ECE3411

Lecture#1b

Introduction to C Programming

Sung-Yeul Park

Department of Electrical & Computer Engineering

University of Connecticut

Email: sung_yeul.park@uconn.edu

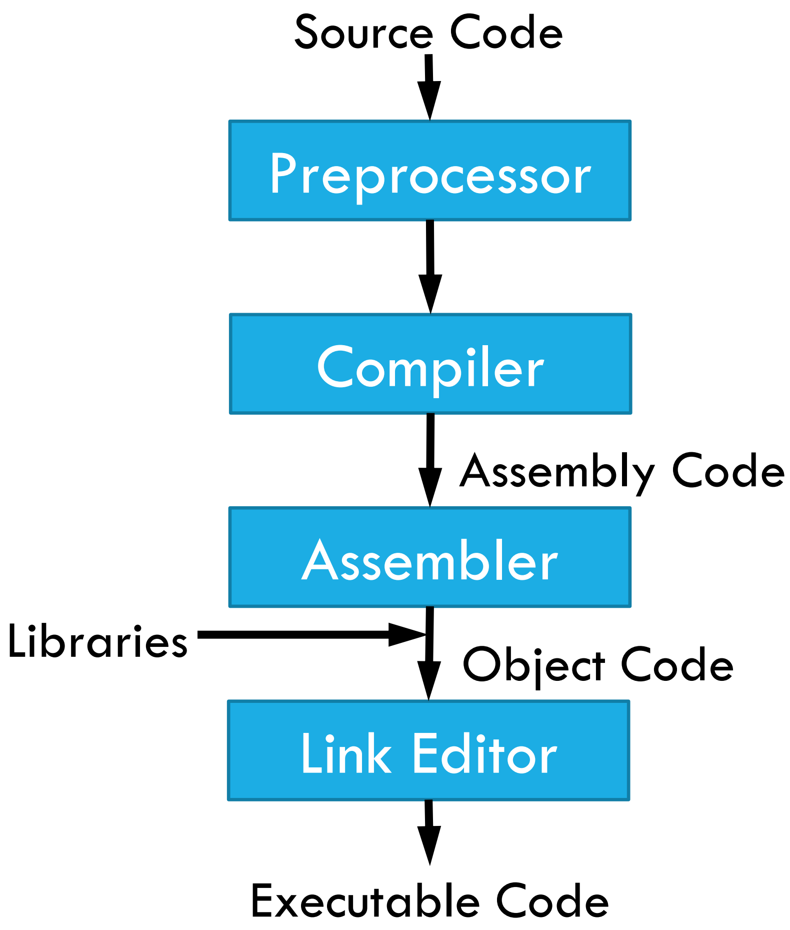
Slides adopted from John Chandy

Introduction to C Programming

- The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s.
- C is mother language of all programming languages used for systems programming.
- It is procedure-oriented and also a mid level programming language.
- C++ is a general-purpose object-oriented programming language.
- C# is a multi-paradigm programming language.

The C Compilation Model

- **The Preprocessor** accepts source code as input and is responsible for
  - Removing comments
  - Interpreting special preprocessor directives denoted by #.
  - Examples: **#include <stdio.h>** , **#define begin {** , **#define end }**
- **The C compiler** translates source to assembly code.
- **The assembler** creates object code.
- **The Link Editor** combines any library functions referenced in the source code with the **main()** function to create an executable file.



Basic data types

`char`	Stored as 8 bits. Unsigned 0 to 255. Signed -128 to 127.
`short int`	Stored as 16 bits. Unsigned 0 to 65535. Signed -32768 to 32767.
`int`	Same as either `short int` or `long int`
`long int`	Stored as 32 bits. Unsigned 0 to 4294967295. Signed -2147483648 to 2147483647
`float`	Approximate precision of 6 decimal digits (single precision).
`double`	Approximate precision of 14 decimal digits (double precision).

Constants

- Numerical Constants
  - Constants like 12 or 253 are stored as `int` type (No decimal point).
  - Numbers with a decimal point (21.53) are stored as `float` or `double`.
- Character and string constants
  - **`'c'`** , a single character in single quotes are stored as char.
  - Some special character are represented as two characters in single quotes.
    - **`'\n'`** = newline
    - **`'\t'`**= tab
    - **`'\\'`** = backlash
    - **`'\"'`** = double quotes
    - **`‘\r’`** = carriage return
  - A sequence of characters enclosed in double quotes is called a string constant or string literal.
    - For example : **`"Hello"`**

