第2篇

面向过程的程序设计

第3章 程序设计初步
第4章 函数与预处理
第5章 数组
第6章 指针
第7章 自定义数据类型

第3章 程序设计初步

3.1 面向过程的程序设计和算法

在面向过程的程序设计中，程序设计者必须指定计算机执行的具体步骤，程序设计者不仅要考虑程序要“做什么”，还要解决“怎么做”的问题，根据程序要“做什么”的要求，写出一个个语句，安排好它们的执行顺序。怎样设计这些步骤，怎样保证它的正确性和具有较高的效率，这就是算法需要解决的问题。

3.1.1 算法的概念

一个面向过程的程序应包括以下两方面内容：

(1) 对数据的描述。在程序中要指定数据的类型和数据的组织形式，即数据结构**(data structure)**。

(2) 对操作的描述。即操作步骤，也就是算法**(algorithm)**。

对于面向过程的程序，可以用下面的公式表示：

程序=算法+数据结构

作为程序设计人员，必须认真考虑和设计数据结构和操作步骤(即算法)。

算法是处理问题的一系列的步骤。算法必须具体地指出在执行时每一步应当怎样做。

不要认为只有“计算”的问题才有算法。广义地说，为解决一个问题而采取的方法和步骤，就称为“算法”。

计算机算法可分为两大类别：数值算法和非数值算法。数值算法的目的是求数值解。非数值算法包括的面十分广泛，最常见的是用于事务管理领域。目前，计算机在非数值方面的应用远远超过了在数值方面的应用。

C++既支持面向过程的程序设计，又支持面向对象的程序设计。无论面向过程的程序设计还是面向对象的程序设计，都离不开算法设计。

3.1.2 算法的表示

1. 自然语言

用中文或英文等自然语言描述算法。但容易产生歧义性，在程序设计中一般不用自然语言表示算法。

2. 流程图

可以用传统的流程图或结构化流程图。用图的形式表示算法，比较形象直观，但修改算法时显得不大方便。

3. 伪代码(pseudo code)

伪代码是用介于自然语言和计算机语言之间的文字和符号来描述算法。如

```
if x is positive then
    print x
else
    print-x
```

用伪代码写算法并无固定的、严格的语法规则，只需把意思表达清楚，并且书写的格式要写成清晰易读的形式。它不用图形符号，因此书写方便、格式紧凑，容易修改，便于向计算机语言算法(即程序)过渡。

4. 用计算机语言表示算法

用一种计算机语言去描述算法，这就是计算机程序。

3.2 C++程序和语句

由第1章已知，一个程序包含一个或多个程序单位(每个程序单位构成一个程序文件)。每一个程序单位由以下几个部分组成：

(1) 预处理命令。如#include命令和#define命令。

(2) 声明部分。例如对数据类型和函数的声明，以及对变量的定义。

(3) 函数。包括函数首部和函数体，在函数体中可以包含若干声明语句和执行语句。

如下面是一个完整的C++程序：

```
#include <iostream>          //预处理命令
using namespace std;         //在函数之外的声明部分
int a=3;                     //在函数之外的声明部分
int main( )                  //函数首部
{ float b;                   //函数内的声明部分
    b=4.5;                   //执行语句
    cout<<a<<b;              //执行语句
    return 0;                //执行语句
}
```

如果一个变量在函数之外进行声明，此变量是全局变量，它的有效范围是从该行开始到本程序单位结束。如果一个变量在函数内声明，此变量是局部变量，它的有效范围是从该行开始到本函数结束。

C++程序结构可以用图3．1表示。

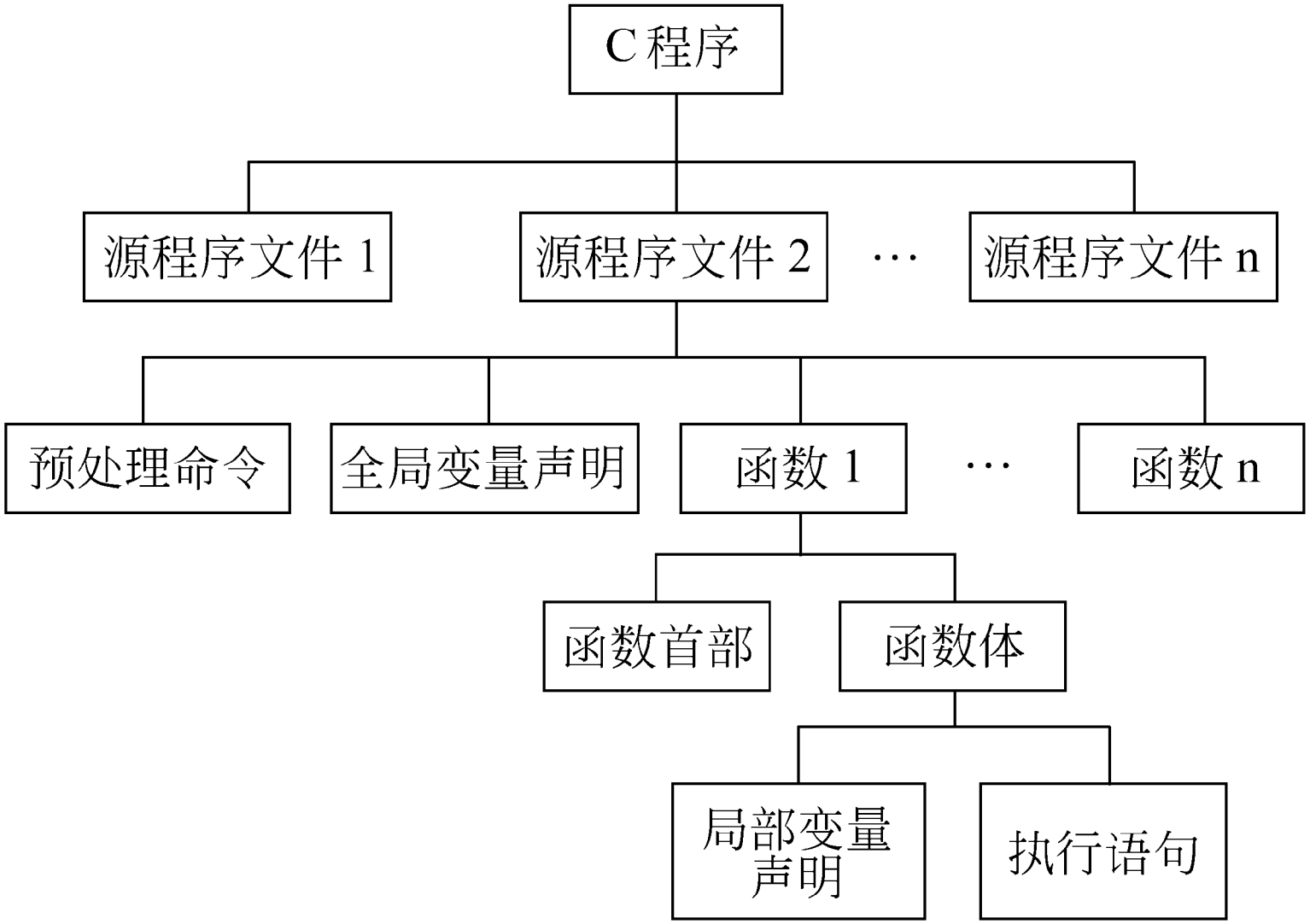
C 程序
源程序文件 1
源程序文件 2
…
源程序文件 n
预处理命令
全局变量声明
函数 1
…
函数 n
函数首部
函数体
局部变量声明
执行语句

图3.1

程序应该包括数据描述（由声明语句来实现）和数据操作（由执行语句来实现）。数据描述主要包括数据类型的声明、函数和变量的定义、变量的初始化等。数据操作的任务是对已提供的数据进行加工。

C++程序中最小的独立单位是语句**(statement)**。它相当于一篇文章中的一个句子。句子是用句号结束的。语句一般是用分号结束的**(**复合语句是以右花括号结束的**)**。

C++语句可以分为以下**4**种：

1. 声明语句

如**int a,b;**在**C**语言中，只有产生实际操作的才称为语句，对变量的定义不作为语句，而且要求对变量的定义必须出现在本块中所有程序语句之前。因此**C**程序员已经养成了一个习惯： 在函数或块的开头位置定义全部变量。在**C++**中，对变量(以及其他对象)的定义被认为是一条语句，并且可以出现在函数中的任何行，即可以放在其他程序语句可以出现的地方，也可以放在函数之外。这样更加灵活，可以很方便地实现变量的局部化(变量的作用范围从声明语句开始到本函数或本块结束)。

2. 执行语句

通知计算机完成一定的操作。执行语句包括：

(1) 控制语句，完成一定的控制功能。**C++**有**9**种控制语句，即

① **if( )~else~**　　（条件语句）
② **for( )~**　　（循环语句）
③ **while( )~**　　（循环语句）
④ **do~while**（ ）　　（循环语句）
⑤ **continue**　　（结束本次循环语句）
⑥ **break**　　（中止执行ｓｗｉｔｃｈ或循环语句）
⑦ **switch**　　（多分支选择语句）
⑧ **goto**　　（转向语句）
⑨ **return**　　（从函数返回语句）

(2) 函数和流对象调用语句。函数调用语句由一次函数调用加一个分号构成一个语句，例如

```
sort(x,y,z);                    //假设已定义了sort函数，它有3个参数
cout<<x<<endl;                  //流对象调用语句
```

(3) 表达式语句。由一个表达式加一个分号构成一个语句。最典型的是： 由赋值表达式构成一个赋值语句。

```
i=i+1                   //是一个赋值表达式
i=i+1;                  //是一个赋值语句
```

任何一个表达式的最后加一个分号都可以成为一个语句。一个语句必须在最后出现分号。

表达式能构成语句是**C**和**C++**语言的一个重要特色。**C++**程序中大多数语句是表达式语句（包括函数调用语句）。

3. 空语句

下面是一个空语句：

;

即只有一个分号的语句，它什么也不做。有时用来做被转向点，或循环语句中的循环体。

4. 复合语句

可以用{ }把一些语句括起来成为复合语句。如下面是一个复合语句。

```
{ z=x+y;
if(z>100) z=z-100;
cout<<z;
}
```

注意：复合语句中最后一个语句中最后的分号不能省略。

在本章中将介绍几种顺序执行的语句，在执行这些语句的过程中不会发生流程的控制转移。

3.3 赋值语句

前面已介绍，赋值语句是由赋值表达式加上一个分号构成。

(1)C++的赋值语句具有其他高级语言的赋值语句的功能。但不同的是： **C++**中的赋值号“＝”是一个运算符，可以写成

a=b=c=d;

