

编程自学者福地，官网 <http://afanihao.cn>

C 语言快速入门

Java 快速入门及系列教程

Unity 游戏开发入门及系列教程

Ubuntu / CentOS Linux 等公开课等众多教程，总有一款符合你的需要！

第 1 章 开始学习 C/C++

1.1 开发平台

在开始学习 C/C++ 语言之前，需要先安装好相关的开发工具，或称开发平台。本书推荐安装 VS2008 或 VS2010, VS2012，其中 VS 是 Visual Studio 的简称，它是微软公司推出的开发平台。它支持多种编程语法，C++ 只是其中之一，我们将 VS 里面的 C++ 环境也称为 VC。

理论上，读者也可以自选其他的编译器或开发平台，例如 gcc, Eclipse, Dev-C++, C-Free 等开发工具。但对于初学者来说，为了避免不必要的困扰，还是建议使用 VS 平台。

1.2 第一个程序

跟大多数编程语言的教程一样，我们的要创建的第一个程序也是 HelloWorld 程序。对照本书附录《用 VC2008 创建项目》，来建立我们的第一程序。

```
////////// CH01_A1 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    printf("Hello,World!\n");  
    return 0;  
}
```

在本教程里，对每一个示例进行编号，比如 CH01_A1 就是这个例子的编号。读者可以官网上下载到所有的示例源码。

我们按照附录里的说明，按 F7 编译这个程序，然后按 CTRL+F5 运行程序，将会弹出来一个黑色的窗口。我们把这个窗口称为控制台窗口，简称控制台。如图 1-1 所示。



图 1-1 控制台窗口的显示

1.2.1 代码解析

这个 HelloWorld 程序代码虽然只有 6 行，但是它的语法细节覆盖前 18 章的内容。因此，对初学者来，是没有必要在第一节就掌握它的所有含义的。

对于本书的读者，在前面 7 章的学习中，大家只要把代码框架照抄下来即可。至于其中的语法细节，会循序渐进地给大家介绍。本书示例使用的代码框架为：

```
#include <stdio.h>

int main()
{

    return 0;
}
```

也就是，这几行代码框架是大家暂时不需要关心的，直接照抄就可了。在每一章，我们都会学习一些新的语法，我们会把相关的练习代码填在 `return 0` 这一行的前面。

1.2.2 在 Windows XP 环境下的问题

如果你的操作系统还是以前的 Windows XP，那么可能发现在运行程序的时候，还没来得及查看内容，窗口自己就关闭了。这个问题的解决办法是，使用以下的代码框架：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Hello,World!\n");
}
```

```
    system("PAUSE");  
    return 0;  
}
```

1.2.3 常见的问题

在创建第一个程序时，大家最常见的问题：

(1) 标点符号

在示例中程序中，标点符号均为英文标点（半角）。而初学者经常会把它写成中文的标点（全角），这样在编译时编译器会报错。

例如，大括号内的每一行末尾有个分号，字符串"Hello,World"是用双引号包围的，等等。

(2) 大小写

对于前面所列的代码框架，每一个字母的大小写是严格区分的，不要把 `main` 写成 `MAIN`。

1.3 代码与程序

我们在 `main.cpp` 文件中写的文本，称为代码（Code）。代码在经过 VC 平台的编译处理之后，生成的 `exe` 文件，称为程序（Program）。程序是交给用户来用的，而代码则是程序员自己保留的。

对于程序员来说，代码是核心价值，是智慧和心血的结晶。程序文件丢了，可以花一分钟重新生成一份；代码如果丢了，那就只能重写了。

1.4 C 语言和 C++ 语言

本书一本 C 语言和 C++ 语言通用的教程。实际上，C 语言只是 C++ 语言前身，对应了本书的第 1-19 章。C++ 语言是在 C 语言的基础上，一方面修改正了 C 语言原有的缺陷，另一方面又新增了一些新的语法。当您学全整个教程，就既掌握 C 语言又掌握 C++ 语言了。

第 2 章 控制台输入与输出

控制台是初学者经常要面对的一个界面，它可以输出显示一些字符，也可用键盘输入一些字符。我们今后的练习都是通过控制台来完成的。在这一章，就是初步学习一下，如何向控制台输出显示一些数据，以下如何从控制台接收用户的输入。

1.5 控制台输出

我们使用 `printf` 操作，来向控制台输出数据。其中，`print` 的意思是打印输出，`f` 代表 `format`，整体上就是格式化打印输出的意思。

在示例 CH02_A1 中，我们使用 `printf` 操作输出了两个文本，代码如下所示。

//////////////////// 例 CH02_A1 //////////////////////

```
#include <stdio.h>

int main()
{
    printf("i am shaofa \n");
    printf("我是谁谁谁 \n");
    return 0;
}
```

使用 `printf` 既可以输出英文文本，也可以输出中文文本。由于中文问题在计算机领域是一个复杂的问题，所以作为初学者来说，建议使用英文或拼音来练习。记住：我们的重点是学习它的语法原理，不一定要用中文的。

`printf` 操作的要素包含以下几点：

- ① 小括号：小括号内的东西称为参数列表
- ② 双引号：双引号里的文本就是要输出到控制台内的文本
- ③ `\n`：这个表示换行，具体意义先不用理会，直接照抄即可。

另外，需要对初学者再次强调：

- ① 代码框架照抄即可

在本书示例中，我们总是使用下面的代码框架，至少每一行究竟是什么含义，初学者先不用理会。（涉及了第 1 章~18 章的语句，需逐步介绍）

```
#include <stdio.h>

int main()
```

```
{  
    return 0;  
}
```

② 读者只需要关心书中讲了什么，而不必关心书中没讲的东西

所有的语法，都是以循序渐进地方式讲明；对于没有讲解地部分，照抄示例代码即可。

1.5.1 输出整数

使用 `printf` 可以输出一个整数。下面的例子中，使用 `printf` 输出整数：

////////// 例 CH02_A2 //////////

```
#include <stdio.h>  
  
int main()  
{  
    printf("I am %d \n" , 33);  
    return 0;  
}
```

编译这个代码，运行程序，输出显示如图 2-1 所示。

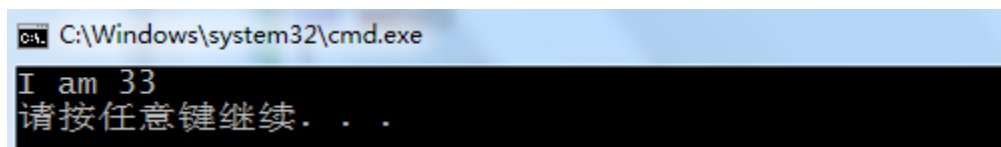


图 2-1 控制台窗口的显示

比较 `printf` 中双引号中的文本，和实际输出的文本，可以发现，在实际的输出结果中对 `%d` 进行了一个替换动作：将 `%d` 替换为了 `33`。

此例中 `printf` 使用的语法的要点：

- ① 小括号内的参数列表以逗号分开，表示有 2 个参数
- ② 第一个参数："`I am %d \n`"，字符串，须以双引号包围
- ③ 第二个参数：`33`，数字

1.5.2 用变量表示整数

我们也可以用下面的方式来输出整数。

////////// 例 CH02_A3 //////////

```
#include <stdio.h>

int main()
{
    int age = 33;
    printf("I am %d \n" , age);
    return 0;
}
```

注意：大括号内的每一行的末尾，都须有一个分号作为结束。

在例 CH02_A3，定义了一个变量 `age`，指定它的值为 33。它的类型为 `int`，代表 `integer`（整数）的意思。

我们可以用这样的句子 `int age = 33;`来定义一个变量，然后在 `printf` 里打印输出这个变量。同样的，`printf` 操作会把 `%d` 替换为 `age` 的值。最终输出结果和前一例相同，如图 2-2 所示。

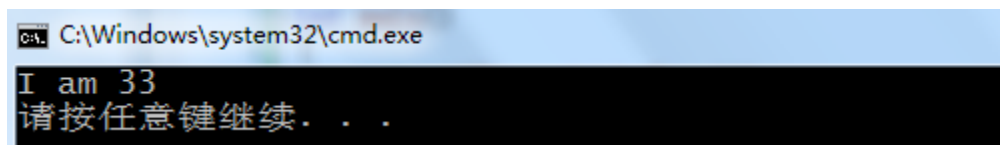


图 2-2 控制台窗口的显示

也可以在同一行内打印两个整数，例如，

////////// 例 CH02_A4 //////////

```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 1209890;
    printf("a: %d, b: %d \n", a, b);
    return 0;
}
```

注意，小括号内有 3 个参数，依次用逗号隔开。

原理仍然一样，`printf` 操作会把 `%d` 替换成后面的数值。此例的运行结果输出如图 2-3 所示。



图 2-3 控制台窗口的显示

1.5.3 指定显示宽度

例如，当我们输出数字 9，默认它只占一位宽度，但我们可以使用%4d 来控制它显示为四位宽度。同理，%5d 表示显示为五位宽度，%8d 表示显示为八位宽度。

下面，我们通过两个例子，来比较一下%d 来%4d 的显示效果。

例 CH02_A5 中使用了%d 来显示数字，

////////// 例 CH02_A5 //////////

```
#include <stdio.h>

int main()
{
    printf("number : %d ,OK\n", 3);
    printf("number : %d ,OK\n", 33);
    printf("number : %d ,OK\n", 333);
    return 0;
}
```

运行结果如图 2-4 所示。

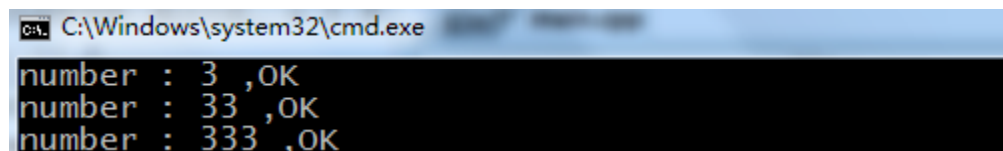


图 2-4 控制台窗口的显示

例 CH02_A6 中使用了%4d 来显示数字，

////////// 例 CH02_A6 //////////

```
#include <stdio.h>

int main()
{
    printf("number : %4d ,OK\n", 3);
    printf("number : %4d ,OK\n", 33);
    printf("number : %4d ,OK\n", 333);
    return 0;
}
```

```
}
```

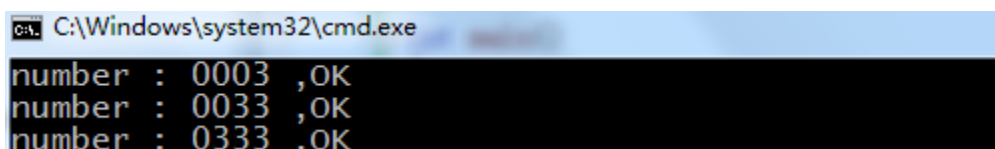
输出结果如图 2-5 所示。



```
C:\Windows\system32\cmd.exe
number : 3 ,OK
number : 33 ,OK
number : 333 ,OK
```

图 2-5 控制台窗口的显示

我们还可以使用%04d 来控制显示宽度，表示当位数不足 4 位时，前面填 0 显示。显示效果如图 2-6 所示。



```
C:\Windows\system32\cmd.exe
number : 0003 ,OK
number : 0033 ,OK
number : 0333 ,OK
```

图 2-6 控制台窗口的显示

1.5.4 输出小数

可以使用 printf 操作来输出小数，使用的格式符为%lf （lf 代表 long float-point）。

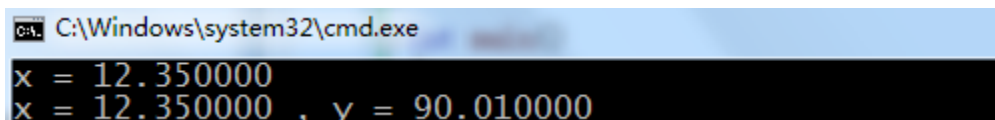
在示例 CH02_B1 中，第一行 printf 输出一个小数，第二行输出了两个小数。代码如下：

////////// 例 CH02_B1 //////////

```
#include <stdio.h>

int main()
{
    printf("x = %lf \n", 12.35);
    printf("x = %lf , y = %lf \n", 12.35, 90.01);
    return 0;
}
```

编译代码，运行程序，结果显示如图 2-7 所示。



```
C:\Windows\system32\cmd.exe
x = 12.350000
x = 12.350000 , y = 90.010000
```

图 2-7 控制台窗口的显示

可见，对于小数来说，printf 就是把%lf 替换为后面的小数的值。

1.5.5 用变量表示小数

我们也可以下面的方式，用一个变量来表示小数。

////////// 例 CH02_B2 //////////

```
#include <stdio.h>

int main()
{
    double a = 12.35;
    printf("x = %lf \n", a);
    return 0;
}
```

在例 CH02_B2 中，定义了一个变量 a，其类型为 double，表示小数。

1.5.6 指定小数点后的位置

我们可以控制小数点后的位数显示。例如，%.2lf，表示在显示输出的时候，小数点后面只保留 2 位，后面的值四舍五入。例如，

////////// 例 CH02_B3 //////////

```
#include <stdio.h>

int main()
{
    double a = 12.35719987;
    printf("x = %.2lf \n", a);
    return 0;
}
```

输出显示如图 2-8 所示。



图 2-8 控制台窗口的显示

1.6 控制台输入

可以 scanf 操作，让用户从控制台输入一个整数或小数。其中，scan 表示输入扫描，f 表示 format，表示接收输入并格式化数据的意思。

1.6.1 输入整数

下面的例子中，使用 `scanf` 操作来接收一个整数输入。

//////////例 CH02_C1 //////////

```
#include <stdio.h>

int main()
{
    int a = 0;

    scanf("%d", &a);

    printf("Got: %d \n", a);

    return 0;
}
```

在例 CH02_C1 中，首先定义了一个 `int` 类型的变量 `a`，然后调用 `scanf` 操作，保存用户的输入。

编译代码，运行这个程序。在光标闪烁的时候，表示此时用户可以输入一个数（如 123）。用户输入一个整数，按回车结束输入。整个显示如图 2-9 所示。



图 2-9 控制台窗口的显示

这表明，`scanf` 可以接收用户的输入，转化成数值保存在变量 `a` 中。

此时，我们应该注意一下 `scanf` 行的写法：

- ① 小括号内的是参数列表，以逗号分隔
- ② 第一个参数是一个字符串，以双引号包围
- ③ 第二个参数，记得要在 `a` 前面加一个 `&` 号
- ④ 字符串内不要加多余的空格、或其他字母和标点，不是无法接收输入。

初学者往往因为在双引号内加了多余的字符，而得不到正确的结果。例如，如果写成下面这样，是得不到数据的：

```
scanf("%d\n", &a); // 错！双引号内不要加\n
```

1.6.2 输入小数

输入小数的过程也很类似，要用一个 `double` 型变量来存放小数，并使用格式符 `%lf`。下面的例子展示了小数的输入。

////////// 例 CH02_C2 //////////

```
#include <stdio.h>

int main()
{
    double f = 0;
    scanf("%lf", &f);
    printf("Got: %lf\n", f);
    return 0;
}
```

编译代码并运行程序，在光标闪烁时，输入一个数 12.345，按回车结束输入。输出显示如图 2-10 所示。



图 2-10 控制台窗口的显示

1.6.3 一次输入多个数

当需要输入多个数时，有两种方法。本教程推荐第（1）种方法，因为它准确可靠，初学者易于掌握、不容易出错。

第（1）种方法：

可以一次接收一个数：每输入一个数，按一次回车。例如，

////////// 例 CH02_C3 //////////

```
#include <stdio.h>

int main()
{
    int a;
    double x;
    scanf("%d", &a);
    scanf("%lf", &x);
}
```

```
printf("Got: %d, %lf \n", a, x);  
return 0;  
}
```

运行程序，输入 12，按回车，输入 123.456，按回车。程序输出结果如图 2-11 所示。



图 2-11 控制台窗口的显示

第（2）种方法：

也可以在一行内输入多个数据，每个数据以逗号分隔，然后按回车。例如，

////////// 例 CH02_C4 //////////

```
#include <stdio.h>  
  
int main()  
{  
    int a;  
    double x;  
    scanf("%d,%lf", &a, &x);  
    printf("Got: %d, %lf \n", a, x);  
    return 0;  
}
```

在例 CH02_C4 中，一行 scanf 便可以接入两个数，注意双引号内的字符串 %d 和 %lf 是以逗号分开的。

运行此程序，一次性地输入 2 个数，并以英文逗号隔开，按回车。输出结果如图 2-12 所示。

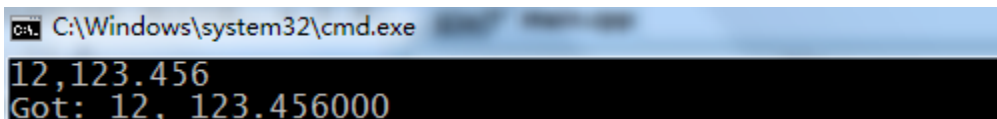


图 2-12 控制台窗口的显示

1.7 注释

可以在代码里加一点注释性的文字，它们不影响原代码的运行。注释的作用，是为了让程序员在阅读起来更清晰，相当于备注的作用。

在 C/C++ 里，有两种注释语法。一种以 // 开头，表示单行注释语句。另一种以 /* */ 包围，表示任何行数的注释。

第一种注释方法（单行注释语法）：如果你只想写一行注释，则可以用 // 开头，编译器把该行里面 // 后面的所有文字视为注释。

```
////////// 例 CH02_D1 //////////  
  
#include <stdio.h>  
  
// 我的第一个测试  
  
int main()  
{  
    double a; // 定义 1 个变量  
    scanf("%lf", &a); // 接收输入  
    printf("Got: %lf \n", a); // 输出显示这个数  
    return 0;  
}
```

第二种注释方法（多行注释语法）：如果注释文字多于一行，可以用 /* 和 */ 包围起来，编译器将中间的文字视为注释。

```
////////// 例 CH02_D2 //////////  
  
#include <stdio.h>  
  
/* 我的第一个测试: 验证 printf 的功能  
它成功了!  
*/  
  
int main()  
{  
    double a = 2.0123;  
    printf("%.3lf \n", a);  
    return 0;  
}
```

1.8 空白

在 C/C++ 代码里，为了美观和可读性，可以在代码里适当一些空白，并不会对程序的产生任何影响。在这里，空白包括空格、制表符和换行。

例如，

```
int a=10;
```

我们可以多点空白，不会对结果产生任何影响。

```
int    a    =    10    ;
```

我们甚至可以把它写在多行里面，在语法上没有影响，结果还是一样，只是代码读起来就比较难看了。例如，

```
////////// 例 CH02_E1 //////////
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int a
```

```
        = 10;
```

```
    printf("Got: %d \n",
```

```
        a);
```

```
    return 0;
```

```
}
```

这样的代码书写方式，虽然在语法上是正确的，而且结果也是正确的。但我们很不提倡，因为它太难以阅读性，降低了代码的可读性。

1.9 常见问题

(1) 格式和类型要对应

在使用 `printf` 和 `scanf` 时，格式符要和类型对应起来。

例如，下面的代码在运行时会有问题：

```
int a = 1;
```

```
double b = 2.2;
```

```
printf("%lf, %d ", a, b);
```

应该是

```
printf("%d , %lf ", a, b);
```

因为%d 是用于打印整数，而%lf 是用于打印小数的，类型要对应起来。

(2) scanf 内加了多余的字符

例如，

```
int a;
```

```
scanf("%d \n", &a); // 错了，不要加空格，也不要加\n
```

1.10 综合例题

写一个程序，让用户输入两个数（允许是小数），然后将这两个数的乘积输出。

```
//////////例 CH02_F1 //////////
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    double a = 0; // 第一个数
```

```
    double b = 0; // 第二个数
```

```
    printf("请输入第一个数: "); // 提示用户输入
```

```
    scanf("%lf", &a);
```

```
    printf("请输入第二个数: "); // 提示用户输入
```

```
    scanf("%lf", &b);
```

```
    double result = a * b;    //计算结果
```

```
    printf("%.3lf x %.3lf = %.3lf \n", a, b, result);
```

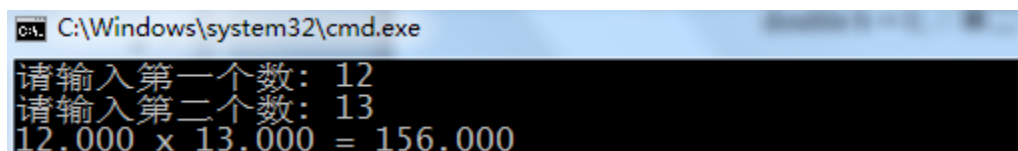
```
    return 0;
```

```
}
```

在这个程序中，首先定义了两个变量 a,b 用于存放小数。然后提示用户输入，当输入的数存到变量 a,b 里。然后用一个算式 $result = a * b$ 来求出乘积，最后将算式和结果打印出来。

注：在 C/C++ 里，乘法符号是*，其具体原理和用法后续会讲，现在只要求会模仿使用即可。

下面是这个程序的输出结果如图 2-13 所示。



```
C:\Windows\system32\cmd.exe
请输入第一个数: 12
请输入第二个数: 13
12.000 x 13.000 = 156.000
```

The image shows a Windows command prompt window with a blue title bar. The title bar text is 'C:\Windows\system32\cmd.exe'. The command prompt has a black background with white text. It shows two prompts: '请输入第一个数:' followed by the input '12', and '请输入第二个数:' followed by the input '13'. Below these, the result of the multiplication is displayed as '12.000 x 13.000 = 156.000'.

图 2-13 控制台窗口的显示

第3章 变量与常量

本章介绍常量和变量的概念。常量是不可以修改的量，变量是可以变化的量。介绍几种基本的变量类型：用于表示整数的类型 `bool/char/short/int`，用于表示小数的类型 `float/double`。

1.11 变量

变量表示可以变化的量。我们在下面的例子中，来观察变量的用法。（注意，前面的行号只是为了方便说明而加上去的，实际写代码的时候不需要输入行号）

////////// 例 CH03_A1 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int a = 1;
05      a = 2;
06      printf("a: %d \n", a);
07      a = 3;
08      printf("a: %d \n", a);
09      return 0;
10 }
```

在示例 CH03_A1 中，

第[04]行：定义一个变量 `a`，用于表示整数，初始值为 1；

第[05]行：将 `a` 赋值为 2，此时 `a` 的值变成 2；

第[06]行：将整数 `a` 打印出来；

第[07]行：将 `a` 赋值为 3，此时 `a` 的值又变成了 3；

第[08]行：再次将整数 `a` 打印出来；

现在，我们可以认识到变量就是一个可以变化的量，通过等号可以改变它的值。而对于代码中的其他行，读者暂时不必关心，随着学习的深入，会逐步给大家介绍。

下面我们具体讨论变量的相关语法细节。

1.11.1 变量的定义

定义一个变量时，要指定以下几个要素：①变量名 ②变量类型 ③ 初始值（可选）。

例如，下面一行代码就是定义了一个变量，

```
int abc = 10;
```

我们分析一下：

- ① 变量名：abc
- ② 变量类型：int，表示它是一个整数类型
- ③ 10：指定其初始值为 10

另外，记得末尾要加上一个分号，否则就是语法错误。

下面我们再看一个例子，在这个例子中定义了多个变量。

////////// 例 CH03_A2 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int a = 1;
05      int b;
06      double c = 10.1;
07      double d;
08      b = 2;
09      d = 10.2;
10      printf("a:%d b:%d c:%.3lf d:%.3lf\n", a, b, c, d);
11      return 0;
12 }
```

在示例 CH03_A2 中，

第[04]行：定义了一个变量 a，类型为 int，表示整数，初始值为 1；

第[05]行：定义了一个变量 b，类型为 int，表示整数，没有指定初始值；

第[06]行：定义了一个变量 c，类型为 double，表示小数，初始值为 10.1；

第[07]行：定义了一个变量 d，类型为 double，表示小数，没有指定初始值；

第[08]行：将 b 赋值为 2

第[09]行：将 d 赋值为 10.2

第[10]行：将变量 a, b, c, d 的值打印显示

1.11.2 变量的命名

在 C/C++中，变量的命名规则为：必须是字母、数字、下划线的组合。可以用字母或下划线开头，但不可以用数字开头。

下面是正确的变量名，

```
int name12 = 0;
int myage = 33;
int good_bye = 0;
```

下面是不正确的变量名，

```
int 12name = 0; // 错误！不能以数字开头！
int my age = 33; // 错误！不能含有空格！只能是字母、数字、下划线的组合！
int good-bye = 0; // 错误！不能含有横杠！只能是字母、数字、下划线的组合！
```

1.11.3 变量的赋值

在 C/C++中，使用等号=可以改变一个变量的值，我们把这个操作称为“赋值”。我们将等号称为赋值操作符，其作用是“把等号右侧的值赋给左侧的变量”。

在下例中，定义了变量 a、b、m，并用赋值操作符来改变它们的值。

////////// 例 CH03_A3 //////////

```
#include <stdio.h>

int main()
{
    int a ;
    a = 1; // 将变量 a 赋值为 1
    a = -2; // 将变量 a 赋值为-2
    double b ;
    b = 1.2; // 将变量 b 赋值为 1.2
    b = -2.2; // 将变量 b 赋值为-2.2
    int m;
    m = a; // 将 a 的值赋给变量 m
    return 0;
}
```

1.12 整型变量

不仅 `int` 类型可以用于表示整数，还有一些其他的类型，也可以用于表示整数。

1.12.1 `char / short / int` 类型

最常用的三种整数类型为 `char`, `short`, 和 `int`, 它们都可以用于表示整数。例如,

```
char a = 12;

short b = 1280;

int c = 1280000;
```

它们的区别在于表示范围不同, 一个 `char` 型变量只能表示从 -128 到 127 之间的整数, 而一个 `int` 型变量则可以表示很大的数[-2147483648, 2147483647]。这意味着, 当一个数比较小、位于 -128 到 127 之间时, 我们可以用 `char` 型变量来表示; 而超出了这个范围时, 就不能用 `char` 型变量来表示。

在示例 CH03_B1 中, 用一个 `char` 型变量表示 1224, 再把变量的值打印显示出来。由于 1224 这个值超出来 `char` 所能表示的范围, 所以打印显示的结果和我们预期的不一样。

```
////////// 例 CH03_B1 //////////

#include <stdio.h>

int main()
{
    char a = 1224;

    printf("value : %d \n", a);

    return 0;
}
```

对于我们初学者而言, 我们大多数时间使用 `int` 型来表示整数, 它是足够大的。在少数情况下, 我们只需要记住 `char` 的表示范围就可以了。

当用 `scanf` 来输入一个整数时, 只能用 `int` 类型, 不能用 `char` 或 `short` 类型。例如,

```
////////// 例 CH03_B2 //////////

#include <stdio.h>

int main()
{
    char a = 0;

    scanf ("%d", &a); // 错! 不能用 char 或 short 接收用户输入! 只能用 int 型变量!

    return 0;
}
```

```
}
```

当用 `printf` 来输出一个整数时，`char`, `short`, `int` 型都可以输出。例如，

```
////////// 例 CH03_B3 //////////
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    char a = 12;
```

```
    printf("a: %d \n", a); // 用 printf 可以输出 char 和 short 型变量
```

```
    return 0;
```

```
}
```

1.12.2 unsigned 无符号类型

无符号类型用于表示非负整数，即大于或等于 0 的数。常用的类型为 `unsigned char`、`unsigned short` 和 `unsigned int`。

例如，

```
    unsigned char a = 12;
```

```
    unsigned int = 120900;
```

同样地，三种无符号类型区别在于它们的表示范围。`unsigned char` 的表示范围最小，是从 0 到 255。而 `unsigned int` 则最大，从 0 到 4294967295。

对于初学者来说，只需要记住 `unsigned char` 的表示范围即可。当我们要表示一个较大的正整数时，直接用 `unsigned int` 即可。

无符号类型不能用于表示负数，例如，下面的代码是有问题的，

```
    unsigned char a = -12; // 不能表示负数
```

注：事实上，前面所学的 `char/short/int` 只是 `signed char/signed short/signed int` 的简写，而关键字 `signed` 给省去了而已。至于 `unsigned` 与 `signed` 之间的本质联系，参考本书附录《有符号整数与无符号整数》。

在用 `printf/scanf` 调用中，无符号整数用 `%u` 作为控制符。同样地，在用 `scanf` 接收输入时，只能使用 `unsigned int`，不能用 `unsigned char` 或 `unsigned short`。

```
////////// CH03_B4 //////////
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```

    unsigned int a = 0;

    scanf("%u", &a); // 输入时只能使用 unsigned int 来接收，不能使用 unsigned char/short

    printf("Got: %u \n", a);

    return 0;

}

```

1.12.3 *bool 布尔类型

(*初学者请跳过本节。在第 20 章以后才会用到)

C++引入 `bool` 类型，用于表示布尔型(boolean)数值，它本质上是一种整数类型。只有两种取值: `true` 或 `false`，注意 `true` 和 `false` 是字面常量，它们是 C++关键字，不是普通文本。

例如，

```

bool ready = true;

bool on = false;

```

`bool` 类型本质上等同于 `char`，使用 `sizeof` 操作可以发现其大小为 1 字节。事实上，字面常量 `true` 的值就是整数 1，`false` 的值就是整数 0。可以用 `printf` 来打印一下它们的值。

```

printf(" %d, %d \n", true, false);

```

程序员可以从不使用 `bool` 这个类型，也能解决所有的问题。事实上，C/C++程序员一般用 `int` 或 `char` 来解决问题。例如，

```

int ready = 1; // 0 表示“假”，非 0 表是“真”

int on = 0;

```

1.12.4 *enum 枚举类型

(*初学者请跳过本节，具体内容见本书附录《枚举类型》)

1.12.5 *long long 长整数类型

(*初学者请跳过本节)

对于初学者来说，32 位整型一般是够用的了 (`int` 和 `unsigned int`)。还有一种表示范围更大的整数类型，`long long` 和 `unsigned long long`，它们是 64 位的整数、占 8 字节的内存。读者对此类型有点印象即可。

1.13 浮点型变量

用于表示小数的类型有两种: `double` 和 `float`，通常称为浮点型(float-point)。

`double` 和 `float` 的主要区别也是表示范围不同。`float` 比 `double` 可以表示的范围要小得多，但对于初学者来说，只需要记住当需要表示高精度的小数时应该用 `double`；当精度要求不高时（7 位有效数字），可以用 `float`。

下面的例子中，定义了 `float` 型的变量，

```
float a = 3.14f; // 注意：数字后面要加一个 f
```

```
float b = -87.9f; // 注意：数字后面要加一个 f
```

在 `printf/scanf` 中，`float` 型 `"%f"` 作为控制符，`double` 型用 `"%lf"` 作为控制行。

例如，

```
float a = 10.135;
```

```
printf("%.2f ", a); // 输出 float 型
```

```
float b;
```

```
scanf ("%f", &b); // 从控制台接受输入
```

事实上，`lf` 代表的是 `long float-point`，而 `f` 代表的是 `float-point`。

1.14 数的进制表示

在计算机领域，常用二进制、十进制和十六进制来表示一个数。

1.14.1 数的十进制表示

日常生活中我们用的数都是十进制表示的。举例来说，家里养了几只兔子，要表示兔子的数量，人们发明了几个符号：0 1 2 3 4 5 6 7 8 9，其中，0 表示没有兔子，9 表示养了九只兔子。

随着数量的增多，当超过九只之后，人们没有发明更多的符号来表示它，而是用多位符号来表示一个数字。例如，符号"123"表示的数量是一百二十三，从数学上看，每一个符号代表的权重是不一样的。举例来说：

$$123 = 1 \times 100 + 2 \times 10 + 3 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

$$9527 = 9 \times 10^3 + 5 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$

所谓十进制，就是一共有十个符号序列，逢十进一。每一位的权重是 10 的 $N-1$ 次方， N 表示其位置。这就是十进制的含义。

1.14.2 数的十六进制表示

十六进制和十进制本质上是同一个意思：人们发明了十六个符号：0 1 2 3 4 5 6 7 8 9 A B C D E F，依次代表的兔子的数量为零、一、二、三、四、...、九、十、...、十五。逢十六进一，所以再往上增长就是 10, 11, ..., 1A, 1B, ... 1F, 20, 21, ...

对于一个十六进制的表示的数，其每一位的权重是 16 的 N-1 次方。据此可以很容易地将一个十六进制的数换算成我们熟悉的十进制。

例如，

$$A2(x) = A * 16^1 + 2 * 16^0 = 162 \text{ (d)}$$

$$A2B3(x) = A * 16^3 + 2 * 16^2 + B * 16^1 + 3 * 16^0 = 41651 \text{ (d)}$$

为了区别不同的进制，我们在数字后面标注(x)表示十六进制（Hex, Hexadecimal），用 (d) 标识十进制（Decimal），用(b)标识二进制（Binary）。这只是我们在本书中自己定义的一种描述方式。

在 C/C++语法中，用 0x 或 0X 开头来表示十六进制的数。

例如，下面定义的两个整型变量的初始值都是 12。

```
int a = 12; // 默认为十进制
```

```
int b = 0x0C; // 当以 0x 开头时，以十六进制表示。
```

注意，可以写成 0x0C 或 0x0c，都是允许的。

1.14.3 数的二进制表示

在掌握了 16 进制之后，2 进制的问题也就迎刃而解，它们的数学原理都是一样的。对于 2 进制来说，人们发明了一共两个符号：0 和 1，再往上增长就要用多位符号来表示了，每一位的权重是 2 的 N-1 次方。因此，二进制数是这么增长的，例如从零到七的二进制表示：

0

10

11

100

101

110

111

根据每一位的权重，也能很容易地将二进制换算成我们所熟悉的十进制的数。

$$00000000 = 0 \text{ (十进制)}$$

$$00000001 = 1 \text{ (十进制)}$$

$$00001101 = 13 \text{ (十进制)}$$

$$11111111 = 255 \text{ (十进制)}$$

计算方法:

$$00001101 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$

1.15 变量与内存

本节内容至关重要。

1.15.1 二进制存储

在计算机领域，数字最终是以二进制存储和表示的。这是因为二进制只有两个符号，容易用物理量来代表。

例如，一共设八个物理开关，其中√标识是开，否则为关。

√		√			√	√	
---	--	---	--	--	---	---	--

我们根据其状态对其进行数字化，把开当成 1，关当成 0，

1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---

这意味着，这组开关的物理状态可用于表示数字 10100110 (b)。

在计算机领域，把每一个开关称为位（Bit），八个位构成一个字节（Byte）。显然，一个字节能够表示从 00000000(b)到 11111111(b)共 256 个数。

1.15.2 内存

内存用于存储数据的物理器件，拆开电脑，一般可以发现一块长方形的内存条。内存由很多个单元组成，每个单元可以认为是八个物理开关组成的，因此它可以表示一个八位的二进制数，即一个字节的数。

那么一般的内存条里有多少个单元呢？这个容易是很大的，一般以 GB 为单位。在计算机里，一般用 KB、MB、GB、TB（B 代表 Bytes）这几个单位来衡量大小。它们的关系是：

$$1 \text{ KB} = 2^{10} \text{ B} , (\text{Kilo Bytes})$$

$$1 \text{ MB} = 2^{10} \text{ KB}, (\text{Mega Bytes})$$

$$1 \text{ GB} = 2^{10} \text{ MB}, (\text{Giga Bytes})$$

$$1 \text{ TB} = 2^{10} \text{ GB}, (\text{Tera Bytes})$$

其中 $2^{10}=1024$ ，近似为 1000，所以也近似等于日常生活中的 1，1000，1000000，1000000000 这样的千倍关系。

1.15.3 变量的大小

当程序运行时，每一个变量其实都对应于一块内存。而变量的值，其实就是物理内存里那几个字节里存储的数据。

我们知道，一个 `char` 型变量可以表示的范围很小，只能表示 $[-128, 127]$ 。而一个 `int` 型变量的表示范围则很大，能表示一般对应了 4 个字节的内存，能表示 $[-2147483648, 2147483647]$ 。为什么有这么大差异呢？其本质原因是：在内存里，`char` 型变量占据了一个字节，而 `int` 型变量占据了四个字节。一个字节能表示的范围是 `00~FF`，只能表示 256 个数。四个字节表示的范围则能表示 `00000000~FFFFFFFF`，这是一个很大的范围。

我们把变量在内存里所占的字节数，称为变量的大小。在 C/C++ 语言里，可以用操作符 `sizeof` 来测量一个变量或类型的大小。在示例中，用 `sizeof` 来测量 `int` 类型的大小，其结果为 4。

////////// 例 CH03_C1 //////////

```
#include <stdio.h>

int main()
{
    int a = 0;

    printf("size: %d \n", sizeof(a)); // 用 sizeof 测量变量 a 的大小
    printf("size: %d \n", sizeof(int)); // 用 sizeof 测量类型 int 的大小

    return 0;
}
```

用同样的方法测量其他类型的大小，结果为：`char(1)`，`short(2)`，`int(4)`，`float(4)`，`double(8)`，显然地，如果一个类型占的字节越多，它能表示的范围就越大、精度就越高。例如，`double` 占了 8 个字节，而 `float` 只有 4 个字节，因而 `double` 要比 `float` 能存储更多的信息（存储更多的有效数字）。

1.15.4 变量在内存中的表示

我们以 `char` 型和 `int` 型变量为例，阐述变量在内存中的表示。

在代码中定义两个变量，为了方便描述，使用十六进制来表示数字。

```
char a = 0x1A; // 即十进制的 26
```

```
int b = 0x1A2B3C4D; // 即十进制的 439041101
```

变量 `a` 和 `b` 在内存中的表示为：



虽然我们不知道在一个容量巨大的内存条中，究竟是哪几个单元对应了变量 `a`，`b`。但我们可以肯定的是，至少是存在着一个单元对应于 `a`，存着四个单元对应于 `b`。

注：实际上，我们可以在 VC 中直接观测到变量在内存中的几个单元，具体方法参考本书附录《VC2008 调试方法》。例如，我们可以观察到变量 b 在内存中对应的那个字节，如图 3-1 所示，对应的几个字节已经被圈中标识。

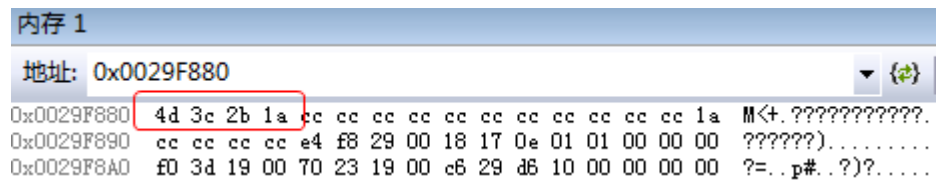


图 3-1 VC 的内存窗口

1.15.5 变量的地址

我们对内存中的每个单元进行编号，从 0 开始依次增长，依次为 0, 1, 2, ..., 1000, 1001, ..., 1000000000, 1000000001, ... 把这个编号称为内存单元的地址。那么每个内存单元都有一个地址，用十六进制表示的话，地址范围是 00000000~FFFFFFFF。

在程序运行时，对于每一个变量，它在内存中都对应了几个内存单元。我们把它对应的内存单元的首地址（第一个字节的地址），称为变量的内存地址，简称为变量的地址。显然，变量的地址是一个整数。

通过操作符&可以取得变量的地址。在示例 CH03_C2 中，用&a 取得 a 的地址，并用格式符%08X 将这个地址以 16 进制形式打印显示。

```
////////// CH03_C2 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    char a = 0x1A;  
    int b = 0x1A2B3C4D;  
    printf("a : address = %08X \n", &a); // &a 表示变量 a 的地址  
    printf("b : address = %08X \n", &b); // &b 表示变量 b 的地址  
    return 0;  
}
```

1.15.6 理解变量的赋值

变量的值，其实就是这个变量对应的那几个内存单元的值。

因而，当我们在代码中对变量赋值的时候，其实修改了那几个内存单元的值。

//////////示例 CH03_C3 //////////

```
01 #include <stdio.h>
02 int main()
03 {
04     int b = 0x1A2B3C4D;
05     b = 0x50505050;
06     return 0;
07 }
```

程序执行第[04]行后，b 对应的内存的值如图 3-2 中的圈中所示。

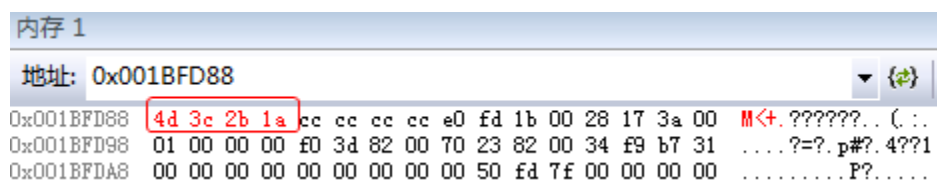


图 3-2 VC 的内存窗口

程序执行第[04]行后，b 对应的内存的值发生了变化。如图 3-3 中圈中所示。

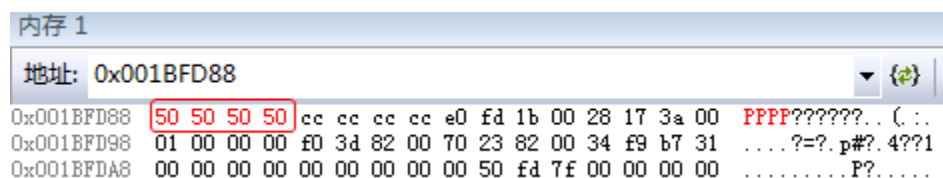


图 3-3 VC 的内存窗口

1.16 const 常量

当在类型名前面加上关键字 `const` 后，表示它是一个只读的量。这种变量不能修改它的值，因而称为常量。例 CH03_D1 中，试图对 `const` 常量 `a` 和 `pi` 进行赋值操作，这是语法不允许的，编译器会报错。

////////// 例 CH03_D1 //////////

```
#include <stdio.h>

int main()
{
    const int a = 10;

    const double pi = 3.1415926;

    a = 11; // 错误! 不允许赋值操作!
```

```

    pi = 3.1415927; // 错误! 不允许赋值操作!

    return 0;
}

const 常量是只读的，可以读取它的值，或者用 printf 打印出来。

////////// 例 CH03_D2 //////////

#include <stdio.h>

int main()
{
    const int MAX_SIZE = 1024;

    printf("a: %d \n", MAX_SIZE); // 读取 const 常量的值

    int b = MAX_SIZE; // 将 MAX_SIZE 的值赋值给变量 b

    printf("b: %d \n", b);

    return 0;
}

```

1.17 *字面常量

（*初学者可以跳过本节）

字面常量（Literal Constant），指的是在代码里写在字面上的值。

例如，

```

    int a = 123;

    那么代码里的 123 字样，就是一个字面常量。

    在例 CH03_D1 中，对我们已经认识的字面常量作了注释。

    ////////// 例 CH03_D1 //////////

    #include <stdio.h>

    int main()
    {
        int a = 123; // 123 是字面常量

        double b = 3.14159; // 3.14159 是字面常量

        float c = 123.4f; // 123.4f 是字面常量

        printf("%d %lf %f \n", a, b, c);
    }

```

```
printf("I have %d numbers \n", 3); // 3 是字面常量
return 0;
}
```

对于初学者来说，暂时不需要对字面常量有深度的理解。目前只需要记住一点：常量不可以被赋值。

例如，以下代码是有语法错误的：

```
#include <stdio.h>
int main()
{
    10 = 10; // 错误！常量不能被赋值！
    12 = 13; // 错误！常量不能被赋值！
    return 0;
}
```

1.17.1 字面常量的类型

C/C++是强类型的语言，所有的变量和常量都是有类型的。例如，12 是 `int` 型，12.0 是 `double` 型，12.0f 是 `float` 型。

例如，在给变量赋值的时候，要注意赋值符左右两侧的类型是否匹配：

```
int a = 12;
double b = 12.0;
float c = 12.0f;
```

1.18 常用类型的范围

C/C++里常用的类型，及表示范围如表 3-1 所示。

表 3-1 常用的变量类型

类型	sizeof	表示范围	说明
char	1	-128~127	$-2^7 \sim (2^7 - 1)$
short	2	-32768~32767	$-2^{15} \sim (2^{15} - 1)$
int	4	-2147483648~2147483647	$-2^{31} \sim (2^{31} - 1)$
unsigned char	1	0~255	$0 \sim (2^8 - 1)$
unsigned short	2	0~65535	$0 \sim (2^{16} - 1)$
unsigned int	4	0~ 4294967295	$0 \sim (2^{32} - 1)$

float	4	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$	7 位有效数字
double	8	$-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$	15 位有效数字
long long	8	很大	$-2^{63} \sim (2^{63}-1)$
unsigned long long	8	很大	$0 \sim (2^{64}-1)$

第 4 章 数组

本章介绍数组的语法和使用。数组用于表示若干个相同类型的量。

1.19 引例

下面例子将引出一个问题，本章的语法规则是用于解决这种问题。

设一个学生的成绩在 0~100 之间，则可以用 `char` 型变量来表示：

```
char a = 98;
```

设一个班有 30 个学生，如何表示他们的成绩呢？可以用 30 个 `char` 型变量来表示：

```
char a00 = 98;
```

```
char a01 = 95;
```

```
...
```

```
char a29 = 88;
```

一共定义 30 个变量，`a00~a29`，每个变量表示一位同学的成绩。这样虽然可以勉强完成数据的表示，但是有一个显然的问题：代码过于臃肿和机械。如果 30 个同学的成绩要用 30 个变量变量，那如果有 100 个同学，难道要定义 100 个不同的变量吗？

在 C/C++ 里，使用数组的语法，只需要定义一个变量，就可以表示 100 个同学的成绩：

```
int arr[100];
```

1.20 数组的定义

数组用于表示一组数值。例如，

```
char arr[5];
```

其中，`arr` 称为“数组变量”，简称“数组”。它表示 5 个 `char` 型数据。我们把每一个数据称为一个“元素”。

数据组的定义中包含以下几个要求：

- ① 元素类型：上例中元素类型为 `char`
- ② 元素的个数：中括号内指定个数，例如上面的数组长度为 5
- ③ 数组的名称：上例中数组的名称为 `arr`

数组的意义，是相当于把 N 个同类型的变量排列在一起。比如，对于 `char arr[5]` 就是说把 5 个 `char` 排列在一起。

同样的，我们定义基本类型的数组。例如，

```
////////// CH04_A1 //////////  
  
char alpha[10]; // 10 个 char 型元素  
  
short years[20]; // 20 个 short 型元素  
  
int numbers[30]; // 30 个 int 型元素  
  
float scores[3]; // 3 个 float 型元素  
  
double values[12]; //12 个 double 型元素
```

1.20.1 数组的命名

数组的命名规则和变量名的规则相同，即“数字、字母、下划线的组合，但不能以数字开头”。

首先，变量的名字要符合其意义，这要求我们在命名时要能做到“词能达义，顾名思义”，不要给变量起一个不相关的名字。

其次，数组变量的名字一般使用小写字母。如果由多个单词组成，则中间以下划线分开。

注：你可以较为随意的定义一个变量名，但是这样“随意”地写代码，会使代码难以阅读，“可读性”降低。因此建议按照推荐的方式来给变量命名。

1.20.2 数组的长度必须是常量

数组的长度在中括号内指定，必须是一个整型常量。例如，

```
int arr[12];
```

不能用变量来表示一个数组的长度，例如，以下定义是错误的，

```
int size = 12;
```

```
int arr [ size ]; // 编译器报错！数组的长度必须是常量！
```

1.21 数组的基本方法

1.21.1 数组的初始值

可以定义数组的时候，指定每一个元素的初始值。例如，

```
char arr[5] = { 90,91,92,93,94 };
```

其语法要素为：

- ① 使用大括号，大括号末尾加上分号
- ② 大括号内指定初始值，每个初始值以逗号隔开，但最后一个数组末尾不加逗号

下面再介绍一些特殊的写法。

（1）不指定初始值

定义时可以不初始化。例如，

```
char arr[5]; // 定义了一个长度为 5 的 char 型数组，不指定初始值
```

（2）只指定一部分初始值

```
char arr[5] = {90,91 }; // 只指定前 2 个元素的初始值
```

注意：此种情况下，只能定义前几个元素的值。

如果在定义的时候只能给出后面几个元素的值，则必须手工地把前面的元素设置一个初始值，例如，

```
char arr[5] = {0,0,0, 90,91 }; // 只知道后 2 个元素的值，于是把前 3 个元素的值设为 0
```

（3）只有初始值，没有长度

中括号的长度可以省略不写。当长度不写明时，编译器会根据初始化列表中的元素的个数来计算其长度。

```
char arr[ ] = { 90,91, 92, 93 }; // 中括号没有写明长度，编译计算得到其长度为 4
```

（4）按位清零

```
char arr[5] = { 0 };
```

则所有元素的值都是 0

1.21.2 访问数组中的元素

例如，用一个数组变量 `values` 来存放 5 个 `int` 型整数，

```
int values[5] = {1, 2, 3, 4, 5};
```

则用 `values [0]`表示数组的第 1 个元素，

例如，

```
////////// CH04_B1 //////////
```

```
values[0] = 11; // 将第 1 个元素的值设为 11  
printf("%d \n", values[0]); // 打印第一个元素的值
```

也就是说，利用数组名加中括号可以访问数组中的每个元素，中括号内的数值表示元素的位置（称为“下标”或“索引”）。元素的下标从 0 开始计算，这意味着，第一个元素使用 `values[0]`，第二个元素使用 `values[1]`，.....，以此类推。

例如，下面的代码用于对 5 个元素求和，

```
// 对所有的元素求和  
int total = values[0] + values[1] + values[2] + values[3] + values[4];
```

例如，下面的代码用于计算第一个数和第五个数的差值

```
// 第一个数和第五个数的差值  
int delta = values[0] - values[4];
```

1.21.3 用 `sizeof` 取得数组的大小

数组的大小由元素的类型和元素的个数共同决定。例如，

```
int arr[100];  
int size = sizeof(arr); // 大小为 100*4
```

1.22 数组的内存视图

在第 3 章中我们知道，每一个变量都与一块内存区域对应，修改变量的值实质上就是修改对应内存的值，而读取变量的值就是读取相应内存的值。

对于数组来说，它也是变量，而且相当于若干个基本变量排列在一起。那么很容易理解，数组在内存视图角度来看，是直接对应了若干个内存单元。

定义一个 `short` 数组

```
short a[4] = { 0x1111, 0x2222, 0x3333, 0x4444 };
```

由于一个 short 占 2 个字节，所以 4 个 short 就占了 8 个字节的内存。在内存中，a[0],a[1],a[2],a[3]可以视为连续排列的 8 个 short 型变量。下面的示意图阐述了这个概念，如图 4-1 所示。

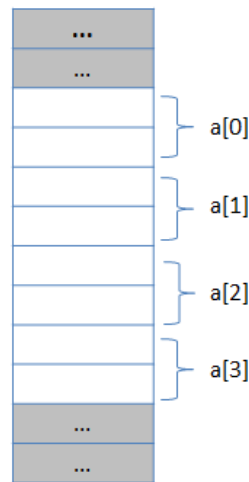


图 4-1 内存视图示意图

注：在 VC 的调试状态下的“内存”窗口中，可以直接观看数组 a 对应的内存。如图 4-2 所示，方框线内的 8 个字节就是数组 a 所占据的内存。其中 11 11 是 a[0]，22 22 是 a[1]，33 33 是 a[2]，44 44 是 a[3]。可以很直观的看到，它们是紧密排列的。

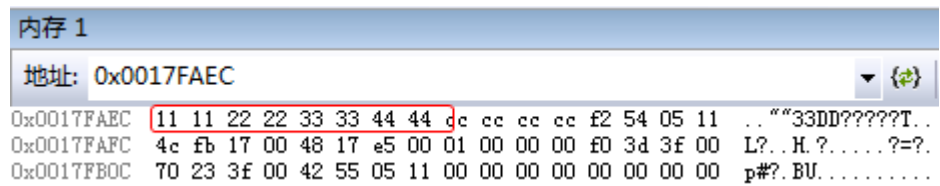


图 4-2 VC 的内存窗口

1.23 常见问题

(1) 初始化列表的长度不能大于数组的长度

在下面的代码中，数组中的长度为 5，大括号却有 6 个初始值。编译器报错。

```
int values [5] = { 1,2,3,4,5,6 }; // 错误: 初始化列表的长度不能大于数组长度
```

(2) 越界访问

对于一个长度为 N 的数组，其下标的合法范围是从 0 到 N-1。一个不合法的下标访问称为“越界”访问。

例如，

```
int values [ 5] = { 1, 2,3 ,4, 5};
```

```
values[5] = 123; // 越界
```

对于上述代码，编译器无法检查错误，但在运行时会出错，通常程序崩溃，或者出现其他意想不到的错误。

对于由编译器能够检查出来的错误，称为“编译错误”，或“语法错误”。

当编译成功、但运行时出错的错误，称为“运行时错误”。

对于初学者来说，遇到的经常是“编译错误”，这是因为大家还不熟悉基本语法。但是当大家熟悉基本语法之后，大家将来经常要面对的都是“运行时错误”。

(3) 只在初始化时才可以用初始值列表

以下代码正确：

```
int array[4] = { 1, 2, 3, 4};
```

以下代码错误：

```
int array[4];
```

```
array = { 1, 2, 3, 4}; // 错误！不存在这个语法
```

只能使用[]对单个元素赋值，正确地写法是下面这样：

```
array[0] = 1;
```

```
array[1] = 2;
```

```
array[2] = 3;
```

```
array[3] = 4;
```

1.24 数组的使用实例

1.24.1 实例 1

定义一个数组，存放 5 个 int 型数据，并将初始值设为 1，2，3，4，5。

```
////////// 例 CH04_C1 //////////
```

```
int b[5];
```

```
b[0] = 1;
```

```
b[1] = 2;
```

```
b[2] = 3;
b[3] = 4;
b[4] = 5; // 正确! 按索引赋值
```

1.24.2 实例 2

定义另一个数组，其中元素的值与上一个例子中的值相反。

////////// 例 CH04_C2 //////////

```
int a[5] = {1,2,3,4,5};
int b[5];
b[0] = a[4];
b[1] = a[3];
b[2] = a[2];
b[3] = a[1];
b[4] = a[0];
```

1.24.3 实例 3

交换第二个元素和第三个元素的值。

////////// 例 CH04_C3 //////////

```
int a[5] = {1,2,3,4,5};
int t = a[1]; // 用 t 记录第二个元素的值
a[1] = a[2]; // 改变第二个元素的值
a[2] = t;    // 改变第三个元素的值
```

1.24.4 实例 4

可以使用 scanf，读取用户的输入保存到数组的元素。

////////// 例 CH04_C4 //////////

```
int a[4];
scanf("%d", &a[0]); // 读取用户输入，保存到 a[0]
```

```
double b[10];

scanf("%lf", &b[1]); // 读取用户输入，保存到 b[1]

printf("Got: %d, %lf\n", a[0], b[1]);
```

1.25 多维数组

二维数组和更高维的数组在实践工程中很少用到，不是本章的重点。

1.25.1 二维问题的表示

有些问题用一维数组难以表示。例如，现有在一个 4x4 的棋盘，每个单元格里放一个数字。使用一维数组表示：

```
int array[16];
```

用(x,y)表示其坐标的化，(3,4)表示第 3 行第 4 行。那么在(3,4)的位置对应的元素是 array[11]，对(3,4)操作也就是对 array[11]进行操作。其坐标与索引的转换公式是：

$$\text{index} = (x-1) * 4 + (y-1)$$

显然，这样表示不直观，阅读起来有些吃力，访问一个元素还先用公式转换先把坐标转换成索引才行。

在 C++中，可以用 2 维数组来表示这个问题。下面的代码中，直接用一个二维数组来定义棋盘，

```
int array[4][4];

array[2][3] = 128; // 对坐标(3,4)位置进行赋值
```

1.25.2 二维数组的定义

二维数组的定义：

```
type name [ N1 ] [ N2 ] ;
```

其中，type 是元素类型, name 是数组变量的名称，N1 是第一维的大小，N2 是第二维的大小。

可以用行、列的概念来理解二维数组。第一个下标是行号，第二下标是列号。

例如，对于一个 4x3 的表格 a[4][3]，其下标依次是

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]

a[3][0]	a[3][1]	a[3][2]
---------	---------	---------

至于 3 维数组、4 维数组以至于 N 维数组，完全可以类似得到。由于高维数组并不常用，这里不再赘述。

1.25.3 二维数组的初始化

二维数组仍然可以大括号初始化，由于有两维，所以可以用二层大括号来分别初始化每行。下面的代码中，对一个 4 行 3 列的数组进行初始化，

```
int a[4][3] =
{
    { 11, 12, 13 },
    { 21, 22, 23 },
    { 31, 32, 33 },
    { 41, 42, 43 }
};
```

1.25.4 二维数组的本质

二维数组，乃至三维、四维等高维数组，只是在形式上比一维数组更直观更容易操作，其本质仍然是一维数组。可以从内存视频清楚地看出这个结论。

比如，对于 `int a[4][3]`，在内存中对应连续的 12 个 `int`：`a[0][0]`, `a[0][1]`, `a[0][2]`, `a[1][0]` ... `a[3][2]`（注意和一维数组的对应顺序）。

注：千万不要误以为二维数组在内存中它是矩阵方式排列的。实际上，它在内存中也是紧凑排列的若干个元素，只是编译器帮了个忙，它帮忙把二维坐标转换成一维坐标了。

第 5 章 字符与字符数组

本章介绍如何表示一个字符，以及如何表示一串字符。

1.26 字符是什么

观察键盘上的按键，要以把字符分为几类：

字母：a b c ... z

数字：0 1 2 ... 9

标点：+ - * / ; , . 等等

控制字符：Tab, Enter 等等

把这几十字符分别用一个数字与之对应，表示这种对应关系的表称为 ASCII 码表。把一个字符对应的那个数字，称为该字符的 ASCII 码。例如，在表 5-1 中列出小写字母('a' ~ 'z')，大写字母('A' ~ 'Z')，数字 ('0' ~ '9') 的 ASCII 码。可以查看本书附录《ASCII 码表》来查看全部规定。

表 5-2 部分字符的 ASCII 码

'0' ⇔ 48
'1' ⇔ 49
...
'9' ⇔ 57
...
'A' ⇔ 65
'B' ⇔ 66
...
'Z' ⇔ 90
...
'a' ⇔ 97
'b' ⇔ 98
...
'z' ⇔ 122
...

注：ASCII 的全称是 American Standard Code for Information Interchange，美国标准信息交换代码，该表由国际标准组织制定。

1.27 字符的表示

在计算机里，所有数据必须以数字的形式表示，字符也不例外。

根据 ASCII 码表的规定，每个字符一个数字表示，而这个数字在 0-127 之间。在 C/C++里，char/short/int 都可以表示整数，由于字符的数值范围较小，我们选用 char 型变量来代表字符。

例如，

```
char ch = 65; // 则 ch 代表的是大写字母'A'
```

```
char ch2 = 49; // 则 ch 代表的是数字'1'
```

显然，如果用 short 或 int 来表示字符的话就显得大材小用，浪费空间。

1.28 字符的显示

可以使用 printf 将一个字符显示到控制台，使用的格式化字符串为 %c。

例如，下面的代码用于将 65 所代表的字符显示出来（即大写字母 A）：

```
printf("Got: %c \n", 65 );
```

运行的结果如图 5-1 所示。

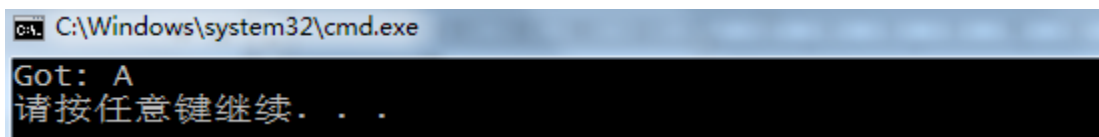


图 5-1 控制台窗口的输出显示

例如，下面的代码将显示 49 所代表的字符（即数字 1）：

```
char ch2 = 49;
```

```
printf("Show: %c \n", ch2 );
```

例如，我们打印显示 "1+2" 这个字符串：

```
printf("%c%c%c \n", '1', '+', '2' );
```

例如，我们打印几个空格字符：

```
printf("%c%c%c \n", 'A', ' ', ' ');
```

其中，空格字符 ' ' 就是在单引号中间加上一个空格。

1.29 字符常量

在 C/C++ 代码中，直接用字符常量来代表一个字符的 ASCII 码。使用单引号表示。例如，

```
char ch = 'A';
```

则 'A' 就是字符常量，它是一种字面常量，表示一个整数 65。

字符常量在任何时候都和它的 ASCII 码是等价的。也就是说，虽然在形式上它是写成了 'A'，但编译器在处理代码的时候总是把它当成 65 来处理的。

所以，以下几种写法是等价的：

```
printf("Got: %c \n", 65 );  
printf("Got: %c \n", 'A');  
printf("Got: %c \n", 0x41);
```

由于字符常量完全等价于一个整数，所以我们可以这么写：

```
char ch1 = 'A' + 1; // 结果为 66  
char ch2 = 'B' - 1; // 结果为 65  
char ch3 = 'C' - 'A'; // 结果为 2
```

显然我们也可以用 int 和 short 来表示字符：

```
int ch1 = 'A';  
short ch2 = '9';
```

下面的代码用于显示字符 'Y' 的 ASCII 码：

```
printf (" %d \n", 'Y'); // 'Y' 是一个整数，所以可以用 %d 显示出来
```

加到这里

1.30 字符数组

在 C/C++ 里，用一个 char 型数组来表示一串字符，称为“字符数组”。把这一串字符称为“字符串”。

字符两种初始化方法：

（1）像普通数组一样初始化

```
char str[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

（2）特殊的初始化方法

```
char str[6] = "hello";
```

使用第（2）种方式在代码上要简洁一些。这两种方法的最终结果一样，就是在占据了 6 个字节的内存，其值依次值：

68	65	6C	6C	6F	00
----	----	----	----	----	----

在本教程中，有时也会表示成下面的样子，它们是同一数据不同表示方法，本质上一样。

h	e	l	l	o	'\0'
---	---	---	---	---	------

需要注意的时，当用字符数组来存储字符串时，必须以'\0'结尾。我们把'\0'称为字符串的结束符，它的 ASCII 码数值为 0。注意，'\0'和 0 两种写法是完全等效的，都表示整型常量 0，所以本书中时而将它们混用。

1.30.1 输出字符串

可以用 `printf` 来向控制台输出一个以 0 结尾的字符串，使用的格式符为 `%s`。例如，

```
char str[6] = "hello";
printf("string: %s \n", str);
```

1.30.2 输入字符串

我们可以使用 `gets` 来获取一个字符串。在示例 CH05_C1 中，使用 `gets` 将用户输入的字符串存入到数组 `buf` 中。注意，数组的长度要足够大，以免用户输入太长的字符串。

////////// 例 CH05_C1 //////////

```
#include <stdio.h>

int main()
{
    /* 用户在控制台输入一个字符，
       按回车结束，
       所有输入的字符被存储到 buf 数组中
    */

    char buf[128]; // 定义一个足够大的数组
    gets(buf);

    // 把刚才输入的内容再打印输出
    printf("Got: %s \n", buf);

    return 0;
}
```

注：对于 `gets` 的使用，我们现在只需要学会其简单的用法即可，暂是不必知道其语法原理。
在使用上：①用于接入收入的数组 `buf` 要足够大 ②`gets(buf)` 的小括号里是数组名字。