Special characters

- Some special character are represented as two characters in single quotes

Escape Sequence	Meaning	Description
\n	Newline	Moves the cursor to the next line
\r	Carriage return	Moves the cursor to the beginning of the line
\t	Horizontal tab	Inserts a tab space
\\	Backslash	Prints a backslash (\)
\'	Single quote	Prints a single quote (')
\"	Double quote	Prints a double quote (")
\b	Backspace	Moves the cursor one position back
\f	Form feed	Advances the paper feed (rarely used)
\a	Alert (bell)	Produces a beep sound (if supported)
\v	Vertical tab	Moves the cursor down a vertical tab space
\0	Null character	Marks the end of a string

Variables

- Variable names correspond to locations in the computer's memory
- Every variable has a name, a type, a size and a value
- Declaring a Variable
  - Each variable used must be declared. Example :

```
datatype var1, var2,…;
```

  - Declaration announces the data type of a variable and allocates appropriate memory location.
  - Initializing value to a variable in the declaration itself:

```
datatype var = expression;
```

  - Examples:

```
int sum;
char newLine = '\n';
float epsilon = 1.0e-6;
```

Variables

- Identifiers
  - An identifier is a sequence of letters and digits but must start with a letter.
  - Identifiers are used to name variables, functions etc.
  - Identifiers are case sensitive.
  - Valid: `Root, _getchar, __sin, x1, x2, x3, x_1, If`
  - Invalid: `324, short, price$, My Name`

Global and Local variables

- Global Variables
  - These variables are declared outside all functions.
  - Lifetime of a global variable is the entire execution period of the program.
  - Can be accessed by any function defined below the variable's declaration, in a file.
- Local Variables
  - These variables are declared inside some functions.
  - Lifetime of a local variable is the entire execution period of the function in which it is defined.
  - Cannot be accessed by any other function.
  - In general, variables declared inside a block are accessible only in that block.

Arithmetic Operators

- `A = B` → Assignment: A gets the value of B
- `A + B` → Add A and B together
- `A - B` → Subtract B from A
- `A * B` → A multiplied by B
- `A / B` → A divided by B
- `A % B` → Modulo: Integer remainder of A/B

Example:

```
int A = 11;
int B = 4;
int X = A / B;        // X gets the value 2. Since X is an integer,
                      //   the fractional part is ignored.
int Y = A % B;        // Y gets the value 3 since A=BX+Y
```

Bitwise Operators

Bitwise operators map input bit vectors to the same sized output bit vector

- `~A` → Bitwise complement of A
- `A & B` → Bitwise AND of A and B
- `A | B` → Bitwise OR of A and B
- `A ^ B` → Bitwise XOR of A and B
- `A << B` → Bitwise left shift A by B positions
- `A >> B` → Bitwise right shift of A by B positions

Bitwise Operators Examples

Let A = 0b110 and B = 0b010 then

- A represents the bit vector 110
- B represents the bit vector 010

```
~A       = 0b001
A & B    = 0b110 & 0b010 = 0b010
A | B    = 0b110 | 0b010 = 0b110
A ^ B    = 0b110 ^ 0b010 = 0b100
A << 1   = 0b110 << 1    = 0b100
A >> 1   = 0b110 >> 1    = 0b011
```

We use bitwise operators frequently to manipulate the register values.

Compound Assignments

- `A++` → `A = A + 1`
- `A--` → `A = A - 1`
- `A += B` → `A = A + B`
- `A -= B` → `A = A - B`
- `A *= B` → `A = A * B`
- `A /= B` → `A = A / B`
- `A %= B` → `A = A % B`
- `A &= B` → `A = A & B`
- `A |= B` → `A = A | B`
- `A <<= B` → `A = A << B`
- `A >>= B` → `A = A >> B`

Control Structures: if/else statement

```
if(<condition>)
        <statement>
```

```
if((<condition>)
{
        /* Block of statements */
}
```

```
if((<condition>) {
        /* Block of statements */
} else if ((<condition1>) {
        /*  ...  */
} else if ((<condition2>){
        /*  ...  */
} else {
        /* other statements */
}
```

- if statement can be used to execute some code if the condition is met.
- It can be used to execute a single code statement or a block of statements.
- if/else statement defines the alternate code to execute if the if-condition is not met.
- Note: if/else statements can be strung together with more if/else statements to add conditions to the ‘else’ parts.

Comparison Operators

- `A == B` → A is equal to B?
- `A != B` → A is NOT equal to B?
- `A > B` → A is greater than B?
- `A < B` → A is less than B?
- `A >= B` → A is greater than/equal to B?
- `A =< B` → A is less than/equal to B?

Logical Operators

Logical Operators map the inputs to either **TRUE** (Logical 1) or **FALSE** (logical 0)

These operators result in a single bit output

- `!A` → **NOT** A
- `A && B` → A **AND** B
- `A || B` → A **OR** B

Example:

```
if (A || (B && C) || !D)
{
   //do something;
}
```

`if` statement is only satisfied if

- A is logical high
  OR
- B **AND** C are logical high
  OR
- D is logical low.

