HOPE

（共二册）

# C++问题解答

- 本书每章末尾的练习都做了详细的解答，主要是论述写一个程序的多样性，除了写出足够的代码外，还带有许多测试用例。本书提供的解答方法在一般书中是少见的。


北京希望电脑公司

# C++ 问 题 解 答

刘学功　魏群　皇甫广宇　　译
吴妙生　　校

北京希望电脑公司
一九九一年八月

# 序言

本书是＜＜C++程序设计语言＞＞的姊妹篇,对＜＜C++程序设计语言＞＞中每章末尾的练习都做了详细的解答,主要是论述写一个程序的多样性。除了写出足够的代码外,还带有许多测试用例。本书提供的解答方法在一般书中是少见的。

书中所有解答都可使用 AT&T cfront 编译器(1.2 和 2.0 版)进行编译,也可在使用 UNIX 系统和 MS-DOS 的一系列机器上运行。

由于在对每个习题进行解答之前都对相关内容进行了讨论,所以读者也可单独使用本书,尤其对有一定 C 语言或 C++语言基础的读者来说,阅读本解答会起到事半功倍的效果。

译者

# 目　　录

# 第一章　C＋＋概述

这本书是 Bjarne Stroustrup 所著的《C＋＋程序设计语言》的姊妹篇，后面均以 C＋＋书代替《C＋＋程序设计语言》。

为什么写这本书

当学习一门新的程序设计语言时，在能够编写自己的程序之前，必须经过广泛的学习，包括

. 阅读该语言的手册（有关 C＋＋的书），
. 学习已有的程序，
. 与使用该语言编写程序的程序员进行讨论
. 比较不同的解决方法
. 学习该语言的范例，特性，技巧和风格

假设读者正在进行上述第一阶段的过程，即正在阅读 C＋＋书。因为 C ＋＋是一门新语言，所以完成其它各项步骤很不容易——没有多少可以供仔细阅读的 C＋＋程序，而且富有经验的 C＋＋程序设计员也很少。当然，许多 C 程序和程序设计员是很得力的，而且由于 C ＋＋实际上是这种语言的高级形式，因此可以从它们之中学习到一些东西。但对于学习 C＋＋的新特性，这些是没有帮助的。

《C＋＋程序设计语言》在每章末尾都有大量的练习，这本书包括对这些练习的讨论和解答，希望读者在参考本书的答案之前，先做每个练习。如果读者做完了练习，再参照作者的解答和讨论，比较这两种答案，找出不同的原因。

读者必须明白，对于每个练习，没有绝对正确的答案。如果读者的解答与作者的思路有很大差别，读者应该仔细查看不同之处，并努力理解两种解答的工作原理。

通过做练习和参照本书中的解答，将帮助读者完成学习精妙的 C＋＋新语言的其它步骤。最终读者将很快成为一个好的 C＋＋程序设计员。

**练习和解答：**

每个练习的开始都拷贝了 C＋＋书中的练习内容，紧接着就是练习的答案。也包括对练习难点的评估（由 C＋＋书中）。标准是（＊1）练习在 5 分钟内解出，（＊2）练习在一个小时内解出，（＊3）练习一天内解出，（＊4）练习一周内解出，（＊5）练习一个月内解出。当然，读者可以根据自己已有的经验和知识改变这些期限。

每个解答只使用已经介绍过的语言特性。例如，第七章以前的解答没有使用异出类型。有时候建议使用后面章节中介绍的特性进行改进。

所有的章节和页号均对应于 C＋＋一书。——例如， 4.3.2。 在 C＋＋书末尾中的＂参考手册＂章节，以一个小写字母 r 表示——例如， r.2.4.1。由于练习多种多样，有的短，有的相对长一些，因此解答也是各种各样。短习题通常进行一点或两点说明，而长习题能提供编写现实的或近似于现实的程序片的实际经验。因此，短习题的解答包含了代码以外的讨论部分，而长习题的解答包含了更多工作代码的例子。

注意各种可选择解答的效率、可移植性和机器独立性。在适合于考虑效率的地方都给出

了有关的讨论，并对这个题目的更多内容给出了参考指示。

所有解答都已使用基于 AT&T Cfront 翻译器的编译器（1.2 和 2.0 版本）进行测试，并且在执行 UNIX 系统 V（2 和 3 版本）和 MS —DOS 的一系列机器上运行过。如果到别的系统，例如 Berkeleg UNIX 系统中执行程序，一些必要的修改都已注明。虽然几乎不可能保证测试情况包含了所有的可能，但力图保证这些代码编译和运行的正确性。