1.30.3 理解结束符的作用

字符串的末尾必须为一个数字 0 作为结束符，它是一个字符串结束的标识。如果一串字母不以 0 结束，那它就不算是一个有效的字符串。

例如，下面定义了一个字符数组，前 3 个字符是 `b,a,d`，但末尾并不是 0，那它不是一个有效的字符串，在用 `printf` 输出的时候会有问题：

```
char buf[4] = { 'b', 'a', 'd', -1 };  
printf("Got: %s\n", buf);
```

控制台输出时，会打印出一些乱码字符，比如“烫烫烫”这样的乱码，这表明我们要打印的字符串没有以 0 结尾，是一个无效的字符串。运行结果如图 5-2 所示。



图 5-2 控制台窗口的输出显示

`\0` 字符决定了字符串的长度，`\0` 后面的字符不会被 `printf` 打印出来。正是从这个意义上，才把 `\0` 称为结束符的。例如，下面定义一个 9 字节的 `char` 数组，

```
char str[9] = { 'G', 'o', 'o', 'd', 0, 'B', 'y', 'e', 0 };  
printf("str: %s\n", str);
```

其输出结果为 `Good`，而后面的几个字符 `Bye` 是不会被显示输出的，属于无效内容。

我们把结束符 `\0` 前面的有效字符的个数，称为字符串的长度。也就是说，从第一个字符开始往后数，一直到达结束符 0，中间的非 0 字符的个数，就是该字符串的长度。

例如，计算下面几个字符串的长度。

```
char str1[9] = { 'G', 'o', 'o', 'd', 0, 'B', 'y', 'e', 0 }; // 长度为 4  
char str2[9] = { 0, 'o', 'o', 'd', 0, 'B', 'y', 'e', 0 }; // 长度为 0
```

1.30.4 字符串的截断

利用字符串的结束符，可以轻易的截断一个字符串，例如，

```
char buf[32] = "hello,world";  
buf[5] = 0;
```

```
printf("Result: %s \n", buf);
```

由于 buf[5]被设置成了结束符，所以字符串 buf 的有效部分就是"hello"了。

1.30.5 常见问题

(1) 字符数组不够大

例如，下面的字符数组不够存储 5 个字符+1 个结束的，至少要 6 个字节才够存储。

```
char buf[5] = "hello"; // 数组不够大，至少为字符串长度+1 才够
```

1.31 转义字符

1.31.1 转义字符的概念

如果我们想换显示，怎么做到呢？可以使用一些特殊的字符，如'\n'，表示换行。在下面的代码中，输出一个字符'1'，换行，再输出一个字符'2'，再换行。

```
printf("%c%c%c%c", '1', '\n', '2', '\n');
```

需要注意的是，以反斜杠开头的'\n'表示单个字符，其 ASCII 码为 10（0x0A）。

在 C/C++语言里，存在着用反斜杠开头的一些特殊的字符常量，它们承担特殊的控制功能，称为“转义字符”。

常见的转义字符在表 5-2 中列出。

表 5-2 常见的转义字符及说明

转义字符	值	意义
\a	0x07	(alert)响铃警告
\n	0x0A	(nextline)换行
\t	0x09	(tab) 制表位
\b	0x08	(backspace)回退
\r	0x0D	(return) 回车
\\	0x5C	反斜杠\
\"	0x22	双引号"
\0	0x00	字符串结束符

1.31.2 转义字符的使用举例

(1) 换行符

```
printf("Hello \nWorld \n");
```

输出结果如图 5-3 所示。

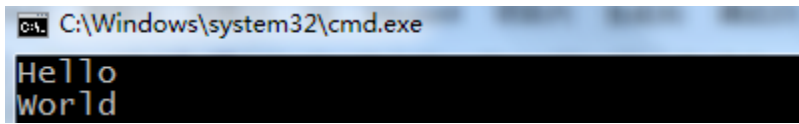


图 5-3 控制台窗口的输出显示

(2) 反斜杠

```
printf("C:\\Windows\\System32\\ \n");
```

这在表示 windows 的文件路径时经常用到，输出结果如图 5-4 所示。

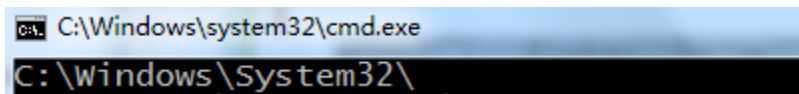


图 5-4 控制台窗口的输出显示

(3) 双引号

```
printf("He said \"good bye! \" and left.\n");
```

输出结果如图 5-5 所示。

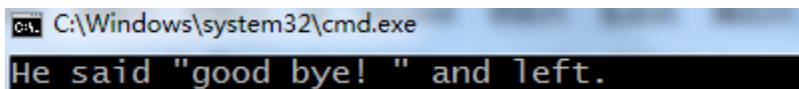


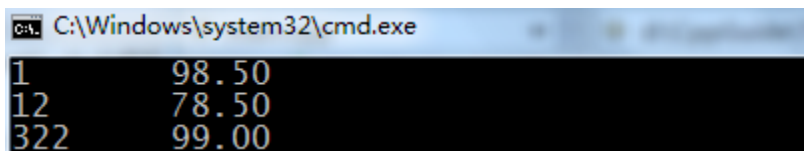
图 5-5 控制台窗口的输出显示

(4) 制表符

下面的代码，使用\t制表符，使得输出的数据纵向按制表位对齐，

```
printf("%d\t%.2f\n", 1, 98.50f);  
printf("%d\t%.2f\n", 12, 78.50f);  
printf("%d\t%.2f\n", 322, 99.00f);
```

输出结果如图 5-6 所示。



```
C:\Windows\system32\cmd.exe
1      98.50
12     78.50
322    99.00
```

图 5-6 控制台窗口的输出显示

第 6 章 表达式与操作符

表达式(expression)是一个式子，它有一个值。它是操作数(operand)与操作符(operator)的组合。本章介绍如何使用各种类型的操作符来构造表达式。

1.32 算术表达式

由算术运算符连接起来的表达式，称为算术表达式。算术操作符有 5 个，如表 6-1 所示。

表 6-3 算术操作符

操作符	说明
+	相加
-	相减
*	相乘
/	相除
%	取模，只适用于整数

其中，相加、相减、相乘都和普通的算术概念一致，对于整型和浮点型都适用。

例如：

```
3 + 4 ; // 其值为 7
3 - 4 ; // 其值为-1
3 * 4 ; // 其值为 12
0.2 + 0.4 ; // 其值为 0.6
0.2 - 0.4; // 其值为-0.2
0.2 * 0.4; // 其值为 0.08
```

对于整数来说，相除是取倍数，取模是取余数。

例如，

```
22 / 5 ; // 其值为 4 （22 是 5 的 4.4 倍，结果只取整数部分）
22 % 5 ; // 其值为 2 （22 除以 5，余数为 2）
```

对于浮点数来说，相除后仍然是浮点数（保留小数部分）。浮点数不能进行取模运算。

`22.0 / 5.0; // 其值为 4.4`

`123.5 % 10 ; // 编译器报错！浮点型不能进行模运算！`

组合在一起的时候，其优先顺序是 `*/%+-` 的顺序。当然，如果记不清的话，直接用括号 `()` 来显式地指定结合顺序。

`a + b * 10 / c % d - e`

1.33 赋值表达式

用等号可以对变量进行赋值。等号在这里被称为赋值操作符。例如：

`a = 10; // 其值是 10`

`b = 11.23; // 其值是 11.23`

需要特别注意的是，在赋值表达式中，整个表达式本身的值就是变量最终的值。用以下代码检查一下：

`int a;`

`printf("test: %d \n", a=10); // 打印输出为 10,`

1.33.1 左值

在 C/C++ 中，可以放在等号左值的值，称为左值。一般来说，一个左值最终肯定对应一块内存，修改它的值其实就是修改了那一块内存的值。

(1) 变量可以被赋值，它是左值。由于变量对应一块内存，修改变量的值，就是修改了内存的值。

(2) 数组元素可以被赋值，它也是左值。数组元素与对应着内存，例如 `arr[0] = 123` 表示修改第一个元素对应的内存的值。

(3) 字面常量不能被赋值，它不是左值。例如，下面的代码是错误的。

`1000 = a; // 编译器报错`

1.33.2 理解赋值运算

赋值运算在具体操作时有两步：

- ① 计算出等号右侧的表达式值
- ② 将值写入到左侧变量对应的内存中

下面，举一个例子，来解释这一过程。

`int a = 1;`

`int b = 2;`

```
a = a + b; // 此行为赋值运算
```

我们看一下这一步赋值运行的具体过程。首先，我们知道 CPU 中存在一个加法器单元，其功能是实现加法运算。只要将两个数传给它，它就能执行运算，求出两个数的和，并把结果放在输出单元里。如图 6-1 所示。

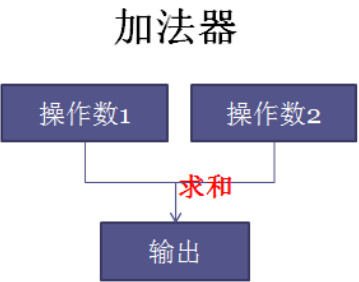


图 6-1 加法器示意图

在 CPU 执行 `a = a + b` 时，

- ① 求右侧表达式 `a + b` 的值

从 `a` 取值送到“操作数 1”单元，从 `b` 取值送到“操作数 2”单元。得到结果，存在“输出”单元。

- ② 从加法器的“输出”单元取值，存放 `a` 对应的内存。

1.33.3 赋值与算术运算合并

算术运算可以和赋值运行简写，于是增加了一些简写的操作符，如表 6-2 所示。

表 6-2 赋值操作符

操作符	说明
<code>+=</code>	<code>a += b</code> 相当于 <code>a = a + b</code>
<code>-=</code>	<code>a -= b</code> 相当于 <code>a = a - b;</code>
<code>*=</code>	<code>a *= b</code> 相当于 <code>a = a * b;</code>
<code>/=</code>	<code>a /= b</code> 相当于 <code>a = a / b;</code>
<code>%=</code>	<code>a %= b</code> 相当于 <code>a = a % b;</code>

1.33.4 等号串连的写法

多个赋值可以串连在一行内。例如，

```
a = b = c = 10;
```

相当于

```
(a = (b = (c = 10))) ;
```

1.34 关系表达式

关系表达式是用于表示两个数的大小关系的式子。关系表达式有关系操作符连接起来。

像

```
10 > 9
```

```
a < b
```

等就是关系表达式，而大于号>和小于号<就是关系操作符。关系操作符如表 6-3 所示。

表 6-3 关系操作符

操作符	说明
<	小于
<=	小于或等于
>	大于
>=	大于或等于
==	等于
!=	不等于

关系表达式的值有两种: 1（真）或 0（假），也就是说要么为真、要么为假。在 C++中以零表示假，以非零表示为真。例如：

```
10 > 9; // 真，其值为 1
```

```
10 > 11; // 假，其值为 0
```

可以用 printf()函数来检查一下这个结论：

```
printf("test: %d \n", 10 > 11); // 打印显示为 0
```

```
printf("test: %d \n", 11 > 10); // 打印显示为 1
```

由于表达式是右值，因此可以赋值给变量。

```
int value = 9 > 10;
```

```
printf(" values is : %d \n", value); // 打印结果： 0
```

1.35 条件表达式

条件表达式，也称为问号表达式，它是由条件操作符组成的、带判断条件的表达式。其一般形式为，

```
expr1 ? expr2 : expr3
```

意义：

当 `expr1` 为“真”时，表达式的值为 `expr2`

当 `expr1` 为“假”时，表达式的值为 `expr3`

例如，

```
0 ? 11 : 22; // 条件为假（0），整个表达式的值为 22
```

```
1 ? 11 : 22; // 条件为真（非 0），整个表达式的值为 11
```

```
1 ? 11 : 22; // 条件为真（非 0），整个表达式的值为 11
```

```
(a = 10) ? 11 : 22; // 条件值为 10（非 0），整个表达式的值为 11
```

```
(3 + 4) ? 11 : 22; // 条件值为 7，整个表达式的值为 11
```

可见，在阅读条件表达式的时候，第一步就是计算一下问号前面的条件的值。如果条件值为真（非 0），则取冒号前面的值；如果条件值为假（0），则取冒号后面的值。

下面列出几个例子：

```
printf ("%d \n", 0 ? 100 : 200 ); // 假，显示 200
```

```
printf ("%d \n", 1 ? 100 : 200 ); // 真，显示 100
```

```
printf ("%d \n", -2 ? 100 : 200 ); //真，显示 100
```

```
printf ("%d \n", (5 + 1) ? 100 : 200 ); //真，显示 100
```

又如，用 `int` 型变量 `score` 来表示一个学生的成绩，当大于等于 60 时，显示 T；否则显示 F。代码实现如下：

```
int score = 90;
```

```
printf ("%c \n", score > 60 ? 'T': 'F');
```

再例如，当成绩在 90 以上为 A，60 以下时为 C，中间的为 B。代码实现如下：

```
int s = 80;

char result = (s > 90 ? 'A': ( s<60 ? 'C':'B' ));

printf ("%c \n", result);
```

1.36 逻辑表达式

逻辑操作符包括逻辑非，逻辑与，逻辑或，如表 6-4 所示。

表 6-4 逻辑操作符

操作符	说明
!	逻辑非
&&	逻辑与
	逻辑或

1.36.1 逻辑非

其语法形式为：

```
! expr
```

即，在表达式之前加一个感叹号，表示对该表达式取非。具体操作是，如果 `expr` 的值是"真"，则取非后变成"假"(0)；如果 `expr` 为"假"，则取非后为“真” (1)。

```
printf("%d \n", ! 23); // 打印结果： 0

printf("%d \n", ! 0);  // 打印结果： 1

printf("%d\n", !( 10 > 9)); // 打印结果： 0
```

1.36.2 逻辑与

逻辑与，表示“并且”的意思。其形式为：

```
expr1 && expr2
```

判断规则：当 `expr1` 为真、并且 `expr2` 为真时，结果为真；否则为假。

例如：

```
2 && -1; // 真真=> 真

1 && 0 ; // 真假=> 假
```

`0 && 4 ; // 假真=> 假`

`0 && 0 ; // 假 0=> 假`

`3>4 && a; // 假* => 假 (不论 a 是真是假, 结果总是假)`

提前结算: 当 `expr1` 为假时, 结果已经明了, 此时 `expr2` 不会被计算。

例如,

```
int a = 4;
```

```
int b = (3 > 4) && (a=123); // 提前结算, 右式 a=123 不会被执行, 所以 a 值不变
```

1.36.3 逻辑或

逻辑或, 表示“或者”的意思。其形式为:

`expr1 || expr2`

判断规则: 当 `expr1` 为真、或者 `expr2` 为真时, 结果为真; 否则为假。

例如:

`3 || 2 ; // 真真=>真`

`1 || 0 ; // 真假=> 真`

`3<4 || 2 ; // 假真=>真`

`0 || 0 ; // 假假=> 假`

`4>3 || a; // 真* => 真 (不论 a 是真是假, 结果总是真)`

提前结算: 当 `expr1` 为真时, 结果已经明了, 此时 `expr2` 不会被计算。

例如,

```
int a = 4;
```

```
int b = (5==5) || (a=123); // 提前结算, 右式 a=123 不会被执行, 所以 a 值不变
```

1.37 逗号表达式

逗号表达式是从逗号分开的一组表达式，计算的时候从左到右依次计算，最后一个表达式的值为整个表达式的值。

`ex1, ex2, ex3, ..., exN`

其中, `exN` 是一个表达式。

逗号表达式，是以逗开隔开的一系列表达式。其形式为：

`expr1, expr2, expr3, ..., exprN`

其运算规则是，从左到右依次计算每个子式的值，并把最后一个子式的值作为整个表达式的值。

例如，

```
int a = 4;
printf("%d \n", (a+12, !3, a=1)); // 输出为 1，因为最后一个子式 a=1 的值为 1
```

又如，

```
int nnn = (1, 2, 3, 4);
printf("nnn = %d \n", nnn); //打印结果： 4
```

又如，

```
int a = 2;
int b = 3;
int nnn = (a+b, a-b, a*b);
printf("nnn = %d \n", nnn); //打印结果： 6
```

1.38 自增/自减操作符

`++`是自增操作符，表示对变量加 1；`--`是自减操作符，表示对变量的值减 1。只能作用于整型变量，不能作用于浮点型变量。

例如：

```
int a = 10;
a ++; //相当于 a=a+1
a --; //相当于 a=a-1
```



```
++a; //相当于 a=a+1
```

```
--a; //相当于 a=a-1
```

1.38.1 前置

前置时，先执行自增/自减操作，再执行所在行的表达式。

```
int a = 10;  
printf("%d", ++a);
```

相当于：

```
int a = 10;  
a += 1;  
printf("%d", a);
```

1.38.2 后置

后置时，先计算本行的表达式，之后再执行自增操作。

```
int a = 10;  
printf("%d", a++);
```

相当于：

```
int a = 10;  
printf("%d", a);  
a += 1;
```

注：自增和自减操作符只要简单掌握即可，它们不是 C/C++语法的重点，甚至完全不用它们没有关系。在 C/C++中，要尽量把代码写得简单易读，不要在一行内大量使用自增/自减操作符。

1.39 *位操作符

(*初学者可以跳过本节，这不会影响到后续的学习)

表 6-5 位操作符

操作符	说明
~	按位取反
<<	左移
>>	右移

<<=	左移并赋值
>>=	右移并赋值
&	按位与
^	按位异或
	按位或
&=	按位与赋值
^=	按位异或赋值
=	按位或赋值

所有的位操作只适用于整数，即 `char`, `short`, `int`, `unsigned char`, `unsigned short`, `unsigned int`。更具一点，只有无符号整数才适合使用位操作。

1.39.1 按位表示

首先，一个字节由 8 个位组成，这里先以最短的整型 `unsigned char` 来说明位的含义和用法。例如，下面定义了 2 两个变量 `M` 和 `N`，

```
unsigned char M = 0xA7, N = 0xE3;
```

其中，`M` 的按位表示为 (最左侧为高位 `bit7`，最右侧为低为 `bit0`)

1	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---

`N` 的按位表示为(最左侧为高位 `bit7`，最右侧为低为 `bit0`)

1	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

可以类推一下，`unsigned short` 就是 16 个位，`unsigned int` 是 32 位。

1.39.2 位运算规则

需要理解与、或、取反、异或这 4 种位操作。两个位 `A` 与 `B` 进行位运算是，它们的运行规则是：

“与”运算： `A=1` 且 `B=1` 时，所得结果是 1；否则为 0

1	&	1	=	1
1	&	0	=	0
0	&	1	=	0
0	&	0	=	0

“或”运算：A=1 或 B=1 时，所得结果是 1；否则为 0

1		1	=	1
1		0	=	1
0		1	=	1
0		0	=	0

“异或”运算：A 与 B 不同时，所得结果是 1；否则为 0

1	^	1	=	0
1	^	0	=	1
0	^	1	=	1
0	^	0	=	0

“取反”运算：1 取反得 0；0 取反得 1

$\sim 1 = 0$;
 $\sim 0 = 1$;

现在利用上面的规则，对 M 与 N 进行位运算，

与运算：M & N = ?

1	0	1	0	0	1	1	1	= 0xA7
1	1	1	0	0	0	1	1	= 0xE3
<hr/>								
1	0	1	0	0	0	1	1	= 0xA3

或运算：M | N = ?

1	0	1	0	0	1	1	1	= 0xA7
1	1	1	0	0	0	1	1	= 0xE3
<hr/>								
1	1	1	0	0	1	1	1	= 0xE7

异或运算：M ^ N = ?

1	0	1	0	0	1	1	1	= 0xA7
1	1	1	0	0	0	1	1	= 0xE3
<hr/>								
0	1	0	0	0	1	0	0	= 0x44

取反运算： $\sim M$ = ?

1	0	1	0	0	1	1	1	= 0xA7
<hr/>								
0	1	0	1	1	0	0	0	= 0x58

可以自己编写代码，测试一下运算的结果，与上面计算的结果比较。

```
#include <stdio.h>

int main()
{
    unsigned char M = 0xA7, N = 0xE3;
    unsigned char result;
    result = M & N;
    printf("%02X \n", result);
    result = M | N;
    printf("%02X \n", result);
    result = M ^ N;
    printf("%02X \n", result);
    result = ~M;
    printf("%02X \n", result);
    return 0;
}
```

在掌握了&|^~四种位操作符之后，&=，|=，^=就比较容易理解和掌握了。

$M \&= N$; 相当于 $M = M \& N$

$M |= N$; 相当于 $M = M | N$

$M \wedge= N$; 相当于 $M = M \wedge N$

1.39.3 移位操作

下面再看一下移位运算。再看一下 $M(0xA7)$ 的表示：

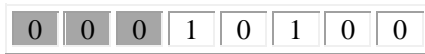
1	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---

(1) 右移：

例如， $M \gg 1$ 表示 M 的所有位右移一位，左侧填充 0，移位后的结果表示如下：

0	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---

例如， $M \gg 3$ 表示 M 的所有位右移三位，左侧填充 0，移位后的结果表示如下



例如， $M \gg 8$ 表示 M 的所有位右移八位，左侧填充 0（原有 8 个位全部移走，剩下的填充为 0），移位后的结果表示如下

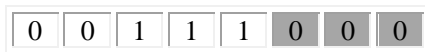


(2) 左移:

例如， $M \ll 1$ 表示 M 的所有位左移一位，右侧填充 0，移位后的结果表示如下



例如， $M \ll 3$ 表示 M 的所有位左移三位，右侧填充 0，移位后的结果表示如下



例如， $M \ll 8$ 表示 M 的所有位左移八位，右侧填充 0（原有 8 个位全部移走，剩下的全填充为 0），移位后的结果表示如下



用代码检查一下上述手工运行的结果是否正确，

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    unsigned char M = 0xA7;
```

```
    unsigned char result;
```

```
    result = M >> 1;
```

```
    printf("%02X \n", result);
```

```
    result = M >> 3;
```

```
    printf("%02X \n", result);
```

```
    result = M >> 8;
```

```
    printf("%02X \n", result);
```

```
    result = M << 1;
```

```

printf("%02X \n", result);

result = M << 3;

printf("%02X \n", result);

result = M << 8;

printf("%02X \n", result);

return 0;

}

```

对于无符号整型(unsigned char / unsigned short / unsigned int)来说，移位操作是很直观和容易理解的。实际工程应用中，只对无符号整型进行移位操作。

对有符号整型(char / short / int)来说，其移位操作的规则比较复杂、且难以理解。幸运的是，在工程一般不对有符号整数进行移位操作，因为对有符号数移位不直观，也没有什么实际意义。负整数在右移时，高位（符号位）不变，左侧填充 1。而其他情况下，均与无符号数的移位规则相同。

1.39.4 应用举例

定义变量 unsigned char flag = 0;

(1) 置 1 操作

将 flag 的 bit5 设置为 1，其余位保持不变

```
flag |= (1u << 5); // 注：1u<<5 等于 0010 0000
```

(2) 清 0 操作，其余位保持不变

将 flag 的 bit5 清空为 0:

```
flag &= ~(1u << 5); // 注：~(1u<<5)等于 1101 1111
```

(3) 判断 bit5 是否为 1

下面的代码用于求 value 的 bit5 的值。可以发现，若变量 value 的 bit5 为 1，则 is_set 值为 1；否则，is_set 的值为 0。所以，is_set 就是 bit5 的值。

```

unsigned char value = 0xC6; // 二进制: 1010 0110

unsigned char mask = 1u << 5; // 二进制: 0010 0000

unsigned char is_set = (value & mask) ? 1: 0; // is_set 的值就是 value 的 bit5 的值

```

1.39.5 例题：把 unsigned int 整数转为 4 个字节

对一个 unsigned int 型变量来说，它占据了 4 个字节。但是如何取得 4 个字节的值呢？可以用移位实现。例如以下的代码中，将变量 a 右移 24 位，则 a 的最高 8 位(bit24-bit31)被移到了低 8 位上，可以直接转成 unsigned char 来保存。依此类推，可以得到 4 个字节的值。

```
unsigned int a = 0x12345678;

unsigned char buf[4];

buf[0] = a >> 24;

buf[1] = a >> 16;

buf[2] = a >> 8;

buf[3] = a;
```

此时，我们得到的结果是: buf[0]: 0x12, buf[1]: 0x34, buf[2]: 0x56, buf[3]: 0x78

1.40 类型的转换与提升

在 char / short / int / double / float 之间，是可以互相转换的。

例如，

```
int a = 12.34;
```

此语句中左侧为 int 型变量，右侧为 double 型的值，结果是 a 的值为 12，小数部分被截断。所以编辑器给出一个警告，提示“从 double 到 int 可能丢失数据”。如图 6-2 所示，



图 6-2 VC 的输出窗口

如果反过来，把一个 int 型数值赋值给 double 型变量，则不会损失数据。例如下面的代码，

```
double a = 12;
```

此语句中左侧为 double 型，右侧为 int 型，由于 double 的表示范围大于 int，所以不会丢失数据。

以上给事实给了我们一个基本的概念：每一种类型都有一种表示范围，从高到低转换会发生数据损失或截断，从低到高转换不会发生数据损失。它们由低到高依次是 char < short < int < double, float < double。由于 int 和 float 都是 4 个字节，它们之间的转换没那么简单。

在算术表达式中，当不同的类型混合运算时，编辑器会根据级别最高的类型先进行提升，把所有的操作数都操升到最高级别，然后再进行运算。这样做的目的是避免数据丢失。

例如，在下面的表达式中，最高类型为 `double`，所以编译器自动将各操作数都提升为 `double` 型再进行计算：

```
12 + 78.5 ;
```

以上的转换由编译器自动完成，称为“隐式转换”。除此之外，可以使用“显式转换”强制的将类型向上或向下转换。例如，

```
(double) 100 ; // 向上提升为 double，整个表达式的值类型变成 double
```

```
(double) 100/8; // 将 100 向上提升为 double，致使所有的操作数都向上提升，最终表达式的值为 12.50;
```

```
100/8; // 对比一下，这个表达式是两个 int 作整除，值为 12
```

```
13 + (int) 25.3; // (int)将 25.3 强制转换成 25
```

1.41 优先级与结合顺序

当一个表达式中出现多个运算符时，需要考虑优先级的问题。以最简单的算术运算符为例，*/的优先级是高于+-的优先级的。观察下面的代码，

```
int a = 10 + 11 * 12 / 13 - 14 * 15;
```

根据优先级顺序，高优先级的先结合，相当于，

```
int a = (10 - (((11 * 12) / 13) + (14 * 15)));
```

还好，这些 C++ 的乘除加减和数学里的相似，所以我们凭借有限的数学知识，可以迅速判断出哪些操作数被先结合。但是情况再复杂一点，就很难迅速判断了，因为操作符的种类繁多，很难记住的优先级高低。

例如，

```
a && !b || c + d > 100 && c < 10 （逻辑表达式混用）
```

```
a + flag & 0x0E + b < 128 ? 1 : 0 (位表达式、条件表达式)
```


1.41.1 使用括号

幸运的是，想要完全记住所有操作符的优先级是不太可能的，这样做也没有必要。原因是，这么复杂的表达式，即使你看的懂，也不能保证别人也看得懂。当一个表达式越复杂时，其可读性就越差。代码的可读性是一件非常重要的事情。

那么怎样才能既不用太多记忆，又保证正确性和可读性呢？答案很简单：**积极使用括号**。

无论何种场景，显式地使用小括号来指定结合顺序，会使你的代码的可读性大增，并且能保证完全正确。当你合理的使用小括号后，别人可以很容易地读懂你的代码，不会有误解和歧义。

比如，我们对下面这种比较复杂的式子使用小括号，

```
a && ( !b ) || ((c + d > 100) && (c < 10))  
a + (flag & 0x0E) + (b < 128 ? 1 : 0) (位表达式、条件表达式)
```

1.41.2 常用的优先级

实际上，在前面每一节的操作符的表格里，操作符已经是按优先级排列的了。读者应该记住它们的相对优先级就可以了。例如，在算术操作符中，要记住 `*` / 比 `+` - 高就够用了，在逻辑操作符里，要记住 `!` 高于 `&&` 高于 `||` 就可以了。

这样就可以少写一些小括号，毕竟如果小括号太多了，那可读性反而会下降。

例如，由于大多程序员都明白 `! && ||` 这个顺序，所以它们联立时可以不加小括号，

```
a && !b || (c + d > 100) && (c < 10)
```

总结为一条原则：**常见的优先级不加小括号，没有把握的优先级加小括号**。

第 7 章 语句

本章介绍语句的概念，以及几种常用的几种控制语句 if / switch / for / while。

1.42 什么叫语句

简单地说，以分号结束的一行，称为一条语句（Statement）。

例如，下面的代码包括了 4 条语句：

```
int b = 1;

int a = b;

a = 10;

printf("hello world!\n");
```

然而多个语句也可以写在同一行里：

```
int b = 1; int a = b; a = 10;
```

这样写在语法上没有任何问题，只是可读性较差。这也说明，分号才是划分语句的标志，即使同一行里也可以存在多个语句。

1.42.1 空语句

一个语句的内容可以空，而只有一个分号。空语句的语法地位和普通的一条语句是相同的。

例如，下面的这些只有单个分号的语句称为空语句，

```
; // 空语句

; // 空语句

int a = 10; // 正常语句
```

1.42.2 复合语句

可以将大括号将多个语句又大括号组合起来，称为复合语句。例如，

```
////////// 例 CH07_A1 //////////

#include <stdio.h>

int main()
{
    int a = 10;

    //下面的复合语句里包含了 3 条单语句

    {
        a += 2;
```

```

        a *= 3;

        printf("a=%d \n", a);
    }
    return 0;
}

```

复合语句在语法上和一条单语句的地位是相同的，所以在一条单语句出现的地方，都可以替换为复合语句。

(1) 复合语句内可以含有 0 条语句

也就是说，大括号内可以为空。

例如，

```

int main()
{
    //下面的复合语句里包含了 0 条单语句

    {
    }

    return 0;
}

```

(2) 复合语句内可以嵌套复合语句

复合语句对上一个层次来说，相当于一条单语句。所以，复合语句内还有嵌套复合语句。例如，

```

int main()
{
    int a = 10;
    {
        // 嵌套一个复合语句
        {
            a += 2;
            a *= 3;
        }
        // 嵌套一个复合语句
    }
}

```

```

        {
            a -= 1;
            a /= 2;
        }
    }
    return 0;
}

```

1.43 if 语句

if 语句用于判断一个条件是否成立：当条件成立时，做一件事情；当条件不成立时，做另一件事情。其基本语法形式为：

```

if (expr)
    statement1
else
    statement2

```

语法规则：

当 `expr` 为真（非 0）时，执行语句 `statement1`；否则，执行语句 `statement2`。

其中，`statement1` 和 `statement2` 允许是一条单语句、空语句、或复合语句。

例如，用 `x` 表示一次考试的成绩，当 `x` 大于或等于 60 分时，提示 `Pass`，否则提示 `Fail`。可以用 if 语句来实现，示例代码如下：（为了方便说明，在每行前面加上行号标注，行号部分不属于代码）

```

////////// 例 CH07_B1 //////////
01  #include <stdio.h>
02  int main()
03  {
04      int x = 61; // x 可以通过 scanf 让用户输入
05      if(x >= 60)
06          printf("Pass !\n");
07      else
08          printf("Fail !\n");

```

```
09         return 0;
```

```
10     }
```

过程分析：

第 1 种情况：令 $x=61$

=> 执行[04] : `int x = 61`

=> 执行[05] : 表达式 $x \geq 60$ 的值为 1，执行[08]

=> 执行[08] : `printf("Pass !\n")`

=> 执行[09] : `return 0`

第 2 种情况：令 $x=59$

=> 执行[04] : `int x = 61`

=> 执行[05] : 表达式 $x \geq 60$ 的值为 0，执行[06]

=> 执行[06] : `printf("Fail !\n")`

=> 执行[09] : `return 0`

1.43.1 使用复合语句

根据 if 语句的语法，在 if 和 else 后面只可以接一条单语句，或者一个复合语句。所以，当需要由多条语句来完成一个处理时，可以将多个条语句组合成复合语句。例如，

//////////////////// 例 CH07_B2 //////////////////////

```
01  #include <stdio.h>
```

```
02  int main()
```

```
03  {
```

```
04      int x = 61; // x 可能通过 scanf 让用户输入
```

```
05      if(x >= 60)
```

```
06      {
```

```
07          printf("Pass !\n");
```

```
08          printf("Go for a celebration! \n"); // 庆祝一下
```

```
09      }
```

```
10      else
```

```
11      {
```

```

12             printf("Fail !\n");
13             printf("Pay more efforts !\n"); // 更须努力
14         }
15     return 0;
16 }

```

记住，绝对不能写成下面这样，这样是有语法错误的。

```

if(x >= 60)
    printf("Pass !\n");
    printf("Go for a celebration! \n"); // 庆祝一下
else
    printf("Fail !\n");
    printf("Pay more efforts !\n"); // 更须努力

```

注：从可读性的角度来说，本书建议读者在 `if` 和 `else` 后面统一加复合语句，即使只有一条语句，也加上大括号。

1.43.2 最简形式 if

我们可以把 `else` 的处理部分省略，只保留 `if` 部分。其语法形式为：

```

if (expr)
    statement1

```

语法规则：当 `expr` 为真（非 0）时，执行语句 `statement1`。

在示例 CH07_A3 中，只对 `x>=60` 的情况做了处理；如果不满足条件，则什么都不做。

////////////////// CH07_B3 //////////////////////

```

01  #include <stdio.h>
02  int main()
03  {
04      int x = 61; // x 可能通过 scanf 让用户输入
05      if(x >= 60)
06      {
07          printf("Pass !\n");

```

```

08             printf("Go for a celebration! \n"); // 庆祝一下
09         }
10         printf("That's over \n");
11         return 0;
12     }

```

过程分析：

第 1 种情况：令 $x=61$

=> 执行[04] : `int x = 61`

=> 执行[05] : 表达式 $x \geq 60$ 的值为 1，执行[07]

=> 执行[07] : `printf("Pass !\n")`

=> 执行[08] : `printf("Go for a celebration! \n")`

=> 执行[10] : `printf("That's over \n")`

=> 执行[11] : `return 0`

第 2 种情况：令 $x=59$

=> 执行[04] : `int x = 61`

=> 执行[05] : 表达式 $x \geq 60$ 的值为 0，执行[10]

=> 执行[10] : `printf("That's over \n")`

=> 执行[11] : `return 0`

1.43.3 完全形式 if...else if ... else if...else

本小节介绍 if 语句的完全形式，它是对多个条件逐一判断和处理的一种语句。其语法形式为：

```

if (expr1)
    statement1
else if(expr2)
    statement2
else if ...
...
else
    statementN

```

语法规则：（多组条件 `expr1, expr2, ...`，逐个判断，如果满足条件则执行相应语句）

如果 `expr1` 非 0，则执行 `statement1`，然后退出语句；

如果 `expr2` 非 0，则执行 `statement2`，然后退出语句；

... ..

如果所有条件均未满足，则执行最后一个 `else` 语句。

注意：

（1）如果有一个 `else if` 得到满足，那么后面就不会接着判断其他条件了。

（2）最后面的 `else` 语句可以省略，即 `if ... else if ... else if`，最后可以没有 `else` 语句

例如，仍然用 `x` 表示一次考试的成绩，分为 ABCDE 五档成绩，位于 `[90,100]` 为 A，`[80,90]` 为 B，`[70-80]` 为 C，`[60,70]` 为 D，`[0-60]` 为 E。在示例 CH07_B4 中，用 `if` 语句实现。

////////// 例 CH07_B4 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int x = 61; // x 可能通过 scanf 让用户输入
05      if(x>=90)
06          printf("Got A \n");
07      else if(x>=80)
08          printf("Got B \n");
09      else if(x>=70)
10          printf("Got C \n");
11      else if(x>=60)
12          printf("Got D \n");
13      else
14          printf("Got E \n");
15      printf("That's over \n");
16      return 0;
17 }
```

过程分析：令 `x=71`

=> 执行[04] : `int x = 71`

=> 执行[05] : 表达式 $x \geq 90$ 的值为 0, 继续判断其他条件
=> 执行[07] : 表达式 $x \geq 80$ 的值为 0, 继续判断其他条件
=> 执行[09] : 表达式 $x \geq 80$ 的值为 0, 条件满足, 执行此条 else if 语句
=> 执行[10] : printf("Got C \n"), 然后跳出 if, 不再判断其他条件
=> 执行[15] : printf("That's over \n")
=> 执行[16] : return 0

1.44 switch 语句

这一节介绍 switch 语句, 它的作用是根据不同的选项, 跳转到不同的分支处理。其语法形式为:

```
switch(expr)
{
case OPTION_1:
    break;
case OPTION_2:
    break;
case OPTION_....:
    break;
default:
    break;
}
```

其中, **expr**: 表达式的值必须为整型, **OPTION**: 必须为整形常量。

语法规则: 根据 **expr** 的值, 寻找匹配的 case。如果某个 **OPTION_X** 与 **expr** 相等, 则跳转到 **OPTION_X** 处执行。如果没有任何匹配的 case, 则跳到 default 处执行。如果不存在 default 标签, 则退出 switch 语句。

其语法要素为:

- ① **expr** 必须为整型, 不能是小数或其他类型
- ② **OPTION** 必须是整型常量, 不能是变量。
- ③ case 与 default 行称为标签, 行尾以冒号结束
- ④ 允许不写 default 标签

⑤ 允许不写任何 case 标签

我们来看一个例子，通过这个例子来学习它的用法。

例 CH07_C1：周一去上班，周二去出差，其他待在家。代码如下：

//////////例 CH07_C1 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int day = 2; // 用户输入数字 0-6，代表周日、周一到周六
05      switch(day)
06      {
07          case 1:
08              printf("Go to office\n");
09              break;
10          case 2:
11              printf("Go out \n");
12              break;
13          default:
14              printf("Stay at home \n");
15              break;
16      }
17      return 0;
18  }
```

过程分析：

第 1 种情况：令 day=2

=> 执行[05]：switch(day)，寻找匹配的 case ... 发现第[10]行匹配，跳过去执行...

=> 执行[11]：printf("Go out \n")

=> 执行[12]：break; (跳出 switch 语句)

=> 执行[17]：return 0;

第 2 种情况：令 `day=3`

=> 执行[05] : `switch(day)`...没有匹配的 `case`...发现 `default`，跳过去执行...

=> 执行[14] : `printf("Stay at home \n");`

=> 执行[15] : `break;` (跳出 `switch` 语句)

=> 执行[17] : `return 0;`

1.44.1 匹配

标签的匹配过程详细分解为 3 步：

(1) 寻找匹配的 `case`

(2) 如果没有 `case` 匹配，寻找 `default`

(3) 如果没有 `case` 匹配，也没有 `default`，则退出 `switch` 语句

1.44.2 跳转与执行

当发现到匹配的标签后，直接跳转到该标签的下一行继续执行。注意，标签本身不算是一行语句。跳过到指定位置后，程序会一直执行，直到：

(1) 到达 `switch` 语句的末尾（大括号的结束），自然退出 `switch` 语句

(2) 或者遇到 `break` 语句，则直接退出 `switch` 语句

其中，第 (2) 条是需要仔细理解的。在关键字 `break` 后直接加上分号，就是称为 `break` 语句，其作用是跳出当前层次的 `switch` 语句。

每个分支处理后面都应该加上一条 `break` 语句，表示分支处理结束。但如果不加 `break` 语句，也并不是语法错误，只是运行的结果可能会显得意外，那么运行时会发生什么呢？

在示例 CH07_C2 中，每个 `case` 和 `default` 分支处理中都没有加 `break` 语句，令 `x=1`，观察运行的结果。

////////// 例 CH07_C2 //////////

```
01  #include <stdio.h>
02  int main()
03  {
04      int day = 1; // 用户输入数字 0-6，代表周日、周一到周六
05      switch(day)
06      {
07          case 1:
08              printf("Go to office\n");
```

```

09         case 2:
10             printf("Go out \n");
11         default:
12             printf("Stay at home \n");
13     }
14     return 0;
15 }

```

程序分析：令 day=1

=> 执行[05] : switch(day) ... 发现第[07]行 case 匹配，跳过去执行...

=> 执行[08] : printf("Go to office\n");

=> 执行[10] : printf("Go out \n");

=> 执行[12] : printf("Stay at home \n");

=> 执行[14] : return 0;

是的，虽然语法上没有错误，可是运行结果并不是我们所希望的。我们原本希望当输入 day=1 时，只执行 Go to office。可是由于我们没有加 break 语句，使得程序继续往下执行，连续执行了后面的 Go out 和 Stay at home。

1.44.3 注意事项

在一般情况下，当我们自己书写代码时，按照常规的框架来写，不太可能发生错误。如下面所示意的这样：

```

switch (...)
{
case ...
    break;
case ...
    break;
default:
    break;
};

```

但当我们阅读别人的代码时，如果别人的代码不是这么规范，就得注意了。switch 语句可能出错的语法细节比较多，我们一一核对一下。

- (1) 检查有没有 break 语句，如果没有 break 语句，则程序会继续往下执行的。

(2) 多个标签可以写在一起

```
switch (...)
{
case 1: case 2:
case 3: case 4:
    printf("yes, 1234 is ok \n");
    break;
}
```

表示当选项为 1, 2, 3, 4 时都跳转到相同的位置执行

(3) default 标签的位置是自由的

像下面这样写也没问题。

```
switch (...)
{
default:
    break;
case ...
    break;
};
```

(3) switch 和 case 的选项值都必须是整型

像下面这样是有语法错误的，

```
switch(1.234)
{
}
```

(4) case 的选项值必须是常量

在下面的代码中，case 的值用了变量，这样是不允许的。

```
int OPTION = 3;
int choice = 0;
```

```
switch(choice) // OK, 此处可以是变量
{
case OPTION: // 语法错误: case 的值必须是常量
}
```

1.45 for 语句

1.45.1 引例

本小节通过一个例子, 来说明存在什么样问题, 而本节的语法规则是用于解决此类问题的。

引例:

现有一个长度为 100 的 int 数组, 要求初始化各元素的值为 1,2,3,..., 100。我们可以不怕麻烦地写上一百行, 来完成这个要求:

```
int a[100];
a[0] = 1;
a[1] = 2;
... 此处略去几十行 ...
a[99] = 100;
```

对于这样的不胜其烦的工作, 有没有什么好的解决办法呢?

1.45.2 for 语句

使用 for 语句, 可以完成一些可以用循环去完成的、带有规律性的工作。其语法形式为:

```
for ( expr1 ; expr2; expr3)
    statement
```

其中, `expr1`, `expr2`, `expr3` 是 3 个表达式, `statement` 是一条单语句或复合语句。

语法规则: (循环如何被执行)

- ① 初始化: 执行 `expr1`。把 `expr1` 称为初始化表达式。
- ② 终止条件: 执行 `expr2`。若 `expr2` 的值为真, 则执行第③步。如果 `expr2` 为假, 则退出 for 语句。
- ③ 循环体: 执行 `statement`
- ④ 后置表达式: 执行 `expr3`
- ⑤ 继续下一轮循环: 跳到第②步

简而言之：每一轮开始之前，都要先判断一个 `expr2`，如果 `expr2` 为假则退出循环。

现在我们使用 `for` 语句来完成引例中的要求。代码如下：

////////// 例 CH07_D1 //////////

```
#include <stdio.h>

int main()
{
    int a[100];
    int i;
    for(i=0; i<100; i++)
    {
        a[i] = i + 1;
    }
    return 0;
}
```

程序分析：

- ① 初始化 `i = 0` （只执行一次）
- ② 判断终止： `i` 为 0， `i<100` 为真，条件满足，继续循环
- ③ 循环体： `a[i] = i + 1`；则 `a[0]` 的值为 1
- ④ 后置语句： `i++` ， `i` 的值变成 1
- ⑤ 判断终止： `i` 为 1， `i<100` 为真，条件满足，继续循环
- ⑥ 循环体： `a[i] = i + 1`；则 `a[1]` 的值为 2
- ⑦ 后置语句： `i++` ， `i` 的值变成 2
- ⑧ 判断终止： 如此反复... ..
- ⑨ 判断终止： : `i` 为 100， `i<100` 为假，退出 `for` 语句

1.45.3 变形 1：省略初始表达式

`for` 语句的变形的一种，就是省去初始表达式。下面的写法和例 CH07_D1 的效果是完全一样的。

////////// 例 CH07_D2 //////////

```
#include <stdio.h>
```

```

int main()
{
    int a[100];
    int i = 0; // 在 for 语句之前面初始化
    for( ; i<100; i++) // for 语句中省去第一个表达式
    {
        a[i] = i + 1;
    }
    return 0;
}

```

1.45.4 变形 2: 省略初始表达式

for 语句的变形的一种，就是省去第二个表达式，表示总是执行循环体。通常情况下，可以把判断终止的语句写在 for 语句里面，并使用 **break** 语句跳过 for 循环。

下面的写法和例 CH07_D1 的效果是完全一样的。

////////// 例 CH07_D3 //////////

```

#include <stdio.h>

int main()
{
    int a[100];
    int i;
    for( i=0 ; ; i++) // 省去第 2 个表达式
    {
        if(i>=100)
        {
            break; // break 用于退出 for 语句
        }
        a[i] = i + 1;
    }
    return 0;
}

```


1.45.5 变形 3: 省略后置表达式

for 语句的变形的一种，就是省去第三个表达式，取而代之地是，把后置语句写在循环体的最后面。

```
////////// 例 CH07_D4 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    int a[100];  
    int i;  
    for( i=0 ; i<100 ; ) // 省去第 3 个表达式  
    {  
        a[i] = i + 1;  
        i++; // 后置过程写在大括号里面  
    }  
    return 0;  
}
```

1.45.6 变形 4: 全部置空

全部置空也是可以的，相当于没有初始化，没有后置语句，永远执行的循环。

```
for( ; ; )  
{  
    printf("... never stop ..... \n");  
}
```

由于没有终止条件，所以上述循环体被无限次地执行下去。

1.45.7 break 语句

如 for 语句中存在 break 语句，则它的作用是跳出 for 语句，退出循环。前面已经有演示了。

1.45.8 continue 语句

continue 语句如果存在于 for 语句的大括号内，当 continue 语句被执行时，表示结束本轮、直接进入下一轮循环。即，continue 后的语句在本轮被忽略、不被执行。

例: 打印 1,100 之间的偶数，并统计偶数的个数。

```
////////// CH07_D5 //////////
```

```

#include <stdio.h>

int main()
{
    int count = 0;
    for(int i=1 ; i<= 100; i+=1 )
    {
        if(i % 2)
            continue; // 后面的语句被跳过...

        count ++;
        printf("Even Number: %d \n", i);
    }
    printf("Total : %d \n", count);
    return 0;
}

```

1.46 while 语句

while 语句也用于实现循环，其基本形式为：

```

while(expr)
    statement

```

其中，`expr` 表达式，`statement` 是一条单语句或复合语句。

语法规则：

- ① 执行 `expr`。若 `expr` 的值为真，则执行②。如 `expr` 为假，则退出 `while` 语句。
- ② 执行 `statement`
- ③ 跳到第①步，继续下一轮循环

简而言之：每一轮开始之前，都要先判断一下 `expr` 的值，如果 `expr` 为假则退出循环。

`while` 语句和 `for` 语句本质上没有区别，都是用于实现循环。使用 `for` 语句能够实现的功能，可以很容易地改用 `while` 语句来实现。

例如，可以使用 `while` 语句来完成例 CH07_D1 中的功能，将一个长度为 100 的数组填充上初始值，可以这么实现：

////////// 例 CH07_E1 //////////

```

#include <stdio.h>

int main()
{
    int a[100];

    int i = 0; // 初始化

    while(i<100)
    {
        a[i] = i + 1;

        i++; // 后置过程
    }

    return 0;
}

```

简单地分析一下这个程序：

- ① i 的初始值为 0
- ② 当 i<100 时，执行循环体。这意味着，循环体一共被执行了 100 次。

1.46.1 变形：判断判断内置

`while` 的循环体中也可以存在 `break` 语句和 `continue` 语句，其意义和 `for` 语句中阐述的完全一样：`break` 语句用于退出循环，`continue` 循环用于跳过本轮、直接进入下一轮循环。

`while` 语句也有一种小小的变形，那就是条件判断的表达式内置，放在大括号来判断。在下例中，`while(1)` 表示循环体总是被执行。`if(i>=100) break` 表示当 `i>=100` 成立时退出循环。

////////// 例 CH07_E2 //////////

```

#include <stdio.h>

int main()
{
    int a[100];

    int i = 0; // 初始化

    while(1) // 总是执行
    {
        if(i>=100) break; // 条件判断放在大括号里面

        a[i] = i + 1;
    }
}

```

```

        i++; // 后置过程
    }
    return 0;
}

```

1.46.2 例题

下面通过一个实例来练习 `while` 语句的用法。

例：要示用户在控制台输入多个整数，并保存到数组中。当用户输入的数字是 0 时，结束输入。最多允许输入 128 个整数。

我们可以用一个长度为 128 的数组来存储用户的输入，并用一个变量 `count` 来计数。每当用户输入一个数，我们把它存入数组中。注意，`count` 的值要初始化为 0。具体的实现代码如下：

```

//////////例 CH07_E3 //////////
#include <stdio.h>

int main()
{
    int numbers[128]; // 最多接收 128 个数
    int count = 0; // 一共接收多少个少数
    while(1)
    {
        printf("输入一个正整数:");
        int ch = -1;
        scanf("%d", &ch);
        if(ch <= 0)
        {
            // 输入数字是 0、或非法输入时，结束输入
            break;
        }
        else
        {
            // 输入数字不是 0，则保存此数字
            numbers[count] = ch;

```

```

        count ++;
    }
}
// 输出这些数
printf("----- result -----\n");
for(int i=0; i<count; i++)
{
    printf("%d ", numbers[i]);
}
}

```

1.47 do...while 语句

除了 for 语句、while 语句之外，还有一种 do...while 语句可用于表示循环。其基本形式为：

```

do
{
    statement
}while(expr);

```

其中，expr 表达式，statement 是一条单语句或复合语句。

语法规则：

- ① 执行 statement。
- ②若 expr 的值为真，则执行①。如 expr 为假，则退出 while 语句。

简而言之：先执行一轮，再检测 expr 的值，如果 expr 为真则接着执行下一轮。如果为假则结束循环。

C/C++语言中的三种循环语法 for, while, do...while 本质没有区别，可以互相转换。只是在形式上和 while 语句相比，do...while 语句适合用于“至少执行一次”的循环，它是先执行一次、再决定要不要继续循环的。

1.47.1 例题

假设用户的密码是一个 3 位整数。令用户输入密码，如果输入成功，则提示"Welcome"。如果输入失败，提示"Bad Password!"，并提示其重新输入密码。最多输入 3 次，如果 3 次均未成功，则提示"User Locked"。

我们用 do...while 语句来实现这个功能，代码如下（读者可以把它改成 for 或 while 来实现）：

```
//////////例 CH07_F1 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    int key = 123; // 密码  
    int times = 0; // 尝试次数  
    int passed = 0; // 0: 未通过验证, 1: 已输入密码,通过验证  
    do  
    {  
        times ++; // 尝试的次数  
  
        // 提示用户输入密码  
        int input = 0;  
        printf("Please input your password: ");  
        scanf("%d", &input);  
  
        // 检查密码是否正确  
        if(key == input)  
        {  
            passed = 1;  
            break;  
        }  
        else  
        {  
            printf("Bad Password!\n");  
        }  
    } while (times < 3);
```

```

// 有没有通过验证

if(passed)
{
    printf("Welcome!\n");
}
else
{
    printf("User Locked!\n");
}

return 0;
}

```

1.48 综合例题 1

例：一个班级有 15 人进行了一次考试，他/她们的考试分数用整型数组表示，对分数划分三个 3 等级：大于 80，A 档；大于 60，B 档；其他，C 档。

在例 CH07_G1 中，通过一个 for 循环来遍历数组、访问数组中的每个元素。用 if 语句来判断它的值，如果属于 A 档，则将计数 num_of_A 增加 1，依次类推。代码如下：

//////////例 CH07_G2 //////////

```

#include <stdio.h>

int main()
{
    int score[23] =
    {
        89, 98, 80, 87, 56, 57, 68, 68,
        65, 87, 78, 77, 65, 98, 67
    };

    int num_of_A = 0;
    int num_of_B = 0;
    int num_of_C = 0;
    for(int i=0; i<23; i++)
    {
        int s = score[i];
    }
}

```

```

    if (s >= 80)
    {
        num_of_A ++;
    }
    else if(s >= 60)
    {
        num_of_B ++;
    }
    else
    {
        num_of_C ++;
    }
}

printf("A: %d, B: %d, C: %d \n",num_of_A, num_of_B, num_of_C );
return 0;
}

```

1.49 综合例题 2

打印如下图所示的图形（ $n \times n$ 个字符，对角线是*号，其他为空格）。左图是 10x10，右图是 9x9 的字符阵，如图 7-1 所示。



图 7-1 字符阵图形

以 10x10 为例， $n=10$ ，先找找其规律：

第 0 行：0 列为*，9 列为*

第 1 行：1 列为*，8 列为*

...

第 i 行: i 列为*, n - 1 - i 列为*

参考代码如下:

//////////例 CH07_G2 //////////

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int n = 10;
```

```
    for(int i=0; i<n; i++) // i: 行号
```

```
    {
```

```
        for(int j=0; j<n; j++) // j:列号
```

```
        {
```

```
            if(j == i || j == n-i-1)
```

```
            {
```

```
                printf("*");
```

```
            }
```

```
            else
```

```
            {
```

```
                printf(" ");
```

```
            }
```

```
        }
```

```
        printf("\n"); // 每行末尾
```

```
    }
```

```
}
```

第08章 函数

本章介绍函数的概念。函数是一个被包装了的功能模块，送给它一些输入参数，它经过运算得出一个结果并输出返回。

1.50 引例

函数是一个可以完成指定功能的模块，它能执行任务，并把结果返回来。通常我们把函数描述为一个功能盒，送进去一些输入数据，它就产生一些输出数据。如图 8-1 所示。

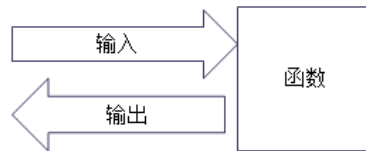


图 8-1 函数模块化示意图

打个比方，榨汁机可以为是一个函数，我们放进去一些水果，让它开始工作，它执行任务，把果汁返回。我们把送进去的“水果”称为输入参数，把函数的结果“果汁”称为返回值。可以用图 8-2 表示榨汁机的工作过程。

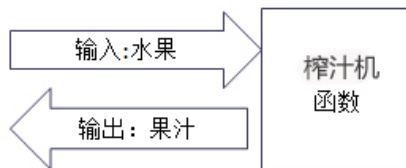


图 8-2 榨汁机模块示意图

作为使用者，我们通常不会关心函数内部是如何工作的。我们关心的是，它要求输入什么形式的数据，以及它能够返回什么的数据。

再比如，有一个函数，它可以用来求一个数的三次方。这个函数的输入是一个数 a ，输出（返回值）是 a 的三次方。我们把这个函数命名为 **Cube**，表示求三次方。当输入为 2 时，返回值为 8。当输入为 3 时，返回为 27。如图 8-3 所示。

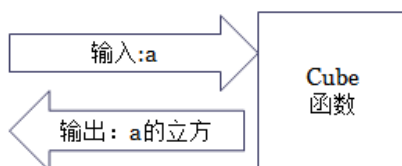


图 8-3 Cube 函数示意图

那么，在 C/C++ 中，函数是如何定义的呢？

1.51 初步认识函数

下面介绍一下函数的定义。其语法形式为：

```
return_type  name ( arguments )  
  
{  
    body  
}
```

其中，

name: 函数名。最好起一个有意义的名字。

arguments: 参数列表。每个参数以逗号分开，如 `int a, int b`。

return_type: 返回值类型。

body: 函数体。即函数的具体实现。

在下面的例子中，首先定义了一个函数 `cube`，用于求一个数的三次方。然后在 `main` 函数里使用了这个函数。

////////// 例 CH08_A1 //////////

```
#include <stdio.h>  
  
int cube (int a)  
{  
    int result = a * a * a; // 计算出结果  
    return result; // 返回结果  
}  
  
int main()  
{  
    int n = cube ( 12 ); // 调用函数  
    printf(" result: %d \n", n);  
    return 0;  
}
```

这里定义一个函数，它的几个要素为：

函数名: `cube`

参数列数: `int a`，表示我们应该传入一个 `int` 型的值

返回值： `int` ，表示该函数将返回一个 `int` 型的值

由大括号包围的若干语句，称为函数体。在函数体中，至少有一条 `return` 语句，该语句用于指定函数返回值。也就是说，我们先计算出结果，然后用 `return` 语句将这个值返回。

下面讲一下这个函数是如何调用的。

```
int n = cube ( 12 ); // 调用函数
```

使用函数名来调用一个函数，在小括号传入对应参数的值，然后用一个变量 `n` 来保存函数的返回值。

`cube` 函数需要一个 `int` 型的参数，我们传入的是一个 `int` 型的值 12，那么在 `cube` 函数运行时，首先会对参数列表里的变量（参变量）进行初始化：

```
int a = 12;
```

然后再进入函数体，执行

```
int result = a * a * a; // 计算出结果
```

```
return result; // 返回结果
```

我们总结一下：

（1）定义函数

- 确定函数的名字
- 确定需要哪些参数
- 确定函数返回值的类型
- 函数体：计算出结果，用 `return` 返回

（2）调用函数

调用函数时，不用关心函数体里的实现，只关心它的名字、参数列表和返回值类型。

1.52 函数的定义

本节具体展示如何定义一个函数。

1.52.1 函数名

函数名的命名规则，就和 C/C++ 里的所有命名规则一样，“字母、数字和下划线的组合，但不能以数字开头”。

除此之外，为了让我们的代码更容易读懂（可读性），我们应该：

（1）命名要有意义

让函数的名字反映其功能，让人可以顾名思义。

例如：

`save_data` 保存数据

`get_input` 获取输入

`average` 求平均值

（2）规范化命名

通常，在 C/C++ 有两种函数命名法。本书中两种方式都有使用。

第一种：全小写，每个单词以下划线分隔，如 `save_data`。这是 C 语言里的常见方式。

第二种：每个单词的首字母大写，不用下划线。如 `SaveData`。这是 C++ 里的常见方式。

1.52.2 参数列表

参数列表在函数名后面的小括号里指定，每个参数要有类型、参数名，每个参数之间用逗号分隔。下面是一些示例：

```
int test (); // 允许不带参数
```

```
int test (int a); // 带一个参数：参数类型 int, 参数名 a
```

```
int test (int a, double b); // 带一个 int 型参数，一个 double 型参数
```

在函数的实现中，那些在函数内部不能确定的量，就定为参数，放在参数列表里。

例如，函数 `Cube` 用于求一个数的立方。算法是不变的，但输入的那个数是不确定的。因此，用一个 `int` 型参数表示输入的数。

```
int cube (int n);
```

例如，求两个数的平方和。自然地，函数需要外界来指定这两个数，于是这两个数定为参数。

```
int average(int a, int b);
```

通常一个函数在规定其任务的时候，它所需要的大部分参数就基本明确了。

1.52.3 返回值类型

要为函数确定一个合适的返回值类型。

例如，

函数 `sum` 用于求两个整数的和。由于两个整数的和仍然是整数，所以我们可以 `int` 类型作为返回值类型。

```
int sum (int a, int b);
```

函数 `Average` 用于三个整数的平均值。由于三个整数的平均值可能为小数，所以我们应该用 `double` 型作为返回值类型。

```
double average (int a, int b , int c)
```

1.52.4 函数的实现

大括号里的语句称为函数的实现，又叫函数体。

通常，我们先计算出结果，然后有一个 `return` 语句将结果返回给调用者。

例如，

```
////////// 例 CH08_B1 //////////
```

```
double Average (int a, int b , int c)
{
    double result = (a + b + c) / 3.0; // 计算出结果
    return result; // 用 return 语句返回
}
```

注意，有些初学者以为 `printf` 打印出来的就是返回值。这是错误的，一定要记住返回值必须用 `return` 语句，跟 `printf` 没有关系。

```
double Average (int a, int b , int c)
{
    double result = (a + b + c) / 3.0; // 计算出结果
    printf("%d", result); // 初学者易犯的错误：不要 printf，请用 return 语句
}
```

1.52.5 return 语句的用法

`return` 语句的作用：

- (1) 返回一个值
- (2) 函数立即退出

当 `return` 语句被执行后，无论 `return` 语句处在什么层次的复合语句内，函数立即退出。函数内的任何语句都不再执行。

在示例 CH08_B2 中，`printf("bbbb\n")` 不会被执行。

////////// 例 CH08_B2 //////////

```
int test()
{
    printf("aaaa\n");
    return 0; // 此处函数退出

    printf("bbbb\n"); // 不被执行

    return 1; // 不被执行
}
```

这意味着，`return` 语句不必一定得写在函数体的最下面一行。实际上，只要我们已经得到函数的结果，就可以立即用 `return` 返回结果并退出函数。

例：判断一个字母是大小还是小写。如果是大写字母，返回 1。如果是小写字母，返回-1。如果根本不是字母，返回 0。在例 CH08_B3 中，定义了函数 `test`，并在 `main` 函数里调用它。

////////// CH08_B3 //////////

```
#include <stdio.h>

int test(char ch)
{
    if(ch >= 'A' && ch <= 'Z')
    {
        return 1;
    }

    else if(ch >= 'a' && ch <= 'z' )
    {
        return -1;
    }

    return 0;
}

int main()
```

```

{
    int r = test ('M');
    printf("result: %d \n", r);
    return 0;
}

```

1.53 函数的调用

在 C/C++里，把函数的使用称为“调用” (Call, Invoke)。在调用一个函数时，必须知道这个函数的名称、参数的个数及类型、返回值类型。

1.53.1 函数的调用过程

我们用前面的求立方的例子，来说明函数的调用过程。

////////// 例 CH08_C1 //////////

```

01  #include <stdio.h>
02  int Cube (int a)
03  {
04      int result = a * a * a; // 计算出结果
05      return result; // 返回结果
06  }
07  int main()
08  {
09      int n = Cube ( 4 ); // 调用函数：传入参数，得到返回值
10      printf(" result: %d \n", n);
11      return 0;
12  }

```

在 C/C++中，程序总是从 main 函数开始运行，于是程序是从第[09]行开始的：

=> 执行[09]：调用 Cube (4)

=> 执行[03]：进入函数 Cube，参变量被初始化 int a = 4

=> 执行[04]：计算 result 为 64

=> 执行[05]：return 被执行，程序退出。

=> 执行[09]：回到原先函数调用的地方，Cube 函数返回值 64 被赋值给了 n

=> 执行[10] : printf 打印出 result 的值

=> 执行[11] : return 被执行, main 函数返回。当 main 函数返回时, 意味着整个程序退出。

1.53.2 注意参数的顺序

在传递参数值给函数时, 要注意参数的顺序。

例如, 一个函数用于求一个数的 n 次方式, 其函数定义为:

////////// 例 CH08_C2 //////////

```
int Power(int base, int n)
{
    int result = 1;
    for(int i=0; i<n; i++)
    {
        result *= base;
    }
    return result;
}
```

这个函数要求第一个参数是基数的值, 第二个参数是幂的值。当我们求 2 的 5 次方时, 应该 Power(2, 5), 而不是 Power(5, 2)。要注意这个顺序问题。

