Introduction To C Programming

Lesson 03

Control Structures, Preprocessor

Objectives

- Understand C control structures
- Understand simple preprocessor directives

if

- Used for conditional execution

```
if (expression) statement;
if (expression)
    statement;
if (score > 5000)
    printf(“you win!\n”);
```

- The expression is evaluated for truth. If it is true then the statement is executed.

if

- C has a liberal interpretation of truth. Any non zero value is ‘true’. Only 0 is considered ‘false’.
- So 1, 500, -23 and 32000 are all true
- It is permissible to have the true statement be an entire code block:

```
if (expression)
{
     statement;
     statement;
}
```

if

```
if (x == 3)
    printf (“Isn’t that special?\n”);

if (y % 2 == 0 && x != 44)
{
    printf (“Give your name: “);
    scanf (“%s”, yourName);
}
```

if

```
if (a + b)
{
    printf (“%d + %d isn’t zero”
      ”\nGive a and b: “, a, b);
    scanf (“%d %d”, &a, &b);
}
if (bad_data(x, y, z) == FALSE)
    if (x + y + z > 0)
        printf (“Sum is positive\n”);
```

if

```
if (score >= 90)
    printf (“You get an ‘A’!\n”);
    printf (“Congratulations!\n”);
```

- Second statement is executed regardless of the result of the expression in the ‘if’ statement

```
if (x = 5)
    printf (“x is positive\n”);
```

if

- if (expression)

```
    statement;
else
    statement;
```

```
if (score > 5000)
    rating = EXPERT;
else
    rating = NOVICE;
```

if

```
if (expression)
{
        statement1;
        statement2;
}
else
{
        statement3;
        statement4;
}
```

if

```
if (user == NEW) {
    protect_system();
    welcome();
    disp_help();
}
else {
    user = EXPERT;
    bullet_proof_system();
    expert_menu();
}
```

if else if Ladder

```
if (expression)
    statement;
else if (expression)
    statement;
. . .
else if (expressionN)
    statementN;
else
    default_statement;
```

if

```
if (--i)
    if (j + k < 12)
        printf(“Hello there!\n”);
    else
        printf(“Goodbye there!\n”);
else
    printf(“This is a test!\n”);
```

if

```
if (x >= 0)
    if (x < 100)
        printf(“x is within range\n”);
else
    printf(“x is negative\n”);
```

- The else does *not* match up with the first ‘if’!

while

- The while loop is an entry condition loop.

```
while (expression) statement;
```

- It is often written as:

```
while (expression)
    statement;
```

- Or as:

```
while (expression) {
    statement1;
    statement2;
}
```

while

```
x = 0;
while (x++ < 100)
    printf(“C is my favorite\n”);

int x = 0;
while (x < 100) {
    printf(“C is my favorite\n”);
    ++x;
}
```

while

- Be careful of the accidental infinite loop
- A deliberate infinite loop may be created:

```
while (1)
{
    /* body of loop */
}

#define FOREVER      1
while (FOREVER)
{ ...
```

while

```
x = 10;
while (x--)
    do_something();
x = 100;
while (x--)
    ;
```

- Be careful of:

```
x = -10;
while (x--);
```

for

- The for loop provides all three parts of a normal loop - 1) initialization, 2) test, 3) increment

```
    for (expr1; expr2; expr3)
        statement;
```

```
x = 0;
while (x < 100) {
     doSomething();
     ++x;
}
```

for

```
for (x = 0; x < 100; ++x)
    doSomething();
```

- Infinite loop:

```
for ( ; ; )
    doSomething();
```

Comma

- You can use the comma operator to extend the for loop's capability
- The comma operator separates expressions, causing them to be executed in order, left to right

```
x = 1, y = 2, z = 3;
```

for

```
for (x = 0, y = 100; x < 100; ++x, --y)
    printf(“x  = %d, y = %d\n”, x, y);
```

- Notice that in the initialization step, both x and y are initialized. Also that in the update section, x is incremented and y is decremented

do while

- In addition to ‘while’ and ‘for’ there is also ‘do until’. Format:

```
do
{
     /* body of loop */
} while (expression);
```

