*第十章 拾遗*

第一节：结构体成员函数

第二节：C语言中的bool

第三节：预编译指令与条件编译

第四节：位运算与位段结构

我们说过，C语言是一门易学难精的编程语言，前面9章的内容已经涵盖了C语言的绝大部分语法规则。如果读者已经顺利来到第10章，那么说明读者已经学会了C语言的基础（而不是学会了C语言）。而有些C语言知识比较分散，无法被添加到前面的章节中，所以我们把这些知识单独抽取出来，组成最后一章。

10.1结构体成员函数

我们知道，结构体的成员可以是任何基本数据类型，也可以是任何自定义类型（结构体、联合体、枚举）。而事实上，结构体的成员甚至可以是函数，在第7章，我们曾介绍过函数指针，往结构体的成员中添加函数的关键就是函数指针。

1. #include <stdio.h>
2. #include <stdlib.h>
3. typedef struct node
4. {
5. int data;
6. void(\*getData)(struct node \*);
7. void(\*showData)(struct node \*);
8. }node;
9. node \*createNode();
10. void deleteNode(node \*);
11. void getData(node \*);
12. void showData(node \*);
13. int main()
14. {
15. node \*n=createNode();
16. n->getData(n);
17. n->showData(n);
18. deleteNode(n);
19. getchar();
20. return 0;
21. }
22. node \*createNode()
23. {
24. node \*temp = malloc(sizeof(node));
25. if (temp == NULL)
26. {
27. printf("内存分配失败\n");
28. getchar();
29. exit(1);
30. }
31. temp->getData = getData;
32. temp->showData = showData;
33. }
34. void deleteNode(node \*this)
35. {
36. free(this);
37. }
38. void getData(node \*this)
39. {
40. printf("请输入一个整数：\n");
41. rewind(stdin);
42. scanf("%d", &(this->data));
43. rewind(stdin);
44. }
45. void showData(node \*this)
46. {
47. printf("data的值：%d",this->data);
48. }

程序10-1-1

如果发现程序10-1-1不能通过编译，请检查源文件的后缀，如果是.cpp，改成.c即可。10-1-1是一个合法的C语言程序，但它不是合法的C++程序，this是C++中的一个关键字，有着特殊的含义，但在C语言中，它可以被当作标识符来被我们定义。

程序的某次运行结果如下：

请输入一个整数：

**10**

data的值：10

我们在第3到8行声明了一个结构体struct node，它有3个成员，data是它的数据，getData和showData则是它的“成员函数”，这两个函数的作用是对结构体执行某种操作。

createNode的作用是创建并初始化一个node的实例变量，可以看出，我们没有初始化data，而是设置了getData和showData两个函数指针的值，它们分别指向getData和showData函数，这里不用担心重命名的问题，就好像我们完全可以把data改成main一样。

之后，我们用n->getData(n)的形式来输入了n->data的值，然后用类似的方式输出了n->data。

最后，我们用deleteNode来销毁了之前创建的实例变量。

C语言是一门优秀的语言，也是很合适的程序设计入门语言，但是通常来讲，一个程序员或软件工程师都会不止一门编程语言。除了C语言，还应该学习面向对象的编程语言。而我们之所以介绍结构体的成员函数，就是为了在一定程度上解释面向对象的编程语言的底层实现。

为什么我们非要用createNode和deleteNode函数来创建和销毁实例变量？因为这代表了面向对象中的构造函数（**constructor**）与析构函数（**destructor**），这两个函数的作用分别是在实例变量创建的时候为一些成员数据做初始化工作（也包括内存申请）和在实例变量不再有用时回收它并做一些收尾工作（包括回收实例变量本身和在构造函数中申请的内存）。

在真正的面向对象编程语言中，当我们想调用它的某个成员函数的时候，是不需要把它自己的地址当成参数传递的，比如n->getData(n)直接写成n->getData()即可。但是所谓的不需要，只是对于程序员来说的不需要，编译器还是为生成一个隐藏的指针来接受实例变量的地址，这个指针就叫this指针，面向对象的编程语言中，this通常会是一个关键字。程序10-1-1中，我们放弃了对C++的兼容，把C++的关键字this作为我们自定义的标识符来使用，其目的就在于向读者展示面向对象编程语言的一些底层细节。