1.53.3 函数的传值调用

这小节讲阐述一个重要的概念: 传值调用。

先看一个例子。在例 CH08_C3 中, Test(n)的调用会不会修改 n 的值呢? 有人会认为 n 的值变成了 1001, 有人会认为 n 的值保持不变。

////////// 例 CH08_C3 //////////

```
#include <stdio.h>

void Test(int a)
{
    a += 1000;
}

int main()
{
```

```

    int n = 1;

    Test(n);

    printf("Now: n=%d \n", n); // n 的值是 1001 吗?

    return 0;

}

```

经过测试，我们发现无论在 Test 函数里怎么操作，main 函数里的变量 n 的值始终不变。所以我们有以下结论。

(1) 参变量 a 和原变量 n 是两个不同的变量

Test 函数里的参变量 a，和 main 函数里的 n，是两个不同的变量，对应于不同的内存地址。所以，在 Test 函数里对 a 的修改是不会影响到 n 的值的。

我们知道，一个变量的地址用 & 操作符可以取得，并在第 3 章中已经学会了如何打印一个变量的地址。于是我们用下面的代码检验一下，看 a 和 n 这两个变量的地址是否相同。

```

//////////例 CH08_C4 //////////
#include <stdio.h>

void Test(int a)
{
    a += 1000;

    printf("a 的地址: %08X \n", &a);
}

int main()
{
    int n = 1;

    printf("n 的地址: %08X \n", &n);

    Test(n);

    printf("Now: n=%d \n", n); // n 的值是 1001 吗?

    return 0;

}

```

经过测试，可以发现 a 和 n 的地址不同，从而证明了 a 和 n 是不同变量。

(2) 传的是变量的值

实际上，函数的调用过程是传值调用。Test(n)所表示的意思是，“把 n 的值传给函数”，而不是说把 n 这个变量传给函数。

在前面“函数的调用过程”这一节里，我们已经展示了参变量的初始化。也就是说，在进入函数时相当于执行了一句这样的操作：

```
int a = n; // 把 n 的值赋给参变量 a
```

1.53.4 忽略返回值

当调用一个函数时，虽然这个函数有返回值，但是可以忽略这个返回值。也就是说，不用它的返回值。

例如，

```
//////////例 CH08_C5 ////////////  
  
#include <stdio.h>  
  
int Test(int a)  
{  
    printf("this is a test \n");  
    return a* a;  
}  
  
int main()  
{  
    Test ( 2 ); // 忽略 Test 函数的返回值  
    return 0;  
}
```

1.53.5 直接使用返回值

不需要定义一个变量来接收返回值，可以直接使用它。

```
//////////例 CH08_C6 ////////////  
  
#include <stdio.h>  
  
int Test(int a)  
{  
    return a* a;  
}  
  
int main()
```

```

{
    printf("Result: %d \n", Test ( 2 )); // 忽略 Test 函数的返回值
    return 0;
}

```

1.54 全局变量和局部变量

在函数内定义的变量，叫做局部变量（Local Variable）。（包含参变量）

在函数外定义的变量，叫做全局变量（Global Variable）。

我们来看一个全局变量的例子。在例 CH08_D1 中，变量 `number` 定义在函数体之外，它是一个全局变量。代码如下所示：

```

////////// 例 CH08_D1 //////////

#include <stdio.h>

int number = 0; // 定义在函数之外，是全局变量

void Add(int n)
{
    number += n;
}

int main()
{
    number = 10;
    Add(1);
    printf("now: %d \n", number);
    return 0;
}

```

全局变量常用于存储一些全局性的数据，它在各个函数中均可以访问。比如，在上例中，在 `main` 函数可以访问 `number`，将 `number` 赋值为 10。在 `Add` 函数也可以访问 `number`，将 `number` 的值增加 `n`。

如果某些功能用局部变量不方便完成，可以考虑使用全局变量。

例如，我们要求一个数组的所有元素的平均值。然而，我们并不知道如何将一个数组内的值全部传递给函数，这时候我们可以用全局变量来实现。在例 CH08_D2 中，将数组定义在函数之外，以便在 `Average` 函数中可以直接访问其值。

```

////////// 例 CH08_D2 //////////

#include <stdio.h>

// 定义一个全局变量

int data[8] = {1,2,3,4,5,6,7,8};

// 不必传入参数

double Average()
{
    // 在函数里直接读取全局变量 data

    int total = 0;
    for(int i=0; i<8; i++)
    {
        total += data[i];
    }

    return total / 8.0;
}

int main()
{
    double result = Average();

    printf("result: %.3lf \n", result);

    return 0;
}

```

什么时候用局部变量？什么时候用全部变量？原则很简单：如果能用局部变量完成，那就不要用全局变量。

1.55 变量的作用域与生命期

1.55.1 变量的作用域

每一个变量，都有一个有效范围，称为“作用域”。在这个范围之内，这个变量是可以访问的。

局部变量的作用域：

- (1) 从定义之处起生效
- (2) 至大括号结束后失效 (该变量所在的大括号)

在下例中，第一个 `printf` 行中可以访问变量 `b`，因为此处还在变量 `b` 的作用范围内。而第二个 `printf` 行就已对超出一变量 `b` 的作用域，因而编译器报错。

```
#include <stdio.h>

int main()
{
    if(1)
    {
        int b = 10; // b 生效

        printf("b: %d\n", b); // 可以访问 b
    } // b 失效

    printf("b: %d\n", b); // 编译错误!

    return 0;
}
```

1.55.2 变量的生命期

变量的生命期，是指在程序运行的时候，变量存在的时间。“生命期”这个概念，实质上与“作用域”说的是同一件事。作用域是从代码角度来说问题，而生命期则是从运行时的角度来说问题。

（1）全局变量的生命期是有恒的

在程序运行期间，该全局变量始终存在，始终可以访问。

（2）局部变量的生命期是短暂的

局部变量的生命期和其作用域是一致的。自定义之处，变量的生命期开始，变量是有效的。当超出作用域后，该变量生命期结束，该变量失效、不再可以访问。

也就是说，局部变量超出它所在的大括号，其生命期就结束了。

1.56 变量名重名问题

这一节介绍变量名的重名问题。为了便于进一步说明作用域和生命期的相关原理，按照大括号的层次进行分级。将全局变量定位 0 级。如下所示：

```
int test()
{
    int a = 0; // 第 1 层级

    if(1)
```

```

{
    int b = 0; // 第 2 层级
    while(1)
    {
        int c = 0; // 第 3 层级
    }
}
return 0;
}

```

如果代码已经使用规范的缩进的话，那么缩进的位置刚好对应了这个变量的层级。

（1）不同函数内的变量，允许重名

如果两个变量在不同的函数里，那肯定是可以重名的，两者之间不会有影响。

在例 CH08_E1 中，Test 函数中的变量 a 和 b 和 main 函数中的同名变量 a,b 是完全没有影响的。

```

////////// CH08_E1 //////////
#include <stdio.h>
int Test(int a)
{
    int b = a + 10;
    int c = a + b;
    return c;
}
int main()
{
    int a = 10;
    int b = Test(a);
    return 0;
}

```

（2）可以和上一层次的变量重名

一个变量可以和它上面层次的变量重名。当重名时，优先找本层级的变量。

////////// 例 CH08_E2 //////////

```
#include <stdio.h>

int main()
{
    int a = 1;
    if(1)
    {
        int a = 2; // 定义一个同名的变量
        printf("level2: a=%d \n", a); // 访问的是本层级的变量 a，打印值为 2
    }
    printf("end: a= %d\n", a); // 打印值为 1
    return 0;
}
```

(3) 就近原则

当重名发生时，实际被访问的变量是跟本层次最近的那个变量。在下例中，不同的层次和有一个变量 a，那实际被访问的变量是离它最近的变量。所以输入值为 101。

////////// 例 CH08_E3 //////////

```
#include <stdio.h>

int a = 100;

int main()
{
    int a = 101;
    if(1)
    {
        printf("a=%d \n", a); // 最近的变量 a 是 101
    }
    return 0;
}
```

注：显然地，只在两个变量处于同一层级时，才不允许重名。

1.57 函数声明与函数定义

当一个 cpp 文件里有多个函数，且它们之间有调用关系时，那么它们的先后顺序就变得复杂。我们需要保证，一个函数调用之前，该函数已经被定义。

例如，现在的任务是打印所有的字母的 ASCII 码的值。我们可以通过几个函数来实现。在例 CH08_E1 中，我们定义函数 `is_alpha` 用于判断一个字符是否为字母，在函数 `print_ascii` 又调用了 `is_alpha` 函数。

```
////////// 例 CH08_F1 //////////  
  
#include <stdio.h>  
  
// 如果 ch 是字母，则返回 1；否则返回 0  
  
int is_alpha(char ch)  
{  
    if(ch >= 'a' && ch <= 'z')  
        return 1;  
    if(ch >= 'A' && ch <= 'Z')  
        return 1;  
    return 0;  
}  
  
// 打印所有字母的 ASCII 码  
  
void print_ascii()  
{  
    for(int i=0; i<127; i++)  
    {  
        if(is_alpha(i))  
        {  
            printf("%c - %d \n", i, i);  
        }  
    }  
}  
  
int main()  
{  
    print_ascii();  
}
```

```

    return 0;
}

```

总体来说，这个 `cpp` 文件中的三个函数呈如下调用关系

```
main() -> print_ascii() -> is_alpha()
```

因而它们在书写顺序上要求保证 `is_alpha` 在最前面，`print_ascii` 在中间。可以想象，当函数越来越多时，要保证顺序就会变得麻烦。

1.57.1 函数的声明

函数的声明（Declaration），在形式上就是把函数定义中的函数体去掉，只保留函数名、参数列表、返回值类型。并以分号结束。例如，

```

int is_alpha(char ch);
void print_ascii();

```

此外，还有一个术语：函数原型（Prototype），也是指函数名、参数列表和返回值类型这三个要素。不过函数原型通常用于日常沟通和文档描述中，并不是一个语法成分。

当存在一个函数被声明后，它就可以被直接调用，而不一定要把函数定义也放在前面。这就解决了前面说的函数调用顺序问题。我们通常把各个函数的声明先写在前面，然后函数定义就不需要保证顺序了。

例如，我们对例 `CH08_F1` 采用函数声明的写法，

```
//////////例 CH08_F2 //////////
```

```

#include <stdio.h>

// 声明本页面内的函数

int is_alpha(char ch);
void print_ascii();

int main()
{
    print_ascii();
    return 0;
}

// 打印所有字母的 ASCII 码

void print_ascii()
{
    for(int i=0; i<127; i++)

```

```

    {
        if(is_alpha(i))
        {
            printf("%c - %d \n", i, i);
        }
    }
}

```

// 如果 ch 是字母，则返回 1； 否则返回 0

```

int is_alpha(char ch)
{
    if(ch >= 'a' && ch <= 'z')
        return 1;
    if(ch >= 'A' && ch <= 'Z')
        return 1;
    return 0;
}

```

1.57.2 函数声明相关问题

(1) 在函数声明中，参数名是可以省略的

例如，

```
int is_alpha( char ); // 只需要指定参数类型，不需要参数名字
```

(2) 函数声明中的参数名，和函数定义时的参数名可以不同

例如：

```

void test( char  a); // 声明里的参数名无关紧要
void test( char  m)
{
}

```

(3) 函数声明的作用

函数声明，作用是向编译器声明，存在这么一个函数，名字是什么，参数类型是什么，返回值是什么。这样编译器便心中有数，可以为我们检查函数调用的正确性。

1.58 main 函数

main 函数是一个特殊的函数。无论程序中有多少个函数，第一个被执行的函数总是 main 函数。换句话说，main 函数是程序的入口。

main 函数的原型为： `int main()` 。

1.59 参数的隐式转换

一般来说，我们要求传递的值的类型，和函数要求的参数类型应该是一致的。

例如，在例 CH08_G1 中，Power 函数要求传入的两个参数为 `int, int` 类型。在调用的时候，`Power(2,5)`，传入的是实际类型是 `int, int`，因而参数类型完全匹配，没有问题。

```
////////// CH08_G1 //////////  
  
#include <stdio.h>  
  
int Power(int base, int n)  
{  
    int result = 1;  
    for(int i=0; i<n; i++)  
    {  
        result *= base;  
    }  
    return result;  
}  
  
int main()  
{  
    int result = Power(2, 5);  
    printf("result: %d \n", result);  
    return 0;  
}
```

我们考虑，如果参数类型不匹配，是不是就一定不能调用了呢？比如，我们调用 `Power(2.1, 5)`，试图求 2.1 的 5 次方。此时，传入的参数值的类型为 `(double, int)`，不匹配，但编译器还是允许通过。

其原理是这样的：当参数类型不匹配时，编译器尝试对参数进行隐式转换。如果允许隐式转换，则编译通过。如果不能隐式转换，则编译不通过。

当我们调用 `Power(2.1, 5)` 时，编译器相当于做了如下尝试：

```
int base = 2.1 ; // 参数不匹配，但支持隐式转换，base 的值为 2（小数部分被截断）
```

```
int n = 5; // 参数匹配
```

所以，编译器是允许其编译通过的，只是报了一个警告（数据被截断）。我们实际运行的结果也显示，`Power(2.1, 5)` 和 `Power(2, 5)` 的结果都是 32，这是因为参数的值被截断为整数了。

1.60 *函数名重载

（*初学者可以跳过本节）

在 C++ 里，允许两个函数的名字相同，称为函数名重载。也就是说，在 C++ 里只有名字相同、参数列表也相同（参数个数，参数类型）也相同的函数，也被认定为重复。

例如，以下两个函数是不同的函数：

```
double find_max(double a, double b); // 求两个数中的较大值
```

```
double find_max(double a, double b, double c); // 求三个数中的最大值
```

当重载函数被调用时，编译器会根据参数的个数和类型来匹配不同的函数。在示例 CH08_C1 中，重载了两个名为 `find_max` 的函数，并在 `main` 函数中调用。

```
////////// 例 CH08_H1 //////////
```

```
#include <stdio.h>
```

```
// 求两个数中的较大值
```

```
double find_max(double a, double b)
```

```
{  
    return a > b ? a : b;  
}
```

```
// 求三个数中的最大值
```

```
double find_max(double a, double b, double c)
```

```
{  
    double m = a > b ? a : b;  
    return m > c ? m : c;  
}
```

```

}
int main()
{
    double m1 = find_max(1.1, 2.2);
    double m2 = find_max(1.1, 2.2, 3.3);
    return 0;
}

```

注：在比较函数是否相同时，忽略其返回值。例如，以下两个函数是否重复呢？

```

double Test (double a, double b)
{
}

int Test ( double m, double n)
{
}

```

根据规则，在比较时只比较函数名称和参数列表，由它们的名称都是 `Test`，参数都是 `(double, double)`，所以这两个函数是重复的，编译器会报错。返回值类型不参与比较。

1.61 *重载函数的匹配

当函数名被重载后，函数的匹配过程过程：首先寻找能精确匹配的函数；如果未能精确匹配，则尝试找一个可以模糊匹配的函数（隐式转换）。

- ① 精确匹配：参数个数相同，类型相同
- ② 模糊匹配：参数个数相同，类型不同、但支持隐式转换

1.61.1 精确匹配

在例 CH08_J1 中，调用 `find_max(1,2)`，其参数值的类型为 `(int, int)`，因而精确匹配到第 2 个 `find_max(int, int)`。

```

//////////例 CH08_J1 //////////

#include <stdio.h>

// 求两个 double 数中的较大值

double find_max(double a, double b)

```

```

{
    return a > b ? a : b;
}
// 求两个 int 数中的较大值
int find_max(int a, int b)
{
    return a > b ? a : b;
}
int main()
{
    double m1 = find_max(1,2); // 参数值的类型(int, int)
    return 0;
}

```

1.61.2 模糊匹配

示例 CH08_J2 展示了没有精确匹配，但可以模糊匹配的情形。实际输入的参数值为(int, int)，而备选的功能为 find_max(double, double)。虽然未能精确匹配，但是支持对参数进行隐式转换，所以最终匹配成功。

```

////////// 例 CH08_J2 //////////
#include <stdio.h>
// 求两个数中的较大值
double find_max(double a, double b)
{
    return a > b ? a : b;
}
// 求三个数中的最大值
double find_max(double a, double b, double c)
{
    double m = a > b ? a : b;
    return m > c ? m : c;
}
int main()

```

```

{
    double m1 = find_max(1, 2);
    return 0;
}

```

下面的例子 CH08_J3 则展示了，有多个备选函数，均支持隐式转换的情形。实际输入的参数值为(int, int)，没有精确匹配的函数，但存在 find_max(double, double)和 find_max(float, float)均支持隐式转换。在这种情况下，编译器无法自己决定使用哪一个函数，编译器报错。

////////// 例 CH08_J3 //////////

```

#include <stdio.h>

// 求两个数中的较大值

double find_max(double a, double b)
{
    return a > b ? a : b;
}

// 求两个数中的较大值

float find_max(float a, float b)
{
    return a > b ? a : b;
}

int main()
{
    int m1 = find_max(1, 2);
    return 0;
}

```

通常我们应该对参数进行显式类型转换，以明确使用哪个版本的重载函数。例如，find_max((double) 1, (double) 2)则能精确匹配到 find_max(double, double)这个函数。

注：在原来的 C 语言中，不允许函数名称重复。具体请参考本书附录《C++和 C 的区别》。

1.62 *参数的默认值

(*初学者可以跳过本节)

在 C++中，引入了参数默认值的语法。此语法在 C 语言中不支持。

在函数的声明或定义之处，将参变量指其默认值。在函数调用时，如果指定了这个参数，则使用实际传入的参数值。或未指定，则使用默认值。

在例 CH08_K1 中，函数 Show 的第 3 个参数具有默认值 1。所以，调用 Show(400,300)时没有指定第 3 个参数，则第 3 个参数取默认值 1。

////////// 例 CH08_K1 //////////

```
#include <stdio.h>
```

```
void Show ( int x, int y, int z=1) // 第 3 个参数具有默认值
```

```
{
```

```
    printf("location: (%d, %d), layer: %d \n", x, y, z);
```

```
}
```

```
int main()
```

```
{
```

```
    Show(400,300); // 未指定第 3 个参数，则第 3 个参数取默认值 1
```

```
    Show(200,300, 2); // 指定第 3 个参数
```

```
    return 0;
```

```
}
```

相关的注意事项：

（1）具有默认值的参数必须列在后面。

例如，

```
void Show ( int x, int y=1, int z=1) // 可以
```

```
void Show ( int x=1, int y, int z) // 错误！带默认值的参数应该放在后面！
```

（2）当函数的声明与定义分开时，应该把默认值写在声明里，不能写在定义里

////////// 例 CH08_K2 //////////

```
#include <stdio.h>
```

```
void Show ( int x, int y, int z=1); // 函数声明里加默认值
```

```
int main()
```

```
{
```

```
    Show(400,300); // 未指定第 3 个参数，则第 3 个参数取默认值 1
```

```
    Show(200,300, 2); // 指定第 3 个参数
```

```

        return 0;
    }

    void Show ( int x, int y, int z) // 函数定义之处：不能加默认值
    {
        printf("location: (%d, %d), layer: %d \n", x, y, z);
    }

```

1.63 *内联函数

（*初学者请跳过本节。本节地位：不重要，可以忽略）

将函数定义前添加一个 `inline` 关键字，则该函数称为内联的。将函数声明为内联的，有什么好处呢？

对于以下函数，

```

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

```

我们知道，当调用函数时要做一些基本的入栈、出栈操作，这些操作由编译器自己完成、程序员无法看见，我们将它称为函数调用的基本开销。这就是说，函数调用本身是有成本的，即使一个函数的函数体为空，但在调用它的时候仍然耗费了一定的 CPU。

假设这个这些基本开销等价于 N 行指令，函数体编译条件到 M 行指令；如果你的函数体太短，就可能会发生 $M < N$ 的情形。这相当于开了一列火车，火车上只坐了一个旅客，这是效率比较低的事情。

避免比较这种低效率事情的发生，当发现函数体较短时，将函数声明为 `inline`，此时编译器会做特定的优化：不使用函数调用机制，而是把代码逻辑直接“内联”到调用点上。这使得代码的运行效率提高，因为省去了函数调用机制。所以 `inline` 函数是形式上函数，但并不是按函数机制来调用的。

下面的代码中，将函数 `max` 修饰为 `inline` 的，

```

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

void main()

```

```
{
    int a = max (10, 13); // 函数调用的规则没变
}
```

至于什么情况叫“短”，什么情况叫“长”，这是由编译器自己衡量的。即使你将函数声明为了 `inline`，编译器也不一定会按 `inline` 编译。如果它发现你的代码实际上是很“长”的，它还是会按函数调用的方式来编译。

1.64 *函数的递归调用

(*初学者可以跳过本节。递归是算法课的重点，但并不是语法课的重点。)

在 C++ 中，一个函数在定义后可以被其他函数调用，而 `main` 函数则是第一个被执行的函数。在普通情况下，函数的调用关系可能是这样子的：

```
main() -> A() -> B() -> C()
```

实际上，还有一种特殊函数调用情形，示意如下：

```
互相调用： main() -> A() -> B() -> A() ...
```

```
调用自己： main() -> A() -> A() ..
```

当一个函数直接或者间接地调用到了自己时，称为递归调用。

递归调用常用于实现特定的算法，它可以简化算法的实现。其主要思路时，缩减问题的规模，将一个高阶的问题转化为低阶相同问题的调用。

例：已知有一列数： 1, 1, 2, 3, 5, 8, 13, ... （数学中的斐波那契数列）

其规律是： $F_1=1$, $F_2=1$, $F_n=F_{(n-1)}+F_{(n-2)}$ ，其中 F_n 为第 n 个数。要求写一个函数，来求第 n 项的值 F_n 。

```
////////// 例 CH08_L1 //////////
```

```
#include <stdio.h>
```

```
int Fn(int n)
```

```
{
```

```
    // 终止条件
```

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

```
        return 1;
```

```
    if(n==2)
```

```

        return 1;
    // 缩减问题规模
    return Fn(n-1) + Fn(n-2);
}

int main()
{
    for(int i=1; i<= 10; i++)
    {
        printf("%d ", Fn(i));
    }
    return 0;
}

```

其中，求 F_n 是一个 N 阶问题，在算法中转化为对第 $N-1$ 阶和 $N-2$ 阶相同的问题的调用。

递归算法的特点：

(1) 将高阶问题降为低阶相同问题

如， `return Fn(n-1) + Fn(n-2);`

(2) 必须设置终止条件，避免无限制递归

如， `if(n==1) return 1;` 当 n 为 1 是最低阶，结束递归

(3) 可以替换为非递归算法，改用循环语法来实现

所有的递归算法，都可以改用 `for` 或 `while` 的语法来实现。

(4) 注意控制递归的深度

递归调用的嵌套次数不能太多，否则可能会耗尽相关系统资源，导致程序崩溃。

在示例 CH08_L2 中，展示了如何用非递归算法来求得斐波那契数列的第 n 项的值

////////// 例 CH08_L2 //////////

```

int Fn (int n)
{
    if(n==1) return 1;

```

```
if(n==2) return 1;

int an_2 = 1;
int an_1 = 1;
int an = 0;
for(int i=3; i<=n ; i++)
{
    an = an_2 + an_1; //  $A(n) = A(n-2) + A(n-1)$ 
    an_2 = an_1;
    an_1 = an;
}
return an;
}
```

第9章 指针

指针类型用于表示内存地址，本质上是一个整数。对于指针类型的变量，可以使用星号操作，对指定的内存地址进行读写。

1.65 内存地址的表示

我们知道，变量和内存是对应的，每个变量都对应于若干字节的内存。当读取一个变量的值时，其实就是读取内存的值。当修改一个变量的时候，其实就是修改了内存的值。

例如，下面定义的变量 `a` 在内存中对应 4 个字节：

```
unsigned int a = 0xA0A0A0A0; // a 内存中的值： A0 A0 A0 A0
```

```
a = 0xB1B1B1B1; // a 内存中的值： B1 B1 B1 B1
```

我们把变量所对应的内存的地址，简称为变量的地址。变量的地址是一个整数，可以用操作符 `&` 来取得。例如，

```
int a = 0;
```

```
double b = 0;
```

```
printf("%08X \n", &a); // 把地址按十六进制打印
```

```
printf("%08X \n", &b); // 把地址按十六进制打印
```

变量的地址是一个整数，理论上可以用 `int` 或 `unsigned int` 表示。但在 C/C++ 中，为了突出强调它是一个内存地址，提出了一些新的整数类型，即指针类型。

1.66 指针的概念

在 C/C++ 里，提出专门的类型来表示变量的地址：

`char*`：表示一个 `char` 型变量的地址

`short*`：表示一个 `short` 型变量的地址

`int*`：表示一个 `int` 型变量的地址

`float*`：表示一个 `float` 型变量的地址

`double*`：表示一个 `double` 型变量的地址

`unsigned char*`：表示一个 `unsigned char` 型变量的地址

... ..

总之，使用 `XXX*` 来表示 `XXX` 型变量的地址。其中 `XXX` 可以为已知的任何数据类型，如 `char`, `int`, `double` 等等。要注意的是，`XXX*` 是作为一个整体使用的，星号是这个类型名的一部分。

我们把这种带星号的类型，称为指针类型。

由这种类型定义的变量，称为指针类型的变量，简称指针变量、或称为指针。

注：指针，Pointer，意思是 Point to an address（指向一个地址）。

例如，

```
int a = 1;
int* p = &a;
```

这里定义了一个 `int*` 型的变量 `p`，其值为变量 `a` 的地址。在口语中，我们可以读作“指针 `p` 指向了变量 `a` 的地址”。

（1）指针变量也是变量

即是变量，那么就是可以变的。

例如，

////////// 例 CH09_A1 //////////

```
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 11;
    int* p = &a; // p 指向变量 a 的地址
    p = &b;      // 修改 p 的值，现在指向了变量 b 的地址
    return 0;
}
```

（2）不同类型的指针，不能互相赋值

在 C/C++ 中，虽然把带星号的类型统称为指针类型，但准确地说 `XXX*` 和 `YYY*` 是两种不同的类型。例如，`char*` 的全称是“`char` 型指针类型”，而 `int*` 是“`int` 型指针类型”。它们是完全不同的类型，之间不能通用。

例如，下面的代码中演示了两个类型不同的指针，之间是不能赋值的。

```
int a = 10;
```

```
int* pa = &a; // pa: int*型
```

```
double* pb = pa; // 错！！ 左侧 double*, 右侧 int*, 类型不同。
```

这是因为，int*用于指向 int 型变量，double*用于指向 double 型变量，类型不同，自然是不能互相赋值的。

(3) 指针是一个整数类型

在用 printf 打印时，通常使用的格式符为%p，其中 p 代表 pointer。

```
int m = 1;
```

```
int* addr = &m;
```

```
printf("addr: %p\n", addr);
```

(4) 星号的位置比较自由

在书写指针类型时，星号的位置相对自由。

```
int*   a; // int 与*连着写
```

```
int *  a; // int 与*中间有空白
```

```
int   *a; // *与变量名连着写
```

哪一种写法都是可以的，没有区别。

(5) 同类型变量的混合定义

有一种混合定义的写法，例如

```
int a, *p, m, *p2; // 定义了 4 个变量，a 是 int，p 是 int*，m 是 int，p2 是 int*
```

```
int b, *p3 = &b; // 定义 2 个变量：b 是 int，p3 是 int*并设置初始值为&b
```

不过，这两种写法都是不推荐的。本书推荐的写法是，每行只定义一个变量，绝不使用混合定义。

```
int b;
```

```
int* p3 = &b; // 推荐写法：每行只定义一个变量
```


1.67 星号操作

前面我们已经掌握了一种修改内存的办法：变量赋值。修改变量的值，就是修改了变量所在内存的值。

在引入指针之后，我们可以使用另外一种办法来修改内存值：星号操作。可以使用星号操作符 * 作用于指针变量上，来直接读写内存的值。

例如，

////////// 例 CH09_B1 //////////

```
int main()
{
    int a = 0;

    int* p = &a; // p 指向 a

    *p = 0xA0A0A0A0; // 写操作：修改 p 指向的内存的值
    *p = 0x13141516;  // 写操作：修改 p 指向的内存的值

    int b = *p; // 读操作：读取 p 指向的内存的值
    int c = *p + 2; // 读操作：读取 p 指向的内存的值

    return 0;
}
```

星号操作是一种按地址访问的技术，只有知道了这块内存的地址，就可以直接修改这块内存的值。

内存操作分为两类：读操作，写操作。当 *p 作为左值出现被赋值时，就是写操作。当 *p 作为右值时，就是读操作。

(1) 只有指针类型才知道星号操作

普通的整数类型，比如 int 型，是不支持星号操作的。例如：

```
int addr = 0x12345678; // addr: 普通 int 型变量
*addr = 0; // 编译错误！ 只有指针才支持星号操作！
```

(2) 其他指针类型的用法是一样的

在本章中，主要使用 int* 来做语法讲解，在实际使用中，其他的指针类型如 char*, double* 它们的用法是完全一样的。例如，

```

char a = 78;

char* p = &a;

*p = 79;

double m = 1.1;

double* pm = &m;

*pm = 1.2

```

(3) 区分星号的上下文

在定义一个变量时，星号表示指针类型。例如 `int* p`，其中 `int*` 作为一个整体出现，表示变量的类型。

在星号操作里，星号表示读写内存的值。例如 `*p = 123`。

所以说，星号到底是什么意思，取决于它出现的位置（上下文）：它是在定义一个指针变量，还是在进行星号操作。

```

////////// 例 CH09_B2 //////////

#include <stdio.h>

int main()
{
    int a = 10;

    int b = 11;

    int* p; // 定义一个指针, 注意 int* 作为一个整体

    p = &a; // p 指向 a

    printf("%d \n", *p); // *p: 读内存的值

    p = &b; // p 指向 b

    printf("%d \n", *p); // *p: 读内存的值

    return 0;
}

```

1.68 指针与数组

我们知道，数组在内存就相当于一串紧密排列的变量。实际上，数组名代表的就是这一块内存的首地址。

例如，在示例 CH09_C1 中，数组 `arr` 对应了一块长为 16 字节的内存，而数组名本身就表示了这个内存的地址。我们可以用 `%p` 把它打印显示出来。

//////////例 CH09_C1//////////

```
#include <stdio.h>

int main()
{
    int arr[4]; // 在内存中表现为 4 个并排的 int

    printf("address: %p \n", arr); // 数组名就是内存地址

    return 0;
}
```

对于一个 int 型数组来说，其地址类型为 int*，这意味着数组名本身的类型是 int*。我们可以用一个指针变量来存放数组的首地址。在示例 CH09_C2 中，指针 p 和 p2 都指向了数组的第一个元素的地址。

//////////例 CH09_C2//////////

```
#include <stdio.h>

int main()
{
    int arr[4] = {1,2,3,4};

    int* p = arr;    // arr 本身的类型就是 int*

    int* p2 = &arr[0]; // 第一个元素的地址

    return 0;
}
```

1.68.1 指针加减法

指针变量的加减法，和普通整型的加减法，其含义是不一样的。当指针加减时，是以元素为单位进行移动。例如，p+1 表示从移到后一个元素，p-1 表示移到前一个元素。

////////// 例 CH09_C3 //////////

```
#include <stdio.h>

int main()
{
    int arr[4] = {0,1,2,3};

    int* p = arr; // 指向 arr[0]

    p += 2;    // 指向 arr[2]

    printf("%d \n", *p); // 打印输出为 2
}
```

```

    p -= 1; // 指向 arr[1]

    printf("%d \n", *p); // 打印输出为 1

    return 0;
}

```

1.68.2 指针与数组的转换

指针和数组之间的操作是极为灵活的，可以互相转换。设有指针 `int* p` 和数组 `int arr[4]`，下面演示了它们之间的转换。

（1）p 指向 arr 的任意一个元素

第一种方法：

```
p = arr + 3; // 指向 arr[3]
```

第二种方法：

```
p = &arr[3]; // 指向 arr[3]
```

（2）给数组元素赋值

第一种方法：

```
arr[3] = 10;
```

第二种方法：

```
* (arr + 3) = 10;
```

第三种方式：

```
int* p = arr + 3;
```

```
*p = 10;
```

（3）把 p 当成数组使用

```
int* p = &arr[1]; // p 指向 arr[1]
```

```
p[0] = 0xAA; // p[0]: 自 p 开始的第 0 号元素, 即 arr[1]
```

```
p[1] = 0xBB; // p[1]: 即 arr[2]
```

（4）长度为 1 的数组

普通变量 `int a` 可以视为长度为 1 的数组来操作。

```
int a = 10;
```

```
int* p = &a;
p[0] = 11; // 长度为 1 的数组
```

(5) 数组的遍历

第一种方法:

```
int arr[4] = {0,1,2,3};
for(int i=0; i<4; i++)
{
    printf("%d \n", arr[i]);
}
```

第二种方法: 用指针遍历, 注意终止条件判断为 $p < arr + 4$

```
int arr[4] = {0,1,2,3};
for(int* p= arr; p<arr+4; p++) // 注: arr+4
{
    printf("%d \n", *p);
}
```

1.69 指针作为函数的参数

指针类型可以作为参数传递。在函数里, 可以使用这个指针, 可以访问目标内存的值。

1.69.1 例子

```
//////////例 CH09_D1 //////////
#include <stdio.h>
void test(int* p) // 把一个内存地址传给一个函数
{
    *p += 1;
}
int main()
{
    int a = 0;
    test(&a); // a 的值变为 1
```

```

    printf("a : %d \n", a);
    return 0;
}

```

在 main 函数里，将变量 a 的地址传给了函数；在进入函数 test 里面执行时，相当于对参数变量作了一个初始化：

```
int * p = & a;
```

接着，在 test 函数里使得星号操作修改了内存的值。此内存对应了变量 a，所以自然地变量 a 的值也就被修改了。

1.69.2 例子

```

//////////////////例 CH09_D2 ////////////////////
#include <stdio.h>
void sum (int* a, int* b, int* out)
{
    int result = *a + *b; // 读取输入
    *out = result; // out 保存输出, 不使用返回值
}
int main()
{
    int a=10, b=11;
    int out = 0;
    sum(&a, &b, &out);
    return 0;
}

```

在这个例子里，函数 sum 用于求 3 个数的和。不同于以前的做法，这里全部用指针作为参数。把返回值设为 void，并用一个参数 out 来保存返回值。

1.69.3 例子

```

////////////////// 例 CH09_D3 ////////////////////
void swap(int* a, int* b)
{
    int t = *a;

```

```

    *a = *b;

    *b = t;
}

int main()
{
    int a=10, b=11;

    swap(&a, &b);

    return 0;
}

```

在这个例子中，经过 swap 函数的处理，变量 a,b 的值被交换。

1.69.4 输入参数与输出参数

对于函数来说，用于保存输出值的参数叫做输出参数。例如，前面第 2 例中第三个参数 out，其作用是保存输出的值，所以它是输出参数。其他的参数，如果仅用于输入信息，那么就是输入参数。

输出参数通常使用指针来实现。它是很有用的一个特性，有很多时候必须使用指针才能完成相同的功能。举例来说，如果一个函数有多个返回值，由于函数只能设一个返回值，那么更多的返回值只能通过输出参数来实现。

例如，完成一个函数，求两个数的平方和、立方和。

////////// 例 CH09_D4 //////////

```

#include <stdio.h>

void compute(int a, int b, int* ss, int* sc)
{
    *ss = a*a + b*b;

    *sc = a*a*a + b*b*b;
}

int main()
{
    int ss, sc;

    compute(3,4, &ss, &sc);

    return 0;
}

```

特别地，当一个参数既用于输入、又用于输出，就称之为“输入输出参数”。

1.70 数组作用函数的参数

如何将一个数组的内容传递给参数呢？由于数组名就是地址，因此，我们只要把数组的地址和长度传递给函数即可。

在例 CH09_E1 中，函数 avg 用于求一个数组的各个元素的平均值。示例代码如下：

////////// 例 CH09_E1 //////////

```
int avg(int* p, int len)
{
    int sum = 0;
    for(int i=0; i<len; i++)
    {
        sum += p[i];
    }
    return sum/len;
}
```

在调用时，只需要把数组的首地址和长度传进去，就可以实现其平均值的计算了。

```
int main()
{
    int arr[] = {0, 1, 2, 3};
    int ret ;
    ret = avg(arr, 4); //从 arr[0]到 arr[3]
    ret = avg(arr, 3); // arr[0] .. arr[2]
    ret = avg(arr + 1, 3); // arr[1] .. arr[3]
    return 0;
}
```

显然，对于函数 avg 来说，它没法区分你传进来的是个数组的第 1 个元素，还是数组的第 2 个元素的地址。对它来说，只需要一个地址，一个长度。因此，我们可以 avg(arr, 4)求的是 a[0] a[1] a[2] a[3]的平均值，而 avg(arr + 1, 3)则是求 a[1] a[2] a[3]的平均值。

注意事项：

(1) 以下两种方法完全等价,没有任何区别。

```
int avg(int* p, int len)
```

和

```
int avg(int p[], int len);
```

```
int avg (int p[4], int len); // 中括号内的数字会被忽略, 指定 4 或 40 都是一样的, 没有用
```

(2) 传递数组时, 总是要另外传递长度信息

不能只把首地址传给函数, 这样是不够的。

```
int avg(int* p, int len); // 总是要传长度信息
```

1.71 指针作为函数的返回值

函数的返回值也可以指针类型。例如, 在下面的例子中, `get()`函数的返回值是一个指针, 这个指针指向了一个全局变量。

```
////////// 例 CH09_F1 //////////
```

```
#include <stdio.h>
```

```
int number = 1;
```

```
int* get()
```

```
{
```

```
    return &number; // 返回一个全局变量的指针
```

```
}
```

```
int main()
```

```
{
```

```
    int* p = get(); // 使用 get 函数的返回值
```

```
    *p = 12;
```

```
    return 0;
```

```
}
```

在把指针作为函数的返回值时, 要注意保证指针指向的变量的有效性。在上例中, 指针指向了一个全局变量, 由于全局变量的生命期是永恒的, 所以返回这样的指针没有问题。

但是, 如果你把一个局部变量的地址返回, 那么会有问题。

```
////////// 例 CH09_F2 //////////
```

```

#include <stdio.h>

int* get()
{
    int number = 1;
    return &number; // 返回一个局部变量的指针
}

int main()
{
    int* p = get();
    *p = 12; // 错误! p 指向的变量已经失效, 所以不能用星号操作!
    return 0;
}

```

在这里, 我们只需要知道从语法上说, 指针是可以作为返回值的。知道这点就够了。至少在什么工程背景下才会把指针作为返回值, 是我们在后续的章节中要说的。

1.72 const 指针

在普通指针类型前面加上关键字 `const` 修饰, 就得到一个新的类型, 称为 `const` 指针类型。相应的变量称为 `const` 指针变量, 简称 `const` 指针。

例如:

```

int a = 0;
const int* p = &a; // p 为 const 指针

```

加不加 `const`, 有什么差异呢?

(1) 不加 `const`

```

int a = 10;
int* p = &a;
*p = 11; // 内存可写
int b = *p; // 内存可读

```

(2) 加上 `const` 修饰

```
int a = 10;
const int* p = &a;
*p = 11; // 错误!内存不可写!
int b = *p; // 内存可读
```

简而方，`const`的作用是封禁了星号操作里的写内存功能。这意味着，使用 `const` 指针只能读取内存的值，而不能写内存的值。

`const` 指针的常用于限定函数的参数。例如，

```
int test(const int* p, int len)
{
}
```

把参数指针限定为 `const`，是用于显示地指定：“该函数是输入参数，在函数里只是读取内存的值、而不会修改这个内存的值”。

注意事项：

(1) 注意区分星号操作和指针算术操作

`const` 封禁的星号操作，只是星号操作受到影响。而普通的算术操作，如指针赋值、指针加减法是不受影响的。例如，

```
int avg (const int* p, int len)
{
    for(int i=0; i<len; i++)
    {
        printf("%d \n", *p); // 星号操作: 可以读
        p = p + 1; // 指针加减法没问题
    }
}
```

(2) 另一个不常见的语法

有另外一个知识点，被经常见于各种考试，但在实际工程中却不会用到。

```
int a;
int b;
int* const p = &a; // 这是另一种语法
```

```
p = &b; // 语法错误: p 不能修改
```

显然地, 这种生僻语法也许不知道会更好一些。对我们最有用、也是极为常见的, 则本节介绍的 `const` 指针的语法。

1.73 void* 型指针

`void*`型指针仅表示一个内存地址, 它不指明内存里存放的是何种类型的数据。

`void*`型指针不支持加减法。由于 `void*` 仅仅表示指针一块内存, 这一块内存存储未知类型的数据。因此, `void*` 也不支持星号操作。实际应用中, `void*` 几乎等同于 `int`, 只是用于表明其是一个指针。

例如,

```
void* p = &a; // 可以, 没有问题。
```

```
p += 1; // 编译错误: p 没有类型, 不支持加减法
```

```
*p = 1; // 编译错误: 不支持星号操作, 编译器不知道目标单元的大小和类型
```

`void*`可以和其他指针类型转换,

```
int* p1 = 0;
```

```
void* p2 = p1; // int* => void*
```

```
p1 = (int*) p2; // void* => int*, 需要强制转换
```

1.74 安全地使用指针

指针用于直接操作内存, 对程序员来说, 它既是一个强大有力的工具, 同时也是危险的根源。如果不掌握指针的安全使用, 那么程序的崩溃就是一件很容易碰见的事情。

我们先看一个例子, 看看程序的崩溃是多么容易:

```
////////// 例 CH09_G1 //////////
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int * p;
```

```
    printf("%d \n", *p);
```

```
    return 0;
```

```
}
```

这个例子虽然看起来很荒唐，但却是很多初学者的亲身经历。运行一下这个程序，可以发现程序崩溃也。也许我们还不清楚它为何崩溃，但它至少说明了一件事：“指针有风险，使用须谨慎”。

安全地使用指针，需要有对指针足够清楚的认识。其实安全使用指针的核心原则，就是要弄清楚这两个问题：

- ① 这个指针指向了哪儿？
- ② 这个指针指向的那块内存是否可用？

只要我们能清楚地回答这两个问题，那程序的安全性就有了保证。为了便于读者的理解，本文列出常见的情况加以说明。

1.74.1 指针指向了哪里

就目前我们学到的知识来说，指针只可能指向两个地方：

- (1) 指向了变量或数组
- (2) 指向了 0，即空指针

举例来说，

第一种情况：

```
int n;  
int* p1 = &n; // p1 指向的内存：一个变量  
int arr[4];  
int* p2 = arr; // p2 指向的内存：一个数组
```

第二种情况：

```
int* p = 0; // 空指针
```

也就是说，一个指针要么指向变量和数组，要么指向 0，不允许有第三种情况发生。这可以解释为什么先前的程序会崩溃。

```
int * p; // 这个指针指向哪了？一个莫名其妙的地方。。  
printf("%d \n", *p); // 立即崩溃：既然不知道它指向哪里，怎能操作那一块内存？
```

当一个指针未赋值时，其值是一个不确定的值，也就是指向了一个不确定的内存地址。把这样的未赋值的指针称为野指针（Wild Pointer）。

1.74.2 空指针的使用

值为 0 的指针，称为空指针。

```
int* p = 0;
```

当指针为空时，不能使用星号操作，否则程序会崩溃。

```
int* p = 0;
```

```
printf("%d \n", *p); // 空指针不能用星号操作，目标内存不允许访问
```

空指针是程序员可以接受的一种情况，只需加一个 if 判断就能解决！

```
if(p) // 用 if 语句判断指针是否为空
```

```
{
```

```
    printf("%d \n", *p);
```

```
}
```

1.74.3 安全使用规范

(1) 杜绝野指针

当一个指针初始化时，要么指向一个变量或数组，要么初始化为 0。如果你不确定让它指向哪，那就把它设为 0。

在程序中无法检测野指针，但空指针却可以很容易的检测。例如，在 test 函数，在使用指针变量 p 之前，检查它是否为空指针，示范代码如下：

```
void test(int* p)
```

```
{
```

```
    // 在使用指针前判断
```

```
    if( p != 0)
```

```
    {
```

```
        printf("%d \n", *p);
```

```
    }
```

```
}
```

(2) 严防数组越界

当指针指向数组时，要注意不要越界访问。在下例中，数组的长度为 4，并用一个指针来指向数据的元素。p+=4 后，p 已经越出数组的有效范围，此时对 p 的星号操作就是越界访问。

////////// 例 CH09_G2 //////////

```
#include <stdio.h>

int main()
{
    int arr[4];
    int* p = arr;
    p += 4; // p 已经超界
    *p = 12; // 越界访问!
    return 0;
}
```

(3) 目标内存是否已经失效

我们用指针指向一个变量，如果目标变量已经失效（超出作用域 / 生命期结束），那这个指针就不允许继续使用。

例如，在 CH09_G3 中，get()函数返回的是一个局部变量的地址，当 get()返回后，局部变量 number 的生命期结束。这意味着指针 p 指向了一个无效的变量，因而是一种错误的用法。

////////// CH09_G3 //////////

```
#include <stdio.h>

int* get()
{
    int number = 123;
    return &number;
}

int main()
{
    int* p = get();
    printf("value: %d \n", *p);
    return 0;
}
```

又如，在 CH09_G4 中，指针 p 指向了一个局部变量 a，在程序运行至 printf 行时指针 p 所指向的变量 a 已然失效，所以此时 p 的星号操作是错误的。

//////////例 CH09_G4 //////////

```

#include <stdio.h>

int main()
{
    int* p = 0;
    if(1)
    {
        int a = 123; // a 生命期开始

        p = &a; // 指向 a

    } // a 生命期结束

    printf("value: %d \n", *p); // 目标对象已经失效！不可以星号操作！

    return 0;
}

```

再例如，在例 CH09_G5 中，函数 `get` 返回的指针指向了 `arr[1]`，在对 `p` 进行星号操作的地方，目标对象是有效的，所以没有问题。

```

//////////////////////例 CH09_G5 ////////////////////////

#include <stdio.h>

int* get(int* arr, int i)
{
    return &arr[i]; // 返回数组 arr 的第 i 个元素的指针
}

int main()
{
    int arr[4] = {0, 1, 2, 3};
    int* p = get(arr, 1);
    *p = 22; // p 指向的位置是 arr[1]，arr 当前有效

    printf("value: %d \n", *p);

    return 0;
}

```

综上所述，安全使用指针的核心问题，就是要是确定指针指向的目标内存/目标对象是有效的。不要指向一个不确定的地方，也不要指向一个已经失效的对象。

1.75 *二重指针

(*初学者可以跳过本节，二重指针及多重指针不是重点)

指针也是变量，凡是变量就是有地址。

所以，`int**`类型就是 `int*`型变量的地址。

```
int a = 10;

int* p = &a; // p 指向了变量 a

int**pp = &p; // pp 指向了变量 p
```

我们将带两个星号的类型称为二重指针，将带 `N` 个星号的类型称为 `N` 重指针。但实际上，无论是二重指针还是多重指针，其可读性都是比较差的，在程序中应该尽量避免使用。

下面举一个例子，来示例二重指针的使用。

////////// 例 CH09_H1 //////////

```
#include <stdio.h>

int number = 0;

void set(int** p)
{
    *p = &number;
}

int main()
{
    int* p = NULL;

    set(&p);

    *p = 123; // p 指向了 number

    return 0;
}
```

对于我们最重要的、也是最经常被使用的，还是一维指针的用法。

1.76 *二维数组与指针

(*初学者可以跳过本节，二维数组本身就不是重点，与指针结合更为罕见)

定义了一个二维数组，如何传递给一个函数？对于二维数组来说，其数组名也是首地址，我们可以使用和一维数组类似的方法，来传递二维数组给函数。

1.76.1 作为函数的参数

例 CH09_K1 展示了如何将二维数组作为参数传递，在定义参数时，注意要指定第二维的大小（即列的大小）。

```
////////// 例 CH09_K1 //////////  
  
#include <stdio.h>  
  
void test( int data[][4] , int rows) // 注意：要指定列值  
{  
    for(int i=0; i<rows; i++)  
    {  
        for(int j=0; j<4; j++)  
            printf("%d , ", data[i][j]);  
    }  
}  
  
int main()  
{  
    int arr[2][4] =  
    {  
        { 1, 2, 3, 4 },  
        { 11, 12, 13, 14 },  
    };  
    test(arr, 2);  
    return 0;  
}
```

注意：不能写成 `void test(int data[][] , int rows)`。

还有另外一种写法，在本质上与上一种写法相同，但可读性上则显得稍难理解。

```
void test2(int (*p)[4], int rows)  
{  
    for(int i=0; i<rows; i++)  
    {  
        for(int j=0; j<4; j++) printf("%d , ", p[i][j]);  
    }  
}
```

```
}
```

1.76.2 二维数组与指针的转换

可以和一种指针类型，来遍历二维数组的每一行。

```
int aa[6][4]; // 6 行，4 列
```

```
int (*row0)[4] = aa; // row0: 指向第 0 行
```

```
int (*row1)[4] = aa; // row1: 指向第 1 行
```

```
(*row0)[0] = 1; // 第一种使用方法：指针访问法：访问第 0 行第 0 列
```

```
row0[0][1] = 2; // 第二种使用方法：像二维数组一样访问：访问第 0 行第 1 列
```

```
row1[0][0] = 3; // 访问第 1 行第 0 列
```

简单解释一下，

```
int (*row0)[4] = aa;
```

这是定义了一个变量，变量名为 `row0`，初始值为 `aa`，即指向数组的第一行。这个变量的类型则比较复杂，是一个列值为 4 的数组类型。

显然，这种指针的定义法过于复杂，如果要扩展到三维数组，更是没法描述。实际上，在任何场合要，都要避免这种复杂的写法。要知道，一个程序失去了可读性，那就无所谓质量可言了。

第 10 章 结构体

结构体是将多种基本类型组合起来，组建一个自定义的类型。

1.77 引例

下面我们将展示一个问题，通过这个问题来说明为什么需要结构体这种语法。

在下面的表格记录了多个联系人的相关信息，每一个联系人都有以下字段：ID，姓名，手机号。（可以把 ID 理解为一个数字编号）。如表 10-1 表示。

表 10-1 手机相关信息

ID	姓名	手机号
201501	John	18601011223
201502	Jennifer	13810022334
201503	AnXin	18600100100
201504	Unnamed	13111011011

基于我们目前已经学过的知识，我们可以使用多个数组来表达这个表格里的数据：

////////// 例 CH10_A1 //////////

```
int id[4] =
{
    201501, 201502, 201503, 201504
};
char name[4][16] =
{
    "John", "Jennifer", "AnXin", "Unnamed"
};
char phone[4][16] =
{
    "18601011223", "13810022334", "18600100100", "13111011011"
};
我们可以把所有数据显示出来，
void DisplayAll()
{
```

```

for(int i=0; i<4; i++)
{
    printf("%d \t%s\t%s\n", id[i], name[i], phone[i]);
}
}

```

这样是可以完成数据的表示的，但也有不少缺点和问题。

(1) 信息比较松散，不直观。在这里用了 3 个分散的数组。

(2) 容易重名。如果还有另外一个表格，也有 ID，姓名和手机号，那变量名的命名就会有重复。

(3) 数据存取不方便。如何不方便呢，下面说明这个问题。

比如，现在要按 ID 来查找一个联系人，将联系人的信息返回。我们用一个函数 Find 来实现这个查找功能：

```

int Find(int id, char name[], char phone[])
{
    for(int i=0; i<4; i++)
    {
        if(ids[i] == id)
        {
            // 复制姓名和手机号
            strcpy(name, names[i]);
            strcpy(phone, phones[i]);
            return i;
        }
    }
    return -1;
}

int main()
{
    char name[16];
    char phone[16];
    Find(201502, name, phone);
}

```

```
    return 0;
}
```

在 `Find` 函数里，定义多个输出参数，将需要的字段存到输出字段里。现在扩展一下问题，如果表格中有多达几十个字段：年龄，单位，住址，`email`，QQ 号等等，难道要定义几十个输出参数才行吗？

所以，我们希望存在一个自定义的类型 `Contact`，它里面包含了所有的字段信息。然后，我们可以像下面这样使用它：

```
Contact persons[4]; // Contact: 一种新的类型
```

```
Contact Find(int id); // 直接返回一个 Contact 对象
```

或

```
int find(int id, Contact* p); // 使用一个输出参数
```

这样的语法才是简洁直观的！本章所提出的“结构体”的语法，正是要解决这个问题。

1.78 结构体

在 C/C++ 中，存在一些基本类型：`char/short/int`，`float/double` 等，但它们还不够用。

我们可以将基本类型组合起来，形成一个新的自定义类型。这种类型称为结构体类型。例如，

```
struct Contact
{
    int id;
    char name[16];
    char phone[16];
};
```

这样，就定义一个自定义类型 `Contact`，它组合了一个 `int` 型数据，两个 `char` 数组。这种由关键字 `struct` 引导的语法称为“结构体”（`structure`）。

`struct` 语法的基本形式为：

```
struct TypeName
{
    Members
};
```

其中，`TypeName` 是新的类型的名字，`Members` 是成员变量的列表。

注意：大括号末尾一定要加上分号。

1.79 结构体的基本用法

结构体类型和基本类型在使用起来是非常相似的。

1.79.1 变量定义和初始化

可以使用结构体类型作为类型名，来定义一个变量。

(1) 定义一个变量

```
////////// CH10_B1 //////////
```

```
    Contact a; // 定义一个变量，不指定初始值
```

(2) 定义并指定初始值

和数组的初始值语法类似，可以用大括号将各个成员的初始值列出。如果有多个成员，则需要顺序一致。例如，

```
    Contact b =  
    {  
        201501,    // id: 一个 int 值  
        "Jennifer", // name: 一个 char 型数组  
        "13810022334" // phone: 一个 char 型数组  
    };
```

指定初始值时，

- ①注意使用大括号，末尾要加分号结束
- ②各字段的初始值要与该字段的类型匹配
- ③各初始值以逗号分开，最后一个成员后面不需要加逗号
- ④初始值的顺序要与成员变量的顺序一致

和数组一样，可以只初始化部分成员变量，例如：

```
Contact a = { 201501, "John" }; //只初始化前 2 个成员
```

也可以直接清零，一次性将所有字段均置为 0，

```
Contact a = {0};
```

1.79.2 访问结构体的成员

可以使用点号访问结构体变量的成员，例如，

////////// 例 CH10_B1 //////////

```
Contact c;  
c.id = 201501; // 访问 c.id  
strcpy(c.name, "John"); // 访问 c.name  
strcpy(c.phone, "18601011223"); // 访问 c.phone
```

注：strcpy 函数用于拷贝字符串，使用时需要在文件中添加一行 #include <string.h>

用法示例：

```
char dst[128];  
strcpy(dst, "hello"); // 将"hello"拷贝到目标缓冲区 dst 中
```

注：大括号只能用于初始化时，不能在赋值的时候使用。以下的代码是错误的：

```
Contact b ;  
b = { 201501, "Jennifer", "13810022334" }; // 错！赋值时不能使用大括号！
```

1.79.3 定义结构体数组

可以和基本类型一样，定义数组，表示并排的多个对象。

```
Contact cs[4];  
也可在定义的同时初始化，注间最后一个成员后面不需要加逗号  
Contact cs[4] =  
{  
    {201501, "John", "18601011223" },  
    {201502, "Jennifer", "13810022334" },  
    {201503, "AnXin", "18600100100" },  
    {201504, "Unnamed", "18601011223" }  
};
```

同样的，数组的元素类型是结构体，所以我们可以先用中括号[]取得元素，然后再用点号访问元素内的成员。例如，


```

for(int i=0; i<4; i++)
{
    printf("Id: %d, Name: %s, Phone: %s \n",
        cs[i].id, // 第 i 成元素的 id 的值
        cs[i].name, // 第 i 成元素的 name 的值
        cs[i].phone); // 第 i 成元素的 phone 的值
}

```

1.80 结构体的内存视图

结构体类型就是将多种基本类型组合起来，因而从内存视图看来，一个结构体变量在内存里也就相当于多个基本类型变量的组合。

1.80.1 结构体的成员

我们定义一个名为 Object 的结构体，内部添加两个成员 id 和 name，如下所示：

////////// 例 CH10_C1 //////////

```

struct Object
{
    int id;
    char name[8];
};

```

定义一个 Object 类型的变量，

```
Object obj = { 0x12345678, "shaofa" };
```

那么在内存中，obj 占据了 12 字节的内存，前 4 个字节是 id，后 8 个字节是 name。下面的表格中，示意了每一字节的值（十六进制）：

78	56	34	12	73	68	61	6f	66	61	00	00
----	----	----	----	----	----	----	----	----	----	----	----

注：在 VC 的调试窗口（“内存”窗口）中，可以直接观测&obj 处的内存的值，如图 9-1 所示。我们注意方框内的 12 个字节的值，正是变量 obj 占据的 12 个字节。

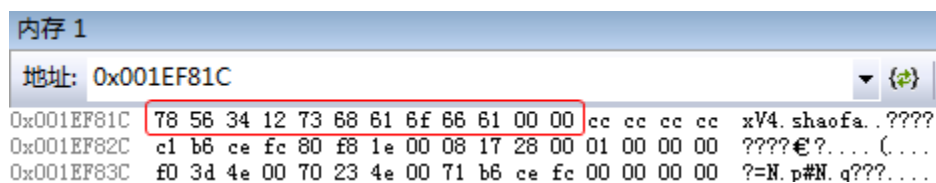


图 9-1 在内存窗口中观察结构体的值

1.80.2 对齐与填充

结构体的大小跟各个成员的体积有关系，但有时会略大于各成员的体积之和。例如，对于例 CH10_B2 中的结构体 Object 类型，它有两个成员分别为 char 和 int 型。一个 char 占据 1 个字节，一个 int 占据 4 个字节，那是不是说 Object 的大小就是 5 字节呢？

////////// 例 CH10_C2 //////////

```
struct Object
```

```
{
```

```
    char a;
```

```
    int b;
```

```
};
```

我们用 sizeof 检测一下，可以发现，该类型的大小实际上是 8 字节的。

```
Object obj = { 1, 0x55555555 };
```

```
printf("size: %d \n", sizeof(obj));
```

在 VC 的“内存”调试窗口中，观察 obj 的值，如图 9-2 所示。

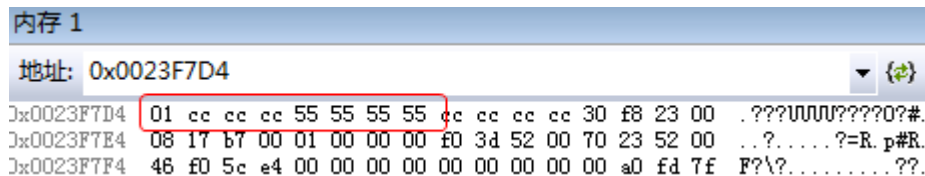


图 9-2 在内存窗口中观察结构体的值

其中，01 代表 obj.a，55 55 55 55 代表的是 obj.b，那在 obj.a 和 obj.b 之间还多出了一些字节，这些字节称为填充（padding）。

为什么会存在填充字节呢？

虽然在语法中没有规定，但编译器通过会了为“对齐”的考虑，在结构体的成员之间填充一些字节。例如，一个 int 型成员应该放在一个 4 字节对齐的地址上，而一个 short 型成员要求其地址是 2 字节对齐的。

所谓 N 字节对齐，指的是它的内存地址能够被 N 整除。

例如，

内存地址 0x8A9891F4 是 4 字节对齐的，但不是 16 字节对齐的

内存地址 0x8A9891F3 不是 4 字节对齐的（末尾是 3）

内存地址 0x8A9891F8 是 8 字节对齐的

注：为什么需要地址对齐？这是根据 CPU 的指令要求，只能在对齐的地址上存取，属于指令集的语法要求。但并不是所有的 cpu 都有此限制。

1.81 结构体的更多用法

1.81.1 结构体的赋值

和基本类型的用法类似，结构体类型的变量也是要以可以互相赋值的，例如，

////////// 例 CH10_D1 //////////

```
Contact a = { 20141003, "AnXin", "18600100100"};
```

```
Contact b;
```

```
b = a; // 赋值
```

赋值的结果是将 a 的值拷贝给 b。

1.81.2 结构体指针

在结构体类之后加上*号，就表示相应的指针类型。例如，

```
Contact* p = &a; // 定义结构体指针
```

对于指针指向的对象，我们可以用箭头->操作符来访问对象的成员，例如，

```
p->id = 20141011; // 使用->访问对象的成员
```

```
strcpy(p->phone, "13800100100"); // 使用->访问对象的成员
```

注：也可以写成 (*p).id，但不推荐这么写，通常应该使用->操作符。

1.81.3 作为函数的参数

和基本类型一样，可以作为函数参数

////////// 例 CH10_D1 //////////

```
// 传值方式
```

```
void test(Contact a)
```

```
{
```

```
    printf("id: %d , name: %s \n", a.id, a.name);
```

```

}
// 传地址方式
void test(Contact* p)
{
    printf("id: %d , name: %s \n", p->id, p->name);
}

```

下面是函数调用的示例代码，

```

Contact a = { 20141003, "AnXin", "18600100100"};
test(a); // 传值方式
test(&a); // 传地址方式

```

1.81.4 作为函数的返回值

和基本类型一样，结构体类型也可以作为函数返回值类型，例如，

```

Contact create (int id)
{
    Contact a;
    a.id = id;
    return a; // 像基本类型一样，直接 return
};

```

1.81.5 作为结构体的成员

成员的类型也可以是结构体类型。

例如，

////////// 例 CH10_D2 //////////

```

struct Color
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};

struct Object

```

```

{
    int x, y;
    Color rgb; // 此成员是 Color 类型
};

```

可以像下面这样定义 Object 对象并指定初始值，

```

Object obj =
{
    1, // x
    2, // y
    { 0xFF, 0x00, 0x00 } // rgb
};

```

可以像下面这样访问对象 obj 的成员 rgb 下面的成员：

```

printf("obj: (%d,%d) color: #02X%02X%02X \n",
    obj.x, obj.y,
    obj.rgb.r,
    obj.rgb.g,
    obj.rgb.b);

```

1.82 *结构体的特殊写法

（*初学者可以跳过本节）

结构体还有一些特殊的写法，通常这些写法不被推荐。

1.82.1 没有类型名

可以只定义一个结构体变量，但并不为这个结构体定义一个类型名。也许此时只是想定义一个变量，由于只定义一次变量，所以不需要类型名也是可以的。

在示例 CH10_E1 中，定义了一个结构体变量 info，这个 info 没有类型名，但它是可以使用的。（注：编译器内部会给它起一个类型名）

//////////例 CH10_E1//////////

struct // 该 struct 没有命名，编译器会内部分配一个名字

```

{
    int user_id;

```

```

    char guid[128];
}info;
int main()
{
    info.user_id = 98780987;
    strcpy(info.guid, "{09f140d5-af72-44ba-a763-c861304b46f8}");
    return 0;
}

```

1.82.2 函数内部的结构体类型

结构体类型的定义允许放在函数内部，这么定义的类型就只能在函数内可见。由于 `struct` 语法的初衷是要定义一个被多处使用的自定义类型，正常情况下应该定义在函数体之外的。

```

//////////例 CH10_E2 //////////
int main()
{
    // 结构体类型也可以在函数内定义，此类型只在本函数内可见

    struct Contact
    {
        int id;
        char name[16];
        char phone[16];
    };
    Contact a = { 20141003, "AnXin", "18601011223"};
    return 0;
}

