C 源程序常见错误分析

一、C 语言出错有两种情况：

1、语法错误。指编程时违背了 C 语法的规定，对这类错误，编译程序一般都能够给出“出错信息”，并且告诉在哪一行出错及出错的类型。只要仔细检查，是可以很快发现错误并排除的。

2、逻辑错误。程序并无违背语法规则，但程序执行结果与原意不符。这是由于程序设计人员写出的源程序与设计人员的本意不相同，即出现了逻辑上的混乱。

例如：

```
unsigned char i=1;
unsigned int sum=0;
while (i<=100)
    sum=sum+i;
    i++;
```

在上例中，设计者本意是想求从 1 到 100 的整数和，但是由于循环语句中漏掉了大括号，使循环变为死循环而不是求累加。对于这种错误，C 编译通常都不会有出错信息(因为符合 C 语法，但有部分编译系统会提示有一个死循环)。对于这类逻辑错误，比语法错误更难查找，要求程序设计者有丰富的设计经验(不会有类似的错误)和有丰富的排错经验(通过仿真能够很快发现问题)。

二、初学者在编写 C 源程序时常见错误及分析

1、忘记定义变量就使用

例如：

```
main ()
{
   x=3;
   y=x;
}
```

在上式中看似正确，实际上却没有定义变量 x 和 y 的类型。C 语言规定，所有的变量必须先定义，后使用。因此在函数开头必须有定义变量 x 和 y 的语句，应改为：

```
main ()
{
   int x,y;
   x=3;
   y=x;
}
```

2、变量没有赋值初就直接使用。

例如：

```
unsigned int addition (unsigned int n)
{
   unsigned int i;
   unsigned int sum;
   for (i=0;i<n;i++)
   sum+=i;
   return (sum);
}
```

上例中本意是计算 1 到 n 之间整数的累加和，但是由于 sum 没有赋初值，sum 中的值是不确定的，因此得不到正确的结果。应改为如下：

```
unsigned int addition (unsigned int n)
{
   unsigned int i;
   unsigned int sum=0;
   for (i=0;i<n;i++)
   sum+=i;
   return (sum);
}
```

或者将 sum 定义为全局变量(全局变量在初始化时自动赋值“0”)。

```
unsigned int sum;
unsigned int addition (unsigned int n)
{
   unsigned int i;
   for (i=0;i<n;i++)
       sum+=i;
   return (sum);
}
```

3、输入输出的数据类型与所用格式说明符不一致

例如：

```
main ( )
{
   int a=3,b=4.5;
   printf("%f %d\n",a,b);
```

```
}
```

在上例中，a 与 b 变量错位，但编译时并不给出出错信息，输出结果为：

```
0.000000    16402
```

它们并不是按赋值的规则进行转换，如把 3 转换成 3.0，把 4.5 转换成 4，而是将存储单元中的数据按格式符的要求的宽度直接输出，如 b 占 4 个字节却用“%d”说明，则只有最后两个字节中的数据当成一个整数输出，a 也相同，将 a 地址前两个字节(并不属于 a)与变量 a 的两个字节当成一个 4 个字节的浮点数输出。

4、没有注意数据的数值范围

8 位单片机适用的 C 编译器，对字符型变量分配一个字节，对整型变量分配二个字节，因此有数值范围的问题。有符号的字符变量的数值范围为-128~127，有符号的整型变量的数值范围为-32768~32767。其它类型变量的范围这里就不再一一列举，请读者参见相应编译器的使用手册。

例如：

```
main ()
{
   char x;
   x=300;
}
```

在上例中，有很多读者会认为 x 的值就是 300，实际上却是错误的。

300 的二进制为 0b100101100，赋值给 x 时，将赋值最后的 8 位，高位截去，因此 x 的值实际上为 0b101100(即整数 44)。

如果将 500 赋给一个有符号的字符型变量时，变量内存储的值还会变成负数，由读者自行分析原因。

5、输入变量时忘记使用地址符号

常见是忘记使用地址符：

例如：

```
main ()
{
   int a,b;
   scanf ("%d%d",a,b);
}
```

应改为：

```
scanf ("%d%d",&a,&b);
```

6、输入时数组的组织方式与要求不符

```
scanf ("%d %d",a,b);
```

如果输入数据格式为：

```
3,4
```

则是错误的，两个数据之间应用空格分来分隔，应为：

```
3 4
```

7、误把“=”作为关系运算符“等于”