为保证在将代码转抄时不产生任何错误，程序代码是用电子设备直接从程序文本装入的，没有经过任何的手工拷贝。

### 与 C 的依赖关系

C＋＋依赖于 C 库，以获得 C 所支持的多数库函数。然而，相关的支持库也随系统的不同而不同。除非另外说明，这里列出的所有解答只使用那些在 ANSI C 标准 《ANSI 1988》和“IEEE Portable Operating System Environment 草案（也称作 POSIX） 《IEEE 1988》”所讨论的库。与其它 C 语言的主要版本和它的库之间的明显区别将在适当的地方加以讨论。

### C＋＋的发展

由于 C＋＋是一门正在发展的语言，所以在 C＋＋书出版以后又对这门语言进行了一些修改。附录 A 介绍了在本书的出版之前加到这门语言中的新特性。这些特性中的一部分是对这门语言的补充，另一部分是为使 C 语言更为有效而做的说明。除了一些细小的差别之外，现在可以认为 C＋＋就是 ANSI 标准 C 的高级形式（详见附录 A）。

尽管作者使用编译器编译比 C＋＋书所描述的更为新的语言版本，但这些新特性没有用来做练习的初始解答。在适当的地方，对这些新特性做了一些注释，或说明新特性如何影响列出的解答。有时候写出了使用新特性所作的解答。

### 附录

正如上面所提到的，附录 A 包含了一些在 C＋＋书出版之后的增加内容。附录 B 描述了一些本书中介绍和使用的标题文件，如＜Swap.h＞。

# 第二章　　说明和常量

## 2.1　习题 2.1(*1)

编写"Hellc ,world"程序（ 1.1.1）并运行。

### 2.1.1 生成你的第一个程序

第一步是生成包含这个程序的文件。许多有用的书都显示了怎样在不同的操作系统中生成、编辑和运行文件。参考表列出了市场上的一些关于 UNIX 操作系统的书。文件可以起名类似于 hello.c, hello.C, hello.cc, hello.cpp 或 hello.cxx，这依赖于正在使用的编译器所选择的惯例。

hello.c 文件应为：

```
#include <stream.h>
main()
{
        cout << "Hello, world\n";
}
```

在作者的 UNIX 操作系统下，可键入：

```
$CC  hello.c --o hello
```

CC 是调用 C＋＋编译器的命令。-o hello 表明将被编译程序存入一个名为 hello 的文件中。为运行这个程序，接着键入：

```
$hello
Hello, world
$
```

### 2.2.2 一些改进

对前面那个程序所做的改进是：在 main（）中，末尾增加一个 return 0 语句，以便该程序不会向调用它的环境返回一个自由值。与之等效的另一种方法是在程序末尾调用 exit(0)。C 语言的习惯是 0 返回值暗示成功，而一个非 0 返回值暗示着某个类型的失败。《Kernighan/Ritchie 1978》这本书中的所有程序都包含了这样的返回语句。

另一个改进是表示出 main（）调用时的参数：第一个参数是一个整数，表示调用该程序时的参考项目；第二个参数是一个指向串数组的指针，这些串是调用该程序时的参数。即使这些参数不被使用，最好也要对函数的参数进行说明。如果说明函数参数类型时没有指明参数的名字，C＋＋编译器仍须在调用的地方查对参数的类型。本书后面的所有程序都说明了 main（） 函数中的参数的类型。对于这种风格的进一步讨论，参看练习 3.17。下面显示出上面的程序加上这两点后的形式：

```
#include <stream.h>
```

```
int main(int, char**)
{
    cout << "Hello, world\n";
    return 0;
}
```

## 2.2 练习 2.2(*1)

对于在 2.1 中的每个说明，做下面几点：如果说明不是一个定义，为它写一个定义。如果说明是一个定义，为它写一个不是定义的说明。

定义：ch

此定义

```
char ch;
```

可有如下说明

```
extern char ch;
```

定义：count

此定义

```
int count=1;
```

可有如下说明

```
extern int count;
```

定义：name

此定义

```
char * name = "Bjarne";
```

可有如下说明

```
extern char  * name;
```

定义： complex

此定义

```
struct complex {float re,im;};
```

规定了这个结构的成员。在看到该定义之前，程序不可能定义 complex 类型的 变量，但它可以先使用该结构的一个说明来说明指向这个类型的指针：

```
struct complex;
complex   *x;
```

