句中，也不能再定义与外层相同的语句标号。



```
void f()
{ .....
  goto L; //OK
  .....
L: ...
  .....
  { .....
    L: ... //Error
    .....
  }
  .....
  goto L; //OK
  .....
}
```

```
void g()
{ .....
  goto L; //Error
  .....
}
```




函数原型作用域

- 函数声明中，形式参数的作用域从函数原型开始到函数原型结束。

```
void f(int x, double y); // x和y的作用域从  
                        // “(” 到 “)” 结束
```



名空间作用域

- 在一个源文件中要用到在两个外部源文件中定义的两个不同全局程序实体，而这两个全局程序实体的名字相同。
- C++提供了名空间解决上述的名冲突问题。
 - 在一个名空间中定义的全局标识符，其作用域为该名空间。
 - 当在一个名空间外部需要使用该名空间中定义的全局标识符时，可用加上该名空间的名字来调用。

- ● **//模块1**
namespace A
{ int x=1;
void f()
{
}
}

//模块2
namespace B
{ int x=0;
void f()
{
}
}

//模块3

- 1、
... **A::x** ... **//A中的x**
A::f(); **//A中的f**
... **B::x** ... **//B中的x**
B::f(); **//B中的f**
- 2、
using namespace A;
... **x** ... **//A中的x**
f(); **//A中的f**
... **B::x** ... **//B中的x**
B::f(); **//B中的f**
- 3、
using A::f;
... **A::x** ... **//A中的x**
f(); **//A中的f**
... **B::x** ... **//B中的x**
B::f(); **//B中的f**



本章内容

- 基于过程抽象的程序设计
- C++函数
- 标识符的作用域和变量的生存期
- 递归函数
- C++标准库函数
- 函数的进一步讨论
- C++编译预处理命令

递归函数

- 函数在其函数体中直接或间接地调用本身
- 分而治之的设计方法:
 - 把一个问题分解成若干个子问题，而每个子问题的性质与原问题相同，只是在规模上比原问题要小。每个子问题的求解过程可以采用与原问题相同的方式来进行。

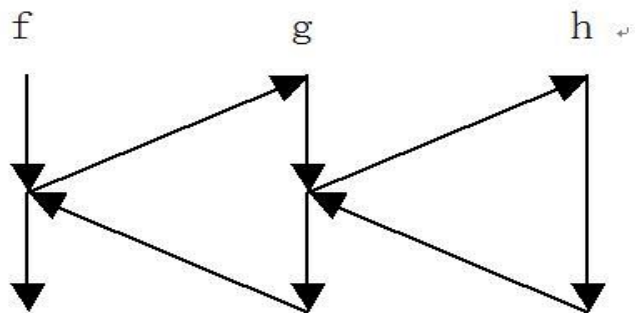
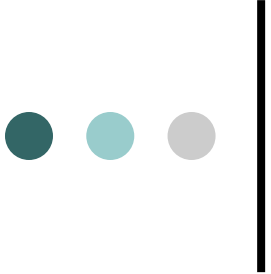


图 4-3 函数的嵌套调用及返回。



○ 直接递归

```
void f()
{ .....
  ... f() ...
  .....
}
```

○ 间接递归

```
extern void g();
void f()
{ .....
  ... g() ...
  .....
}
void g()
{ .....
  ... f() ...
  .....
}
```



递归条件和结束条件

○ 递归函数的描述：

- **递归条件**。指出**何时进行递归调用**，它描述了问题求解的一般情况，包括：分解和综合过程。
- **结束条件**。指出**何时不需递归调用**，它描述了问题求解的特殊情况或基本情况



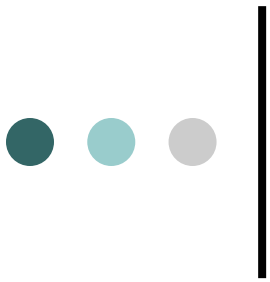
例：求第n个fibonacci 数 (递归解法)

```
int fib(int n)
{ if (n == 1)
    return 0;
  else if (n == 2)
    return 1;
  else
    return fib(n-2)+fib(n-1);
}
```


例：解汉诺塔问题

• 汉诺塔问题：有A, B, C三个柱子，柱子A上穿有n个大小不同的圆盘，大盘在下，小盘在上。现要把柱子A上的所有圆盘移到柱子B上，要求每次只能移动一个圆盘，且**大盘不能放在小盘上**，移动时可借助柱子C。编写一个C++函数给出移动步骤，如：n=3时，移动步骤为：
 $A \rightarrow B, A \rightarrow C, B \rightarrow C, A \rightarrow B, C \rightarrow A, C \rightarrow B, A \rightarrow B$ 。