在数学和其它高级语言中，都是把“=”作为关系运算符“等于”，因此容易将程序误写为：

```
if (a=b)
 c=0;
else
    c=1;
```

在上例中，本意是如果 a 等于 b，则 c=0，否则 c=1。但 C 编译系统却认为将 b 赋值给 a，并且如果 a 不等于 0，则 c=0，当 a 等于 0，则 c=1，这与原设计的意图完全不同。应将条件表过式更改为：

```
a==b
```

8、语句后面漏加分号

C 语言规定语句末尾必须有分号，分号是 C 语句不可缺少的一部分，

例如：

```
main ()
{
   unsigned int i,sum;
   sum=0;
   for (i=0;i<10;i++)
       {sum+=i}
}
```

很多初学者认为用大括号括起就不必加分号，这是错误的，即使该语句用大括号括起来，也必须加入分号。在复合语句中，初学者往往容易漏写最后一个分号。上例应改为如下形式：

```
main ()
{
 unsigned int i,sum;
   sum=0;
   for (i=0;i<10;i++)
       {sum+=i;}
}
```

当漏写分号而出错，光标将停留在漏写分号的下一行。

9、在不该加分号的地方加了分号

```
#include "io8515v.h";
```

由于伪指令不是 C 程序语句，因此后面不能加分号。

初学者也常在判断语句的条件表达式后面加入分号，
例如：

```
main ()
{
   unsigned int i,sum;
   sum=0;
   for (i=0;i<10;i++);
   sum+=i;
}
```

在上例中，在 for 的表达式后面中入分号，则 C 编译认为循环体是一个空操作，这与设计者的本意不符。

10、对应该有花括号的复合语句，忘记加花括号
例如：

```
unsigned char i=1;
unsigned int sum=0;
while (i<=100)
      sum=sum+i;
      i++;
```

我们在前面举过这个例子，应改为：

```
unsigned char i=1;
unsigned int sum=0;
while (i<=100)
      {
      sum=sum+i;
      i++;
      }
```

11、括号不配对

当一个复合语句中使用多层括号时，常会出现这类错误；也常出现大括号不配对的现象，都是粗心所致。
例如：

```
while ((c=getchar ()!='a')
      putchar(c);
```

少了一个右括号。

12、没有注意大写字母和小写字母代表不同的标识符

例如：

```
main ()
{
int a,b,c;
a=0;
B=1;
C=a+B;
}
```

在上例中，C 编译系统会提示变量 B、C 没有定义。应改为：

```
main ()
{
int a,B,C;
a=0;
B=1;
C=a+B;
}
```

13、引用数组元素时误用圆括号

```
main ()
{
int i,a[10];
for (i=0;i<10;i++)
    scanf ("%d",a(i));
}
```

通常情况下，C 程序编译出错，但是如果恰好有一个函数 a()，则通常情况都可以通过编译，那查起错误来就更麻烦了。

14、引用数组元素超界

例如：

```
main ()
{
int i,a[5]={1,2,3,4,5};
for (i=1;i<=5;i++)
    printf ("%d",a[i]);
}
```

上例中，本意是想输出数组 a 的全部元素，实际上，定义的数组 a[5]中，只有 a[0]~a[4]，5 个元素，并不存在 a[5]。应改为如下形式：

```
main ()
{
int i;
int a[5]={1,2,3,4,5};
```

```
    for (i=0;i<5;i++)
        printf ("%d",a[i]);
    }
```

15、对二维或多维数组定义和引用的方式不对

例如：

```
    main ()
    {
      int a[5,4],
      /*   其它程序   */
    }
```

在 C 语中，对二维数组和多维数组在定义和引用时必须将每一维数组中的数据分别用方括号括起来，因此定义一个二维数组，应改为：

```
    int a[5][4];
```

16、误以为数组名代表整个数组

例如：

```
    main ()
    {
      int a[5]={1,2,3,4,5};
      printf ("%d,%d,%d,%d,%d",a);
    }
```

