第一章　基本内容

当我们开始对 C++ 和面向对象的程序设计进行研究时，必须先做一些说明。由于面向对象的程序设计这一领域对你可能是全新的，所以，需要掌握大量的术语。本书做了一种新的尝试，在每一章节都增加一些新的论题，从而使你逐步掌握整个语言。

本书中的 1 至 4 章集中讨论 C++ 中非面向对象的程序设计方面新增加的内容。从第 5 章开始讨论面向对象的程序设计技术。

C++ 中的注释及常量

CONCOM. CPP　　Constants and comments

```
01                                      // Chapter 1 -- Program 1
02
03  #include <iostream. h>          /* This is the stream definition file */
04
05  void print _ it(const int data _ value);
06
07  main()
08  {
09  const int START = 3;            // The value of START cannot be changed
10  const int STOP = 9;             // The value of STOP cannot be changed
11  volatile int CENTER = 6;        /* The value of CENTER may be changed
12                                      by something external to this
13                                      program.                    */
14  int index;                      /* A normal C variable              */
15
16     for (index = START ; index < STOP ; index++)
17          print _ it(index);
18  }       /* End of program */
19
20
21  void print _ it(const int data _ value)
22  {
23     cout << "The value of the index is " << data _ value << "\n";
24  }
25
26
27
28
29  // Result of execution
```

```
30  //
31  // The value of the index is 3
32  // The value of the index is 4
33  // The value of the index is 5
34  // The value of the index is 6
35  // The value of the index is 7
36  // The value of the index is 8
37
```

示例程序 concom.cpp 中有若干 C++ 引入的新内容，我们从注释开始。

C++ 中，注释以双斜线“//”开头，可以在一行中的任何位置开始，直至行尾并自动终止。ANSI C 定义的注释方法也能在 C++ 中使用，即以“/*”开头，“*/”结尾，如程序中第 11 行至 14 行。新的注释定义方法较为理想，因为一般很少在程序中一次加入多行注释。使用 ANSI C 的注释时，一定要写上注释结束符“*/”，否则就会出错。进行程序设计时，可以混用这两种注释方法。凡是需注释的地方都用 C++ 的注释方法写出，这样可以在行尾自动终止，而在调试程序期间，需跳过暂不执行的程序段可以用 ANSI C 的注释方法将该段程序定义成注释，这样可一次括出一段程序。C++ 中这两种注释可以相互嵌套。

应该提醒的是，当程序中定义的变量、常数、函数的名字与其实际含义相同时，几乎可以不使用注释。悉心选择的变量名、函数名基本上就可以说明程序的功能。因此，你应该力争在你的程序中做到这一点。

关键字 const 和 volatile

在程序的第 9 至 11 行中有两个新的关键字 const 和 volatile。关键字 const 用来定义一个常量。在第 9 行中，该常量的类型为 int，命名为 START，并且初始化为 3。编译程序将不允许你有意无意地改变 START 的值，因为它已经声明为常量。如果还有另一个变量名为 STARTS，系统将不允许你将 STARTS 错拼为 START，并为它赋新值。对这类错误编译程序将给出错误信息帮助你定位错误。此外，系统允许在定义常量时不指出其初值。

请注意关键字 const 也在 21 行中用于函数头的定义，用来指出名为 data_value 的形式参数在函数体中是常量。任何向这个变量赋予新值的企图都将导致编译错误。这种用法可增加编译程序查错的能力。

关键字 volatile 也是 ANSI C 标准的一部分。一个 volatile 变量的值除了可以由程序员来修改以外，还可能存在其它手段来修改这个值。诸如中断计时器会引起值的变化。当编译程序优化代码时，需要了解这种值是否可以由某种外部机制改变。研究代码优化方法是十分有意义的，但它已超出了这本书的范围。注意，常量也可以是 volatile 型的，这意味着你不能改变它的值，但是系统通过一些硬件功能却可以做到这一点。

先不要考虑第 23 行中的输出语句，我们将在后面的章节中详细讨论它。如果你对 K&RC 风格的编程很有经验的话，你会发现第 5 至 21 行有些奇特。它说明了 ANSI C 标准所采用的函数原型定义方法。我们将在第 4 章中详尽地讨论这一点。在 C 中，原型是可选的，但在 C++ 中是必需的。基于这一理由，这本指南的第 4 章将全部用于对原型的讨论。