而在其他大多数语言中赋值号不是运算符，上面的写法是不合法的。

(2) 关于赋值表达式与赋值语句的概念。在**C++**中，赋值表达式可以包括在其他表达式之中，例如

```
if((a=b)>0) cout<<"a>0"<<endl;
```

按语法规定**if**后面的**( )**内是一个条件。现在在**x**的位置上换上一个赋值表达式**“a=b”**，其作用是：先进行赋值运算（将**b**的值赋给**a**），然后判断**a**是否大于**0**，如大于**0**，执行**cout<<"a>0"<<endl;** 。在**if**语句中的**“a=b”**不是赋值语句而是赋值表达式，这样写是合法的。不能写成

```
if((a=b;)>0) cout<<"a>0"<<endl;
```

因为在**if**的条件中不能包含赋值语句。**C++**把赋值语句和赋值表达式区别开来，增加了表达式的种类，能实现其他语言中难以实现的功能。

3.4 C++的输入与输出

输入和输出并不是C++语言中的正式组成成分。C和C++本身都没有为输入和输出提供专门的语句结构。输入输出不是由C++本身定义的，而是在编译系统提供的I/O库中定义的。

C++的输出和输入是用“流”(stream)的方式实现的。图3.2和图3.3表示C++通过流进行输入输出的过程。

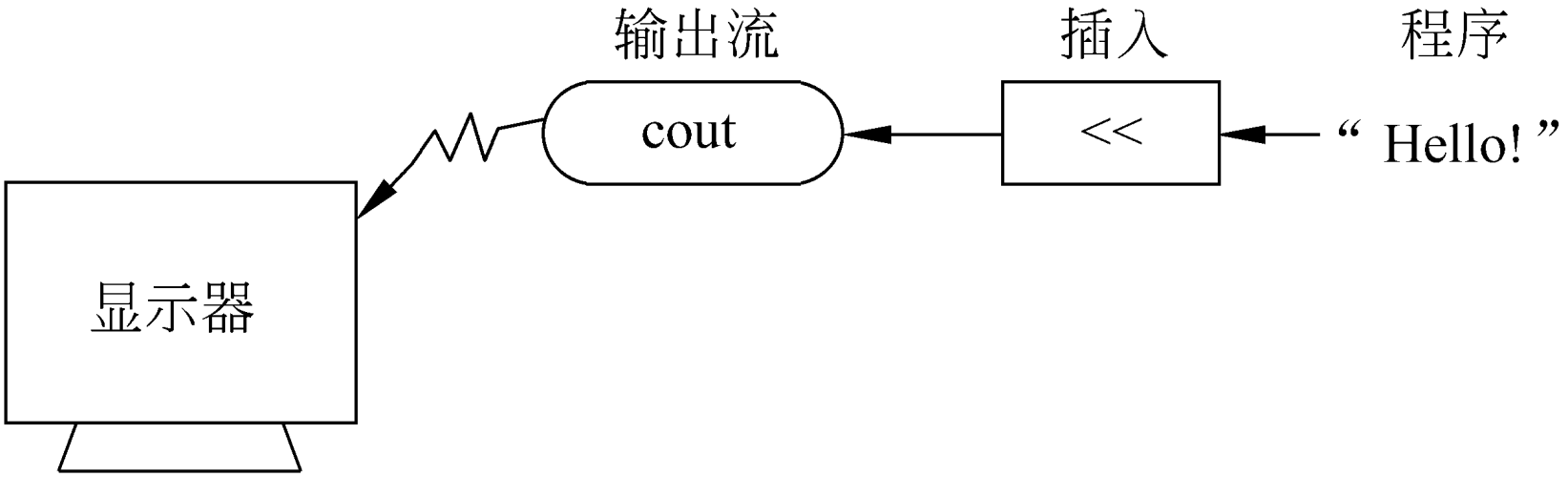

图3.2

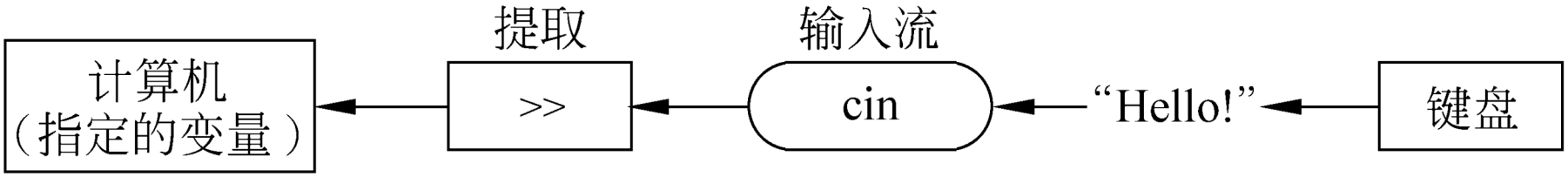

图3.3

有关流对象**cin**、**cout**和流运算符的定义等信息是存放在**C++**的输入输出流库中的，因此如果在程序中使用**cin**、**cout**和流运算符，就必须使用预处理命令把头文件**stream**包含到本文件中：

```
#include <iostream>
```

尽管**cin**和**cout**不是**C++**本身提供的语句，但是在不致混淆的情况下，为了叙述方便，常常把由**cin**和流提取运算符“**>>**”实现输入的语句称为输入语句或**cin**语句，把由**cout**和流插入运算符“**<<**”实现输出的语句称为输出语句或**cout**语句。根据**C++**的语法，凡是能实现某种操作而且最后以分号结束的都是语句。

*3.4.1 输入流与输出流的基本操作

cout语句的一般格式为

cout<<表达式1<<表达式2<<……<<表达式n;

cin语句的一般格式为

cin>>变量1>>变量2>>……>>变量n;

在定义流对象时，系统会在内存中开辟一段缓冲区，用来暂存输入输出流的数据。在执行**cout**语句时，先把插入的数据顺序存放在输出缓冲区中，直到输出缓冲区满或遇到**cout**语句中的**endl**(或′\n′，**ends**，**flush**)为止，此时将缓冲区中已有的数据一起输出，并清空缓冲区。输出流中的数据在系统默认的设备(一般为显示器)输出。

一个**cout**语句可以分写成若干行。如

```
cout<<"This is a simple C++ program."<<endl;
```

可以写成

```
cout<<"This is "          //注意行末尾无分号
<<"a C++ "
<<"program."
<<endl;                   //语句最后有分号
```

也可写成多个**cout**语句，即

```
cout<<"This is ";         //语句末尾有分号
cout <<"a C++ ";
cout <<"program.";
cout<<endl;
```

以上**3**种情况的输出均为

This is a simple C++ program.

注意 不能用一个插入运算符“<<”插入多个输出项：

```
cout<<a,b,c;              //错误，不能一次插入多项
cout<<a+b+c;              //正确，这是一个表达式，作为一项
```

在用**cout**输出时，用户不必通知计算机按何种类型输出，系统会自动判别输出数据的类型，使输出的数据按相应的类型输出。如已定义**a**为**int**型，**b**为**float**型，**c**为**char**型，则

```
cout<<a<<' '<<b<<' '<<c<<endl;
```

会以下面的形式输出：

4 345.789 a

与**cout**类似，一个**cin**语句可以分写成若干行。如

```
cin>>a>>b>>c>>d;
```

可以写成

```
cin>>a                    //注意行末尾无分号
   >>b                    //这样写可能看起来清晰些
   >>c
   >>d;
```

也可以写成

```
cin>>a;
cin>>b;
cin>>c;
cin>>d;
```

以上3种情况均可以从键盘输入： **1 2 3 4** ↙

也可以分多行输入数据：

1↙

2 3↙

4↙

在用**cin**输入时，系统也会根据变量的类型从输入流中提取相应长度的字节。如有

```
char c1,c2;
int a;
float b;
cin>>c1>>c2>>a>>b;
```

如果输入

1234 56.78↙

注意： **34**后面应该有空格以便和**56.78**分隔开。也可以按下面格式输入：

1 2 34 56.78↙　　(在**1**和**2**之间有空格)

不能用**cin**语句把空格字符和回车换行符作为字符输入给字符变量，它们将被跳过。如果想将空格字符或回车换行符(或任何其他键盘上的字符)输入给字符变量，可以用**3.4.3**节介绍的**getchar**函数。

在组织输入流数据时，要仔细分析**cin**语句中变量的类型，按照相应的格式输入，否则容易出错。

*3.4.2 在输入流与输出流中使用控制符

上面介绍的是使用**cout**和**cin**时的默认格式。但有时人们在输入输出时有一些特殊的要求，如在输出实数时规定字段宽度，只保留两位小数，数据向左或向右对齐等。**C++**提供了在输入输出流中使用的控制符(有的书中称为操纵符)，见书中表**3.1**。

需要注意的是： 如果使用了控制符，在程序单位的开头除了要加**iostream**头文件外，还要加**iomanip**头文件。

举例： 输出双精度数。

double a=123.456789012345;对a赋初值

(1) cout<<a;输出： 123.456

(2) cout<<setprecision(9)<<a;输出： 123.456789

(3) cout<<setprecision(6);恢复默认格式(精度为6)

(4) cout<< setiosflags(ios∷fixed);输出： 123.456789

(5) cout<<setiosflags(ios∷fixed)<<setprecision(8)<<a;输出： 123.45678901

(6) cout<<setiosflags(ios∷scientific)<<a;输出： 1.234568e+02

(7) cout<<setiosflags(ios∷scientific)<<setprecision(4)<<a; 输出： 1.2346e02

下面是整数输出的例子：

int b=123456;对b赋初值

(1) cout<<b;输出： 123456

(2) cout<<hex<<b; 输出： 1e240

(3) cout<<setiosflags(ios∷uppercase)<<b;输出： 1E240

(4) cout<<setw(10)<<b<<','<<b; 输出： 123456，123456

(5) cout<<setfill('*')<<setw(10)<<b;输出： ** 123456**

(6) cout<<setiosflags(ios∷showpos)<<b;输出： +123456

如果在多个**cout**语句中使用相同的**setw(n)**，并使用**setiosflags(ios∷right)**，可以实现各行数据右对齐，如果指定相同的精度，可以实现上下小数点对齐。

例**3.1** 各行小数点对齐。

```
#include <iostream>
#include <iomanip>
using namespace std;
int main( )
{
double a=123.456,b=3.14159,c=-3214.67;
cout<<setiosflags(ios∷fixed)<<setiosflags(ios∷right)<<setprecision(2);
    cout<<setw(10)<<a<<endl;
    cout<<setw(10)<<b<<endl;
    cout<<setw(10)<<c<<endl;
return 0;
```

输出如下：

123.46 (字段宽度为**10**，右对齐，取两位小数)