需要注意的是，对于面向对象的编程语言来说，成员函数占用的内存是固定的，再多的实例变量也不会增加成员函数占用的内存。但是在C语言中，结构体的“成员函数”却需要占用内存，实例变量越多，占用内存越多。程序10-1-1介绍的思想是面向对象特性中最基本的“封装”，也是C语言能最大程度模仿的面向对象特性。这种思想是很好的，但是实际上是没有必要往结构体中添加函数指针的，getData(n)甚至还比n->getData()方便。

10.2 C语言中的bool

在C语言中，我们通常用0表示假，用非0的值表示真，在前面的章节中我们也都是这么做的。很多编程语言中都有bool型的数据用来专门表示真或假，bool型的数据取值只有2个：true和false，分别表示真和假。早期的C语言中，表示bool型数据的方式有很多，有用char的，也有用int的。而C99及之后的C语言标准都支持bool型数据，它就是\_Bool，它只占用1个字节的内存，不过可取的值仍然不止true和false。

C99标准中，除了\_Bool，还引入了一个相关的头文件，即stdbool.h，在这个头文件中，有这样几个宏：

#define bool \_Bool

#define true 1

#define false 0

如果包含了stdbool.h头文件，我们就能像其它高级语言那样使用bool、true、false这3个“关键字”了。

1. #include <stdio.h>
2. #include <stdbool.h>
3. int main()
4. {
5. printf("\_Bool的大小是：%d\n", sizeof(\_Bool));
6. printf("bool的大小是：%d\n", sizeof(bool));
7. bool b = true;
8. if (b)
9. {
10. printf("b的值为true\n");
11. }
12. getchar();
13. return 0;
14. }

程序10-2-1

程序的运行结果如下：

\_Bool的大小是：1

bool的大小是：1

b的值为true

10.3预编译指令与条件编译

我们对预编译指令早就不陌生了，前面的章节中也见过很多预编译指令了。以#开头的语句就是预编译指令，并且预编译指令不以分号结尾。#include和#define都是我们再熟悉不过的了，#pragma也见过几次。

这一节中，我们将主要介绍与条件编译有关的预编译指令。程序10-1-1以及之前章节中的很多程序都无法在C++编译器中通过编译，但是用户接受到的提示信息只是“this只能在非静态成员函数或非静态数据成员初始值设定项的内部引用”或者其它的一些莫名其妙的错误提示，显然解决问题的方案是把文件后缀改成.c，但是用户无法从这样的错误提示中找出有用的信息。为了解决这个问题，我们将用上#erro指令。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #ifdef \_\_cplusplus
4. #erro 你只能在C语言环境下编译此文件，或者你需要把文件后缀改成.c
5. #endif
6. int main()
7. {
8. int \*p=malloc(4);
9. free(p);
10. getchar();
11. return 0;
12. }

程序10-3-1

我们知道，int \*p=malloc(4)这样的写法只在C语言中成立，在C++中必须写成int \*p=(int \*)malloc(4)或者int \*p=reinterpret\_cast<int \*>malloc(4)（reinterpret\_cast是C++的一种强制类型转换运算符，和C语言不同，C++拥有很多种类型转换运算符用来处理不同的情况）。

为了能让程序10-3-1能在文件后缀为.cpp的时候给用户以清晰的提示，我们用到了#erro。#ifdef \_\_cplusplus和#endif是成对出现的，就好像之前第8章介绍的#ifndef和#endif一样。#ifdef的意思是，如果定义了某某宏，就怎样怎样，直到遇见#endif。\_\_cplusplus（c前面有两个下划线）是编译器预定义的一个宏，当编译器以C++的方式来编译一个文件的时候，就会为这个文件自动生成这个宏，所以我们可以以此为依据，在预编译阶段判断文件后缀是否为.cpp。如果#ifdef后面的条件满足，那么#erro就会执行，#erro的作用是停止编译并输出错误信息，这里我们写的错误提示非常明确了，用户很轻易就能看出问题所在。

不过，这样还是不够的，有没有一种方法，能让程序10-3-1能同时在C和C++的编译器中被编译成功呢？答案是可以的，这就需要用到#else了。#else通常和#if、#ifdef或者#ifndef一起出现，一个#if和#endif之间可以有多个#else。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*p;
6. #ifndef \_\_cplusplus
7. p = malloc(4);
8. #else
9. p = new int[1];
10. #endif
11. #ifndef \_\_cplusplus
12. free(p);
13. #else
14. delete p;
15. #endif
16. getchar();
17. return 0;
18. }