请用你的 C++ 系统编译、执行此程序。检查结果与注释中给出的是否相同，这对你是十分有益的。编译它的首要目的在于证明你的编译系统已被装载并正常执行了。

范围操作

SCOPEOP. CPP　　Scope operator

```
01                                          // Chapter 1 — Program 2
02  #include <iostream. h>
03
04  int index = 13;
05
06  main()
07  {
08  float index = 3. 1415;
09
10     cout << "The local index value is " << index << "\n";
11     cout << "The global index value is " << ::index << "\n";
12
13     ::index = index + 7; // 3 + 7 should result in 10
14
15     cout << "The local index value is " << index << "\n";
16     cout << "The global index value is " << ::index << "\n";
17
18  }
19
20
21
22
23  // Result of execution
24  //
25  // The local index value is 3. 1415
26  // The global index value is 13
27  // The local index value is 3. 1415
28  // The global index value is 10
29
```

名为 scopeop. cpp 的示例程序说明了 C++ 的另一个新结构。在 K&R C 及 ANSI C 中均没有对应的内容。这里，允许访问名为 index 的全程变量，即使函数 main( )中有一个相同名字的

局部变量也可以这样做。在变量名前使用双冒号“::”,如 11、13、16 行,就告诉系统,我们希望利用在第 4 行中定义的名为 index 的全程变量,而非在第 8 行中定义的局部变量。

使用这一技术可以在任一时刻使用全程变量。用于计算,作为函数的参数,或任何其它目的。但因为它使得程序阅读困难,故最好还是使用不同的变量名,但如果有必要的话,你可以这样使用。

这也就是说,即使是被局部变量所掩盖,范围操作使得我们仍能使用全程变量。在上机编译、执行完此程序之前,请不要急于进入下面的内容。

I/O 流库

MESSAGE. CPP　　The stream libray

```
01                                          // Chapter 1 — Program 3
02  #include <iostream.h>
03  #include <string.h>
04
05  main()
06  {
07  int index;
08  float distance;
09  char letter;
10  char name[25];
11
12     index = -23;
13     distance = 12.345;
14     letter = 'X';
15     strcpy(name,"John Doe");
16
17     cout << "The value of index is " << index << "\n";
18     cout << "The value of distance is " << distance << "\n";
19     cout << "The value of letter is " << letter << "\n";
20     cout << "The value of name is " << name << "\n";
21
22     index = 31;
23     cout << "The decimal value of index is " << dec << index << "\n";
24     cout << "The octal value of index is " << oct << index << "\n";
25     cout << "The hex value of index is " << hex << index << "\n";
26     cout << "The character letter is " << (char)letter << "\n";
27
28     cout << "Input a decimal value --> ";
29     cin >> index;
```

```
30      cout << "The hex value of the input is " << index << "\n";
31  }
32
33
34
35
36  // Result of execution
37  //
38  // The value of index is -23
39  // The value of distance is 12.345
40  // The value of letter is X
41  // The value of name is John Doe
42  // The decimal value of index is 31
43  // The octal value of index is 37
44  // The hex value of index is 1f
45  // The character letter is X
46  // Input a decimal value -->  999
47  // The hex value of the input is 3e7
```

查看名为 MESSAGE.CPP 的示例程序。这是第一个涉及面向对象的程序设计的程序，尽管它十分简单。在这个程序中，我们定义了一些变量并为之赋值，然后在 17 至 20 行以及 23 至 26 行的输出语句中使用。新的操作符 cout 是对标准装置即监视器的输出函数，但其工作方式与我们熟悉的 printf()有所不同，因为我们不必告诉系统我们正在输出的值是什么类型。

C++ 同 C 语言一样，语言本身并没有任何输入输出功能。但是通过定义流(stream)库，就以一种很精巧的方式增加了输入输出功能。

操作符<<，有时被称为“置入”操作符，或更恰当地称为“插入”操作符，它告诉系统输出操作符后面的变量、常量，但是由系统决定如何输出数据。在第 17 行，我们首先让系统输出一个串，这通过向监视器拷贝字符完成，然后我们让它输出 index 的值。请注意，我们并未告诉系统它的类型是什么或者如何输出其值。由于我们不告诉系统输出数据的类型是什么，因此就由系统确定其类型是什么，并据此输出值。系统根据缺省定义决定输出数据所需的位数。在此例中，我们看到系统确实只使用了输出数据所必需的位数，即无前导空白亦无尾空白，这对这种输出来说是很好的。最后，输出换行符，代码行以分号“;”终止。