3.14

-3214.67

先统一设置定点形式输出、取两位小数、右对齐。这些设置对其后的输出均有效(除非重新设置)，而**setw**只对其后一个输出项有效，因此必须在输出**a,b,c**之前都要写**setw(10)**。

3.4.3 用getchar和putchar 函数进行字符的输入和输出

C++还保留了C语言中用于输入和输出单个字符的函数，使用很方便。其中最常用的有**getchar**函数和**putchar**函数。

1. putchar函数（字符输出函数）

putchar函数的作用是向终端输出一个字符。例如

```
putchar(c);
```

它输出字符变量 c 的值。

例**3.2** 输出单个字符。

```
#include <iostream>        //或者包含头文件stdio.h：  #include <stdio.h>
using namespace std;
int main( )
{char a,b,c;
 a='B';b='O';c='Y';
 putchar(a);putchar(b);putchar(c);putchar('\n');
 putchar(66);putchar(79);putchar(89);putchar(10);
 return 0;
}
```

运行结果为

```
BOY
BOY
```

可以看到： 用**putchar**可以输出转义字符，**putchar('\n')**的作用是输出一个换行符，使输出的当前位置移到下一行的开头。**putchar(66)**的作用是将**66**作为**ASCII**码转换为字符输出，**66**是字母**'B'**的**ASCII**码，因此**putchar(66)**输出字母**'B'**。其余类似。**putchar(10)**中的**10**是换行符的**ASCII**码，**putchar(10)**输出一个换行符，作用与**putchar('\n')**相同。

也可以输出其他转义字符，如

putchar('\101') （输出字符**'A'**，八进制的**101**是**'A'**的**ASCII**码）

putchar('\'') （输出单引号字符'）

putchar('\015') （输出回车，不换行，使输出的当前位置移到本行开头）

2. getchar函数（字符输入函数）

此函数的作用是从终端（或系统隐含指定的输入设备）输入一个字符。**getchar**函数没有参数，其一般形式为**getchar**（ ）函数的值就是从输入设备得到的字符。

例**3.3** 输入单个字符。

```
#include <iostream>
using namespace std;
int main( )
{char c;
 c=getchar( ); putchar(c+32); putchar('\n');
 return 0;
}
```

在运行时，如果从键盘输入大写字母**'A'**并按回车键，就会在屏幕上输出小写字母**'a'**。

请注意，**getchar( )**只能接收一个字符。**getchar**函数得到的字符可以赋给一个字符变量或整型变量，也可以不赋给任何变量，作为表达式的一部分。例如，例**3.3**第**5**行可以用下面一行代替：

putchar（getchar（）+32）；putchar('\n');

因为**getchar( )**读入的值为**'A'**，**'A'+32**是小写字母**'a'**的**ASCII**码，因此**putchar**函数输出**'a'**。此时不必定义变量**c**。

也可用**cout**输出**getchar**函数得到字符的**ASCII**的值：

cout<<getchar( );

这时输出的是整数**97**，因为用**getchar( )**读入的实际上是字符的**ASCII**码，现在并未把它赋给一个字符变量，**cout**就按整数形式输出。如果改成

cout<<(c=getchar( )); //设**c**已定义为字符变量

则输出为字母**'a'**，因为要求输出字符变量**c**的值。

可以看到用**putchar**和**getchar**函数输出和输入字符十分灵活方便，由于它们是函数所以可以出现在表达式中，例如

cout<<(c=getchar( )+32);

3.4.4 用scanf和printf函数进行输入和输出

在C语言中是用printf函数进行输出，用scanf函数进行输入的。C++保留了C语言的这一用法。在此只作很简单的介绍。

scanf函数一般格式是

scanf(格式控制，输出表列)

printf函数的一般格式是

printf(格式控制，输出表列)

例3.4 用scanf和printf函数进行输入和输出。

```
#include <iostream>
using namespace std;
int main( )
{int a; float b; char c;
 scanf("%d %c %f",&a,&c,&b);          //注意在变量名前要加地址运算符&
 printf("a=%d,b=%f,c=%c \n",a,b,c);
 return 0;
}
```

运行情况如下：

```
12 A 67.98↙(本行为输入，输入的3个数据间以空格相间)
a=12,b=67.980003,c=A(本行为输出)
```

输入的整数12送给整型变量a，字符'A'送给字符变量c，67.98送给单精度变量b。

3.5 编写顺序结构的程序

例3.5 求一元二次方程式$ax^2+bx+c=0$的根。a,b,c的值在运行时由键盘输入，它们的值满足$b^2-4ac \geq 0$。根据求x1,x2的算法。它可以编写出以下C++程序：

```
#include <iostream>
#include <cmath>          //由于程序要用到数学函数sqrt，故应包含头文件cmath
using namespace std;
int main( )
{float  a,b,c,x1,x2;
 cin>>a>>b>>c;
 x1=(-b+sqrt(b*b-4*a*c))/(2*a);
 x2=(-b-sqrt(b*b-4*a*c))/(2*a);
 cout<<"x1="<<x1<<endl;
 cout<<"x2="<<x2<<endl;
 return 0;
}
```

运行情况如下：

4.5 8.8 2.4 ↙

x1=-0.327612

x2=-1.17794

如果程序中要用到数学函数，都要包含头文件**cmath**(也可以用老形式的头文件**math.h**，但提倡使用**C++**新形式的头文件，请参阅第**14**章**14.3**节)。在写程序时，一定要注意将数学表达式正确地转换成合法的**C++**表达式。

可以看到： 顺序结构的程序中的各执行语句是顺序执行的。这种程序最简单，最容易理解。

3.6 关系运算和逻辑运算

往往要求根据某个指定的条件是否满足来决定执行的内容。例如，购物在**1000**元以下的打九五折，**1000**元及以上的打九折。

C++提供**if**语句来实现这种条件选择。如

```
if amount<1000 tax=0.95;      //amount代表购物总额，tax代表折扣
else tax=0.9;      //若amount<1000，条件满足，tax=0.95，否则tax=0.9
pay=amount*tax;          //pay为实付款
```

流程可以用图**3.4**表示。

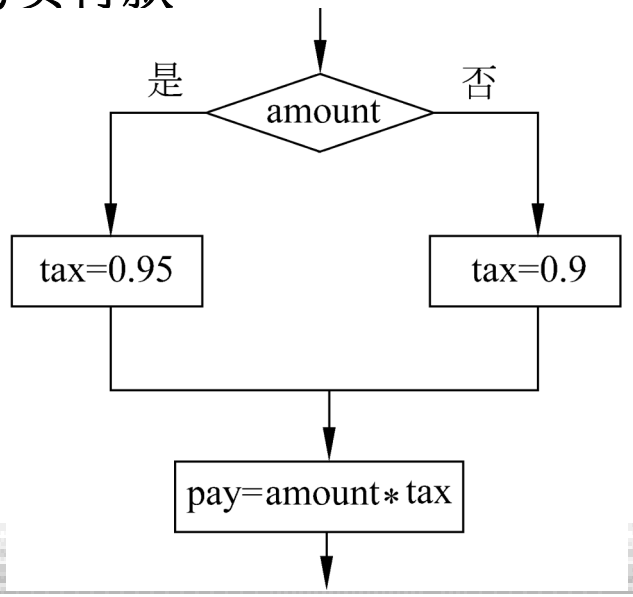

图**3.4**

3.6.1 关系运算和关系表达式

上面**if**语句中的**“amount<1000”**实现的不是算术运算，而是关系运算。实际上是比较运算，将两个数据进行比较，判断比较的结果。**“amount<1000”**就是一个比较式，在高级语言中称为关系表达式，其中**“>”**是一个比较符，称为关系运算符。

C++的关系运算符有：

① <	(小于)	优先级相同 (高)
② <=	(小于或等于)	
③ >	(大于)	
④ >=	(大于或等于)	
⑤ ==	(等于)	优先级相同 (低)
⑥ ！ =	(不等于)	

关于优先次序：

① 前4种关系运算符（<，<=，>，>=）的优先级别相同，后两种也相同。前4种高于后两种。例如，“>”优先于“==”。而“>”与“<”优先级相同。

② 关系运算符的优先级低于算术运算符。

③ 关系运算符的优先级高于赋值运算符。

例如：

c>a+b 等效于 **c>(a+b)**

a>b==c 等效于**(a>b)==c**

a==b<c 等效于**a==(b<c)**

a=b>c 等效于**a=(b>c)**

用关系运算符将两个表达式连接起来的式子，称为关系表达式。关系表达式的一般形式可以表示为

表达式 关系运算符 表达式

其中的“表达式”可以是算术表达式或关系表达式、逻辑表达式、赋值表达式、字符表达式。例如，下面都是合法的关系表达式：

a>b, a+b>b+c，(a==3)>(b==5), 'a'<'b', (a>b)>(b<c)

关系表达式的值是一个逻辑值，即“真”或“假”。例如，关系表达式“**5==3**”的值为“假”，“**5>=0**”的值为“真”。在**C**和**C++**中都用数值**1**代表“真”，用**0**代表“假”。

如果有以下赋值表达式：

d=a>b　　则**d**得到的值为**1**

f=a>b>c　　**f**得到的值为**0**

3.6.2 逻辑常量和逻辑变量

C语言没有提供逻辑型数据，关系表达式的值(真或假)分别用数值1和0代表。C++增加了逻辑型数据。逻辑型常量只有两个，即false(假)和true(真)。

逻辑型变量要用类型标识符bool来定义，它的值只能是true和false之一。如

```
bool found,flag=false;      //定义逻辑变量found和flag，并使flag的初值为false
found=true;                 //将逻辑常量true赋给逻辑变量found
```

由于逻辑变量是用关键字bool来定义的，因此又称为布尔变量。逻辑型常量又称为布尔常量。所谓逻辑型，就是布尔型。

设立逻辑类型的目的是为了看程序时直观易懂。

在编译系统处理逻辑型数据时，将**false**处理为**0**，将**true**处理为**1**。因此，逻辑型数据可以与数值型数据进行算术运算。

如果将一个非零的整数赋给逻辑型变量，则按“真”处理，如

```
flag=123;                    //赋值后flag的值为true
cout<<flag;
```