程序10-3-2

无论是在C还是C++编译器中，程序10-3-2都能通过编译。如果\_\_cplusplus宏没有定义，那么编译器将会编译第7行和第13行，反之则会编译第9行和第15行。new和delete是C++中的和动态内存操作有关的关键字，我们只需要知道它的作用即可，其它的不必过多了解。很明显，第11行的空白行可以插入很多其它代码，这些代码也许会对p指向的内存做各种操作，不管p的内存是通过哪种方式产生的。

常用的预编译指令还有#elif和#undef，#elif相当于else if（但是不需要对应一个#endif）。而#undef的作用则是取消已经定义好的一个宏，方便对其重新定义。

1. #include <stdio.h>
2. #define NUM 10
3. #ifndef NUM
4. #define NUM 20
5. #elif NUM!=20
6. #undef NUM
7. #define NUM 20
8. #endif
9. int main()
10. {
11. printf("%d",NUM);
12. getchar();
13. return 0;
14. }

程序10-3-3

程序10-3-3中，演示了#elif和#undef的使用方式。第3到第12行的意思是，如果没有定义NUM这个宏，那我们就定义一个，值为20，而如果已经定义了NUM，并且NUM的值不为20，那么就取消之前已经定义的NUM，并且重新定义一个NUM，其值为20。所以第2行的#define NUM 10改成#define NUM 20或者其它都不会影响程序的运行结果。

10.4位运算与位段结构

之前的学习中，我们处理内存的最小单位一直都是字节，然而我们知道，在字节下面，还有更小的单位，即“位”，8个位组成1个字节。我们不仅可以对整个字节进行操作，还可以对单个的位做各种运算，很多时候位运算都会使程序效率有所提升。

位运算分为6种类型，取反、左移、右移、与、或、异或。

在具体地介绍各种位运算之前，我们先简单讲解整数在内存中的表示。我们知道，char类型的数据占用1个字节，也就是8位，这意味着它能表示的数字有256个。我们又说，char类型数据能表示的范围是-128~127，而不是想象中的0~255，我们可以大胆地猜测，128~255实际上对应于-128~-1，只是计算机通过某种“协议”，将128~255的数判定为负数。

128~255对应的二进制是10000000~11111111，问题的关键就在于第1位，当第1位为0的时候，计算机认为这是一个正数，当第1位为1的时候则认为是负数。我们知道，不管是int还是short还是char，都可以用unsigned来修饰，unsigned的意思是无符号，如果一个变量是unsigned的，那么它的第1位就不会被当成符号位。同样是10000000，当它储存在char型数据里的时候，它就是-128，否则就是128。

现在我们知道怎么判断一个数是否为负数了，那么我们应该怎么得到一个负数的二进制形式呢？以char型数据为例，通过观察不难发现，1+(-1)的值是256（-1实际上是255），而对于任意的两个数x和-x（0<x<128），我们不妨认为x=1+n，那么-x很明显就是-1-n，最终，x+(-x)的值也就等于1+(-1)等于256了。

255对应的二进制数是11111111，两个char型数据的和为255的时候，它们正好是“互补”的，比如10000000和01111111，即第1个数字为1的位，第2个数字的对应位就为0，反之亦然。所以，求一个负数-x的二进制形式的步骤是：

1. 求出x的二进制形式y
2. 求出与y“互补”的数z
3. 在z的基础上+1

这样，最终的结果就是-x的二进制形式了。

说完了负数的储存，我们来介绍一下取反运算。取反运算等价于我们刚刚说到的求“互补”的数。取反运算符是：~，需要注意的是，取反运算不会改变原有的数，而是会返回取反后的值。

1. #include <stdio.h>
2. int main()
3. {
4. char v1 =100, v2;
5. v2 = ~v1;
6. printf("%d %d ",v1,v2);
7. getchar();
8. return 0;
9. }

程序10-4-1

程序运行结果如下：

100 -101

可以看出，取反运算没有改变v1本身的值，但是结果却被v2捕捉到了，我们说过，对于一个0~128之间的数x来说，-x=~x+1，所以~x就等于-x-1了，这样我们就能理解为什么v2的值是-101了。

