

---

# Programming for Engineers

## Introduction to C

---

ICEN 200– Spring 2018  
Prof. Dola Saha



UNIVERSITY  
AT ALBANY  
State University of New York

# Simple Program

---

```
1 // Fig. 2.1: fig02_01.c
2 // A first program in C.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     printf( "Welcome to C!\n" );
9 } // end function main
```

Welcome to C!

**Fig. 2.1** | A first program in C.

# Comments

---

- `// Fig. 2.1: fig02_01.c`  
`// A first program in C`
  - begin with `//`, indicating that these two lines are **comments**.
  - Comments **document programs** and improve program readability.
  - Comments do not cause the computer to perform any action when the program is run.
  - You can also use `/*...*/` **multi-line comments** in which everything from `/*` on the first line to `*/` at the end of the line is a comment.
  - We prefer `//` comments because they're shorter and they eliminate the common programming errors that occur with `/*...*/` comments, especially when the closing `*/` is omitted.

# Preprocessor

---

## *#include Preprocessor Directive*

- **#include** <stdio.h>
  - is a directive to the C preprocessor.
- Lines beginning with # are processed by the preprocessor before compilation.
- Line 3 tells the preprocessor to include the contents of the standard input/output header (<stdio.h>) in the program.
- This header contains information used by the compiler when compiling calls to standard input/output library functions such as `printf`.

# Blank Lines, Spaces, Tabs

---

- You use blank lines, space characters and tab characters (i.e., “tabs”) to make programs easier to read.
- Together, these characters are known as **white space**. White-space characters are normally ignored by the compiler.

# Main Function

---

## *The main Function*

### ➤ `int main( void )`

- is a part of every C program.
  - The parentheses after `main` indicate that `main` is a program building block called a **function**.
- C programs contain one or more functions, one of which *must* be `main`.
- Every program in C begins executing at the function `main`.
- The keyword `int` to the left of `main` indicates that `main` “returns” an integer (whole number) value.

# Main Function

---

- We'll explain what it means for a function to “return a value” when we learn about Functions.
- For now, simply include the keyword `int` to the left of `main` in each of your programs.
- Functions also can receive information when they're called upon to execute.
- The `void` in parentheses here means that `main` does not receive any information.

# Body of Function

---

- A left brace, {, begins the **body** of every function
- A corresponding **right brace**, }, ends each function
- This pair of braces and the portion of the program between the braces is called a **block**.



# Output Statement

---

## *An Output Statement*

- `printf( "Welcome to C!\n" );`
  - instructs the computer to perform an **action**, namely to print on the screen the **string** of characters marked by the quotation marks.
  - A string is sometimes called a **character string**, a **message** or a **literal**.
  - The entire line, including the `printf` function (the “f” stands for “formatted”), its **argument** within the parentheses and the semicolon (`;`), is called a **statement**.
  - Every statement must end with a semicolon (also known as the **statement terminator**).
  - When the preceding `printf` statement is executed, it prints the message `Welcome to C!` on the screen.
  - The characters normally print exactly as they appear between the double quotes in the `printf` statement.

# Output Statement

---

## *Escape Sequences*

- Notice that the characters `\n` were not printed on the screen.
- The backslash (`\`) is called an **escape character**.
- It indicates that `printf` is supposed to do something out of the ordinary.
- When encountering a backslash in a string, the compiler looks ahead at the next character and combines it with the backslash to form an **escape sequence**.
- The escape sequence `\n` means **newline**.
- When a newline appears in the string output by a `printf`, the newline causes the cursor to position to the beginning of the next line on the screen.

# Output Statement

---

- Because the backslash has special meaning in a string, i.e., the compiler recognizes it as an escape character, we use a double backslash (`\\`) to place a single backslash in a string.
- Printing a double quote also presents a problem because double quotes mark the boundaries of a string—such quotes are not printed.
- By using the escape sequence `\"` in a string to be output by `printf`, we indicate that `printf` should display a double quote.

# Common Escape Sequences

---

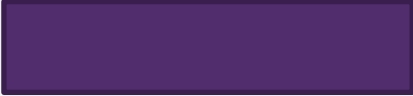
Escape sequence	Description
<code>\n</code>	Newline. Position the cursor at the beginning of the next line.
<code>\t</code>	Horizontal tab. Move the cursor to the next tab stop.
<code>\a</code>	Alert. Produces a sound or visible alert without changing the current cursor position.
<code>\\</code>	Backslash.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

**Fig. 2.2** | Some common escape sequences .


# Multiple Printf's

---

```
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     printf( "Welcome " );
9     printf( "to C!\n" );
10 } // end function main
```



```
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     printf( "Welcome\nto\nC!\n" );
9 } // end function main
```



# Multiple Printf

```
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     printf( "Welcome " );
9     printf( "to C!\n" );
10 } // end function main
```

Welcome to C!

```
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     printf( "Welcome\n\tto\n\tC!\n" );
9 } // end function main
```

Welcome  
to  
C!