输出为数值**1**。

3.6.3 逻辑运算和逻辑表达式

有时只用一个关系表达式还不能正确表示所指定的条件。

C++提供3种逻辑运算符：

(1) && 逻辑与 **(相当于其他语言中的AND)**

(2) || 逻辑或 **(相当于其他语言中的OR）**

(3) ! 逻辑非 **(相当于其他语言中的NOT)**

逻辑运算举例如下：

a && b 若**a,b**为真，则**a && b**为真。

a||b 若**a,b**之一为真，则**a||b**为真。

!a 若**a**为真，则**!a**为假。

书中表3．2为逻辑运算的“真值表”。

在一个逻辑表达式中如果包含多个逻辑运算符，按以下的优先次序：

(1) !（非）→ **&&**（与）→ **||**（或），即**“!”**为三者中最高的。

(2) 逻辑运算符中的**“&&”**和**“||”**低于关系运算符，**“!”**高于算术运算符。

例如：

(a>b) && (x>y) 可写成 **a>b && x>y**

(a==b) || (x==y) 可写成 **a==b || x==y**

(!a) || (a>b) 可写成 **!a || a>b**

将两个关系表达式用逻辑运算符连接起来就成为一个逻辑表达式，上面几个式子就是逻辑表达式。逻辑表达式的一般形式可以表示为

表达式 逻辑运算符 表达式

逻辑表达式的值是一个逻辑量“真”或“假”。前面已说明，在给出逻辑运算结果时，以数值1代表“真”，以0代表“假”，但在判断一个逻辑量是否为“真”时，采取的标准是： 如果其值是0就认为是“假”，如果其值是非0就认为是“真”。例如：

(1) 若a=4，则!a的值为0。因为a的值为非0，被认作“真”，对它进行“非”运算，得“假”，“假”以0代表。

(2) 若a=4,b=5，则a && b的值为1。因为a和b均为非0，被认为是“真” 。

(3) a,b值同前，a-b||a+b的值为1。因为a-b和a+b的值都为非零值。

(4) a,b值同前，!a || b的值为1。

(5) 4 && 0 || 2 的值为1。

在C++中，整型数据可以出现在逻辑表达式中，在进行逻辑运算时，根据整型数据的值是0或非0，把它作为逻辑量假或真，然后参加逻辑运算。

通过这几个例子可以看出： 逻辑运算结果不是0就是1，不可能是其他数值。而在逻辑表达式中作为参加逻辑运算的运算对象可以是0（“假”）或任何非0的数值（按“真”对待）。如果在一个表达式中的不同位置上出现数值，应区分哪些是作为数值运算或关系运算的对象，哪些作为逻辑运算的对象。

实际上，逻辑运算符两侧的表达式不但可以是关系表达式或整数(0和非0)，也可以是任何类型的数据，如字符型、浮点型或指针型等。系统最终以0和非0来判定它们属于“真”或“假”。例如'c' && 'd'的值为1。

可以将表3.2改写成书中表3.3形式。

熟练掌握C++的关系运算符和逻辑运算符后，可以巧妙地用一个逻辑表达式来表示一个复杂的条件。例如，要判别某一年(year)是否为闰年。闰年的条件是符合下面两者之一：①能被4整除，但不能被100整除。②能被100整除，又能被400整除。例如2004、2000年是闰年，2005、2100年不是闰年。

可以用一个逻辑表达式来表示：

```
(year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

当给定year为某一整数值时，如果上述表达式值为真(1)，则year为闰年；否则year为非闰年。可以加一个“!”用来判别非闰年：

```
!((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
```

若表达式值为真(1),year为非闰年。也可以用下面的逻辑表达式判别非闰年：

(year % 4 != 0) || (year % 100 == 0 && year % 400 !=0)

若表达式值为真，year为非闰年。请注意表达式中右面的括号内的不同运算符（%,!,&&,==）的运算优先次序。

3.7 选择结构和 i f 语句

if语句是用来判定所给定的条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。

3.7.1 if语句的3种形式

1. if（表达式）语句

例如：

```
if(x>y) cout<<x<<endl;
```

这种if语句的执行过程见图3.5(a)。

2. if（表达式）语句1 else 语句2

例如：

```
if (x>y) cout<<x;
else  cout<<y;
```

见图3.5(b）。

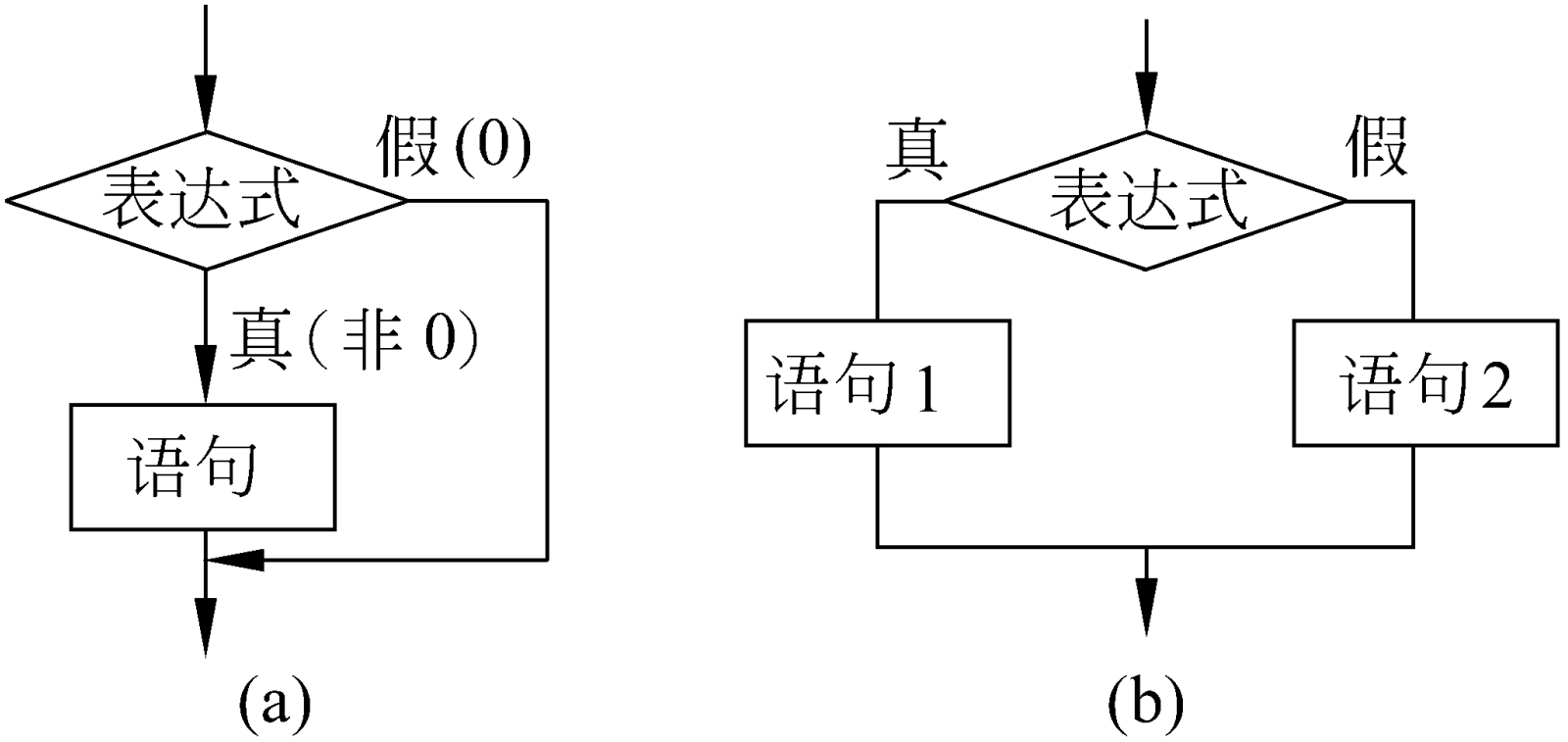

图3.5

3. if(表达式1) 语句1

else if(表达式2) 语句2

else if(表达式3) 语句3

…

else if(表达式m) 语句m

else 语句n流程图见图3.6。

例如：

```
if    (number>500) cost=0.15;
else if(number>300) cost=0.10;
else if(number>100) cost=0.075;
else if(number>50) cost=0.05;
else cost=0;
```

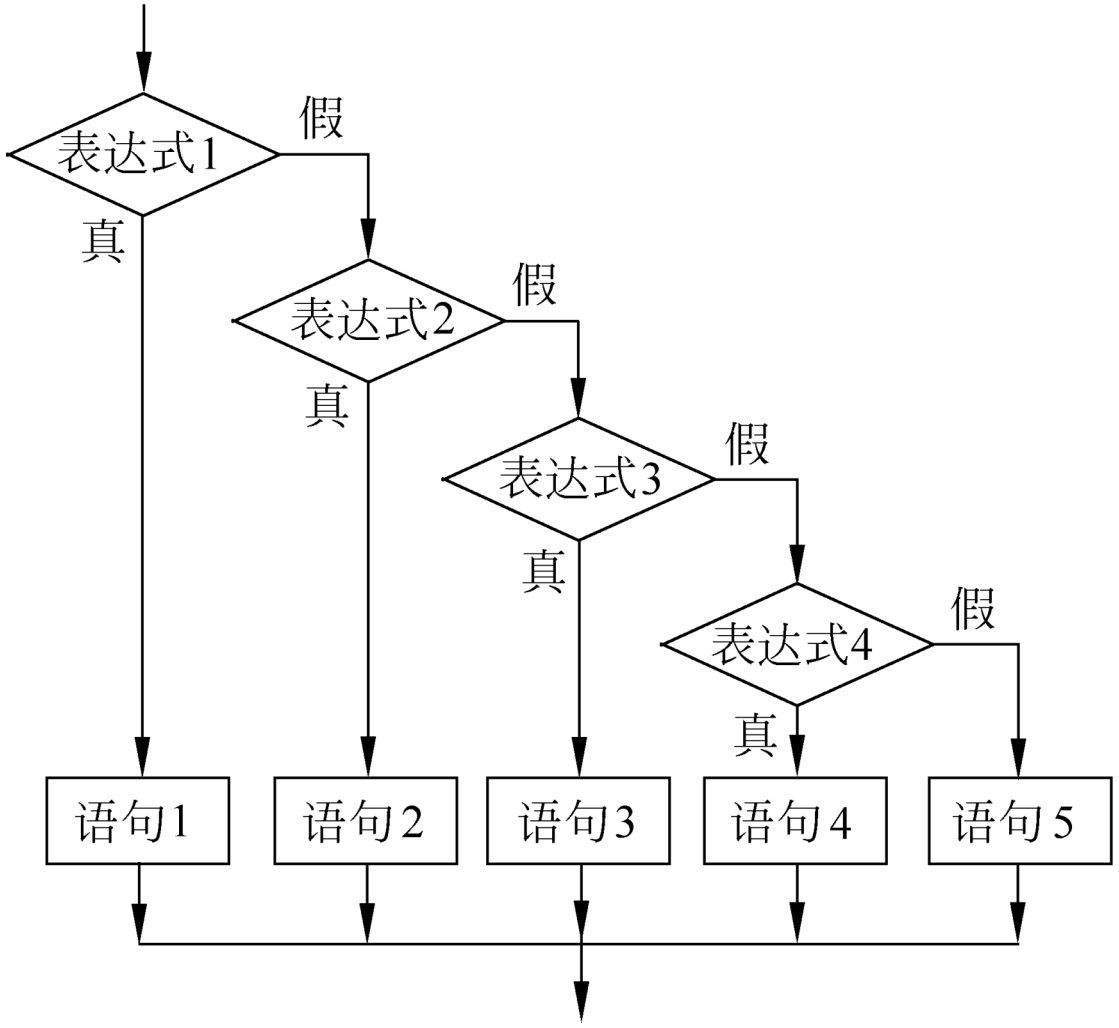

图3.6

说明：

(1) 从图**3.5**和图**3.6**可以看到： **3**种形式的**if**语句都是由一个入口进来，经过对“表达式”的判断，分别执行相应的语句，最后归到一个共同的出口。这种形式的程序结构称为选择结构。在**C++**中**if**语句是实现选择结构主要的语句。

(2) 3种形式的**if**语句中在**if**后面都有一个用括号括起来的表达式，它是程序编写者要求程序判断的“条件”，一般是逻辑表达式或关系表达式。

(3) 第**2**、第**3**种形式的**if**语句中，在每个**else**前面有一分号，整个语句结束处有一分号。

(4) 在**if**和**else**后面可以只含一个内嵌的操作语句（如上例），也可以有多个操作语句，此时用花括号“｛｝”将几个语句括起来成为一个复合语句。

例3.6 求三角形的面积。