我们再来看看左移运算和右移运算，它们运算符分别是<<和>>，作用是把一个数的每1位向左或者向右移动指定的位数，比如x<<1就表示把x的每1位都向左移动1位。我们同样以char类型的数据为例，如果x的值为1，那么它对应的二进制形式就是00000001，把它的所有位都向左移动1位之后，得到的结果就是00000010了。需要注意的是，如果我们把x的每1位都移动8位，那么最终结果就是是这样：1 0000 0000，这样数据就有9位了，如果接受结果的数据容不下9位，那么就只会接受后面8位。左移运算和右移运算同样不会改变原有的数据。

1. #include <stdio.h>
2. int main()
3. {
4. char v1 = 1,v2;
5. for (int i = 0; i < 9; i++)
6. {
7. v2 = v1 << i;
8. printf("%d ",v2);
9. }
10. getchar();
11. return 0;
12. }

程序10-4-2

程序运行结果如下：

1 2 4 8 16 32 64 -128 0

可以看到，每往左移动1位，数值就乘2，所以，假设有一个数据x，那么x=x\*2完全可以写成x=x<<1，程序运行效率会变得更好，如果能保证数据不会溢出的话。同样的道理，右移1位相当于除2。

与预算的运算符是&，也就是取地址运算符。需要注意的是，进行与运算的时候，需要两个数据同时参与，而进行取地址运算时，只能有一个数据，编译器正是根据这点差别来区分二者的。

前面我们介绍过逻辑与运算，如果有两个条件A和B，当且仅当A和B同时为真时，A&&B才为真。位运算中的与运算与此类似，它会比较两个数的对应位，当且仅当对应位都为1时，结果才为1，否则为0。

如：00001001&10001000的值为00001000。

1. #include <stdio.h>
2. int main()
3. {
4. char v1 = 1,v2=3;
5. printf("%d ",v1&v2);
6. v2 = v2 & 0;
7. printf("%d ", v2);
8. getchar();
9. return 0;
10. }

程序10-4-3

程序运行结果如下：

1 0

v1是00000001，v2是00000011，两者进行&运算，结果自然就是00000001了。而第6行的v2=v2&0是对v2进行清0操作。虽然这样的效率实际上比起v2=0还要更低一些，因为前者是先计算出v2&0的结果（0），然后把结果赋值给v2，而后者是直接赋值。

或运算与逻辑或运算很类似，运算符是|，只要两个数的对应位中有1个为1，返回结果就为1。

假设我们要编写一个RPG游戏，人物可能处于不同的状态，如濒死、睡眠、烧伤等等。如果我们用不同的变量来分别表示这些状态，那么就需要3个变量，这样显然很浪费空间。而我们可以利用或运算的机制来实现一个变量储存多种状态。

1. #include <stdio.h>
2. #define STATE0 1
3. #define STATE1 2
4. #define STATE2 4
5. int main()
6. {
7. char state=0;
8. state = state | STATE0;
9. state = state | STATE1;
10. state = state | STATE2;
11. if ((state&STATE0) == STATE0)
12. {
13. printf("角色的状态0处于激活状态\n");
14. }
15. if ((state&STATE1) == STATE1)
16. {
17. printf("角色的状态1处于激活状态\n");
18. }
19. if ((state&STATE2) == STATE2)
20. {
21. printf("角色的状态2处于激活状态\n");
22. }
23. getchar();
24. return 0;
25. }

程序10-4-4

程序的运行结果如下：

角色的状态0处于激活状态

角色的状态1处于激活状态

角色的状态2处于激活状态

程序10-4-4中，我们预先定义了3个宏，1、2、4来分别表示3种不同的状态，它们对应的二进制数正好是00000001、00000010和00000100。

主函数中，我们声明了一个变量state用来模拟状态，初始值为0，表示没有任何状态。然后我们对state进行了3次|运算，这样state最终的值就是00000111了，最后3位分别储存着3种状态。

想要判断角色是否处于某种状态，需要用到&运算。即让state和该种状态对应的值进行&运算。00000111&00000001的结果肯定就是00000001了，而00000110&00000001的结果当然就不是，我们可以以此为依据来判断角色是否处于某种状态。

