ECE 264 Advanced C Programming

2009/01/21

Contents

1 Boolean Logic **1**

2 Control Flow **3**

2.1 if 3

2.2 for 4

2.3 while 5

2.4 switch-case 6

3 Common Mistakes in Flow Control **8**

3.1 Brackets and `if-else` Pairs 8

3.2 if-else 9

3.3 `default` in `switch` 11

1 Boolean Logic

C uses Boolean logic to control programs' flow. The following are commonly used logic expressions

- `a > 0` : true if a is larger than zero
- `a && b`: true if both a is true and b is true
- `a || b`: true if a is true or b is true, or both are true

- `a == b`: true if a and b have the same value. Be careful when you use this when a or b (or both) is a floating-point number. Due to limited precision, two floating-point numbers may be slightly different.

  ```
  /* precision.c */
  double val = 1e-7;
  int cnt;
  for (cnt = 0; cnt < 10000000; cnt ++)
  {
    val += 1e-7;
  }
  printf("%f %e %d\n", val, val - 1, (val == 1.0));
  ```

  The output is

  ```
      1.000000 9.975017e-08 0
  ```

  showing that val is not exactly 1.

- `a` $<=$ `b`: true if a is smaller than or equal to b
- `!  a`: true if a is false

You can use a combination of different conditions. For example

```
      if ((a > 0) && (b < c))
```

is true if `a` is greater than 0 and `b` is smaller than `c`.

When you have a complex condition, remember to use parentheses for clarity. If the expression is too complex, break it into several conditions. **Writing a clear program can help you discover mistakes more easily.**

Minimal evaluation (also called *short-circuit evaluation*):

If `a` is true in `(a || b)`, `b` is not evaluated.

If `a` is false in `(a && b)`, `b` is not evaluated.

Therefore, the order is **not** symmetric. For example

```
   if ((index < size) && (array[index] == 0))
```

is different from

```
   if ((array[index] == 0) && (index < size))
```

when `index` exceeds `size`.

2 Control Flow

2.1 if

If a computer program can do only one thing, the program isn't particularly useful. Imagine that you go to an on-line store and the store sells only one item. You cannot choose anything else. A program is more useful if it can make some decisions; for example, you decide to buy a book on C programming not a book on Java programming and you want the book to arrive sooner (and pay more for shipping).

C provides several ways to control the execution of a program. All of them require decisions based on true-false logic:

```
if (something is true)
{
    do something
}
else /* this part is optional */
{
    do something else
}
```

This "something" can be jumping to another location of the program and executing the code there. We have seen several examples using C's control

```
/* ifargc.c */
int main(int argc, char * argv[])
{
  int val1;
  int val2;
  if (argc < 3)
    {
      fprintf(stderr, "need two numbers\n");
      return -1;
    }
  val1 = (int)strtol(argv[1], (char **)NULL, 10);
  val2 = (int)strtol(argv[2], (char **)NULL, 10);
  printf("%d + %d = %d\n", val1, val2, add(val1, val2));
  return 0;
}
```

This uses an `if` condition. If the value of `argc` is smaller than 3, the program prints an error message and return -1. Otherwise, the program continues to assign the values to `val1` and `val2`.

2.2 for

Another example:

```
/* foragrc.c */
for (cnt = 0; cnt < argc; cnt ++)
{
  printf("%s\n", argv[cnt]);
}
```

This `for` block is equivalent to the following

```
/* goto.c */
cnt = 0;
repeat_label:
if (! (cnt < argc))
{
  goto done_label;
}
printf("%s\n", argv[cnt]);
cnt ++;
goto repeat_label;
done_label:
```

In general, you should avoid `goto` because too many goto's can make the program's flow hard to analyze.

```
/* for3.c */
int sum;
int cnt;
sum = 0;
for (cnt = 0; cnt < vecSize; cnt ++)
{
  sum += vec[cnt];
}
```

The compiler does not care about the format (space, tab...) but a program can be harder to read. Many text editors will indent your C programs, for example *emacs* and *eclipse*. In *eclipse*, click the right mouse button, select `Source` and `Format`, or press `Shift-Control-F`. **Proper indentation will reduce the chance of mistakes.**

```
/* badindent.c */
int sum; int cnt;
sum = 0;     for (cnt = 0; cnt <
                    vecSize;
                  cnt ++) {
       sum +=
         vec[cnt];
}
```

2.3 while

There is one important restriction of using `for`. We have to know how many iterations to execute in advance. Suppose a program needs a positive number from a user

```
/* whilescan.c */
do
{
  printf("enter a positive number: ");
  scanf("%d", & cnt);
} while (cnt <= 0);
printf("Correct! %d is positive.\n", cnt);
```