# Scanf

---

```
1 // Fig. 2.5: fig02_05.c
2 // Addition program.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     int integer1; // first number to be entered by user
9     int integer2; // second number to be entered by user
10
11     printf( "Enter first integer\n" ); // prompt
12     scanf( "%d", &integer1 ); // read an integer
13
14     printf( "Enter second integer\n" ); // prompt
15     scanf( "%d", &integer2 ); // read an integer
16
17     int sum; // variable in which sum will be stored
18     sum = integer1 + integer2; // assign total to sum
19
20     printf( "Sum is %d\n", sum ); // print sum
21 }
```



# Formatted Input

---

## *The scanf Function and Formatted Inputs*

- The next statement
  - `scanf( "%d", &integer1 ); // read an integer` uses `scanf` to obtain a value from the user.
- The `scanf` function reads from the standard input
- This `scanf` has two arguments, `"%d"` and `&integer1`.
- The first, the `format control string`, indicates the type of data that should be input by the user.
- The `%d conversion specifier` indicates that the data should be an integer (the letter `d` stands for “decimal integer”).
- The `%` in this context is treated by `scanf` (and `printf` as we’ll see) as a special character that begins a conversion specifier.
- The second argument of `scanf` begins with an ampersand (`&`)—called the `address operator` in C—followed by the variable name.



# Scanf

---

- The `&`, when combined with the variable name, tells `scanf` the location (or address) in memory at which the variable `integer1` is stored.
- The computer then stores the value that the user enters for `integer1` at that location.
- The use of ampersand (`&`) is often confusing to novice programmers or to people who have programmed in other languages that do not require this notation.
- For now, just remember to precede each variable in every call to `scanf` with an ampersand.

# Printing with a Format Control String

- `printf( "Sum is %d\n", sum ); // print sum`
  - calls function `printf` to print the literal `Sum is` followed by the numerical value of variable `sum` on the screen.
  - This `printf` has two arguments, `"Sum is %d\n"` and `sum`.
  - The first argument is the format control string.
  - It contains some literal characters to be displayed, and it contains the conversion specifier `%d` indicating that an integer will be printed.
  - The second argument specifies the value to be printed.
  - Notice that the conversion specifier for an integer is the same in both `printf` and `scanf`.
- Calculations in `printf` statement
  - We could have combined the previous two statements into the statement
    - `printf( "Sum is %d\n", integer1 + integer2 );`

# Find error & correct it

---

```
scanf( "d", value );  
printf( "The product of %d and %d is %d"\n, x, y );  
firstNumber + secondNumber = sumOfNumbers  
Scanf( "%d", anInteger );
```

# Variables

---

## *Variables and Variable Definitions*

- `int integer1; // first number to be entered by user`  
`int integer2; // second number to be entered by user`  
`int sum; // variable in which sum will be stored`  
are **definitions**.
- The names `integer1`, `integer2` and `sum` are the names of **variables**—locations in memory where values can be stored for use by a program.
- These definitions specify that the variables `integer1`, `integer2` and `sum` are of type `int`, which means that they'll hold **integer** values, i.e., whole numbers such as 7, -11, 0, 31914 and the like.

# Variables

---

- All variables must be defined with a name and a data type before they can be used in a program.
- The preceding definitions could have been combined into a single definition statement as follows:
  - `int integer1, integer2, sum;`but that would have made it difficult to describe the variables with corresponding comments

# Identifiers

---

## *Identifiers and Case Sensitivity*

- A variable name in C is any valid **identifier**.
- An identifier is a series of characters consisting of letters, digits and underscores (`_`) that does *not* begin with a digit.
- C is **case sensitive**—uppercase and lowercase letters are different in C, so `a1` and `A1` are different identifiers.

