Control and Control Flow

Overview
✓ Begin with high level view of program execution
✓ Introduce concept of control flow
✓ Identify kinds of control flow
✓ Examine how to control movement through program
✓ Present some examples of good coding style

Program Execution
Program execution
Term used to describe how program’s instructions executed
Simplistic high-level view suggests
One instruction executed after another
Until done

While correct in one sense
In practice process is a bit more complex

Let’s examine word how from above in greater detail

How comprises
✓ Order in which instructions executed
✓ Which instructions executed
✓ How often instructions are executed

Will learn that effectively managing how
Can have significant impact on quality of our program

On examination we find
Four basic ways program may execute
• Sequentially
• Branch
• Loop
• Function call

Will examine each in greater detail starting with sequential execution

Sequential Execution
Sequential execution
Means one statement executed following another
In sequence
Each statement in program
Evaluated before proceeding to next

Statement may be
• Simple
  Single expression
  Written as
  expression;

  Each expression terminated with ;

  Illustrated in following code fragments
  
a = b;
a = sqrt (c);

• Compound
  Set of simple expressions
  Enclosed in {}
  Not terminated with ;
  {
    statement0;
    statement1;
    ...
    Statementn-1;
  }

  Compound expression can be used
  Anywhere simple expression used

  Braces – called curly braces – specify
  Group of enclosed expressions
  Treated as a block or set

*Flow of control* for sequential execution
  Proceeds in order from first statement to last
  Top to bottom of block or program
Conditional – Branch Execution

Branch or Conditional Branch execution
Means based upon specified criteria
One set of statements executed
Instead of others

Flow of control for conditional branch execution
Illustrated in accompanying diagram

Simple Branch – if
Basic form of branch is if construct
if

```plaintext
syntax
    if (expression) statement;
```

expression is evaluated
If it is non zero
    statement is evaluated
Otherwise
    Evaluation of statement is skipped

Note:
Evaluation criterion does not say if expression is TRUE
    It says expression is non-zero

Non-zero means that if expression has
Any positive or negative value other than 0
The evaluation of expression succeeds and statement is evaluated

Observe:
If statement is compound
    Must enclose in {} – curly brackets

Example
Consider following code fragment

```plaintext
int x = 2;
if (x == 3)
    printf ("Success !!!\n");
    printf ("Time To Go Home \n");
```
Will print
Time To Go Home

If this is intended – ok
Indentation suggests not

If not intended
Must write
if (x==3)
{
printf ("Success !!!\n");
printf ("Time To Go Home \n");
}

**Good Style**
Always use { }  
Whether *statement* is simple or compound

**Good Style**
Write
if (3==x)
Instead of
if (x==3)

Expression
if (x==3)

Syntactically correct
However
If expression unintentionally gets written as
if (x=3)

• x is assigned value 3
• Will always evaluate to non-zero

Preferred form
if (3==x)

Since 3 is an rvalue – can only appear on right hand side of assignment
If the following appears

if (3=x)

Compiler will complain

2 Way Branch
Let’s now look at a 2 way branch

```
syntax
if (expression)
    statement_0
else
    statement_1
```

At first doesn’t seem to offer anything beyond basic `if`

Let’s see…
Are these the same…
```
if (x > 0)
    printf ("Big \n");
printf ("Small\n");
```

```
if (x > 0)
    printf ("Big \n");
else
    printf ("Small \n");
```

What if x has value 3

What prints…

Now let `statement_1` in the else clause be of form

1. if (expression)
   statement_a

or
These provide different results

Consider the constructs

1. if (expression<sub>0</sub>)
   statement<sub>0</sub>
   else if (expression<sub>1</sub>)
   statement<sub>1</sub>

   and

2. if (expression<sub>0</sub>)
   statement<sub>0</sub>
   else if (expression<sub>1</sub>)
   statement<sub>1</sub>
   else
   statement<sub>2</sub>

• For case 1
  Possible statement<sub>0</sub> and statement<sub>1</sub>
  Not executed
  Depends upon values of
  expression<sub>0</sub>
  expression<sub>1</sub>

• For case 2
  statement<sub>2</sub>
  Represents the default case

  Can easily eliminate if
  No explicit default

  May use for error checking
  Covers the impossible case
Examine the following bit of code
Does the code fragment print what is desired

    if (x > 5)
        if (x > 10)
            printf ("x is really Big \n");
        else
            printf ("x is just Big\n");

Indentation suggests
    else goes with the outer if

Compiler associates else
    With inner if
    Associates with most recent if

Very difficult to find in complex program
    Best to use {}
    To specify and control the desired interpretation

**Good Style**
    Indentation helps
    To make code more readable
    To simplify debugging
    Indentation *does not*
    Modify flow of control

**Multiway Branch**
    Can make multiway decision block
    Cascade series of
        *if else* statements

    Control or decision variable
        *Not* required to be same for each decision
Example
if (expression_0)
  statement_0
else if (expression_1)
  statement_1
else if (expression_2)
  statement_2
else if (expression_3)
  statement_3
else if (expression_4)
  statement_4
else
  statement_5