- 当 $n=1$ 时，只要把圆盘从A移至B就可以了（输出： $A \rightarrow B$ ）。而当 n 大于1时，我们可以把该问题分解成下面的三个子问题：
 - 把 $n-1$ 个圆盘从柱子A借助柱子B移到柱子C
 - 把第 n 个圆盘从柱子A 移到柱子B
 - 把 $n-1$ 个圆盘从柱子C 借助柱子A移到柱子B



```
#include <iostream>
```

```
using namespace std;
```

```
void hanoi(char a, char b, char c, int n) // 把n个圆盘从a柱子借助c  
                                         // 柱子到b柱子
```

```
{ if (n == 1)
```

```
    cout << "1: " << a << "→" << b << endl; //把第1个盘子从a  
                                         //柱子移至b柱子
```

```
else
```

```
{    hanoi(a,c,b,n-1); //把n-1个圆盘从a柱子借助b柱子移至c柱子
```

```
    cout << n << ": " << a << "→" << b << endl; // 把第n个圆盘  
                                         //从a柱子移至b柱子
```

```
    hanoi(c,b,a,n-1); //把n-1个圆盘从c柱子借助a柱子移至b柱子
```

```
}
```

```
}
```



递归与循环的选择

- 对于一些递归定义的问题，用递归函数来解决会显得比循环更为自然和简洁。
- 与循环的不同：
 - 循环是在**同一组变量**上进行重复操作；而递归则是在**不同的变量组**（属于递归函数的不同实例）上进行重复操作。
- 递归的缺陷：
 - 函数调用是需要**开销**，栈空间的大小也会限制**递归的深度**。
 - 递归算法有时会出现**重复计算**。



本章内容

- 基于过程抽象的程序设计
- C++函数
- 标识符的作用域和变量的生存期
- 递归函数
- C++标准库函数
- 函数的进一步讨论
- C++编译预处理命令



C++标准库函数

- 定义了一些语言本身没有提供的功能。包含了C语言标准库的功能和针对C++提供的新功能。
- 在C++标准库中，根据功能对定义的程序实体进行了分类，把每一类程序实体的声明分别放在一个头文件中。
- 在C++中，把从C语言保留下来的库函数，
 - 重新定义在名空间std中；
 - 对相应的头文件进了重新命名：*.h -> c*



本章内容

- 基于过程抽象的程序设计
- C++函数
- 标识符的作用域和变量的生存期
- 递归函数
- C++标准库函数
- 函数的进一步讨论
- C++编译预处理命令



解决小函数的低效问题

- C++提供了两种解决办法：
 - 宏定义
 - 内联函数



宏定义

- #define <宏名>(<参数表>) <文字串>
 - 在编译前， 将<宏名>用<文字串>替换
 - <文字串>中出现的由<参数表>被替换为实际使用的地方提供的参数

```
#define max(a,b) (((a)>(b))?(a):(b))
int main()
{
    int x = 2, y = 3;
    max(x, y);
}
```



内联函数 (1)

- 内联函数的作用是建议编译程序把该函数的函数体展开到调用点。
- 在函数定义中，返回类型之前加上一个关键词 `inline`
 - 例如：

```
inline int max(int a, int b)
{ return a>b?a:b;
}
```



内联函数 (2)

- 内联函数形式上属于函数，它遵循函数的一些规定，如：参数类型检查与转换。
- 使用内联函数时应注意以下几点：
 - 编译程序对内联函数的限制。
 - 内联函数名具有文件作用域。

带缺省值的形式参数 (1)

- 在C++中允许在定义或声明函数时，为函数的某些参数指定默认值。
- 例如，对于下面的函数声明：

```
void print(int value, int base=10);
```

下面的调用：

```
print(28); //28传给value; 10传给base
```

```
print(32,2); //28传给value; 2传给base
```

带缺省值的形式参数 (2)

指定函数参数的默认值时，应注意：

- 有默认值的形参应处于形参表的右部。

例如： `void f(int a, int b=1, int c=0); //OK`

`void f(int a, int b=1, int c); //Error`

- 对参数默认值的指定只在函数声明处有意义。
- 在不同的源文件中，对同一个函数的声明可以对它的同一个参数指定不同的默认值；在同一个源文件中，对同一个函数的声明只能对它的每一个参数指定一次默认值。

// file.cpp

void f(int a, int b, int c) // 函数定义

{ }

// file1.cpp

void f(int a, int b, int c=2); // ok

.....

void f(int a, int b=1, int c=0); // Error, 对参数c指定了两次默认值

.....

f(1,2); // 编译为f(1,2,2);

// file2.cpp

void f(int a, int b=1, int c=0); // ok

.....

f(1); // 编译为f(1,1,0);

f(1,2); // 编译为f(1,2,0);



函数名重载

- 对于一些功能相同、参数类型或个数不同的函数，有时给它们取相同的名字会带来使用上的方便。

- 例如，把下面的函数：

```
void print_int(int i) { ..... }
```

```
void print_double(double d) { ..... }
```

定义为：

```
void print(int i) { ..... }
```

```
void print(double d) { ..... }
```



对重载函数调用的绑定

- 确定一个对重载函数的调用对应着哪一个重载函数定义的过程称为绑定。
- 对重载函数调用的绑定由**编译程序根据实参与形参的匹配情况(与函数返回类型无关)**来决定：
 - 精确匹配
 - 提升匹配
 - 标准转换匹配
 - 自定义转换匹配
 - 匹配失败



