ECE 264 Advanced C Programming

Contents

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 */
   break;
 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 */
   break;
 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.

4 Binary Numbers

We are used to decimal numbers (base = 10) but computers use *binary numbers* (base = 2). A decimal number may have 10 different digits: 0, 1, 2, 3, ..., 9. A binary number can have only 0 or 1. The following table shows how to convert some numbers

decimal	binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

What should we for converting any decimal number to the corresponding binary number, and vice versa? Suppose $b_n b_{n-1} ... b_1 b_0$ is a binary number and b_i is the $(i+1)^{th}$ digit from the right end. The leftmost bit b_n is called the *most significant bit* (MSB) and the rightmost bit b_0 is called the *least significant bit* (LSB). The corresponding decimal number is $b_0 + 2(b_1 + 2(b_2 + 2(b_3 ... + b_n) = b_o + 2b_1 + 4b_2 + 8b_3 ... + 2^n b_n$

$$\sum_{i=0}^{n} 2^i b_i. \tag{1}$$

One important property of binary numbers (or any number using any base) is

$$\sum_{i=0}^{n-1} 2^i b_i < 2^n. \tag{2}$$

To convert a decimal number $d > 0$ into a binary number, we first find n such that $2^{n-1} < d \leq 2^n$. This makes $b_n = 1$. Then, we find the binary number for $d - 2^n$. The procedure continues until we reach zero.

How do we add two binary numbers? It is the same as adding two decimal numbers, except carries occur for 2, not 10.

0	1	1	0	1	13
0	0	1	1	1	7
1	0	1	0	0	20

How do we handle subtractions? We first *complement* each bit, i.e. change 1 to 0 and 0 to 1. This new number is called *one's complement*. Then, we add 1 to the number and create *two's complement*. For example, -7 is represented as

original	0	0	1	1	1
one's complement	1	1	0	0	0
two's complement	1	1	0	0	1

Binary subtraction is performed by adding the two's complement and ignore the carry (if occurs).

0	1	1	0	1	13
1	1	0	0	1	-7
0	0	1	1	0	6

Many applications require setting and testing only some bits, not the whole integer. For example, there are four LEDs and each can be on (1) or off (0). To control these LEDs, we need only four bits and one `char` is sufficient. We do `not` need to use four `int` because that would waste too much storage space. If we want to turn off all LEDs, we set the value to 0000_b, namely zero. If we want to turn on all LEDs, we set the value to 1111_b, or 15_d. If we want to turn on the center two LEDs, the value is 0110_b or 6_d.

Another example is representing colors. We usually use RBG for red, blue, and green and use 8 bits for each. One single color needs 24 bits and one 32-bit `int` is sufficient; 11111111_b means the brightest of the color.

Red	Green	Blue	Color
1111 1111	1111 1111	1111 1111 $_b$	white
1111 1111	0000 0000	0000 0000 $_b$	(brightest) red
0000 0000	1111 1111	0000 0000 $_b$	(brightest) green
1111 1111	1111 1111	0000 0000 $_b$	(brightest) yellow
0000 1111	0000 1111	0000 1111 $_b$	gray
0000 0011	0000 0011	0000 0011 $_b$	(darker) gray

We can combine 4 binary bits into 1 *hexademical* digit:

binary	hexadecimal	decimal
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

In C, a hexadecimal number has prefix "0X": 0XA, 0XB ... We can rewrite the brightest red as 0XFF0000 and the brightest blue as 0X0000FF.