当我们在第 17 行调用 cout 输出语句时，我们实际上调用了两种不同的功能，用它同时输出字符串和整型变量。这是面向对象的程序设计的第一个示例，这里我们只是向系统传送信息以输出一个值，并让系统找到一种适当的功能来完成之，并不需要用户确切的告诉系统如何输出数据。这只是面向对象的程序设计的一个十分基本的例子，在后面我们将逐步深化。

在 18 行上，我们让系统输出另一个串，其后为一个浮点数，以及一个只有一个换行符的串。此时，我们让它输出一个浮点数，但并没有告诉系统它是浮点数，因而又由系统根据其类型选择适当的输出格式。我们确实失去了一点对程序的控制，然而，因为我们不知道在小数点前、后各需要几位数字，因此只能让系统决定输出数据的格式。

变量 letter 的类型为 char，并且在第 14 行中被赋值为大写的 X，在 19 行中作为字母被打印。

C++ 中还有几种可用于流的操作符，它们保证了用户在使用流时有足够的灵活性。编译程序的文档文件中列出了其它格式命令的细节。cout 和 printf()语句能以任意方式混合使用，两种语句都导致对监视器的输出。

有关流库的更多信息

在 C++ 中定义流库是为了提高语言的执行效率。printf()函数在早期的 C 语言中即被开发，并被定义为可满足用户的所有要求。这就使它成为一个巨大的函数，其中有许多功能只有极少数人使用，而对大多数程序员来说其规模已成为一种负担。C++ 定义了一些小的特殊目的的流库。为程序员再提供一些有限的格式安排命令，这些命令对大多编程工作来说已足够了。如果用户需要更复杂的格式安排能力，亦可以在 C++ 中使用 printf()函数。这两种类型的输出能自由混合。

23 至 26 行说明了流库的其它一些特性，它能以一种十分灵活然而可控制的格式输出数据。从 23 至 25 行，index 值分别按十进制，八进制，十六进制格式输出，在输出中使用特殊的流操作符 dec、oct、hex 时，所有后继的输出均按该数制输出。查看第 30 行，这里 index 的值按十六进制格式输出。这是因为在 25 行选择了 hexdecimal 数制。如果输出中没有这些特殊的流操作符，系统缺省定义为十进制格式。

操作符 cin

除了操作符 cout，还有一个操作符 cin，用于从标准设备(通常为键盘)读取数据。cin 操作符使用“>>”，通带被称为“从何处获取”或称为“提取”操作符，它具有 cout 操作符的绝大多数灵活性，在程序的 28 行至 30 行给出了使用 cin 的一个简单的例子。特殊的流操作符 dec、oct、hex 用来对输入流选定数制，如果没有指明，输入流缺省为十进制格式。

除了 cout 操作符和 cin 操作符外，还有一个标准的操作符 cerr，它用于向出错处理设备输出。该设备不能改向文件。这三个操作符 cout、cin、cerr 与 C 语言中的 stdout、stdin 和 stderr 流指针相对应，其用法将在本书的其它部分给出。

文件流操作

FSTREAM. CPP　　File streams

```
01                                          // Chapter 1 — Program 4
02  #include <iostream. h>
03  #include <fstream. h>
04  #include <process. h>
05
06  void main()
07  {
```

```
08  ifstream infile;
09  ofstream outfile;
10  ofstream printer;
11  char filename[20];
12
13      cout << "Enter the desired file to copy ----> ";
14
15      cin >> filename;
16
17      infile.open(filename, ios::nocreate);
18      if (! infile) {
19          cout << "Input file cannot be opened. \n";
20          exit(1);
21      }
22
23      outfile.open("copy");
24      if (! outfile) {
25          cout << "Output file cannot be opened. \n";
26          exit(1);
27      }
28
29      printer.open("PRN");
30      if (! printer) {
31          cout << "There is a problem with the printer. \n";
32          exit(1);
33      }
34
35      cout << "All three files have been opened. \n";
36
37  char one_char;
38
39      printer << "This is the beginning of the printed copy. \n\n";
40
41      while (infile.get(one_char)) {
42          outfile.put(one_char);
43          printer.put(one_char);
44      }
45
46      printer << "\n\nThis is the end of the printed copy. \n";
47
48      infile.close();
49      outfile.close();
50      printer.close();
51
```

```
52  }
53
54
55
56  // Result of execution
57  //
58  // (The input file is copied to the file named "COPY")
59  // (The input file is printed on the printer
```