```

1.82.3 紧凑的形式

还有几种紧凑的定义形式，但这些写法都是不推荐的。例如，在示例 CH10_E3 中，定义了一个 `Object` 类型，在后面立即定义了变量 `a` 和 `p`。其中，`a` 是 `Object` 类型，`p` 是指针类型。

```

//////////例 CH10_E3 //////////
struct Object
{
    int x;

```

```

        int y;
    } a, *p;    // 在大括号末尾立即定义变量

int main()
{
    a.x = 1;
    a.y = 2;
    p = &a;
    return 0;
}

```

推荐的写法是：

```

struct Object
{
    int x;
    int y;
};

Object a; // 分开写

Object* p; // 分开写

```

1.82.4 C 语言中的 struct 定义

C++对原有的 C 语言进行了扩展和优化，其中 struct 语法就做了一些优化。在原来的 C 语言中，struct 的定义是这样子的，

```

struct Object
{
    int x;
    int y;
};

```

Object 类型的定义语法和 C++中的一样，但在 C 语言中定义结构体变量的时候，struct 关键字必须要加上。例如，

```

struct Object a; // 关键字 struct 不可以省略

struct Object* p; // 关键字 struct 不可以省略

struct Object objs[128]; // 关键字 struct 不可以省略

```

这意味着，在 C 语言中 `struct Object` 必须连在一起写才是一个类型。这个语法是有点啰嗦的。于是在 C++ 中对其进行简化，直接使用 `Object` 作为一个类型名。

1.83 结构体的命名

C/C++ 中涉及的命令规则都是一样的，结构体类型的命名也不例外，那就是“数字、字母、下划线的组合，不能以数字开头”。

但在实践中，还有额外的规范需要被遵循，这些规范是“编码规范”的组成部分。“编码规范”，是在语法规则的基础上，对我们的代码风格进行约束，使得代码更易被阅读、减少出错机率、提高工作效率。

首先，一个通行的规范是：命名要有意义，一个好的名字应该直接反映其意义。

例如，

定义一个表示学生信息的结构体时，我们可以命名为 `Student` 或 `StudentInfo`，那么别人从这个类型的名字但能够推测出其要表示的意义。如果你把它命名为 `Teacher`，就会给别人带来歧义：名字叫 `Teacher`，然而内容却表示的是学生信息？

其次，结构体的命名上，一般有两种常见形式：

(1) 纯小写，以下划线分开每个单词，在旧的 C 语言中常使用这种风格

例如， `good_job`，`large_buffer` 等。

(2) 每个单词以大写开头，在 C++ 中推荐使用这种风格

例如，`GoodJob`，`LargeBuffer`

再者，结构体内的成员变量，通常是小写，并以下划线分隔每个单词。

例如，

`count`，`name_list`，`obj_size`

1.84 传值与传地址

这一节我们介绍“传值和传地址”问题，它是 C/C++ 语言的一个核心议题，也是 C/C++ 语言区别于其他语言的一个标志。

在传送函数参数时，如果传入的是一个对象的值，称为“传值”方式；如果传入的是一个对象的地址，则称为“传地址”方式。在示例 CH10_F1 中，函数 `Test1` 和 `Test2` 的功能是一样的，都是打印出 `Object` 的内容。区别是函数 `Test1` 是传值方式，`Test2` 是传地址方式。


```

////////// 例 CH10_F1 //////////
#include <stdio.h>

struct Object
{
    int id;
    char name[256];
};

// 传值方式
void Test1(Object a)
{
    printf("id: %d, name: %s \n", a.id, a.name);
}

// 传地址方式
void Test2(Object* p)
{
    printf("id: %d, name: %s \n", p->id, p->name);
}

int main()
{
    Object obj = { 123, "shaofa" };
    Test1(obj); // 传入对象的值
    Test2(&obj); // 传入对象的地址
    return 0;
}

```

(1) 传值方式

在函数 Test1 被调用时，有两个 Object 对象，其中一个叫 obj，定义在 main 函数里。另一个叫 a，是函数 Test1 里的局部变量。

在传值调用时，将 obj 的值赋给了参变量 a，相当于如下操作：

```
Object a = obj;
```

因此说，对象 a 是对象 obj 的一个复制品，我们分析这一步操作的资源消耗：

① 从内存资源的耗费上看（空间角度），对象 a 耗费 260 个字节的内存

② 从 CPU 资源的耗费上看（时间角度），从对象 obj 到对象 a 需要复制 260 个字节

（2）传地址方式

在函数 Test2 被调用时，传入一个对象 obj 的地址，即有一个 Object* 指针来表示对象 obj 的地址。

```
Object* p = &obj;
```

我们来分析这一步操作的资源消耗：

① 从内存资源的耗费上看（空间角度），指针 p 占据了 4 个字节，所有的指针其实只是一个整数（表示内存地址）

② 从 CPU 资源的耗费上看（时间角度），不需要复制数据，只需传递一个整数地址。

通过上述比较，我们可以发现传地址方式在性能上是远远占优的，它使用了较少的资源完成了相同的事情。在内存上，传值方式使用了较少的内存；在 CPU 消耗上，传值方式所耗费的 CPU 操作也非常少。

所以，在 C/C++ 里，当一个对象的体积较大时（结构体、数组），总是使用传地址方式来传递参数。

1.85 *位字段 bitfield

（* 初学者请跳过本节，本节语法很难，但基本没有用处，可以忽略）

在 struct 中可以定义一种特殊的成员：位字段(bit-field)。它是在变量名后加冒号，并标明该字段占用了多少个位。例如，

////////// 例 CH10_G1 //////////

```
struct Object
{
    unsigned char a : 1;
    unsigned char b : 2;
    unsigned char c : 3;
};
```

位字段的使用方法和普通方法基本一致，

```
Object obj;
obj.a = 1;
obj.b = 3;
```

```
obj.c = 4;
```

唯一要注意的是它的可取值范围。对于 **unsigned** 整型来说，如果只占了 1 位，则只能取值 [0~1]，如果占了 2 位则取值 [0~3]，占了 3 位则 [0~7]，依次类推。如果赋给它一个超过范围的值，则会被截断。

注：关于 bit-field 的更多阐述参考本书附录《位字段》。

1.86 实例 1

有三台手机，相关信息在表 10-2 中所列，定义适当的结构来表示这个表格里的信息，并在屏幕上这些信息打印出来。

表 10-2 手机相关信息

型号	制造商	上市年份	参考价格
RongYao	Huawei	2012	1800
Mi4	XiaoMi	2013	1900
Note3	Samsung	2013	2600

表中的每行称为一条记录，每列称为字段，首先要定义一个合适的 **struct** 来表示这种结构化的信息，

```
////////// 例 CH10_H1 //////////
```

```
struct CellPhone
```

```
{
    char model[32]; // 型号
    char producer[32]; // 制造商
    short year; // 上市年份
    short price; // 参考价格
};
```

一共 3 行记录，所以创建一个有 3 个对象的数组，并依次对其赋值，

```
int main()
{
    CellPhone phones[3];
    strcpy(phones[0].model, "RongYao");
    strcpy(phones[0].producer, "Huawei");
    phones[0].year = 2012;
```

```

phones[0].price = 1800;

strcpy(phones[1].model, "Mi4");
strcpy(phones[1].producer, "XiaoMi");
phones[1].year = 2013;
phones[1].price = 1900;

strcpy(phones[2].model, "Note3");
strcpy(phones[2].producer, "Samsung");
phones[2].year = 2013;
phones[2].price = 2600;

for(int i=0; i<3; i++)
{
    CellPhone* p = &phones[i];
    printf("%s \t%s \t%d \t%d \n",
           p->model, p->producer, p->year, p->price);
}
return 0;
}

```

第 11 章 *联合体

(*本章介绍的是一个过时的语法，初始者请跳过)

联合体(union)，在定义形式上和 struct 差不多，但是意义很大。在实际工程中我们一般不会使用这个语法。学习本章的一个主要目的，只是为了读懂一些老旧的 C 语言代码。

1.87 概念

联合体(union)用于实现一词多义，其本意是想节省空间。在例 CH11_A1 中，定义一个 union 类型 SomeData，

```
////////// 例 CH11_A1 //////////  
  
union SomeData  
{  
    unsigned int a;  
    unsigned char b;  
    char c[10];  
};
```

虽然在形式上和 struct 语法类似，但要表达的意思完全不同。在 SomeData 中，成员 a,b,c “共享同一片内存空间”。换句话说，有一块 10 字节的内存空间，它 “既是 a，又是 b，也是 c”。这理解起来自然是比较困难的。

现在我们定义一个变量

```
SomeData v;
```

则 v 占据了 10 个如下图所示，为 10 个字节的内存空间。其中，a 对应的前 4 个字节[0..3]，b 对应的是[0]，c 对应的是[0..9]。由于未初始化，所有各内存单元的值为原始值，

x	x	x	x	x	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---

下面，给 v.a 赋值，则前 4 个字节发生变化，

```
v.a = 0x12345678;
```

0x78	0x56	0x34	0x12	x	x	x	x	x	x
------	------	------	------	---	---	---	---	---	---

再给 v.b 赋值，则第 1 个字节发生变化，

```
v.b = 0xAA;
```

0xAA	0x56	0x34	0x12	x	x	x	x	x	x
------	------	------	------	---	---	---	---	---	---

再给 v.c 赋值

```
strcpy(v.c, "LiMing");
```

'L'	'i'	'M'	'i'	'n'	'g'	0	x	x	x
-----	-----	-----	-----	-----	-----	---	---	---	---

再给 v.a 赋值，则前 4 个字节发生变化，

```
v.a = 0xDDDDDDDD;
```

0xDD	0xDD	0xDD	0xDD	'n'	'g'	0	x	x	x
------	------	------	------	-----	-----	---	---	---	---

上面是说的是写操作，反过来读操作也是一样的。v.a 是将前 4 个字节视为 unsigned int, v.b 是取第 1 个字节，v.c 是将前 10 个字节视为 char 数组。

这就是所谓的“共享空间”。它是同一份数据的不同视图。当你改变 a 的值时，同时也就改变了 b 和 c 的值，因为它们是同一块内存的不同表示。

1.88 进一步理解 union

union 的大小，就是所有成员里体积最大的那个成员 大小。比如，在 SomeData 里，成员 c 最大，占 10 个字节，所以 SomeData 的大小就是 10 个字节。

我们可以通过指针操作，直接模拟 union 的功能。

首先，定义一片缓冲区，

```
unsigned char buf[10];
```

然后在这片缓冲区上施加一定的操作。

① 把 buf 视为 unsigned int,

```
unsigned int* p = (unsigned int*) buf;
```

```
*p = 0x12345678;
```

② 把 buf 视为 unsigned char,

```
unsigned char* p = (unsigned char *) buf;
```

```
*p = 0xAA;
```

③ 把 buf 视为 char,

```
char* p = (char *) buf;
```

```
strcpy(p, "LiMing");
```

显然，指针强转语法虽然写起来稍微复杂了一些，但是理解起来却是相对容易的。

1.89 实例 1

在 C/C++ 网络编程时，底层的 API 在表示 IP 地址时会使用到一个 union 类型 IN_ADDR，现在分析一下它的定义，（为了读者理解的方便，作了适当的改编）

```
struct in_addr
{
    union
    {
        struct
        {
            u_char s_b1,s_b2,s_b3,s_b4;
        } S_un_b;

        struct
        {
            u_short s_w1,s_w2;
        } S_un_w;

        u_long S_addr;
    } S_un;
};
```

我们知道，IP 地址的一般格式是 a.b.c.d，其中每个数字都是界于 0~255 之间，例如，192.168.1.1，显然，用 4 个字节刚才才能装得下这个数据，

可以定义一个数组来表示，

```
unsigned char ip[4];
ip[0] = 192;
```

```
ip[1] = 168;
```

```
ip[2] = 1;
```

```
ip[3] = 1;
```

我们在反观 `struct in_addr` 的定义，发现它就是试图将这 4 个字节的数据表式成了 3 种形式，用户可以使用任一形式来操作 IP 地址（下面示例代码中忽略字节序的问题，仅在于讲解 `union` 的概念）：

(1) 4 字节形式:

```
struct
```

```
{
```

```
    u_char s_b1,s_b2,s_b3,s_b4;
```

```
} S_un_b;
```

使用方法,

```
in_addr addr;
```

```
addr.S_un.S_un_b.s_b1 = 0xC0; // 192
```

```
addr.S_un.S_un_b.s_b2= 0xA8; // 168
```

```
addr.S_un.S_un_b.s_b3= 0x01;
```

```
addr.S_un.S_un_b.s_b4 = 0x01;
```

(2) 2 个 short 形式

```
struct
```

```
{
```

```
    u_short s_w1,s_w2;
```

```
} S_un_w;
```

使用方法,

```
in_addr addr;
```

```
addr.S_un.S_un_w.s_w1 = 0xC0A8; // 192.168
```

```
addr.S_un.S_un_w.s_w2 = 0x0101; // 1.1
```

(3) 1 个 int 形式

```
u_long S_addr;
```

使用方法,


```
in_addr addr;
addr.S_un.S_addr = 0xC0A80101; // 192.168.1.1
```

这种应用场景可以总结为：相同数据的不同的表现形式。所操纵的数据都是 4 字节的 IP 地址，但是可以按 `unsigned char` 操作，也可以 `unsigned int` 操作。

1.90 union 的替代方案

`union` 是一个过时的语法，我们可以用下方案来替代实现。

1.90.1 方案 1

事先定义若干类型 `TypeA, TypeB, ...`，表示不同情况下的数据结构，

```
struct TypeA {};
struct TypeB {};
struct TypeC {};
```

外部定义一个变量，用于指示该数据是什么类型。用一个足够长的 `buffer` 来存储数据，

```
char type = 0;
char buf[128];
if(type == 1)
{
    TypeA* t = (TypeA*) buf;
    ...
}
if(type == 2)
{
    TypeB* t = (TypeB*) buf;
    ...
}
if(type == 3)
{
    TypeC* t = (TypeC*) buf;
```

```
...  
}
```

1.90.2 方案 2

定义 TypeA, TypeB ...时, 第一个字段固定为 char type,

```
struct TypeX  
{  
    char type; // 此字段必须放在最前面  
    .... 其他字段 ....  
};
```

这样的话, 就不必再用外部的变量来指示其数据类型, 因为它的第一个字节已经标识了其类型,

```
char buf[128];  
if(buf[0] == 1)  
{  
    TypeA* t = (TypeA*) buf;  
    ...  
}  
if(buf[0] == 2)  
{  
    TypeB* t = (TypeB*) buf;  
    ...  
}  
if(buf[0] == 3)  
{  
    TypeC* t = (TypeC*) buf;  
    ...  
}
```

第 12 章 动态分配内存

1.91 引例

我们先来看一下目前存在的问题，这些问题的解决就是要用本章所要介绍的技术。

假设我们有一个结构体 `Movie` 来保存一个视频信息，

```
struct Movie
{
    int id;
    char title[64]; // 影片
};
```

我们用一个 64 字节的数组来保存电影的名字。64 字节看起来很长，能够容纳多数电影的名字，比如"The Matrix", "The Bourne Identity", 但也有少数电影名字超出 64 字节，比如"Borat: Cultural Learnings of America for Make Benefit Glorious Nation of Kazakhstan"。

归纳起来，当用数组保存一个字符串时，有 2 个问题：

- (1) 空间的浪费：绝大多数电影的名字小于 64 字节，比如"The Matrix"只占了 10 个字节
- (2) 空间的不足：对于有些电影来说，64 字节竟然还不够用。

如何在用一块刚好够大的内存（既不多，也不少）来保存数据呢？在 C/C++中，使用动态分配内存的技术来解决此类的问题。

1.92 动态分配内存

在操作系统中存在一个内存管理器(Memory Manager，简称 MM)，它负责管理一堆闲置内存。应用程序可以向 MM 申请一块指定大小的内存，来存储自己的数据；当使用完毕后，再把内存归还（释放）给 MM。

内存的申请和释放使用两个函数来实现：`malloc/free`。

1.92.1 malloc 申请内存

应用程序调用 `malloc` 函数可以申请一块指定大小的内存，其函数原型为：

```
#include <stdlib.h>
void* malloc(int size);
```

参数：

size: 内存空间的大小，以字节为单位

返回值： 申请而来的这块内存的首地址

用法示例：

```
char* p = (char*) malloc (84); // 申请一块 84 字节的空间
```

内存管理器 MM 并不关心这一块内存的用途，所以 malloc 的返回值是 void*，仅表示内存的地址。应用程序可以用来存储任何类型的数据，例如，申请一块内存，用来存放 100 个 int 型数据。示例代码如下：

```
////////// CH12_A1 //////////  
  
int* p = (int*) malloc(100 * 4); // 申请 100*4 字节  
  
for(int i=0; i<100; i++)  
{  
  
    p[i] = i * i; // 使用这块内存  
  
}
```

需要注意的是，要申请空间的大小以字节为单位，应用程序自己负责计算一共需要多少字节。比如，如果要存储 100 个 int，那么所需空间的大小应该是 400 字节，可以 malloc(100*4) 或者 malloc(100 * sizeof(int))来指定大小。

malloc 的返回值指向了这块申请到的内存，应用程序需要把它强制转换为指定的数据类型。这块内存和数组没有本质区别，用法完全相同。根据第 9 章，数组本质上就是一块连续的内存，两者是一样的。

1.92.2 free 释放内存

```
#include <stdlib.h>  
  
void free(void* ptr);
```

在使用完毕后，应用程序应当调用 free 函数来释放内存，当内存交还给内存管理器。传入的参数就是先前用 malloc 得到的指针。

1.93 内存管理器与堆

现在我们知道，内存管理器 MM 的职责是提供内存服务。它管理的内存区域称为堆（Heap），也就是说，malloc 得到的内存的位置是在堆区。而通常所说的堆内存，也就是指通过动态分配内存技术向 MM 申请到的内存。

我们需要知道，系统中只有一个 MM，系统中的所有进程（运行着的程序）都向同一个 MM 来申请内存。堆的容量有限，并不是取之不尽的。这意味着，你申请的资源越多，别人能得到的资源就越少。极端的情况，你申请了过多的内存，别的应用程序因为无法申请到内存而崩溃。

当应用程序用 `malloc` 申请了一块内存，如果不调用 `free` 归还，那么 MM 是不会主动向应用程序讨还的。所以，应用程序应该自觉地“及时”归还内存。什么才叫及时呢？就是不需要的时候，就应该归还，不应该长期“霸占”着堆内存。

综上所述，在使用动态分配内存技术时要遵守以下原则：

(1) 尽可能少的申请内存

够用就行。不要暴殄天物，不要申请超过自己需要的内存。

(2) 尽可能快的释放

用完了应该及时释放。在某些情况下，如果暂时不需要，也应该立即释放；等需要的时候再重新申请就可以了。

也就是说，尽量可能少的占用堆内存。

1.93.1 堆的内部管理

为何 `free` 的时候只需一个首地址呢？为什么不传递长度？

实际上，MM 对借出的内存块进行标识

(p0, n0) (p1, n1) (p2, n2) ...

它内部已经保证任意两块内存不会“交叠”，即不会重叠，不会把一块内存同时借给两个应用程序使用。所以，每块内存的首地址都是不同的，在 `free` 的时候只需要指明首地址即可。

注：这么解释只是为了方便大家的理解，内存管理采用的真正数据结构可能要复杂的多。

1.93.2 内存泄露

当程序长年累月运行时，就需要格外地关注“及时 `free` 内存”的问题。如果你不及时释放内存，那么你的程序申请的内存会累积得越来越多，导致系统无内存可用。这种情况就称为内存泄露。

什么样的程序才会长年累月的运行呢？服务器程序，当一个程序做服务运行时，可能一次运行好几个月。

如何检查一个程序是否内存泄露呢？当程序运行时，在系统的“任务管理器”中会记录一个进程的内存使用量，如图 12-1 所示。

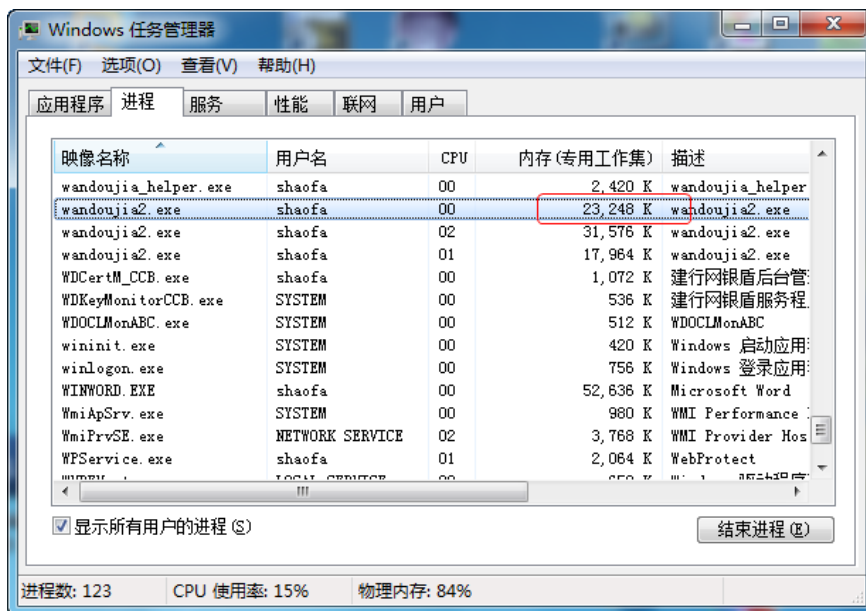


图 12-1 在任务管理器中观察程序占用的内存

如果我们发现这个内存的使用量随着时间逐渐增长，那么无论它增长得快与慢，都是一个需要关注的问题——因为你的程序是要长期运行的，日积月累，即使一个慢速的内存泄露都将是致命的。

1.94 对象的生命期

在这里，引入对象的概念来讨论生命期的概念。

假定有结构体的定义如下：

```
struct Object
{
    int id;
    char name[64];
};
```

1.94.1 对象的分类

对象分为三类：①全局对象 ②局部对象 ③动态对象。

(1) 当我们定义一个变量时，

```
Object a;
```

则变量 `a` 对应了一个对象，类型为 `Object`，地址为 `&a`。如果这个变量是全局变量，则 `a` 称为全局对象。如果变量 `a` 是局部变量，称为 `a` 是局部对象。

(2) 当我们用 `malloc` 动态申请内存时，

```
Object* p = (Object*) malloc (sizeof (Object)) ;
```

此时，则 p 指向了一个对象，类型为 Object，地址为 p。该对象的内存是动态分配的，称为动态对象。

总之，一个对象总是对应了一块内存，对象的值就是内存里的数据。我们在描述一个对象时，只需要知道它的类型和地址就够了。

比如，对于一个全局对象或局部对象，

```
Object a;  
  
Object* p = & a; // 对象类型为 Object, 地址为&a,  
  
p->id = 123;  
  
strcpy(p->name, "something");
```

又如，对于一个动态对象，

```
Object* p = (Object*) malloc (sizeof (Object)) ; //对象类型为 Object, 地址为 p,  
  
p->id = 123;  
  
strcpy(p->name, "something");
```

1.94.2 对象的生命期

- (1) 全局对象：生命期地永恒的，只在程序退出时对象才失效
- (2) 局部对象：生命期是临时的，在超出作用域后对象立即失效
- (3) 动态对象：生命期是动态的，在 malloc 时生命生效，在 free 时失效

这也是动态对象的另一层含义：程序员可以动态控制它的生命期，如果从来不 free 它的话，那这个对象就是一直有效的。

从内存区域来说，动态对象对象处于堆区，全局对象则处于全局数据区，局部对象处于栈区。其中，堆是一个系统级的内存区域，由内存管理器管理。全局数据区和栈则是应用程序自己拥有的内存区域。

注：全局数据区和栈的具体内涵不在本书中阐述。读者只需要知道这几个句语术语，并掌握三种类型的对象的生命期的区别。

1.95 实例 1

现在我们来解决引例中的问题。

(1) 修改结构体的定义，用一个指针来表示影片名称，同时添加一字段 title_len 来记录内存空间的大小。

////////// 例 CH12_B1 //////////

```
struct Movie
```

```
{  
    int id;  
    char* title; // 影片名称  
    int title_len; // 影片名称长度  
};
```

(2) 申请一块空间来保存影片名称

根据影片名称的实际长度，申请适当大小的空间。用 `strlen` 函数来计算字符串的长度。

```
void SetTitle(Movie* m, const char* title)
```

```
{  
    m->title_len = strlen(title); // 影片名称的实际长度  
    m->title = (char*) malloc(m->title_len + 1); // 申请一块空间  
    strcpy(m->title, title); // 将数据存储到这块空间  
}
```

(3) 最后要记得释放内存

```
int main()
```

```
{  
    Movie m = { 0 }; // 清零  
    m.id = 1;  
    SetTitle(&m, "The Bourne Identity");  
    printf("id: %d, title:%s\n", m.id, m.title);  
  
    // 最后要记得释放内存  
    if(m.title) free(m.title);  
    return 0;  
}
```


1.96 实例 2

示例：用 Citizen 表示一个市民，用 Car 表示一个辆车。他起初没有车，但未来可能有一辆车。

```
////////// CH12_C1 //////////
```

```
struct Car
```

```
{  
    char maker[32]; // 制造商  
    int price; // 价格  
};
```

```
struct Citizen
```

```
{  
    char name[32]; // 名字  
    int deposit; // 存款  
    Car* car; // NULL 时表示没车  
};
```

定义一个对象,开始没车.....

```
Citizen shaofa = { "shaofa", 100, NULL };
```

后来，他可能买了一辆车.....

```
void buy(Citizen* owner)
```

```
{  
    // 创建一个对象  
    Car* car = (Car*) malloc(sizeof(Car));  
    strcpy(car->maker, "Chevrolet");  
    car->price = 10;
```

```
    // 保存此对象 (确切地说是记住了指针)
```

```
    owner->car = car; //车有了
```

```
owner->deposit -= car->price; // 钱没了  
}
```

终有一天，这车会报废.....

```
void discard(Citizen* owner)  
{  
    free(owner->car); // 此对象被销毁  
    owner->car = NULL; // 回到无车状态  
}
```

也有可能会卖给别人.....

```
void sell(Citizen* owner, Citizen* other)  
{  
    Car* car = owner->car;  
    car->price *= 0.5; // 半价出售  
    other->car = car; // 别人拥有了这辆车  
  
    owner->deposit += car->price; // 收回一部分成本  
    // free(car); // oh,no! 不能 free，这车在别人手里  
    owner->car = NULL; // 回到无车状态  
}
```

1.97 常见问题

(1) 用 malloc 申请的内存，用完后用 free 释放

malloc 与 free 是配套使用的。

(2) 不是通过 malloc 得到的内存，绝不能用 free 释放

例如，

```
int a = 0;  
int* p = &a;
```

`free(p);` // 运行时错误！`p` 不是 `malloc` 来的，不能用 `free` 释放。

显然，`p` 指向的内存并不是从 MM 那里“借”来的。

(3) 及时归还，再借不难

动态分配内存的一个使用原则：用完了要及时 `free`。

(4) 不能只 `free` 一部分

从 MM 借来的内存，必须一次性全部还回，不允许只还一部分。例如下面的代码，`malloc` 的时候申请了 16 个字节，但是 `free` 的时候却试图只还 12 个字节，

```
char* p = (char*) malloc[16];
```

```
free(p + 4); // 这样不允许的！
```

(5) 什么时候 `malloc` 返回 `NULL`？

在 `malloc` 调用之时，如果 MM 无足够的空闲内存，则 `malloc` 返回失败。发生这种情况的原因，往往是你要申请的内存太大了。当然，也有可能是你的程序写的有问题，不停地申请内存却没有释放，就下面这样的代码，最终就导致系统运行缓慢或崩溃。

```
while(1)
{
    void* ptr = malloc(1024*128); // 终将用尽！
}
```

(6) `malloc` 用的内存一定要 `free` 吗？

如果你一共就申请了很小的内存（比如，只申请 1 个字节），其实不 `free` 也没关系，MM 是不会向你讨还的，对整个系统影响不大。

(7) 程序退出时，忘了 `free` 内存怎么办？

当程序退出时，所以曾经 `malloc` 的内存，都会自动地被释放归还给内存管理器。

第 13 章 链表

链表（Linked-List），是一个数组的组织方式。它是工程实践中最常使用的一种数据结构，是一个必须要掌握的概念。

1.98 概念

把若干个对象用指针串连起来，形成一个链状的数据结构，称之为“链表”。如图 13-1 所示。



图 13-1 链表示意图

本章介绍如何在 C/C++ 中实现链表。

首先，用 `struct` 语法定义一个类型。下面例子中，以 `Student` 来存储一个学生的学号和姓名：

```
struct Student
{
    int id;
    char name[16];
    Student* next; // 添加一个指针, 用于指向下一个对象
};
```

注意，其中添加一个成员变量 `next`，用于指向下一个对象。

1.99 链表的构造

下面构造一个链表，用于演示各个对象“串联”起来的效果。

（1）先准备好 4 个对象

////////// 例 CH13_A1 //////////

```

Student ss[4] =
{
    {201501, "John",          0 },
    {201502, "Jennifer",     0 },
    {201503, "AnXi",        0 },
    {201504, "Unnamed",     0 }
};

```

此时，这 4 个对象在内存中一字排开，紧密排列，它们还没有被“串”起来。

(2) 把这 4 个对象“串”起来

```

ss[0].next = &ss[1];
ss[1].next = &ss[2];
ss[2].next = &ss[3];
ss[3].next = 0;

```

至此，一个“链表”构造完毕。

1.99.1 头节点与末节点

当若干个对象被“串”起来之后，只需要知道第一个对象，就可以访问到链表中的每一个对象。我们把链表中每个对象，称为“节点”。第一个节点也称为“头节点”或者“链表头”。

例如，我们使用链表头的头部，可以依次访问到链表中每一个节点。

////////// 例 CH13_A1 //////////

```

Student* p = &ss[0];
while(p)
{
    printf("ID: %d, name: %s\n", p->id, p->name);
    p = p->next; // 下一个对象
};

```

这一过程，称为链表的遍历。

链表的最后一个节点称为“末节点”或“尾节点”。末节点的 next 必须设置为 NULL，它是用于标识这个链表的结束的。当我们遍历链表时，遍历到 NULL 认为是结束。

1.99.2 链表头的作用

链表头可以用于代表整个链表。这是因为，只是知道了第一个节点，则所有节点数据都有办法得到。

```
Student* stu_list = &ss[0];
```

1.99.3 常见问题

(1) 为什么一个 struct 中的成员类型可以是“自己”？

```
struct Student
{
    int id;
    char name[16];
    Student* next; // 这是 Student*，不是 Student
};
```

确切地说，next 是类型是 Student*，而不是 Student。在 C/C++ 中，所有的指针类型本质上都是一个整型，大小固定为 4 字节（32 位系统上）所以在包含一个整型是没有问题。

下面的代码才是有问题的：

```
struct Student
{
    int id;
    char name[16];
    Student next; // 编译错误！类型嵌套了！
};
```

(2) 为什么最后一个节点的 next 必须设置为 NULL？

最后一个节点的 next 必须是 NULL，否则无法判断一个链表的结束。

1.100 有头链表

一个链表，当有 N 个对象串起来是，使用第一个对象来指代整个链表。可以向链表中添加对象，也可以删除对象。我们考虑一个问题：如果把链表中所有对象链表，这个链表没有了链表头，该如何表示？

1.100.1 概念

在具体的工程实践中，链表有两种实现方式：无头链表，有头链表。

无头链表：没有固定的头节点。所有的节点都包含了有效数据。（上节课演示的链表就是无头链表）。当链表中的节点全部删除时，或者向链表的头部之前插入一个节点时，此时链表在表示上存在困难。

有头链表：用一个固定的头节点来指代整个链表，所有的对象挂在这个头节点下面，而头节点本身不包含有效数据。在图 13-2 所示，这是一个有头链表（有固定的头部），链中含有 1 个头节点和 4 个数据节点。

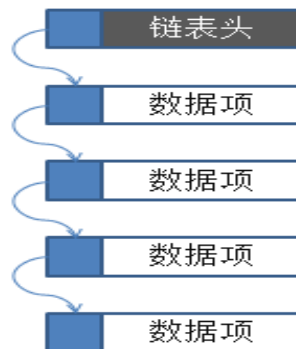


图 13-2 使用固定头节点的“有头链表”示意图

使用有头链表的目的是简化链表操作，使之容易实现。在本节中介绍有头链表的常用操作。为了便于说明问题，将头部节点称为“**头节点**”，将后面的节点称为“**数据节点**”。

1.100.2 定义一个有头链表

只需要定义一个对象作为其节点，将成员 `next` 初始化为 `NULL`。

```
Student m_head = {0, "head", NULL};
```

或写作

```
Student m_head = {0};
```

当有对象加入时，直接加在它的后面就可以。当它的 `next` 为 `NULL` 时，表示该链表没有数据节点（链表长度为 0）。

1.100.3 添加一个节点

////////// 例 CH13_B1 //////////

```
void add(Student* obj)
```

```
{
```

```

obj->next = m_head.next;
m_head.next = obj->next;
}

```

链表中的对象，一般都动态创建的。比如，当用户需要添加一个对象时，从控制台输入信息，然后用 malloc 动态创建一个节点，插入到链表中。

比如，原链表中有 4 个对象 A,B,C,D，调用 add 函数，添加一个对象 X。这个过程如图 13-3 所示。

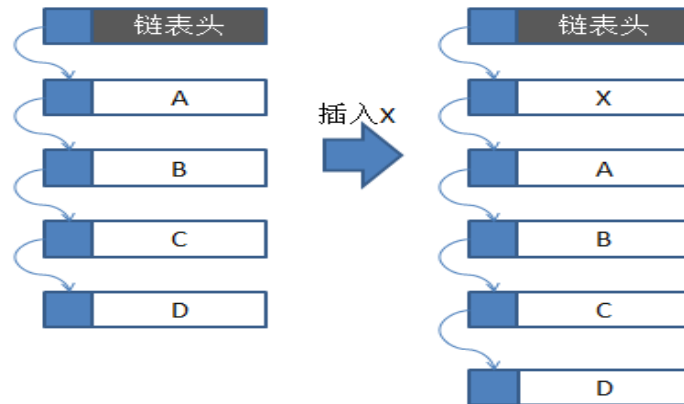


图 13-3 在链表中插入一个节点

下面的代码，首先动态创建一个对象，然后调用 add 函数插入到链表，

////////// 例 CH13_B1 //////////

```

Student* obj = (Student*)malloc (sizeof(Student));
obj->id = 12;
strcpy(obj->name, "X");
add(obj);

```

前面的 add 函数是直接把新的节点插入到了最前面，也可以把节点附加到末尾。代码如下，

////////// 例 CH13_B1 //////////

```

void add(Student* obj)
{
    Student* p = &m_head;
    while(p->next)
        p = p->next; // 找到最后一个对象
}

```



```

p->next = obj; // 把 obj 挂在最后一个对象后面
obj->next = NULL; // 现在 obj 作为最后一个对象
}

```

1.100.4 有头链表的遍历

在遍历时，有头链表的头节点由于不含有数据，是不参与遍历的。实际遍历时，只访问链表中的数据节点。示例代码如下。

```

////////// 例 CH13_B1 //////////
void show_all()
{
    Student* p = m_head.next;
    while(p)
    {
        printf("ID: %d, name: %s\n", p->id, p->name);
        p = p->next; // 下一个对象
    }
}

```

1.100.5 按顺序插入节点

已经介绍了两种添加节点的方法：添加到最前面、或者添加到末尾。

这里再介绍一种按排序插入的情况。例如，要求链表中的 **Student** 对象的 **id** 按从小到大顺序。

原链表中已经存在 1，3，4，8 四个节点，新入一个 ID 为 5 的节点，如图 13-4 所示。

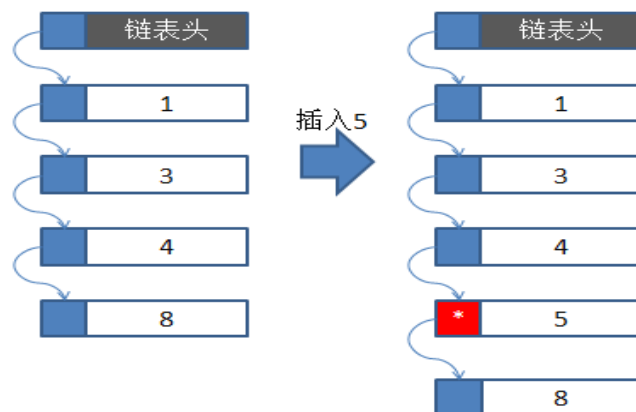


图 13-4 在链表中按顺序插入节点

如何实现按顺序插入呢？在插入时，先遍历链表，并比较 ID 的值，找到目标位置，并记录前一个节点为 `pre`。当找到位置之后，把新节点直接挂在 `pre` 后面就行了，这个操作很简单：

```
obj->next = pre->next;
```

```
pre->next = obj;
```

示例 CH13_B1 的 `insert` 函数展示了如何按顺序插入，相关代码如下，

```
////////// 例 CH13_B1 //////////
```

```
int insert(Student* obj)
```

```
{
```

```
    Student* cur = m_head.next; // 当前节点 current
```

```
    Student* pre = &m_head; // 上一个节点 previous
```

```
    while(cur)
```

```
    {
```

```
        if(obj->id < cur->id) break; // 找到这个位置
```

```
        pre = cur;
```

```
        cur = cur->next; // 找到最后一个对象
```

```
    }
```

```
    // 插入到 pre 节点的后面
```

```
    obj->next = pre->next;
```

```
    pre->next = obj;
```

```
    return 0;
```

```
}
```

1.100.6 查找和删除节点

当需要删除一个节点时，先找目标节点，并记录该节点的前一个节点 `pre`。然后删除这个节点，方法很简单：

```
pre->next = obj->next; // 从链表中移除
```

```
free(obj); // 释放资源
```

示例 CH13_B1 的 `remove` 函数展示了如何删除一个节点，相关代码如下，

////////// 例 CH13_B1 //////////

```
void remove(int id)
{
    Student* cur = m_head.next; // 当前节点 current
    Student* pre = &m_head; // 上一个节点 previous
    while(cur)
    {
        if(id == cur->id) // 找到这个位置
        {
            // 删除该节点

            pre->next = cur->next;

            free(cur);

            break;
        }
        pre = cur;
        cur = cur->next; // 找到最后一个对象
    }
}
```

1.100.7 和无头链表的比较

“无头链表”的操作相对复杂，尤其是插入和删除操作。另外，当链表长度为 0 时难以表示。当需往链表头插入节点时，也难以表示。

“有头节点”仅仅是增加了一个固定的头部，就使得各种操作大大简化。虽然在内存空间上是浪费了一个节点的空间，但由于链表往往很长，只多用一个节点的空间往往这种浪费是可以忽略不计的。显然，“有头节点”的利大于弊，推荐使用。

第 14 章 引用

“引用”是 C++ 中新引入的概念，在 C 中没有。引用的本质和指针相同，相当于是指针语法的另一种写法。读者在初次学习时，可以先行跳过本章，在学到 20 章的时候再回头来好好学习本章。也就是说，在 20 章之前是不涉及本章语法的。

1.101 引用的定义

在类型之后加上一个 & 符号，该变量的类型即为“引用”类型。例如，

////////// 例 CH14_A1 //////////

```
struct Object
{
    int value;
};
int main()
{
    Object a = {1};
    Object& r = a; // r 引用的对象是 a
    return 0;
}
```

在此例中，变量 `a` 是一个类型为 `Object` 的对象，变量 `r` 则是对象 `a` 的一个引用。也就是说，总共只有一个对象。`r` 只是相当于这个对象的一个别名。

所有对引用的访问，实质上都是在访问被引用的那个对象。

例如，

```
r.value = 2; // 对 r 的访问就是对 a 的访问
printf("value: %d \n", a.value);
```

当我们改用指针的语法实现，一切就变得容易理解。

```
Object* p = &a;
p->value = 2;
```

所达到的效果是完全一样的。也许当我们使用引用的语法的时候，会让时代码看起来是要简洁一些。

1.102 与指针的区别

引用和指针的最大区别，就是引用在定义的时候必须初始化关联到一个对象（必须有对象）。而指针在初始化的时候，则可以初始化空指针（没有对象）。

例如，

```
Object* p = NULL; // 允许在定义的时候不指向任何对象
```

而在定义引用的时候，必须指向一个实实在在的对象。如果在定义一个引用的时候不初始化，则编译器报错。例如，以下代码有语法错：

```
Object a;
```

```
Object& r; // 语法错！定义引用类型的变量，必须初始化！
```

简而言之，引用从“出生”开始，便与某个对象绑定，中途无法解绑。比如在示例 CH14_B1 中，引用 r 与对象 a 绑定，所有对 r 的操作本质上都是对 a 的操作。

```
////////// CH14_B1 //////////
```

```
int main()
```

```
{
```

```
    Object a = {1}; // 第一个对象
```

```
    Object b = {2}; // 第二个对象
```

```
    Object & r = a; // r 与第一个对象绑定
```

```
    r = b; // 这并不是重新绑定。这只是第二个对象的值赋值给了第一个对象
```

```
    printf("a: %d \n", a.value);
```

```
    return 0;
```

```
}
```

相对之下，指针的使用则较为灵活，一个指针可以先指向对象 a，再指向对象 b，完全被用问题。

```
int main()
```

```
{
```

```
    Object a = {1}; // 第一个对象
```

```
    Object b = {2}; // 第二个对象
```

```
    Object * p = & a; // p 指向对象 a
```

```
    p->value = 11; // 修改了对象 a 的值
```

```
    p = &b ; // p 变为指向对象 b
```

```
    p->value = 22; // 修改了对象 b 的值
```

```
    return 0;
}
```

从某种程度上讲，“引用”可以视为功能受限的指针。

1.103 简单的例子

下面给一些简单的例子，让读者加深对引用的印象。

(1)

```
int a = 123;
int & r = a; // 则 r 的值就是 123
```

(2)

```
double a = 12.345;
double* p = &a;
double r = *p; // 则 r 的值是 12.345
```

(3) 引用的地址，就是被引用的对象的地址

```
Object a = { 1 };
Object& r = a; // r 引用的是对象 a
Object* p = &r; // p 指向对象 a
p->value = 2;
```

1.104 作为函数的参数

引用类型可以作为函数的参数，可以达到与指针相同的效果。

例如，我们有一个函数 Test，其参数为指针类型，代码如下：

```
void Test ( Object* p )
{
    p->value += 1;
}

int main()
{
    Object a = { 1 };
}
```

```

    Test(&a);
    return 0;
}

```

也可以使用引用的语法来实现，代码如下，

////////// CH14_C1 //////////

```

void Test(Object& r)
{
    r.value += 1;
}

int main()
{
    Object a = { 1 };
    Test(a);
    return 0;
}

```

这两种写法在功能上是完全一样的。当使用“引用”来实现时，在传参时直接传入对象 `a`，在调用函数 `Test` 的时候对参变量进行了初始化：

```
Object& r = a;
```

那么接下来，在 `Test` 函数里的所有对 `r` 的操作，实质上都是在操作被引用的那个对象。

1.105 作为函数的返回值

引用也可以作为函数的返回值。

首先，指针是可以作为函数的返回值的，其意义就是把某个对象的地址返回。例如，在下面的例子中有一个全局变量 `the_object`，在函数 `Test()` 里返回了的指针指向了这个对象。

```

Object the_object = { 123 }; // 全局对象

Object* Test()
{
    return &the_object; // 返回某个对象的指针
}

int main()
{

```

```

    Object* p = Test();

    p->value = 456; // 指针指向的对象有效

    return 0;
}

```

同样地，用引用的语法也可以实现相同的功能，如例 CH14_D1 所示。

```

////////////////// CH14_D1 ////////////////////

Object the_object = { 123 }; // 全局对象

Object& Test()
{
    return the_object; // 返回某个对象的引用
}

int main()
{
    Object& r = Test();

    r.value = 456; // 被引用的对象有效

    return 0;
}

```

和指针一样，引用也存在安全性的问题：被引用的对象是有效的吗？只有有效的对象才是可以操作的。例如，下面的代码引用一个无效的对象。

```

////////////////// 例 CH14_D2 ////////////////////

Object& Test()
{
    Object the_object = { 123 }; // 局部对象

    return the_object; // 返回某个对象的引用
}

int main()
{
    Object& r = Test();

    r.value = 456; // 被引用的对象已经失效

    return 0;
}

```


Test 函数的返回值引用了一个局部对象，当 Test 函数返回后，此对象已经失效。因此返回的引用不能够被访问，如果你要访问它的话，有可能引起程序的崩溃，因为那个对象已经不存在了。

1.105.1 函数返回左值

当函数返回值的类型为引用时，我们称该函数的返回值是左值。这意味着，我们可以下面的方式来使用这个函数。

```
Test().value = 456;
```

实际在，在 C/C++ 里当我们说一个函数返回左值时，意思就是它返回了一个引用类型。

1.106 const 引用

和 const 指针对比，也有 const 引用的语法，表示的意思也是相同的：限定被引用的对象为只读的，不能修改对象的值。常用作函数的参数。

例如，在下面的代码中试图修改 const 引用的值，会引起编译器报错。

```
void Test ( const Object& r)
{
    r.value = 111; // 编译器报错！无法修改 const 对象的值！
}
```

而下面的代码是没有问题的，它只是在读取一个 const 引用对象的值。

```
void Test ( const Object& r)
{
    printf("value: %d \n", r.value);
}
```

第 15 章 字符串

本章讨论的字符串，通常被称作 C 风格字符串（C-Style String），它的特征是：

- ① 在内存中紧密排列的一串字符，以 0 结尾。
- ② 以内存的首地址来代表该字符串, `char*`

虽然我们在本章把字符串中的内容称为“字符”，但仍然强调一遍，所谓的字符实质还是一个介于 0-127 之间的整数。具体的含义请回顾第五章中的说明。

1.107 字符串的三种形式

字符串通常以三种形式存在：字符数组、动态字符串和字符串常量。当然，本质上它们是相同的：有一块内存，内存中存放了一串字符、且以 0 结束。为了容易表达，下面把这一块存放了字符串的内存，称为一个字符串对象。

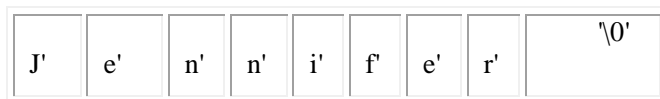
1.107.1 字符数组

当以 `char` 型数组来存放字符串时，数组名是字符串的首地址。

例如，定义和初始化一个数组：

```
char name[ ] = "Jennifer";
```

则字符串 `name` 在内存中占据 9 个字节，形象地表示为：



1.107.2 动态字符串

可以动态地分配一块内存，然后在这块内存里存放一串字符。`也就是说，这个字符串对象在堆上。

```
char* str = (char*) malloc(12);  
str[0] = 'g';  
str[1] = 'o';  
str[2] = 'o';  
str[3] = 'd';  
str[4] = 0;
```

1.107.3 字符串常量

在第三章，我们知道所有的字面常量都有它的类型，例如，

```
100; // 类型为 int
```

```
100u; // 类型为 unsigned int （注意以 u 结尾）
```

```
3.1415926; // 类型为 double
```

```
3.14f; // 类型为 float （注意以 f 结尾）
```

以下代码中，定义一个 float 类型的变量，并给它赋值，

```
float a = 3.14; // 错! 等号右侧的值的类型是 double
```

```
float b = 3.14f; // OK。右侧值的类型是 float
```

现在，我们来认识一种新的字面常量：字符串字面常量。在代码中以双引号包括，包含 0..N 个字符，称为字符串字面常量。

例如，下面几个都是字符串型字面常量，

```
"Hello,World!" ;
```

```
"Jennifer" ;
```

```
"x=%d, y=%s ";
```

字符串字面常量的类型为 const char*，

```
const char* a = "Hello,World!";
```

其中，const char* 意味着这个指针指向的内存是只读的，不能修改里面的内容。实际上，字符串字面常量的值是被存放于一块只读内存区域（常量数据区）。

1.108 字符串常量的多行表示

当一个字符串常量要表达的内容特别长时，可以用两种方式表示。

第（1）种方法：使用双引号将多段文本连接起来。两个字符串中间可以被空白分开，不影响最终效果。例如，

```
const char* str = "hello" "world" ;
```

相当于

```
const char* str = "helloworld";
```

所以，我们在定义较长的字符串常量时，建议使用这种方式。例如，在网络通讯中，发送的文本字符串经常长达上百个字符，如下面的代码所示：

```
const char* http_request =  
    "GET /index.html HTTP/1.1\r\n"  
    "Host: www.afanihao.cn\r\n"  
    "Connection: keep-alive\r\n"
```

```
"\r\n";
```

第(2)种方法：在行末添加一个反斜线，例如，下面用字符串常量来表示一首歌的歌词，由于歌词很长，需要分多行书写。

```
const char* song =  
"Sleep, sleep, my love, my only,\nDeep, deep, in the dung and the dark, \nBe not afraid and be not lonely!\nThis is the hour when frogs and thrushes \nPraise the world from the woods and rushes \nRest from care, my one and only,\nDeep in the dung and the dark! ";
```

注意：末尾的反斜线表示本行结尾，继续到下一行。因此，行首的空白也属于字符串的内容，我们需要每一行顶格书写才对。

1.109 字符串与普通数据

字符串是以 `char*` 表示的，它指向了字符串对象的首地址。

然而，并不是所有的 `char*` 都被称作字符串。实际上，只要在把这块内存用于存储字符串的时候，才把它称作字符串。如果只是把它用于保存一些普通数字，则不把它称作字符串。

例如，定义一个 `char` 型数组，保存一些数字。

```
char data[4] = { 1, -1, 2, -2 };
```

可见，是不是字符串，只是根据它的用途而定的，当它用于存放一串可打印字符串时，才把它叫做字符串。

1.110 字符串的遍历

遍历字符串，指的是从前往后访问每一个字符。有两处方法实现遍历：索引遍历和指针遍历。无论哪种遍历，都需要检测结束符 `\0` 来判断其是否结束。

下面的代码中，使用 `for` 循环语句来遍历每个位置，打印该位置的字符；如果该字符为 `0`，表示已经到了字符串的结尾。

```
////////// 例 CH15_A1 //////////  
  
int show_string (const char* str) // 按索引遍历字符串  
{  
    for(int i=0; str[i] ; i++)  
    {  
        printf("%c ", str[i]);  
    }  
}
```

```

    }
    return 0;
}

```

结束符'\0'的作用，就是用于标识字符串的结尾。

下面的代码是用指针来进行遍历，实现了与前一个例子相同的功能，

```

////////// 例 CH15_A2 //////////

int show_string (const char* str) // 用指针遍历字符串
{
    const char* p = str;
    while (*p)
    {
        char ch = *p ++;
        printf("%c ", ch);
    }
    return 0;
}

```

1.111 字符串长度

字符串的长度，是指从第一个字符开始、一直到末尾的结束符，中间的有效字符的个数。长度是不包含末尾的'\0'在内的。

下面的代码用于计算字符串的长度，

```

////////// 例 CH15_B1 //////////

int GetLength (const char* p)
{
    int count = 0;
    while( p[count] )
        count ++;
    return count;
}

```

实际上，我们一般不需要自己写这样的函数，而是使用<string.h>里的 strlen 函数来计算字符串的长度。

```

////////// 例 CH15_B2 //////////

```

```

#include <stdio.h>

#include <string.h> // 包含这个一行

int main()
{
    int n = strlen("hello,world");

    printf("size: %d \n", n);

    return 0;
}

```

注：普通的数组和内存都需要用“首地址”+“长度”来指定，而字符串对象则只需要“首地址”就能可以指定，原因就在于它的数据本身是带了结束标识的。

1.112 字符串复制

字符串的复制，是指将源字符串的每一个字符挨个复制到目标缓冲区。最终须保证目标缓冲区的字符串末尾有一个'\0'字符。

以下代码中，将源字符 src 复制目标地址 dst，

////////// 例 CH15_C1 //////////

```

#include <stdio.h>

int main()
{
    char src [] = "hello"; // 源

    char dst[128]; // 目标

    int i = 0;

    while (1)
    {
        dst[i] = src[i] ;

        if(src [i] == 0) break; // 末尾的 0 也拷贝过去

        i ++;

    }

    printf("result: %s \n", dst);

    return 0;
}

```

最终, 目标 `dst` 字符串的前 6 字符是 `h'e'l'l'o` 这五个字符。



相关注意事项:

(1) 目标缓冲区要足够大, 防止越界

在前面的例子中, `dst` 有 128 个字节的空间, 而源字符串长度为 5, 所以是放得下的。

(2) 目标缓冲区保证以 0 结束

下面这种拷贝方法是不对的, 因为它没有把末尾的 0 拷贝到目标缓冲区, 导致目标字符串没有以 0 结尾:

```
while (src[i])
{
    dst[i] = src[i];
    i++;
}
```

(3) 可以使用 `<string.h>` 里的 `strcpy` 函数

////////// 例 CH15_C2 //////////

```
#include <string.h> // 引用头文件
```

```
...
```

```
char src [] = "hello"; // 源
```

```
char dst[128]; // 目标
```

```
strcpy(dst, src); // 用 strcpy 函数
```

1.112.1 区分浅拷贝和深拷贝

当用指针来存储字符串时, 就需要注意浅拷贝和深拷贝的问题。

例如,

```
char* p1 = "hello,world";
```

```
char* p2 = p1;
```

这种简单的指针赋值，就称为浅拷贝。它的本质有一句话表示：两个指针(p1,p2)指向了同一个字符串对象。

而下面的代码中，新申请一块相同大小的内存，然后把字符串内容拷贝到这块内存。那么，就存在两个字符串对象（两块内存），它们的内容相同（都是"hello,world"）。

```
char* p2 = (char*) malloc ( strlen(p1) + 1);  
strcpy(p2, p1);
```

这种则称为深拷贝，一句话表示：两个指针(p1, p2)，分别指向两个字符串对象。

1.113 字符串比较

字符串也可以比较是否相等，以及大小关系。在字符串比较时，是逐个字符依次比较，当所有字符全部相同时才认为两者相等。两个字符则是按它们的 ASCII 码值大小进行比较的。

例如：第 5 个字符分出高低

```
"Jack" : 'J' 'a' 'c' 'k' '\0'  
"Jacky" : 'J' 'a' 'c' 'k' 'y' '\0'
```

由于第 5 个字符 '\0' < 'y'，所以 "Jack" < "Jacky"

例如：第 2 个字符分出高下，剩余字符不再比较

```
"Jack" : 'J' 'a' 'c' 'k' '\0'  
"John" : 'J' 'o' 'h' 'n' '\0'
```

由于第 2 个字符 'a' < 'o'，所以 "Jack" < "John"

例如，有一串名字，

```
"Jennifer"  
"Jack"  
"Jacky"  
"Angle"  
"Micheal"
```

如果按从小到大排序的话，则排序结果为："Angle" "Jack" "Jacky" "Jennifer" "Michael"。

我们一般直接使用<string.h>里的 strcmp 函数来比较两个字符串，

在 string.h 中，strcmp 函数用于比较字符串，strcmp(a,b)当相等时返回值为 0，当 a<b 时返回值为-1。当 a>b 时返回值为 1。

////////// 例 CH15_D1 //////////

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

int main()
{
    int ret = strcmp("Jack", "Jacky");

    printf("Result: %d \n", ret); // 返回值-1, 表示"Jack" < "Jacky"

    return 0;
}
```

对于初学者来说,最常犯的一个错误便是把字符串的比较写成了指针的比较。在示例 CH15_D2 中,要求用户输入一个字符串,保存到缓冲区 input 中。然后希望在输入为"yes"时输出显示 "agree", 否则显示"don't agree"。显然,它这里的字符串比较的写法是错误的。

//////////例 CH15_D2 //////////

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

int main()
{
    char input[128];
    gets(input);
    if(input == "yes" ) // 这不是字符串比较
    {
        printf("agree \n");
    }
    else
    {
        printf("don't agree \n");
    }
    return 0;
}
```

其中, if(input == "yes")无法达到目的,它执行的是两个指针的数值比较,应该改成 if(strcmp(input, "yes") == 0) 这样的写法,才是对字符串的内容进行比较。

1.114 字符串插入和删除

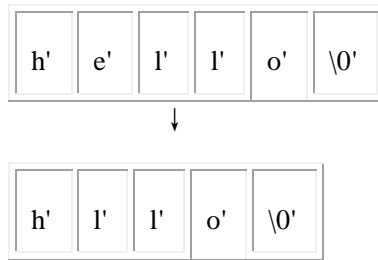
1.114.1 删除字符

给定了一个字符串，如果想删除其中的一个字符，该怎么做呢？

例如，对于字符串 `s`，

```
char s[] = "hello";
```

要求把字母 `e` 给删除，变成 `hlllo`。从内存视图来看这一过程，相当于：



这相当于，把 `s[2] ~ s[5]` 这几个内存中的值各自前移一个字节。即，

```
s[1] = s[2]
```

```
s[2] = s[3]
```

```
s[3] = s[4]
```

```
s[4] = s[5]
```

用一个循环表示的话就是 `s[i - 1] = s[i], 2 ≤ i ≤ 5`。

下面的代码中，函数 `Erase` 用于删除字符串中的某个字符，

```
////////// 例 CH15_E1 //////////
```

```
void Erase(char* src, int index)
{
    for (int i = index + 1; ; i++)
    {
        src[ i - 1 ] = src [ i];
        if (src[i] == '\0') break;
    }
}
```

1.114.2 插入字符

向字符串插入字符时，首先要保证原数组空间足够大。这是因为，在向指定位置插入字符后，后面的字符需要依次后移。例如，

```
char s[128] = "hello";
```

插入一个 i 字符，变成 hiello，相当一依次进行以下操作：

```
a[6] = a[5]; // '\0'
```

```
a[5] = a[4]; // 'o'
```

```
a[4] = a[3]; // 'l'
```

```
a[3] = a[2]; // 'l'
```

```
a[2] = a[1]; // 'e'
```

```
a[1] = 'i'; // 插入一个字符
```

下面的代码中，函数 my_insert 用于在源字符串中插入一个字符，

```
////////// 例 CH15_E1 //////////
```

```
void Insert (char* src, int index, char ch) /* index:要插入的位置; ch:待插入的字符 */
```

```
{
```

```
    int len = strlen(src); // 取得长度
```

```
    for(int i= len; i> index; i--)
```

```
        src[i] = src[i-1]; // 逐个后移
```

```
    src[index] = ch; // 在此位置插入字符
```

```
}
```

1.114.3 插入和删除的效率问题

在字符串的中间插入和删除都是一个高成本的操作，即使你只增删除了一个字符，也要把后面的所有字符都移动一遍。

比如，把下面的字符串里的所有的 'a' 删掉。

```
"China is a great country with a long history"
```

按前面的删除方法，有 4 个 'a'，每删除一个 'a' 都要把后面的所有字符移动一遍，一共要移动 4 次数据。这个成本太高了。在这种情况下，我们于其在原字符串对象上删除和移动操作，还不如新建一个字符串对象。

在例 CH15_E2 中，Erase 函数并没有在原字符串的基础上操作，而是新申请了一块内存，把有用的字符拷贝过来，然后把新建的这个字符串对象返回。这样效率更高一些。

```

////////// 例 CH15_E2 //////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// 从字符串中的查找和删除字符

char* Erase(const char* text, char del)
{
    int len = strlen(text); // 原字符串长度
    int count = 0;
    char* copy = (char*) malloc(len + 1);
    for(int i=0; i<len; i++)
    {
        char ch = text[i];
        if(ch != del )
        {
            copy[count] = ch;
            count ++;
        }
    }
    copy[count] = 0; // 添加结束符
    return copy;
}

int main()
{
    char input[128] = "China is a great country with a long history";
    char* copy = Erase(input, 'a');
    printf("result: %s \n", copy);
    free(copy);
    return 0;
}

```

1.115 字符串的分割

一个字符串有若干信息组成，每一段信息中间有分隔符分开。解析这个字符串，得到每一段内容，称为字符串的分割。

例如，一个字符串包含了若干单词，以逗号分隔。

"hello,world"

那么，如何将它分割为"hello"和"world"两个字符串呢？

我们知道，所谓字符串实际是由三个要素组成的：首地址，末尾的 0，中间的字符。据此，我们可以采用下面的方式来实现分割。

比如，char str[] = "hello,world";

h	e	l	l	o	,	w	o	r	l	d	\0
---	---	---	---	---	---	---	---	---	---	---	----

只要把 str[5]设置为 0，就得到了两个字符串，

h	e	l	l	o	\0	w	o	r	l	d	\0
---	---	---	---	---	----	---	---	---	---	---	----

////////// 例 CH15_F1 //////////

```
#include <stdio.h>

int main()
{
    char str[] = "hello,world";
    str[5]=0;
    char* part0 = str;
    char* part1 = str + 6;
    printf("part0: %s \n", part0);
    printf("part1: %s \n", part1);
    return 0;
}
```

也就是说，我们只需要把相应的分隔符位置设为 0，然后记录一下每一分段的起始位置，就完成了字符串分割的任务。按照这种设计思路，我们在例 CH15_F2 中，用 split 函数实现字符串的分割，将每一段字符串的首地址保存在 parts 中，返回值表示分段数目。

////////// 例 CH15_F2 //////////

```

#include<stdio.h>

int split(char text[], char* parts[])
{
    int count = 0; // 分段的个数

    int start = 0; // 每一分段的首地址

    int flag = 0; // 遍历 text, 标识当前是否处于有效字符


    int stop = 0; // 是否到达结束
    for(int i=0; !stop ; i++)
    {
        char ch = text[i];
        if(ch == 0) stop = 1; // 结束循环
        if(ch == ';' || ch == '\0' || ch == ' ' || ch == '\t' )
        {
            if(flag) // 遇到分隔符, 且当前状态为 flag=1
            {
                flag = 0;
                text[i] = 0; // 修改为结束符,完成分段
                parts[count] = text + start; // 记录首地址
                count ++;
            }
        }
        else
        {
            if(!flag) // 遇到有效字符, 且当前状态为 flag=0
            {
                flag = 1;
                start = i;
            }
        }
    }
}

```

```

    return count; // 返回分段个数
}

int main()
{
    char text[] = "hello,world,,good,\tmorning ";
    char* parts[16];
    int count = split(text, parts);
    return 0;
}

```

1.116 用数组，还是用指针

例如，定义一个 Object 的结构体，里面要保存一个整数和一个字符串。那么有两种方式。

第一种写法，用数组保存字符串，

```

struct Object
{
    int id;
    char name[64];
};

```

第二种写法：用指针保存字符串，

```

struct Object
{
    int id;
    char* name;
};

```

两种方法都是可以的。下面具体比较一下它们之间的优劣。

1.116.1 数组方式

优点：① 安全，不必维护指针 ② 操作简单，直接拥有对应的内存

缺点：① 浪费空间 ② 不适用较长的字符串

用数组保存字符串，其优点是简单、安全，不需要动态分配内存，操作起来方便。但一般只适用于较短（<4096）的字符串。因为当数组长度较大时，结构变量无法放在函数栈上，使用起来不方便。

数组的长度必须设置为足够长。以名字为例，有的人名字长、有的人名字短，因此在设置数组的长度时要预先估计其最大可能长度。这里定义为 `char name[64]`，表示名字的长度不能超过 63（末尾要加'\0'）。这说明用固定数组保存字符串时，缺点是会浪费空间。

