COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 4 - Basic Constructs in C

Zheng Zheng

McMaster University

Winter 2023

(Loosely) Adapted from C: How to Program 8th ed., Deitel & Deitel

A Simple Sample

```c
// A REALLY simple program in C
#include <stdio.h>

// the 'main' function begins program execution
int main(void) {
   printf("Hello World!\n");
} // end function main
```

Comment Your Code! (Or Else!)

- In Python, # designates a single-line comment.
  - In C, // is used.
- C also has Multi-line comments!

```
1 /* This is a multi-line comment!
2  *
3  *
4  * Yup, still going...
5  *
6  */
```

- /* begins a multi-line comment.
- */ ends the comment.

Comment Your Code! (Or Else!) (cont.)

Here are some guidelines for commenting code in C:

- At the top of the file, indicate:
  - The author
  - The date the program was created
  - The date the program was last modified
  - The purpose of the program
- Comment each function to indciate what its purpose is. This includes any assumptions about the inputs, properties of the outputs and invariants that will hold throughout execution.
- Comment the end of each function with something like "end of function x". This will make it much easier to navigate your code. `But do not over comment.`

Preprocessor Directives

- In Python, we access libraries using `import ...`
- In C, we use `#include<...>`
  - Lines beginning with # are **preprocessor directives**
  - Preprocessor directives are processed before the program is parsed.
  - ***.h** files are known as **header files**. Many of C's most important libraries are stored in header files.
  - `stdio.h` contains the definition for `printf` (and much more besides!)
    - C standard library
  - Adding the line `#include<stdio.h>` to the beginning of each C program should become reflexive!

Being a Blockhead

- In Python, statement blocks are indicated using indentation.

```
1 def max2(x,y) :
2   if (x > y) :
3     return x
4   else :
5     return y
```

- In C, statement blocks are indicated using { and }

```
1 int max2(int x, int y) {
2   if (x > y) {
3     return x;
4   } else {
5     return y;
6   }
7 }
```

- In addition, *all C statements are semicolon terminated;*

Whitespace Doesn't Matter!

This C program...

```
#include<stdio.h>

int main (void) {
  int x = 17;
  bool y = False;
  if (y == False) {
    return x;
  } else {
    return -1;
  }
}
```

Whitespace Doesn't Matter! (cont.)

Is *identical* to this C program...

```
#include<stdio.h>
int main (void) { /* Midline comments! */ int x = 17;
    bool y
= False;

          if (y ==          False){return x;
} else { return -1;}}
```

At least as far as the compiler is concerned!
(Clearly, one of these is preferrable...)

The `main` Event

- In Python, execution begins at the first line of the script, and terminates on the last line.
- In C, execution begins at the first line of the `main` function(!), and terminates either when execution reaches a `return` statement inside of `main`, or when the program reaches the last line of `main`.
- A `main` function is required for compilation
- Trying to put regular statements in the global namespace will result in *Syntax Errors A'Plenty!*

We'll talk about other functions in the next few weeks.

The `main` Event (cont.)

```
1 ...
2 int main (void) {
3     ...
4 }
5 ...
```

- The `int` keyword indicates that `main` returns an integer value.
  - A return value of 0 indicates the program exited normally (i.e., without runtime errors).
  - Any other return value typically indicates the program exited abnormally (i.e., errors happened!)
- Giving `void` as an argument indicates that this program is ignoring any passed arguments. The `void` keyword may be omitted.

printf() and stdout

Printing strings should be nothing new, but let's go over it anyways.

- `printf()` is equivalent to Python's `print()` function
  - The biggest difference is that `printf()` does *not* automatically append `\n` to the end of a string.
- Example

```
1  printf("From one string ");
2  printf("to the next,\nEveryone Lo");
3  printf("ves Lisp!");
```

String Formatting!

String formatting should also be nothing new, but there are some important differences.

- In Python, strings are delimited by either double or single quotes (`"Hello World"` ≡ `'Hello World'`)
- In C, single and double quotes have different meanings!
  - Double quotes are *string* delimiters.
  - Single quotes are *character* delimiters.