用这种方式解决了两个结构定义之间的循环依赖关系。第二个结构的说明是在第一个结构的说明之前，以便第一个结构可有一个志向第二个结构的指针：

```
struct second;
struct first {second *p2;};
struct second {first *p2;};
```

C++需要第一个说明以便知道符号 second 是一个类型规定。

定义：cvar

此定义

```
    complex cvar;
```
可有如下的一个说明
```
     extern complex cvar;
```
说明：sqrt
下面函数体的空缺
```
extern complex sqrt （comples）;
```
表示这是一个说明。这个函数的定义应为

```
#include <complex.h>
#include <math.h>
// determine sqrt of x + yi
complex sqrt(complex z)
{
    double x = real(z);     double y = imag(z);

    // the easy ones: y == 0
    if (y == 0.0)
        if (x < 0.0)
            return complex(0.0, sqrt(-x));

        else
            return complex(sqrt(x), 0.0);

    // almost as easy: x == 0
    if (x == 0.0)
        if (y < 0.0)
            {
            double x = sqrt(-y / 2);
            return complex(x, -x);
            }
        else
            {
            double x = sqrt(y / 2);
            return complex(x, x);
            }

    // convert to polar and take the root
    //
    //          2     2  1/2
```

```
    // r = ( x   + y )
    //
    // theta =     = arc tan(y / x)
    //
    //  1/2     1/2
    // z      = r    (cos   /2 + i sin   /2)
    double root_r = sqrt(sqrt(x * x + y * y));
    double half_t = atan2(y, x) / 2.0;
    return complex(root_r * cos(half_t),
                   root_r * sin(half_t));
}
```

说明： error—number

此说明

```
    extern int error—number;
```

可有一个该整数的定义，表示为

```
   int error—number;
```

定义：point

这个 typedef 定义

```
     typedef comples point;
```

没有一个单独的说明语法。通过 typedef 说明的所有类型名需要定义类型的组成部分；不可能说明一个没有这样一个定义的 typedf。

定义：real

这个函数定义

```
float real（complex *p） {retum p->re;}
```

可被说明为

```
extern float real （copmlex *9）；
```

定义：pi

这个 const 的定义 const double pi…3.1415926535897932385;

没有一个单独的说明语法。所有的常量说明都被缺省赋给静态连接，给它们下定义必须包括初始值。但如果定义变量使用

```
extern const double pi=3.1415926535897932385
```

则这个变量可以在另一个文件中使用下面的说明：

```
extern const double pi;
```

说明： user

此说明

```
struct user;
```

并没有定义结构的成员。一个完全的定义应该是

```
struct user{char*name; int uid,gid;};
```

**2.3 练习 2.3 (*1)**

写出下面各个说明：一个指向字符的指针；一个 10 整数的向量；一个 10 整数向量的起始地址；一个指向字符串向量的指针；一个指向字符指针的指针； 一个常整数；一个指向常整数的指针；以及一个指向整数的常指针。对每个说明设置初始值。

由于阅读这些描述都是从左到右，因此写说明时最好由里向外。

说明一个指向一个字符的指针

一个指向字符的指针开始就是指针形式：

```
*PC
```

然后加上它所指向的类型：

```
char *pc
```

为初始化这个指针，需用该字符的地址：

```
char c;                 //character
char *pc=&c;            //pointer to a character
```

说明一个 10 整数的向量

一个 10 整数的向量首先是向量：

```
iv[ ]
```

然后加上它的尺度：

```
iv [10]
```

最后写上该向量包含什么：

```
int iv [10]
```

为初始化该向量，需用一个元素表：

```
// vector of 10 int
int iv[10]={0,1,2,3,4,5,6,7,8,9};
```

说明一个 10 整数向量的起始地址

一个 10 整数向量的起始地址开始是地址形式：

```
&riv
```

然后加上向量：

```
&riv[]
```

然后加上尺度：

```
&riv[10]
```

最后是该向量的类型：

```
int &riv[10]
```

为初始化这个起始地址需用一个相同类型而不带起始地址的变量名：

```
int &riv[10]=iv;  //  reference to vector of 10 ints
```