Control Structures: switch statement

```
switch (<condition>)
{
    case <label1> :
        <statements 1>
         break;
    case <label2> :
        <statements 2>
        break;
    default :
        <statements 3>
}
```

- Used as a substitute for lengthy if statements that look for several conditions of some variable.

Control Structures: Loops

```
while ( <condition> )
{
    <statements>
}
```

```
for (<init>; <condition>; <update>)
{
    <statements>
}
```

```
do
{
    <statements>
}
while (<condition> );
```

- while loop: While the `condition` statement is true, execute the statements in the loop.
- for loop: Similar to the while loop. `init` initializes a variable, `condition` is a conditional expression, `update` is a modifier, like an increment (x++).
- do-while loop is similar to while loop. It ensures that the block of statements is executed at least once.

Break and Continue statements

- `break` is used to terminate a loop immediately.
- `continue` is used to skip the subsequent statements inside the loop.

Examples:

```
while(<condition1>){
      <statements>
      if(<condition2>)
            break;
      <statements>
}
```

```
while(<condition1>){
      <statements>
      if(<condition2>)
            continue;
      <statements>
}
```

Type conversion

- The operands of a binary operator must have the same type and the result is also of the same type.
- Integer division: `c = (9 / 5)*(f - 32)`
- The operands of the division are both int and hence the result also would be int.
- For correct results, one may write `c = (9.0 / 5.0)*(f - 32)`
- In case the two operands of a binary operator are different, but compatible, then they are converted to the same type by the compiler. The mechanism (set of rules) is called Automatic Type Casting.

  ```
  c = (9.0 / 5)*(f - 32)
  ```

- It is possible to force a conversion of an operand. This is called Explicit Type casting.

```
c = ((float) 9 / 5)*(f - 32)
```

Functions

- Functions are blocks of code that perform a number of pre-defined commands to accomplish something productive.
  - Library Functions
  - User Defined Functions
- Function prototypes are usually declared in the header files.
- General format for a function prototype

```
return-type function_name ( arg_type arg1, ..., arg_type argN );
```

- General format for a function body

```
return-type function_name ( arg_type arg1, ..., arg_type argN )
{
      /* Code for function body  */
}
```

Functions Example

```c
#include <stdio.h>
int mult ( int x, int y );     // Function Prototype

int main()
{
    int x, y, z;
    printf( "Please input two numbers to be multiplied: " );
    scanf( "%d", &x );  // Call to a library function
    scanf( "%d", &y );  // Call to a library function
    z = mult( x, y );           // Call to a user-defined function
    printf( "The product of your two numbers is %d\n", z );
}

/* Function Body */
int mult (int x, int y)
{
    return x * y;
}
```

Programming an MCU

- General Program Structure

```
int main()
{
    initialization code
    while (1) {
        main code
    }
}
```

- Arduino

```
void setup()
{
    initialization code
}

void loop()
{
    main code
}
```

Programming an MCU

- Event driven code

```
int main()
{
    initialization code
    while (1) {
       if (event1) {
          ...
       }
       if (event2) {
          ...
       }
    }
}
```

Programming an MCU

- Common programming paradigm
  - Wait for an event to happen

```
while(serial_port_ready == 0) {
}
read_serial_port();
```

  - Not very efficient. Code spins doing useless work
  - Especially bad if the I/O is relatively slow

Programming an MCU

- Use interrupts instead
  - Create an interrupt handler

```
ISR(serial_port_interrupt)
{
   read_serial_port();
}
```

  - Interrupt handler triggers only when the serial port is ready
  - Interrupt handler has overhead

Programming an MCU

- Try not to use delay functions

```
delay(10); // delay for 10 seconds
```

- This code just spins doing no useful work

- Use timers instead
  - Set a timer to trigger at desired time
  - Can get sub microsecond accuracy

Programming an MCU

- Things to be aware of
  - Not much data memory
    - Be careful creating huge arrays
    - Be conscious of data types – 8-bit vs. 32-bit
    - Complex data structures can be tricky
  - Not much program memory
    - Code needs to be more space efficient
  - Limited libraries
  - No operating system
    - Memory allocation, multiple processes, files, security, virtual memory, etc.

AVR128DB48 specifics

- `int` is a 16-bit value not 32-bit like on your computers
- Pointers are 16 bits long
- `printf` won't work until we setup the serial port to allow text output
- Because of the limited space in the microcontroller, `printf` is not fully functional
  - No `%f`
- Floating point operations are really slow on the microcontroller
- There is only 16K of memory for variables