

# Computer Science & Engineering 150A

## Problem Solving Using Computers

### Lecture 02 - Introduction To C

Christopher M. Bourke

Spring 2009

[cbourke@cse.unl.edu](mailto:cbourke@cse.unl.edu)

- ▶ C Language Elements
- ▶ Variable Declarations and Data Types
- ▶ Executable Statements
- ▶ General Form of a C Program
- ▶ Arithmetic Expressions
- ▶ Formatting Numbers in Program Output
- ▶ Interactive Mode, Batch Mode, and Data Files
- ▶ Common Programming Errors

## Overview of C Programming

This chapter introduces C – a high-level programming language developed in 1972 by Dennis Ritchie at AT&T Bell Laboratories.

This chapter describes the elements of a C program and the types of data that can be processed by C. It also describes C statements for performing computations, for entering data, and for displaying results.

## C Language Elements

- ▶ Preprocessor Directives
- ▶ Syntax Displays for Preprocessor Directives
- ▶ "int main()" Function
- ▶ Reserved Words
- ▶ Standard Identifiers
- ▶ User-Defined Identifiers
- ▶ Uppercase and Lowercase Letters
- ▶ Program Style

## Preprocessor Directives

- ▶ The *C preprocessor* modifies the text of the C program before it is passed to the compiler.
- ▶ Preprocessor directives are C program lines beginning with a # that provide instructions to the C preprocessor.
- ▶ Preprocessor directives begins with a #, either `#include` or `#define`.
- ▶ Predefined libraries are useful functions and symbols that are predefined by the C language (standard libraries).

## #include and #define

- ▶ `#include<libraryName>` gives the program access to a library
- ▶ Example: `#include<stdio.h>` (standard input and output) has definitions for input and output, such as `printf` and `scanf`.
- ▶ `#define NAME value` associates a constant *macro*
- ▶ Example:

```
1 #define KMS_PER_MILE 1.609
2 #define PI 3.14159
```

## Comments

Comments provide supplementary information making it easier for us to understand the program, but comments are ignored by the C preprocessor and compiler.

- ▶ `/* */` - anything between them will be considered a comment, even if they span multiple lines.
- ▶ `//` - anything after this and before the end of the line is considered a comment.

## Function main

- ▶ The point at which a C program begins execution is the `main` function:

```
1 int main(void)
```

- ▶ **Every** C program must have a main function.
- ▶ The main function (and every other function) body has two parts:
  - ▶ Declarations - tell the compiler what memory cells are needed in the function
  - ▶ Executable statements - (derived from the algorithm) are translated into machine language and later executed

## Function main

- ▶ The main function contains *punctuation* and *special symbols*
  - ▶ Punctuation - Commas separate items in a list, semicolons appear at the end of each statement
  - ▶ Special Symbols: `*`, `=`, `{`, `}`, etc.
  - ▶ Brackets mark the beginning and end of the body of function main.

## Reserved Words

- ▶ A word that has special meaning in C
  - ▶ `int` - Indicates the main function returns an integer value,
  - ▶ `double` - Indicates the memory cells used to store these values will store real numbers.
- ▶ Always lower case,
- ▶ Can not be used for other purposes,
- ▶ Appendix E has a full listing of reserved words (ex: `double`, `int`, `if`, `else`, `void`, `return` etc.)

## Standard Identifiers

- ▶ Standard identifiers have a special meaning in C (assigned by standard libraries).
- ▶ Standard identifiers can be redefined and used by the programmer for other purposes
  - ▶ Not recommended If you redefine a standard identifier, C will no longer be able to use it for its original purpose.
- ▶ Examples: input/output functions `printf`, `scanf`

## User-Defined Identifiers

We choose our own identifiers to name memory cells that will hold data and program results and to name operations (*functions*) that we define (more on this in Chapter 3).

- ▶ An identifier must consist only of letters [`a-zA-Z`], digits [`0-9`], and underscores.
- ▶ An identifier cannot begin with a digit (and *shouldn't* begin with an underscore).
- ▶ A C reserved word cannot be used as an identifier.
- ▶ An identifier defined in a C standard library should not be redefined.

## User-Defined Identifiers

- ▶ Examples: `letter_1`, `Inches`, `KMS_PER_MILE`
- ▶ Some compilers will only see the first 31 characters
- ▶ Uppercase and lowercase are different (`Variable`, `variable`, `VARIABLE` are all *different*)
- ▶ Choosing identifier names:
  - ▶ Choose names that mean something,
  - ▶ Should be easy to read and understand,
  - ▶ Shorten only if possible
- ▶ Don't use `Big`, `big`, and `BIG` as they are easy to confuse
- ▶ Identifiers using all-caps are usually used for preprocessor-defined identifiers (`#define`)

## Program Style

A program that "looks good" is easier to read and understand than one that is sloppy (i.e. good spacing, well-named identifiers).