- Use when the loop is indefinite but must be executed at least once

do while

```
display_3_choice_menu();
do
{
    ch = get_users_choice();
}
while (ch != ‘A’ && ch != ‘B’ && ch != ‘C’);
```

switch

- The ‘switch’ is used to select among a group of possibilities. Format:

```
switch (expression)
{
case constant_1: statement_1;
    break;
case constant_2: statement_2;
    break;
default: statement_n;
    break;
}
```

switch

- The constant values on the case statements must be resolvable at compile time
- When execution at a case label begins, it continues until it “falls out” of the switch, or a “break” is executed.
- If a “break” is executed, the switch is terminated

switch

- The default label is executed if none of the case constants match the switch expression. It is not required. (It need not be at the bottom - it can be placed anywhere).
- If no case statement matches the expression, the switch is terminated and nothing happens
- All case labels must be unique. Only integer constants can be used

switch

- Multiple case constants that execute the same statements can be combined.

```
case ‘0’: case ‘1’: case ‘2’: case ‘3’:
case ‘4’: case ‘5’: case ‘6’: case ‘7’:
case ‘8’: case ‘9’:
    statement1; break;
```

- Multiple constants can’t be put in one case label. This is illegal:

```
    case ‘0’ - ‘9’:
```

break and continue

- There are two keywords that are essentially disguised “goto”s - ‘break’ and ‘continue’
- Unlike the real goto, these are considered acceptable and can be used without guilt
- In a switch, break is sort of a ‘goto’ to the end of the construct
- break can be used in any of the C loops; when executed the loop terminates and execution continues after the loop

break

```
while ((c = getchar()) != EOF) {
    if (c == ‘\n’)
        break;
    putchar(ch);
}
```

*/ * can be better written as: */*

```
while ((c = getchar()) != EOF && c !=
  ‘\n’) {
    putchar(ch);
}
```

continue

- The ‘continue’ can be used inside ‘while’, ‘for’ or ‘do while’, but not ‘switch’
- When continue is executed, all execution stops, remaining statements in the loop are skipped and the next iteration of the loop begins

```
while ((c = getchar()) != EOF) {
   if (c == ‘ ‘ || c == ‘\t’ || c == ‘\n’)
      continue;
   ++count;
}
```

Better:

```
while ((c = getchar()) != EOF) {
    if (c != ‘ ‘ && c != ‘\t’ && c != ‘\n’)
        ++count;
}
```

goto

- C does have a ‘goto’ statement. It’s format is:

```
goto label;
. . .
label:
```

- goto has a bad reputation because, as Kernighan and Richie state, it is “infinitely abusable”. About the only time it can be justified is in error control from deep within many nested loops. <u>*It should be avoided!*</u>

exit()

- A C program normally terminates after the last statement in main(). But you can terminate the program at any point by calling ‘exit()’. Format:

  ```
  void exit(int status);
  ```

- ‘status’ is usually an exit code indicating the state of the program when exit was called. 0 usually means it terminated normally
- It is better to use return() from main() because exit() has a different effect in C++

Preprocessor

- The preprocessor processes C source code before compilation. Preprocessor statements are referred to as preprocessor “directives”
- Preprocessor directives are not C statements
- All preprocessor directives begin with a ‘#’

#define

- Use to define symbol constants and macros
  - Macros are covered in intermediate C

```
#define    RATE    0.00534
```

#include

- Use to locate a C file and load it in as part of the program.

```
#include <stdio.h>
#include “myfile.h”
```

- Use the first method to load in a ‘system’ file, and the second method to load in a file that’s part of your program
- The “.h” extension is usually used to indicate a C ‘header’ file

#if, #ifdef, ...

- Directives used for indicating conditional compilation situations

```
#if MYFILE_H
. . .
#endif

#define MYFILE_H
#ifdef MYFILE_H
. . .
#endif
```

#if, . . .

- How would you comment out a bunch of code that has a lot of comments in it (remember that you can't nest comments)?

#if, . . .

```
#if 0
    statement1; /* comment */
    statement2; /* comment */
    statement3; /* comment */
#endif
```

#if, . . .

```
File foo.h              File bar.h
#ifndef FOO_H           #include “foo.h”
#define FOO_H           . . .
. . .
#endif

File foobar.c
#include “foo.h”
#include “bar.h”
```