查看名为 FSTREAM. CPP 的示例程序，这是一个使用文件流的例子。

在此程序中，共引入了三个文件，第一个文件用于读，第二个用于写，第三个文件用于打印，从而说明针对文件的流操作的语义。上一程序中的流操作与本程序中的流操作之间的唯一区别是，上一程序中，流是由系统打开的，而在本程序则是由用户自己打开的。你可能注意到，在名为 printer 的流中使用<<操作符的方法与我们在上一程序中的用法完全相同。最后，为培养好的编程习惯，在结束程序前关闭所有前面已打开的文件。

I/O 标准文件库也可在 ANSI C 中使用，并且同流库一样简单方便。若要了解 I/O 库流文件的更多信息，请参看 Bjarne Stroustrup 的书，或参看你的编译文档。

请编译并执行此程序，当执行它时，需要一个文件来拷贝。你可以输入当前目录中存在的任意 ASCII 文件的文件名。

变量定义

VARDEF. CPP　Variable　definitions

```
01                                          // Chapter 1 - Program 5
02  #include <iostream. h>
03
04  int index;
05
06  main()
07  {
08  int stuff;
09  int &another _ stuff = stuff;          // A synonym for stuff
10
11      stuff = index + 14;                //index was initialized to zero
12      cout << "stuff has the value " << stuff << "\n";
13      stuff = 17;
14      cout << "another _ stuff has the value " << another _ stuff << "\n";
15
```

```
16  int more _ stuff = 13;                    //not automatically initialized
17
18      cout << "more _ stuff has the value " << more _ stuff << "\n";
19
20      for (int count = 3;count < 8;count++) {
21          cout << "count has the value " << count << "\n";
22          char count2 = count + 65;
23          cout << "count2 has the value " << count2 << "\n";
24      }
25
26  static unsigned goofy;                    //automatically initialized to zero
27
28      cout << "goofy has the value " << goofy << "\n";
29  }
30
31
32
33
34  // Result of execution
35  //
36  // stuff has the value 14
37  // another _ stuff has the value 17
38  // more _ stuff has the value 13
39  // count has the value 3
40  // count2 has the value D
41  // count has the value 4
42  // count2 has the value E
43  // count has the value 5
44  // count2 has the value F
45  // count has the value 6
46  // count2 has the value G
47  // count has the value 7
48  // count2 has the value H
49  // goofy has the value 0
```

查看名为 VARDEF. CPP 的示例程序，其中增加了一些对提高程序的清晰性和易读性很有帮助的内容。在 C++ 中，全程变量和静态变量被自动初始化为 0。在第 4 行的变量 index 及 26 行的 goofy 被自动初始化为 0，当然，如果需要，你还可以将之初始化为任意值。全程变量有时被称为外部变量，因为它们对任何函数来说都是外部的。

那些在函数中间定义的变量，不能自动初始化。但它将具有被定义时所分配位置处的值，该值必须认为是一个垃圾值。在第 8 行中，名为 stuff 的变量就不具有一个有效值，但却可能有

一个不能用于任何有效目地的垃圾值。在第 11 行，根据 index 的初值为 stuff 赋予了一个值，并且显示在监视器上供你检验。

参考变量

注意第 9 行的宏代替符号 &，它将变量 another _ stuff 定义成一个参考变量，这是 C++ 中新增加的内容。参考变量最好不要频繁使用，无论采用何种方式。但为了完整起见，我们仍将讨论它的操作。参考变量不象其它变量，它的操作就象是一个自适应的指针。在它初始化后，参考变量就成为变量 stuff 的一个同义词，改变数值将导致 another _ stuff 的值改变，因为它们指向同一变量。在程序中，同义词可合法地用于访问变量的值。需要指出，定义参考变量时必须初始化为指向某一变量，否则编译程序将报告出错，参考变量一旦初始化，就不能改为指向另一变量。