This will keep asking the user until the user enters a positive number. The following is an example of execution:

```
enter a positive number: -9
enter a positive number: -7
enter a positive number: 0
enter a positive number: 1
That's right; 1 is a positive number.
```

In C,

```
do  /* some code */  while(condition);
```

will **execute at least once** because the condition is checked after the code. We can also move the `while` condition to that top. In that case, the code may not execute at all. The following example is equivalent to `for`:

```
/* whilefor.c */

int cnt = 0;
while (cnt < vecSize)
{
  printf("%d\n", vec[cnt]);
  cnt ++;
}

/* same as */
for (cnt = 0; cnt < vecSize; cnt ++)
{
  printf("%d\n", vec[cnt]);
}
```

This example shows that `while` can implement `for`.

2.4 switch-case

Sometimes, you want to distinguish several cases. For example, a computer game has to check whether a user presses up (u), down (d), left (l), and right (r) keys. This can be done by using several `if`'s.

```
/* multiif.c */
if (key == 'u') {
  /* move up */
 }
 else {if (key == 'd')
     { /* move down */} else
     {
       if (key == 'l')
         {
           /* move left */
         } else {if (key == 'r') { /* move right */
         }
         else
           {
             /* invalid, error */
           }
       }
     }
 }
```

There is a better way to handle this situation:

```
/* switch.c */
switch (key)
{
 case ’u’:
   /* move up */
   break;
 case ’d’:
   /* move down */
   break;
 case ’l’:
   /* move left */
   break;
 case ’r’:
   /* move right */
 default:
   /* invalid, error */
}
```

Using `switch` makes the code easier to read. **It is necessary to put `break` before the next case; otherwise, the code in the next case will also be executed**. In the next example, pressing ’U’ and ’u’ executes the same code.

```
/* switch2.c */
switch (key)
{
 case ’u’: /* no break */
 case ’U’:
   /* move up */
   break;
 case ’d’:
 case ’D’:
   /* move down */
   break;
 case ’l’:
   /* move left */
   break;
 case ’r’:
   /* move right */
 default:
   /* invalid, error message */
}
```

Forgetting to add break in correct locations is a common mistake.

3 Common Mistakes in Flow Control

3.1 Brackets and `if-else` Pairs

Flow control is critical in most programs. Therefore, it is very important to write the control correctly. The following are some common mistakes and how to avoid them. In C, the following two pieces of code are equivalent

```
if (a > 0)
{
    b = -1;
}
```

and

```
if (a > 0)
   b = -1;
```

If there is only one statement controlled by `if`, it is unnecessary to use brackets. However, the **first (using brackets) is better because it prevents you from making the following common mistake**. If you add another statement later, without the bracket, you may add the statement directly and the new statement is no longer controlled by the condition.

```
if (a > 0)
   b = -1;
   c = -2;
```

is equivalent to

```
if (a > 0)
{
    b = -1;
}
c = -2; /* not controlled by a's value */
```

and is different from

```
       if (a > 0)
       {
           b = -1;
           c = -2; /* controlled by a’s value */
       }
```

Adding the brackets can reduce the chance of mistakes.

3.2 if-else

In C, `else` **corresponds to the closest** `if`.

What is the value of `z` after executing this code?

```
/* ifelse1.c */
int x = 1;
int y = 2;
int z = 3;
if (x > 10)
     if (y > 4)
        z = -1;
else
  z = -2;
```

Is `z` 3, -1, or -2? Which of the following two corresponds to the code above?

```
/* ifelse2.c */
if (x > 10)
{
  /* nothing between this bracket */
  if (y > 4)
    {
      z = -1;
    }
  else
    {
      z = -2;
    }
  /* and this bracket will be executed */
}
/* z unchanged since x > 10 is false */
```

or

```
/* ifelse3.c */
if (x > 10)
{
  if (y > 4)
    {
      z = -1;
    }
}
else
{
  z = -2; /* z is changed to -2 because x < 10 */
}
```

The answer is `z` = 3 (unchanged) because the `else` corresponds to the closest (second) `if`. This is another reason **you should add brackets to ensure that the code is exactly what you want. You should indent the code because proper indentation helps you visually find the mistakes.** This is very easy since many tools can do it for you, including *emacs*, *eclipse*, or a shell program called *indent*.

3.3 `default` in `switch`

You should always add the `default` condition at the bottom of a `switch` block. You may think that the cases have covered all possible scenarios. However, it is common that you miss one case. Adding `default` and printing an error message can help you discover the mistake early.