When multiway branch
Based upon different values for same variable can use switch statement
Similar to case statement
In other languages

Switch
Let’s now introduce and examine the switch statement

```
syntax
switch (exp) statement
```

Statement or switch body is expression of the form:

```
{  # Body enclose in curly brackets
  case label: statement;
  .
  .
  default-label: default;
}
```
Control expression exp
    Must be integral type
Case labels
    Must be integral type constant expression
    • 1, 2, 3
    • String that evaluates to a constant
    • enum
default
    Default label
body
    Simple or compound statement
    Enclosed in curly brackets

Execution
1. Control expression evaluated
2. If value of control expression equal to some case label
    Control flow transferred to point indicated by label
    Execution proceeds from that point.
3. If value not equal to any label and default label present
    Control transferred to default
4. If value not equal to any label and default label not present
    Control transferred to statement after switch

Example
multiwayBranch0.c

Consider following code fragment
#include <stdio.h>

int main(void)
{
    int value;
    printf("Enter number: ");
    scanf("%i", &value);
    switch (value)
    {
        case 1:
            printf("*");
        case 2:
            printf("**");
        case 3:
            printf("***");
    }
case 4:
    printf ("****");
default:
    printf ("done\n");
}
return 0;
}

If value set to 3
Control transferred to case 3 label
Execution proceeds from that point

7 stars will be printed followed by word done

Probably not what was intended

Now look at modified example
Example
multiwayBranch1.c

#include <stdio.h>
int main()
{
    int value;

    printf ("Enter number: ");
    scanf ("%i", &value);

    switch (value)
    {
    case 1:
        printf ("*");
        break;
    case 2:
        printf ("**");
        break;
    case 3:
        printf ("***");
        break;
    case 4:
        printf ("****");
        break;
default:
    printf ("done\n");
}

printf("\n");
return 0;
}

In this example
3 stars will be printed
Difference is
break statement

Statement evaluation proceeds as follows
1. Control expression evaluated
2. If value of control expression equal to some case label
   Control flow transferred to point indicated by label

break
Causes execution of smallest enclosing switch statement to be terminated
   Typically end of switch marked by closing right parenthesis

After break
Control transferred to point just beyond terminated statement
To right side of right parenthesis

If the value 5 is entered
  No stars printed ; no default
     then
  done printed ; default

Case labels do not have to be in numerical order

Note:
Evaluation of control expression stated
   If value of control expression equal to some case label
   Control flow transferred to point indicated by label
Means execution proceeds from that point forward
   Not only code associated with label
Must use break to force exit of case body
Example
multiwayBranch2.c

User is prompted for value

Possible input values used as
case labels

#include <stdio.h>

int main()
{
    int value;

    printf("Enter number: ");
    scanf("%i", &value);

    switch (value)
    {
        case 1: case 3: case 5: case 7:
            printf("The number is prime\n");
            break;
        default:
            printf("The number is not prime\n");
    }

    return 0;
}

Inputs 1, 3, 5, or 7

Print
    The number is prime

All others
    Fall through to default case
**Loops and Iterations Execution**

*Loop or Iteration execution*

Means

- One statement or set of statements
  - Repeatedly executed
  - Based upon specified criteria

Number of possible iteration types

- Single
- Multiple
- Infinite

*Flow of control* for loop or iteration execution

Illustrated in accompanying diagram

Can make decision to loop

- Before
- After

Executing the block of statements

C has three statements to support

Loops and Iterations

- `while`
- `do-while`
- `for`

**while**

`while` is the simplest loop construct

**Syntax**

```plaintext
while (expression) statement
```

**Execution:**

1. Evaluate `control expression`
2. If `exp != zero`
   - Evaluate `statement`
3. Repeat process
4. Execution complete
   - When `expression` evaluates to zero

**Note:**

Evaluation criterion *does not* say if `expression` is TRUE
It says *expression* is not zero
  Control can be transferred out using
    break
    return
    goto

*Note:*
This *does not* say should be

**Good Style**
Want only single entry or exit point from loops or iterations
  Work with *control expression* rather than asynchronous exit

*Observe:*
  With *while* construct
    If *control expression* is initially zero
      *Statement* is never executed

    If *control expression* is always non-zero
      Have infinite loop

**do - while**
Similar to simple while

  **syntax**
  do statement while (expression)

Execution:
1. Evaluate *statement*
2. Evaluate *control expression*
3. If exp != zero
   Repeat process
4. Execution complete
   When *expression* evaluates to zero
   Control can be transferred out using
     break
     return
     goto
Observe:
With *do - while* construct
Statement always executed once
If *control expression* is always non-zero
Have infinite loop

for statement
Considerably more general either form of while
*while, do - while*
Can be implemented using *for*

**Syntax**
```
for (exp_0; exp_1; exp_2)
{
  statement(s)
}
```

Expressions exp_0, exp_1, exp_2
all optional

exp_0
Initialize loop variable(s)
exp_1
Test loop variable
  if not zero
    Continue
  else
    Terminate
exp_2
Update loop variable