以这种方式使用参考变量会导致程序非常复杂。但还有另一种用法可以使得代码非常清晰且易于理解。我们将在第 4 章讨论这种用法。

定义也是可执行语句

按一般 C 语言的观点，你会发现 16 行的语句很奇怪，但在 C++ 中这是合法的。在任何位置插入可执行语句是合法的，说明一个新变量也是合法的，因为在 C++ 中数据定义语句也被定义为可执行语句。在这种情况下，我们定义一个新变量 more _ stuff 并将之初始化为 13。它的有效范围是从定义点开始直至定义它的块结束为止。因此，在本程序中直至程序尾它均有效。变量 goofy 甚至直到第 26 行才定义。

定义和说明是什么

与 ANSI C 一样，在 C++ 中定义和说明指的是两件不同的事情。它们确实不一样并有不同的含义。我们将花费一点时间来定义在 C++ 中其含义是什么。说明(declaration)提供给编译器一些字符型信息，用于诸如类型、函数等方面。但它们不定义任何在可执行程序中可使用的实际代码，如果愿意，允许你为同一件事给出多个说明。另一方面，定义(definition)给出在可执行程序中确实存在的一些内容，如一些可使用的变量，或一些可执行的代码，并且需要你在程序中为一个条目给出唯一的定义，简言之，一个说明向程序中引入一个名字，而一个定义引入一些代码。

如果我们说明一个结构，只说明了一个格式，用来告诉编译程序当我们后面定义一个或多个该种类型的变量时，如何存储数据。但是，当我们定义一个该种类型的变量时，我们不仅实实在在地指出了它们的名字，供编译程序使用，而且分配了一个存储区域来存放变量的值。因此，当我们定义一个变量时，我们实际上同时说明并定义了它。

本书中，我们会反复引用这种定义，因此，如果你现在还不是十分清楚的话，以后会搞清的。

对 for 循环的改进

请注意第 20 行定义的 for 循环，这种循环方式比 ANSI C 中使用的方式要清楚一些，因为循环索引变量是在循环内部定义的，循环索引变量的有效范围是从其说明开始直至包含它的

块尾。在本例中，其范围为到第 29 行，因为第 29 行的右括号对应着定义该变量前面的左括号。由于该变量仍然可使用，它可用作另一个循环的索引变量，如同一个一般的整型变量那样在可合法使用的位置使用。名为 count2 的变量在循环的每一遍中被说明及初始化，因为它是在 for 循环块中定义的，其范围为循环体。因此每当循环结束，它就被自动取消。它被说明、初使化、取消了 5 次，每一次循环对应一次。

我们注意到变量 count2 在 22 行被赋予了一个数值，但当打印时，输出的是一个字符。这是因为 C++(2.0 版以上)很注意使用正确的类型。

最后，象前面指出的那样，名为 goofy 的静态变量在 26 行被说明并自动赋予初值 0。其适用范围是从定义点开始，直至定义它的块尾，即第 29 行。

请编译并执行该程序。

运算优先级

C++ 运算优先级的定义与 ANSI C 是完全相同的，因此这里不必再次定义。但当出现操作符重载时，会有一点不同，对此我们将在后面研究。当操作符重载时，其工作方式与语言预定义的方式有所不同。

不要担心前面的这段话，在我们学习了后面的内容后你就会清楚的。

编程练习

1. 编写一个程序，使用 cout 函数，在显示器上显示 3 次你的姓名、出生日期。定义的变量名尽量与其含义接近。

2. 编写一个程序，带有常量及 volatile 变量，并试图修改常量的值，观察编译程序将给出的错误信息。

3. 编写一个程序，使用流，用三个不同的语句交互式地读入你的出生日。按八进制、十进制、十六进制格式打印你的生日。

第二章　复合类型

枚举类型

ENUM. CPP　The enumerater type

```
01                                          // Chapter 2 — Program 1
02  #include <iostream.h>
03
04  enum game_result {win, lose, tie, cancel};
05
06  main()
07  {
08  game_result result;
09  enum game_result omit = cancel;
10
11      for (result = win;result <= cancel;result++) {
12          if (result == omit)
13              cout << "The game was cancelled\n";
14          else {
15              cout << "The game was played ";
16              if (result == win)
17                  cout << "and we won!";
18              if (result == lose)
19                  cout << "and we lost.";
20              cout << "\n";
21          }
22      }
23  }
24
25
26
27
28  // Result of execution
29  //
30  // The game was played and we won!
31  // The game was played and we lost.
32  // The game was played
33  // The game was cancelled
34
```