Escape sequence	Description
`\n`	Newline. Position the cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the cursor to the next tab stop.
`\a`	Alert. Produces a sound or visible alert without changing the current cursor position.
`\\`	Backslash. Insert a backslash character in a string.
`\"`	Double quote. Insert a double-quote character in a string.

Reading from stdin

The following program uses the scanf standard library function to read keystrokes from the stdin buffer.

```
1  // Program to add two numbers with user prompts
2  #include <stdio.h>
3
4  int main (void) {
5    int i1;
6    int i2;
7    printf("Enter your first integer\n");
8    scanf("%d", &i1);
9    printf("Enter your second integer\n");
10   scanf("%d", &i2);
11   printf("The sum is %d\n", (i1 + i2));
12 } // end of main
```

scanf and stdin

The standard library function `scanf` reads characters from the standard input buffer `stdin`.

```
1 scanf("%d", &i1);
```

- The first argument is a **format control string**, which indicates the data type that should be input by the user. (`%d` means `int`)
- The second argument is the variable we want `scanf` to put the data in, prepended with the **address of operator** `&`.
- We will be covering `&` in depth when we talk about **pointers**...

Revisiting `printf`

```
1 printf("The sum is %d", (i1 + i2));
```

- To print the value of a variable, you must:
  - use a **format specifying placeholder** in the first argument
  - supply the variable as the second argument
- Each data type has it's own placeholder.

Declaring Variables

You may have noticed that the above program uses variables

- Variables in C work the same as in other C-based languages.
- In contrast to Python, C variables require explicit declaration.
- A variable must be declared with a data type (in this case `int`), like so:

```
1  int x, y;
2  float z;
```

- Variables may also be declared with an initial value:

```
1  int x = 7, y = 8;
2  float z = 3.14;
```

- This is known as **instantiation**.

Fundamental Data Types

Declaration	Size (bytes)	placeholder
`short int`	2	%hd
`unsigned short int`	2	%hu
`unsigned int`	4	%u
`int`	4	%d
`long int`	8	%ld
`unsigned long int`	8	%lu
`long long int`	8	%lld
`unsigned long long int`	8	%llu
`signed char`	1	%c
`unsigned char`	1	%c
`float`	4	%f
`double`	8	%lf
`long double`	16	%Lf

Don't give me that Bool!

You may have noticed that the foregoing slide didn't have booleans on it!

- Boolean support was added to C in ISO/IEC 9899:1999, which came out in 1999.
- Unlike most other languages, booleans are not part of the C prelude! In order to use them, you have to include the standard boolean library.
- Add this line to your set of include statements:

```
1 #include<stdbool.h>
```

- Before this library, C programmers would use `int`'s to represent boolean values.
  - 0 ≡ False
  - All other values ≡ True (1 is typically used).

More Memory Concepts

- **Variables** are units of memory that have been assigned an identifier.
- The amount of memory allocated is dependent on the data type the variable is declared with.
- The specific arrangement of 1's and 0's at the memory location the variable indicates is the value of that variable.
- When a new value is assigned to a variable, the underlying memory is overwritten with the new value (i.e., the process is *destructive*).
- Reading a variable is *non-destructive*.

Static vs Dynamic Typing

- In Python, variables don't need to be declared with a data type (**Dynamic Typing**)
  - The Python interpreter manages the memory representation of variables
- In C, the type declaration tells the memory system how much memory to reserve for the variable, so the information must be present!
- This is known as **Static Typing**.
- Variables in the same program will not necessarily be allocated adjacent memory cells!

Operator? Get me Chicago!