精确匹配 (1)

- 类型相同
- 对实参进行“微小”的类型转换：
 - 数组变量名->数组首地址
函数名->函数首地址 等等



精确匹配 (2)

- 例如，对于下面的重载函数定义：

```
void print(int);
```

```
void print(double);
```

```
void print(char);
```

下面的函数调用：

```
print(1); 绑定到函数： void print(int);
```

```
print(1.0); 绑定到函数： void print(double);
```

```
print('a'); 绑定到函数： void print(char);
```



提升匹配 (1)

- 按整型提升规则P38提升实参类型
- 把float类型实参提升到double; 把double类型实参提升到long double



提升匹配 (2)

- 例如，对于下述的重载函数：

```
void print(int);
```

```
void print(double);
```

根据提升匹配，下面的函数调用：

```
print('a'); 绑定到函数： void print(int);
```

```
print(1.0f); 绑定到函数： void print(double);
```



标准转换匹配 (1)

- 任何算术类型P24可以互相转换
- 枚举类型可以转换成任何算术类型
- 零可以转换成任何算术类型或指针类型
- 任何类型的指针可以转换成void*
- 派生类指针可以转换成基类指针
- 每个标准转换都是平等的。



标准转换匹配 (2)

- 例如，对于下述的重载函数：

`void print(char);`

`void print(char *);`

根据标准转换匹配，下面的函数调用：

`print(1);` 绑定到函数 `void print(char);`

`print(0);` 根据规则(3)绑定失败；



自定义转换匹配 (1)

- 例如，对于下述的重载函数：

```
void print(char);
```

```
void print(double);
```

下面函数调用

`print(1);` 精确匹配、提升匹配以及标准匹配均不成功，则自定义转换匹配：

```
print((char)1)或者print((double)1)
```

重载函数绑定与返回类型无关

- 例如，对于下述的重载函数：

```
int pow(int, int)
```

```
double pow(double, double)
```

下面函数调用

```
double d2 = pow(2.0, 2); // 绑定失败，使用  
自定义转换匹配，pow(int(2.0), 2) 或者  
pow(2.0, double(2))
```




本章内容

- 基于过程抽象的程序设计
- C++函数
- 标识符的作用域和变量的生存期
- 递归函数
- C++标准库函数
- 函数的进一步讨论
- C++编译预处理命令



C++ 编译预处理命令 P377

- C++ 程序中 can 写一些供编译程序使用的编译预处理命令，来对编译过程给出指导，其功能由编译预处理系统来完成。
- 三种编译预处理命令：
 - 文件包含命令（#include）
 - 宏定义（#define）命令
 - 条件编译命令

宏定义 (1)

○ 宏定义格式

- #define <宏名> <文字串>
- #define <宏名>(<参数表>) <文字串>
- #define <宏名>
- #undef <宏名>

○ 宏定义是在编译之前的文字替换!

宏定义 (2)

- 宏定义后面不加;
- 宏名与参数表之间不加空格
- 宏定义要注意是否加括号。

● 例如:

```
#define max(a,b) a>b ? a:b
```

10 + max(x,y) + z 将被替换成:

10 + x>y ? x:y + z 等价于

((10+x)>y) ? x:(y+z)

宏定义 (3)

- 宏定义的缺点

- 重复计算

例如：

```
#define max(a,b) (((a)>(b))?(a):(b))
```

max(x+1,y*2)将被替换成：

```
(((x+1)>(y*2))?(x+1):(y*2))
```

- 不进行参数类型检查和转换
- 不利于一些工具对程序的处理



条件编译命令的常用格式 (1)

#ifdef / #ifndef <宏名>

<程序段1>

[#else

<程序段2>]

#endif



条件编译命令的常用格式 (2)

```
#if <常量表达式1> / #ifdef <宏名> / #ifndef <宏名>
    <程序段1>
#elif <常量表达式2>
    <程序段2>
.....
#elif <常量表达式n>
    <程序段n>
[#else
    <程序段n+1>]
#endif
```



条件编译的作用

- 条件编译中的宏名可以在程序上用`#define`定义，也可以在编译器的选项中给出
- 条件编译的作用：
 - 基于多环境的程序编译
 - 程序调试
 -



基于多环境的程序编译

```
#ifdef UNIX
```

```
..... //适合于UNIX环境的代码
```

```
#elif WINDOWS
```

```
..... //适合于WINDOWS环境的代码
```

```
#else
```

```
..... //适合于其它环境的代码
```

```
#endif
```

```
..... //适合于各种环境的代码
```



程序调试 (1)

```
#ifdef DEBUG  
..... //调试信息  
#endif
```

程序调试 (2)

- 利用标准库中定义的宏：assert

```
#include <cassert> //或<assert.h>
```

```
.....
```

```
assert(x == 1); //断言
```

- assert的定义：

```
.....
```

```
#ifdef NDEBUG
```

```
#define assert(exp) ((void)0)
```

```
#else
```

```
#define assert(exp) ((exp)?(void)0:<输出诊断信息并调  
    用库函数abort>)
```

```
#endif
```