```
#include <iostream>
#include <cmath>                //使用数学函数时要包含头文件cmath
#include <iomanip>              //使用I/O流控制符要包含头文件iomanip
using namespace std;
int main( )
{
 double a,b,c;
 cout<<"please enter a,b,c：";
 cin>>a>>b>>c;
 if (a+b>c && b+c>a && c+a>b)
  {                             //复合语句开始
   double s,area;               //在复合语句内定义变量
   s=(a+b+c)/2;
   area=sqrt(s*(s-a)*(s-b)*(s-c));
   cout<<setiosflags(ios::fixed)<<setprecision(4); //指定输出的数包含4位小数
   cout<<"area="<<area<<endl;   //在复合语句内输出局部变量的值
  }                             //复合语句结束
else cout<<"it is not a trilateral!"<<endl;
return 0;
}
```

运行情况如下：

please enter a,b,c：2.45 3.67 4.89↙

area=4.3565

变量**s**和**area**只在复合语句内用得到，因此在复合语句内定义，它的作用范围为从定义变量开始到复合语句结束。如果在复合语句外使用**s**和**area**，则会在编译时出错，系统认为这两个变量未经定义。将某些变量局限在某一范围内，与外界隔离，可以避免在其他地方被误调用。

3.7.2 if语句的嵌套

在**if**语句中又包含一个或多个**if**语句称为**if**语句的嵌套。一般形式如下：

if( )

if( )语句**1**

else 语句**2**　　内嵌**if**

else

if( )语句**3**

else 语句**4**　　内嵌**if**

应当注意**if**与**else**的配对关系。**else**总是与它上面最近的、且未配对的**if**配对。假如写成

if( )

if( )语句**1**

else

if( )语句**2**

else 语句**3**

内嵌**if**

编程序者把第一个**else**写在与第一个**if(**外层**if)**同一列上,希望**else**与第一个**if**对应,但实际上**else**是与第二个**if**配对,因为它们相距最近，而且第二个**if**并未与任何**else**配对。为了避免误用，最好使每一层内嵌的**if**语句都包含**else**子句(如本节开头列出的形式),这样**if**的数目和**else**的数目相同,从内层到外层一一对应,不致出错。

如果**if**与**else**的数目不一样,为实现程序设计者的企图,可以加花括号来确定配对关系。例如:

```
if( )
{ if ( ) 语句1}      //这个语句是上一行if语句的内嵌if
else 语句2           //本行与第一个if配对
```

这时{ }限定了内嵌**if**语句的范围，{ }外的**else**不会与{ }内的**if**配对。关系清楚，不易出错。

3.8 条件运算符和条件表达式

若在**if**语句中，当被判别的表达式的值为“真”或“假”时，都执行一个赋值语句且给同一个变量赋值时，可以用简单的条件运算符来处理。例如，若有以下**if**语句：

```
if (a>b) max=a;
else max=b;
```

可以用条件运算符(？：)来处理：

```
max=(a>b)?a：b;
```

其中“(a>b)?a：b”是一个“条件表达式”。它是这样执行的：如果(a>b)条件为真，则条件表达式的值就取“?”后面的值，即条件表达式的值为 a ，否则条件表达式的值为“：”后面的值，即**b**。

条件运算符要求有**3**个操作对象，称三目（元）运算符，它是**C++**中惟一的一个三目运算符。条件表达式的一般形式为

表达式**1**？表达式**2** ：表达式**3**

条件运算符的执行顺序是：先求解表达式**1**，若为非**0**（真）则求解表达式**2**，此时表达式 2 的值就作为整个条件表达式的值。若表达式**1**的值为**0**（假），则求解表达式**3**，表达式**3**的值就是整个条件表达式的值。**“max=(a>b)?a：b”** 的执行结果是将条件表达式的值赋给**max**。也就是将**a**和**b**二者中的大者赋给**max**。条件运算符优先于赋值运算符，因此上面赋值表达式的求解过程是先求解条件表达式，再将它的值赋给**max**。

条件表达式中，表达式1的类型可以与表达式2和表达式3的类型不同。如

x? 'a' : 'b'

如果已定义x为整型变量，若x=0，则条件表达式的值为字符'b'的ASCII码。表达式2和表达式3的类型也可以不同，此时条件表达式的值的类型为二者中较高的类型。如有条件表达式x>y?1：1.5，如果x≤y，则条件表达式的值为1.5，若x>y，值应为1，由于C++把1.5按双精度数处理，双精度的类型比整型高，因此，将1转换成双精度数，以此作为表达式的值。

例3.7 输入一个字符，判别它是否为大写字母，如果是，将它转换成小写字母；如果不是，不转换。然后输出最后得到的字符。

```
#include <iostream>
using namespace std;
int main( )
{
  char ch;
  cin>>ch;
  ch=(ch>='A' && ch<='Z')?(ch+32)：ch;       //判别ch是否大写字母，是则转换
  cout<<ch<<endl;
  return 0;
}
```

3.9 多分支选择结构和switch 语句

switch语句是多分支选择语句，用来实现多分支选择结构。

它的一般形式如下：

```
switch（表达式）
    {case 常量表达式1：语句1
    case 常量表达式2：语句2
     …
    case 常量表达式n：语句n
    default：语句n+1
    }
```

例如，要求按照考试成绩的等级打印出百分制分数段，可以用**switch**语句实现：

```
switch(grade)
    {case 'A':  cout<<"85~100\n";
     case 'B':  cout<<"70~84\n";
     case 'C':  cout<<"60~69\n";
     case 'D':  cout<<"<60\n";
     default  :  cout<<"error\n";
    }
```

说明：

(1) switch后面括号内的“表达式”，允许为任何类型。

(2) 当**switch**表达式的值与某一个**case**子句中的常量表达式的值相匹配时，就执行此**case**子句中的内嵌语句，若所有的**case**子句中的常量表达式的值都不能与**switch**表达式的值匹配，就执行**default**子句的内嵌语句。

(3) 每一个**case**表达式的值必须互不相同，否则就会出现互相矛盾的现象（对表达式的同一个值，有两种或多种执行方案）。

(4) 各个**case**和**default**的出现次序不影响执行结果。例如，可以先出现**“default：…”**，再出现**“case 'D'：…”**，然后是**“case 'A'：…”**。

(5) 执行完一个**case**子句后，流程控制转移到下一个**case**子句继续执行。**“case**常量表达式**”**只是起语句标号作用，并不是在该处进行条件判断。在执行**switch**语句时，根据**switch**表达式的值找到与之匹配的**case**子句，就从此**case**子句开始执行下去，不再进行判断。例如，上面的例子中，若**grade**的值等于**'A'**，则将连续输出：

85~100
70~84
60~69
<60
error

因此，应该在执行一个**case**子句后，使流程跳出**switch**结构，即终止**switch**语句的执行。可以用一个**break**语句来达到此目的。将上面的**switch**结构改写如下：

```
switch(grade)
    {case 'A': cout<<"85~100\n"; break;
     case 'B': cout<<"70~84\n"; break;
     case 'C': cout<<"60~69\n"; break;
     case 'D': cout<<"<60\n"; break;
     default : cout<<"error\n"; break;
    }
```

最后一个子句（**default**）可以不加**break**语句。如果**grade**的值为**'B'**，则只输出**“70~84”**。流程图见图**3.7**。

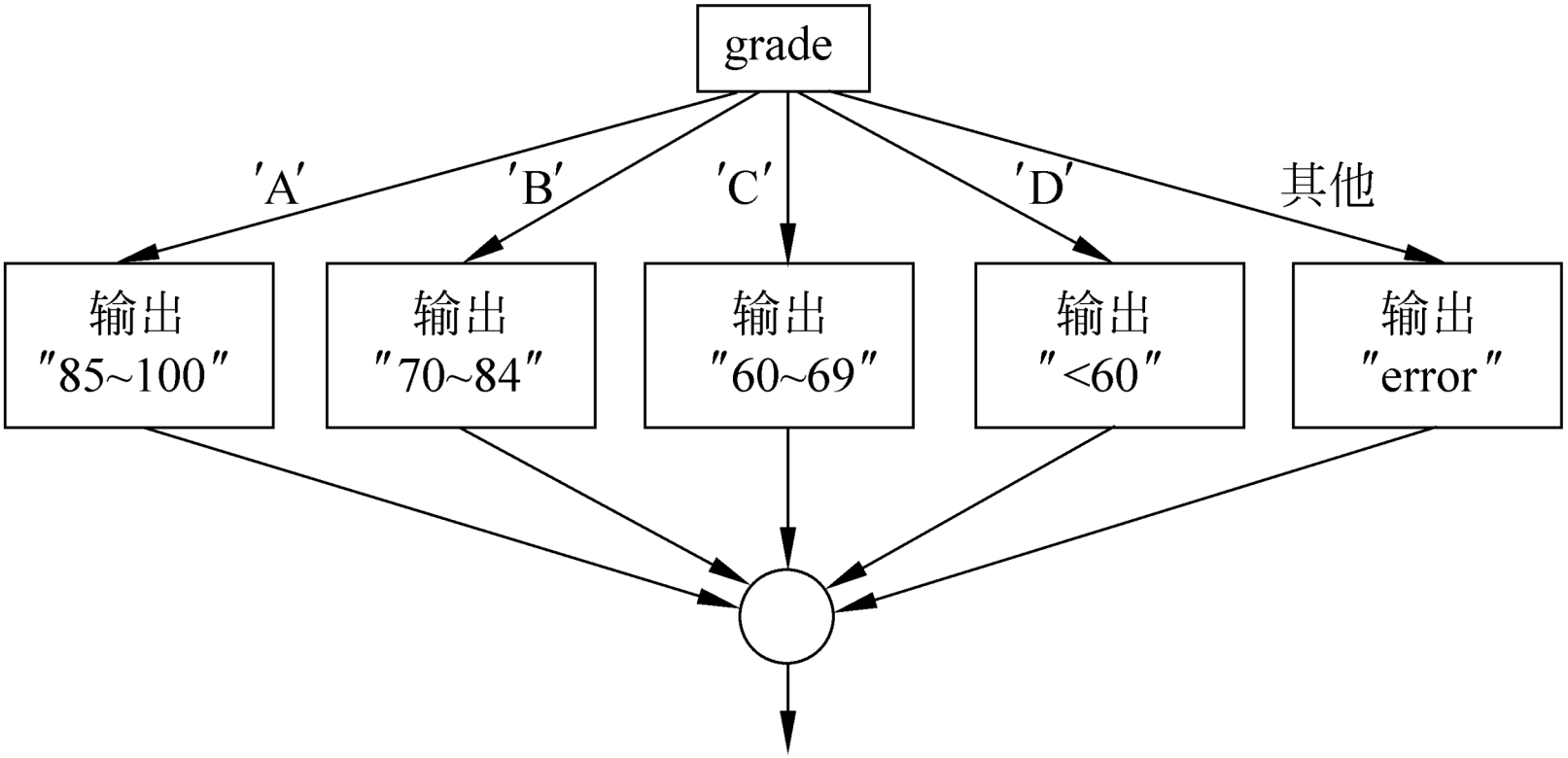

图**3.7**

在**case**子句中虽然包含一个以上执行语句，但可以不必用花括号括起来，会自动顺序执行本**case**子句中所有的执行语句。

(6) 多个**case**可以共用一组执行语句，如