Description	Syntax
Increment (postfix)	`x ++`
Decrement (postfix)	`x --`
Increment (prefix)	`++ x`
Decrement (prefix)	`-- x`
Negation	`-x`
Arithmetic Addition	`x + y`
Arithmetic Subtraction	`x - y`
Aritmetic Multiplication	`x * y`
Aritmetic Division	`x / y`
Aritmetic Modulus	`x % y`

Incremental Improvement

- The increment and decrement operators (`++` and `--` respectively) either add or subtract 1 from the operand.
- `++` and `--` use **implicit assignment**, so no assignment operator is required!
- Whether the operator is prefix or postfix effects the semantics
  - If the operator is prefix (`++x`), the increment/decrement is executed *before* the containing expression.
  - If the operator is postfix (`x--`), the increment/decrement is executed *after* the containing expression.
- Because of this implicit assignment, using `++` or `--` on an integer literal is a *syntax error*!

Incremental Example

```c
#include <stdio.h>
int main() {
    int var1 = 5, var2 = 5;

    // var1 is displayed
    // Then, var1 is increased to 6.
    printf("Variable 1 = %d\n", var1++);

    // var2 is increased to 6
    // Then, it is displayed.
    printf("Variable 2 = %d\n", ++var2);

    return 0;
}
```

```
Variable 1 = 5
Variable 2 = 6
```

It's All Relational

Relational Operators	Syntax
Equality	`x == y`
Inequality	`x != y`
Greater than	`x > y`
Greater than or equal to	`x >= y`
Less than	`x < y`
Less than or equal to	`x <= y`

Logical Operators	Syntax
Not	`!x`
And	`x && y`
Or	`x || y`

For Your Next Assignment...

Description	Syntax	Equivalent to
Assignment	`x = y`	–
Assignment plus addition	`x += y`	`x = x + y`
Assignment plus subtraction	`x -= y`	`x = x - y`
Assignment plus multiplication	`x *= y`	`x = x * y`
Assignment plus division	`x /= y`	`x = x / y`
Assignment plus modulus	`x %= y`	`x = x % y`

iffy Subject Matter

If statements are a bit different in C vs Python.

```
1  if (<condition>) {
2    <statments>
3  } else {
4    <statements>
5  }
```

In particular, elif is replaced with:

```
1  if (<condition1>) {
2    <statements>
3  } else if (<condition2>) {
4    <statements>
5  } else {
6    <statements>
7  }
```

Let's `switch`, just in `case`

Python dropped `switch` blocks in favour of `elif`.

```
switch (x) {
  case 1: // executes if x == 1
    break;
  case 2: // executes if x == 2
    // no break means control flows to next case
  case 3: // executes if x == 2 || x == 3
    break;
  default: // executes if x != 1, 2 or 3
}
```

Let's `switch`, just in `case` (cont.)

- In this example, `x` is the **controlling expression**.
- The value of the controlling expression is compared to each **case label**, which is a literal of the return type of the controlling expression.
- Execution jumps to the corresponding case, and exits the switch block when it hits a `break` statement.
- This means that execution may pass through *mulitple cases* before exiting the switch block.
- `default` functions like the terminating `else` in an if-else chain. If no case matches the value of the controlling expression, execution jumps to the default clause.

Let's `switch`, just in `case` (cont.)