Execution:
1. Evaluate exp_0 if present
2. Evaluate exp_1 if present
If zero
    Execution of complete
If not zero or not present
    Continue execution
3. Execute the body
4. Evaluate exp_2 if present

Control transferred out of the loop
1. exp_1 evaluates to zero
2. break, return, goto

**Good Style**
Want only single entry or exit point from loops or iterations

Work with *control expression* rather than asynchronous exit

**Observe:**
If exp_0, exp_1, exp_2 not present
Have infinite loop

**Example**

```
simpleLoop0.c

#include <stdio.h>

int main(void)
{
    int i = 0; // loop variable
    char myChar = 'a'; // something to display

    for (i = 0; i < 5; i++)
    {
        // Write to display
        printf("The character is, ta da: %c\n", myChar);
    }

    return 0;
}
```

- 16 of 23 -
Loop iterates 5 times
1. Prints character
2. Increments character
3. Repeats

Example

simpleLoop1.c
#include <stdio.h>

int main()
{
    int value, result = 2;
    for (;;)
    {
        // Write to display
        printf("Enter a character: ");

        // Read the keyboard
        value = getchar();
        value = value - '0';

        // Do some calculations and display result
        getchar();
        result = value + 4;
        printf("The calculations show: %i\n", result);
        result = 0;
    }
    return 0;
}

Loop iterates forever
1. Writes to the display – prompts user for input
2. Read keyboard input
3. Does a calculation
4. Displays the result
5. Repeats
   ✓ Why is 2nd getchar() used
   ✓ What does line: value = value - '0'; do
break – continue Execution

*break and continue execution*

Used to alter flow of control

Inside loops

switch statement

break only

break

1. Causes execution of smallest enclosing
   while, do, for, switch
   to be terminated
2. Resume execution at point immediately beyond terminated statement

continue

1. Causes execution of smallest
   while, do, for
   to be terminated
2. Resume execution at end of enclosing body

while (expression)
{
    statement
    continue or break
    :
    :
    statement
}

do
{
    statement
    continue or break
    :
    :
    statement
}
while (expression)
for (e_0; e_1; e_2)
{
    statement
    continue or break
    :
    :
    statement
}

Example
simpleLoop2.c

Program
Loops forever
Accepts string of characters
Prints the first 4
breaks out of for loop on
EOF, CR
breaks out of while
EOF
continues through input string – skips
if substring > 4 characters

#include <stdio.h>
int main()
{
    int i, value;
    while (1)
    {
        printf ("Enter a word: ");

        for (i=0; i++)
        {
            value = getchar();

            /*
             * Test for EOF or new line
             * EOF will terminate
             * newline will ask for new word
             */
}

if ((EOF == value) || ('\n' == value))
    break; // not a good choice

/*
* if the input string has more than 4 characters discard by not printing
*/

else if (i >= 4)
    continue; // not a good choice

    printf ("%c", value);
}

printf ("\n");
if (EOF == value)
{
    printf ("Done\n");
    break;
}

} return 0;

Questions:
Are the break and continue statements necessary
Could program be written without them

**return – goto Execution**

*return and goto execution*
Means of radically and asynchronously altering flow of control
Within a program

**return**
Used to terminate current function
Perhaps return value
Control resumes in calling function

```
syntax
    return expression
```

*expression is optional*
goto

Used to transfer control to any statement within a function

```plaintext
syntax
    goto label;

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

Use of goto not recommended
- Confusing
- Leads to errors
- Difficult to debug
- Unstructured

Rules if used
1. Don’t jump into else portion of if - else construct from if or from outside.
2. Don’t jump into if portion of if - else construct from else or from outside.
3. Don’t jump into body of switch from the outside.
4. Don’t jump into a compound statement from the outside.

If absolutely necessary
- Use break, continue, return
- Instead of goto whenever possible

**Function Execution**

*Function execution*

Means of executing set of closely related expressions
- Collected to perform
  - Single
  - Limited number of tasks … 2 - 3
- On data
  - One statement or set of statements
    - Repeatedly executed

How do functions and main program relate

Simple view
- Program stored in consecutive addresses in memory
  - 3000, 3001, 3002.....
Memory addressed by Program counter
Contains the address of the next instruction to be executed

Let functions be stored
myFunct0() be at 4000
myFunct1() be at 5000

Let call to myFunct0() occur at 3200
✓ Program counter saved on the stack
   Value 3201 on stack
   Want the next address

✓ 4000 put into PC
   Next instruction to be executed will be at 4000

If at 4050 call myFunct1()
   Process repeated

To return
4051 put into PC

More execution
3201 put into PC
Back where started

Caution:
Stack is finite size
Push executed too many times
   Stack overflow
   Loose addresses

Flow of control for function
a. Return to original calling routine
b. Switch to another function
c. Invoke itself
d. Exit with error
Summary

In this lesson we

✓ Began with high level view of program execution
✓ Introduced concept of control flow
✓ Identified kinds of control flow
✓ Examined how to control movement through program
✓ Presented some examples of good coding style
✓ Should be comfortable major means of managing flow of control in a C program