例如，保存名字 "jack" 只需要 5 个字节，浪费了 64-5=59 字节。保存 "jennifer" 需要 9 个字节，浪费了 64-9=55 字节。当内存空间足够、而且字符串不太长时，优先使用数组来保存字符串。

可以使用字符串复制的方法来为结构的字符串成员赋值：

```
Object obj;
```

```
strcpy (obj.name, "Whatever");
```

拷贝两个对象时，直接赋值即可：

```
Object obj2 = obj;
```

1.116.2 指针方式

优点：① 节省空间 ② 适用于较长的字符串

缺点：① 安全风险 ② 操作相对复杂

用指针保存字符串，就是根据字符串的实际上长度，分配一块长度恰当的内存。其好处是节省了空间，然而操作起来却相对要复杂一些。

在例 CH15_F1 中，Create 函数用于初始化一个 Person 对象，而 Destroy 则用于销毁一个对象。

```
////////// CH15_G1 //////////  
  
#include <stdio.h>  
  
#include <stdlib.h>  
  
#include <string.h>  
  
struct Object  
{  
    int id;  
    char* name;  
};  
  
void Create(Object* p, int id, const char* name)
```



```

{
    p->id = id;

    int len = strlen(name);
    p->name = (char*) malloc(len + 1);
    strcpy(p->name, name);
}

void Destroy(Object* p)
{
    free(p->name);
}

int main()
{
    Object obj;
    Create(&obj, 1, "something");
    printf("id: %d, name: %s \n", obj.id, obj.name);
    Destroy(&obj);
    return 0;
}

```

当以这种方式来存储字符串时，还要注意对象拷贝的问题。不能像下面这样复制对象了：

```
Object obj2 = obj; // 浅拷贝，两个对象指向同一块内存是不正确的设计方式
```

这样的对象 obj2 实际上是无法使用的。因为当 obj 被 Destroy 之后，所指向的内存被 free，那 obj2 也无法使用了。obj2 应该拥有一块自己的内存。

每个 Object 对象都应该拥有自己的一块内存（用于存储字符串）。我们得添加一个 Copy 函数，以深拷贝的方式来完成对象的复制：

```

void Copy(Object* dst, const Object* src)
{
    dst->name = (char*) malloc ( strlen(src->name) + 1);
    strcpy(dst->name, src->name);
    dst->id = src->id;
}

```


第 16 章 标准 C 函数库

标准 C 函数库，是由 ANSI 组织定义一系列标准函数，它在各种平台、各种编译器上都被支持，通常也被称为 ANSI C 函数库。这意味着，用 ANSI C 函数库写的代码，可以在 Windows 上编译运行，也可以在 Linux 上编译运行，且运行结果一致。

1.117 stdio.h

标准输入/输出函数，stdio 代表 standard input/output。其中文件操作函数，如 fopen/fclose 等将在后面的章节中介绍。

```
#include <stdio.h>

char getchar(void); // 控制台输入一个字符

int putchar(int c); // 控制台输出一个字符

char *gets(char *s); // 控制台输入一个字符串

int puts(const char *s); // 控制台输出一个字符串

int printf(const char *format, ...); // 控制台格式化输出

int scanf(const char *format, ...); // 控制台格式化输入

int sprintf(char *s, const char *format, ...); // 字符串格式化输出

int sscanf(const char *s, const char *format, ...); // 字符串格式化输入
```

1.117.1 getchar 与 putchar

从控制台输入/输出一个字符，例如，

```
char ch = getchar();

putchar('A');
```

在控制台输入一个字符后，按回车键完成输入，getchar 函数返回输入的字符的值。

1.117.2 gets 与 puts

从控制台输入/输出字符串。gets 用于输入字符串，例如，

```
char buf[256];

gets(buf);
```

在控制台输入若干字符后，按回车键完成输入，gets 函数将输入的字符存入 buf 并以'\0'结尾，返回的字符串指针即指向输入参数 buf。

puts 用于输出字符串，例如，

```
puts("Hello,world");
```

可以用 printf 和 %s 来替代 puts 的作用。

1.117.3 printf 与 scanf

比 gets/puts 的功能加强，在输入/输出的时候加上格式化的功能。printf/scanf 已经用过无数次了，这里就不再重复讲了。

1.117.4 sprintf 与 sscanf

适用于 string 版本的格式化输入/输出，其目标不是控制台，而是一个字符串。这两个函数非常有用。

用 sprintf 格式化一个字符串，例如，

```
char buf [256];  
sprintf (buf, "Name: %s, Age: %d, Height: %.2f \n", "LiMing", 30, 1.68 );
```

此代码将目标 buf 格式化为: Name: LiMing, Age 30, Height: 1.68

用 sscanf 从一个具有格式的字符串中提取固定字段，和 scanf 的用法类似，要求在格式上要严格匹配。其返回值表示成功提取到的字段的个数。以下代码从字符中提供取年月日的值。

//////////例 CH16_A1 //////////

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
char* src = "2014-12-11";
```

```
int year, month, day;
```

```
int n = sscanf ( src,
```

```
    "%d-%d-%d",
```

```
    &year, &month, &day);
```

```
if( n == 3) // 成功提取了 3 个字段
```

```
{
```

```
    printf("success: %d, %d, %d \n", year, month, day);
```

```
}
```

```
return 0;
```

```
}
```

注：和 `scanf` 一样，`sscanf` 在提取整数时必须用 `int`，不能用 `short` 或 `char` 型。

注：`sscanf` 只适用于提取数字字段，不适合提取字符串类型的字段。而且，当字符串的格式变得复杂时，我们一般无法用 `sscanf` 得到正确结果。对于比较复杂的格式，应该采用第 15 章的方法，对字符串进行手工解析和提取。

1.118 math.h

提供了一系列数学相关的函数，如三角函数、指数/对数、幂/根号等。

```
#include <math.h>

double abs(double x); // 取绝对值

double cos(double x); // 余弦 cos, 参数是弧度值

double sin(double x); // 正弦 sin, 参数是弧度值

double tan(double x); // 正切 tan, 参数是弧度值

double ceil(double x); // 向上取整, 即不小于 x 的最小整数

double floor(double x); // 向下取整, 即不大于 x 的最大整数

double exp(double x); // 求  $e^x$ 

double log(double x); //  $\ln(x)$ , 以  $e$  为底的对数

double log10(double x); //  $\lg(x)$ , 以 10 为底的对数

double pow(double x, double y); // 求幂  $x^y$ 

double sqrt(double x); // 求平方根  $x^{1/2}$ 
```

需要补充说明的是，这里所列的几乎所有函数都至少有 2 个版本，分别是 `double` 型和 `float` 型参数。例如，

```
double sqrt(double x);

float sqrt(float x);

以下代码编译错误，

////////// 例 CH16_B1 //////////

#include <math.h>

int main()
```

```

{
    double result = sqrt (16); // 编译错误！
    return 0;
}

```

为什么会有编译错误呢？因为字面常量 16 是 int 型，在匹配函数时，sqrt(float)与 sqrt(double)均被匹配。（参考第八章“重载函数的匹配”）此时应该做显式类型转换，如下面的代码所示：

```

double r1 = sqrt ((double)16); // OK
float r2 = sqrt ((float)16); // OK

```

1.118.1 abs 求绝对值

例如，

```
double a = abs (-12.34);
```

1.118.2 cos/sin/tan 三角函数

其单位都是弧度值。例如，

```

const double PI = 3.1415926535898;
double ret = sin ( PI / 2); // sin (pi/2)结果应为 1

```

1.118.3 ceil /floor 取整

用于向上/向下取整。例如，

```

double ci = ceil ( 12.87 ); // +13
double fi = floor(12.87); // +12
ci = ceil ( -12.87 );      // -12
fi = floor( -12.87);      // -13

```

1.118.4 exp / log / log10 / pow / sqrt 指数/对数/幂/平方根

例如，求 e6 例

```
double ret = exp (6.0);
```

求 ln (128),

```
double ret = log (128.0 );
```

求 lg (128),

```

double ret = log10 (128.0);
求 log264,
double ret = log ( 64.0 ) / log (2.0);
求 613,
double ret = pow(6.0, 13.0);
求 84/3,
double ret = pow(8.0, (double)4/3);
求 161/2,
double ret = sqrt (16.0);

```

1.119 time.h

time.h 中提供了时间/日期相关的函数。这里仅列出常用的几个函数。

```

#include <time.h>
struct tm *localtime(const time_t *tod); // 当“秒值”转成“年月日时分秒”
time_t mktime(struct tm *tptr); // 将“年月日时分秒”转成“秒值”
time_t time(time_t *tod); // 取得当前时间

```

其中，需要介绍 time_t 和 struct tm 这两个类型。

1.119.1 time_t

time_t 是一个 typedef 的类型，目前在各种操作系统上 time_t 类型都是一个整数类型，差不多就是

```

typedef long time_t;

```

它的单位是秒。如果需要将 time_t 打印显示，那么要将类型转换 int 型再使用，例如，

```

////////// CH16_C1 //////////
#include <stdio.h>
#include <time.h>

```

```

int main()
{
    time_t start = 1000;
    time_t end = 1020;
    printf("time eclipse: %d seconds !\n", (int)(end - start));
    return 0;
}

```

注：关于 **typedef** 的语法和用途，请参考附录《**typedef** 的用法》。

1.119.2 struct tm

tm 是一个结构，它的定义是

```

struct tm
{
    int tm_sec;    /* seconds after the minute - [0,59] */
    int tm_min;    /* minutes after the hour - [0,59] */
    int tm_hour;   /* hours since midnight - [0,23] */
    int tm_mday;   /* day of the month - [1,31] */
    int tm_mon;    /* months since January - [0,11] */
    int tm_year;   /* years since 1900 */
    int tm_wday;   /* days since Sunday - [0,6] */
    int tm_yday;   /* days since January 1 - [0,365] */
};

```

它用于表示日期/时间，有几个字段组成：年，月，日，时，分，秒，weekday 和 yearday。其中，

年：从 1900 开始算，tm_year = 114 表示年份 1900 + 114 = 2014

月：范围是[0,11], tm_mon = 11 表示月份 12

日: 范围是[1,31], tm_mday = 24 表示该月的第 24 天

时: 范围是[0,23], tm_hour = 13 表示下午 13 时

分: 范围是[0,59], tm_min = 40 表示第 40 分钟

秒: 范围是[0,59], tm_sec = 40 表示第 40 秒钟

tm_wday: 范围是[0,6], 星期日是 0, 星期一是 1, ..., 星期六是 6

tm_yday: 范围是[0,365], tm_yday=299, 表示当年的第 300 天

当 tm_min 和 tm_sec 都为 0 时, 表示整点时间。如 12:00:00。

例如, 2014-12-11 11:47:12 这个时间可以用下面的代码赋值:

```
tm info;  
info.tm_year = 2014 - 1900; // 2014 年  
info.tm_mon = 12 - 1; // 12 月  
info.tm_mday = 11; // 11 日  
info.tm_hour = 11; // 11 时  
info.tm_min = 47; // 47 分  
info.tm_sec = 12; // 12 分
```

1.119.3 time: 取得系统当前时间

time 函数可以取得系统当前时间, 返回值是一个秒值。例如,

```
time_t now = time (NULL);
```

为什么一个整数秒值就可以表示当前的时间呢? 是这么规定的, time 函数返回的是自 1970-1-1 00:00:00 这个时间点开始至当前时刻的时间差。它是一个比较大的整数, 例如 1418270153 表示的是 2014-12-11 11:55:53 这个时刻。

利用 time 函数时可以计算程序运行了多少时间, 如下面的代码:

```
time_t start = time (NULL);  
//... DoSomething ...  
time_t end = time (NULL);  
printf("Time cost : %d seconds", end - start );
```

当然这个 DoSomething 要运行相当时间才行，因为 time 的粒度较大，返回是秒值。如果你的 DoSomething 只耗费了几毫秒的话，那么用 time 根本无法衡量。不过，可以成倍的量化一下，比较，将 DoSomething 连续运动 10000 次，看需要的总时间，然后再平均一下得出单次需要的时间。

```
time_t start = time (NULL);  
for( int i=0; i< 10000; i++)  
{  
    //... DoSomething ...  
}  
time_t end = time (NULL);  
int avg = (end - start ) / 10000;
```

1.119.4 localtime: 取得年月日时分秒

虽然用一个 time_t 整数来表示当前系统时间是比较方便的，但有时候还是希望能转化成年月日时分秒的形式来显示，毕竟肉眼无法直接看一个 time_t 值到底是哪一个日期。

localtime 函数可以将 time_t 所表示的时间转化成年月日时分秒，例如，

////////// 例 CH16_C2 //////////

```
#include <stdio.h>  
#include <time.h>  
int main()  
{  
    time_t t = time(NULL);  
    tm info = *localtime(&t);  
    printf("%04d-%02d-%02d %02d:%02d:%02d \n",  
        info.tm_year + 1900,  
        info.tm_mon + 1,  
        info.tm_mday,  
        info.tm_hour,  
        info.tm_min,  
        info.tm_sec);  
  
    return 0;  
}
```

localtime 的返回值是 tm*类型，应该用一个 tm 变量将内容保存起来。

事实上，用 time_t 来记录时间更方便，只用一个整数（占 4 个字节）就表达了日期和时间信息。在保存和传输的时候，应该尽量用 time_t 类型。只是在最终显示的时候，用 localtime 转成人类易读的 yyyy-mm-dd HH:MM:SS 格式。

1.119.5 mktime：构造时间

当已知了年月日时分秒信息，可以用 mktime 换算成 time_t 值。例如，把 2014-12-11 11:47:12 转成 time_t 值，

```
// 2014-12-11 11:47:12

tm info;

info.tm_year = 2014 - 1900; // 2014 年

info.tm_mon = 12 - 1; // 12 月

info.tm_mday = 11; // 11 日

info.tm_hour = 11; // 11 时

info.tm_min = 47; // 47 分

info.tm_sec = 12; // 12 分

time_t t = mktime(&info);

if( t < 0)

{

    // 输入值不正确或不完整, 转换失败

}
```

读者可能会问，为什么 tm_wday 和 tm_yday 没有赋值呢？是的，只要把年月日时分秒这 6 个字段赋值了，对于 mktime 来说输入信息就足够了，mktime 函数在返回 time_t 的同时会计算好 tm_wday 和 tm_yday 并回填到 info 参数里。这表明 info 同时也是输出参数。

如果输入值不正确或者不完整，则 mktime 返回-1，表示转换失败。

1.120 stdlib.h

这一节将介绍 stdlib.h 里提供的 API，下面列表其中主要的几个函数：

```
#include <stdlib.h>

double atof(const char *s);

int atoi(const char *s);
```

```
int rand(void);  
void srand(unsigned int seed);  
int system(const char *s);
```

1.120.1 atoi / atof 字符串转成数字

例如，

```
int n = atoi("1280");  
double f = atof("12.80");
```

也可以用 sscanf 来完成相同的事情，例如，

```
int n;  
double f;  
sscanf("1280", "%d", &n);  
sscanf("12.80", "%f", &f);
```

相比之下，使用 atoi 和 atof 更简洁一些。

1.120.2 rand / srand 随机数生成

在抽签、抽奖等涉及“随机事件”的应用场景中，需要随机数生成函数。由于计算机中并没有随机性和偶然性，想制造一个随机数实际上是一件比较困难的事情。要真正随机的数字，需要购置在非常昂贵的硬件，这些硬件利用某些自然科学的规则来生成随机数，因为价格昂贵，所以专业的随机数生成器只用于专业用途。我们这里介绍的适用普通 PC 机能够胜任的“伪随机数”生成函数，能够生成近似随机的数据就可以了。

rand 函数用于生成随机数，该函数返回一个整数。调用以下代码测试一下：

```
////////// 例 CH16_D1 //////////  
  
for(int i=0; i<10; i++)  
{  
    printf("%d \n", rand());  
}
```

控制台输出了 10 个完全没有规律的数字，因为完全无规律可循，所以称它为随机数。这里为了显示方便，只生成了 10 个随机数，实际上可以改成 1000 次、10000 次试试，会发现它确实是杂乱无章、完全没有规律的出现的。

那么为什么说它是“伪随机数”呢？说明它没有真正的实现“随机”。可以这么验证一下，把相同的程序反复运行数次，会发现每次程序运行输出的结果都是相同的一个序列的数字。例

如，第一次运行程序的时候输出 13435 31833 5075 19863 30565 11677 1339 4096 31105 9088 等 10 个随机数，当关闭程序再次运行时，输出的 10 个随机数还是这 10 个数。

为了解决这个问题，`stdlib.h` 里提供了 `srand` 函数。`srand` 函数用于为程序设置一个种子 (seed)，当种子不同时，程序产生的随机数序列也不同。

在 CH16_D2 中，分别调用 `srand(11)` 和 `srand(12)`，观察之后产生的随机数序列。

////////// 例 CH16_D2 //////////

```
srand(11); // 确保每次程序启动后，传入不同的 seed
for(int i=0; i<10; i++)
{
    printf("%d \n", rand());
}
```

实际上，如果不调用 `srand`，相当于默认的执行了 `srand(1)`，所以先前每次输出的都是 `srand(1)` 的结果。

那么，如何确保每次程序启动时，能传入不同的 `seed` 呢。通常的办法是传入当前的时间，每次每程序的启动时间是不同的，也就确保使用了不同的种子。

////////// 例 CH16_D3 //////////

```
srand(time(NULL));
for(int i=0; i<10; i++)
{
    printf("%d \n", rand());
}
```

注：`srand` 只需要 `main()` 启动时运行一次就够了。不需要每次 `rand()` 之前都调用 `srand`。

在实际应用中，通常要对 `rand()` 结果取模操作，使结果坐落在一个区间内。

比如，要在 100 和 160 之间随机取一个数。

```
int r = rand() % 60 + 100; // 60 为区间长度，结果区间[100-159]
```

比如，要在 0.00 和 1.00 之间取一个小数

```
int r = rand() % 100; // 结果区间[0,99]
```

```
double rp = r / 100.0; // 转成小数
```

或者

```
double r = rand() / (double)RAND_MAX;
```

其中, RAND_MAX 在 VC 下的值是 32767, 在其他操作系统下可能是不同的值。

应用举例: 彩票随机下注。一种彩票叫“七星彩”, 每次生成的中奖号码是 7 个属于 0~9 之间的随机数字。那么可以使用 CH16_D4 的程序, 来随机生成一注号码。

```
////////// 例 CH16_D4 //////////  
  
#include <stdio.h>  
#include <time.h>  
#include <stdlib.h>  
  
int main()  
{  
    srand (time (NULL));  
    int code[7]; // 一注号码为 7 个数字  
    for(int i=0; i<7; i++)  
    {  
        int r = rand () % 10;  
        code[i] = r;  
    }  
    return 0;  
}
```

1.120.3 system 调用系统命令行

用 system 函数可以调用系统命令行, 在 windows 下可以执行 DOS 命令行, 在 linux 下 SHELL 命令行。比如, 删除 d:\aaa.pdf 这个文件,

```
system ("del /F /Q d:\\aaa.pdf");
```

其实原则上并不限于 DOS 命令, 所以的命令都可以运行的。例如, 调用浏览器打开一个网站,

```
system ("explorer http://www.afanihao.cn");
```

1.121 string.h

string.h 中提供了一系统内存操作函数及字符串操作函数。在学习本节之前, 一定要先学习第 15 章中, 掌握字符串的意义。

```
#include <string.h>

char *strcat(char *s1, const char *s2); // 拼接字符串

char *strchr(char *s, int c); // 查找字符

int strcmp(const char *s1, const char *s2); // 字符串比较

char *strcpy(char *s1, const char *s2); // 拷贝字符串

char *strstr(char *s1, const char *s2); // 查找子串

size_t strlen(const char *s); // 计算长度


int memcmp(const void *s1, const void *s2, size_t n); //按内存比较

void *memcpy(void *s1, const void *s2, size_t n); // 按内存拷贝

void *memmove(void *s1, const void *s2, size_t n); // 移动数据

void *memset(void *s, int c, size_t n); // 按字节填充内存
```

1.121.1 strcpy 拷贝字符串

strcpy(a,b)用于将字符串 b 拷贝到目标缓冲区，
如，

```
char buf[128];

strcpy(buf, "LiMing"); // 目标缓冲区内容拷贝为"LiMing"
```

1.121.2 strcat 拼接字符串

strcat(a,b)用于将字符串 b 拼接于字符串 a，也就是说把字符串拷贝到目标字符串的末尾。
此函数要求目标缓冲区足够大。

////////// 例 CH16_E1 //////////

```
#include <stdio.h>

#include <string.h>

int main()
{
    char a [128] = "hello";
    char b [] = "world";
    strcat(a, b);
```

```

    printf("result: %s \n", a); // "helloworld"
    return 0;
}

```

1.121.3 strcmp 比较字符串

关于字符串比较的意义在第 15 章已经讲述。strcmp(a,b)用于比较字符串，当返回为 0 时表示完全相等，小于 0 时表示 a<b，大于 0 时表示 a>b。

```
int ret = strcmp("Jack", "Jacky"); // 返回值 ret=-1，表示"Jack"<"Jacky"
```

1.121.4 strlen 求字符串长度

字符串求长度时，结束符'\0'不计算在内。例如，

```
int n = strlen("LiMing"); // 返回长度 n 为 6
```

1.121.5 strchr 查找字符

strchr(s, c)用于在字符串 s 中查找字符 c，并返回第一处匹配的位置。其返回值是 char*类型，表示匹配的位置，

////////// 例 CH16_E2 //////////

```

char* s = "LiMing";
char* p = strchr(s, 'M'); // 返回值 p 指向字符'M'的地址
if(p!= NULL)
{
    printf("find: %s \n", p);
}

```

读者要注意理解这个返回值的意思，在上例中，返回值 p 指向了'M'的地址。如果要得到目标字符'M'的索引（数组下标），可以这样，

```
int pos = (int) (p - s); // s 指向'L'的地址, p 指向 'M'的地址
```

1.121.6 strstr 查找子串

strstr(src, sub)用于在字符串 src 中查找子串 sub，返回第一处匹配的位置。和 strchr 一样，返回的是 char*指针，表示子串的起始位置，

////////// 例 CH16_E3 //////////

```

char* s = "LiMing is doing homework";
char* p = strstr(s, "ing");
if(p!= NULL)

```



```

{
    printf("find: %s\n", p);
}

```

1.121.7 memset 内存填充

memset(dst, val, n)用于设置内存的位置，将内存里的 n 个字节，全部设置为 val 的值。这里，并不区分这一块内存里装的是什么类型数据，统一按字节设置。

////////// 例 CH16_E4 //////////

```

unsigned char buf[128];
memset(buf, 0, sizeof(buf)); // 全部填充为 0
memset(buf, 0xFF, 128); // 全部填充为 0xFF
memset(buf, 0x55, 100); // 前 100 个字节填充为 0x55
memset(buf+100, 0x77, 10); // 100..109 填充为 0x77
unsigned int abc[4];
memset(abc, 0x55, sizeof(abc)); // 按字节填充为 0x55，所以 a[0]为 0x55555555;

```

1.121.8 memcpy 内存拷贝

memcpy(dst, src, n)用于从 src 拷贝 n 个字节到 dst，其中 src 和 dst 是任意类型的指针。

1.121.9 memcmp 内存比较

memcmp(a, b, n) 按字节比较两块内存 a,b 的内容。比较的规则是逐字节比较、当相等时比较下一字节，仅当 n 个字节全部相同，则返回 0。可以从下面的实现代码中理解其功能，

```

int Compare(void* a, void* b, int n)
{
    for(int i=0; i<n; i++)
    {
        unsigned char v1 = ((unsigned char*) a)[i];
        unsigned char v2 = ((unsigned char*) b)[i];
        if(v1 < v2) return -1;
        else if(v1 > v2) return 1;
    }
    return 0;
}

```

需要注意的是，按字节比较时，无论输入的 void* 中存放何种数据，统一按 unsigned char 来比较。

例如，下面的代码，

```
char a[3] = { 1, 2, 3 };
char b[3] = { 1, 2, 4 };
char c[3] = { 1, -1, 3 };
int ret;

ret = memcmp(a, b, 3); // 容易目测得到 a<b

ret = memcmp(a, c, 3); // ?
```

其中，a 和 b 的比较是容易得出结果的，但 a 与 c 的比较呢？由于要转成 unsigned char 再比较，所以 c>a。

1.121.10 memmove 移动数据

memmove(dst, src, n) 用于在内存中移动数据，将开始于 src 的 n 个字节移动到 dst 位置，这个函数的强大之处在于它允许 src 和 dst 有交迭。

例如，利用这个函数我们可以实现在字符串中插入字符，当插入字符串如果将插入点之后的所有字符后移，

```
////////// 例 CH16_E5 //////////
#include <stdio.h>
#include <string.h>
// 在第 index 位置插入字符 ch
void insert(char* buf, char ch, int index)
{
    int size = strlen(buf); // 原长度
    char* p = buf + index; // 待插入位置
    memmove(p+1, p, size-index); // 后移
    *p = ch; // 插入字符
}

int main()
{
    char str[128] = "hello";
    insert(str, 'i', 1);
```

```
    return 0;  
}
```

其中，`memmove(p+1, p, size-index)`表示要移动 `size-index` 个字节，起始位置为 `p`，目标位置为 `p+1`。显然，这两块区域是有交迭的，`memmove` 的好处就是允许内存区域交迭。

第 17 章 文件操作

在本章中，我们将学会如何在代码中将数据存储到文件，以及如何从文件中读取数据。

1.122 认识文件

说起文件，我们可能再熟悉不过了。电脑里保存着形形色色的文件，有图片文件（*.jpg，*.bmp），有视频文件（*.mp4，*.avi），有音乐文件（*.mp3），有 Word 文档（*.docx）等等。

文件的作用是为持久化存储数据。所谓持久化，是指当我们关闭电脑电源后数据依次存在；再次打开电脑时，还可以重新加载显示这些数据。

文件具有一些属性，以我们在项目中使用的源文件 `main.cpp` 为例：

- 文件名：`main.cpp`
- 路径，表示该文件存储的位置，例如，`D:\Cpp\Hello\`
- 长度，表示该文件内存储了多少字节的数据
- 内容，该文件中存储了什么东西
- 权限，只读 / 读写，当文件为“只读”时，该文件不允许写入。

在计算机世界里，所有的数据最终都是以二进制的数字存储。和内存的存储表示类似，文件的数据最终也都是用数字表示的。

例如，我们在 C:\盘下新建一个文本文件 `abc.txt`，然后在里面输入几个字：`hello`，保存这个文件。然后我们用 UltraEdit（一款编辑软件）打开它，在 UltraEdit 界面下按 `Ctrl+H` 键切换到十六进制的显示模式，如图 17-1 所示。



图 17-1 在 UltraEdit 中按十六进制查看

可以发现，当十六进制显示时，这个文件里的内容和在内存中显示的一致。也就是说，文件存储和内存存储是一个道理，都是以数字形式存储的。

1.123 保存数据

我们使用 ANSI C 中的 `stdio.h` 里的相关函数来进行文件读写操作。

保存数据到文件的步骤：

- ① `fopen`: 打开文件
- ② `fwrite`: 写入数据
- ③ `fclose`: 关闭文件

在示例 CH17_A1 中，首先用 `fopen` 打开一个文件，然后把几字节的数据用 `fwrite` 写进去，最后用 `fclose` 关闭这个文件。

```
////////// 例 CH17_A1 ////////////  
  
#include <stdio.h>  
  
int main()  
{  
    const char* filename = "c:/aaa.txt";  
  
    FILE* fp = fopen (filename, "wb" ); // 打开文件  
  
    if(fp == NULL)  
    {  
        printf("failed to open file!\n");  
        return -1;  
    }  
  
    char buf[] = "hello";  
  
    int n = fwrite (buf, 1, 5, fp); // 向文件中写入 5 个字节的数据  
  
    fclose (fp); // 关闭文件  
  
    return 0;  
}
```

运行这个程序，然后用 UltraEdit 打开 c:\盘下的 `aaa.txt`，观察里面的内容。

1.123.1 fopen 打开文件

`fopen` 函数用于打开文件，得到一个 `FILE*` 指针，该指针指代该文件，后续的 `fwrite` / `fclose` 等函数都需要传入这个文件指针。

`fopen` 的函数原型为：

```
FILE* fopen(const char *filename, const char *mode);
```

其中，

Filename: 表示要打开的文件路径，例如 `c:/test/abc.txt`，表示在 `c:\test\` 目录下的 `abc.txt`

mode: 固定使用 `"wb"` (`w` 表示 `write`, `b` 表示 `binary`)

用法示例：

```
FILE* fp = fopen("c:/aaa.txt", "wb" );  
  
if(fp == NULL)
```

```
{  
    printf("文件打开失败!\n");  
}
```

1.123.2 fclose 关闭文件

当数据写入完毕，该文件指针不再被使用时，要及时调用 `fclose` 函数来关闭该文件。其原型为，

```
int fclose(FILE* stream);
```

参数: `stream` 就是前面 `fopen` 的返回值

用法示例:

```
fclose(fp);
```

1.123.3 fwrite 写入数据

`fwrite` 用于文件中写入数据。什么叫数据？在 C++ 中，数据就是一段内存里面的值。所以，你只需要指定这段内存的首地址，以及数据的长度（字节）数，就可以将这些字节的数据写入文件。

`fwrite` 的函数原型为：

```
size_t fwrite(const void *buf, size_t size, size_t nelem, FILE *stream);
```

参数：

`stream`，就是前面 `fopen` 的返回值

`buf`，要写入的数据的首地址

`size`，总是传 1

`nelem`，数据长度（字节数）

用法示例：

```
char buf[] = "hello";
```

```
fwrite(buf, 1, 5, fp);
```

1.124 读取数据

读取数据，就是将曾经写入的那些字节再取出来，放到内存缓冲区里。

读取数据也分为三步：

① `fopen`: 打开文件

② fread: 读取数据

③ fclose: 关闭文件

在示例 CH17_B1 中，首先用 `fopen` 打开文件，得到一个可供操作的文件指针。然后用 `fread` 来从文件中读取数据到内存缓冲区。操作完毕，最后关闭文件。

////////// 例 CH17_B1 //////////

```
#include <stdio.h>

int main()
{
    const char* filename = "c:/aaa.txt";
    FILE* fp = fopen (filename, "rb" );
    if(fp == NULL)
    {
        printf("failed to open file!\n");
        return -1;
    }

    char buf[128] ;
    int n = fread (buf, 1, 128, fp);

    fclose (fp);

    return 0;
}
```

需要注意的是，当需要从文件中读取数据时，`fopen` 传入的第 2 个参数是"rb"，而不是"wb"。也就是说，当我们打算写入数据时，就用"wb"方式来打开文件；当我们打算读取数据时，就用"rb"方式来打开文件。其中，w 代表 write，r 代表 read，b 代表 binary。

注：我们一般打开文件要么读，要么写，一般不需要既读又写。

1.124.1 fread 读取数据

使得 `fread` 函数从文件中读取数据，其函数原型为，

```
size_t fread(void *buf, // 存储到目标内存地址

    size_t size, // 设为 1

    size_t nelem, // 最多读取多个字节

    FILE *stream);
```

返回值：实际读取到的字节的个数

参数：

stream: 前面 fopen 的返回值

buf: 内存缓冲区，用于存储数据的内存位置

size: 恒为 1

nelem: 最多读取多少个字节

返回值：实际读取的字节数。如果返回-1，表示读取失败。

用法示例：

```
char buf[128];  
  
int n = fread (buf, 1, 128, fp);
```

显然，如果文件中的数据不足 128 字节，则返回值 **n** 就是实际的字节数。如果文件中字节数超 128 字节，那这次操作只读取 128 字节。

这意味着，如果文件很长，那我们一般是反复调用 **fread** 来读取文件内容，直到读取完毕。比如，文件的内容有十几兆（1M=1,000,000）字节的话，我们也许会每次读取 1024 字节，直到读完。类似使用下面的代码：

```
char buf[1024]; // 设立一个缓冲区，用于存放读取到的字节数据  
  
while(! feof (fp))  
{  
  
    int n = fread (buf, 1, 1024, fp); // 每次读取 1024 字节  
  
    if( n == 0) // 文件系统忙碌(busy), 或者到达文件末尾(EOF)  
        continue;  
  
    if( n < 0) // 出错  
        break;  
  
    if( n > 0) // 处理数据  
    {  
        }  
}
```


1.125 数据的存储格式

数据，无论它要表达的是什么信息，它本身总是有由一串字节组成的。比如，一个整数可以由 `int` 表示，共 4 个字节。一个字符串 `helloworld`，占了 10 个字节。

数据可以用多种格式存储到文件。比如，一张图片，可以按 `bmp` 格式存储到文件，也可以按 `jpg` 格式存储到文件。尽管使用的具体存储格式有差异，但它们遵守了一个共同的原则：可以存进去，也可以读出来。这里所说的“读出来”，是指读出数据并恢复原状（能将数据恢复显示为一张图片）。

又比如，我们要保存两个整型变量 `x,y`，假设它们是一个点 `Point` 的坐标。我们可以几种方式来存储它。如果这种方式“能写入能读出”，那它就是有效的；如果只能写入，但不能读取还原信息，那它就是无效的存储格式。

第（1）种方式：

////////// 例 CH17_C1 //////////

```
int x = 100;
int y = 200;

fwrite(&x, 1, 4, fp); // x: 写入 4 个字节
fwrite(&y, 1, 4, fp); // y: 写入 4 个字节
```

当以这种方式写入时，一共写入 8 个字节。我们可以用相应的代码，从文件中读取数据，并恢复为 `x,y` 坐标。

```
int x, y;

fread(&x, 1, 4, fp);
fread(&y, 1, 4, fp);
```

这意味着，这种存储格式满足“可以存进去，也可以读出来”的原则，是一种有效的存储格式。

第（2）种方式：

////////// 例 CH17_C2 //////////

```
int x = 100;
int y = 200;
char buf[128];

sprintf(buf, "x=%d,y=%d", x, y);
fwrite(buf, 1, strlen(buf), fp);
```

这种方式是把一个字符串"x=100,y=200"写入文件，一共 11 个字节。在读取它时，如何解析读到的字节、还原为 x,y 坐标信息呢？最基本的办法，我们可以采用第 15 章中介绍的字符串解析的办法，利用分隔符（,和=）来提取出信息。由于两个字段都是整数，也可以直接使用 sscanf 来提取。

```
char buf[128];

int n = fread(buf, 1, 128, fp);

if(n > 0)
{
    buf[n] = 0;

    int x, y;

    sscanf(buf, "x=%d,y=%d", &x, &y); // 解析出 x,y 的值

    printf("loading ... x=%d, y=%d \n", x, y);
}
```

不管它是否困难，我们是可以把数据再解析还原的，因此这也是一种有效的存储方式。

第（3）种方式，

```
int x = 123;

int y = 456;

char buf[128];

sprintf(buf, "%d%d", x, y);

fwrite(buf, 1, strlen(buf), fp);
```

以这种方式存储时，存入字符串"123456"共 6 个字节，但是由于缺少分隔符，我们无法识别这两个数字，是没有办法还原数据的，因此它不是一种有效的存储方式。比如，它有可能是（12，3456），也有可能是（1234，56），这种存储是失败的，缺少足够的信息，无法还原。

1.126 存储格式：按字节存储

按字节存储是一种简单有效的方式。我们已经的数据类型，要么是整数，要么是浮点数，要么是结构体，要么是一个字符数组。在存储它们时，只要把它们的内容直接存到文件里就可以了，在读取时才原封不动的读取出来。

例如，

（1）存储 char 型整数

```
char ch = 12;

fwrite(&ch, 1, 1, fp); // 存
```

```
fread(&ch, 1, 1, fp); // 取
```

(2) 存储 int 型整数

```
int n = 12;  
fwrite(&n, 1, sizeof(int), fp); // 存  
fread(&n, 1, sizeof(int), fp); // 取
```

(3) 存储 double 型小数

```
double val = 123.345;  
fwrite(&val, 1, sizeof(val), fp); // 存  
fread(&val, 1, sizeof(val), fp); // 取
```

(4) 存储结构体数据

```
Object obj = { 123 };  
fwrite(&obj, 1, sizeof(obj), fp); // 存  
fread(&obj, 1, sizeof(obj), fp); // 取
```

在存储结构体时，由于结构体间可能在填充，会造成一定程度的空间浪费。（关于结构体间的 padding，参见第 10 章）。如果需要节省空间，可以把结构体内的每个字段拿出来逐个存储一下。

(5) 存储字符串

```
char name[32] = "shaofa";  
fwrite(name, 1, 32, fp); // 存  
fread(name, 1, 32, fp); // 取
```

当然，这样存储字符串时，我们实际存储了 32 个字节，但有效的数据实是前 6 个字节，存在一定程度的浪费。也许它不是一种优秀的存储办法，但它至少是一个可行的办法。

1.126.1 指针指向的对象

当变量类型为指针时，指针本身并没有必要存储（指针本身只是一个地址），我们要存储的是它指向的那个对象。

比如 Car 表示一个汽车，Citizen 表示一个市民，它们的定义如下，

////////// 例 CH17_D1 //////////

```
struct Car
{
    char maker[32]; // 制造商
    int price; // 价格
};
struct Citizen
{
    char name[32]; // 名字
    int deposit; // 存款
    Car* car; // NULL 时表示没车
};
```

那么，在存储一个 Citizen 对象的信息时，在保存它的 car 的信息时需要注意。由于它的 car 可能为 NULL，也有可能指向一个 Car 对象，所以我们先行写入一个字节，用于标识他到底有没有车。

```
if(who.car != NULL)
{
    fwrite("Y", 1, 1, fp); // 存入一个字节'Y'
    fwrite(who.car->maker, 1, 32, fp);
    fwrite(&who.car->price, 1, 4, fp);
}
else
{
    fwrite("N", 1, 1, fp); // 存入一个字节'N'
}
```

在读取时，先读取这个标识字节。如果为'Y'，表明下面就是 Car 的数据。如果为'N'，表示下面没有 Car 的数据。这样的保存方式可以有效的节省空间。

```
char has = 'N';
fread(&has, 1, 1, fp);
if(has == 'Y') // 先看有没有 car 的信息，如果为'Y'再继承读取
```

```

{
    Car* car = (Car*) malloc(sizeof(Car));
    fread(car->maker, 1, 32, fp);
    fread(&car->price, 1, 4, fp);
}

```

1.126.2 Run-Length Encoding 存储

Run-Length Encoding (RLE)，是一种常见的编码技术，我们可以借用这种技术来存储字符串信息。

前面说过，对字符串的存储来说，我们可以按固定长度存储，例如，

```

char name[32] = "shaofa";
fwrite(name, 1, 32, fp); // 存
fread(name, 1, 32, fp); // 取

```

显然地，这样的存储方式存在的很大的缺陷。当字符串太短时（"shaofa"只有 6 个字节），则浪费了较多空间。借用 RLE 的类似技术，可以减少空间的浪费。

我们按这种方式存储字符串：

[Length] [Content]

其中，

Length: 有 2 字节存储，表示接下来的 Content 有多少字节

Content: 要存储的内容

按这种存储方式，写入时：

////////// CH17_D2 //////////

```

char name[32] = "shaofa";
unsigned short length = strlen(name);
fwrite(&length, 1, 2, fp); // 写入 2 个字节，表示字符串的长度
fwrite(&name, 1, length, fp); // 写入字符串的有效字节

```

读取时，

```

char name[32];
unsigned short length ;
fread(&length, 1, 2, fp); // 先读 2 个字节，得到字符串的长度
fread(&name, 1, length, fp); // 读出指定长度的有效字节

```

```
name[length] = 0; // 加上结束符
```

1.127 存储格式：文本化存储

当数据量比较少时，可以把数据格式化为文本的形式存储。我们常见的各种配置文件，一般都是以文本形式存储的，例如*.xml，*.cfg。比如，需要配置文件中存放 IP 地址和端口号，那么可以按 XML 格式存储：

```
<?xml version="1.0" encoding="gb2312" ?>

<Root>

  <ip> 192.168.10.60 </ip>

  <port> 8080 </port>

</Root>
```

当数据以文本化存储时有两个缺点，一方面显著的增加了存储空间，另一方面构造和解析工作都变得麻烦了。以上述 XML 为例，在此 XML 中是可以解析出 IP 和 Port 这两个字段，但解析的难度还是不小的，比起按字节存储就麻烦多了。

但文本化存储也有显著的优点，就是这个配置文件可以直接用记事本打开，其内容是直观的，是可以直接人手编辑的。追求内容的直观性，就是人们采用文本化存储的一个原因。然而要再次强调的是，只有数据量不大的时候才适用用文本化存储。

当我们选择文本化存储数据时，通常的方案是按行存储的。在一段文字之后，用"\n"作为行尾符分隔开。例如，我们可以用以下方式来存储前面的 IP 和 Port 两个字段。我们注意到，IP 是一个字符串，而 Port 是一个整数。

```
ip=192.168.1.100 (换行)

port=8080(换行)
```

这就是一经常被采用的按行存储的配置文件的格式。每一行保存一个字段，行末符为"\n"。每一行为 key=value 的格式，key 是字段的名称，value 是字段的值。

1.127.1 fprintf 按行格式化写入

当选择按行存储时，我们可以使用 fprintf 函数，这个函数可以像 printf 一样直接格式化文本并输出到文件中。

例如，

```
////////// 例 CH17_E1 //////////

char ip[]="192.168.1.100";

int port = 8080;

fprintf (fp, "ip=%s\n", ip);
```

```
fprintf (fp, "port=%d\n", port);
```

`fprintf` 的参数的用法和 `printf` 类似，区别是第一个参数是文件指针，指向待输出的文件。

1.127.2 fgets 按行读取

如果一个文件是按行存储的，那我们可以使得 `fgets` 函数来读取每一行。在例 CH17_D1 中，按行读取配置文件，每次读取一行，直到文件结束。然后对读到的每一行进行解析即可。

////////// 例 CH17_E1 //////////

```
char buf[512]; // 缓冲区，用于存储一行的内容

while(! feof(fp))
{
    char* line = fgets(buf, 512, fp);
    if(line)
    {
        // 解析这一行
    }
}
```

需要注意的是，`fgets` 这个函数每次读取一行（即读到 `\n` 时算一行），把一行的内容存到缓冲区 `buf` 中，最后一个 `\n` 字符也是被存到了 `buf` 里了。

每一行的内容是 `key=value` 格式，末尾还有一个 `\n` 字符。如何从这一行中解析出 `key` 和 `value`？可以参考第 15 章中的办法，具体的解析代码留给读者来完成。

1.128 文件的随机访问

在描述一个文件的可访问属性时，有两个术语：

顺序访问（Sequential Access）：按顺序访问，不能跳跃

随机访问（Random Access）：随意跳到一个位置访问

举例来说，当在看一个 MP4 电影时，默认情况下随着时间的推移进度条也在走动，这就是顺序访问。当你拖动进度条滑块时，可以跳到影响的任意位置，此时进行的就是随机访问。

注：文件是不是支持“随机访问”，是由物理存储和系统驱动决定的。一般来说，我们使用的硬盘都是支持“随机访问”的。

注：本篇只讨论在“读”文件时候的随机访问技术，也就是说，文件中的数据已经齐备，现在使用随机访问技术来读取任意位置处的数据。

1.128.1 fseek 随机访问

在 ANSI C 函数里，使用 `fseek` 函数可以实现对文件的随机访问。其函数原型如下，

```
int fseek(FILE *stream, long offset, int mode);
```

参数列表：

`stream`：文件指针

`offset`：表示偏移值，是一个整数

`mode`：使用的方式，指定起点位置（可以取 `SEEK_SET`, `SEEK_END`, `SEEK_CUR`）

返回值：0，操作成功；-1，操作失败

用法示例：

```
fseek(fp, 100, SEEK_SET); //跳到第 100 个字节的位置
```

```
fseek(fp, 100, SEEK_END); //跳到倒数 100 字节的位置
```

```
fseek(fp, -100, SEEK_CUR); //跳到当前位置往前 100 个字节
```

```
fseek(fp, 100, SEEK_CUR); //跳到当前位置往后 100 个字节
```

什么叫当前位置呢？

1.128.2 文件位置指示器

文件位置指示器（file-offset indicator）：每个被打开的文件对象 `FILE*`，其数据结构里都有一个位置指示器，表示当前的读/写位置。（即，当前位置到文件头的距离）

（1）当 `fopen` 打开文件时，位置指示器的值为 0

（2）当 `fread` 读取字节时，位置指示器的值会增加相应的字节数

例如，读取 128 个字节，则位置指示器的值就增加 128，继续 `fread`，则继续增加

（3）当 `fseek` 时，会调整位置指示器的值

例如，

```
fseek(fp, 100, SEEK_SET); //则位置指示器的值被设定为 100
```

```
fseek(fp, 100, SEEK_END); //则指示器的值为 filesize-100，倒数 100 字节的位置
```

1.128.3 随机访问示例

举例：假设一个文件中按字节保存了 100 个 Object 对象的数据。

```
////////// 例 CH17_F1 //////////
```

```
struct Object
```

```
{
```



```

    int id;

    int value;

};

```

下面使用随机访问的技术，直接读取文件中第 57 个和第 82 个的数据，

////////// 例 CH17_F1 //////////

```

    Object obj = {0};

    // 跳到指定位置

    fseek(fp, 56 * sizeof(Object) , SEEK_SET);

    fread(&obj, 1, sizeof(Object), fp);

    printf("id=%d, value=%d \n", obj.id, obj.value);

    // 跳到指定位置

    fseek(fp, 81 * sizeof(Object) , SEEK_SET);

    fread(&obj, 1, sizeof(Object), fp);

    printf("id=%d, value=%d \n", obj.id, obj.value);

```

1.128.4 fseek 的物理限制

fseek 在物理上是受到限制的，不提倡频繁调用。在物理上，硬盘、U 盘等外部存储器属于“慢速存储设备”，不提倡频繁的读写。否则有可能导致：

- (1) 速度慢、效率低
- (2) 降低设备的使用寿命

而在 fseek 的时候，每一次 fseek 都要移动物理“磁头”，因而不能频繁的 fseek，以免损坏物理设备。也许随着未来技术的革新，采用最新的技术的硬盘会降低 fseek 的危害，但是目前来说 fseek 还是不宜频繁调用的。

通常情况下，我们会每次读取一大块（比如 8096 字节）数据，然后在内存里处理数据。尽量减少 fseek 的调用，是一个编程中要注意的问题。

1.128.5 文件被重复打开的情况

在 C/C++ 里，允许一个文件被同时打开多次，而每个 FILE* 指针都可以被独立的读取，在逻辑上是不受影响的。例如，

```

例如，fp1 和 fp2 操作的是同一个文件，

FILE* fp1 = fopen("abc.xyz", "rb");

FILE* fp2 = fopen("abc.xyz", "rb");

```

由于在 FILE* 结构中各自记录了自己的读取位置，所以它们“在逻辑上”是不受影响。然而，在物理上，同时读一个文件时必然是有着影响的，至少速度会变慢，效率会变低。

1.129*文件打开模式

当我们用 ANSI C 函数库中的 `fopen` 函数来打开文件时，`fopen` 的第二个参数表示打开文件的模式。我们常用的模式有 "rb", "wb", "ab"。

"rb": 读模式。当读一个文件时使用。如果该文件不存在，则 `fopen` 返回 NULL。

"wb": 写模式。在写一个文件时使用。

(1) 目录不存在，则 `fopen` 返回 NULL

例如，`fopen("c:/test/abc.txt", "wb")` 打算在 `c:\test\` 目录下创建一个 `abc.txt`，但是文件系统中不存在 `c:\test\` 这个目录，则 `fopen` 失败。

(2) 目录存在，且该目录下没有该文件：则创建一个新文件，返回文件指针

(3) 目录存在，且该目录下已经有这个文件，则打开这个文件，并把文件内容清空，返回文件指针。

(4) 目录存在，且该目录下已经有这个文件，但这个文件是只读的：则文件打开失败，返回 NULL 指针。

除此之外，还有一个 "ab" 模式可能会被用到，其中 `a` 代表 `append`。它表示是附加模式。和 "wb" 的作用类似，在写一个文件时要吧使用 "ab" 模式，表示打开文件但不清空里面的内容。

最后，还有一些不推荐使用的模式，如 "wt" "rt" "at"，本教程中建议永远不要使用带 "t" 的模式。原因是：① `b` 模式已经满足需要，且不存在问题 ② `t` 模式存在较多问题。比如，在 `fseek` 时 `t` 模式会给你带来困惑。又比如，在 windows 和 linux 下 `t` 模式的表现不一样。总之一句话：永远不要使用 `t` 模式。

1.130*常见问题

(1) `fread` 和 `fwrite` 的参数倒底是什么意思？

```
size_t fwrite(const void *buf, size_t size, size_t nelem, FILE *stream);
```

```
size_t fread(void *buf, size_t size, size_t nelem, FILE *stream);
```

按照这个函数设计的原始意图，`fwrite` 和 `fread` 是按块写入的，其中 `size` 表示每一块的大小，而 `nelem` 表示的是一共有多少块。而返回值则是写入/读取的块的个数。

比如，我们把 100 个 `Object` 数据写入文件时，每一块数据的大小就是 `sizeof(Object)`，一共有 100 块，所以，

```
Object objs[100];
```

```
fwrite ( objs, sizeof(Object), 100, fp);
```

然而，我们可以把 `size` 参数设为 1，把 `nelem` 参数表示总共有多少字节，这样在效果上是完全一样的。这样，`fread/fwrite` 的返回值就表示实际读取/写入的字节数。

```
fwrite(objs, 1, 100*sizeof(Object), fp);
```

不管怎么样，在现代操作系统中，存储设备上每一块数据的大小是由驱动层决定的。你在应用里指定每一块多大并不会对底层的块大小有影响。

在本教程中，建议将第 2 个参数总是设置为 1，目的是让 `fread/fwrite` 的返回值的单位是字节。这相当于对 ANSI C 的函数接口做了一个优化，这样做不存在任何劣势。

第 18 章 多文件项目及编译过程

本章介绍如何在 C++ 项目中使用多个源文件和头文件，以及项目的编译过程。

1.131 extern

虽然在前面各章节的示例中都是只由一个单独的 main.cpp 文件完成，但实际上一个 C++ 项目可以由多个 cpp 文件构成的。我们可以在一个 cpp 文件中访问在另一个 cpp 中定义全局变量和函数。

1.131.1 extern 声明全局函数

设我们的项目里有 A.cpp 和 B.cpp，那么我们可以在 A.cpp 文件中调用在 B.cpp 里定义的函数。

示例：新建一个 VC 的项目，向里面添加两个文件 other.cpp 和 main.cpp，代码内容如下面所示。

```
////////// 例 CH18_A1 : other.cpp //////////
```

```
double get_area(double r)
{
    return 3.14 * r * r;
}
```

```
////////// 例 CH18_A1 : main.cpp //////////
```

```
#include <stdio.h>

extern double get_area(double r); // 用 extern 声明一个外部函数

int main()
{
    double r = 1.24;

    double area = get_area(r); // 调用外部的函数

    printf("result: %.3lf\n", area);

    return 0;
}
```

函数 get_area 的定义在 other.cpp 中，我们想在 main.cpp 中调用这个函数。那么，就必须在 main.cpp 里用 extern 声明这个函数，写法如下：

```
extern double get_area(double r);
```

以关键字 `extern` 修饰，后面加上函数的原型（函数原型的写法见第 8 章）。其中，关键词 `extern`（external，外部的），它不仅可以声明一个外部的函数，还可以声明一个外部的全局变量。**所谓外部，就是指在这个项目中的其他文件里。**它的作用是告诉编译器某个全局函数定义在外部，也就是说定义在项目的某个文件里。

注：在声明全局函数时，关键字 `extern` 是可以省略不写的。

1.131.2 extern 声明全局变量

我们也可以在 A.cpp 里访问 B.cpp 里全局变量。在示例 CH18_A2 中，other.cpp 里定义了一个全局变量 `number`，在 main.cpp 里可以访问它。那么自然地，需要在 main.cpp 里声明这个变量。

```
////////// 例 CH18_A2 : other.cpp //////////
```

```
int number = 0; // 定义一个全局变量
```

```
////////// 例 CH18_A2 : main.cpp //////////
```

```
#include <stdio.h>
```

```
extern int number; // 声明一个全局变量
```

```
int main()
```

```
{
```

```
    number = 22;
```

```
    printf("number: %d \n", number);
```

```
    return 0;
```

```
}
```

我们看到，在 main.cpp 里用 `extern` 声明了一个全局变量 `number`。也就是说，想要使用别的 cpp 里的全局变量，那么就得先声明它。

要学会区分变量的声明和定义。声明变量的时候，前面要加上关键字 `extern`，后面不能加初始值。定义变量的时候，可以有初始值。

以下为变量声明语句（Declaration）：

```
extern int a;
```

```
extern double b;
```

```
extern float numbers[5];
```

以下为变量定义语句 Definition

```
int a = 10;
```

```
double b;
```

```
float numbers[5] = {1.0, 1.1};
```

注：在声明变量的时候不能加初始值，所以下面的声明语句都是错误的。

```
extern int a = 10; // 语法错！不能加初始值！
```

```
extern float numbers[5] = {1.0, 1.1}; // 语法错！不能加初始值！
```

注：在声明全局变量时，必须在前面加上关键字 **extern**，不能省略。

1.131.3 深入理解 extern

extern 的作用是通知编译器，在本 **cpp** 中要用到某个符号，这个符号可能不在本 **cpp** 中定义。它表示在某个 **cpp** 文件中，存在这么一个函数/全局变量。需要明白的是，这个符号可以在别的 **cpp** 中定义，也可以在本 **cpp** 中定义。

在示例 CH18_A3 中，在 **main.cpp** 的前面两行声明全局符号 **abc** 和 **Print()**，它们的定义就在本 **cpp** 里，这样是没有问题的。

```
////////// 例 CH18_A3 : main.cpp //////////
```

```
#include <stdio.h>
```

```
extern int abc; // 声明一个全局变量
```

```
extern void Print(int); // 声明一个全局函数
```

```
int main()
```

```
{
```

```
    abc = 11;
```

```
    Print(abc);
```

```
    return 0;
```

```
}
```

```
int abc = 0; // 全局变量的定义
```

```
void Print(int val)
```

```
{
```

```
    printf("value: %d \n", val);
```

```
}
```

1.132 多文件项目的生成

C++项目的生成过程主要分为两步：编译和链接。在 VC 中，我们按 F7 或从菜单“生成解决方案”时，它是自动的帮我们完成了这两步，并把中间的一些信息显示到“输出”窗口中。

如果代码中有语法错误，则编译过程终止，并在“输出窗口”提示错误的位置（文件名、行号、错误原因）。如果没有语法错误，那么继续进行链接过程。如果链接成功，则生成一个可执行文件*.exe。

1.132.1 第一阶段：编译

编译（Compile），这一阶段的任务是处理每个 cpp 文件，将 cpp 文件中的代码转换为中间文件。在 VC 中，中间文件的后缀是*.obj，可以在输出的 Debug 目录中找到这个中间文件。

假设项目中存在多个 cpp 文件：A.cpp, B.cpp,

则，

A.cpp -> A.obj

B.cpp -> B.obj

....

如果某个 cpp 文件中有语句错误，则整个生成动作以失败终止。

我们发现，在编译的过程各个 cpp 文件都不区分顺序的，谁先谁后都一样。只要声明了一个函数为 extern，就可以调用它。编译器并不检查是否真得存在这个符号。

事实上，全局符号的声明仅在于协助编译器进行类型检查。比如说，如果你没有声明一个函数，那编译器如何检查你的函数调用是否写对了呢（参数列表是否匹配）？

1.132.2 第二阶段：链接

如果前一阶段编译成功，则进行到链接过程。此过程的作用是将各个 obj 文件综合在一起，生成可执行程序。

A.obj, B.obj, ... -> Test.exe

在链接阶段，编译器会检查所有 extern 的符号是否真的存在。

如果你使用了一个 extern 的符号，但这个符号却没有定义，在所有的 cpp 中都没有找到它的定义。那么编译器会给你报告一个 undefined reference 的错误（未定义的符号）。示例 CH18_B1 展示了这个错误（main.cpp 中想调用 Print 函数，但整个项目中没有这个函数的定义）：

```
////////// CH18_B1 : other.cpp //////////
```

```

#include <stdio.h>

void Print2(int val) // 有一个函数： 名字是 Print2， 不是 Print
{
    printf("value: %d \n", val);
}

////////// CH18_B1 : main.cpp //////////

#include <stdio.h>

extern void Print(int); // 声明一个全局函数

int main()
{
    Print(123);

    return 0;
}

```

执行项目的生成，编译器的输出窗口显示：

1>正在编译...

1>main.cpp

1>other.cpp

1>正在生成代码...

1>正在编译资源清单...

1>Microsoft (R) Windows (R) Resource Compiler Version 6.0.5724.0

1>Copyright (C) Microsoft Corporation. All rights reserved.

1>正在链接...

1>main.obj : error LNK2019: 无法解析的外部符号 "void __cdecl Print(int)"
(?Print@@YAXH@Z)，该符号在函数 _main 中被引用

1>D:\Cpp\Hello\Debug\Hello.exe : fatal error LNK1120: 1 个无法解析的外部命令

其中，我们发现在“编译”阶段并没有报告错误，但在“链接”阶段报错：无法解析的外部符号。

如果同一个函数在整个项目中有多处定义，那么也是一个错误，在链接阶段会报错。在示例 CH18_B2 中，在 main.cpp 和 other.cpp 里都定义了一个相同的函数 Print（名字相同、参数列表相同，参见第八章）。

//////////例 CH18_B2 : other.cpp //////////


```
#include <stdio.h>

void Print(int val)
{
    printf("value: %d \n", val);
}

//////////例 CH18_B2 : main.cpp //////////
```

```
#include <stdio.h>

extern void Print(int); // 声明一个全局函数

int main()
{
    Print(123);
    return 0;
}

void Print(int val)
{
    printf("value: %d \n", val);
}
```

执行项目的生成，编译器的输出窗口显示：

1>正在编译...

1>main.cpp

1>other.cpp

1>正在生成代码...

1>正在编译资源清单...

1>Microsoft (R) Windows (R) Resource Compiler Version 6.0.5724.0

1>Copyright (C) Microsoft Corporation. All rights reserved.

1>正在链接...

1>other.obj : error LNK2005: "void __cdecl Print(int)" (?Print@@YAXH@Z) 已经在 main.obj 中定义

1>D:\Cpp\Hello\Debug\Hello.exe : fatal error LNK1169: 找到一个或多个多重定义的符号

可以发现，在“编译”阶段仍然是没有错误的，但在“链接”阶段报错：符号 **Print** 被重复定义。

我们应该避免符号重定义的情况。这会编译器带来困惑：你有两处定义，它应该使用哪一处定义呢？这显然是一个人为的错误，应该修正，删除重复的定义。

1.132.3 *用伪代码表示整个过程

可以用一段伪代码表示编译和链接的过程，(并非真正的代码，仅用于解释说明)

```
int build ()
{
    /***** Comile *****/

    for ( file : *.cpp ) // 处理项目中所有的 cpp 文件
    {
        if ( compile ( file ) < 0 ) // 编译 cpp 文件
            return -1; // 编译失败，中止编译
        else
            create *.obj // 编译成功，生成中间文件 *.obj
    }

    /***** Link *****/

    if( link ( *.obj ) < 0 ) // 把所有*.obj 链接在一起
        return -1;
    else
        create executable // 生成可执行文件
    return 0;
}
```

1.132.4 全量编译与增量编译

全量编译，是指将所有的 cpp 文件重新编译一下。在 VC 中，执行“重新生成解决方案 (Rebuild)”菜单时，进行的就是全量编译。

增量编译，是指只对有改动的文件进行编译。当执行“生成解决方 (Build)”时，执行的是增量编译，这也是大多数编译器默认的动作。

1.133 头文件#include 指令

头文件 (Header File)，其后缀名一般为 .h。相应地，我们把.cpp 文件称为“源文件”。

1.133.1 为什么需要头文件

先阐述一下我们遇到的问题。这些问题的存在，就是头文件这种语法被发明的原因。

考虑我们有两个文件 `main.cpp` 和 `other.cpp`，其中 `other.cpp` 中定义了一个函数 `Test`，而在 `main.cpp` 里希望调用这个函数。这看起来很简单，用我们刚刚学过的 `extern` 语法就可以解决。但问题来了，这个 `Test` 函数用一个结构类型 `Object` 作为参数。

```
////////// 例 CH18_C1 : other.cpp //////////
```

```
#include <stdio.h>

struct Object
{
    int value;
};

void Print(Object* obj)
{
    printf("value: %d \n", obj->value);
}
```

```
////////// 例 CH18_C1: main.cpp //////////
```

```
#include <stdio.h>

struct Object
{
    int value;
};

extern void Print(Object* obj);

int main()
{
    Object obj;
    obj.value = 123;
    Print(&obj);
    return 0;
}
```

我们可以发现一个明显存在的问题：同一个结构的定义要在不同的 `cpp` 里重复写好几遍。如果不写的话，就是语法错误。由于每个 `cpp` 是独立编译的，你在 `other.cpp` 中定义的结构类型对 `main.cpp` 没有任何影响，所以我们在 `other.cpp` 和 `main.cpp` 要把 `Object` 的定义抄两遍。

如果这个问题还嫌不够大，我们可以引申出一个更严重的问题。假设我们对 `Object` 类型进行扩展，增加一个成员变量，那我们就得修改每个 `cpp` 中的 `Object` 的定义。这是一件不科学，而且容易失误的操作。你很可能就把某一个处的定义忘了同步修改，类似例 `CH18_C2` 的局面。

////////// 例 CH18_C2 : other.cpp //////////

```
#include <stdio.h>

struct Object
{
    int value;
};

extern void Print(Object* obj);

int main()
{
    Object obj;
    obj.value = 123;
    Print(&obj);
    return 0;
}
```

////////// 例 CH18_C2 : main.cpp //////////

```
#include <stdio.h>

struct Object
{
    int value;
    int which;
};

void Print(Object* obj)
{
    printf("value: %d \n", obj->value);
    printf("which: %d \n", obj->which);
}
```

在上例 `CH18_C2` 中，在 `other.cpp` 中修改 `Object` 的定义，增加一个成员 `which`。但是却忘记了同步修改 `main.cpp` 中的定义。结果会是怎么样的呢？这是一个明显的失误，但在编译的时候编译器不会报错。

我们需要一种解决方案，在机制上避免这处问题的发生。

1.133.2 使用头文件

我们在项目中新增一个头文件 **Object.h**，然后把 **Object** 的类型定义写在里面，然后在需要它的 **cpp** 里加上

```
#include "Object.h"
```

这样便解决了前面所说的问题。我们在需要扩展 **Object** 的结构时，只需要修改一处：只要修改 **Object.h** 里的类型定义即可。

```
////////// 例 CH18_C3 : Object.h //////////
```

```
struct Object
```

```
{
```

```
    int value;
```

```
    int which;
```

```
};
```

```
////////// 例 CH18_C3 : other.cpp //////////
```

```
#include <stdio.h>
```

```
#include "Object.h"
```

```
void Print(Object* obj)
```

```
{
```

```
    printf("value: %d \n", obj->value);
```

```
    printf("which: %d \n", obj->which);
```

```
}
```

```
////////// 例 CH18_C3 : main.cpp //////////
```

```
#include <stdio.h>
```

```
#include "Object.h"
```

```
void Print(Object* obj);
```

```
int main()
```

```
{
```

```
    Object obj;
```

```
    obj.value = 123;
```

```
    obj.which = 456;
```

```
    Print(&obj);  
    return 0;  
}
```

头文件的写法：

(1) 后缀名一般为 .h

(2) 内容一般为几种：类型定义、extern 函数声明、extern 变量声明。

用头文件解决问题：例如，添加一个 Student.h,