查看名为 ENUM. CPP 的示例程序，其中使用了一个枚举型的变量，枚举型变量在 C++ 中的使用方式与 ANSI C 中几乎完全一样，但有一个小的例外，即当定义枚举类型的变量时可以不使用关键字 enum，当然若需要也可以使用，对用户来说，定义一个变量时，象 C 语言那样使用该关键字可能更清楚一点，并且你很可能选择这样做。

在本程序中，第 9 行使用了关键字 enum，在第 8 行中略去了，这就告诉我们该关键字确实是可选的。理解程序的其余部分对你来说应该没有问题。研究它之后，请编译并执行之，以便检验输出结果。

简单结构

STRUCTUR. CPP　The structure

```
01                                    // Chapter 2 — Program 2
02  #include <iostream. h>
03
04  struct animal {
05      int weight;
06      int feet;
07  };
08
09  main()
10  {
11  animal dog1, dog2, chicken;
12  animal cat1;
13  struct animal cat2;
14
15      dog1. weight = 15;
16      dog2. weight = 37;
17      chicken. weight = 3;
18
19      dog1. feet = 4;
20      dog2. feet = 4;
21      chicken. feet = 2;
22
23      cout << "The weight of dog1 is " << dog1. weight << "\n";
24      cout << "The weight of dog2 is " << dog2. weight << "\n";
25      cout << "The weight of chicken is " << chicken. weight << "\n";
26  }
27
```

```
28
29
30
31  // Result of execution
32  //
33  // The weight of dog1 is 15
34  // The weight of dog2 is 37
35  // The weight of chicken is 3
36
```

查看名为 STRUCTUR. CPP 的示例程序，其中使用了一个非常简单的结构。该结构与 ANSI C 中使用的结构并没有什么不同，只是当定义该类型的变量时，也可以不使用关键字 struct。11 和 12 行给出了没有关键字的变量定义方式，而在 13 行中则使用了关键字 sturct。这样，在 C++ 中，可由用户来决定选用哪种编程风格。

请注意，这基本上是一个有效的 ANSI C 程序，几个细微的差别是：使用流库的方法、C++ 的注释方法，以及在两行中没有使用关键字 struct 等。

在你仔细研究了程序之后，编译并执行它，因为下一个示例程序与它很类似，只是引入了 C++ 的另一种结构——类。

一个非常简单的类

CLASS1. CPP　The class

```
01                                  // Chapter 2 — Program 3
02  #include <iostream. h>
03
04  class animal {
05  public:
06      int weight;
07      int feet;
08  };
09
10  main()
11  {
12  animal dog1, dog2, chicken;
13  animal cat1;
14  class animal cat2;
15
16      dog1. weight = 15;
```

```
17      dog2. weight = 37;
18      chicken. weight = 3;
19
20      dog1. feet = 4;
21      dog2. feet = 4;
22      chicken. feet = 2;
23
24      cout << "The weight of dog1 is " << dog1. weight << "\n";
25      cout << "The weight of dog2 is " << dog2. weight << "\n";
26      cout << "The weight of chicken is " << chicken. weight << "\n";
27  }
28
29
30
31
32  // Result of execution
33  //
34  // The weight of dog1 is 15
35  // The weight of dog2 is 37
36  // The weight of chicken is 3
37
```

查看名为 classl. cpp 的示例程序，它给出了 C++ 中类的例子。这是关于类的第一个例子，但并不是最后一个，因为引入类是 C++ 比 ANSI C 及其它程序语言优越的一个主要原因。你可能注意到第 4 行上使用的关键字 class，如同上一程序中关键字 struct 的用法一样，它们是非常类似的结构。虽然其含义不同，但就目前而言，我们可以只关心其相似性。