In industry, programmers spend considerably more time on program maintenance than they do on its original design or coding.

## Style Tips

### Rigorous Comments

- ▶ The number of comments in your program doesn't affect its speed or size.
- ▶ Always best to include as much documentation as possible in the form of comments.
- ▶ Begin each program or function with a full explanation of its inputs, outputs, and how it works.
- ▶ Include comments as necessary throughout the program

## Style Tips

### Naming Conventions

- ▶ Give your variables *meaningful* names (identifiers)
- ▶ `x`, `y` may be good if you're dealing with coordinates, but bad in general.
- ▶ `myVariable`, `aVariable`, `anInteger`, etc are *bad*: they do not describe the purpose of the variable.
- ▶ `tempInt`, `PI`, `numberOfStudents` are good because they do.

## Style Tips

### CamelCaseNotation

- ▶ Old School C convention: separate compound words with underscores
- ▶ `number_of_students`, `interest_rate`, `max_value`, etc.
- ▶ Underscore (shift-hyphen) is inconvenient
- ▶ Solution: camelCaseNotation - connect compound words with upper-case letters.
- ▶ Example: `numberOfStudents`, `interestRate`, `maxValue`, etc.
- ▶ Much easier to shift-capitalize
- ▶ Much more readable
- ▶ Ubiquitous outside of programming: MasterCard, PetsMart, etc.

## Anatomy of a Program

```
1  /*  
2  * Converts distances from miles to kilometers.  
3  */  
4  #include <stdio.h> /* printf, scanf definitions */  
5  #define KMS_PER_MILE 1.609  
6  
7  int main(void)  
8  {  
9      double miles, kilometers;  
10     printf("How many miles do you have?");  
11  
12     scanf("%lf",&miles);  
13  
14     kilometers = miles * 1.609;  
15     printf("You have %lf kilometers.", kilometers);  
16  
17     return 0;  
18 }
```

Labels pointing to code elements:

- Comments (lines 1-3)
- Preprocessor Directives (lines 4-5)
- Reserved Words (line 7)
- Variables (line 9)
- Special Symbols (line 10)
- Punctuation (line 18)

## Anatomy of a Program

```
/* Converts distances from miles to kilometers. */
#include <stdio.h> /* printf, scanf definitions */
#define KMS_PER_MILE 1.609 /* conversion constant */
int main(void)
{
    double miles; /* distance in miles */
    double kms; /* equivalent distance in kilometers */

    /* Get the distance in miles */
    printf("Enter the distance in miles> ");
    scanf("%lf", &miles);

    /* Convert the distance to kilometers. */
    kms = KMS_PER_MILE * miles;

    /* Display the distance in kilometers. */
    printf("That equals %f kilometers.\n", kms);

    return(0);
}
```

Diagram labels for the code above:

- preprocessor directive: `#include <stdio.h>`
- standard header file: `<stdio.h>`
- comment: `/* Converts distances from miles to kilometers. */`
- constant: `1.609`
- reserved word: `int`
- variable: `miles`, `kms`
- standard identifier: `printf`, `scanf`
- reserved word: `return`
- special symbol: `*`, `&`, `%`, `.`, `f`, `\n`
- punctuation: `;`

## Variable Declarations and Data Types

- ▶ Variable Declaration
- ▶ Data Types

## Variables Declarations

- ▶ Variables - a name associated with memory cells (`miles`, `kilometers`) that store a program's input data. The value of this memory cell can be changed.
- ▶ Variable declarations - statements that communicate to the compiler that names of variables in the program and the kind of information stored in each variable.
  - ▶ Example: `double miles, kms;`
  - ▶ Each declaration begins with a *unique* identifier to indicate the type of data
  - ▶ Every variable used must be declared before it can be used

## Data Types

- ▶ Data Types: a set of values and a set of operations that can be used on those values.
- ▶ In other words, it is a classification of a particular type of information.
  - ▶ Integers `int`
  - ▶ Doubles `double`
  - ▶ Characters `char`
- ▶ The idea is to give semantic meaning to 0's and 1's.

## Data Types

### Integers

- ▶ Integers are whole numbers, negative and positive
- ▶ Declaration: `int`
- ▶ The ANSI C standard requires integers be *at least* 16 bits: in the range -32767 to 32767
- ▶ One bit for the *sign* and 15 for the number
- ▶ Modern standard that `int` types are 32 bits. Range:  $-2^{31} = -2,147,483,648$  to  $2,147,483,648 = 2^{31}$
- ▶ Newer systems are 64-bit. What range does this give?