```
struct Student  
{  
    int id;  
    char name[32];  
};
```

1.133.3 #include 指令的原理

我们来看一下#include 一行：

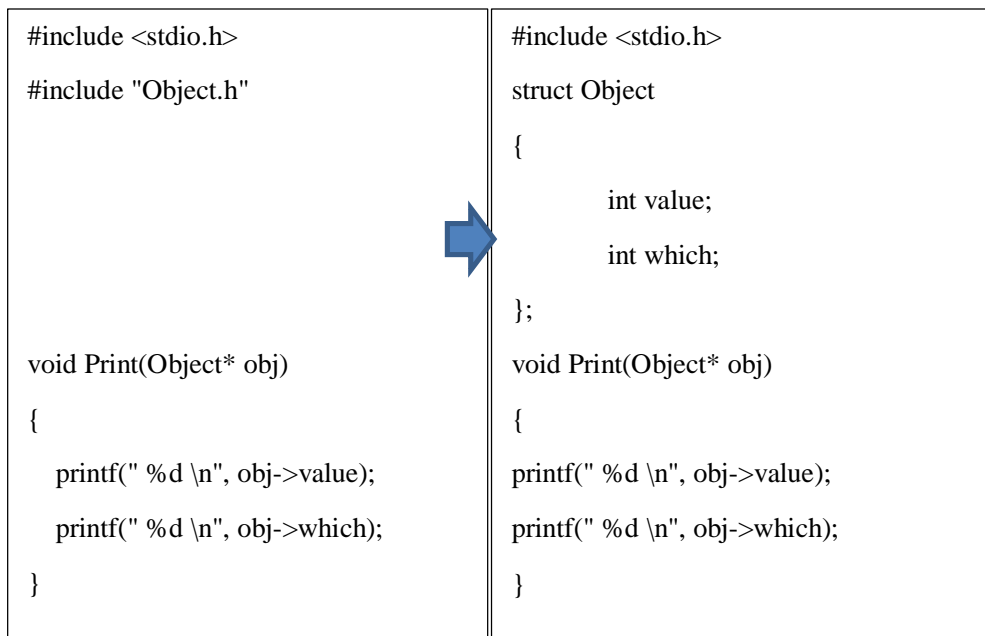
```
#include "Object.h"
```

其中，#include 我们称之为一条“预处理指令”。实际中 C/C++中预处理指令有很多种，其他的指令例如#define 会在后续章节中介绍。这条指令后面的"Object.h"表示要包含的头文件的路径，以双引号包围。

下面阐述一下它的功能原理。在 C++项目的生成过程中，除了前面介绍的“编译”、“链接”之外，其实在它们前面还有一个过程：预处理过程（Pre-Compile）。

预处理过程的作用是，编译器在处理每个 cpp 之前，首先将文件里的所有预处理指令进行处理，形成一个中间文件，然后再对这个中间文件进行“编译”。这个中间文件是临时生成的，我们看不到，但它确实会临时存在。

以#include 指令为例，编译器会在“预处理”阶段，把 cpp 里的所有#include 行，替换为相应的头文件内容，得到一个中间文件.....下图是一个示意图，表达了这个过程。



左侧框内是 other.cpp 的内容。右侧是将#include 替换后形成的中间文件的内容。

注：如果头文件里还有#include，则反复替换，直到没有任何#include 指令为止。

1.133.4 头文件的重复包含问题

如果要检查某个 cpp 文件有没有语法错误，需要要把预处理指令代入一下，才能够看清楚它有没有问题。当然，这个代入过程可以草稿纸上进行，或者直接脑海里模拟。

比如，在下面的代码中，我们重复包含了两次 Object.h，这会有问题吗？

```
////////// CH18_C4 //////////
#include <stdio.h>
#include "Object.h" //
#include "Object.h" // 重复包含相同的头文件
int main()
{
    Object obj;
    obj.value = 123;
    obj.which = 456;
    return 0;
}
```

我们把#include 指令替换，得到的中间文件为：

////////// 预处理后的中间文件//////////

```
#include <stdio.h>
```

```
struct Object
```

```
{
```

```
    int value;
```

```
    int which;
```

```
};
```

```
struct Object
```

```
{
```

```
    int value;
```

```
    int which;
```

```
};
```

```
int main()
```

```
{
```

```
    Object obj;
```

```
    obj.value = 123;
```

```
    obj.which = 456;
```

```
    return 0;
```

```
}
```

显然，这样的代码是有问题的，它把一个类型重复定义了两次。

1.133.5 头文件里放哪些内容

头文件里，一般放以下内容：

- (1) 共用的类型定义：如果一个类型要在多个 `cpp` 中使用，可以放在头文件里。
- (2) `extern` 函数声明，`extern` 变量声明
- (3) 嵌入包含其他头文件

1.134 宏定义 `#define` 指令

前面已经介绍，所有以 `#` 开头的行，称为预处理指令。这一节介绍一种 `#define` 指令，通常称为宏定义。

在老旧的代码，经常会看到一些`#define`的行，它们通常的用法多样、比较复杂。但需要强调的是，大部分用法在C++里都不提倡使用。我们学习它的目的，只是为了能看懂前人写的一些老的代码，并不是要鼓励你使用它。

在本节里，介绍`#define`的两个用法：

- (1) 定义一个“数值”
- (2) 定义一个“算式”

1.134.1 `#define` 一个数值

使用`#define`可以起到定义一个“常量”的效果，但此处的常量并非严格意义上的C++里的常量，只是类似的效果。

例如，我们有代码：

```
////////// CH18_D1 //////////  
  
#define PI 3.14  
  
int main()  
{  
    double r = 1.2;  
    double area = PI * r * r;  
    return 0;  
}
```

在预处理时，采用的是“原文替换”的方法。所以，预处理后的中间文件大概是这个样子：

```
int main()  
{  
    double r = 1.2;  
    double area = 3.14 * r * r; // 将PI 替换为 3.14  
    return 0;  
}
```

看起来`#define`的语法极其简单，只需要做一个“原文替换”就行了。但是，这里面掩盖着陷阱，并被“出题专家”们所青睐，几乎是各种考试中必考的试题。

例如，

```
#define MUL 1 + 2  
  
int main()  
{
```

```

    int a = 4 * MUL; // a 的值将是多少？
    return 0;
}

```

你可能会不假思索地回答：a 的值是 12。但是你错了。我们将 MUL 进行“原文替换”，检查一下：

```

#define MUL 1 + 2

int main()
{
    int a = 4 * 1 + 2; // 将 MUL 替换
    return 0;
}

```

明白了吗？它做的是“原文”替换，并没有对 1+2 先进行计算。下次这个坑你还会跳吗？

如果你坚持使用这些旧式语法，那么记得给宏定义加一些括号，就可以避免“意外”发生：

```

#define MUL (1 + 2)

```

在本教程中，推荐使用普通变量或 `const` 常量来实现目标功能。编译器在编译时，会先把常量部分先行计算，例如，

```

////////// 例 CH18_D2 //////////

#include <stdio.h>

const int MUL = 1 + 2; // 使用 const 常量替换宏定义

int main()
{
    int a = 4 * MUL; // MUL 的值是 3, a 的值是 12
    printf("%d, %d", MUL, a);
    return 0;
}

```

1.134.2 #define 一个算式

使用 `#define`，可以定义一个类似“函数”的东西。当然，它并不是函数，只是类似函数而已。同样地，学习这种用法的目的，只是为了能够看懂一些老旧的代码。

看一面的例子，

```

////////// CH18_D3 //////////

```

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

int main()
{
    int a = MAX(10, 12);
    return 0;
}
```

虽然样子看起来像“函数”，有“参数”一样的东西，但它显然不是函数。在预处理时，仍然是“原文替换”的原则。

```
int main()
{
    int a = 10 > 12 ? 10: 12; // 代入、替换、展开
    return 0;
}
```

这里面有同样的陷阱。例如，

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

int main()
{
    int a=1,b=2;
    int result = MAX (a+b, a-b); // result 是多少？实际代入了才知道，式子比较复杂
    return 0;
}
```

如果你坚持使用宏定义，那么给每一个“参数”都加上括号，就可以避免意外发生：

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

在本教程中，建议你使用 inline 函数来替换这种宏定义方法。

1.134.3 几个常见的宏定义

(1) NULL 空指针

```
#define NULL 0
```

或

```
#define NULL (void*)0
```

(2) RAND_MAX 的宏定义

此定义在第 16 章用到，在使用 rand 函数时可能会用到。


1.135* 条件编译指令 #if

(初学者可以选择跳过本节内容，但本节内容还是比较常用的)

条件编译指令也有很多种格式，这里不一一介绍，只介绍其中的几种。


1.135.1 #if...#endif

在例 CH18_E1 中，

<pre>////////// 例 CH18_E1 ////////// #include <stdio.h> int main() { #if 1 printf("aaaa\n"); #else printf("bbbb\n"); #endif return 0; }</pre>		<pre>////////// 预处理之后 ////////// #include <stdio.h> int main() { printf("aaaa\n"); return 0; }</pre>
---	---	--


也就是说，经过预处理之后，某些行相当于不存在、根本不参与编译。

1.135.2 #ifdef...#endif


<pre>////////// 例 CH18_E2 ////////// #include <stdio.h> #define ENABLE_OUTPUT int main() { #ifdef ENABLE_OUTPUT printf("aaaa\n"); #endif return 0; }</pre>		<pre>////////// 预处理之后 ////////// #include <stdio.h> int main() { printf("aaaa\n"); return 0; }</pre>
---	---	---

#ifdef 表示如果对应的宏有定义，则相应的代码被编译。

我们可以使用#undef 指令去除定义：

<pre>////////// 例 CH18_E3 ////////// #include <stdio.h> #define ENABLE_OUTPUT #undef ENABLE_OUTPUT int main() { #ifdef ENABLE_OUTPUT printf("aaaa\n"); #endif return 0; }</pre>		<pre>////////// 预处理之后 ////////// #include <stdio.h> int main() { return 0; }</pre>
--	---	---

#ifndef 表示的意思和#ifdef 恰好相反：当相应的宏不存在时，才编译相应的代码。

<pre> //////////////////// 例 CH18_E4 ////////////////////// #include <stdio.h> #define ENABLE_OUTPUT int main() { #ifndef ENABLE_OUTPUT printf("aaaa\n"); #endif return 0; } </pre>	 <pre> //////////////////// 预处理之后 ////////////////////// #include <stdio.h> int main() { return 0; } </pre>
---	--

1.135.3 解决头文件重复包含的问题

通常，我们要对头文件用条件编译指令对其保护，之后便可以对其重复包含了。

例如，

```

//////////////////// CH18_E5 : Object.h //////////////////////
#ifndef _OBJECT_H
#define _OBJECT_H
struct Object
{
    int value;
    int which;
};
#endif

//////////////////// CH18_E5 : main.cpp //////////////////////
#include "Object.h"
#include "Object.h" // 此头文件已经被保护，可以重复包含
int main()
{
    Object obj;

```

```

    obj.value = 1;
    obj.which = 2;
    return 0;
}

```

我们把 main.cpp 经过预处理展开之后，结合条件编译指令的语法，可以发现并没有语法错误。于是“头文件重复包含”的问题得到解决。

```

////////// 预处理展开 main.cpp //////////

#ifndef _OBJECT_H
#define _OBJECT_H
struct Object
{
    int value;
    int which;
};
#endif

#ifndef _OBJECT_H
#define _OBJECT_H
struct Object
{
    int value;
    int which;
};
#endif

int main()
{
    Object obj;
    obj.value = 1;
    obj.which = 2;
    return 0;
}

```

1.136 *main 函数的参数和返回值

（初学者可以跳过本节）

无论你的 C++ 项目里有多少个文件，名字为 `main` 的函数只能有一处定义，因为它是程序的入口函数。

`main` 函数可以有多种形式，但标准形式是下面两种：

- ① `int main()`
- ② `int main(int argc, char* argv[])`

本节的主要内容是讨论 `main` 函数参数和返回值。也就是讨论以下两个问题：

- （1）`main` 函数的参数从哪来？
- （2）`main` 的返回值送给谁了？

1.136.1 main 函数的参数

```
int main(int argc, char* argv[])
```

其中，`argc` 表示命令行参数的个数，`argv` 表示命令行参数的值。所谓命令行参数，就是我们在命令行窗口（DOS 窗口/SHELL 窗口）中运行这个程序时，传给它的参数。

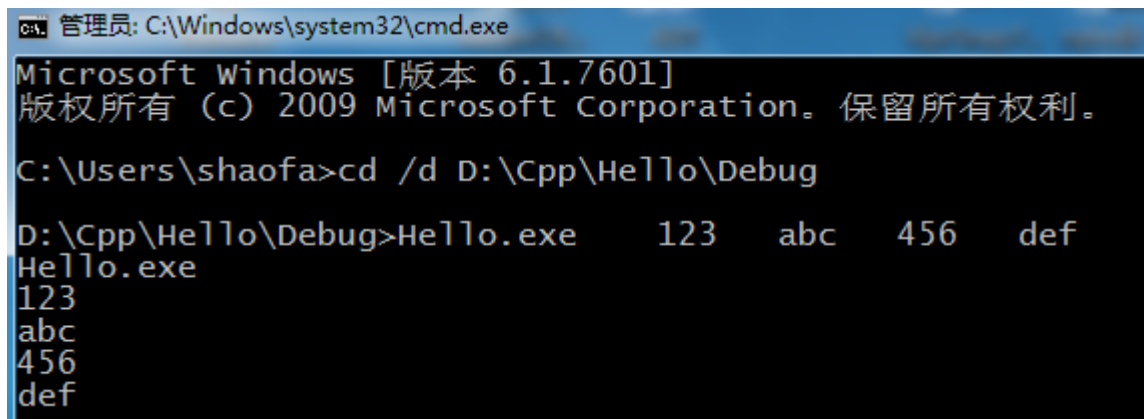
////////// 例 CH18_F1 //////////

```
#include <stdio.h>

int main(int argc, char** argv)
{
    for(int i=0; i<argc; i++)
    {
        printf("%s\n", argv[i]);
    }

    return 0;
}
```

我们编译这个程序，然后打开系统的命令行窗口，切换到 `Debug` 目录所在的路径。然后输入：`Hello.exe 123 abc 456 def`，控制台显示如图 18-1 所示。



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\shaofa>cd /d D:\Cpp\Hello\Debug

D:\Cpp\Hello\Debug>Hello.exe 123 abc 456 def
Hello.exe
123
abc
456
def
```

图 18-1 命令行窗口中输入命令行参数

这说明了，argc 的值是 5，而 argv[] 中存储的字符串分别就是 "Hello.exe", "123", "abc", "456", "def"。

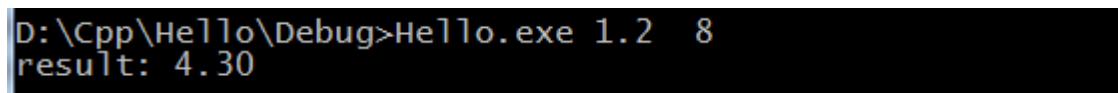
这意味着我们可以通过 main 函数的参数 argc 来 argv 来接收用户的输入。下面的例子 CH18_F2 演示了如何利用 argc 和 argv 来做一个命令程序。

////////// 例 CH18_F2 //////////

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

int main(int argc, char** argv)
{
    double a = atof(argv[1]);
    int n = atoi(argv[2]);
    double result = pow(a, n);
    printf("result: %.2f\n", result);
    return 0;
}
```

这个程序的作用是求一个数 a 的 b 次方，其中 a 和 b 可以在命令行参数中指定。例如，下面是求 1.2 的 8 次方，输入命令参数如图 18-2 所示。



```
D:\Cpp\Hello\Debug>Hello.exe 1.2 8
result: 4.30
```

图 18-2 命令行窗口中输入命令行参数

1.136.2 main 函数的返回值

main 函数的返回值是 int 类型。一般来说，当程序返回 0 表示该任务成功完成。非 0 表示遇到错误或异常情况。但问题是，当 main 的 return 语句被执行时，整个程序已然退出，那么这个返回值还要返给谁呢？

答案是，谁调用了这个程序，返回值就给它。当我们在 cmd 命令行窗口中运行该 exe 时，返回值就交给了 cmd 这个窗口程序。当我们调用 system 函数运行这个程序时，system 的返回值就是这个程序的 main 函数的返回值。

例如，我们有一个程序 aaa.exe，main 函数返回值为-10。

我们建立另外一个 C++ 项目，在代码里有 system 函数调用 aaa.exe，那么 system 函数的返回值就是-10。

```
int ret = system("D:\\Cpp\\aaa.exe"); // 指定 aaa.exe 的全路径
printf("ret=%d \n", ret);
```

1.137 *static 的用法

（初学者可以跳过本节）

本节介绍关键字 static 的用法，当 static 修饰一个函数时，称之为静态函数。当 static 修饰一个变量时，称之为静态变量。

1.137.1 static 修饰变量

当 static 修饰一个全局变量时，该变量的生命期未变，还是和普通全局变量一样。所不同的是，该变量的名字被限制在本 cpp 页面内可见。也就是说你无法再用 extern 来声明并访问它。例如，

```
////////// CH18_G1: other.cpp //////////
static int number = 0; // number 被 static 修饰
////////// CH18_G1: main.cpp //////////
#include <stdio.h>
extern int number; // 错！ number 不可见！
int main(int argc, char** argv)
{
    number = 111;
    printf("value: %d \n", number);
}
```

```
    return 0;
}
```

可以把这种变量定义在函数之内，变量的生命期未变（性质不变，本质还是本局变量），但可见范围进一步缩小：仅在函数内可见。

////////// 例 CH18_G2 //////////

```
#include <stdio.h>

void Test()
{
    static int number = 0; // 性质仍然是全局变量，但仅在函数内可以访问它
    number ++;
    printf("number: %d \n", number);
}

int main(int argc, char** argv)
{
    Test();
    Test();
    Test();
    return 0;
}
```

运行此程序，结果如图 18-3 所示。



```
number: 1
number: 2
number: 3
```

图 18-3 控制台输入显示

这种写法有一定的迷惑性。注意：被 `static` 修饰的都是全局变量，生命期都是永恒的。它写在函数里面，仅仅表示它的可见范围比较小，仅在函数里可见。如果你实在不能理解这个运行结果，可以试着把变量定义挪到类的外面，本质是一样的。

1.137.2 static 修饰函数

当 `static` 修饰一个函数时，该函数就不能够在别的 `cpp` 里用 `extern` 声明和访问了。也就是说，可见范围缩小，仅在本 `cpp` 内可见了。

////////// 例 CH18_G3 : other.cpp //////////

```
#include <stdio.h>

static void Test()
{
    printf("this is a test ...\n");
}

////////// 例 CH18_G3 : main.cpp //////////

extern void Test(); // 不可以! Test 只在 other.cpp 内可见!

int main()
{
    Test();
    return 0;
}
```

编译时会提示“无法解析的外部符号”，找不到 Test 函数。

第 19 章 面向对象编程

面向对象编程 (Object-Oriented Programming, OOP)，是一种设计思想。它不局限于语言。无论是 C 语言，还是 C++ 语言，抑或 Java，B 语言，甚至像脚本语言、汇编语言，都可以实现面向对象编程。显然地，相对于纯 C 的语法来说，用 C++ 语言来实现面向对象编程会容易一些。

在本章中，介绍如何利用 C 语言的技术来实现面向对象编程。

1.138 面向对象设计的过程

在接到一个任务（需求）时，如何进行面向对象的设计呢？可以分为以下几步。

（1）假设存在一个对象

要定义一下这个对象应该具有提供哪些服务。所谓服务，也就是我们可以用它来做什么。一个对象可能提供若干服务，每一项服务对应一个函数。把这样的功能函数称为“函数接口”。

（2）确定函数接口

一方面，要确定每一个函数接口的具体的参数列表：输入什么、输出什么。另一方面，要确定如何使用这些函数，有无先决条件、先后次序。

比如说，一般都要用 `create()` 函数用于创建一个对象，`destroy()` 用于销毁一个对象。对象在被成功创建之后才能使用。

（3）实现这些函数

前两步是完成了设计，它们是最重要、最困难的过程。最后这一步则是将设计好的函数接口用代码实现出来。

1.139 实例演示

1.139.1 需求与设计

让我们通过一个实例来感受一下面向对象的设计过程。

我们的需求是：已经几个 `int` 型数组，要求求出它们的并集。所谓并集，就是包含了所有元素、但每个元素都不重复。例如，有以下数组：

```
int a[] = { 1, 2, 3};
```

```
int b[] = { 4, 5, 6};
```

```
int c[] = { 1, 3, 5, 7};
```

则它们的并集结果中应该包含 1,2,3,4,5,6,7 这几个元素。

按照面向对象的设计步骤：

（1）假定存在一个对象，命名为 `ResultSet`

它有以下功能：可以向它存入任意个 `int` 型整数。它接收输入，内部去除重复。可以查询当前一共保存了多少个数。

（2）将功能细化为函数接口

让我们来分析一下，它应该有哪些函数接口。

首先，一个对象的创建和销毁，必须有一对函数来完成。

```
ResultSet* create();
```

```
void destroy(ResultSet* obj);
```

由 `create` 来创建得到一个对象。当对象用完之后，调用 `destroy` 来销毁它。

我们需要获取它内部保存的数据及长度，于是定义两个函数来获取，

```
int count(ResultSet* obj);
```

```
int at(ResultSet* obj, int i);
```

其中，`count()`用于查询元素的个数，而 `at()`则用于取得第 `i` 个元素。

然后需要有一些函数接口可以向里面存入数据，

```
int push(ResultSet* obj, int a);
```

```
int push(ResultSet* obj, int arr[], int n);
```

第一个 `push()`向对象里存入一个数，如果存入成功，则返回 1；否则，该数已经存在，则返回 0。

第二个 `push()`向对象里存入几个数，返回实际存入的个数。也就是说，重复的就不计算在内了。

那么到此为止，设计工作告以段落。也就是说，把函数接口和功能定义出来了，最重要的事情就已经做完了。

1.139.2 写出代码框架

我们把设计好的框架写出来，如下所示。对象类型为 `ResultSet`，先不添加任何成员变量。然后把所有的函数框架写好，返回值取默认值，使之可以编译通过。为每个函数的功能添加一段注释，说明其功能、输入输出参数的意义、返回值的意义。

```
/* 对象类型 */
```

```
struct ResultSet
```

```
{
```

```
};
```

```
/* 创建一个对象
```

返回值：对象，若为 NULL 表示创建失败

```
*/
```

```
ResultSet* create()
```

```
{
```

```
    return NULL;
```

```
}
```

```
/* 销毁对象
```

obj: 对象指针

```
*/
```

```
void destroy(ResultSet* obj)
```

```
{
```

```
}
```

```
/* 查询已存储的元素的个数
```

obj: 对象指针

返回值：对象中所存储的元素的个数

```
*/
```

```
int count(ResultSet* obj)
```

```
{
```

```
    return 0;
```

```
}
```

```
/* 取得第 i 个元素的值
```

obj: 对象指针

i: 索引

```
*/
```

```
int at(ResultSet* obj, int i)
```

```
{
```

```
    return 0;
```

```
}
```

```
/* 存入一个数
```

a: 待存入的数

返回值：1，表示存入成功。0，表示未存入

```
*/
```

```
int push(ResultSet* obj, int a)
```

```
{
```

```
    return 0;
```

```
}
```

```
/* 存入一组数
```

arr: 数组的首地址

n: 数组长度

返回值：实际存入的个数。去除重复。

```
*/
```

```
int push(ResultSet* obj, int arr[], int n)
```

```
{
```

```
    return 0;
```

```
}
```

然后，我们规定一下函数使用方法。也就是说，在函数还没有实现之前，我们就已经规定好了这些函数接口是如何调用的。它的使用方法应该是下面这样：

```
int main()
```

```
{
```

```
    // 创建对象
```

```
    ResultSet* obj = create();
```



```

// 存入数据

int a[] = { 1, 2, 3};
int b[] = { 4, 5, 6};
int c[] = { 1, 3, 5, 7};
push(obj, a, 3);
push(obj, b, 3);
push(obj, c, 4);

// 查看结果

for(int i=0; i<count(obj); i++)
{
    printf("%d ", at(obj, i));
}

// 销毁对象

destroy(obj);

return 0;
}

```

1.139.3 实现各个函数接口

前面已经规定了这个对象应该有哪些函数接口、每个函数接口应该具有的功能。在实际工程应用中，这些就是最重要的工作。

下面，就需要把这些函数接口实现出来。具体实现的时候，有一定的灵活性。你可以自由选择一种方案，使用数组、链表或其他方式，总之，能满足这些函数接口的要求就好。

在这里使用一种简单实现方案：使用数组来存储内部数据。这种实现方案并不完美，比如，数组的容量是固定的，要么空间不足、要么浪费空间。然而，在实际工程中并不一定要做的尽善尽美，即使不完美的方案也是可以接受的。假设我们已经知道一共存储的数据个数是很少的，那么用一个数组也是未尝不可的。

首先，完善一下对象的类型，添加一个数组来保存数据。

```

struct ResultSet
{
    int data[200];
    int n;
};

```

其中，数组 `data` 用作存储数据，而 `n` 则是记录一下里面一共存储了多少数据。显然，在对象初始化的时候应该把 `n` 设置为 0。

然后，把 `create/destroy` 填上正确的代码。

```
ResultSet* create()
{
    ResultSet* obj = (ResultSet*) malloc(sizeof(ResultSet));
    obj->n = 0;
    return obj;
}

void destroy(ResultSet* obj)
{
    free(obj);
}
```

然后，把 `count/at` 填上正确的代码。

```
int count(ResultSet* obj)
{
    return obj->n;
}

int at(ResultSet* obj, int i)
{
    return obj->data[i];
}
```

然后，把 `push` 填上正确的代码，这两个函数是核心业务函数。在第一个 `push` 函数里，要把一个整数 `a` 存入，那么要先检查一下该数是否已经存在，如果已经存在则直接返回 0。在第二个 `push` 函数里，调用了第一个版本的 `push`，将一个数组内的所有数据存放、并记录存入的实际个数。

```
int push(ResultSet* obj, int a)
{
    for(int i=0; i<obj->n; i++)
```

```

    {
        if(a == obj->data[i])
            return 0;
    }

    obj->data[obj->n] = a;
    obj->n += 1;
    return 1;
}

int push(ResultSet* obj, int arr[], int n)
{
    int total = 0;
    for(int i=0; i<n; i++)
    {
        if(push(obj, arr[i]))
            total ++;
    }
    return total;
}

```

到此为止，相关函数接口已经完全实现。

1.140 封装

此时，我们应该重点考虑一下 `count` 和 `at` 这两个函数接口，它们为什么要这么定义？

按照我们先前对结构体的语法的认识，结构体内的成员变量是可以直接访问的，那么，看起来根本不需要 `count/at` 这两个函数，我们直接在 `main` 函数里是可以访问这两个变量的，就像下面的代码这样？

```

for(int i=0; i<obj->n; i++) // 直接访问 obj->n
{
    printf("%d ", obj->data[i]); // 直接访问 obj->data
}

```

但是经过深入的思考我们可以发现，不应该直接访问结构体的成员变量，虽然语法上是允许的。这样会破坏设计的另一个原则：封装。

所谓封装，就是将内部的东西封闭起来，外部若想访问，就必须通过指定的函数接口。这样，一来可以避免外部调用者的随意操作、损坏内部的数据，二是给实现的人员带来自由度。

为什么用函数会增加自由度呢？可以反过来看一下，如果外部直接访问 **ResultSet** 内部的成员变量，那么，就是规定死了 **Result** 必须用数组、且各个成员变量的变量都固定了。这样就无法选择别的方案来实现。而用函数 `count/at` 来访问的话，可以随意修改实现方案，不会影响到外部的调用代码。

也就是说，**ResultSet** 的实现者可以换一个实现方案，比如用链表来实现各个函数接口。他不必通知外部调用者，因为外面的代码无须做出任何改变。

第 20 章 类

1.141 类和成员变量

类(class)，是对结构体 struct 的增强，也是用于自定义类型的。图 20-1 演示了如何从 struct 的语法过滤到 class。

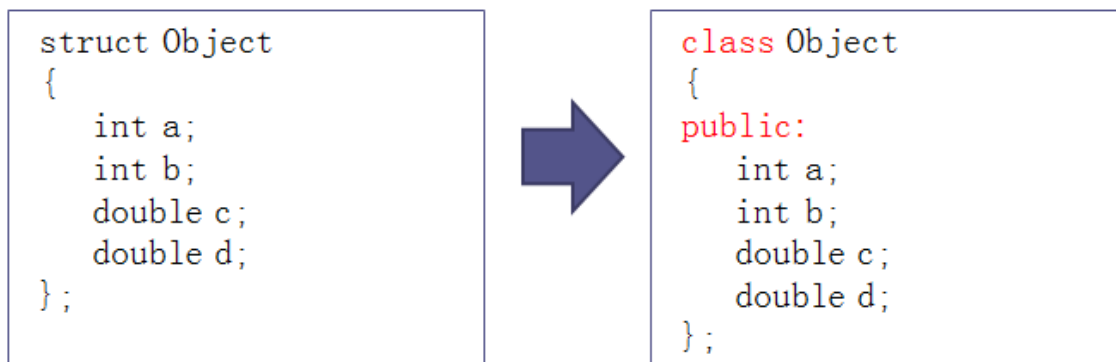


图 20-1 由 struct 过渡到 class

- (1) 把关键字 struct 改为 class
- (2) 增加一个修饰符 public，并以冒号结束

这样，我们便得到一个自定义的类型 Object。

////////// 例 CH20_A1 //////////

```
class Object
```

```
{
```

```
public:
```

```
    int a;
```

```
    int b;
```

```
    double c;
```

```
    double d;
```

```
};
```

而它的使用方法和传统的 struct 也暂时没有区别，以下代码中演示了 Object 类型和 Object* 指针类型的用法。

////////// 例 CH20_A1 //////////

```
int main()
```

```
{
```

```
    Object obj;
```

```

    obj.a = obj.b = 1;
    obj.c = obj.d = 2.2;
    Object* p = &obj; // 指针类型
    printf("%d, %d, %lf, %lf\n", p->a, p->b, p->c, p->d);
    return 0;
}

```

1.141.1 了解访问修饰符

下面说一下访问修饰符 `public` 的作用。我们知道，在 `struct` 语法中，对于 `struct` 中定义的每一个成员变量，外部都可以直接读取和修改它。这是一个不好的特性，在设计上我们称之为破坏了“封装”。也就是说，`struct` 内部的数据和它的算法逻辑是对应的，外面不应该随便修改它。

访问修饰符就是这个作用，使用 `public` 或 `private`，可以设定成员变量的访问权限。当成员被 `public` 修饰时，外面就可以直接访问这个成员。当被 `private` 修饰时，外部就不可以直接读写这个成员。例如，在下面的代码，试图访问 `Object` 类的 `private` 成员，编译器会报错。

```

////////// 例 CH20_A2 //////////
#include <stdio.h>
class Object
{
public:
    int a;
    int b;
private:
    double c;
    double d;
};
int main()
{
    Object obj;
    obj.a = obj.b = 1; // a,b 被 public 修饰，可以访问
    obj.c = obj.d = 2.2; // c,d 被 private 修饰，不可以访问, 编译器报错
    Object* p = &obj;
    p->a = 11;
}

```

```
p->c = 13.0; // 编译错误！ private 成员！  
return 0;  
}
```

1.141.2 访问修饰符的用法

(1) public/private 后面要加冒号

(2) 习惯上顶格书写，但不顶格也不是错误。甚至把多个修饰符也在同一行，也不是错误。

例如，

```
class Object  
{  
public :  
private:  
private: public: // 多个修饰符也在同一行，也没问题  
};
```

(3) 每个成员只受到前面最近一个修饰符限制

```
class Object  
{  
public:  
private:  
public:  
    int a; // 受 public 限制  
    int b; // 受 public 限制  
private:  
public:  
private:  
    double c; // 受 private 限制  
    double d; // 受 private 限制  
};
```

(4) 对于 class 类型来说，成员属性默认是 private 的。

```
class Object
```

```

{
    int a; // 前面找不到修饰符，默认为 private
    int b; // 前面找不到修饰符，默认为 private
public:
    double c;
    double d;
};

```

1.142 类和成员函数

定义在 class 内的函数，称为该类的成员函数（Member Function）。我们把成员变量和成员函数统称为类的成员（Members）。

例如，在类 Object 内定义一个 Test()函数，示例如下。

////////// 例 CH20_B1 //////////

```

class Object
{
public:
    int x;
    int y;
    void Test()
    {
        printf("hello,world!\n");
    }
};

```

对成员函数的调用方法和成员变量差不多，也是用点号或者箭头。下面的示例子中展示了如何调用 Object 类的成员函数 Test()。

////////// 例 CH20_B1 //////////

```

int main()
{
    Object obj;
    obj.Test(); // 用点号访问成员函数
    Object* p = &obj;
}

```



```

    p->Test(); // 用箭头访问成员函数

    return 0;

}

```

1.142.1 访问修饰符的限制

对于成员函数，同样受到访问修饰符的限制，而且规则是完全一样的。也就是说，访问修饰符 `public` / `private` 对成员变量和成员函数施加同样的限制。

例如，在 `Object` 类中添加一个 `Test()` 函数，受 `public` 限制。再加一个 `Test2()` 函数，受 `private` 限制，则在 `main` 函数中不允许调用 `Test()` 函数。示例代码如下。

////////// 例 CH20_B2 //////////////////////////////////////

```

class Object
{
public:
    int x;
    int y;
    void Test()
    {
        printf("hello,world!\n");
    }

private:
    void Test2() // 该函数被 private 修饰，不能被外部访问
    {
        printf("Private function!\n");
    }
};

```

在 `main` 函数中访问 `obj.Test()` 函数时，编译器会报错。示例代码如下。

////////// 例 CH20_B2 //////////////////////////////////////

```

int main()
{
    Object obj;

    obj.Test2(); // 编译错误！禁止访问 private 成员！

    Object* p = &obj;
}

```

```

    p->Test2(); // 编译错误！禁止访问 private 成员！
    return 0;
}

```

1.142.2 了解 this 指针

现在，我们希望在 Object 的 Test()函数里，将成员变量 a,b 打印出来，怎么实现呢？

```

class Object
{
public:
    int a;
    int b;
    void Test()
    {
        // 打印 a,b 的值
    }
};

```

第（1）种方法：我们使用现有的知识，可以解决这个问题。也就是为 Test 函数设置一个参数，将对象 Object 对象的指针传入。这种方法是毫无问题的。示例代码如下：

```

//////////例 CH20_B3 //////////
#include <stdio.h>

class Object
{
public:
    int a;
    int b;
    void Test(Object* that) // 传入对象指针
    {
        printf("a=%d, b=%d\n", that->a, that->b);
    }
};

int main()
{

```

```

    Object obj;

    obj.a = 1;

    obj.b = 2;

    obj.Test(&obj);

    return 0;

}

```

第（2）种方法：使用 **this** 指针。在 C++ 中，为了简化第（1）种方法中的代码，默认地为每个成员函数传入一个 **this** 指针，指向了这个对象本身。我们不必显式的传递 **Object* that** 了，因为编译器已经隐含地为我们做了这件事，我们直接使用就好。

下面的代码中，直接使用 **this** 指针来访问对象的成员变量 **a,b**：

```

////////// 例 CH20_B4 //////////

#include <stdio.h>

class Object
{
public:
    int a;
    int b;

    void Test()
    {
        printf("a=%d, b=%d\n", this->a, this->b);
    }
};

int main()
{
    Object obj;

    obj.a = 1;

    obj.b = 2;

    obj.Test();

    return 0;

}

```

记住，使用 **this** 指针只是为了简化我们的代码，可以少写一个参数。如果你是个守旧的人，那么完全可以无视 **this** 指针的存在，坚持使用第（1）种办法。

1.142.3 this 指针的用法

前面说过，`this` 指针在一个隐含的函数参数，你如果不想用 `this` 指针，就得自己在函数里添加一个 `Object* that` 类似的参数。

(1) 使用 `this` 指针，可以访问类的所有成员变量

(2) 使用 `this` 指针，可以访问类的所有成员函数

显然，使用 `this` 指针访问时，是不受 `public/private` 限制的，因为它是从类的成员函数里访问。这属于从内部访问，而在 `main` 函数里访问则算是从外部访问。

例如，在下面的 `Add` 函数可以访问 `x,y`，尽管 `x,y` 被修改为 `private`，但是由于是在类的成员函数内部访问，所以完全不用考虑 `private`。

////////// 例 CH20_B5 //////////

```
#include <stdio.h>
```

```
class Object
```

```
{
```

```
private: // x,y 是 private 的, 但用 this->可以访问
```

```
    int x;
```

```
    int y;
```

```
public:
```

```
    void Set(int a, int b)
```

```
    {
```

```
        this->x = a;
```

```
        this->y = b;
```

```
    }
```

```
    int Add()
```

```
    {
```

```
        return this->x + this->y; // 用 this->调用其他成员
```

```
    }
```

```
    void Test()
```

```
    {
```

```
        printf("Sum: %d \n", this->Add()); // 用 this->调用其他成员
```

```
    }
```

```
};
```

```

int main()
{
    Object obj;
    obj.Set(1, 2);
    obj.Test();
    return 0;
}

```

从例 CH20_B5 中我们在发现以下几点：

- (1) 在 Set 和 Add 函数可以可以访问成员变量 x, y
- (2) 在 Test 函数中可以访问成员函数 Add，也是通过 this 指针进行调用

为了进一步简化我们的代码，this->调用可以省略不写，例如，

```

void Set(int a, int b)
{
    x = a; // 省略 this->
    y = b; // 省略 this->
}

int Add()
{
    return x + y; // 省略 this->
}

void Test()
{
    printf("Sum: %d \n", Add()); // 省略 this->
}

```

1.143 变量名字的覆盖

这一节讨论一下 C++ 里的变量名重名的问题。在重名的时候，遵守一条“就近原则”。现在我们已经认识了几种位置的变量：

- * 全局变量：定义在函数体之外
- * 成员变量：类中
- * 成员函数内的局部变量：定义在类的成员函数里

(1) 在成员函数里：当局部变量与成员变量重名时，该变量指向的是局部变量。

例如，在下面的代码中，成员函数 Test()里有一个局部变量 x，而类 Object 中也有一个成员变量 x，那么，printf 这一行代码会把哪一个 x 给打印出来呢？

////////// 例 CH20_C1 //////////

```
class Object
{
public:
    int x; // 成员变量

    void Test(int x) // 局部变量（参变量）
    {
        printf("x = %d \n", x); // 最近的：函数内定义的 x
    }
};
```

就近原则：距离此 x 最近的是参数里的局部变量。我们可以在代码中测试一下，给 obj.x 设置为 123，给 Test 传值 456，检查 printf 输出的值是 123、还是 456？

```
int main()
{
    Object obj;
    obj.x = 123;
    obj.Test(456);
    return 0;
}
```

试验结果表明，打印出来的数字是 456，说明参变量被打印。

如果我们要指明要访问是成员变量的 x，则要加上 this->前缀来显式的指定。例如，

```
void Test(int x)
{
    printf("x = %d \n", this->x); // 显式指定：访问 this->x
}
```

```
}
```

(2) 在成员函数里：当成员变量与全局变量重名时，该变量指向的是成员变量。

以 CH20_C2 为例进行说明。printf 中访问的 x 是成员变量 x，而不是全局变量 x。原因是，成员变量 x 离得最近。

```
////////// 例 CH20_C2 //////////  
  
#include <stdio.h>  
  
int x = 789; // 全局变量  
  
class Object  
{  
public:  
    int x;  
    void Test()  
    {  
        printf("x = %d \n", x); // 最近的 x:成员变量 x  
    }  
};  
  
int main()  
{  
    Object obj;  
    obj.x = 123;  
    obj.Test();  
    return 0;  
}
```

如果要显式地指定一定要访问全局变量 x，则可以全局范围符::

```
void Test()  
{  
    printf("x = %d \n", ::x); // 使用::来显式地指定全局变量 x  
}
```

(3) 在成员函数与全局函数重名时，默认指成员函数

其原理和（2）是一样的。当函数重名时，也是用::来访问全局函数。

```
////////// CH20_C3 //////////
```

```
#include <stdio.h>
```

```
// 全局函数
```

```
int Sum(int x, int y)
```

```
{
```

```
    return x + y;
```

```
}
```

```
class Object
```

```
{
```

```
public:
```

```
    // 成员函数
```

```
    int Sum(int x, int y)
```

```
    {
```

```
        return 2*(x + y);
```

```
    }
```

```
    void Test()
```

```
    {
```

```
        printf("sum: %d \n", Sum(3,4)); // 指向成员函数
```

```
        printf("sum: %d \n", ::Sum(3,4)); // 指向全局函数
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    Object obj;
```

```
    obj.Test();
```

```
    return 0;
```

```
}
```


1.144 命名规范

在 C++ 中，无论是函数名、变量名、类名、类型名，命名的语法都是一致的：“字母、数字下划线的组合，可以用字母、下划线开头，不可以用数字开头”。

在实际工作中，为了提高代码的可读性，对命名还有一些额外的要求。一是要能顾名思义，让名字反映其意义；二是要遵守一定的规范，符合一定的格式。这一节里，就重点给大家阐述命名规范的问题。

1.144.1 类名的命名规范

下面，列出一些实际存在于各种工程库里的类，请观察它们的命名规律：

Rect : 用于表示一个屏幕上的矩形

MainFrame : 表示一个图形窗口程序的主框架

InetAddress : 表示网络地址

StringList : 表示字符串的链表

MemoryPool : 表示管理内存的内存池

BufferedReader : 表示带缓存的一个读取器

PdfWriter : 表示一个可以生成 PDF 格式文档的生成器

XmlParser : 一个 XML 格式的文本解析器

显然，这些名字都是符合第（1）项要求的，通过它的命名就能直接看懂它的大致意义，甚至不需要任何注释说明。

除此之外，这些类的命名遵循了以下规范：①每个单词的首字母大写 ②使用名词形式 ③尽量使用单词的全拼，当太长的时候可以缩写。为什么要使用名词形式？因为“类”是一种类型的定义，类型应该是一个名词。

1.144.2 成员函数的命名规范

对于 C++ 类的成员函数，其命名规范为：①每个单词的首字母大写 ②使用动词形式。

下面给出一些示例：

Open 打开

Close 关闭

Add 添加

Remove 删除

SendMessage 发送消息

RecvMessage 接收消息

我们可以发现，成员函数的命名有以下规律：

- (1) 通常是成对出现的，例如 Open/Close，Send/Recv
- (2) 有一些常用的名称供我们选择，我们一般不需要去使用一个生僻的名字

1.144.3 成员变量的命名规范

成员变量的命名规范：①小写开头 ②通常加上前缀 m_，其中 m 代表的是 member 的意义。

例如

m_number

m_server

m_port

m_buffer;

m_maxsize;

注：成员变量的命名形式的要求并不严格，一般以 m_ 开头即可。

1.145 类的封装

封装是程序设计中的一个概念，它指针对一个模块的。可以把一个类认为是一个模块，在设计一个类的时候，把数据隐藏在内部，把函数接口暴露在外面。这便是封装。

经过封装之后，外部调用者只能通过 public 的函数接口来访问该模块。由于数据已经被 private 限制，所以外部不能直接访问其数据（成员变量），而只能通过函数接口来访问数据。显然，public/private 修饰符是实现封装技术的一种有效手段。

相比于 C 时代的 struct 语法，可以更能深刻明白访问修饰符的重要性。在以前 struct 中定义的成员变量，总是被外部可以随意的访问，这很有可能会破坏模块数据的完整性。而在 C++ 时代，使用访问修饰符来限定其访问权限。

public: 该成员可读/可写

private: 该成员不可读/不可写

此外，我们可以实现成员变量的只读或者只写。首先，把成员变量修饰为 private。然后，添加一个 Get/Set 函数来访问这些成员变量。（在编程领域，将这些简单的 Get/Set 函数称之为 getter 和 setter）。

在下面的示例中，成员变量 m_value 是只读的，因为只有 getter 没有 setter。

```

class Object
{
public:
    // 定义一个 getter
    int GetValue()
    {
        return m_value;
    }
private:
    int m_value;
};

```

而在下面的示例中，成员变量 `m_value` 是只写的（只能写，不能读）。

```

class Object
{
public:
    // 定义一个 setter
    void SetValue(int v)
    {
        m_value = v;
    }
private:
    int m_value;
};

```

1.146 类的分离式写法

可以把成员函数的定义写在类体之外，遵循以下步骤：（1）在类里保留成员函数的声明（2）在类外定义该成员函数，函数名字前上加上类名的范围前缀。例如，

```

////////// 例：CH20_D1 //////////
#include <stdio.h>

```

```

class Object
{
public:
    int x;

    void Test(); // (1) 成员函数的声明
};

void Object::Test() //(2) 成员函数写在外边，加上类名限定 Object::
{
    printf("..... testing .....\\n");
}

int main()
{
    Object obj;
    obj.Test();
    return 0;
}

```

通常，我们它们分别写在头文件(.h)和源文件(.cpp)中。下面的示例 CH20_D2 展示这种写法。

//////////////////// 例 CH20_D2: Object.h //////////////////////

```

class Object
{
public:
    int x;

    void Test(); // (1) 成员函数的声明
};

```

//////////////////// 例 CH20_D2: Object.cpp //////////////////////

```

#include <stdio.h>

#include "Object.h"

```

```

void Object::Test() //(2) 成员函数写在外边，加上类名限定
{

```

```

        printf("..... testing .....\\n");
    }

    //////////////// 例 CH20_D2: main.cpp ///////////////////

#include <stdio.h>
#include "Object.h"

int main()
{
    Object obj;
    obj.Test();
    return 0;
}

```

1.147 *const 对象与 const 函数

(本节内容初学时可以跳过)

1.147.1 const 成员函数

可以将一个成员函数修饰为 `const`，则表示在该函数中不能修改类的成员变量的值。例如，在例 CH20_E1 中，成员函数 `Test()` 被修饰为 `const`，则在 `Test()` 函数里不能修改成员变量的值。

```

    //////////////// 例 CH20_E1 ///////////////////

#include <stdio.h>

class Object
{
public:
    void Test() const // 后面加关键字 const
    {
        //value = 1; // const 函数里不能修改成员变量
        printf("%d \\n", value);
    }
public:
    int value;
};

```

```

int main()
{
    Object obj;
    obj.value = 123;
    obj.Test();
    return 0;
}

```

1.147.2 const 版本与非 const 版本

在类中，成员函数都可以同时定义两个版本：`const` 版本和非 `const` 版本，例如，

////////// 例 CH20_E2 //////////

```

class Object
{
public:
    // const 版本
    void Test() const
    {
        printf("const : %d \n", value);
    }
    // 普通版本
    void Test()
    {
        value += 1;
        printf("non-const: %d \n", value);
    }
public:
    int value;
};

```

那么，当对象是 `const` 类型时，就调用 `const` 版本的 `Test`；当对象是非 `const` 类型时，就调用的是普通版本的 `Test`。例如，

```

void use_it(const Object* p)
{

```

```

    p->Test(); // 调用的是 const 版本的 Test()
}
void use_it2(Object* p)
{
    p->Test(); // 调用的是普通版本的 Test()
}

```

注意：

- (1) 如果只定义了 `void Test() const`，那么 `Object*` 可以调用 `Test()`。
- (2) 如果只定义了 `void Test()`，那么 `const Object*` 不可以调用 `Test()`。

在示例 CH_E3 中，只定义了 `void Test()`，没有定义 `void Test() const`，那么函数 `void use_it(const Object* p)` 在编译时会报错。

////////// 例 CH20_E3 //////////

```

class Object
{
public:
    void Test() // 普通版本
    {
        value += 1;
        printf("non-const: %d \n", value);
    }
public:
    int value;
};
void use_it(const Object* p)
{
    p->Test(); // 编译器报错！找不到 const 版本的 Test()!
}

```

第 21 章 构造与析构

1.148 引例

在 C++ 中，我们引入对象的概念，并且广泛采用面向对象的设计思想来解决问题。那么，随便定义的一个变量就能称之为对象吗？下面，我们举一个例子来说明：**C++ 的对象只在“初始化”之后，才是真正意义上的对象。**

定义一个 Circle 类代表一个坐标圆，其成员变量 x,y 表示圆心坐标，而 radius 则表示圆的半径。我们在 main 函数创建一个对象 c1，并使用它。

```
////////// 例 CH21_A1 //////////  
  
#include <stdio.h>  
  
class Circle  
{  
public:  
    int x, y;  
    int radius;  
};  
  
int main()  
{  
    Circle c1; // 定义一个对象, 并使用它  
  
    printf("location:(%d,%d), radius:%d \n", c1.x, c1.y, c1.radius);  
  
    return 0;  
}
```

运行此程序，可以发现此程序崩溃，或者输出结果混乱。这尴尬的局面：c1 作为一个对象，却无法被使用。它的数据未被初始化，所以都是无效数据。

有人可能会提议，先在代码里检测一下它是否有效，有效之后再使用它。但是我们会发现，根本没有办法检测它的数据是否有效。原因仍然是，它没有被初始化。

所以，我们的结论是：**对象只有在初始化之外才能被使用。**

1.149 构造函数

由于每个类都需要有一个函数对其进行初始化，所以在 C++ 中引入构造函数的概念。构造函数的使命是对生成的对象进行初始化。

构造函数是类的一种特殊的成员函数，它有以下特点：

(1) 函数名与类名必须相同

(2) 没有返回值

例如，下面为类 `Circle` 增加一个构造函数，

////////// CH21_B1 //////////

```
class Circle
{
public:
    Circle () // 构造函数：没有返回值
    {
        x = y = 0;
        radius = 1;
    }
public:
    int x, y;
    int radius;
};
```

1.149.1 重载构造函数

构造函数也可以被重载，这意味着一个类可以有多个构造函数，只要它们的参数列表不同就可以。

```
class Circle
{
public:
    Circle ()
    {
        x = y = 0;
        radius = 1;
    }
    Circle(int x, int y, int r)
    {
```

```

        this->x = x;

        this->y = y;

        this->radius = r;
    }

public:
    int x, y;
    int radius;
};

```

1.149.2 构造函数的调用

构造函数和普通成员函数不一样，一般不显式调用。

在创建一个对象时，构造函数被自动调用。（由编译器完成）

例如，

```

Circle a;

Circle b(1,1, 4);

```

它们在内部实质上是分别调用了不同的构造函数，虽然表面上没有这个函数调用的过程。我们可以单步调试来验证，或者加上一打印语句来验证这个问题。

在示例 CH21_B3 中，在两个构造函数里分别加上打印语句，观察程序运行的结果，可以验证不同的构造函数被调用了。

////////// 例 CH21_B3 //////////

```

#include <stdio.h>

class Circle
{
public:
    Circle ()
    {
        x = y = 0;
        radius = 1;
        printf("构造函数(1) \n");
    }

    Circle(int x, int y, int r)

```

```

{
    this->x = x;
    this->y = y;
    this->radius = r;
    printf("构造函数(2) \n");
}

public:
    int x, y;
    int radius;
};

int main()
{
    Circle a;
    Circle b(1,1, 4);
    return 0;
}

```

构造函数的作用：使得对象在创建时就被初始化，避免了对象未被初始化的情况。通俗点说，对象一“出生”就是有效的，不存在“半成品”的对象。

相对于以前学过的类型的初始化，我们可以比较一下：

```
int a(10); // 将 a 初始化为 10, 也可以写成 int a = 10;
```

```
int* p = NULL; // 将指针初始化为空指针
```

```
Student s={1,"shaofa"}; // 结构体的初始化
```

在 C++ 中，由于构造函数总是被调用，从而保证了对象总是被初始化的，有效避免了“忘记初始化”这种情况的发生。

1.150 析构函数

析构是构造的反过程，表示的是对象的销毁过程。构造函数（**constructor**），是在对象创建时被调用；析构函数（**destructor**），则是在对被销毁时被调用。对于有些对象来说，在它的生命期中申请了一次资源，那么当它的生命期结束时，必须记得释放和归还这些资源。

析构函数也不是普通的成员函数：

（1）名称固定：类名前加上波浪线~

(2) 没有返回值

(3) 不能带参数

例如，

```
class Object
{
public:
    ~Object()
    {
    }
};
```

注：析构函数只能有一个，不允许重载

1.150.1 析构函数的调用

析构函数从不显式地在代码调用，而总是由编译器隐含地调用的。当一个对象生命期结束时，它的析构函数会被自动调用。

示例 CH21_C1 展示了析构函数的隐含调用。

////////// 例 CH21_C1 //////////

```
#include <stdio.h>

class Object
{
public:
    ~Object()
    {
        printf("销毁...\n");
    }
    int value;
};

int main()
{
    {
        Object obj;
```

```

        obj.value = 1;
        printf("..... aaa .....\\n");
    } // 局部变量 obj 生命期到此结束

    printf("..... bbb .....\\n");
    return 0;
}

```

运行此程序，得到的结果如图 21-1 所示。

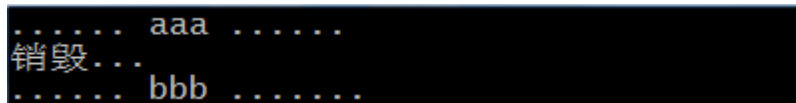


图 21-1 控制台输出显示

可以看出，虽然我们没有显式的去调用析构函数，但是它还是被调用了。当局部变量 `obj` 超出作用域后，该变量失效，对象被销毁。

1.150.2 在析构函数中释放资源

析构函数主要是负责善后和清理工作。其中一个重要的职责就是释放资源。比如，在生命期中曾经申请的内存，曾经打开的文件，此时都应该释放和关闭了。

下面用一个例子来说明资源的释放问题。定义一个类 `Text` 用于存储一个字符串，其内部申请了一块内存，用于保存字符串。

```

////////// 例 CH21_C2 //////////

class Text
{
public:
    Text()
    {
        m_buf = NULL; // 初始化为空指针
        m_size = 0;
    }

private:
    char* m_buf;
    int m_size;
};

```

添加一个 Copy 函数，用于保存一个新的字符串。注意，在保存新的字符串时，要先清理释放原来的内存资源。

```
class Text
{
public:
    // 拷贝一个字符串
    void Copy(const char* str)
    {
        // 释放原来的内存
        if(m_buf)
        {
            free(m_buf);
            m_buf = NULL;
        }
        // 创建一块新内存，拷贝其内容
        m_size = strlen(str)+1;
        m_buf = (char*)malloc(m_size);
        strcpy(m_buf, str);
    }
    ... 其他部分代码省略...
};
```

现在问题来了：此对象在生命期间用 malloc 申请了一块内存，那么在生命结束时应该记得释放这块内存。于是我们添加一个析构函数，

```
class Text
{
public:
    ~Text()
    {
        if(m_buf) free(m_buf); // 释放资源
    }
    ... 其他部分代码省略...
```

```
};
```

至此，Text 类已经比较完善，不会产生内存泄露的情况。

```
int main()
{
    Text t;
    t.Copy("aaaa");
    printf("内容: %s \n", t.GetText());
    t.Copy("bbbb");
    printf("内容: %s \n", t.GetText());
    return 0;
}
```

1.151 自动生成的构造/析构函数

如果我们没有构造函数，或者没有写析构函数，那么编译器会隐含地为我们添加一个。相当于我们写了一个空的构造/析构函数：

```
class Object
{
public:
    Object();
    {
    }
    ~Object()
    {
    }
};
```

然而，只要我们写了一个构造函数，那编译器就不会为我们隐含地生成构造函数了。

1.152 默认构造函数

一个类的构造函数可能有很多个，它们的参数列表不同。把那个不需要传参的构造函数，称为默认构造函数（Default Constructor）。

例如，

```
Object(); // 此构造函数无参数，是默认构造函数
```

或者

```
Object(int a=10, int b=11); // 构造函数的所有参数都有缺省值，也是默认构造函数
```

默认构造函数有什么用呢？

(1) 创建一个对象时，不用指定任何参数

例如，

```
Object obj;
```

(2) 如果没有默认构造函数，则无法创建数组

例如，下面的类 `Object` 由于没有默认构造函数，所以在试图创建数组时编译器会报错。

```
////////// 例 CH21_D1 //////////
```

```
#include <stdio.h>
```

```
class Object
```

```
{
```

```
public:
```

```
    Object(int v)
```

```
    {
```

```
        this->value = v;
```

```
    }
```

```
    int value;
```

```
};
```

```
int main()
```

```
{
```

```
    Object objs[4]; // 编译器报错!Object 类没有默认构造函数可用!
```

```
    return 0;
```

```
}
```

所以，一般情况下最好给类添加一个默认构造函数。

1.153 构造函数的初始化列表

构造函数的作用是对成员变量进行初始化。我们一种两种方式进行初始化，一种是我们已经知道的，

例如，

```
class Circle
{
public:
    Circle()
    {
        this->x = 0;
        this->y = 0;
        this->radius = 1;
    }
private:
    int x;
    int y;
    int radius;
};
```

另一种方式是使用初始化列表。其语法为，在构造函数的小括号（）后面、大括号{}之前，加上一段由冒号引导、用逗号隔开的初始化列表。例如，

////////// 例 CH21_E1 //////////

```
class Circle
{
public:
    Circle():x(0), y(0), radius(1)
    {
    }
private:
    int x;
    int y;
    int radius;
};
```

```
};
```

它的语法要点是：

- ① 位置：位于小括号之前，大括号之前
- ② 以冒号引导，逗开隔开
- ③ 建议按顺序对每个成员进行初始化
- ④ 初始化时用 `m(v)` 的形式，其是 `m` 为成员变量名，`v` 是初始化的值

1.153.1 参数名与成员变量名可以相同

例如，成员变量名是 `a`，传入的构造参数的名称也是 `a`：没有关系，写法不变。

```
class Object
{
public:
    Object(int a, int b) : a(a), b(b)
    {
    }
private:
    int a, b;
};
```

1.153.2 成员本身也是 `class` 类型

当成员类型为 `class` 类型时，也可以在初始化列表中进行初始化。例如，

////////// 例 CH21_E2 //////////

```
class Circle
{
public:
    Circle():x(0), y(0), radius(1), obj(8,8)
    {
    }
private:
    int x,y;
    int radius;
```

```
    Object obj;  
};
```

1.153.3 混合使用两处初始化方式

本节介绍的这两种初始化方式是可以混合使用的。例如，

```
class Circle  
{  
public:  
    Circle(): obj(8,8)  
    {  
        x= y = 0;  
        radius = 1;  
    }  
private:  
    int x,y;  
    int radius;  
    Object obj;  
};
```

1.153.4 必须使用初始化列表的情形

(1) 当成员没有默认构造函数时

在例 CH21_E2 中，由于成员 **Object obj** 它没有默认构造函数，那么只能在初始化列表中对它初始化，否则编译器报错。

(2) 当成员为引用类型时

```
////////// CH21_E3 //////////  
#include <stdio.h>  
class Object  
{  
public:  
    Object(int& a): ref(a)  
    {  
    }  
private:
```

```

        int& ref;
    };
int main()
{
    int value = 1;
    Object obj(value);
    return 0;
}

```

1.154 构造与析构的顺序

在构造时：首先构造每一个成员，调用每个成员的构造函数进行构造。然后执行对象自己的构造函数。当有多个成员时，默认按顺序依次构造每个成员（除非是在初始化列表中指定了初始化的顺序）。

在析构时：首先调用每个成员的析构函数，然后执行对象自己的析构函数。过程与前面恰好相反。当有多个成员时，按相反的顺序依次析构每个成员。

在下面的示例中，将展示这种构造和析构的顺序：

////////// 例 CH21_F1 //////////

```

#include <stdio.h>

class ChildA
{
public:
    ChildA()
    {
        printf("创建 A ...\n");
    }
    ~ChildA()
    {
        printf("销毁 A ...\n");
    }
};

class ChildB
{

```

```

public:
    ChildB()
    {
        printf("创建 B ...\n");
    }
    ~ChildB()
    {
        printf("销毁 B...\n");
    }
};

class Object
{
public:
    Object()
    {
        printf("创建 Object...\n");
    }
    ~Object()
    {
        printf("销毁 Object...\n");
    }
private:
    ChildA a;
    ChildB b;
};

int main()
{
    Object obj;
    return 0;
}

```

程序运行的结果如图 21-2 所示，读者自己自己分析验证一下。



```
创建A ...
创建B ...
创建Object...
销毁Object...
销毁B...
销毁A ...
```

图 21-2 控制台输出显示

1.155 分离式写法

和普通的成员函数类似，构造函数和析构函数也可以把定义写在类之外。在语句上没有什么特别之外，下面给出一个例子，读者直接参考即可。

```
////////// CH21_G1 //////////

class Circle
{
public:
    Circle() ;
    Circle(int x, int y, int radius);
    ~Circle();

private:
    int x, y;
    int radius;
};

Circle::Circle() : x(0), y(0), radius(1)
{
}

Circle::Circle(int x, int y, int radius) : x(x), y(y), radius(radius)
{
}

Circle::~~Circle()
{
}
```

和普通成员函数一样，需要在函数名前加上前缀 `Circle::` 才可以。

1.156*无名对象

如果一个对象只是被临时使用一次，那么可以不为该对象命名。这是一种简洁的写法。

比如，一个类 `Point` 的定义如下，

```
class Point
{
public:
    Point(int x, int y) : x(x), y(y)
    {
    }
    int x;
    int y;
};
```

现在定义一个函数 `Test()`，其参数采用传值方式，

```
void Test(Point pt)
{
    printf("location: x=%d, y=%d \n", pt);
}
```

那么，下面的两行代码，

```
Point p(400,300);
Test(p);
```

可以简化为：

```
Test ( Point(400, 300) );
```

即直接用 `Point(400,300)` 构造一个无名对象，并把它传值给函数。由于这个对象连名字都没有，所以它只能被使用一次。

可见，无名对象的语法用于临时构造一个对象，并使用它的值。例如，

```
Point other = Point(400, 300);
```

其中，等号右侧构造了一个无名对象，作为右值，将值赋给左侧变量。

1.157* 构造函数与类型转换

当构造函数可以传一个参数时，则该构造函数还有一个特殊的使命：类型转换。例如，类 Object 有一个构造函数如下，

```
////////// CH21_H1 //////////  
  
class Object  
{  
public:  
    Object(int v) : value(v)  
    {  
    }  
  
    void Print() const  
    {  
        printf("Value:%d\n", value);  
    }  
  
private:  
    int value;  
};
```

其中的构造函数带了一个参数，所以可以充当类型转换的功能。例如，

```
int main()  
{  
    Object a(1);  
    a = 2; // 发生了隐式类型转换, 相当于 a = (Object) 2;  
    a.Print();  
    a = (Object)3; // 显式类型转换  
    a.Print();  
    return 0;  
}
```

再例如，有两个全局函数 Test 和 Test2，注意它们的参数类型。

```
void Test(Object obj)  
{  
    obj.Print();  
}
```



```
}  
void Test2(const Object& obj)  
{  
    obj.Print();  
}  
int main()  
{  
    Object a(1);  
    Test(4); // 传参时发生隐式类型转换  
    Test2(5); // 传参时发生隐式类型转换  
    return 0;  
}
```

第 22 章 动态创建对象

在引入 class 和构造函数之后，malloc 无法胜任创建对象的功能。引用 new/delete 操作符来动态的创建对象和销毁对象。

1.158 回顾 malloc/free

1.158.1 malloc 可以申请一块内存

在第 12 章中，我们学习了使用 malloc/free 来动态申请一段指定大小的内存。举一个例子，我们要申请一段可以容纳 100 个 int 的空间，则，