那么要怎样才能解除某种状态呢？解除状态需要用到异或运算，异或运算的运算符是^，它的作用是比较两个数的对应位，若对应位相等，则返回0，否则返回1。

即：00000001^00001001的结果是00001000。

1. #include <stdio.h>
2. #define STATE0 1
3. #define STATE1 2
4. #define STATE2 4
5. int main()
6. {
7. char state=0;
8. state = state | STATE0;
9. state = state | STATE1;
10. state = state | STATE2;
11. if ((state&STATE0) == STATE0)
12. {
13. printf("角色的状态0处于激活状态\n");
14. }
15. if ((state&STATE1) == STATE1)
16. {
17. printf("角色的状态1处于激活状态\n");
18. }
19. if ((state&STATE2) == STATE2)
20. {
21. printf("角色的状态2处于激活状态\n");
22. }
23. state = state^STATE0;
24. if ((state&STATE0) != STATE0)
25. {
26. printf("角色的状态0被解除\n");
27. }
28. getchar();
29. return 0;
30. }

程序10-4-5

程序运行结果如下：

角色的状态0处于激活状态

角色的状态1处于激活状态

角色的状态2处于激活状态

角色的状态0被解除

我们知道，当程序运行到第23行的时候，state的值是00000111，这时候让它与表示状态0的00000001进行异或运算，并将结果赋值给state，那么最终state的值就会变成00000110，也就达到了取消某种状态的效果。

异或运算可以做到一件很有意思的事情，我们可以在不使用第3个变量的情况下，交换两个相同类型的变量的值。

1. #include <stdio.h>
2. int main()
3. {
4. int num1 = 1, num2 = 2;
5. num1 = num1^num2;
6. num2 = num1^num2;
7. num1 = num1^num2;
8. printf("%d %d",num1,num2);
9. getchar();
10. return 0;
11. }

程序10-4-6

程序运行结果如下：

2 1

读者不妨用纸和笔来模拟一下这个过程。它的基本原理在于，如果a^b的结果是c，那么a^c的结果就是b，并且b^c的结果是a。

如果读者实在无法理解，可以跳过这里，毕竟位运算本身就有很大的难度。并且程序10-4-6中交换两个数的做法通用性很差，实际应用中几乎都要利用临时变量。

最后再来看看位段结构，它本质上是一个结构体。位段结构的声明方式如下：

struct 标识符

{

数据类型1 成员1:所占位数;

数据类型2 成员2:所占位数;

数据类型3 成员3:所占位数;

...

}

其中，数据类型只能为“整数型”（即int、short、char等类型，不包括float和double）。

1. #include <stdio.h>
2. typedef struct
3. {
4. unsigned char state0 : 1;
5. unsigned char state1 : 1;
6. unsigned char state2 : 1;
7. }State;
8. int main()
9. {
10. State state;
11. state.state0 = 1;
12. state.state1 = 0;
13. state.state2 = 0;
14. printf("%d ", state.state0);
15. getchar();
16. return 0;
17. }

程序10-4-7

程序运行结果如下：

1

我们定义了一个位段结构体State，它有3个成员，每个成员占用1位。显然，每个成员占用空间的大小只和冒号后面的数字有关，而和数据类型没有太大关系。那么既然如此，为什么还要指定数据类型呢？原来，冒号后面的数字是有限制的，比如，当数据类型为char的时候，该数字最多只能为8，而如果想要占用更多位，就需要考虑把char换成short或者int了。当然，换成short或者int会增大空间的占用，以程序10-4-7为例，sizeof(State)的值是1，如果我们换成int，那就是4了。

至于为什么要把数据类型指定为unsigned char而不是char，是因为位段结构的3个成员都只占用1位，如果这1位被解析成符号位的话，那我们就很尴尬了，因为没有地方可以存数据。通过这样的方式来处理各种状态似乎比用位运算还要简单一些。

位段结构体的本质是含有位段的结构体，所以它也可以有基本数据类型或者其它的复合类型作为成员。在计算大小的时候依然要满足结构体的大小是构成结构体的最大基本数据类型的大小的整数倍。而计算位段大小的方式也很简单，以程序10-4-7中的State为例，由于3个成员类型相同并且是紧邻的，所以它们会共同占用1个字节，而此时最大的数据类型是unsigned char，所以State占用1个字节。而如果在第5行和第6行之间增加一行：char data;那么State就会占用3个字节，因为此时只有state0和state1紧邻，它们共同占用1个字节，data和state3会分别占用1个字节。