说明一个指向字符串向量的指针

这个说明有两种可能的解释。一个是按上面方法处理这个向量，带一个尺寸来说明它。另一个是将向量看作一个动态的、没有边界的对象。在 C++中，处理一个无界向量要使用

一个指向该向量第一个元素的指针，而且使用 NEW 操作符将它定位。

（解释 1） 一个指向字符串向量的指针开始为指针形式：

*** psv 然后加上数组：**

**(*psv)[]**

（由于前面的规则，需要附加圆括号。）需用一个尺寸，我们选择 4。加上它的尺寸：

**(*psv)[4]**

该数组包含字符串（字符指针）：

**char*(psv)[4]**

为初始化这个数组，需用一个 4 字符串向量：

```
char *cv[4] =          //vector of 4 strings
{"aw","bx","cy","dz"};
char * (*psv)[4] = &cv; //ptr to vector of 4 strings
```

（解释 2）一个指向字符串向量的指针开始为指针形式：

***psv**

然后加上数组（事实上是一个指向该数组头元素的指针）：

****psv**

这个数组包含字符串（字符指针）：

**char ***psv**

为初始化这个数组，需用一个 4 字符串向量（指向一个字符的指针），必须使用 NEW 操作符定位，并在那里给出大小，如

```
char **ps = new char *[4];
char ***psv = &ps;    //ptr to vector of strings
```

注意，这样设置了 PSV 的初始值，但并没有给出每个指针指向的内容。这点必须这样完成：

```
ps[0] = "aw";
ps[1] = "bx";
ps[2] = "cy";
ps[s] = "dz";
```

说明一个指向字符指针的指针

一个指向字符指针的指针开始为指针形式：

```
* p p c
```

然后是另一个指针形式：

```
* * p p c
```

最后是这个字符：

```
c h a r   * * p p c;
```

为初始化这个指针需使用一个指向字符的指针的地址：

```
char *pc;                  // ptr to char
char **ppc = &pc;          // ptr to ptr to char
```

说明一个常整数

一个常整量需使用关键字 i n t 和后面的 c o n s t：

```
const int maxint = 32767;          // constant integer
```

说明一个指向常整数的指针

一个指向常整数的指针开始为指针形式：

```
* p c i
```

然后加上这个常整数：

```
const int *pci
```

可使用刚生成的常整数 m a x i n t 来初始化这个指针：

```
const int *pci=&maxint;
inter to constant int
```

说明一个指向整数的常指针

一个指向整数的常指针开始为常指针形式：

```
*const cpi
```

然后加上这个整数：

```
int *const cpi
```

为初始化这个指针，需使用一个整数的地址：

```
int i = 10;                // int
int *const cpi = &i;       // constant pointer to int
```

## §2．4 练习 2．4（*1．5）

编写一个程序，打印基本类型的指针类型的尺寸。使用操作码的尺寸。

基本类型和指针类型的尺寸

整数基本类型为 **char**, **short**, **int** 和 **long**，加上它们的版本。浮点的基本类型为 **float** 和 **double**。其中每个类型都可以有一个指向它的指针。

而且，有指向 **void** 类型的指针。这样的指针可用作一般指针，如§r．4．6所述。因为一个变量不可能为void类型，所以不可能采用void类型的尺寸。

另外，有指向函数的指针。与void类型相同，sizeof操作符不适用于函数。而且，一个函数可以有任意个返回值和参数类型。但当前多数的执行程序都有相同尺寸的指向所有函数的指针。因此在本例中只使用了一种类型：一个指向无参数的整数函数的指针。

对于每种类型，都将输出一个字符串来描述这个类型，后面是尺寸。为做这一点，生成一个函数来完成输出，传递给它一个描述类型的字符串作为第一个参数，和一个由 **sizeof** 操作符决定的类型尺寸作为第二个参数。注意，由于类型尺寸是由 **sizeof** 操作符决定的，所以被定义为 **unsigned int**。使用流输出函数完成输出功能：

```
void pr(char *type, unsigned int size)
{
    cout << "sizeof(" << type << ") = " <<
        size << "\n";
}
```