```
int* p = (int*) malloc (400);
```

其中，由于 100 个 int 所占的空间是 400 字节，所以要申请 400 字节的空间。显然地，malloc 只是帮我们向内存管理器申请内存，这一块内存的内容是完全无序的。通常，我们申请来内存之后，要对它进行初始化。

比如，全部初始化为 0。

```
for(int i=0; i<100; i++)  
{  
    p[i] = 0;  
}
```

1.158.2 malloc 不是创建对象

malloc 只能为了申请一块内存，但是我们很难把这块内存认为是一个对象，因为它的内容没有被初始化。我们知道，只有经过初始化之后才算是一个有意义的对象。以第 12 章的两个结构体为例，

```
struct Car  
{  
    char maker[32]; // 制造商  
    int price; // 价格  
};  
struct Citizen  
{  
    char name[32]; // 名字  
    int deposit; // 存款
```

```
Car* car; // NULL 时表示没车  
};
```

当我们试图用 `malloc` 来创建一个对象时，

```
Citizen* p = (Citizen*) malloc( sizeof(Citizen));
```

此时的 `p` 指向的只是一块“没有生命”的内存，而不能说它是一个对象。它的内容是无序的，每一个成员的值都可能处于“没有意义”的状态。比如，这个成员 `p->car` 可能是一个野指针。

只有当我们手工为每个成员初始化之后，才勉强算是得到一个对象，像下面的代码这样，我们为 `Citizen` 对象的每一个成员赋以初值，

```
p.name[0] = 0;  
p.deposit = 0;  
p.car = NULL;
```

1.159 用 `new/delete` 创建/销毁对象

在 C++ 中，使用 `new/delete` 这一对操作符来分配和释放对象。需要注意的是，`new` 不但申请内存，而且还把申请来的内存初始化为可用的对象。也许就说，它比 `malloc` 多一个“初始化”的功能。

比如，创建一个 `int` 类型的对象：

```
int* p = new int;  
*p = 100;  
释放  
delete p;
```

再比如，创建一个 `class` 类型的对象，

```
Object* p = new Object; // 创建对象  
p->value = 123;  
p->Print();  
delete p; // 销毁对象
```

`new/delete` 的语法：

当创建/销毁一个类型为 `TYPE` 的对象时，

```
TYPE* p = new TYPE;
delete p;
```

当创建多个对象时，需要在类型后面接一个中括号，中括号中间标明对象的个数。注意，在此种用法下，创建对象用 `new[]`，销毁对象用 `delete[]`，它们是配对使用的。

```
TYPE* p = new TYPE [ N ]; // 创建 N 个对象
delete [ ] p;
```

1.160 new/delete 与 malloc/free 的区别

在 C++ 中，统一使用 `new/delete` 来创建对象，不再使用 `malloc/free`。最根本的原因，是在引入构造/构造的语法之后，`malloc/free` 无法胜任，它们只能申请内存却不能创建对象。

new 操作符：不仅分配了必要的内存，而且调用了相应的构造函数对其初始化，使之成为一个对象。

delete 操作符：先调用相应的析构函数销毁对象，然后再释放相应的内存。

显然，`new/delete` 比 `malloc/free` 的功能增强了。我们通过一个例子来验证一下，在下面这个例子中，我们可以看到在 `new` 的时候 `Object` 类的构造函数被调用，在 `delete` 的时候 `Object` 类的构造函数被调用。

////////// 例 CH22_A //////////

```
#include <stdio.h>

class Object
{
public:
    Object() :value(0)
    {
        printf("构造对象\n");
    }
    ~Object()
    {
        printf("析构对象\n");
    }
}
```

```

void Test()
{
    printf("---- 测试 ----\n");
}

private:
    int value;
};

int main()
{
    Object* p = new Object;
    p->Test();
    delete p;
    return 0;
}

```

运行的结果如图 22-1 所示。

```

构造对象
---- 测试 ----
析构对象

```

图 22-1 控制台输出显示

1.161 为 new 指定初始化参数

在 new 创建一个对象时，如果相应的构造函数有参数，可以在 new 的同时指定其参数。例如，一个类 Circle 表示坐标平面上的圆，它的成员 x,y 表示其圆心坐标，成员 radius 表示圆半径。定义如下，

```

//////////////////// 例 CH22_B //////////////////////////////////
/* Circle: 代表一个圆 */

class Circle
{
public:
    Circle() : x(0), y(0), radius(1)
    {

```

```

    }
    Circle(int x, int y, int radius) :x(x), y(y), radius(radius)
    {
    }
private:
    int x, y; // 圆心坐标
    int radius; // 半径
};

```

它重载了构造函数，一个是默认构造函数，另一个带参构造函数、参数为圆心坐标和半径的值。那么，在用 `new` 创建对象时，可以选择使用哪一个构造函数。例如，

```

Circle* p1 = new Circle(); // 指定使用默认构造函数
Circle* p2 = new Circle; // 同上，两种写法相同
Circle* p3 = new Circle(1,1,4); // 指定使用带参构造函数，传入初始化参数

```

实际上，基本类型的对象如 `int, double` 也是可以在 `new` 的同时指定初始值的。例如，

```

int* a = new int(10);
double* b = new double(12.3);

```

注意，它们使用的是一个小括号，把初始值放在小括号里就可以了。

对于前面我们学习的 `struct` 类型，如果你没有为它们定义构造函数，那就只能在创建对象之后再手工初始化了。（在 C++ 里，`struct` 和 `class` 区别不大，都可以有构造函数了，参考附录《C++ 中 `struct` 和 `class` 的区别》。

```

struct Student
{
    int id;
    char name[128];
};

```

由于没有给它定义构造函数，所以只能像下面这样初始化了：

```

Student* p = new Student;
p->id = 123;
strcpy(p->name, "shao fa");

```

注意：当创建多个对象时，不能指定构造参数，要求必须有一个默认构造函数。这是一个硬件规定。

1.162 默认构造函数的必要性

在这一章可以再次看到默认构造函数的必要性。也就是说，在定义一个类时，最好给它添加一个默认构造函数。否则的话，无法使用 `new []` 来创建多个对象，编译器会报错。

比如，在前面的例子中，`Circle` 中把默认构造函数删掉，只保留带参构造函数，代码如下所示：

```
class Circle
{
public:
    Circle(int x, int y, int radius) :x(x), y(y), radius(radius)
    {
    }
private:
    int x, y; // 圆心坐标
    int radius; // 半径
};
```

那么，就不能使用 `new[]` 来创建多个 `Circle` 对象，下面的语句不能编译通过。

```
Circle* ps = new Circle[12];
```

1.163 注意事项

由于 `new/delete` 也是要向内存管理器来申请的内存，所以 `malloc/free` 里的问题在这里都要注意一下。

- (1) `new` 来的对象，用完了一定记得要 `delete` 掉。
- (2) `new` 与 `delete` 配套使用，`new[]` 与 `delete[]` 配套使用，它们不能混淆使用。
- (3) `new` 来的对象，用完了要及时 `delete`，以免长期占用内存不释放。
- (4) `delete` 之后，该对象不再可用，严禁继续使用。（否则程序崩溃）

关于第(4)条，通常在再 `delete` 操作之后，将指针置为空指针 `NULL`，以免误用。例如，

```
Object* p = new Object();
```

`delete p;` // 此时指向了对象已经失效

`p = NULL;` // 立即置为空指针

第 23 章 继承

继承描述了两个类之前的关系。当类 B 继承于类 A 时，则 A 中的部分成员在 B 类中自动获得。

1.164 引例

在自然界里，存在着“A 是一种 B”这样的关系。其中，A 是一种类型，B 是另一种类型。

例如，

“苹果树 AppleTree ” 是一种 “树 Tree”

“燕子 Swallow ” 是一种 “鸟 Bird”

“小麦 Wheat ” 是一种 “农作物 Crops”

在 C++构成的抽象世界里也存在着这样的关系。

例如，

“视频文件 VideoFile” 是一种 “文件 File”

“MP4 文件 Mp4File” 是一种 “视频文件 VideoFile”

“视频教程 VideoTutorial” 是一种 “教程 Tutorial”

那么，在 C++的语法中如何体现两个类之间的这种关系呢？

1.165 继承的概念

在 C++里，使用继承的语法来表达“B 是一种 A”这种关系。

其语法表示为：

```
class A
{
};

class B : public A
{
};
```

表示类 B 继承于类 A。此时，把 A 称为父类（基类），也 B 称为子类（派生类）。

当 B 继承于 A 时，A 中的所有 public 成员立即被子类 B 继承。

例如，在例 CH23_A1 中，类 Child 继承于类 Base，

////////// 例 CH23_A1 //////////

```
class Base
{
public:
    int x;
    void Print()
    {
        printf("x is %d \n", x);
    }
};

class Child: public Base
{
};
```

看起来子类 Child 里空空如也，但实际上它把父类的 x 和 Print 都继承了过来，可以直接使用。例如，

```
int main()
{
    Child ch;
    ch.x = 1;
    ch.Print();
    return 0;
}
```

子类中只需要把自己的特别的东​​西写出来（特性），而与父类共同的那部分（共性），则通过继承的语法自动得来。

举一个例子，用 Tutorial 表示一部教程，每一个教程应该有一个名字和作者信息。定义这个类如下：

////////// CH23_A2 //////////

```
class Tutorial
{
public:
```

```

    char name[32];
    char author[16];
public:
    void ShowInfo()
    {
        printf("Tutorial: %s, %s\n", name, author);
    }
};

```

然后，一部视频教程 VideoTutorial 也是一种教程，它不但具有“教程”的所有性质，还有自己的特性：它有一个在线播放地址 url，和播放量 visits，它还可以在线播放观看 Play()。定义这个类如下：

```

//////////////////// CH23_A2 //////////////////////
class VideoTutorial : public Tutorial
{
public:
    void Play()
    {
        printf("Playing...abc=%d\n", abc);
    }
public:
    char url[128]; // 在线观看的 URL 地址
    int visits; // 播放量
};

```

所以，当我们创建一个 VideoTutorial 类的对象时，该对象可以像如下方式使用：

```

int main()
{
    VideoTutorial cpp_guide;
    // 当作 Tutorial 类来使用
    strcpy(cpp_guide.name, "C/C++学习指南");
    strcpy(cpp_guide.author, "邵发");
    cpp_guide.ShowInfo();
}

```

```

// 当作 VideoTutorial 类使用

strcpy(cpp_guide.url, "http://111111");

cpp_guide.visits = 10000000;

return 0;

}

```

1.166 访问修饰符 protected

在这一节，我们引入一个新的访问修饰符 `protected`。在 C++ 里的访问修饰符有 3 种：`public`, `private`, `protected`。

当一个类成员被修饰为 `protected` 的时候，有以下规则成立：

- (1) 该成员不能被外部访问，同 `private`
- (2) 该成员可以被子类继承，同 `public`

所以，`public` 和 `protected` 的成员都能够被子类继承。

下面的例子中，演示了 `protected` 修饰的语法特点。

////////// 例 CH23_B1 //////////

```

class Base
{
public:
    Base(): a(1), b(2), c(3) { }

public:
    int a;

protected:
    int b;

private:
    int c;
};

class Child : public Base
{
public:

```

```

void Test()
{
    printf("a=%d \n", a); // a 被继承
    printf("b=%d \n", b); // b 被继承
    // printf("c=%d \n", c); // 语法错！看不见 c，因为 c 没有被继承
}
};

int main()
{
    Child ch;
    ch.a = 101; // a 是 public 的
    //ch.b = 102; // 编译器报错：b 是 protected 修饰的无法访问
    ch.Test();
    return 0;
}

```

注：虽然父类的 `private` 成员变量没有被继承，但是从内存视图上看，子类是包含了父类的所有成员变量的。里面在代码上无法访问父类的 `private` 成员的。

1.167 成员函数的重写

对于从父类继承而来的函数，子类可以重新写一遍，简称“重写”。而且重写时，可以用新的访问修饰符来修饰。

在下面的例子中，父类有一个 `protected` 的函数 `Test()`。子类对其重写，并以 `public` 修饰。则在 `main` 函数中，可以访问子类的 `Test()` 函数。

```

////////// 例 CH23_C1 //////////
#include <string.h>

class Base
{
protected:
    void Test()
    {
        printf("父类 Test...\n");
    }
}

```

```

    }
};

class Child : public Base
{
public:
    void Test()
    {
        printf("子类 Test...\n");
    }
};

int main()
{
    Child ch;

    ch.Test(); // 访问子类的 Test()函数，属性为 public

    return 0;
}

```

有时候函数的重写，只是对父类的函数修补，因此经常会希望在重写的时候调用父类的函数。也就是说，先把父类的那部分代码执行一下，在此基础上然后再补充一点代码。下面的例 CH23_C2 用于展示这个问题：在子类的 Test()函数里调用父类 Base::Test()。

```

////////// 例 CH23_C2 //////////

#include <stdio.h>

class Base
{
public:
    void Test()
    {
        printf("父类 Test...\n");
    }
};

class Child : public Base
{

```

```

public:
    void Test()
    {
        Base::Test(); // 先调用父类的代码
        printf("子类 Test: ...\n");
    }
};

int main()
{
    Child ch;
    ch.Test();
    return 0;
}

```

1.168 虚拟继承

1.168.1 父类指针指向子类对象

可以将父类指针指向一个子类的对象，这是完全允许的。

例如，

```
Tree* p = new AppleTree(); // 左侧为 Tree*，右侧为 AppleTree*
```

从日常生活的逻辑来讲，苹果树是一种树，因而可以把 AppleTree* 视为一种 Tree*。从语法本质上讲，子类对象的前半部分就是父类，因而可以将子类对象的指针直接转化为父类。所以，用父类指针指向子类的对象，是没有问题的。

现在考虑一个问题：对于例 CH23_D1 中定义的两个类 Base 和 Child

```

////////// 例 CH23_D1 //////////

class Base
{
public:
    void Test()
    {
        printf("父类 Test...\n");
    }
}

```

```

    }
};

class Child : public Base
{
public:
    void Test()
    {
        printf("子类 Test...\n");
    }
};

```

下面的代码意味着什么？到底是父类的 Test 还是子类的 Test 被执行了？一方面，p 的类型为 Base*，理应执行 Base::Test()。另一方面，p 指向的那个对象本质是一个 Child*，所以执行 Child::Test()才对。

```

Base* p = new Child();

p->Test();

```

运行一下，我们发现，实际被执行的是 Base::Test()。这和我们期望的也许不太一样：我们认为 p 实际指向了一个 Child 对应，因此 Child::Test()应该被调用。这是一个问题。

1.168.2 虚函数 virtual

解决这个问题的办法是 virtual 语法。将父类的 Test()前面另上 virtual 关键字修饰，这个函数就成为了“虚函数”。我们再作测试，发现 p->Test 输出的子类 Child::Test()，符合我们的预期。代码如下。

```

////////// 例 CH23_D2 //////////

#include <stdio.h>

class Base
{
public:
    virtual void Test() // 加上 virtual 关键字
    {
        printf("父类 Test...\n");
    }
};

```



```

class Child : public Base
{
public:
    void Test()
    {
        printf("子类 Test...\n");
    }
};

int main()
{
    Base* p = new Child;
    p->Test();
    return 0;
}

```

综上，关于虚拟继承，我们要注意两点：

- (1) 如果父类的函数已经声明 **virtual**，那么在子类重写时可以将 **virtual** 省略。反正父类已经将其声明为 **virtual** 了，下面子类的这个函数也就自动 **virtual** 了。
- (2) 如果一个成员函数在设计时就确定了它即将要被子类重写，那么就应该将其声明为 **virtual**。

1.168.3 virtual 析构函数

在继承关系中，父类的析构函数应该被声明 **virtual**，否则，

```

Base* p = new Child;

delete p;

```

将调用父类的构造函数，这将导致严重的错误，因为子类没有被正确析构。

注：构造函数是不能加 **virtual 的。**

1.169 继承关系下的构造与析构

当类 **Child** 继承于类 **Base**，那么当子类对象构造时，会先调用父类的构造函数进行构造，然后再执行自己的构造。当子类对象析构时，过程相反。

先看一个例子，在例 **CH23_E1** 中，我们有父类 **Base** 的定义如下，

```

////////// 例 CH23_E1 //////////

```

```

class Base
{
public:
    Base() : x(0), y(0)
    {
        printf(" .....base destroy ....\n");
    }
    Base(int x, int y): x(x), y(y)
    {
        printf(" .....base create: x=%d, y=%d ....\n", x, y);
    }
    ~Base()
    {
        printf(" .....base destroy ....\n");
    }
public:
    int x, y;
};

```

我们定义子类 Child 如下，在子类中，并没有显式地调用父类的构造函数和析构函数，

```

class Child : public Base
{
public:
    Child()
    {
        printf(" .....child create ....\n");
    }
    ~Child()
    {
        printf(" .....child destroy ....\n");
    }
};

```

但我们在 main 函数测试一下，可以发现父类的构造函数和析构函数都被调用了。

```
int main()
{
    Child c1;
    return 0;
}
```

当没有显式调用父类的构造函数时，父类的默认构造函数被调用。我们注意到父类 **Base** 重载了构造函数，我们可以使用下的写法来显式的指定调用父类的某个构造函数：

```
class Child : public Base
{
public:
    Child() : Base(1, 2) // 显式调用父类的构造函数
    {
        printf(" .....child create ....\n");
    }
    ... 其余部分代码省略...
};
```

1.170*多重继承

（初学者请跳过这一节）

在 C++ 里允许多重继承，一个类可以同时继承于多个父类。定义这个语法的本意：一个孩子有父有母，可以从父母处各自继承一些特点。

语法：

假设用 **Father**, **Mother** 表示二个类，现在定义一个子类 **Child**，同时继承于 **Father** 和 **Mother**，语法形式如下：

```
class Child : public Father, public Mother
{
};
```

在写法上，以冒号引导，每个父类用逗号隔开。

多重继承的结果：从所有父类中，继承他们所有可以被继承的成员（**public/protected**）。例如，在例 CH23_F1 中，类 **Child** 继承于 **AAA1** 和 **AAA2**，

////////// 例 CH23_F1 //////////

```
class AAA1
{
public:
    int x, y;
};

class AAA2
{
public:
    int u, v;
};

class Child : public AAA1, public AAA2
{
public:
    int a, b;
};

int main()
{
    Child ch;

    ch.x = ch.y = 1; // 继承于 AAA1
    ch.u = ch.v = 2; // 继承于 AAA2
    ch.a = ch.b = 3; // 自己的成员

    return 0;
}
```

多重继承的语法是有明显缺陷的：当两个父类有重名的成员时，就给编译器带来了困扰。实际上，多重继承的理念并不现象，我们也很少为一个孩子找多个父类。在 C++ 中，多重继承的语法被变相用于实现“接口”这个设计方法，这在下一节“纯虚函数”里会介绍。

1.171 *继承函数与纯虚类

纯虚函数是一种特殊的虚函数。

纯虚函数的语法：

- (1) 将成员函数声明为 `virtual`

(2) 后面加上 = 0

(3) 该函数不能有函数体

例如，

```
class CmdHandler
{
public:
    virtual void OnCommand(char* cmdline) = 0;
};
```

这样就可以了。包含了纯虚函数类称为纯虚类。通常，也把纯虚函数称为抽象函数，纯虚类称为抽象类。

纯虚类不能够被实例化。例如，下面的写法会被编译器报错：

```
CmdHandler handler; // 编译器报错！
```

纯虚类不能够被实例化。那么，一个能不能够被实例化，还定义它做什么用呢？

实际上，C++里的纯虚类是用于实现设计模式里的“接口”的概念，其地位相当于 Java 里的 interface 语法，或者用于替代 C 语言中的回调机制。它总是用来被人继承的，它只是用于声明“我的子类应该实现这些函数接口”。

1.172 *以 protected/private 方式继承

(初学者请跳过一节。本节语法几乎没用，既不常见，也不常用)

在实际应用中，我们总是使用 public 方式继承。但是在语法上，其实存在三种方式的继承：

```
class B : public A {} ; // public 继承
```

```
class B : protected A {} ; // protected 继承
```

```
class B : private A {} ; // private 继承
```

当 protected 继承时，子类继承了父类的 public/protected 成员，并将继承来的成员的属性降低为 protected。

当 private 继承时，子类继承了父类的 public/protected 成员，并将继承来的成员的属性降低为 private。

```
////////// 例 CH23_H1 //////////
```

```
class Base
```

```
{
public:
    int x;
};
class Child : protected Base
{
};
int main()
{
    Child ch;
    ch.x = 10; // 编译错误! x 被调整为 protected, 外部无法访问
    return 0;
}
```

第 24 章 拷贝构造函数

本章介绍一种特殊的构造函数：拷贝构造函数，它在对象拷贝时被调用。

1.173 定义

拷贝构造函数 (Copy Constructor)，它是一种特殊的构造函数。

(1) 它是构造函数：所以函数名是类名、没有返回值

(2) 它的特殊性：参数形式固定

例如，下面定义的 `Object` 类中，这样的构造函数就是拷贝构造函数。

```
class Object
{
public:
    Object( const Object& other ) // 这是拷贝构造函数
    {
    }
};
```

拷贝构造函数的含义：以一个对象为蓝本，来构造另一个对象。例如，

```
Object b;
```

```
Object a(b); // 或写成 Object a = b;
```

我们把它称作：“以 `b` 为蓝本，创建一个新的对象 `a`”。这意味着，`a` 是 `b` 的一个拷贝，两者的内容应该完全相同。

1.174 拷贝构造函数的调用

拷贝构造函数从不被显式地调用，和普通的构造函数一样，它总是被编译器隐含地调用的。以下三种情况下，拷贝构造函数被调用。

(1) 定义对象

```
Object a;
```

```
Object b(a); // 或写成 Object b = a;
```

注意，这两种写法是完全相同的。

(2) 动态创建对象

```
Object a;
```

```
Object* p = new Object(a); // 注意参数类型的匹配
```

(3) 函数的传值调用

```
void Test(Object obj);
```

我们知道，在传值方式中，参变量是一个原变量的拷贝。例如，

```
Object a;
```

```
Test (a);
```

则创建了一个完全相同的参变量，相当于在 Test 函数定义了一个局部变量 Object obj (a)，这里就会自动调用拷贝构造函数。

我们可以定义一个 Object 类，在拷贝构造函数里加上若干打印语句，来验证上面的结果。例如，

```
////////// 例 CH24_A //////////
```

```
#include <stdio.h>
```

```
/* Circle: 代表一个圆 */
```

```
class Object
```

```
{
```

```
public:
```

```
    Object(int v) : value(v)
```

```
    {
```

```
        printf("构造对象\n");
```

```
    }
```

```
    Object(const Object& other)
```

```
    {
```

```
        this->value = other.value;
```

```
        printf("拷贝构造对象\n");
```

```
    }
```

```
private:
```

```
    int value;
```

```
};
```

```
void Test(Object obj) // 传值调用时，调用了拷贝构造函数
```



```

{
}

int main()
{
    Object a ( 123 );

    Object b ( a ); // 测试 1: 以 a 为蓝本创建对象 b

    Object* p = new Object(a); // 测试 2

    Test(a); // 测试 3

    return 0;
}

```

1.175 默认的拷贝构造函数

通常情况下，我们不需要添加拷贝构造函数，编译器会默认配备一个拷贝构造函数。而且，默认的构造函数通常是足够用的。

默认的拷贝构造函数做些什么事情呢？正像我们所期望的那样，它帮我们完成了对象的拷贝，其规则为：

(1) 如果这个类有父类，则先调用父类的拷贝构造函数。

(2) 对每个成员依次拷贝。如果这个成员是一个基本类型(int,double)等，则直接按值拷贝。如果这个成员是一个自定义 class/struct 类型，则调用其拷贝构造函数进行拷贝。

举例来说，我们有一个类 Object，有若干成员变量，那么我们不需要做任何事情，默认就是能正确的拷贝的。在下面的代码中，对象 b 和 a 的内容相同。

```

////////// 例 CH24_B //////////

#include <stdio.h>

#include <string.h>

class Object
{
public:
    Object(int id, const char* name)
    {
        this->id = id;
        strcpy(this->name, name);
    }
}

```

```

    }

private:
    int id;
    char name[128];
};

int main()
{
    Object a ( 1, "some" );
    Object b ( a );
    return 0;
}

```

1.176 定义拷贝构造函数

由于默认的构造函数往往已经够用，我们一般是不必自己添加一个拷贝构造函数的。

一旦我们决定了要自己写一个拷贝构造函数，就必须完全明白自己在做什么。这是一件很复杂的事情，很容易遗漏。

(1) 首先，要调用父类的拷贝构造函数

否则的话，父类的成员，以及父类的父类的成员，是不会被拷贝的。

(2) 其次，依次构造每一个成员

当我们自己添加一个拷贝构造函数后，编译器不再为我们做任何事情，所有的成员的复制工作都由我们负责。因此，所有的成员都有挨个复制一遍，不能有遗漏。

所以，如果一个类 `Object` 继承于父类 `Base`，那么 `Object` 类的拷贝构造函数通常是这么写的：

```

////////// 例 CH24_C //////////

#include <stdio.h>

#include <string.h>

class Base // 父类
{
public:
    int m1;

```

```

};

class Object : public Base // 继承于父类
{
public:
    Object(int id, const char* name)
    {
        this->id = id;
        strcpy(this->name, name);
    }
    Object(const Object& other)
        : Base(other) // 调用父类的拷贝构造函数
    {
        // 依次复制所有成员
        this->id = other.id;
        strcpy(this->name, other.name);
    }
private:
    int id;
    char name[128];
};

int main()
{
    // 测试
    Object a ( 1, "some" );
    a.m1 = 123;
    Object b ( a );
    return 0;
}

```

我们可以试验一下，假设你不调用 `Base(other)` 这样的语句，那么 `b.m1` 的值和 `a.m1` 的值就不一样。假设我们不写 `this->id = other.id`，那么 `id` 的值也不会被复制。

1.177 深度拷贝

既然拷贝构造函数如此复杂，写了甚至还不如不写，那为什么还要发明这个语法呢？这一节我们将阐述：在何种情况下才需要添加一个拷贝构造函数。

考虑下面的 `Text` 类，它的设计意图是用一个对象来保存一段文本，而文本的长度不是固定的。注意，里面用一个指针来指向一块动态申请而来的内存，在此内存里保存了文本内容。

//////////////////// 例 CH24_D //////////////////////

```
#include <stdio.h>
#include <string.h>
// Text 类中没有定义拷贝构造函数

class Text
{
public:
    Text(const char* str)
    {
        // 申请一块内存, 保存此字符串
        m_size = strlen(str) + 1;
        m_buf = new char[m_size];
        strcpy(m_buf, str);
    }
    ~Text()
    {
        // 释放此字符串
        delete [] m_buf;
    }
private:
    int m_size;
    char* m_buf;
};
```

```

int main()
{
    // 定义第一个对象
    Text t1("helloworld");
    // 第二个对象以 t1 为蓝本进行拷贝
    Text t2(t1);
    return 0;
}

```

在此种情况下，默认的拷贝构造函数还能胜任吗？我们直接运行例 CH24_D 的程序，可以发现程序会崩溃。为什么呢？

t2 是 t1 的拷贝，默认的拷贝结果为：t2.m_size 等于 t1.m_size，t2.m_buf 等于 t1.m_buf。这是致命的。字符串的复制不能用指针直接赋值。

所以，出错的原因为：当 t1 被析构时，t1.m_buf 指向的内存被 delete 掉。而 t2 被析构时，t2 的析构函数被调用、试图去 delete t2.m_buf、而这一块内存已经被 delete 掉了。（注：t1 和 t2 谁先被析构，并不影响结论）。

用一句话来归结这个问题：两个对象 t1,t2 拥有同一块内存。而设计的初衷却是，两个对象各自拥有一块内存。我们把这个问题称为“浅拷贝和深拷贝”。默认的拷贝构造函数进行的是“浅拷贝”，即只对指针进行拷贝、并没有复制内存。我们应该添加一个拷贝构造函数，进行“深拷贝”来解决这个问题。（在第 15 章曾初步解释过这个议题）

为 Text 类定义一个拷贝构造函数，申请一块内存，将 other 里面字符串的内容拷贝过来，这就叫“深拷贝”。

```

Text(const Text& other)
{
    m_size = other.m_size;
    m_buf = new char[m_size];
    strcpy(m_buf, other.m_buf);
}

```

通过实现一个拷贝构造函数，就完成“深拷贝”，两个 Text 对象各自拥有自己的一块字符串内存。

注：如果你觉得深度拷贝在实现起来比较麻烦，可以简单一点、直接禁止用户做拷贝动作。方法是，添加一个拷贝构造函数并用 private 修饰。例如，

```

private:
    Text(const Text& other) { } // 这是一个偷懒的替代解决方案

```

第 25 章 静态成员

在 C++ 中，引用了 `static` 关键字用于定义全局变量和全局函数。

1.178 static 定义全局变量

在 C++ 中如何定义一个全局变量呢？

第（1）种方式：传统方式

```
////////// Object.h //////////  
  
extern int number; // 全局变量的声明  
  
////////// Object.cpp //////////  
  
int number = 1; // 全局变量的定义  
  
////////// main.cpp //////////  
  
#include <stdio.h>  
#include "Object.h"  
  
int main()  
{  
    printf("number: %d \n", number);  
    return 0;  
}
```

第（2）种方式：static 方式

在 C++ 中，引入另一种定义全局变量的方式。

```
////////// CH25_A1: Object.h //////////  
  
class Object  
{  
public:  
    static int number; // 声明 static 变量  
};  
  
////////// CH25_A1: Object.cpp //////////  
  
#include "Object.h"  
  
int Object::number = 1; // static 变量的定义
```

```

////////// CH25_A1: main.cpp//////////
#include <stdio.h>
#include "Object.h"
int main()
{
    Object::number = 2;
    printf("number: %d \n", Object::number);
    return 0;
}

```

我们注意到，static 变量的写法包含以下几项要素：

（1）声明

把声明放在类体大括号之内。和普通的全局变量声明一样，区别只是将关键词 `extern` 换成 `static` 修饰。

如，

```

class Objectt
{
public
    static int number;
    static float scores[3];
};

```

显然，在变量声明语句中是不能加初始值的。

（2）定义

变量定义语句放在类体之外。和普通的全局变量定义一样，区别只是将变量名用类范围符来限定。例如，

```

int Object::number = 1;
float Object::scores[3] = { 80.0f, 90.5f, 100.0f };

```

（3）使用

在使用 static 变量时，就和使用普通的全局变量一样，区别只是要在变量名前加上类范围符的前缀。例如，

```
int main()
{
    Object::number = 2;
    printf("number: %d \n", Object::number);

    Object::scores[0] = 70.0f;
    Object::scores[1] = 75.0f;
    Object::scores[2] = 80.0f;
    return 0;
}
```

注意：虽然这种变量在形式是放在了类的大括号内，但和这个类其实没多少关系。记住它本质上就是全局变量即可，使用起来总是要加上类范围符的前缀。

1.179 static 定义全局函数

在 C++ 中，如何定义一个全局函数呢？

第（1）种方式：传统方式

```
////////// Object.h //////////
extern void Test(); // 全局函数的声明

////////// Object.cpp //////////
// 全局函数的定义

#include <stdio.h>

void Test()
{
    printf("This is a test ...\n");
}

////////// main.cpp //////////
#include "Object.h"

int main()
{
```



```

    Test();
    return 0;
}

```

第（2）种方式：static 方式

```

////////// CH25_A2: Object.h //////////

```

```

class Object
{
public:
    static void Test(); // 声明 static 函数
};

```

```

////////// CH25_A2: Object.cpp //////////

```

```

#include <stdio.h>
#include "Object.h"
void Object::Test()
{
    printf("This is a test ...\n");
}

```

```

////////// CH25_A2: main.cpp //////////

```

```

#include <stdio.h>
#include "Object.h"
int main()
{
    Object::Test();
    return 0;
}

```

我们注意到，static 函数的写法包含以下几项要素：

（1）声明

把声明放在类体大括号之内。和普通的全局函数声明一样，区别只是将关键词 `extern` 换成 `static` 修饰。

如，

```
class object
{
public:
    static void Test(); // 声明 static 函数
};
```

(2) 定义

函数的定义必须在类体之外。和普通的全局函数定义一样，区别只是将函数名用类范围符来限定。例如，

```
void Object::Test() // 函数名前面须加上类范围符
{
    printf("This is a test ...\n");
}
```

(3) 使用

在使用 `static` 函数时，就和使用普通的全局函数一样，区别只是要在函数名前加上类范围符的前缀。例如，

```
int main()
{
    Object::Test();
    return 0;
}
```

1.180 与普通成员的区别

`static` 成员其实就是全局变量/函数，和类的普通成员变量和成员函数没什么关系。

(1) `static` 变量不属于这个类

例如，

```
class Object
{
```

```

public:
    int a;
public:
    static int b;
    static int c;
};
int Object::b = 0;
int Object::c = 0;

```

那么 `sizeof(Object)` 的值为 4，因为它包含了一个成员变量 `a`，而另外两个变量 `b` 和 `c` 和它是没有关系的。

(2) static 函数里没有 this 指针

在第 20 章我们已经清楚，`this` 指针传递的是一个对象。但是显然地，我们在调用 `static` 函数的时候，并没有传递任何对象，甚至根本没有对象被创建。所以不可能有 `this` 指针。

从另一种角度想，`static` 函数就是全局函数而已，哪来的 `this` 指针呢？如果实在理不了的话，就把这个函数挪到类的外面，应该就可以豁然开朗了。

1.181 static 语法的特点

从前面可以看出，`static` 的语法和传统的全局变量/全局函数，除了在形式上有所差别，实质上并无差别。在使用起来也很类似，只是加上了一个类范围符作为前缀。

我们比较一下，`static` 函数和普通全局函数存在两个主要的不同点：

1.181.1 受 public/private 限制

这意味着，如果该函数被 `private` 限制，则不允许外部访问该函数，只能够类内其他函数访问。

在示例 CH25_B1 中，`Object::Test()` 被 `private` 限制，所以在 `main` 函数里不能调用它。但是在成员函数 `Object::Call` 中是可以调用 `Test()` 的。注意，从内部访问 `static` 成员时是不需要加前缀的。

```

////////// CH25_B1: Object.h //////////
class Object
{
public:

```

```

    void Call();

private:
    static void Test(); // 声明 static 函数
};

////////// CH25_B1: Object.cpp//////////
#include <stdio.h>
#include "Object.h"
// static 全局函数
void Object::Test()
{
    printf("This is a test ...\n");
}
// 类的成员函数
void Object::Call()
{
    Object::Test(); // 从内部调用时，前缀 Object::可以省略
}

////////// CH25_B1: main.cpp//////////
#include <stdio.h>
#include "Object.h"
int main()
{
    // Object::Test(); // 此函数不允许被外部调用
    Object obj;
    obj.Call(); // Call()里间接调用了 Object::Test()
    return 0;
}

```

1.181.2 可以自由访问类的其他成员

普通的全局函数是无法访问类的私有成员的。但 `static` 函数可以访问类的所有成员，这算是它的一个小小的优势。

```
////////// CH25_B2: Object.h //////////  
  
class Object  
{  
private:  
    void DoSomething();  
    int x, y;  
  
public:  
    static void Test(Object* obj); // 声明 static 函数  
};
```

类 `Object` 里有成员 `DoSomething()` 和 `x,y` 都是 `private` 的，外部无法访问。而 `Object::Test()` 是 `public` 的，外面可以访问。

```
////////// CH25_B2: Object.cpp //////////  
  
#include <stdio.h>  
#include "Object.h"  
  
// static 全局函数  
void Object::Test(Object* obj)  
{  
    obj->x = 1;  
    obj->y = 2;  
    obj->DoSomething();  
}  
  
// 类的成员函数  
void Object::DoSomething()  
{  
    printf("let's do something ...\n");  
}
```

由 Test()是全局函数，所以不存在什么 this 指针，需要显式的传入一个对象。

```
////////// CH25_B2: main.cpp//////////  
  
#include <stdio.h>  
#include "Object.h"  
  
int main()  
{  
    // Object::Test(); // 此函数不允许被外部调用  
  
    Object obj;  
  
    Object::Test(&obj); // 显式的传入一个对象指针  
  
    return 0;  
}
```

1.182 实例 1：实例计数

在定义某些模块时，可能会使用 static 语法。它的一个优点是可以避免名字冲突，并可以施加 public/private 限制。

下面就介绍一个经典的应用场景：实例计数。前面已经定义了一个类叫 Object，我们想知道在程序的运行过程中，该类产生了多少个实例对象。我们知道无论是 new 出来的对象，直接定义的对象，在创建时总是调用了构造函数/拷贝构造函数，在销毁时总是调用了构造函数。因此可以定义一个静态的成员变量来表示实例数目，在实例被创建的时候加 1，在被销毁的时候减 1。（当然，你用一个普通的全局变量来表示也完全没问题的）。

```
////////// CH25_C1 : Object.h //////////  
  
class Object  
{  
private:  
    static int count ;  
public:  
    static int GetInstanceCount();  
public:  
    Object ();  
    Object (const Object& other);
```

```

    ~Object();
};

////////// CH25_C1 : Object.cpp //////////

#include "Object.h"

// 程序运行时，实例数为 0，没有任何对象被创建

int Object::count = 0;

Object::Object()
{
    count ++; // 创建了一个对象，计数增 1
}

Object::Object(const Object& other)
{
    count ++; // 通过拷贝构造创建了一个对象，计数增 1
}

Object::~~Object()
{
    count --; // 销毁了一个对象，计数减 1
}

int Object::GetInstanceCount()
{
    return count; // 当前运行状态下，有多个对象 Object
}

////////// CH25_C1 : main.cpp //////////

#include <stdio.h>

#include "Object.h"

int main()
{
    Object a1;

    Object* a2 = new Object();

    Object a3(a1);

    printf("count: %d \n", Object::GetInstanceCount());
}

```

```
    return 0;
}
```

1.183 经典应用场景二：单例模式

单例模式是工程开发中经常运行的一种手段：控制某个类的对象的数目，保证在运行时刻全局使用唯一的一个实例。例如，要设计一个系统，系统中有一台打印机 `Printer`，该对象跟一台物理打印机联系。系统中的其他各个模块都可能使用这个打印机，那么肯定不能允许让各个模块都 `new` 一个 `Printer`，因为一共只有一台。这件事不能在口头上告诉项目组的各个开发人员，说你在写代码的时候不能 `new Printer()`，这样是不保险的。

作为 `Printer` 类的提供者，在定义 `Printer` 类的时候就要控制它，让它不可能被别人 `new` 出来多个实例，这种设计方法采用的就是“单例模式”。

首先，将类的构造函数定义为 `private`，这样就避免了别的模块自作主张创建一个对象出来。

```
class Printer
{
private:
    Printer() {};
```

```
    ~Printer() {};
```

```
}
```

显然，由于构造函数被限制为 `private` 了，所以其它项目组成员如果想在他们的代码里来创建 `Printer` 对象，则直接代码根本编译不通过。比如，

```
Printer my_printer; // 编译器报错！构造函数已经被限制为 private！
```

```
Printer* my_printer = new Printer(); // 编译器报错！构造函数已经被限制为 private！
```

单例模式的核心设计思想：（1）不允许外部创建实例 （2）在模块内部创建和管理实例。下面是一种实现方式：

```
////////// CH25_D1 : Printer.h //////////
class Printer
{
private:
    Printer() {}; // 将构造函数修饰为 private
    ~Printer() {};
```

```
public:
```



```
static Printer* GetInstance();  
private:  
    static Printer* instance;  
};  
外部只能通过 Printer::GetInstance()来获取对象的实例。
```

```
////////// CH25_D1 : Printer.cpp //////////  
#include <stdio.h>  
#include "Printer.h"  
Printer* Printer::instance = NULL;  
Printer* Printer::GetInstance()  
{  
    if(instance == NULL)  
    {  
        instance = new Printer();  
    }  
    return instance; // 返回该实例的指针  
}
```

在模块内部创建和管理实例。

```
////////// CH25_D1 : main.cpp //////////  
#include <stdio.h>  
#include "Printer.h"  
int main()  
{  
    Printer* pr = Printer::GetInstance();  
    return 0;  
}
```

外部通过 static 函数来获取实例。

容易发现，我们可以将“单例模式”扩展到有限个实例的场景。比如，只允许 2 个实例。其核心设计思想都是一样的：在模块内部创建和管理多个实例，不允许在外部创建。

第 26 章 朋友成员

1.184 引例

假设一个公司有工程师 `Engineer`，项目经理 `PM`，技术总监 `CTO`，它们的职责关系是 `Engineer` 受 `PM` 管理，`PM` 受 `CTO` 管理。

现在，定义一个 `PM` 类，它拥有一个成员变量 `m_report`，表示周报内容。

```
class PM
{
private:
    WeeklyReport m_report;
};
```

从公司流程规范上，`PM` 的周报只能被 `CTO` 阅读，而普通的 `Engineer` 是无权阅读的。在此，已经将 `m_report` 用 `private` 修饰，保证了所有其他人都无法阅读 `PM` 的周报。那么问题来了，`Engineer` 无权阅读，而 `CTO` 有权阅读，如何在 C++ 里表示这种关系？

1.185 类的朋友

通过将类 `CTO` 声明为 `PM` 的“朋友”，但可以授权 `CTO` 类访问 `PM` 类的所有成员。这个语法通过 `friend` 声明语句完成。

```
class PM
{
    friend class CTO; // 将 class CTO 声明为 PM 的朋友
private:
    WeeklyReport m_report;
};

class CTO
{
public:
    void Read(PM& pm)
    {
        // 可以访问 pm.m_report
    }
}
```

```
};
```

1.186 friend 的语法

通过 **friend** 关键字，可以将一个全局函数 **func**、或者一个类 **B**，声明为类 **A** 的函数。类的朋友可以自由的访问类 **A** 的所有成员，不受访问修饰符的限制。

其语法为：

```
class A
{
    friend void func ( A* p ); // 将全局函数 func 声明为 “朋友”

    friend class B; // 将类 B 声明为 “朋友”

    // 构造函数

    A()
    {
    }

};
```

(1) **friend** 声明语句的位置：放在类体的大括号里，但位置是自由的，可以靠上也可以靠下。习惯上把 **friend** 声明语句放在大括号内的最前面。

(2) 朋友拥有的权利：可以无限制地访问类 **A** 的所有成员，不受 **private/protected** 的限制。

(3) 类 **B** 被声明为 **A** 的朋友，则 **B** 可以访问 **A** 的所有成员。反之不成立：**A** 不是 **B** 的朋友，**A** 无法访问 **B** 的所有成员。这意味着，朋友关系是“单向”的。

把一个全局函数声明为朋友时，需要列出其函数原型，前面加一个关键字 **friend**。

把一个类声明为朋友时，需要在类名前加上 **friend class** 关键字。如例中所示。

1.187 实例 1

```
////////// 例 CH26_A ////////////

#include <stdio.h>

// 已经一个类 Object，一个全局函数 Print()

class Object
{
```

```

    // 将 Print() 声明为 Object 的朋友
    friend void Print(Object* p);
public:
    Object(int v) : value(v)
    {
    }
private:
    int value;
};
// 在全局函数里，访问 Object 的 private 成员
void Print(Object* p)
{
    printf("value: %d \n", p->value);
}
int main()
{
    Object obj(123);
    Print(&obj);
    return 0;
}

```

读者可以自己试验一下：如果不加上 `friend` 声明，则编译器会直接报告错误，因为在 `Print` 中无法访问 `Object` 的私有成员。而加上 `friend` 声明之后就不报错了。

1.188 实例 2

```

////////// 例 CH26_B //////////
/***** 在 CTO 类中查看 PM 类的周报 *****/
#include <stdio.h>
#include <string.h>
// 项目经理: Project Manager
class PM
{

```

```

        friend class CTO;

public:
    PM(const char* name, const char* text)
    {
        strcpy(m_name, name);
        strcpy(m_report, text);
    }
private:
    char m_name[64]; // 经理的名字
    char m_report[256]; // 这一周的周报
};

// 首席技术官
class CTO
{
public:
    // 阅读项目经理的周报
    void ReadReport(PM& p)
    {
        printf("[ %s ] : %s \n", p.m_name, p.m_report);
    }
};

int main()
{
    PM xiaoming("xiaoming", "这一周干了惊天动地的事情");
    CTO mr ;
    mr.ReadReport(xiaoming);
    return 0;
}

```

1.189 进一步地讨论

`friend` 语法有优点，也有缺点，但总体是是缺点大于优点。优点是，它给设计人员带来了一点的灵活度，必要的时候就加上 `friend` 声明，从而破除 `private/protected` 的限制。但这是有代价的：它破坏了类的封装的原则，使得设计有点混乱。

那么，`friend` 语法一般在什么时候使用呢？

当你要设计一个独立功能的模块时，该模块可能是由若干个类组成的。此时，这些模块内部的类之间可以声明为 `friend`，以减少不必要的限制。最终的结果时，模块内部的这些类之间可以互相自由访问，但外面的类则严格受到访问修饰符的限制。从情理上也容易理解这个语法：它们内部是一伙的，因而可以彼此无障碍地访问。

第 27 章 重载操作符

重载操作符，使得一个自定义的类型可以像基本类型一样支持加减乘除等多种操作符。

1.190 引例

C++引入 struct/class 语法的初衷是允许用户定义自己的数据类型，称为自定义类型。而且 C++的设计者是理想主义者，其终极理想是要让自定义类型和基本类型用起来完全一样。怎么才算是一样呢？

以一个表示分数的类 Fraction 作为例子，它具有两个成员变量 num(分子)和 den(分母)。下面是这个类的定义：

```
////////// 例：CH27_A1 //////////  
/* Fraction: 一个表示分数的类 */  
  
class Fraction  
{  
public:  
    Fraction(): num(1), den(1)  
    {  
    }  
    Fraction(int n, int d) : num(n), den(d)  
    {  
    }  
    int num; // 分子  
    int den; // 分母  
};
```

对比一下对基本数据类型的操作。如果我们定义了两个 int 型的 a,b 那么可以对它进行加减乘除赋值等操作，就像下面这样，

```
int a = 2;  
  
int b = 3;  
  
int c = a + b; // 加法操作
```

然而，对于 Fraction 类却不可以施加相同的操作，语法上是不支持的。也就是说，原本我们希望 Fraction 类也可以支持算术操作和赋值操作，但实际上却不可以，编译器会报错。例如，

```
int main()
```



```

{
    Fraction fa(2,3); // 表示 2/3

    Fraction fb(3,5); // 表示 3/5

    Fraction fc = fa + fb; // 语法错误

    return 0;
}

```

本章讨论的就是，如何让一个 `class` 类型支持这些操作符，使用的语法就称为“重载操作符”。

1.191 算术操作符

算术操作符包含 `+` `*` `%` 等，本节以加法操作符为例，介绍重载操作符的一般实现方法。算术操作符的重载一般有两种写法，一种是重载类的操作符，另一种是重载全局操作符。下面以引例中的 `Fraction` 为例，展示重载操作符的实现方法。

(1) 第一种方式：类操作符

此时，操作符函数是类的成员函数，

////////// 例 CH27_A2 //////////

```

class Fraction
{
public:
    // 重载加号操作符

    Fraction operator + (const Fraction& other)
    {
        Fraction result;

        result.den = den * other.den; // 分母相乘

        result.num = num*other.den + den * other.num; // 分子交叉相乘

        return result;
    }
    ... 其他部分代码省略 ...
};

```

(2) 第二种方式：全局操作符

此时，重载的是全局操作符，需要把一个操作符函数（类似于全局函数）声明为类的朋友，代码摘要如下，

```
////////// 例 CH27_A3 //////////  
  
class Fraction  
{  
    friend Fraction operator + (const Fraction& a, const Fraction& b);  
    ... 其他部分代码省略 ...  
};  
  
Fraction operator + (const Fraction& a, const Fraction& b)  
{  
    Fraction result;  
    result.den = a.den * b.den; // 分母相乘  
    result.num = a.num * b.den + a.den * b.num; // 分子交叉相乘  
    return result;  
}
```

重载操作符在形式上类似于函数，有一个固定的名字，带一个参数列表。但不能把它等同于函数，因为有的操作符在形式上是比较奇怪的。下面列出几点注意事项：

- (1) 重载操作符的形式上像一个函数，但它不是函数
- (2) 名称：operator +，这个是固定不变的
- (3) 返回值：类型不变，总是为该对象的类型
- (4) 参数：基本上也不变
- (5) 受 public/protected/private 的限制

1.191.1 参数类型

加法操作符的参数，其参数相当于加号右边的那个操作数（右操作数）。以上的代码中，演示的是两个 Fraction 类型的相加。而实际上，一个分数和一个整数也应该是能够相加的，例如，下面的代码应该被支持：

```
int main()  
{  
    Fraction fa(2, 3);  
    Fraction fc = fa + 4;  
}
```

```

    return 0;
}

```

于是要求我们为 Fraction 类重载一个右操作数类型为 int 的加法操作符，代码如下：

////////// 例 CH27_A4 //////////

```

class Fraction
{
public:
    // Fraction + int
    Fraction operator + (int n)
    {
        Fraction result;
        result.num = num + den * n; // 分母不变，分子变化
        return result;
    }
    ... 其他部分代码省略 ...
};

```

1.191.2 加法的互换性

到目前为止，我们已经支持了 Fraction + Fraction 和 Fraction + int 的支持，但是还有一个缺点。我们知道加法是可以把左右操作数互换位置的， $a + b$ 和 $b + a$ 应该效果一样才对。但我们在代码中还不支持。例如，下面的代码中把一个 int 与一个 Fraction 相加，编译器会报错。

```

int main()
{
    Fraction fa(2, 3);
    Fraction fc = 4 + fa; // int + Fraction 还不支持
    return 0;
}

```

为此，考虑使用全局操作符来实现会比较方便一些。

```

friend Fraction operator + (int a, const Fraction& b);

```

具体的代码留给读者自己去实现。如果你不实现这个操作符，则需要告诉使用者不要进行 int+Fraction 操作。

1.192 赋值操作符 =

赋值操作符的功能和拷贝构造函数类似，它既强大又复杂，一般不需要去重载它。这一组操作符有以下几种：

=	+=	-=	*=	/=	%=	&=	=	~=
---	----	----	----	----	----	----	---	----

默认情况下，自定义类型之间是支持赋值的，而且功能上够用的（除非是需要“深度拷贝”的情形）。例如，我们有两个变量 `Fraction fa, fb`，那么默认是支持互相赋值的：

```
Fraction fa(2,3);  
  
Fraction fb;  
  
fb = fa; // 赋值操作
```

默认它会把所有的成员（包括父类的成员）依次赋值，这正如我们所期望的那样。我们在以下两种情况下，可能考虑自己重载一个赋值操作符：(1) 需要“深度拷贝”时 (2) 需要用不同的数据类型来赋值时，例如 `fa = 4`。

1.192.1 赋以不同的类型

我们希望以下的操作受到支持，即可以把一个 `int` 型的数直接赋值给一个 `Fraction` 的变量。这在情理上是应该受到支持的。为此，我们应该重载赋值操作符，参数设定为 `int` 类型。

////////// 例 CH27_B1 //////////

```
class Fraction  
{  
public:  
    // Fraction = Fraction  
    Fraction& operator= (const Fraction& other)  
    {  
        this->num = other.num;  
        this->den = other.den;  
        return *this;  
    }  
    // Fraction = int  
    Fraction& operator = (int n)  
    {  
        this->num = n;
```

```

        this->den = 1;
        return *this;
    }

```

... 其他部分代码省略 ...

```
};
```

这个例子中，重载了两个赋值操作符，它们的参数不同。下面是调用方法。

```

int main()
{
    Fraction fa(2, 3);
    Fraction fb(5, 6);
    fa = fb; // Fraction = Fraction
    fb = 4;  // Fraction = int
    return 0;
}

```

1.192.2 深度拷贝

在需要“深度拷贝”时，我们几乎没有选择，只有为它重载一个赋值操作符。由于 Fraction 类是不需要深度拷贝的，所以我们还是采用第 24 章的类 Text 来说明问题。

//////////////////// 例 CH27_B2 //////////////////////

```

class Text
{
public:
    // Text = Text
    Text& operator= (const Text& other)
    {
        if(this == &other) return *this; // 避免赋值给自己
        // 删除现有内容
        if(m_buf)
        {
            delete [] m_buf;
            m_buf = NULL;

```

```

    }

    // 申请一块内存，复制内容
    m_size = strlen(other.m_buf) + 1;
    m_buf = new char[m_size];
    strcpy(m_buf, other.m_buf);
    return *this;
}

// Text = const char*
Text& operator = (const char* str)
{
    if(this->m_buf == str) return *this; // 避免赋值给自己
    // 删除现有内容
    if(m_buf)
    {
        delete [] m_buf;
        m_buf = NULL;
    }
    // 申请一块内存，复制内容
    m_size = strlen(str) + 1;
    m_buf = new char[m_size];
    strcpy(m_buf, str);
    return *this;
}

... 其他部分代码省略 ...

};

int main()
{
    Text t1("helloworld");
    Text t2("na");
    t2 = t1;

```

```

        t1 = "changed";
        return 0;
    }

```

1.192.3 相关问题及注意事项

(1) 等号的传导性

所谓传导性，是指要满足 $a = b = c$ 这种赋值方法。赋值表达式本身是一个左值，返回在定义赋值表达式的时候，必须是返回 `return *this;`

(2) 等号的自反性

要允许 $a=a$ 这种，赋值给自己的操作。当然，这相当于什么都没有做。所以在重载赋值操作符的时候，要提前判断一下目标是否和自己相等。

```
if(this == &other) return *this; // 避免赋值给自己
```

(3) 默认的赋值操作

默认的赋值操作是“按成员赋值”的。如果你真的觉得有必要重载赋值操作符，那么就如何在第 24 章说的那样，“不但要负责本类的所有成员，还要负责父类的所有成员”。假设子类叫 `Child`，父类名字叫 `Base`，那么在为子类 `Child` 重载赋值操作符时，需要显式调用父类的赋值操作符，

```

Child& operator= (const Child& other)
{
    Base::operator= (other); // 显式的调用父类的赋值操作符
    ... 其他部分代码省略 ...
}

```

1.192.4 检查项列表

和拷贝构造函数一样，为读者列出一个检查项的列表，以免遗漏：

- (1) 检查每一个成员是否都已经赋值
- (2) 检查是否调用了父类的赋值操作符
- (3) 检查是否赋值给了自己，当 $a=a$ 这种式子调用时什么都不需要做
- (4) 检查是否有成员需要深度拷贝

1.193 自增操作符++与自减操作符--

这里只列出原型，不给出实现。

前置++与前置--应该这么写:

```
class Object
{
public:
    Object& operator ++ () // 前++
    {
        return *this;
    }
    Object& operator -- () // 前--
    {
        return *this;
    }
    ... 其他部分代码省略 ...
};
```

后置++与后置--略有区别, 使用一个 `int` 型的参数。在下面的代码中, 重载了++和--, 并有一个参数类型为 `int`。这个参数不用别的用途, 就是分了区分后置++和后置--的。

```
class Object
{
public:
    Object& operator ++ ( int )
    {
        return *this;
    }
    Object& operator -- ( int )
    {
        return *this;
    }
    ... 其他部分代码省略 ...
};
```

1.194 关系操作符

关系操作符有以下几种可以重载。本节只演示==的重载方法。

==	!=	>	>=	<	<=
----	----	---	----	---	----

逻辑操作符==用于判断两个量是否相等。编译器不提供默认的==操作符, 需要自己书写。例如, 有两个数 `Fraction`, 我们想在数学意义上比较它们的值是否相等。

```
int main()
{
```



```

    Fraction fa(2, 3);
    Fraction fb(4, 6);
    if(fa == fb) // 编译器报错！不支持==这个操作符
    {
    }
    return 0;
}

```

为了让 `Fraction` 类支持直接用 `==` 判断是否相等，需要为它重载 `==` 操作符的支持，自己写代码来决定两个对象值是否相等。这个判断的实现是比较容易的，下面直接给出示例代码。

```

////////// 例 CH27_C1 //////////
class Fraction
{
public:
    // Fraction == Fraction
    bool operator == (const Fraction& other)
    {
        if(num * other.den == den * other.num)
        {
            return true;
        }
        return false;
    }
    ... 其他部分代码省略 ...
};

```

事实上，我们可以重载多个 `==` 操作符，例如，添加下面的代码，就可以支持 `Fraction` 和 `int` 型直接判断大小关系。

```

////////// 例 CH27_C2 //////////
class Fraction
{
public:

```

```

// Fraction == int
bool operator == (int n)
{
    if(num == den * n)
    {
        return true;
    }
    return false;
}
... 其他部分代码省略 ...
};

```

显然，怎么样才算相等、怎么样算是不相等，这个逻辑判断是由程序员自己决定的。比如，我们在比较两个 `Text` 对象时，可以决定怎么样才算是相等。比如，我们可以忽略大小写来判断，这样 `Text("HELLO")` 和 `Text("hello")` 算是相等的。我们也可以规定，在校对 `Text` 时要严格区分大小写。

下面的例子中给出了区分大小时的比较方式。

```

////////// 例 CH27_C3 //////////
class Text
{
public:
    bool operator == (const char* str)
    {
        if(strcmp(m_buf, str) == 0)
        {
            return true;
        }
        return false;
    }
... 其他部分代码省略 ...
};

```

1.195 逻辑操作符

以下几个逻辑操作符理论上可以重载，但在实际应用中往往没有重载的必要。本书不予介绍。

&&	 	!
-------------------	-----------	----------

1.196 类型转换操作符()

类型之间可以强制转换，例如，double 可以转成 int

```
double b = 123.12;
int a = (int) a;
```

现在，对于自定义的 Fraction 类型，我们希望能够直接转换成一个 double 型，用一个小数值来表示其大小。

```
int main()
{
    Fraction fa(4, 5);
    double value = (double)fa;
    printf("value of the fraction: %.3f \n", value);
    return 0;
}
```

默认情况下，编译器并不支持从 Fraction 到 double 的类型转换。为此，我们需要重载类型转换操作符。

```
////////// 例 CH27_D1 //////////

class Fraction
{
public:
    // 类型转换操作符
    operator double()
    {
        return (double)num/den;
    }
}
```

... 其他部分代码省略 ...

```
};
```

在所有的重载操作符里，类型转换操作符的形式最为“怪异”。这是一个语法的规定，不需要过多的理解，只需要强行记住其形式。

```
operator double()
```

其中，double 就是要转换的目标类型。

1.197 元素操作符[]

操作符[] 被用于访问一组元素中的一个元素。默认地，数组是支持下标访问的。例如，

```
int a[5] = {1, 2, 3, 4, 5};
```

```
a[0] = 111; // 写操作
```

```
int b = a[0]; // 读操作
```

其中，[]中的值可以有几种叫法：下标，索引，key，唯一标志符。其作用是用来唯一的标识一个元素。当一个对象设计用于包含多个子项时，就可以重载操作符[]来访问内部的子项。比如，一个字符串 **Text**，它包含了多个元素（每个元素是一个字符），那么我们就希望用[]可以直接访问到它的每一个字符，就像下面这样：

```
Text txt("hello world");
```

```
char ch = txt[0]; // 'h'
```

我们为 **Text** 添加[]操作符的支持，示例代码如下，

```
////////// 例 CH27_E1 //////////
```

```
class Text
```

```
{
```

```
public:
```

```
    char& operator[] (int index)
```

```
    {
```

```
        return m_buf[index];
```

```
    }
```

```
... 其他部分代码省略 ...
```

```
};
```

重载操作符的一般形式

Element& operator [] (Type index)

```
{  
}
```

其中，

(1)名字: operator[], 是固定不变的

(2)返回值: 一般应该返回一个子元素的引用, 表示返回值是一个“左值”, 可读可写

(3)参数: 类型可以自己选择, 用于指定元素的下标。不一定要使用 int, 使用其他类型也可以。

在这里, 当我们要重载[]操作符时, 读者需要打破两个思维定势:

a. 下标不一定从 0 开始, 这个完全由程序员自己来决定。

b. 下标也不一定是整数, 我们也可以使用字符串来作为下标, 这个下标只是一个“唯一标识符”的作用。

1.198 输入输出操作符 >> 与 <<

C++中引入操作符 >> 表示输入, <<表示输出。在本节中, 只给出输出操作符<<的示例。

首先, 需要大家回忆一下在第 6 章介绍的位操作, 那里介绍了>>和<<一种意义: 移位。

是的, 当操作数是整数类型时, 此操作符的意义是右移位、左移位。例如,

```
unsigned int a = 0xAA << 4; // 左移 4 位
```

然后, 在 C++中, 当操作数的整数是自定义 class 类型时, 此操作符还表示输入和输出的功能。下面给出一个示例: 我们定义一个类 Logger, 用于输出打印日志。我们规定其用法为:

```
Logger lg; // 定义一个对象
```

```
lg << 1; // 向控制台输出一个整数 1
```

```
lg << 2.0; // 向控制台输出一个小数 2.0
```

```
lg << "hello"; // 向控制台输出一个字符串
```

```
lg << 1 << ", " << 2.0 << "hello"; // 可以串起来写
```

一般形式:

Logger& operator << (Type value)

```
{
```

```
        return *this;
```

```
    }
```

(1)名称: operator <<

(2)返回值: 左值, 一般返回对象自己

(3)参数: 待输出的数据

下面给出 Logger 类的示例代码,

////////// 例: CH27_F1 //////////

```
class Logger
```

```
{
```

```
public:
```

```
    Logger& operator << (int value)
```

```
    {
```

```
        printf("%d", value);
```

```
        return *this;
```

```
    }
```

```
    Logger& operator << (double value)
```

```
    {
```

```
        printf("%f", value);
```

```
        return *this;
```

```
    }
```

```
    Logger& operator << (const char* value)
```

```
    {
```

```
        printf("%s", value);
```

```
        return *this;
```

```
    }
```

... 其他部分代码省略 ...