## Data Types

### Doubles

- ▶ Doubles are decimal numbers, negative and positive
- ▶ Example: 0.5, 3.14159265, 5, 8.33
- ▶ Declaration: `double`
- ▶ On most systems, doubles are 8 bytes = 64 bits
- ▶ Precision is *finite*: cannot *fully* represent  $\pi$ ,  $\frac{1}{3}$ , etc.
- ▶ An approximation only, but: 15–16 digits of precision

## Data Types

### Characters

- ▶ `char`: an individual character values with single quotes around it.
- ▶ Example: a letter, a digit, or a special symbol
- ▶ Example: `a`, `B`, `*`, `!`
- ▶ You can treat each character as a number: see Appendix A
- ▶ The ASCII standard assigns number (0 thru 255) to each character: `A` is 65, many are non-printable control characters

## Executable Statements

- ▶ Assignment Statements
- ▶ Input/Output Operations and Functions
- ▶ `printf` Function
- ▶ `scanf` Function
- ▶ `return` Statement

## Assignment Statements

- ▶ Assignment statements - stores a value or a computational result in a variable,
- ▶ Used to perform most arithmetic operations in a program.
- ▶ Form: `variable = expression;`
  - ▶ `kms = KMS_PER_MILE * miles;`

The assignment statement above assigns a value to the variable `kms`. The value assigned is the result of the multiplication of the constant macro `KMS_PER_MILE` (1.609) by the variable `miles`.

## Memory of Program

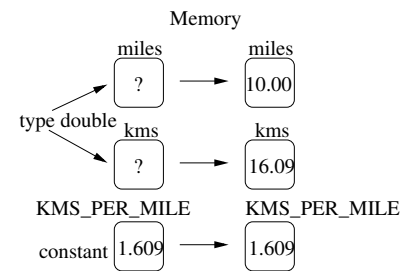


Figure: Memory

## Assignments Continued

- ▶ In C, the symbol `=` is the assignment operator.
- ▶ Read as “becomes”, “gets”, or “takes the value of” rather than “equals”
- ▶ In C, `==` tests equality.
- ▶ Examples:

```
1 int a, b, c;  
2 b = 10;  
3 a = 15;  
4 c = a + b;
```

## Assignments

### Misconception

- ▶ In C you can write:  
`sum = sum + item;`  
`a = a + 1;`
- ▶ These are *not* algebraic expressions
- ▶ This does not imply that `0 = 1`
- ▶ Meaning: `a` is to be given the value that `a` had before plus one
- ▶ Common programming practice
- ▶ Instructs the computer to add the current value of `sum` to the value of `item` then store the result into the variable `sum`.

## Input/Output Operations and Functions

- ▶ Input operation - data transfer from the outside world into memory.
- ▶ Output operation - An instruction that displays program results to the program user.
- ▶ input/output functions - special program units that do all input/output operations. Common I/O functions found in the *Standard Input/Output Library*: `stdio.h`
- ▶ Function call - in C, a function call is used to call or activate a function.
- ▶ Analogous to ordering food from a restaurant. You (the calling routine) do not know all of the ingredients and procedures for the food, but the called routine (the restaurant) provides all of this for you.

## Output Function: printf Function

Included in the `stdio.h` library.

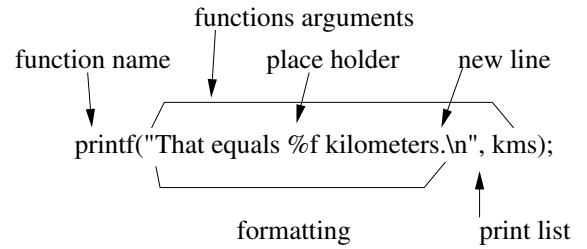


Figure: Parts of the printf function

## Place Holder

A placeholder always begins with the symbol `%`. Also the newline escape sequence `\n`. Format strings can have multiple placeholders.

Placeholder	Variable Type	Function Use
<code>%c</code>	<code>char</code>	<code>printf</code> , <code>scanf</code>
<code>%d</code>	<code>int</code>	<code>printf</code> , <code>scanf</code>
<code>%f</code>	<code>double</code>	<code>printf</code> only
<code>%lf</code>	<code>double</code>	<code>scanf</code> only

## Displaying Prompts

When input data is needed in an interactive program, you should use the `printf` function to display a **prompting message**, or **prompt**, that tells the program user what data to enter.

```
printf("Do you have any questions? ");
```

or

```
printf("Enter the number of items> ");
```

## Input Function: scanf

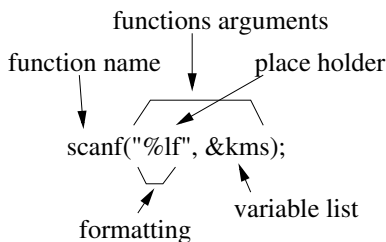


Figure: Parts of the scanf function

- ▶ In general, do not put extra characters in the format string
- ▶ Be sure to use the *ampersand*, `&` with `scanf`!!!