5 Bitwise Operations

5.1 And / Or

Many applications require modifying individual bits. For example, suppose we want to brighten the blue components in an image, we should increase only the bits belonging to blue, without changing the red nor the blue components. In this case, we need to use a *mask* to isolate the blue components.

```
int color = ... /* the color of a pixel */
int blueComponent = color & 0X0000FF; /* bitwise AND */
blueComponent *= 2; /* twice brighter */
if (blueComponent > 0XFF) /* saturate */
{
   blueComponent = 0XFF;
}
color = color | blueComponent; /* bitwise OR */
```

5.2 Shift

Now let's go back to our LEDs. Suppose the rightmost LED is turned on right now (0001_b) and we want to move the on LED left by one (0010_b). We can multiple the number by 2_d or *shift* it left by one bit. In fact, multiplying by 2_d is **equivalent** to shifting left by one bit. Multiplying a number by 2_d^n is equivalent to shifting the number by n bits. There may be, however, performance difference. Shifting left can be much faster than multiplication (dependent on the hardware design).

```
char LEDs = 0x1;
LEDs <<= 1;
/* LEDs is 0x2 */
```

We can also shift right by using $>>$.

Suppose we want to convert an RGB image into a gray-level (0 - 255) image of the red component only. For each pixel, we can find the gray level by using

```
int color .... /* RGB of a pixel */
int redColor = color & 0XFF0000;
unsigned char grayLevel = redColor >> 16;
```

5.3 OR and XOR

We have seen bitwise AND (&). There are two other types of bitwise operations: OR (|) and exclusive or, XOR ($\wedge$).

a	b	a & b	a | b	a ∧ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The rules are

- a & b is true (1) if both a and b are true (1).
- a | b is true (1) if one or both a and b are true (1).
- a ∧ b is true (1) if a and b are different.

Bitwise operations may not seem important or necessary in applications where storage is abundant. However, when you handle large amounts of data, such as images and videos, saving a few bits per pixel can lead to significant savings in cost and execution time. In some embedded systems, memory is limited and you need to be careful how to use storage. If you are designing a spacecraft, adding another memory chip can make the system too heavy to reach space.

6 Operator Precedence

What is the value of

```
3 + 2 * 4
```

Is it

```
(3 + 2) * 4 = 20
```

or

```
3 + (2 * 4) = 11?
```

Multiplication and division have higher *precedence* than addition and subtraction. Therefore, the second is the correct answer. In C, parentheses have the highest precedence so you can always use parentheses to change or to emphasize the precedence you want. On page 84 on ABoC, you can find a table showing the precedence of operators. **In general, use simple expressions to avoid confusion.** Break a complication formula into several simpler ones to ensure correct order of computation. Use parentheses when you are not sure about the precedence rules in C.

What is the output of this program?

```
/* precedence1.c */
#include <stdio.h>
int main(int argc, char * argv[])
{
  int a = 1;
  int b = 2;
  int c = 3;
  b = c + a++;
  printf("%d %d %d\n", a, b, c);
  c = ++a + b;
  printf("%d %d %d\n", a, b, c);
  a = -c + b;
  printf("%d %d %d\n", a, b, c);
  return 0;
}

/*
  2 4 3
  3 4 7
  -3 4 7
*/
```

The value of `b` should be 3 + 1 = 4, right? We know `a ++` increments `a` by one. The question is whether this increment occurs before or after we assign the value to `b`. If it is before, `b` should 5. If it is after, `b` should be 4. It turns out `a ++` increments after and `++ a` increments before the assignment. Thus, the next line has `c = 3 + 4 = 7`. Negation `-c` occurs before addition so `a = -7 + 4 = -3`.

Now, we know how precedence affects the results. How about the next program?

```
/* precedence2.c */
#include <stdio.h>
int main(int argc, char * argv[])
{
```

```
  int a = 1;
  int b = 2;
  int c = 3;
  int d = 4;
  int e = 5;
  int x = e ++ % -- d * c / b - a;
  int y = e ^ -- d * ++ c - b % a;
  printf("%d %d\n", x, y);
  return 0;
}

/*
  2 14
*/
```

The answers are 2 and 14. How do we get the answers? Honestly, I would not even try to analyze the expressions. Whoever wrote the program does not know how to write programs. **It is important to write clear programs.** The statements are too complicated for human to read and to debug. Some people believe that writing mysterious programs enhance job security, "If nobody understands your program, you can't be fired." This is completely wrong. If you talk to managers, they will tell you that they would fire people before their mysterious code is integrated into the overall system. Mysterious code will make you fired sooner.