```
...
case 'A':
case 'B':
case 'C':   cout<<">60\n"; break;
...
```

当**grade**的值为**'A'**、**'B'**或**'C'**时都执行同一组语句。

3.10 编写选择结构的程序

例3.8 编写程序，判断某一年是否为闰年。

```
#include <iostream>
using namespace std;
int main( )
{ int year;
  bool leap;
  cout<<"please enter year：";     //输出提示
  cin>>year;                       //输入年份
  if (year%4==0)                   //年份能被4整除
    {if(year%100==0)               //年份能被4整除又能被100整除
      {if (year%400==0)            //年份能被4整除又能被400整除
         leap=true;                //闰年，令leap=true(真)
       else leap=false;}           //非闰年，令leap=false(假)
     else                          //年份能被4整除但不能被100整除肯定是闰年
        leap=true;}                //是闰年，令leap=true
```

```
  else                          //年份不能被4整除肯定不是闰年
    leap=false;                 //若为非闰年，令leap=false
  if (leap)
    cout<<year<<" is ";         //若leap为真，就输出年份和“是”
  else
    cout<<year<<" is not ";     //若leap为真，就输出年份和“不是”
  cout<<" a leap year."<<endl;  //输出“闰年”
  return 0;
}
```

运行情况如下：

① 2005↙

2005 is not a leap year.

② 1900↙

1900 is npt a leap year.

也可以将程序中第**8~16**行改写成以下的**if**语句：

```
if(year%4!=0)
leap=false；
else if(year%100!=0)
leap=true;
else if(year％400!=0)
leap=false；
else
leap=true；
```

也可以用一个逻辑表达式包含所有的闰年条件，将上述**if**语句用下面的**if**语句代替：

```
if((year%4 == 0 && year%100 !=0) || (year%400 == 0)) leap=true;
else leap=false；
```

例3.9 运输公司对用户计算运费。路程(s)越远，每公里运费越低。标准如下：

s＜250km	没有折扣
250≤s＜500	2%折扣
500≤s＜1000	5%折扣
1000≤s＜2000	8%折扣
2000≤s＜3000	10%折扣
3000≤s	15%折扣

设每公里每吨货物的基本运费为p(price的缩写)，货物重为w(wright的缩写)，距离为 s ，折扣为d(discount的缩写)，则总运费f(freight的缩写)的计算公式为

f = p * w * s * (1 - d)

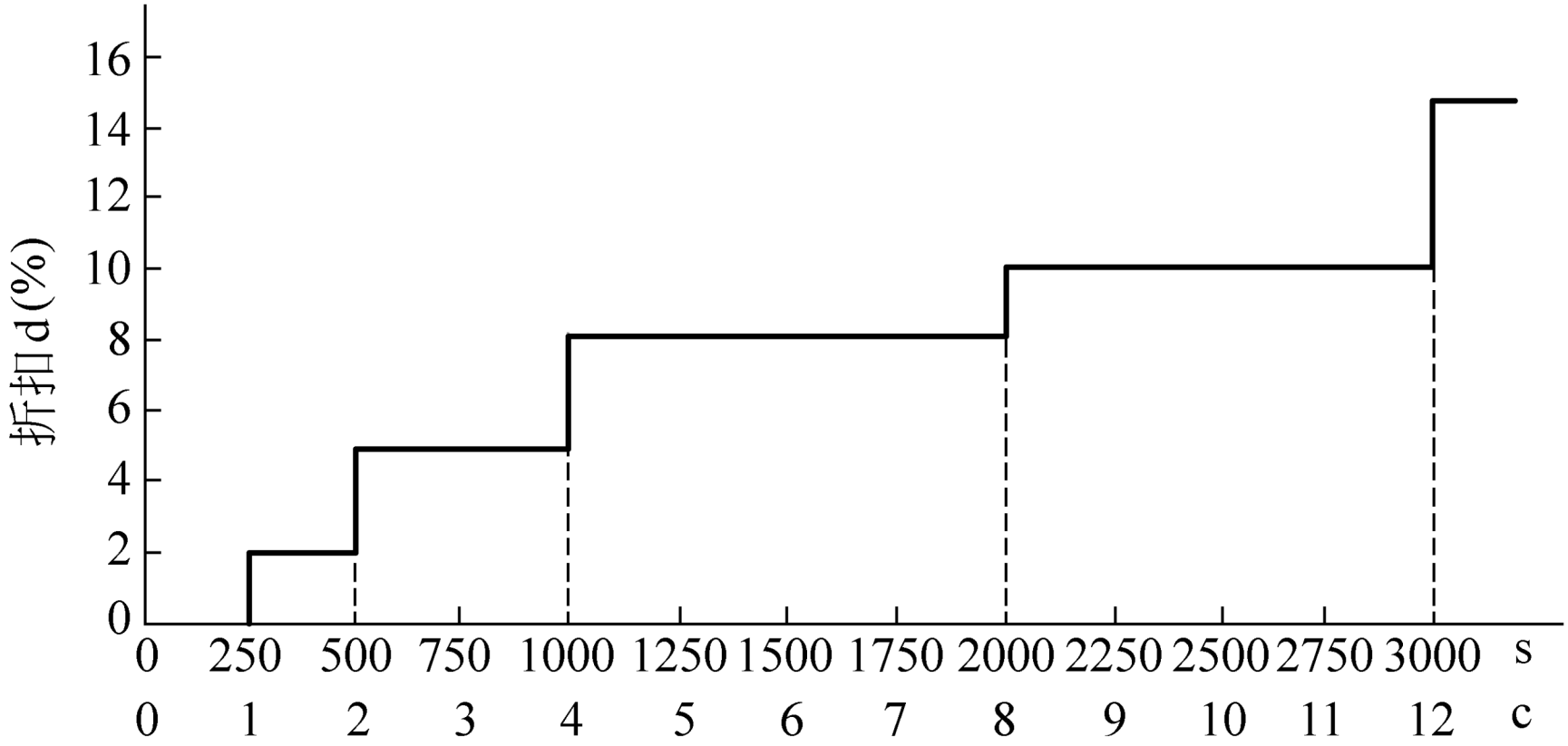
折扣d(%)
16
14
12
10
8
6
4
2
0
0 250 500 750 1000 1250 1500 1750 2000 2250 2500 2750 3000 s
0 1 2 3 4 5 6 7 8 9 10 11 12 c

图3.8

据此写出程序如下：

```
#include <iostream>
using namespace std;
int main( )
{int c,s;
 float p,w,d,f;
 cout<<"please enter p,w,s：";
 cin>>p>>w>>s;
 if(s>=3000) c=12;
 else c=s/250;
 switch (c)
 { case 0：d=0;break;
   case 1：d=2;break;
   case 2：
   case 3：d=5;break;
   case 4：
```

```
    case 5：
    case 6：
    case 7：d=8;break;
    case 8：
    case 9：
    case 10：
    case 11：d=10;break;
    case 12：d=15;break;
  }
   f=p*w*s*(1-d/100.0);
   cout<<"freight="<<f<<endl;
   return 0;
}
```

运行情况如下：

```
please enter p,w,s：100 20 300↙
freight=588000
```

3.11 循环结构和循环语句

在人们所要处理的问题中常常遇到需要反复执行某一操作的情况。这就需要用到循环控制。许多应用程序都包含循环。顺序结构、选择结构和循环结构是结构化程序设计的3种基本结构，是各种复杂程序的基本构造单元。因此程序设计者必须熟练掌握选择结构和循环结构的概念及使用方法。

3.11.1 用while语句构成循环

while语句的一般形式如下：

while (表达式) 语句

其作用是： 当指定的条件为真(表达式为非**0**)时，执行**while**语句中的内嵌语句。其流程图见图**3.9**。其特点是：先判断表达式，后执行语句。**while**循环称为当型循环。

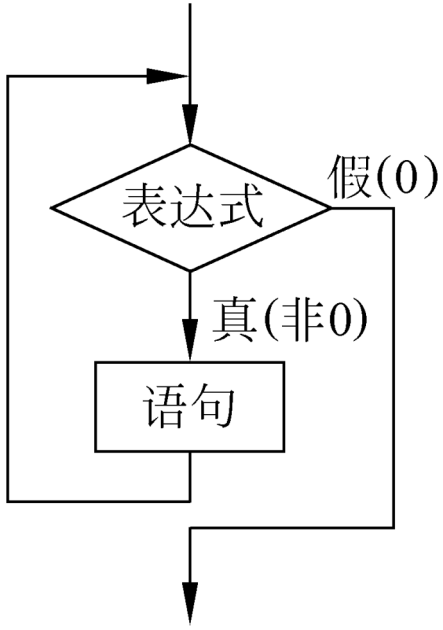

图3.9

例3.10 求1+2+3+…+100。

用流程图表示算法，见图3.10。

根据流程图写出程序：