在上例中，本意是输出数组 a 中的全部元素，但是数组名 a 却只是代表数组的首地址，并不能代表数组中的所有元素，因此并不能得到所需的结果，应改为：

```
    main ()
    {
      int a[5]={1,2,3,4,5};
      printf ("%d,%d,%d,%d,%d",a[0],a[1],a[2],a[3],a[4]);
    }
```

17、混淆字符数组与字符指针的区别

例如：

```
    main ()
    {
      char str[10];
      str="ICCAVR";
      printf ("%s\n",str);
    }
```

在上例中，编译必定出错。因为 str[10]是一个数组，str 代表数组名，是一个常量，不能被赋值，可将 str 改为指针变量，将字符串"ICCAVR"的首地址赋值给指针变量 str，然后在 Printf 函数中输出字符串。如下：

```
    main ()
    {
```

```
    char *str;
    str="ICCAVR";
    printf ("%s\n",str);
}
```

如果坚持要使用数组，一种方式为初始化时赋值，另一种只能在程序中一个一个元素进行赋值。分别如下：

```
main ()
{
    char str[10]="ICCAVR";
    printf ("%s\n",str);
}
```

或

```
main ()
{
    char str[10];
    str[0]='I';str[1]='C';str[2]='C';str[3]='A';str[4]='V';
    str[5]='R';str[6]='\0';str[7]='\0';str[8]='\0';str[9]='\0';
    printf ("%s\n",str);
}
```

要注意：

```
char str[10]="ICCAVR";
```

和

```
char str[10];
str="ICCAVR";
```

是不相同的。

18、在引用指针变量之前没有对它赋值

```
main ()
{
    char *p;
    scanf ("%s",p);
    /*  用户程序  */
}
```

没有给指针变量赋值就使用它，由于指针变量 p 的值不确定，因此有可能误指向有用的存储空间，导致程序运行出错。应当改为：

```
main ()
{
    char *p,str[20];
    p=str;
    scanf ("%s",p);
    /*  用户程序  */
```

```
}
```

19、switch 语句的各分支漏写了 break 语句

例如：

```
switch (time)
{
   case 0 : a=0;
   case 1 : a=0;
   case 2 : a=2;
   default : a=3;
}
```

上例中，本意是根据 time 的值来决定 a 的值，但是最后程序执行的结果都一样(a=3)，因为漏写了 break 语句，程序将从相应的 case 开始顺序执行，应改为：

```
switch (time)
{
   case 0 : a=0; break;
   case 1 : a=0; break;
   case 2 : a=2; break;
   default : a=3;
}
```

20、混淆了字符和字符串的表示形式

例如：

```
char sex;
sex="M";
```

由于 sex 是字符变量，只能存放一个字符，用单引号括起来的是字符常量，才能赋值给一个字符型变量，而用双引号括起来的是字符串常量，它包括两个字符“M”和“\0”，无法存放到字符变量中。应改为：

```
char sex;
sex='M'
```

21、使用自加(++)和自减(--)时出错

例如：

```
main ()
{
   int *p,a[5]={0,1,2,3,4};
   p=a;
   printf ("%d",*p++);
}
```

在上例中，“*p++”本意是指 p 加 1，即指向第 1 个元素 a[1]外，然后输出第一个元素 a[1]的值 1。但实际上是先执行 p++，先使用 p 的原值，使用后再加 1，因此实际执行结果是输出 a[0]的值。应改为如下：

```
main ()
```

```
{
   int *p,a[5]={0,1,2,3,4};
   p=a;
   printf ("%d",*(++p));
}
```

说明：要注意区别 p++、*p++、*(++p)和(*p)++的区别，详见“2.7.4 数组的指针和指向数组的指针变量”一节的介绍。

22、所调用的函数在调用语句之后定义，但在调用之前没有说明

例如：

```
main ()
{
   float x=2.0,y=6.0,z;
   z=max(x,y);
   printf ("%f",z);
}
float max (float x,float y)
{
      return (z=x>y ? x:y)
}
```