# Common Errors & Practices



## Common Programming Error 2.2

*Using a capital letter where a lowercase letter should be used (for example, typing `Main` instead of `main`).*



## Error-Prevention Tip 2.1

*Avoid starting identifiers with the underscore character (`_`) to prevent conflicts with compiler-generated identifiers and standard library identifiers.*



## Good Programming Practice 2.5

*Choosing meaningful variable names helps make a program self-documenting—that is, fewer comments are needed.*

# Assignment Statement

---

- The **assignment statement**
  - `sum = integer1 + integer2; // assign total to sum`  
calculates the total of variables `integer1` and `integer2` and assigns the result to variable `sum` using the assignment operator `=`.
- The statement is read as, “`sum` *gets* the value of `integer1 + integer2`.” Most calculations are performed in assignments.
- The `=` operator and the `+` operator are called binary operators because each has two **operands**.
- The `+` operator’s two operands are `integer1` and `integer2`.
- The `=` operator’s two operands are `sum` and the value of the expression `integer1 + integer2`.



# Memory Concepts

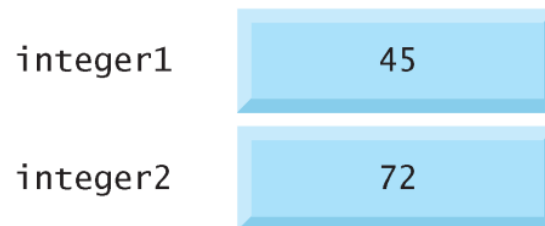
---

- Variable names such as `integer1`, `integer2` and `sum` actually correspond to locations in the computer's memory.
- Every variable has a name, a **type** and a **value**.
- In the addition program, when the statement
  - `scanf( "%d", &integer1 ); // read an integer`
- is executed, the value entered by the user is placed into a memory location to which the name `integer1` has been assigned.
- Suppose the user enters the number 45 as the value for `integer1`.
- The computer will place 45 into location `integer1`.

# Memory Concepts

---

- Whenever a value is placed in a memory location, the value replaces the previous value in that location; thus, this process is said to be **destructive**.
- When the statement
  - `scanf( "%d", &integer2 ); // read an integer`executes, suppose the user enters the value 72.
- This value is placed into location `integer2`, in the memory appears.
- These locations are not necessarily adjacent in memory.



# Memory Concepts

---

- Once the program has obtained values for `integer1` and `integer2`, it adds these values and places the total into variable `sum`.
- `sum = integer1 + integer2; // assign total to sum`
  - replaces whatever value was stored in `sum`.
- This occurs when the calculated total of `integer1` and `integer2` is placed into location `sum` (destroying the value already in `sum`).

---

<code>integer1</code>	45
<code>integer2</code>	72
<code>sum</code>	117

---

# Memory Concepts

---

- They were used, but not destroyed, as the computer performed the calculation.
- Thus, when a value is read from a memory location, the process is said to be **nondestructive**.

# Arithmetic in C

- The arithmetic operators are all binary operators.

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	$f + 7$	<code>f + 7</code>
Subtraction	-	$p - c$	<code>p - c</code>
Multiplication	*	$bm$	<code>b * m</code>
Division	/	$x / y$ or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Remainder	%	$r \text{ mod } s$	<code>r % s</code>

# Integer Division and Remainder

- **Integer division** yields an integer result
- For example, the expression  $7 / 4$  evaluates to **1** and the expression  $17 / 5$  evaluates to **3**
- C provides the **remainder operator**, **%**, which yields the remainder after integer division
- Can be used only with integer operands
- The expression  $x \% y$  yields the remainder after  $x$  is divided by  $y$
- Thus,  $7 \% 4$  yields **3** and  $17 \% 5$  yields **2**



## Common Programming Error 2.6

*An attempt to divide by zero is normally undefined on computer systems and generally results in a fatal error that causes the program to terminate immediately without having successfully performed its job. Nonfatal errors allow programs to run to completion, often producing incorrect results.*

# Parentheses

---

- Parentheses are used in C expressions in the same manner as in algebraic expressions.
- For example, to multiply a times the quantity  $b + c$  we write  $a * ( b + c )$ .

# Rules of Operator Precedence

---

- C applies **rules of operator precedence**, which are generally the same as those in algebra:
  - Operators in expressions contained within pairs of parentheses are evaluated first. Parentheses are said to be at the “highest level of precedence.” In cases of **nested**, or **embedded, parentheses**, such as
    - $( ( a + b ) + c )$   
the operators in the innermost pair of parentheses are applied first.
  - Multiplication, division and remainder operations are applied next.
  - Evaluation proceeds from left to right.
  - Addition and subtraction operations are evaluated next.
  - The assignment operator (=) is evaluated last.



# Order of precedence

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the <i>innermost</i> pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they’re evaluated left to right.
*	Multiplication	Evaluated second. If there are several, they’re evaluated left to right.
/	Division	
%	Remainder	
+	Addition	Evaluated third. If there are several, they’re evaluated left to right.
-	Subtraction	
=	Assignment	Evaluated last.

# Example order of precedence