```
#include <iostream>
using namespace std;
int main( )
{int i=1,sum=0;
   while (i<=100)
    { sum=sum+i;
      i++;
    }
cout<<"sum="<<sum<<endl;
}
```

运行结果为

```
sum=5050
```

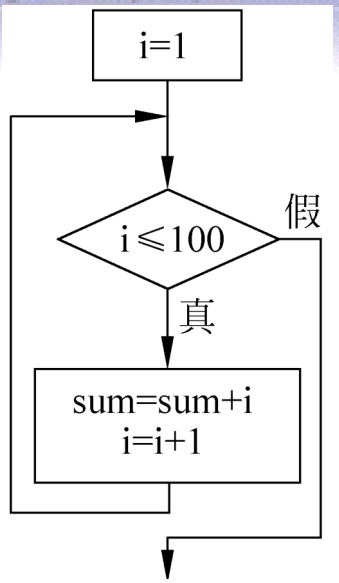

图3.10

需要注意：

(1) 循环体如果包含一个以上的语句，应该用花括号括起来，以复合语句形式出现。如果不加花括号，则**while**语句的范围只到**while**后面第一个分号处。

(2) 在循环体中应有使循环趋向于结束的语句。

3.11.2 用do-while语句构成循环

do-while语句的特点是先执行循环体，然后判断循环条件是否成立。其一般形式为

do

语句

while (表达式);

它是这样执行的：先执行一次指定的语句(即循环体)，然后判别表达式，当表达式的值为非零(“真”)时，返回重新执行循环体语句，如此反复，直到表达式的值等于**0**为止，此时循环结束。可以用图**3.11**表示其流程。

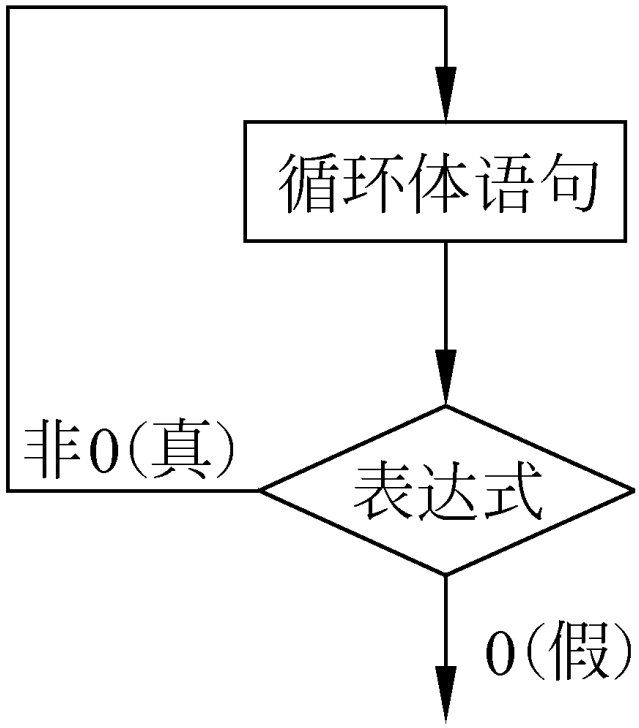

图3.11

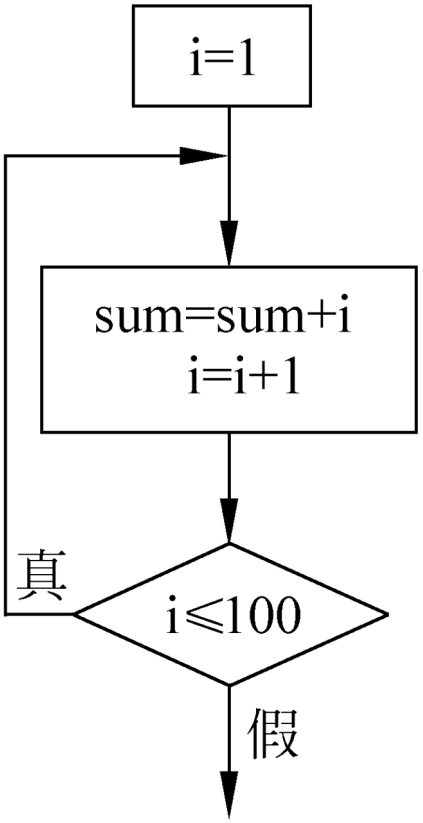

图3.12

例3.11 用do-while语句求1+2+3+…+100。

先画出流程图，见图3.12。

可编写出下面的程序：

```
#include <iostream>
using namespace std;
int main( )
{int i=1,sum=0;
 do
  { sum=sum+i;
    i++;
   }while (i<=100);
 cout<<"sum="<<sum<<endl;
 return 0;
}
```

运行结果与例**3.10**相同。

可以看到：对同一个问题可以用**while**语句处理，也可以用**do　while**语句处理。**do　while**语句结构可以转换成**while**结构。图**3.11**可以改画成图**3.13**的形式，二者完全等价。而图**3.13**中虚线框部分就是一个**while**结构。

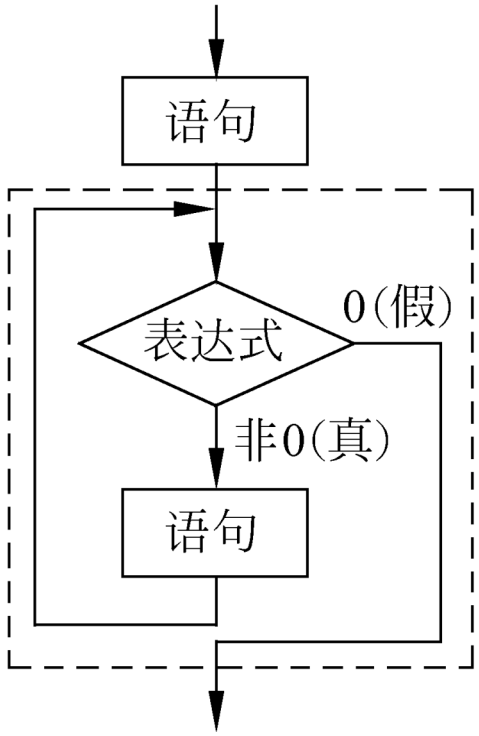

图**3.13**

3.11.3 用for语句构成循环

C++中的for语句使用最为广泛和灵活，不仅可以用于循环次数已经确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况，它完全可以代替while语句。

for语句的一般格式为

for(表达式1；表达式2；表达式3) 语句

它的执行过程如下：

(1) 先求解表达式1。

(2) 求解表达式2，若其值为真(值为非0)，则执行for语句中指定的内嵌语句，然后执行下面第(3)步。若为假(值为0)，则结束循环，转到第(5)步。

(3) 求解表达式**3**。

(4) 转回上面第**(2)**步骤继续执行。

(5) 循环结束，执行**for**语句下面的一个语句。

可以用图**3.14**来表示**for**语句的执行过程。

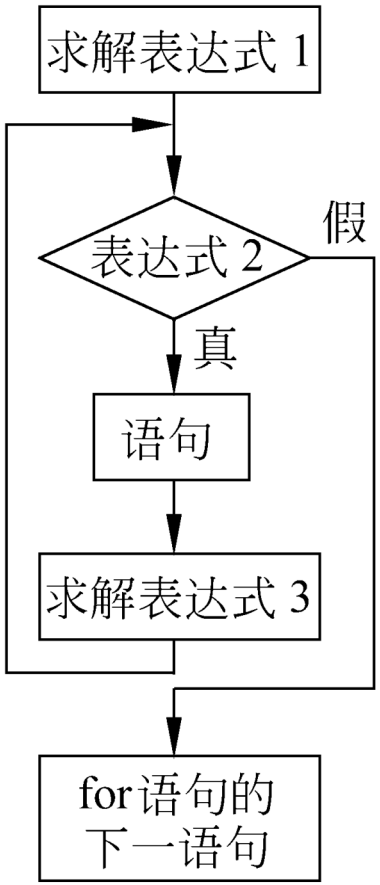

图**3.14**

for语句最简单的形式也是最容易理解的格式如下：

for(循环变量赋初值；循环条件；循环变量增值**)** 语句

例如

```
for(i=1;i<=100;i++) sum=sum+i;
```

它的执行过程与图**3.10**完全一样。它相当于以下语句：

```
 i=1;
  while(i<=100)
  {sum=sum+i;
  i++;
 }
```

显然，用**for**语句简单、方便。

for语句的使用有许多技巧，如果熟练地掌握和运用**for**语句，可以使程序精炼简洁。

说明：

(1) for语句的一般格式中的“表达式**1**”可以省略，此时应在**for**语句之前给循环变量赋初值。

(2) 如果表达式**2**省略，即不判断循环条件，循环无终止地进行下去。也就是认为表达式**2**始终为真。

(3) 表达式**3**也可以省略，但此时程序设计者应另外设法保证循环能正常结束。

(4) 可以省略表达式**1**和表达式**3**，只有表达式**2**，即只给循环条件。

(5) 3个表达式都可省略。

(6) 表达式**1**可以是设置循环变量初值的赋值表达式，也可以是与循环变量无关的其他表达式。

(7) 表达式一般是关系表达式(如**i<=100**)或逻辑表达式(如**a<b && x<y**)，但也可以是数值表达式或字符表达式，只要其值为非零，就执行循环体。

C++中的**for**语句比其他语言中的循环语句功能强得多。可以把循环体和一些与循环控制无关的操作也作为表达式**1**或表达式**3**出现，这样程序可以短小简洁。但过分地利用这一特点会使**for**语句显得杂乱，可读性降低，建议不要把与循环控制无关的内容放到**for**语句中。

3.11.4 几种循环的比较

(1) 3种循环都可以用来处理同一问题，一般情况下它们可以互相代替。

(2) while和do-while循环，是在while后面指定循环条件的，在循环体中应包含使循环趋于结束的语句(如i++，或i=i+1等)。

for循环可以在表达式3中包含使循环趋于结束的操作，甚至可以将循环体中的操作全部放到表达式3中。因此for语句的功能更强，凡用while循环能完成的，用for循环都能实现。

(3) 用while和do-while循环时，循环变量初始化的操作应在while和do-while语句之前完成。而for语句可以在表达式1中实现循环变量的初始化。

3.12 循环的嵌套

一个循环体内又包含另一个完整的循环结构，称为循环的嵌套。内嵌的循环中还可以嵌套循环，这就是多层循环。

3种循环**(while**循环、**do while**循环和**for**循环**)**可以互相嵌套。例如，下面几种都是合法的形式：

(1)