在上例的程序中，max 函数在 main 函数之后定义，在调用之前又没有说明，因此出错。应在调用前对函数进行说明：

```
main ()
{
   float max (float x,float y);
   float x=2.0,y=6.0,z;
   z=max(x,y);
   printf ("%f",z);
}
float max (float x,float y)
{
      return (x>y ? x:y);
}
```

也可以将函数 max 在函数 main 之前定义。

23、误认为形参值的改变会影响实参的值

例如：

```
main ()
{
   int x=3,y=4;
   swap(x,y);
```

```
    printf ("%d,%d",x,y);
}
int swap (int x,int y)
{
    int z;
    z=x;x=y;y=z;
}
```

在上例中，设计者的意图本想是利用 swap 函数交换 x 和 y 的值，但是由于实参和形参之间单向传递，在函数 swap 中改变了 x 和 y 值，main 中的 x 和 y 是不会改变的。可以改为使用指针的形式，如下：

```
main ()
{
      int x=3,y=4;
      int *p1,*p2;
      p1=&x   p2=&y;
      swap(p1,p2);
      printf ("%d,%d",x,y);
}
int swap (int *p1,int *p2)
{
    int z;
    z=*p1;*p1=*p2;*p2=z;
}
```

说明：虽然函数 swap 在调用函数之后定义，而且在函数 main 调用之前又没有说明，但是由于 swap 返回值为整型，C 语言规则返回值为整形的函数在调用之前可以不必说明，因此本例中是符合 C 语法规定的。

24、函数的实参和形参类型不致。

还是使用上例：

```
main ()
{
    int x=3,y=4;
    int *p1,*p2;
    p1=&x, p2=&y;
    swap(p1,p2);
    printf ("%d,%d",x,y);
}
int swap (int p1,int p2)
{
    int z;
    z=p1;p1=p2;p2=z;
```

```
}
```

C 要求实参与形参的类型一致，一个为指向整型变量的指针，另一个为整型变量，类型不同，因此编译出错。

25、不同类型的指针混用

例如：

```
main ()
{
   char x=3,*p1;
   int *p2;
   p1=&x;
   p2=p1;
   printf ("%d,%d",x,y);
}
```

在上例中，设计者本意是想将指针 p1 所指的值赋 p2，但是由于 p1 与 p2 所指向的类型不同，不能赋值。在赋值时必须进行强制类型转换。如：

```
main ()
{
   char x=3,*p1;
   int *p2;
   p1=&x;
   p2=(int *)p1;
   printf ("%d,%d",*p1,*p2);
}
```

指向不同类型的指针变量进行强制转换后赋值，在 C 中是常用的，例如，用 malloc 函数开辟的数据存储单元，函数的返回值是一个空指针(void)，需要用强制转换成指向所需存储类型，如指向一个结构体，用以组成一个链表：

```
struct str
    {
       int a;
       struct str *next;
    }*p1;
/*    用户程序列   */
p1=(struct str *)malloc (size_t,size)
```

在 ICCAVR 中，malloc 函数返回的是 void 类型的指针，将其强制转换为 struct str 类型的结构体指针。

26、混淆数组与指针变量的区别

例如：

```
main ()
{
```

```
    int i,a[5];
    for (i=0;i<5;i++)
        scanf ("%d",a++);
}
```

在上例中，设计者意图通过 a++的操作，引用 a 数组的不同元素。由于 C 规定数组名代表数组的首地址，它的值是一个常量，因此用 a++是错误的。应改为用指针变量来实现，如：

```
main ()
{
    int i,a[5],*p;
    p=a;
    for (i=0;i<5;i++)
        scanf ("%d",p++);
}
```

或：

```
main ()
{
    int a[5],*p;
    for (p=a;p<a+5;p++)
        scanf ("%d",p);
}
```

27、混淆结构体类型与结构体变量的区别

例如：

```
struct worker
    {
      unsigned char num;
      char name[20];
    };
worker.num=1;
```

在上例中，只是说明了一种 struct worker 的结构，但是 C 编译器并没有为这种类型的结构体变量开辟存储空间，因此不能对结构体类型赋值。应用该类型定义了一个结体类型的变量后，才能对这个变量赋值，应改为：

```
struct worker
    {
      unsigned char num;
        char name[20];    };
        struct worker work;
        work.num=1;
```