```
};
```

1.199 操作符 new 与 delete

这几个操作符理论允许被重载，但是无论是业余人员还是专业人员，应该都没有必要重载它们。本书不作详细介绍。

delete	delete[]	new	new[]
---------------	-----------------	------------	--------------

1.200 小结

重载操作符的语法使得用户代码比较简洁一些，用户在使用你的类时，会认为你的类写得比较完善。但是如果不重载操作符的话也完全没有问题，使用同名函数也可以实现相同的功能。例如，赋值函数可以命名为 **Copy/Assign**，取元素可以叫 **At**，判断相等可以用 **Equal**，等等，用相应的函数来实现这些功能是完全可以的。

第 28 章 内部类和名字空间

把一个类定义在另一个类的内部，就称为内部类。内部类可以有效避免名字冲突的问题。名字空间是解决名字冲突的终极解决方案。相对来说，内部类的写法较为常见，但名字空间则很少用到。

1.201 内部类

1.201.1 内部类的语法

把一个类的定义写在另一个类的内部，就称其为内部类。例如，下面的例子中，将类 Inner 定义在 AAA 类的内部，则将 Inner 称为类 AAA 的内部类。

////////// 例 CH28_A1 //////////

```
class AAA
{
public:
    // 定义一个内部类
    class Inner
    {
    public:
        char name[64];
    };
};
```

```
public:
    AAA()
    {
    }
```

};

此时，内部类的类名全称为 **AAA::Inner**，在使用的时候需要用类名的全称。例如，

```
int main()
{
    AAA::Inner a; // 使用该内部类时，类名使用全称 AAA::Inner
    strcpy(a.name, "AnXin"); // 其他使用方法和普通的类没有区别
    printf("Name: %s \n", a.name);
    return 0;
}
```

1.201.2 内部类和普通类的区别

可以说，内部类在使用方法和普通类几乎没有区别。

(1) 内部类的类名需要用全称，例如，AAA::Inner

(2) 内部类的如果被 `private/protected` 修饰，则被隐藏在类内部，外部无法使用这个类。

例如，

```
class AAA
{
private: // 将 Inner 类用 private 修饰符限制，则外部无法使用 AAA::Inner
    // 定义一个内部类
    class Inner
    {
    public:
        char name[64];
    };
};
```

(3) 外部类和内部类互相没有特权

外部类 `AAA` 无法自由的访问内部类的成员，内部类 `Inner` 也无法自由访问外部类的成员。相当于把 `Inner` 写在外面。

注：它们既不是父类和子类的关系（第 23 章继承），也不是朋友关系（第 26 章朋友成员）

1.201.3 内部类的用途

内部类的用途主要是为了避免类名的冲突。当项目中的类越来越多时，避免类名冲突成了一个重要的问题。使用内部类的语法，就可以在类 `AAA` 里定义一个类叫 `Inner`，在另一个类 `BBB` 里也定义一个类叫 `Inner`。那么 `AAA::Inner` 和 `BBB::Inner` 是不冲突的，是两个完全不同的类。

那么在什么情况下我们应该考虑使用内部类呢？

当我们发现一个类型仅在局部使用的时候，比如，只在一个类中需要用到，那么就可以定义一个内部类。这样，就可以定义一个比较短小的名字，而不用担心名字冲突的问题。

1.201.4 内部类的分离式写法

内部类也可以分别写到 `.h` 和 `.cpp` 里，在头文件里写类的声明，而在 `cpp` 文件里写类的定义。在 `cpp` 中，需要用 `AAA::Inner` 全称。

例如，头文件 `AAA.h` 内容如下

////////// 例 CH28_A2 : AAA.h //////////

```
class AAA
{
public:
    class Inner
    {
```

```

        public:
            Inner();
            void Set(int v);
        private:
            int value;
    };

```

```

public:
    AAA();
    Inner Get();

```

```

public:
    Inner m_in;
};

```

源文件 AAA.cpp 内容如下

```

//////////////////// 例 CH28_A2 : AAA.cpp //////////////////////
#include "AAA.h"
// 内部类的定义
AAA::Inner::Inner() : value(11)
{
}
void AAA::Inner::Set(int v)
{
    this->value = v;
}
// 外部类的定义
AAA::AAA()
{
}
AAA::Inner AAA::Get() // 注意：使用 AAA::Inner 全称
{
    return m_in;
}

```

1.201.5 内部 enum 类型

和内部 class 的语法类似，也可以在类内定义 enum 类型。enum 的定义方法请回顾第 3 章。在这里只介绍如何在类里定义 enum，以及使用方法。

```
////////// 例 CH28_A3 //////////  
  
#include <stdio.h>  
  
class Painter  
{  
public:  
    // 用于表示颜色的枚举类型  
  
    enum Color  
    {  
        RED = 0xFF0000,  
        GREEN = 0x00FF00,  
        BLUE = 0x0000FF  
    };  
    // 省略类型的枚举  
  
    enum { NORMAL=0, BOLD};  
};  
  
int main()  
{  
    Painter::Color clr = Painter::RED;  
  
    int style = Painter::BOLD;  
  
    return 0;  
}
```

可见，enum 本身的定义方法没有变化，只是位置写在了类的里面。那么在使用时，必须冠以类名范围符作为前缀。例如，类型写作 `Painter::Color`，字面常量写作 `Painter::RED`。

在类内部定义 enum 是非常常见的技术。类似地，还可以在类里定义 struct 类型，这里就不再一一举例了。

1.202 名字空间

名字空间（namespace），是解决名字冲突问题的终极解决方案。顾名思义，我们可以把很多名字（类名、函数名、全局变量名）定义在一个空间里，然后用一个范围符作为前缀。

1.202.1 名字空间的定义

namespace 的语法形式为，

```
namespace ID
{
}
```

其中，namespace 的名字 ID 是要全局唯一的。（注：namespace 末尾无需分号）

示例如下，

```
//////////////////////////////// 例 CH28_B1 //////////////////////////////////
```

```
// 定义一个名字空间: tinyxml
```

```
namespace tinyxml
{
    // xml 文档
    class Document
    {
    };
    // xml 元素
    class Element
    {
    };
}
```

由于类 Document 和 Element 定义在名字空间里，所以在名字空间之外使用时必须加上范围前缀，即 tinyxml::Document, tinyxml::Element。例如，在 main 函数里使用它们，方法如下，

```
int main()
{
    tinyxml::Document doc; // 使用类名全称，必须加上名字空间的前缀
    tinyxml::Element elem; // 使用类名全称，必须加上名字空间的前缀
}
```

```

    return 0;
}

```

1.202.2 分离式写法

当一个类写在名字空间里时，也可以使用分离式写法，即成员函数分开写到 `cpp` 文件里。方法示例如下：

(1) 添加 `tinyxml.h`

```

////////// 例 CH28_B2: tinyxml.h //////////
namespace tinyxml
{
    // xml 元素
    class Element
    {
    public:
        Element();

    public:
        char name[32];
        char value[128];
    };

    // xml 文档
    class Document
    {
    public:
        Document();

        int Open();

        int AddElement(const Element& elem);
    };
} // namespace 结束

```

(2) 添加 `tinyxml.cpp`

////////// 例 CH28_B2: tinyxml.cpp//////////

```
#include <stdio.h>
#include "tinyxml.h"
namespace tinyxml
{
    Element::Element()
    {
    }

    Document::Document()
    {
    }

    int Document::Open()
    {
        return 0;
    }

    int Document::AddElement(const Element& elem)
    {
        return 0;
    }
}
```

注意：在 namespace 的内部使用 Document, Element 这个类型时无须加上 tinyxml::前缀，只有在名字空间之外使用时才需要加前缀。

而在缩进书写方面，可以把类的定义顶格书写，也可以缩进一格书写。

1.202.3 using 语句：解除前缀

在前面的定义中，在 main 函数中使用名字空间内的名字需要加上前缀。有时候我们会觉得有点麻烦，毕竟每个类多写了一个前缀，使得代码的可读性略有下降。如果我们确定在调用的地方（如 main.cpp）里不会有名字冲突问题，名们可以使用 using 语句来解除前缀。

例如，下面的代码中添加了一行 using namespace 语句，使得 tinyxml 内的所有名字都可以直接使用而不需要前缀。

```
#include <stdio.h>
#include "tinyxml.h"
```

using namespace tinyxml; // 使用整体名字空间

```
int main()
{
    Document doc; // 前缀 tinyxml::可以省略
    Element elem; // 前缀 tinyxml::可以省略
    doc.Open();
    doc.AddElement(elem);
    return 0;
}
```

还有另一种选择，只使用名字空间中的部分名字，其他名字依然需要加前缀。例如，下面的语句中，只 using 了 tinyxml::Document，因此在使用 Document 的时候可以省略前缀，但使用 Element 的时候还是必须要加前缀。

```
#include <stdio.h>
#include "tinyxml.h"

using tinyxml::Document; // 只使用空间内的部分名字

int main()
{
    Document doc; // 前缀 tinyxml::可以省略
    tinyxml::Element elem; // 其他类名依然要加上前缀
    doc.Open();
    doc.AddElement(elem);
    return 0;
}
```

注：使用 using 语句后，此时 tinyxml::可以省略，但是显式加上的话也没有错误（只是略显累赘）。

1.202.4 名字空间里的其他类型

以前我们学过的所有类型都可以被包含在名字空间里，例如枚举 Enum，全局变量，函数，结构体等。用法和上面是类似的，不再举例。

1.202.5 名字空间的应用场景

可以说，名字空间这个语法在实际中并不常用。它一般仅在以下场景下使用：当你需要完成一个大型的模块供别人使用，里面有很多个类型组成；那么，为了避免名字冲突，你可以把你的所有类型都封在一个名字空间里。

显然，只有当我们有机会写这种大型的模块的时候，再考虑使用它。

第 29 章 模板

1.203 函数模板

C++提出模板的语法是为了表达一个通用的算法或逻辑。考虑以下几个函数，都是用来求一个数组里最大的元素。

```
int findmax (int arr[], int len)
{
    int val = arr[0];
    for(int i=1; i<len ; i++)
    {
        if(arr[i] > val) val = arr[i];
    }
    return val;
}

double findmax (double arr[], int len)
{
    double val = arr[0];
    for(int i=1; i<len ; i++)
    {
        if(arr[i] > val) val = arr[i];
    }
    return val;
}

Object findmax (Object arr[], int len)
{
    Object val = arr[0];
    for(int i=1; i<len ; i++)
    {
        if(arr[i] > val) val = arr[i];
    }
    return val;
}
```

```
}
```

以前几个函数要表达的算法是相同的，唯一不同的地方参与运算的数据的类型不同。有没有什么简化的方法，来描述这种与类型无关的算法呢？那就是函数模板。

```
////////// 例 CH29_A1 //////////
```

```
template <typename T>
T findmax (T arr[], int len)
{
    T val = arr[0];
    for(int i=1; i<len ; i++)
    {
        if(arr[i] > val) val = arr[i];
    }
    return val;
}
```

在函数前面加上 `template < typename T>`，用于声明此函数是一个模板。其中的`< typename T>`表示“在下面的代码中，T是一个类型名”。其中，关键字 `typename` 也可以写为 `class`，意义上相同。（但是用 `class` 可能会给大家带来理解上的一点不便，故本书统一用 `typename`）

它要表达的是：无论对于任意类型 T，其算法都是一样的。具体来说，它描述了一个“在数组中求最大值”这个算法，其中数组元素的类型 T 并不重要。无论是 T 是 `int` , `double`, 或是其他类型，这个求最大值的算法是通用的。

以上是函数模板的定义，那么怎么调用它呢？它的使用方法是这样的：

```
int main()
{
    int arr[ 4] = { 1, 42, 87, 100 };
    int result = findmax<int> (arr, 4);
    return 0;
}
```

其中，`max<int> ()`表示对模板进行实例化，用 `int` 类型替换模板中的参数 T。实例化之后，得到了一个具体的函数，就可以直接调用了。

也就是说，`max` 作为一个函数模板，你是不能直接调用它的。比如直接调用 `max(arr, 4)` 这样是不行的。模板在使用时，必须要告诉编译器它的模板参数。

在指定了类型参数之后，函数模板就被实例化，得到一个具体的函数。例如，上述的 `max<int>()`，就是用 `int` 去替换函数模板的参数 `T`。

```
int findmax (int arr[], int len)
{
    int val = arr[0];
    for(int i=1; i<len ; i++)
    {
        if(arr[i] > val) val = arr[i];
    }
    return val;
}
```

当然，这个替换动作是编译器自己完成的，作为程序员，只需要调用的时候使用 `max<int>` 这种形式，就完成实例化了。

又如，数组的排序算法可以写成一个模板。在实现这个算法的时候，并不关心元素的类型是 `int` 还是 `double`，因为算法本身都是一样的。所以可以用模板的语法来实现此类的算法，

```
template <typename T>
void sort ( T* arr, int len)
{
}
}
```

在调用 `sort` 的时候，必须同时指定模板参数，例如，对于一个 `float` 型数组排序，示例代码如下：

```
float arr[4] = { 0.1f, 1.1f, 2.3f, 5.4f};
sort <float> (arr, 4);
```

可以看出，函数模板在使用的时候，是必面是要在尖括号内指定模板参数的。

1.204 类模板

可以将模板的语法作用于类。有的类也是只描述了一些通用的算法和逻辑，适用于多个数据类型。比如，对于链表来说，无论元素的类型是什么，其插入、删除、遍历节点的算法都是通用的。所以可以用一个模板来描述。形如，

```
template < typename T>
class List
{
}
```

```

public:
    void insert(const T& node)
    { ...
    }

    T& front()
    { ...
    }

    int length()
    { ...
    }

};

```

对于上述的模板，其调用方法示例如下：

```

List<int>  mylist;

int len = mylist.length();

int& first = mylist.front();

```

同样的, List<int> 称为是对类模板 List 的实例化。

类模板和普通的类的写法有一点不同。通常情况下，类的声明会写在*.h 文件里，而成员函数定义会集中地写在*.cpp 文件里。类模板却不是这样，类模板的一般写法是把声明和实现都写在*.h 文件里。这并不是意味着类模板不能够以.h/.cpp 的方式分离来写，只是本书中不对其进行介绍。为什么不介绍呢？因为 C++ 中有些语法是实际上不会用到、或不提倡使用的。

对于类模板来说，本章的任务是懂得它的概念，但是一般并不要求自己会写模板。C++ 提供了一个标准模板库(STL, Standard Template Library)， 如何会用这个库，才是学习的重点。这是下一章要介绍的东西。

1.205 模板参数

无论是函数模板还是类模板，其语法大致相似，都是在函数/类之前添加一个 template<...> 来声明其是一个模板。尖括号里称为模板参数列表，

```

template <typename T1, typename T2, ..., typename Tn>

```

通常情况下，在尖括号<>内定义的模板参数都是一个类型。然而，它也可以是值，这在语法上是支持的，虽然这种情况并不常见。例如，在下面的函数模板中就定义一个 int 型的 N 作为参数，

```
template <int N, typename T>
T* create ()
{
    T* str = new T [N];
    return str;
}
```

这个模板参数列表<int N, typename T>，表示需要 2 个参数：一个 int 型的值，和一个类型名。其调用方法举例如下：

```
char* p = create <100, char> ();
```

使用值作为模板参数并没有什么意义。为什么呢？举个例子，以上的类模板其实等价于这么定义：

```
template< typename T>
T* create (int n)
{
    T* str = new T[n];
}
```

1.206 实例 1

栈是数据结构中的一个概念，表示先入后出的线性结构。一般支持以下操作：

- * Push, 将一个元素推进去
- * Pop, 将一个元素弹出来
- * Top, 得到最后一个放进去的元素，但并不弹出
- * Size, 已经推进去的元素的总数

可以发现，栈的这些逻辑不区分元素的类型的：无论要处理的元素类型是哪种，其算法和逻辑都是一样的。这恰好是符合类模板的特征的。

首先，写下类模板的框架，

```
template <typename T>
class Stack
```

```
{  
};
```

接着，写下要实现的函数接口，

```
template <typename T>  
class Stack  
{  
public:  
    bool push (const T& value); // 推入一个元素  
    T pop ();      // 弹出一个元素  
    const T& top(); // 得到栈顶元素  
    int size();    // 已经推入的元素的总个数  
};
```

接着，实现各个接口。一般来说，栈的总大小是有限制的，可以让用户在构造函数里来传入一个最大值参数。栈是用线性结构表示，所以动态分配一个 **buffer** 用于存储元素、并用一个整数来指示栈顶的位置。

完整代码如下：

////////// 例 CH29_B1: Stack.h //////////

```
template <typename T>  
class Stack  
{  
public:  
    Stack(int maxsize)  
    {  
        m_maxsize = maxsize; // 记录此栈的最大长度  
        m_buffer = new T[maxsize]; // 申请一个 buffer 作为存储空间  
        m_size = 0; // 初始长度为 0  
    }  
    ~Stack()
```

```

{
    delete [] m_buffer; // 释放资源
}

// 推入一个元素
bool push (const T& value)
{
    if(m_size >= m_maxsize) return false; // 检查空间是否已满

    m_buffer[m_size] = value; // 存入该值
    m_size ++; // 长度++
    return true;
}

// 弹出一个元素
T pop ()
{
    T last = m_buffer[m_size - 1]; // 弹出栈顶元素
    m_size --; // 长度--
    return last;
}

// 得到栈顶元素
const T& top()
{
    return m_buffer[m_size - 1]; // 取得栈顶元素
}

// 已经推入的元素的总个数
int size()
{

```

```

        return m_size;
    }

private:
    T* m_buffer;
    int m_maxsize;
    int m_size ;
};

////////// 例 CH29_B1: main.cpp //////////

#include <stdio.h>
#include "Stack.h"

void main()
{
    Stack<int> my_stack(10);
    my_stack.push(1);
    my_stack.push(2);
    my_stack.push(3);
    my_stack.push(4);

    while(my_stack.size() > 0)
    {
        int n = my_stack.pop();
        printf("pop : %d \n", n);
    }
}

```


第 30 章 STL 标准模板库

标准模板库 (STL, Standard Template Library) 是 C++ 标准库的最重要的一部分，在 C++ 编程的大部分应用场合都会使用。在前一章介绍了模板的概念和使用方法，可以告诉读者的是，程序员很少需要自己来定义一个模板。大多数时候要做的，只是在工程应用中使用现成的模板库，而 STL 就是你即将频繁使用的一个库。

基本上所有的 C++ 编译器都对 STL 进行了支持，无论你是使用 VC++，还是 GNU 的 g++，都是保证了对 STL 的正确支持的。这意味着，同一份使用 STL 的代码，即使被 VC++ 编译，还可以被 g++ 编译，且运行时的行为表现是一致的。

STL 里封装了几乎所有常见的线性数据结构，罗列如下：

模板	意义
vector	一维向量，相当于数组
list	链表
map	映射。提供(Key, Value)式操作，相当于哈希表
string	char 字符串。
queue	队列。先入先出的线性表。
stack	栈。先入后出的线性表。
set	集合。
deque	双向链表。

其中，vector, list, map, string 是最常用到的模板，本章会重点介绍它们。

虽然有的读者有学习完前一章后，已经懂得自己定义一些模板，如链表类模板。但是，如果 STL 中已经存在你需要的模板，一般请直接使用它们、不要自己进行“重复制作轮子”的行为。

1.207 一般使用方法

先介绍一下 STL 中的各个类模板的一般使用方法。以 vector 为例，分 3 步说明，

① 引用相关的头文件：

```
#include <vector>
```

我们知道，C++ 的头文件是可以省略.h 后缀的，习惯上在引用 STL 的头文件时，是不加后.h 后缀。

② 引用命名空间

```
using namespace std;
```

③ 调用

```
vector<int> arr; // 传入模板参数为 int，表示定义一个元素为 int 型的 vector
arr.push_back(10);
arr.push_back(11);
for(int i=0; i<arr.size(); i++)
{
    int val = arr.at(i);
    printf("%d \n", val);
}
```

一个完整的例子是这样子的，

////////// 例 CH30_A1 //////////

```
#include <stdio.h>

#include <vector> // ① 引用头文件

using namespace std; // ② 引用命名空间
```

```
int main()
{
    vector<int> arr; // ③ 定义实例

    arr.push_back(10);
    arr.push_back(11);
    arr.push_back(12);
    for(int i=0; i<arr.size(); i++)
    {
        int val = arr.at(i);
        printf("%d \n", val);
    }
    return 0;
}
```

1.208 向量 vector

vector 用于实现数组的功能，相当于在数组的基础上封装了一些常用的功能。由于本书是一本教程(guide)而不是参考书(reference)，所以表 30-1 只列出其常用的函数接口并给出使用说明。如果想知道它的全部函数接口，应该使用参考文档（如 MSDN 等文档）。

表 30-1 vector 的主要函数

函数名称	功能
push_back	在尾部添加一个元素
pop_back	的尾部删除一个元素
clear	清空所有元素
at	按索引访问某个位置的元素
front	返回头元素
back	返回尾元素
size	返回元素的个数
capacity	返回当前容量
resize	改变容量的大小
insert	在中间插入元素
erase	删除中间的元素

vector 和普通的数组一样，具有一个容量的概念。可以推断，vector 内部是有一个缓冲区来存储元素的。类似这样的代码，

```
template <typename T>
class vector
{
private:
    T* m_buffer;
    int m_capacity;
};
```

可见，vector 内容上还是数组，只是外在形式上比单纯的数组更易于使用了。它对外封装了前面所列的接口函数，使得读者非常方便地操纵数组。

在创建 vector 对象时，可以显式地指定其初始大小，例如，以下的代码将初始容量设置为 128，这意味着其内部创建一个了可以容纳 128 个元素的缓冲区，

```
vector<int> arr(128);

int capacity = arr.capacity(); // 容量: 128

int size = arr.size(); // 大小: 128, 已经包含 128 个元素

在这种情况下，size 和 capacity 相等。
```

1.208.1 at / front / back

在创建 vector 之后，但可以对 vector 的元素进行访问。at / front / back 这三个函数用于访问数组中的元素，其中，at 用于按索引访问任意位置的元素，front 用于访问头位置的元素，back 用于访问尾位置的元素。

例如，以 at 进行任意位置的访问，(假设 arr 的 capacity>0):

```
////////// CH30_B1 //////////

vector<int> arr(128);

for ( int i=0; i< arr.capacity() ; i++)

{

    int& p = arr.at (i );

    p = 0; // 全部初始化为 0

}
```

看起来很简单，直接用 at 函数可以得到元素的引用（左值），然后直接操作这个引用即可。实际上 vector 还重载了操作符[]，因此以下代码是等价的，

```
int & p = arr.at (0);

int & p = arr[0];
```

使用操作符[] 会显得更简洁一些。但是值得强调的是，如同数组一样，要注意越界问题。无论是 at 还是[]，传入的索引值都要在正确的范围之内 ($0 \leq \text{index} \leq \text{capacity} - 1$)。

另外两个函数 front 和 back 则用于得到其首、尾元素，

```
int& head = arr.front(); // 相当于 head = arr [ 0 ];

int& tail = arr.back(); // 相当于 tail = arr [ arr.capacity() - 1];
```

需要注意的是，对于 const 实例只能其 at / front /back 返回的引用均为 const reference（右值）。例如，

```
// arr 是 const 对象，不能修改其内容

void test ( const vector<int> arr )
```

```

{
    const int& p = arr[0]; // const T& 类型, 只读
    printf ("%d", p);
}

```

`vector` 本质上就是对一个数组的封装，在必要的时候可以取得其内部的缓冲区指针来使用。例如，

```

vector<int> arr ( 128 );

int* p = & arr[0]; // 取得内部缓冲区地址

int size = arr.size ( ) ; // 取得实际内容长度

sendto ( p, len ); //

```

注：这里的 `sendto` 函数只是举例示意：假定存在一个 `sendto` 函数，用于发送一段数据，因而需要传入首地址和长度作为参数。

1.208.2 `push_back` / `pop_back` / `resize` / `clear`

`vector` 的一个显著的特点是，其容量是可以动态调整的。使用 `push_back` 可以在尾部追加一个元素，使用 `pop_back` 删除尾部元素。

例如，

```

vector<int> arr; // capacity:0, size: 0

arr.push_back ( 1 ); // capacity: 4, size : 1

```

当创建对象时如果没有指定其 `capacity`，则默认 `capacity` 为 0，此时 `capacity` 和 `size` 都是 0。当使用 `push_back` 在末尾添加一个元素时：

(1) 如果还有剩余空间可用(`size < capacity`)，则 `size` 加 1，`capacity` 不变

(2) 如果空间已满 (`size = capacity`)，则重新申请一个较大的缓冲区，再把原有的数据拷贝到新缓冲区，再把新元素追加在后面。此时 `size` 加 1，`capacity` 为新的缓冲区容量(一般新容量是原有的 `capacity + 4`，具体取决定 STL 库的实现)。

可见，`push_back` 并不是节省 `cpu` 的操作，它是反复地申请新缓冲区和拷贝大段数据。所以在使用 `vector` 时候，应该先预估一下程序运行时可能需要的最大容量，并在初始化的时候指定其 `capacity` 参数。

如果在迫不得已的情况下一定要调用容量的话，可以使用 `resize` 函数，尽量一步调整到位，不要多次反复地 `resize`。例如，

```

if( arr.size > 1024 )
{

```

```

        arr.resize (2048);
    }

```

pop_back 比 **push_back** 要简单一些，是直接删除末尾的元素。在 **pop_back** 之后，**size** 减 1，**capacity** 不变（内部缓冲区不重新申请）。

clear 函数用于清空元素，**size** 归 0，而 **capacity** 不变（内部缓冲区不重新申请）。

在例 CH30_B2 中，使用 **vector** 来存放数组。定义数组时，指定参数为 128，表示初始容量设置为 128（避免在内部反复分配内存）。代码如下所示：

```

////////// CH30_B2 //////////
#include <stdio.h>
#include <vector>    // ① 引用头文件
using namespace std; // ② 引用命名空间
int main()
{
    vector<int> arr (128); // 指定初始容量，要足够大
    arr.clear (); // size 清 0，capacity 不变（128）
    arr.push_back ( 1 );
    arr.push_back ( 2 );
    arr.push_back ( 3 );
    // ...
    int size = arr.size (); // 计算一下现在一共存入了多少个元素
    return 0;
}

```

1.208.3 iterator / const_iterator

迭代器(iterator)是 **vector** 的内部类，用于对 **vector** 内部的元素进行遍历。

以下的代码定义一个 **iterator** 对象，

```
vector<int> :: iterator iter;
```

其中，**vector<int> :: iterator** 是类型名，表示的是 **vector<int>** 内定义的内部类 **iterator**。

vector 的 **begin** 函数用于取得迭代器的初始位置，**end** 用于取得末尾位置。例 CH30_B3 中展示了如何用迭代器对一个 **vector** 进行遍历操作。

////////// 例 CH30_B3 //////////

```
vector<int> :: iterator iter;
for (vector<int>::iterator iter = arr.begin();
    iter != arr.end (); iter ++)
{
    int& p = *iter; // iterator 类支持*号操作
    printf("%d \n", p);
}
```

由于 iterator 类重载了操作符*和操作符->, 所以它是支持*号操作的, 就像操作普通的指针一样。但为了从可读性考虑, 应该显式的用一个引用来指向这个元素。

```
int& p = *iter;
```

当 vector 对象是 const 对象时, 须使用 const_iterator 进行替代操作, 示例代码如 CH30_B4 所示:

需要对 const vector 对象进行迭代时, 使用内部类 const_iterator, 一般的遍历方法如下:

////////// 例 CH30_B4 //////////

```
void test (const vector<int>& arr)
{
    // 使用其内部类 const_iterator
    for (vector<int>::const_iterator iter = arr.begin();
        iter != arr.end (); iter ++)
    {
        const int& p = *iter; // 这里是 const reference
        printf("%d \n", p);
    }
}
```

实际上, 后面介绍的 list, map, string, queue, deque 等容器模板都是支持迭代器的。也就是说, 它们的类模板内部也都定义了 iterator/ const_iterator 的内部类, 而且迭代方法也都是样的, 后在会依次给出样例。

1.208.4 insert / erase

insert 函数用于向 vector 中间的指定位置插入一个元素, 而 erase 则是从中间删除一个元素。

正如此数组的插入、删除操作一样，向 `vector` 中插入删除操作困难、且效率低下。这是因为，当在中间插入一个元素时，需要把这个位置后面的所有元素都后移一个位置。在删除时，则要把后面的元素都前移一个位置。如此大规模的数据移动操作，效率自然是低的。所以在程序中应尽量毕竟对 `vector` 变量进行 `insert/erase`。

但不管怎样，本书还是介绍一下其使用方法。（此方法适用于后面的 `list` 模板）

为了加深大家的理解，这里使用一个 `struct` 作为了元素类型，

////////// 例 CH30_B5 //////////

```
struct Object
```

```
{  
    int id;  
    char name[32];  
};
```

定义一个全局变量，

```
vector<Object> objs(1000);
```

向里插入一个对象，按 `id` 增序排列，

```
void AddObject(const Object& s)  
{  
    for(vector<Object>::iterator iter = objs.begin();  
        iter != objs.end(); iter ++)  
    {  
        Object& p = *iter;  
        if(s.id < p.id) // 判断比较，插到指定位置  
        {  
            // 在该位置插入新元素 s  
            objs.insert( iter, 1, s);  
            break;  
        }  
    }  
}
```


insert 的第一个参数是 iterator 类型，表示要插入的位置。第二个参数表示要插入的元素的个数。第三个参数表示要插入的元素的值。合起来就是，在指定位置插入 N 个元素值。

erase 函数的使用步骤也是一样，首先要找到目标元素的位置，然后再调用 erase 删除它。

```
void DelObject(int id)
{
    for(vector<Object>::iterator iter = objs.begin();
        iter != objs.end(); iter++)
    {
        Object& p = *iter;
        if(p.id == id)
        {
            objs.erase(iter);
            break;
        }
    }
}
```

这里再次强调一下，vector 操作不适合进行 insert/erase 操作。如果你需要经常性的 insert/erase，那么请使用后面的 list (链表)。

1.209 list

list 对应于数据结构中的链表（单向链表），可以和第 13 章中的链表对照一下。

对应的头文件为<list>。list 与 vector 操作更为灵活，允许从两端(front, back)进行操作。同时也支持中间元素的 insert 和 erase。它不支持随机访问，只支持顺序访问。常用的函数接口如表 30-2 所示：（list 是对链表的封装，vector 是对数组的封装）

表 30-2 list 的主要函数

函数名称	功能
push_back	在尾部添加一个元素
pop_back	的尾部删除一个元素
push_front	在尾部添加一个元素
pop_front	的尾部删除一个元素
clear	清空所有元素

size	返回元素的个数
front	返回头元素
back	返回尾元素
insert	在中间插入元素
erase	删除中间的元素

1.209.1 push_back / pop_back / push_front / pop_front

支持尾端的操作，`push_back` 向末尾添加一个元素，`pop_back` 删除末尾的节点。也支持头端的操作，`push_front` 在头部插入一个元素，`pop_front` 则删除头部的节点。这四个函数都是速度比较快的操作：不需要遍历，不需要移动数据，一步完成。

////////// 例 CH30_C1 //////////

```
#include <stdio.h>

#include <list>    // ① 引用头文件

using namespace std; // ② 引用命名空间

int main()
{
    list<int> mylist;
    mylist.push_back(1);
    mylist.pop_back();
    mylist.push_front(2);
    mylist.pop_front();
    return 0;
}
```

虽然 STL 中有 `queue` 表示队列，`stack` 表示栈，但也可以用 `list` 来实现队列和栈。当你坚持只对 `list` 进行 `push_back` 和 `pop_front` 时，它就是一个队列。当你坚持 `push_back` 和 `pop_back` 时，它就是一个栈。使用上是非常灵活的，怎么用完全取决于你的应用要求。

1.209.2 size / clear

`size` 函数用于取得 `list` 中的节点的个数。而 `clear` 用于清空 `list`。

1.209.3 front / back

`front` 得到头部元素的引用，而 `back` 取得尾部元素的引用。对于 `const list` 来说，返回的引用也是 `const reference`。

```
list < Object > students;

Object& first = students.front();

Object& last = students.back();
```

1.209.4 iterator

`list` 的遍例方法与 `vector` 完全相同，而且对 `list` 来说只能使用 `iterator` 来进行迭代遍例，

```
list < Object > students;

for( list<Object>::iterator iter = objs.begin();
    iter != objs.end(); iter ++)
{
    Object& p = *iter ;
    //... 对 p 进行操作 ...
}
```

同样地，对于 `const list` 对象只能使用 `const_iterator` 进行遍例。需要强调的是，`list<Student>::iterator` 是定义在 `list` 中的内部类，而 `vector<Student>::iterator` 则是定义在 `vector` 中的内部类。它们虽然用法形式相同，但它们是不同的类。

由于 `list` 不支持随机访问，在访问节点时，只能从头部开始遍例来定位到指定的节点。

1.209.5 insert / erase

`list` 作为链表，`insert` 和 `erase` 对于它来说是一个普通操作（不需要像 `vector` 那样大规模移动数据）。

在 `insert` 之前，先找到其位置，

```
void AddObject(const Object& s)
{
    for(list <Object>::iterator iter = objs.begin();
        iter != objs.end(); iter ++)
    {
        Object& p = *iter;
        if(s.id < p.id) // 判断比较，插到指定位置
        {
```

```

        // 在该位置插入新元素 s
        objs.insert( iter, 1, s);
        break;
    }
}
}

```

insert 函数的第一个参数是 iterator 类型，表示插入的位置，第二个参数是元素的值类型。

删除时，先遍历整个链表查找到要删除的节点的位置，然后调用 erase 来删除。

```

void DelObject(int id)
{
    for(list <Object>::iterator iter = objs.begin();
        iter != objs.end(); iter ++)
    {
        Object& p = *iter;
        if(p.id == id)
        {
            objs.erase(iter);
            break;
        }
    }
}

```

其中，erase 的参数是 iterator 类型，指向要删除的节点。

1.210 map

map 表示的是映射表，需要引用的头文件为 <map>。

映射表中存储的每一项数据都由键和值(Key-Value)，每一项数据的 Key 无法和根据互不相同的。通过 Key 值，可以定位到该项。

映射表内部使用特殊的结构来存储数据，并使用特殊的算法来查找数据。它的主要设计目标是用于快速查找。无论表中存储了多少个项，它总能快速地用 Key 查找到对应的项。(此处换行)

对于数组和链表来说，数据项越多，查找得越慢。而映射不同，无论映射表中的项数有多少，它的查找速度几乎不变，这是映射查找和普通的线性查找的一个最显著的区别。（相关原理可以参考“数据结构与算法”中的哈希查找算法）。

查找速度快是 map 的最大优势。在需要快速定位的应用场合，一般需要用 map 来存储。其主要函数如表 30-3 所示。

表 30-3 map 的主要函数

函数名称	功能
insert	插入一个键值对
find	根据 key 来查找 value
clear	清空所有键值对
size	键值对的数目
erase	删除键值对

map 在定义时要指定 Key 和 Value 的类型，如，

```
map<int, Object> objs; // id - Object
```

第一个参数要求是可比较的，要支持操作符==，例如 int, string 都是可以的。对于一个自定义的 class 类型，如果想让它作为 Key 类型用于定位，那么一定要为它重载操作符==。

由于类型名太长，先用 typedef 定义一下。（参考本书附录《typedef 的用法》）。

```
typedef map<int,Object> ObjectMap;
```

1.210.1 使用[]插入数据

使用操作符[]，可以很方便地向 map 中插入数据，推荐使用这种方法。

```
objs[1] = a;
objs[2] = b;
```

如果 key 不存在，则新插入一个新数据项；如果 key 存在，则覆盖原来的值。示例 CH30_D1 展示了这个用法。

```
////////// 例 CH30_D1 //////////
#include <stdio.h>
#include <map>    // ① 引用头文件
using namespace std; // ② 引用命名空间
```

```

struct Object
{
    int id;
    char name[64];
};
typedef map<int,Object>  ObjectMap;
int main()
{
    ObjectMap  objs; // id - Object
    Object a = { 1, "aaaa" };
    Object b = { 2, "bbbb" };
    objs[1] = a;
    objs[2] = b;
    return 0;
}

```

1.210.2 使用 insert 插入数据

也可以使用 `insert` 来插入数据，但操作起来比较麻烦。

`insert` 的参数是类型为 `ObjectMap::value_type` 的对象，这个 `value_type` 是 `map` 的内部类，第一个参数为 `key` 值，第二个参数是 `value` 值。

`value_type` 的定义实际是一个 `pair` 模板，

```
typedef pair<const Key, Type> value_type;
```

例 CH30_D2 中展示如何向一个 `map` 变量中插入数据项。

////////// 例 CH30_D2 //////////

```

ObjectMap  objs; // id - Object
Object a = { 1, "aaaa" };
Object b = { 2, "bbbb" };
objs.insert(ObjectMap::value_type(1, a)); // 添加数据项 key-value
objs.insert(ObjectMap::value_type(2, b));

```

另外需要注意的是，如果 map 已经存在了某个 key 值，那么再次 insert 相同的 key 值是不成功的。例如，以下的代码两次插入的 key 都是 100，则第 2 次是失败的，

```
all.insert(ObjectMap::value_type(1, a)); // key: 100
all.insert(ObjectMap::value_type(1, a)); // key: 100 已经存在,此 insert 不成功
```

1.210.3 查找定位

在 map 中可以根据 key 值来查找 value 值，使用的是 find 函数，示例代码如下，

//////////例 CH30_D3 //////////

```
ObjectMap::iterator iter = objs.find(1);
if(iter != objs.end())
{
    Object& s = iter->second;
    printf("name: %s \n", s.name);
}
```

要注意 map 的 iterator 类指向的类型是 value_type，所以 iter->first 返回的 key，iter->second 返回的是 value。

1.210.4 遍例

map 根据其任务特点，一般是用来查找的，不需要遍历。但作为一个存储了数据项的集合，也是可以对它进行遍例的，通过遍例可以能够得到所有的数据项。

```
for(ObjectMap::iterator iter = objs.begin();
    iter != objs.end(); iter++)
{
    const int& key = iter->first;
    Object& val = iter->second;
    printf("key: %d , val: %s \n", key, val.name);
}
```

注意，iter->first 返回的是 const reference，iter->second 是 reference。

1.210.5 删除

从 map 中删除一个键值对的方法是，先用 find 查找一下看是否存在 key，如果存在再调用 erase 函数删除它。

```
ObjectMap::iterator iter = objs.find(20121303);

if(iter != objs.end())
{
    objs.erase(iter);
}
```

1.211 string

string 提供字符串的功能，使用时需要引用头文件<string>。string 和 vector 类似，vector 是对数组的封装，而 string 是对字符串的封装。前面有专门一章介绍了 C 风格的字符串的用法，显然，从面向对象的角度来说，C 风格字符串是难以操纵的。所以 STL 里提供了面向对象的 string 封装。

string 内部仍然是维护一个 char 型数组，并且也是以 0 结尾的。对外提供了字符串操作常用的函数，如 append, clear, at, length 等操作，并支持赋值、复制等操作，重载了必要的操作符，使用用户可以一个对象的视角来操作字符串。其主要函数如表 30-4 所示。

表 30-4 string 的主要函数

函数名称	功能
append	附加字符串
clear	清空
capacity	容量
size	实际长度
length	实际长度，等同于 size 函数
at	按索引访问字符
find	查找一个字符或一个子串
rfind	从后往前查找一个字符或一个子串
find_first_of	查找匹配若干字符串中的一个字符
find_first_not_of	查找不匹配若干字符串的一个字符
find_last_of	从后往前查找，匹配若干字符中的一个字符

find_last_not_of	从后往前查找，不匹配若干字符中的一个字符
substr	取得一个子串
insert	插入字符或子串
replace	替换字符或子串

构造 string 对象可以用以下几种方式:

```
string str1 ("LiMing"); // 以一个 C 风格字符串构造
string str2 = "WangHua"; // 同上一种方式
string str3 ("abcde", 5); // 参数 1 是 C 字符串，参数 2 参数是长度
string str4; // 空字符串
string str5 = ""; // 同上，空字符串
```

可以用 c_str 函数来获取 string 内部的字符串指针，

```
const char* str = str1.c_str();
```

注：以下写法错误，一定要不用 NULL 来初始化 string，否则是程序会崩溃。

```
string str6 = NULL; // 运行时错误！不能以 NULL 来初始化 string
```

1.211.1 append / clear

append 用于在末尾字符若干字符或字符串，例如，

```
//////////例 CH30_E1 //////////
#include <stdio.h>
#include <string> // ① 引用头文件
using namespace std; // ② 引用命名空间
int main()
{
    string str;
    str.append("something else"); // 附加一个字符串
    str.append("abcde", 5); // 附加一个字符串：拷贝前 5 个字符
    str.append("abcde", 1, 3); // 附加一个字符串：
        // 起点 offset=1, 长度 3，即拷贝"bcd"
```

```

    str.append(2, 'h'); // 附加 2 个 'h' 字符
    return 0;
}

```

string 重载了操作符 +=，也是用于附加字符串的功能，

```

    str += "Hello"; // 附加一个串
    str += "a";    // 附加单个字符
    str += 'b';
    str += 'c';

```

和 vector 一样，当附加字符串时，内部缓冲区 size = capacity 时会自动扩充缓冲区：先申请一个块更大的缓冲区，再把原有数据拷贝过去。

1.211.2 size / capacity / resize / clear / length

string 的内部存储原理和 vector 相同。size 用于得到已经存储的字符的长度（不包括末尾的 0），capacity 表示剩余可用的长度。length 的功能和 size 完全相同。

resize 用于显式地设置 string 内部缓冲区的容量 capacity，clear 用于清空缓冲区内的所有字符。如果希望避免 append 低效的问题，可以事先设定一个大缓冲区，

```

    string str;

    str.resize (16*1024); // 设定最大容量为 16K

    str.clear(); // 清空

```

1.211.3 at

at 用于按索引访问字符，at 函数返回的是一个 reference，因为可以用 at 来修改字符串的内容。例如，

```

    str.at(0) = 'K';
    char ch = str.at(0);

```

string 也重载了操作符[]，作用与 at 等同，例如，

```

    str[0] = 'K';

```

示例 CH30_E2 中，函数 ToLowercase 用于将一个 string 对象从大写变为小写。代码如下所示：

```

//////////例 CH30_E2 //////////
#include <stdio.h>

```

```

#include <string>    // ① 引用头文件

using namespace std; // ② 引用命名空间

void ToLowercase(string& str)
{
    for(int i=0; i<str.length(); i++)
    {
        char& ch = str.at(i); // 取得该位置的 reference
        if(ch >= 'A' && ch <= 'Z')
        {
            ch += ('a' - 'A');
        }
    }
}

int main()
{
    string str = "Hello,World";
    ToLowercase(str);
    printf("Result: %s \n", str.c_str());
    return 0;
}

```

1.211.4 字符串比较

C 风格字符串使用 `strcmp` 函数进行大小比较，而 `string` 是重载了所有关系操作符 (`==`, `!=`, `>`, `>=`, `<`, `<=`)，所以比较起来会更直观一些。

例如，

```

string str1 ("LiMing");
string str2 = "WangHua";
if(str1 < str2)
{
    printf("smaller !");
}
else if(str1 == str2)

```

```

{
    printf("equal");
}
else
{
    printf("bigger");
}

```

1.211.5 字符串查找

C 风格字符串用 `strstr` 函数进行子串查找，并不是太方便。`string` 提供若干查找函数，大大的方便了查找操作。

`find` 函数用于查找一个字符或一个子串，从左到右查找，直到第一次匹配时，返回匹配的位置。如果没有查找到任何匹配，则返回-1。这里所说的位置或偏移，是从 0 开始的，和数组下标的记法相同。

例如，

```
string str = "LiMing is doing homework";
```

查找字符 `i` 的位置，

```
int pos = str.find('i'); // 返回 1
```

源字符串 `str` 中包含了若干个 `i`，第一处 `i` 出现的位置是 1，所以上面的代码返回就 1。如果要接着往后查找，可以为 `find` 指定第 2 个参数，该参数表示起始位置偏移 `offset`。

例，

```
int pos = str.find ('i', 2); // 返回 3
```

以上代码中，第二个参数为 2，表明从位置 2 开始查找，在查找到 `str[3]` 时发现匹配，所以返回 3。

`find` 可以直接查找一个子串，例如，以下代码用于查找子串 "ing"，

```
int pos = str.find("ing"); // 返回 4
```

从左到右查找 "ing"，第一次匹配时，返回子串的首字符 `i` 的位置。

以下代码是在第一次查找的基础上，继续后往查找相同的字符串，

```
int pos = str.find("ing", pos + 3);
```

`rfind` 用于从右往左（或称从后往前）来查找字符或子串，或没有匹配则返回-1，则有匹配则返回子串的首字符的位置。例如，

```
int pos = str.rfind("ing"); // 从右往左查找
pos = str.rfind("ing", pos - 3); // 指定起始点
```

`find_first_of` 函数用于查找若干字符中的一个，只要其中任意一个字符匹配，函数即返回此字符的位置。查找的顺序是从左到右。以下代码用于在字符串中查找第一个元音字母，

```
int pos = str1.find_first_of("aeiouAEIOU");
```

`find_first_not_of` 函数的功能与 `find_first_of` 恰恰相反，以下代码用于查找第一个非元音字母，

```
int pos = str1.find_first_not_of("aeiouAEIOU");
```

同样的，还有 `find_last_of` 和 `find_last_not_of`，功能类似，但是从右往左的顺序查找的。

1.211.6 substr

`substr` 函数用于拷贝部分子串，带两个参数：第一个参数 `offset` 表示起始位置，第二个参数 `len` 表示要截取的子串的长度。第二个参数如果省略，则表示一直截取到末尾。

```
string src ("abcdefg");
string str1 = src.substr(4); // 返回"efg"
string str2 = src.substr(4,2); // 返回"ef"
```

例：姓名字符串由两部分组成，中间以多个空格或 TAB 分开，要求写一个函数将姓和名都提取出来。若输入"Li Ming"，则解析出姓为"Li"名为"Ming"。

////////// 例 CH30_E3 //////////

```
void ParseName(const string& name)
{
    int p1 = name.find_first_of(" \t"); // 空格, TAB
    if(p1 >= 0) // 找到分隔符
    {
        string first_name = name.substr(0, p1);
```

```

    int p2 = name.find_first_not_of(" \t", p1);
    if(p2 >= 0)
    {
        string second_name = name.substr(p2);
        printf("%s,%s\n", first_name.c_str(), second_name.c_str());
    }
}
}

```

1.211.7 insert / replace

和 `vector` 一样，`string` 也不适宜进行插入和删除操作。因为内部的内符是紧密排列的，插入字符则意味着后面的所有字符都要后移，而删除字符串则后面的字符需要前移。因此插入和删除都是涉及大规模的数据移动，如果频繁调用则比较耗费 CPU。

`insert` 函数用于向字符串中间插入字符或子串，用几种形式，

```
string& insert (int pos, const char* str); // 在 pos 处插入 str
```

```
string& insert (int pos, const char* str, int count); // 在 pos 处插入 str[0, count-1]
```

```
string& insert (int pos, const char* str, int offset, int count); // 插入 str[offset, count-1]
```

```
string& insert (int pos, int count, char ch); // 插入若干个字符
```

`replace` 函数用于替换 `string` 中间一个子串 `src`，替换为另外一个子串 `dest`。新子串 `dest` 可能比旧子串 `src` 要长，也可能比 `src` 要短。`replace` 也重载了若干个版本，

```
string& replace(int pos, int num, const char* str);
```

```
string& replace(int pos, int num, const char* str, int count);
```

```
string& replace(int pos, int num, const char* str, int offset, int count);
```

```
string& replace(int pos, int num, int count, char ch);
```

其中，`pos` 和 `num` 共同指定了要替换旧子串 `src` 的位置和长度，而其他参数则指定了新子串 `dest` 的内容和长度。

例如，某工程中有一个 SQL 语中含有两个待定字段{0},{1}，要替换为具体的值。可以先写一个 `do_replace` 的函数，

```
////////// 例 CH30_E4 //////////
```

```
void DoReplace(string& str, const string& src, const string& dst)
```

```
{
```

```

while(1)
{
    int pos = str.find(src); // 查找源子串 src
    if(pos < 0) break; // 不含子串时，退出
    // 替换为目标子串 dst
    str.replace(pos, src.length(), dst);
}
}

```

然后像下面这样调用 DoReplace，完成目标功能，

////////// 例 CH30_E4 //////////

```

string sql = "SELECT * FROM student WHERE id={0} AND name={`{1}` ";
DoReplace(sql, "{0}", "12");
DoReplace(sql, "{1}", "LiMing");
printf("SQL: %s\n", sql.c_str());

```

如果要实现删除子串，也可以借助 replace 函数来实现，只是将目标子串的长度设为 0 就可以了。例如，

```
DoReplace (where, "{0}", ""); // 替换为空串，也就是删除的意思
```

1.211.8 string 作为函数参数

作为函数参数时，string 有几种形式: const string&, string& 和 string。例如，

```

void test ( const string& str );
void test ( string& str );
void test ( string str );

```

而在正规的代码中，一般只会使用 const string& 或 string& 作为参数。

首先，string 作为参数的情况不会遇到，这就是一个简单的“传值/传引用”的问题。传引用的效率比传值要高，传引用时不需要数据拷贝，而传值需要数据拷贝。所以，像这样函数参数一般不要使用，

```
void test (string str); // 传值方式，效率低！
```

其次，`const string&` 和 `string&` 的使用场合，用户要仔细分辨。一般来说，当需要修改参数本身时才用 `string&`；否则一律用 `const string&` 作为参数。正如前面的 `do_replace` 函数的参数形式，

```
void DoReplace (string& str, const string& src, const string& dst)
```

第一个参数是 `string&` 形式，这是因为函数中要对参数进行修改（输出参数）。`src` 和 `dst` 都是 `const string&` 形式，因为函数中不需要对其进行修改。

当使用 `const string&` 作为参数时，实参可以传入 `const char*`，也可以传入 `string`，例如，有函数 `test` 的原型如下：

```
void test (const string& str);
```

调用代码示例如下：

```
test ("hello"); // 可以直接传一个 char*，相当于 test (string("hello"));  
  
test (str); // 也可以传入一个 string 类型的对象
```

1.212stack

`stack` 表示数据结构的栈。前面说过，可以用 `list` 来实现栈。由于栈只是需要对一端进行操作，所以 `vector` 也可以实现。而这一节的目的是向读者介绍栈的常用接口和使用习惯。其主要函数如表 30-5 所示。

表 30-5 stack 的主要函数

函数名称	功能
push	推入（可用 <code>list::push_back</code> 替代）
pop	弹出（可用 <code>list::pop_back</code> 替代）
top	取顶部元素（可用 <code>list::back</code> 替代）
size	内容长度（可用 <code>list::size</code> 替代）

对于栈来说总是在一端进行操作的，称之为顶端。用 `push` 函数从顶部压入一个对象，用 `pop` 可以弹出一个对象，而 `top` 函数则是取得顶部对象的值。

////////// 例 CH30_F1 //////////////////////////////////////

```
Object a = {1, "aaaa"};  
Object b = {2, "bbbb"};  
  
stack<Object> stk; // 定义一个 stack
```



```
stk.push(a); // 推入一个对象

stk.push(b); // 再推入一个对象

stk.pop(); // 弹出一个对象

if(stk.size() > 0)
{
    Object & obj = stk.top(); // 取得顶部对象，返回 reference
    printf("top element: id=%d, name=%s \n", obj.id, obj.name);
}
```

1.213 queue

用样的，队列 queue 并不是必须要掌握的内容，完全可以 list 来代替它。本节的目的是向读者介绍队列的一般接口和使用模式。其主要函数如下 30-6 所示。

表 30-6 queue 的主要函数

函数名称	功能
push	推到尾部（可用 list::push_back 替代）
pop	从头部取出（可用 list::pop_front 替代）
front	取首元素（可用 list::front 替代）
back	取末元素（可用 list::back 替代）
size	长度（可用 list::size 替代）

第 31 章 异常

本章讨论 C++ 里的异常处理机制。所谓异常，就是发生了不该发生的事。然而，无论是该发生的事，还是不该发生的事，我们都是要处理的，不能视而不见。在程序设计中，把该发生的事，称为正常情况；不该发生的事，称为异常情况。

举例说明一下。假定我们的任务是开发一个简易计算器，该计算器可以处理简单的加减乘除的算术式子。用户输入一个字符串，程序返回其运算结果。输出例如，当输入 "12+13" 则返回 25，当输入 "12*13" 则 156。

那么，在这个程序的设计中，我们应该把所有可能发生的事情都考虑到。正常情况下，用户输入了一个合法的算式，我们应该给出运算的结果。异常情况有很多，比如，用户输入 "12/0" 时，由于分母为 0 计算无法进行，我们应该提示它 "分母不得为 0"。比如，用户输入 "12+" 那么这个式子不完整，也是一种异常情况。

在 C/C++ 里，有两种方式来处理异常。一种方式是我们最常使用的，就是错误码方式。简单地说，就是一个函数在成功时返回 0，在不同的异常情况下返回不同的错码。其大致处理框架可以形象的表示如下：

```
int DoSomething()
{
    if( 异常情况 A)
        return -10;
    if( 异常情况 B)
        return -11;
    return 0; // 正常情况
}
```

异常的另一种处理方式，就是本章要介绍的 exception 机制。

1.214 一个例子

C++ 的 exception 处理框架，是由三个关键字实现的：throw / try / catch。其中，throw 可以理解为抛，try 相当于监视，catch 意思捕捉。

- ① throw：当错误发生时，把错误信息 throw 出去。
- ② try...catch：监视一段代码，捕捉里面的错误。

我们用一个简单的例子来展示这一套机制的基本用法。

例如，我们实现一个函数，其作用是将输入的十六进制字符串，转换为整数。

正常情况：例如，输入"0A" 返回 10

异常情况：

① 含有非法字符， 比如， "1AG2"则 G 是非法字节

② 太长，超过了 8 个字符，比如"AAAABBBBCCCC"。由于返回值是 32 位整数，最多只能 8 个字节。

先实现它的正常处理流程，在例 CH31_A1 中展示了它的正常处理流程：当输入字符串在长度小于等于 8、并且保证不含非法字符时，函数 Hex2Int 可以正确完成其使命。

////////// 例 CH31_A1 //////////

```
unsigned int Hex2Int(const char* str)
{
    int size = strlen(str);
    unsigned int result = 0;
    for(int i=0; i<size; i++)
    {
        char ch = str[i];
        unsigned int value = 0;
        if(ch >= 'a' && ch <= 'f')
            value = ch - 'a' + 10;
        else if(ch >= 'A' && ch <= 'F')
            value = ch - 'A' + 10;
        else if(ch >= '0' && ch <= '9')
            value = ch - '0';
        result = result * 16 + value;
    }
    return result;
}
```

下面我们就介绍一下，如何用 exception 机制来处理它的异常情况。

1.214.1 throw

throw 的设计思想很简单，用一句话描述就是“发现错误，就把错误信息 throw 出去”。其语法形式是：

```
throw object ;
```

其中，object 可以是任何类型的对象，比如，一个 int 型的值，一个 double 型的值，一个任何自定义 class 的对象。但一般来说，被 throw 的值中应该携带能描述错误的信息。在 exception 机制中，把被 throw 出去的对象称为 exception。

在示例 CH31_A2 中，我们添加了两行 throw 语句，被 throw 的值是一个 int 型的值，表示错误码。

```
////////// 例 CH31_A2 //////////

unsigned int Hex2Int(const char* str)
{
    int size = strlen(str);
    if(size > 8)
        throw -101; // 把错误信息 throw 出去

    unsigned int result = 0;
    for(int i=0; i<size; i++)
    {
        char ch = str[i];
        unsigned int value = 0;
        if(ch >= 'a' && ch <= 'f')
            value = ch - 'a' + 10;
        else if(ch >= 'A' && ch <= 'F')
            value = ch - 'A' + 10;
        else if(ch >= '0' && ch <= '9')
            value = ch - '0';
        else
            throw -102; // 把错误信息 throw 出去

        result = result * 16 + value;
    }
    return result;
}
```

可以看出，throw 的好处是，几乎不影响原有的代码结构，直接在出错的地方，加上一行 throw 就可以了。

1.214.2 try...catch

使用 try...catch 框架，可以监视一段代码里有没有 exception 对象被抛出。

////////// 例 CH31_A2 //////////

```
int main()
{
    try{
        unsigned int result = Hex2Int("ABCDG");
        printf("Got: %u", result);
    }
    catch(int err)
    {
        printf("Error: %d \n", err);
    }
    return 0;
}
```

其中，try 语句用于监视，如果 try 的大括号内没有 exception 被抛出，则按顺序执行。如果有 exception 被抛出，则退出 try 语句，进入匹配的 catch 语句内进行处理。

在上例中，传入参数"ABCDG"含有不合法字符，在 Hex2Int 内有一个 int 型的 exception 被抛出。程序立即退出 try 语句，printf("Got: %u", result)不被执行。寻找到类型匹配的 catch 语句，进入里面执行 printf("Error: %d \n", err)。

1.215 try 的用法

try 和 catch 总是配套使用的，不能只有 try 没有 catch，或者有 catch 没 try。

try 用于监视大括号内的语句，当内部有异常被抛出时，则退出 try 语句，出错行后面的语句不会被执行。

例如，由于 try 语句内有异常被抛出，则立即退出 try 语句，后面的 printf("Won't get here \n")不会被执行。

////////// CH31_B1 //////////

```
#include <stdio.h>
```

```
int main()
```

```
{
    try
```

```

    {
        throw 12.34;
        printf("Won't get here \n"); // 此行不被执行
    }
    catch(double ex)
    {
    }
    printf("main exit \n");
    return 0;
}

```

又如，在 CH31_B2 中，A()内有异常被抛出，使用 `try` 立即退出，A()后面的 `printf("Won't get here \n")`不会被执行。

////////// 例 CH31_B2 //////////

```

#include <stdio.h>

void A()
{
    throw 12.34;
}

int main()
{
    try
    {
        A();
        printf("Won't get here \n"); // 此行不被执行
    }
    catch(double ex)
    {
    }
    printf("main exit \n");
    return 0;
}

```

1.216 catch 的用法

当 try 语句内监测到有异常被抛出时，首先退出 try 语句，然后试图寻找匹配的 catch 语句。如果有 catch 语句匹配，则进入相应的 catch 语句执行。如果所有的 catch 语句都不匹配，则此异常对象会继承上向一层函数抛出。

当异常对象的类型和 catch 的参数类型相同时，catch 语句被匹配。

在示例 CH31_B1 中，try 语句中有一个 double 型异常被抛出，后面有两个 catch 语句，但是只有参数为 double 类型的 catch 语句是匹配的。

```
////////// CH31_C1 //////////  
  
#include <stdio.h>  
  
int main()  
{  
    try{  
        throw 1.2;  
    }  
    catch(int ex) // 参数类型与异常类型不匹配  
    {  
        printf("Got exception: (int) %d \n", ex);  
    }  
    catch(double ex) // 参数类型匹配  
    {  
        printf("Got exception: (double) %lf \n", ex);  
    }  
    return 0;  
}
```

当有 catch 语句被匹配时，则该异常对象被成功捕捉，或称异常被捕获。

1.216.1 匹配所有类型

当 catch 的参数为省略号时，可以用于匹配所有类型的异常。通常放在所有 catch 语句的最后面，表示缺省的异常处理。例如，

```
////////// 例 CH31_C2 //////////  
  
#include <stdio.h>  
  
int main()
```

```

{
    try{
        throw 1.2;
    }
    catch(int ex)
    {
        printf("Got exception: (int) %d \n", ex);
    }
    catch(...)
    {
        printf("Default Handler ...\n");
    }
    return 0;
}

```

这段 try...catch 代码表示：当异常类型为 int 时，进入第一个 catch 进行处理；当异常类型为其他类型时，统一进入第二个 catch 进行处理。

1.216.2 异常必须被捕获

当一个 exception 被抛出后，它必须被某个 catch 语句捕获。

假设函数调用层次是 main() -> A() -> B() -> C()，在 C()中抛出一个异常，那么首先看 C()中有被有把这个异常 catch 住，如未被捕获，则继续抛到 B()中... 如果一直到顶层的 main()函数，该异常仍未被捕获，则程序立即退出（程序崩溃）。

在例 CH31_C3 中，由于在函数 C()中抛出的异常对象未被任何一级的函数捕获，所以程序崩溃退出。

```

//////////////////// CH31_C3 //////////////////////
#include <stdio.h>

void C()
{
    throw 1.2;
    printf("C exit\n");
}

void B()
{

```



```

        C();
        printf("B exit\n");
    }
void A()
{
    try{
        B();
    }
    catch(int ex)
    {
        printf("Got: (int) %d \n", ex);
    }
    printf("A exit\n");
}
int main()
{
    A();
    printf("main exit \n");
    return 0;
}

```

我们注意到，在函数 A() 中存在 try...catch 的监视，但是 catch 语句未能捕获到异常，所以异常对象被继续向上层抛出。

1.217 throw 的用法

前面的例子已经演示了抛出 int 型数值，double 型数值。实际上，任意类型的对象都可以被抛出。

例如，我们自定义一个类 IOException，用于描述错误信息。该类有两个成员：一个是错误码，一个是错误描述。

```

////////// 例 CH31_D1 //////////
class IOException
{
public:

```

```

IOException(int error, const char* descr)
{
    this->error = error;
    strcpy(this->descr, descr);
}

public:
    int error;
    char descr[128];
};

```

在异常情况发生时，可以构造这么一个 Exception 被抛出，

```

void A()
{
    IOException ex (-10, "File is readonly");
    throw ex;
}

```

或者直接使用无名对象，也是可以的，

```

void A()
{
    throw IOException(-10, "File is readonly");
}

```

这样的异常对象类型比先前的 int, double 要携带了更多的信息，我们不但知道错误码，还能通过 ex.descr 获取描述信息。

```

int main()
{
    try{
        A();
        printf("Won't get here \n"); // 此行不被执行
    } catch(IOException ex)
    {
        printf("(%d) %s \n", ex.error, ex.descr);
    }
}

```

```

    printf("main exit \n");
    return 0;
}

```

throw 语句的注意事项:

- (1) 可以 throw 任意类型的值，但一般要求该值应该携带错误信息，以便上层可以定位和显示错误。
- (2) 当 throw 语句被执行后，函数立即退出，并且将异常对象向上层抛出。

1.218 常见问题

1.218.1 异常与错误的区别

try...catch 能捕获的是异常对象，而不是普通的错误。只有被 throw 语句抛出来的值，才是异常对象，这是一个严格的语法定义。

例如，在 CH31_E1 中，try 语句中函数 A() 对一个空指针进行星号操作，这显然是一个致命错误。但 catch(...) 无法捕获。

```

////////// CH31_E1 //////////
void A(char* p)
{
    *p = 123; // 访问空指针
}

int main()
{
    try{
        A(NULL);
    }catch(...)
    {
        printf("catch all exception...\n");
    }
    return 0;
}

```

在例 CH31_E2 中展示了正确的异常处理办法，即用 throw 来抛出一个异常对象。

```

////////// CH31_E2 //////////
#include <stdio.h>

class NullPointerException // 定义一个类型
{
};

void A(char* p)
{
    if(p == NULL)
        throw NullPointerException(); // 抛出一个对象

    *p = 123; // 访问空指针
}

int main()
{
    try{
        A(NULL);
    }catch(NullPointerException ex) // 捕获这个异常对象
    {
        printf("Got NullPointerException...\n");
    }
    return 0;
}

```

1.218.2 不在构造函数中抛出异常

我们曾一再强调，未初始化函数对象不能称之为对象。类的构造函数用于初始化一个对象，如果在构造函数抛出异常会发生什么事呢？一个对象被构造了一半就退出，这是很不健康的设计方式。

1.218.3 不在析构函数中抛出异常

类的析构函数中也不能抛出异常。

因为在异常被抛出时，当时函数中的所有局部对象都被析构。如果在析构的同时又抛出异常，那么这个就是语法未定义的情形了。结果往往导致程序紊乱。

1.218.4 关于 finally

C++标准里不支持 `finally` 语句，如果你在其他语言（如 `java`）里使用过 `finally`，那么请记住 C++里是没有 `finally` 语法的。

但是某些编译器定义了自己的异常机制，例如 VC 中用 `__try ... __catch ... __finally` 来实现了 `finally` 的功能。笔者不推荐使用与编译器相依赖的语法，这将使你的代码失去可移植性。要相信，使用标准的 `try...catch` 就足够实现你的需求了。

最后，需要说明的是，在是否应该 `try...catch` 机制上面或许还存在争议。一方面，从程序的可读性、和效率方面考虑，有意见认为不适合使用异常机制。另一方面，C++是由 C 发展而来，C/C++的一贯风格是使用错误码来处理异常的。