然后，为输出每个类型的尺寸，调用pr（）函数，列出如下完整的程序

```
// print the sizes of all of the fundamental types
#include <stream.h>

void pr(char *type, unsigned int size)
{
    cout << "sizeof(" << type << ") = " <<
        size << "\n";
}

int main(int, char**)
{
    pr("char",sizeof(char));
    pr("short",sizeof(short));
    pr("int",sizeof(int));
```

```
    pr("long",sizeof(long));
    pr("unsigned char",sizeof(unsigned char));
    pr("unsigned short",sizeof(unsigned short));
    pr("unsigned int",sizeof(unsigned int));
    pr("unsigned long",sizeof(unsigned long));
    pr("float",sizeof(float));      pr("double",sizeof(double));
    pr("void *",sizeof(void *));
    pr("char *",sizeof(char *));
    pr("short *",sizeof(short *));
    pr("int *",sizeof(int *));
    pr("long *",sizeof(long *));
    pr("unsigned char *",sizeof(unsigned char *));
    pr("unsigned short *",sizeof(unsigned short *));
    pr("unsigned int *",sizeof(unsigned int *));
    pr("unsigned long *",sizeof(unsigned long *));
    pr("float *",sizeof(float *));
    pr("double *",sizeof(double *));
    pr("int (*)()",sizeof(int (*)()));
    return 0;
}
```

## §2.5 练习 2.5（*1.5）

编写一个程序，打印出字母‘a’..‘z’和数字‘0’..‘9’以及它们的整数值。对其它要打印字符完成上述功能。使用十六进制完成同样的功能。

打印字母和数字的程序

打印字符的obvious解答是编写一个for循环，以‘a’开始，以‘z’结束，对每个字符使用流输出传动媒介cout。为将字符转换为非十进制的字符串，使用chr()函数：

```
for (int ch = 'a'; ch <= 'z'; ch++)
    cout << chr(ch) << " " << ch << "\n";
```

这个产生了如下完整的程序：

```
// "obvious" solution to print letters and digits
// version 1
#include <stream.h>
int main(int, char**)
{
```

```
    for (int ch = 'a'; ch <= 'z'; ch++)
        cout << chr(ch) << " " << ch << "\n";
    for (ch = '0'; ch <= '9'; ch++)
        cout << chr(ch) << " " << ch << "\n";
    return 0;
}
```

因为对每个将要输出的字符都完成相同的功能，所以可以做一个改进，可以将输出字符的代码压缩为一个单独的函数，p r （）。

```
#include <stream.h>
#include <ctype.h>
void pr(int i)
{
    cout << chr(i) << " " << i << "\n";
}
```

但遗憾的是，前面的循环只适用于使用美国标准信息代码(ASCII)字符集的计算机。使用其它字符集的计算机，如扩展二进制代码十进制互换码(EBCDIC)或 UNIVAC FIELDDATA，可能在某些字母中间混杂有其它字符。例如，EBCDIC 在 i 和 j 之间有一个空洞。打印这些字符间的值将在终端屏幕上产生出人意料的、不必要的结果。

还有一个解答是修改这个程序以便在打印每个字母前先检验之。这就需要使用 isalpha() 函数，在§ 3 . 1 . 2 , 7 8 页讲述，如下所示：

```
// print letters and digits
// version 2
// check each letter before printing
#include <stream.h>
#include <ctype.h>
void pr(int i)
{
    cout << chr(i) << " " << i << "\n";
}

int main(int, char**)
{
    for (int i = 'a'; i <= 'z'; i++)
        if (isalpha(i))
            pr(i);
```

```
    for (i = '0'; i <= '9'; i++)
        pr(i);
    return 0;
}
```

打印所有可打印字符的程序

在8位字符计算机中的ASCII字符集具有值在0至八进制0177（十六进制0X7F）之间的有效字符。但只有那些值在八进制040（十六进制0X20）和八进制0176（十六进制0X7E）之间的字符是可打印的。为输出所有可打印字符，写一个包括这个范围的for循环并调用pr()函数在产生下面的代码：

```
// print all printable characters
// version 1
#include <stream.h>
void pr(int i)  // same pr() as before
{
    cout << chr(i) << " " << i << "\n";
}

int main(int, char**)
{
    for (int i = 040; i <= 0176; i++)
        pr(i);
    return 0;
}
```

为方便起见，我们必须认为8位字符计算机中的ＥＢＣＤＩＣ字符集具有值在0到八进制０３７７（十六进制ＯＸＦＦ）之间的字符。而且，大量的空洞都被散落在可打印字符之间。

我们可以只写一个从0-0337的for循环，并在调用pr()之前使用isprint()函数检验每个字符。ANSI.C在<lim it.h>标题文件中提供一个名为CHAR_MAX的常量以定义最大的字符值。使用这个值写出这个循环为：

```
// print all printable characters portably
// version 2
#include <limits.h>
#include <stream.h>
#include <ctype.h>
```