在第 4 行的 animal 是类的名字。当我们定义该种类型的变量时，如 12 至 14 行，写不写关键字 class 都是可以的，在上一程序中，我们定义了一种结构类型的 5 个变量，而在此程序中，我们定义了 5 个对象(object)。将它们称为对象是因为它们是 class 类型。区别是细微的，而且在目前情况下，区别是可以忽视的。但当我们接着向下看时，我们会发现“类”结构确实非常重要，而且很有价值。这里介绍类只是让你了解一下本手册下面将会有什么内容。

类是一种类型，可用来定义对象。定义方式类似于结构类型可用于定义变量。你的名为 king 的狗是“狗类”中一个特定的例子。同样，一个对象是一个类的一个特定的例子。最好注意到，类是一种通用概念、很快将会有预先定义好的类的库在市场上出现。你可以购买类库，它将完成一些通用的操作，如管理栈、队列、链表、进行数据排序，管理窗口等。事实上，目前已有一些类的库可供使用。

第 5 行中新的关键字 public 后跟一个冒号，在这里是必要的。因为一个类中的变量缺省定义为私有式的。不得将之公有化，因此若不这样定义，我们根本就不可能访问它。请先不要担心此程序，在本书的后面我们会涉及它里面的所有细节。

请编译并执行该程序，看编译程序是否象我们所说的那样工作。请记住这是关于类的第一

个例子，它只指出一些基本内容，而未涉及 C++ 这一强有力结构的用法。

C++ 的自由联合(free union)

UNIONEX. CPP　The union

```
01                                          // Chapter 2 — Program 4
02  #include <iostream. h>
03
04  struct aircraft {
05      int wingspan;
06      int passengers;
07      union {
08          float fuel _ load; // for fighters
09          float bomb _ load; // for bombers
10          int pallets; // for transports
11      };
12  } fighter, bomber, transport;
13
14  main()
15  {
16      fighter. wingspan = 40;
17      fighter. passengers = 1;
18      fighter. fuel _ load = 12000. 0;
19
20      bomber. wingspan = 90;
21      bomber. passengers = 12;
22      bomber. bomb _ load = 14000. 0;
23
24      transport. wingspan = 106;
25      transport. passengers = 4;
26      transport. pallets = 42;
27
28      transport. fuel _ load = 18000. 0;
29      fighter. pallets = 4;
30
31      cout << "The fighter carries "
32                              << fighter. pallets << " pallets. \n";
33      cout << "The bomber bomb load is " << bomber. bomb _ load << "\n";
34
35  }
36
```

```
37
38
39
40  // Result of execution
41  //
42  // The fighter carries 4 pallets.
43  // The bomber bomb load is 14000
```

查看名为 unionex.cpp 的示例程序，它给出了一个自由联合的例子，在 ANSI C 中，所有的联合必须被命名以便使用，但在 C++ 中这一点并不成立。在使用 C++ 时，我们可以使用自由联合，即一个没有名字的联合，该联合嵌入在一个简单的结构中，你会注意到在 11 行中联合的定义后面没有跟变量名。在 ANSI C 中，我们需要为联合命名，并用一种三组名(三个名字用点联合接在一起)来访问该元素。而这里因为是一个自由联合，没有联合名，访问变量采取用点联接的两个名、如 18、22、26、28、29 行中给出的那样。

我们要重申，一个联合导致联合中包含的所有数据均被存放在相同的物理位置上。这样，在任一时刻只有一个变量是实际可用的。事实亦是如此。名为 fuel _ load，bomb _ load 和 pallets 的变量被存放在相同的物理空间，因此需要程序员来跟踪在某一时刻存放的究竟是哪一个变量。你会注意到在第 26 行为 transport 的 pallets 赋予了一个值，在 28 行为 transport 的 fuel _ load 赋了一个值。当为 fuel _ load 赋值后，为 pallets 赋予的值就被冲掉，并且再也不能使用。因为它存放的位置已被 fuel _ load 占用。C++ 中联合的用法与 ANSI C 中的用法是一样的，除了元素的命名方式之外。

程序剩余部分对你来说应该是易于理解的，当你研究并理解了之后，请编译并执行之。

C++ 类型转换

TYPECONV.CPP　Type conversions

```
01                                          // Chapter 2 – Program 5
02  #include <iostream.h>
03
04  main()
05  {
06  int a = 2;
07  float x = 17.1, y = 8.95, z;
08  char c;
09
10      c = (char)a + (char)x;
11      c = (char)(a + (int)x);
```