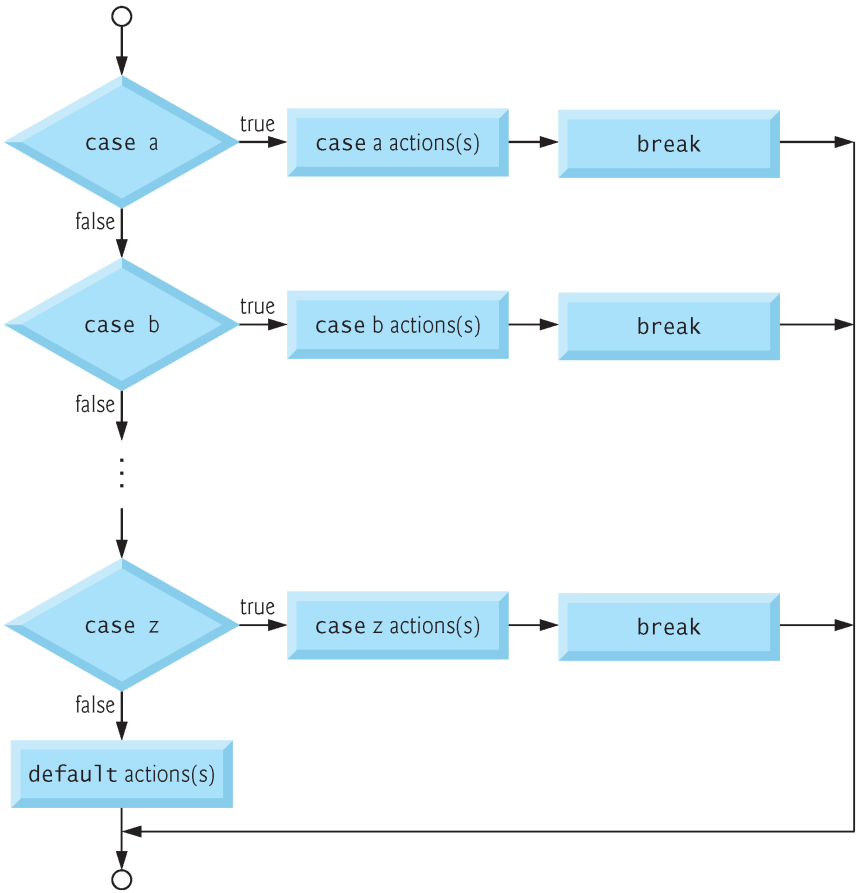

Fig. 4.8 | `switch` multiple-selection statement with `breaks`.

Let's switch to an example!

```
1  #include<stdio.h>
2
3  int main (void) {
4    char grade;
5    printf("Enter your grade: \n");
6    scanf("%c", &grade);
7    switch (grade) {
8      case 'A':
9        printf("Amazing! A smart person! ");
10     case 'B':
11       printf("You have met expectations.\n");
12       break;
13     case 'C':
14       printf("You need to study harder! ");
15     case 'D':
16       printf("At least you passed!\n");
17       break;
```

Let's `switch` to an example!

```
1     case 'F':
2        printf("Well, they're always hiring in the army.\
      n")
3        break;
4     default :
5        printf("That's not even a proper grade!\n");
6        break;
7   }
8 }
```

Cue Demonstration

Going Loopy

In C, there are three types of loops:

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

There are also two control statements for use in loops:

- `break;`
- `continue;`

`do while` I think of Another Pun

- `while` loops in C work the same way as in Python
- `do while` loops are a slight variation:

```
1  // <Initializing Statement>
2  do {
3    // <Body Statements>
4    // <Update Statement>
5  } while (/*<Condition>*/);
```

- `while` loops test their conditions *before* each loop iteration.
- `do while` loops test their conditions *after* each loop iteration.
- This means that a `do while` loop must execute *at least one* loop iteration.
- Aside from that, there is no semantic difference between `while` and `do while` loops.

formidable Coding

- In Python, a for loop is used to iterate over the elements of a data structure (lists, dictionaries, etc.)
- In C, for loops are just syntactic sugar for a while loop.

```c
// Countdown from 10 using a while loop
  int i = 10;
  while (i >= 0) {
     printf("%d...\n",i);
     i --;
  }
// Countdown from 10 using a for loop
  for (int i = 10; i >= 0; i--) {
     printf("%d...\n",i);
  }
```

`continue`... and `break` for lunch!

Two statements may be used to control loop execution outside of the loop's main conditional.

- `break` - exits the loop
  - When the program pointer hits a `break` statement, the loop it's in is immediately terminated, as if it's conditional test had returned false.
  - `break` can be very useful for programs with complex logic
  - The truth literal can even be used as the loop conditional if the program breaks correctly.
  - `break` is also used in `switch case` blocks.
- `continue` - starts next iteration
  - Jumps immediately to the next iteration of a loop.
  - The applications are not as numerous, most people use if-branching to not execute the rest of the code inside a loop, but `continue` can reduce the indentation level of your code.

Acknowledge

The contents of these slides were liberally borrowed (with permission) from slides from the Summer 2021 offering of 1XC3 (by Dr. Nicholas Moore).