```
while( )
{┆
while( )
{…}
}
```

```
(2)
do
{ ┆
do
{…}while( );
}while( );
(3)
for(;;)
{ ┆
for(;;)
{…}
}
```

```
(4)
while( )
{ ┆
do
{…}while( );
 ┆
}
(5)
for(;; )
{ ┆
while( )
{…}
 ┆
}
```

(6)

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

3.13 break语句和continue语句

在**3.9**节中已经介绍过用**break**语句可以使流程跳出**switch**结构，继续执行**switch**语句下面的一个语句。实际上，**break**语句还可以用于循环体内。

break语句的一般格式为

break;

其作用为使流程从循环体内跳出循环体，即提前结束循环，接着执行循环体下面的语句。**break**语句只能用于循环语句和**switch**语句内，不能单独使用或用于其他语句中。

continue语句的一般格式为

continue;

其作用为结束本次循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判定。

continue语句和**break**语句的区别是：**continue**语句只结束本次循环，而不是终止整个循环的执行。而**break**语句则是结束整个循环过程，不再判断执行循环的条件是否成立。如果有以下两个循环结构：

(1)

```
while(表达式1)
{┆
if（表达式2）break
 ┆
}
```

(2)

```
while(表达式1 )
{ ┆
if(表达式2) continue;
 ┆
}
```

程序**(1)**的流程图如图**3.18**所示，而程序**(2)**的流程如图**3.19**所示。请注意图**3.18**和图**3.19**中当“表达式**2**”为真时流程的转向。

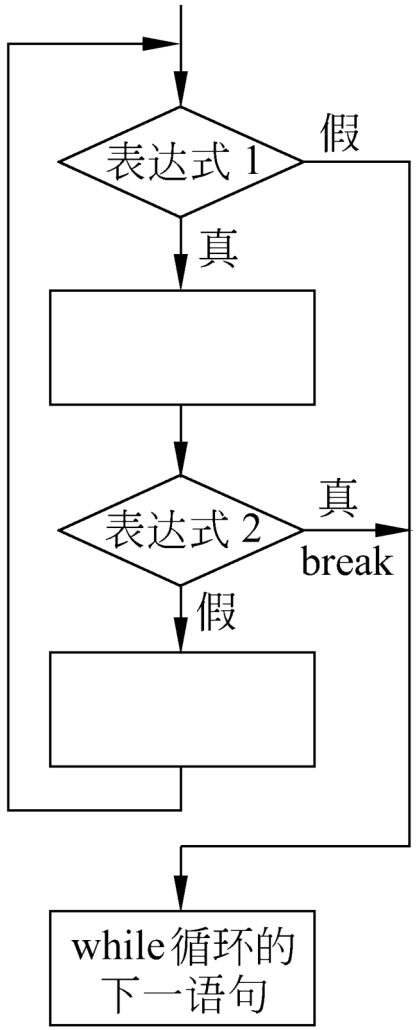
表达式 1
假
真
表达式 2
真
break
假
while循环的
下一语句

图3.18

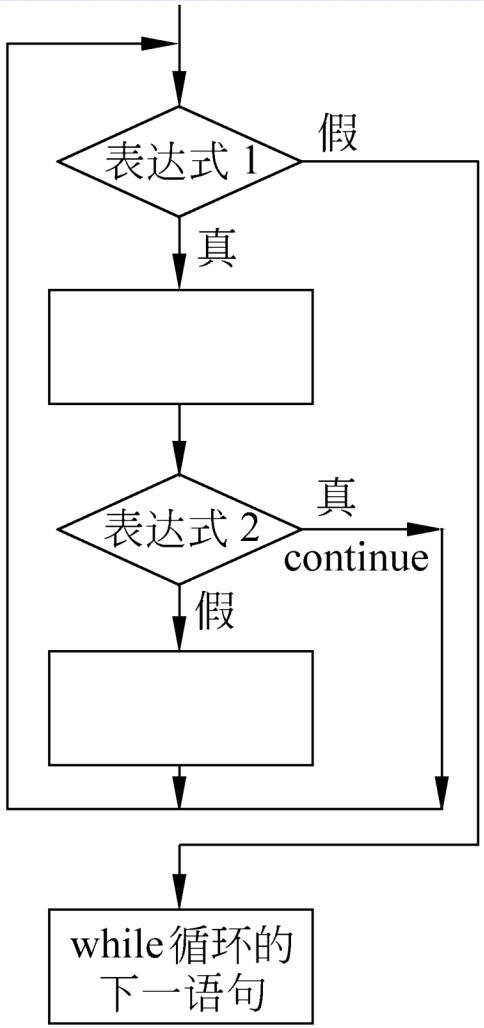
表达式 1
假
真
表达式 2
真
continue
假
while循环的
下一语句

图3.19

3.14 编写循环结构的程序

例3.12 用下面公式求π的近似值。π/4≈1-1/3+1/5-1/7+…直到最后一项的绝对值小于10^{-7}为止。

根据给定的算法很容易编写程序如下：

```
#include <iostream>
#include <iomanip>
#include <cmath>
    using namespace std;
int main( )
{int s=1;
 double n=1,t=1,pi=0;
 while((fabs(t))>1e-7)
   {pi=pi+t;
   n=n+2;
```

```
 s=-s;
 t=s/n;
}
pi=pi*4;
cout<<"pi="<<setiosflags(ios::fixed)<<setprecision(6)<<pi<<endl;
return 0;
}
```

运行结果为

pi=3.141592

注意： 不要把**n**定义为整型变量，否则在执行**“t=s/n;”**时，得到**t**的值为**0(**原因是两个整数相除**)**。

例**3.13** 求**Fibonacci**数列前**40**个数。这个数列有如下特点：第**1**、**2**个数为**1**、**1**。从第**3**个数开始，每个数是其前面两个数之和。即

F1=1　　(n=1)

F2=1　　(n=2)

Fn=Fn-1+Fn-2　(n≥3)

这是一个有趣的古典数学问题：有一对兔子，从出生后第**3**个月起每个月都生一对兔子。小兔子长到第**3**个月后每个月又生一对兔子。假设所有兔子都不死，问每个月的兔子总数为多少？

可以从书中表**3.4**看出兔子数的规律。

根据给出的每月兔子总数的关系，可编写程序如下：

```
#include <iostream>
#include <iomanip>
using namespace std;
int main( )
{long f1,f2;
 int i;
 f1=f2=1;
 for(i=1;i<=20;i++)
   {cout<<setw(12)<<f1<<setw(12)<<f2;
//设备输出字段宽度为12，每次输出两个数
    if(i%2==0) cout<<endl;
//每输出完4个数后换行，使每行输出4个数
    f1=f1+f2;
//左边的f1代表第3个数，是第1、2个数之和
    f2=f2+f1;
//左边的f2代表第4个数，是第2、3个数之和
   }
   return 0;
 }
```

例3.14 找出100~200间的全部素数。

编写程序如下：

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;
int main( )
{int m,k,i,n=0;
 bool prime;      //定义布尔变量prime
 for(m=101;m<=200;m=m+2)     //判别m是否为素数，m由101变化到200，增量为2
   {prime=true;  //循环开始时设prime为真，即先认为m为素数
k=int(sqrt(m));          //用k代表根号m的整数部分
    for(i=2;i<=k;i++)       //此循环作用是将m被2~根号m除，检查是否能整除
if(m%i==0)            //如果能整除，表示m不是素数
```

```
{ prime=false;      //使prime变为假
break;              //终止执行本循环
}
   if (prime)               //如果m为素数
     {cout<<setw(5)<<m;     //输出素数m，字段宽度为5
      n=n+1;                //n用来累计输出素数的个数
     }
   if(n%10==0) cout<<endl;  //输出10个数后换行
  }
 cout<<endl;                //最后执行一次换行
 return 0;
}
```

例**3.15** 译密码。为使电文保密，往往按一定规律将电文转换成密码，收报人再按约定的规律将其译回原文。例如，可以按以下规律将电文变成密码：将字母**A**变成字母**E**，**a**变成**e**，即变成其后的第**4**个字母，**W**变成**A**，**X**变成**B**，**Y**变成**C**，**Z**变成**D**。见图**3.20**。字母按上述规律转换，非字母字符不变。如**"Wonderful!"**转换为**"Asrhivjyp!"**。输入一行字符，要求输出其相应的密码。

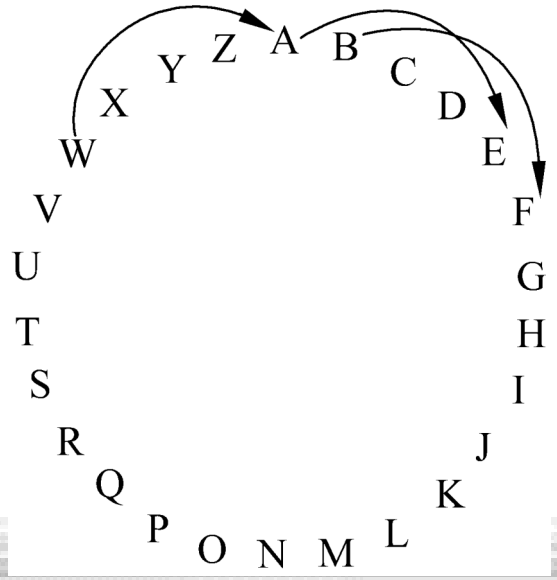

图**3.20**

程序如下：

```
#include <iostream>
using namespace std;
int main( )
{char c;
 while ((c=getchar( ))!='\n')
   {if((c>='a' && c<='z') || (c>='A' && c<='Z'))
      {c=c+4;
       if(c>'Z' && c<='Z'+4 || c>'z') c=c-26;
      }
    cout<<c;
   }
 cout<<endl;
 return 0;
}
```

运行结果如下：

I am going to Beijing!↙

M eq ksmrk xs Fimnmrk!

while语句中括号内的表达式有**3**个作用：①从键盘读入一个字符，这是用**getchar**函数实现的；②将读入的字符赋给字符变量**c**；③判别这个字符是否为**′\n′**(即换行符)。如果是换行符就执行**while**语句中的复合语句(即花括号内的语句)，对输入的非换行符的字符进行转换处理。

按前面分析的思路对输入的字符进行处理。有一点请读者注意：内嵌的**if**语句不能写成

```
if (c>'Z'|| c>'z')  c=c-26;
```

因为所有小写字母都满足“**c>'Z'**”条件，从而也执行“**c=c-26;**”语句，这就会出错。因此必须限制其范围为“**c>'Z' && c<='Z'+4**”，即原字母为**'W'**到**'Z'**，在此范围以外的不是原大写字母**W~Z**，不应按此规律转换。请考虑：为什么对小写字母不按此处理，即写成**c>'z' && c<='z'+4**而只须写成“**c>'z'**”即可。