## Return Statement

```
return 0;
```

This statement returns a 0 to the Operating System to signify that the program ended in a correct position. It does not mean the program did what it was suppose to do. It only means there was no syntax errors. There still may have been logical errors.

## Program Example

```
1 #include <stdio.h>
2
3 int main(void) {
4     char first, last;
5
6     printf("Enter you first and last initials");
7     scanf("%c %c", &first, &last);
8
9     printf("Hello %c. %c. How are you?\n", first, last);
10
11     return 0;
12 }
```

## General Form of a C Program

- ▶ Programs begin with preprocessor directives that provide information about functions from standard libraries and definitions of necessary program constants.
  - ▶ #include and #define
- ▶ Next is the main function.
  - ▶ Inside the main function is the declarations and executable statements.

## General Form of a C Program

```
1 preprocessor directives
2
3 main function heading {
4     declarations
5
6     executable statements
7 }
8 }
```

## Program Style - Spaces in Programs

The compiler ignores extra blanks between words and symbols, but you may insert space to improve the readability and style of a program.

- ▶ You should always leave a blank space after a comma and before and after operators such as \*, -, and =.
- ▶ Indent the body of the main function, as well as between any other curly brackets.

```
1 int main(void) {
2     {
3         {
4             } // End Level 2
5         } /* End Level 1 */
6     }
7     return 0;
8 } // end main
```

## Comments in Programs

Use comments to do **Program Documentation**, so it help others read and understand the program.

- ▶ The start of the program should consist of a comment that includes programmer's name, date of current version, and brief description of what the program does.
- ▶ Include comments for each variable and each major step in the program.
- ▶ For any function, make comments to briefly describe the input to the function, the output of the function, and the use of the function.

## Comments in Programs

Style:

```
1 /*
2  * Multiple line comments are good
3  * for describing functions.
4  */
5
6 /* This /* is NOT */ ok. */
7
8 /* // ok. */
```

## Arithmetic Expressions

- ▶ Operators / and % (Read *mod* or *remainder*)
- ▶ Data Type of Expression
- ▶ Mixed-Type Assignment Statement
- ▶ Type Conversion through Cast
- ▶ Expressions with Multiple Operators
- ▶ Writing Mathematical Formulas in C

## Arithmetic Expressions

- ▶ To solve most programming problems, you will need to write arithmetic expressions that manipulate type `int` and `double` data.
- ▶ Each operator manipulates two operands, which may be constants, variables, or other arithmetic expressions.
- ▶ +, -, \*, / can be used with integers or doubles
- ▶ % can be used only with integers to find the remainder.

## Arithmetic Expressions

Operator	Meaning	Examples
+	addition	5 + 2 is 7 5.0 + 2.0 is 7.0
-	subtraction	5 - 2 is 3 5.0 - 2.0 is 3.0
*	multiplication	5 * 2 is 10 5.0 * 2.0 is 10.0
/	division	5 / 2 is 2 5.0 / 2.0 is 2.5
%	remainder	5 % 2 is 1

## Division

- ▶ When applied to two positive integers
  - ▶ The division operator (/) computes the integer part of the result of dividing its first operand by its second.
  - ▶ Example: the result of  $7 / 2$  is 3
  - ▶ C only allows the answer to have the same accuracy as the operands.
  - ▶ If both operands are integers, the result will be an integer.
- ▶ If one or both operands are double, the answer will be a double.
- ▶ According to the book different C implementations differ on dividing by a negative number.
- ▶ / is undefined when the second operand is  $0.4 / 0 = ?$

## Remainder Operator

The remainder operator (%) returns the integer remainder of the result of dividing its first operand by its second.

- ▶ Similar to integer division, except instead of outputting integral portion, outputs remainder.
- ▶ The operand % can give different answers when the second operand is negative.
- ▶ As with division, % is undefined when the second operand is 0.

## Remainder Operator

### Exercise

#### Problem

What are the results of the following operations?

1.  $51 \% 2$
2.  $100 \% 4$
3.  $101 \% 31$

1.  $51 \% 2 \rightarrow 1$
2.  $100 \% 4 \rightarrow 0$
3.  $101 \% 31 \rightarrow 8$

## Data Type of an Expression

The data type of each variable must be specified in its declaration, but how does C determine the data type of an expression?

- ▶ The data type of an expression depends on the type(s) of its operand(s). If both are of type `int`, then result is of type `int`. If either one or both is of type `double`, then result is of type `double`.
- ▶ An expression that has operands of both `int` and `double` is a mixed-type expression, and will be typed as `double`.

For a mixed-type assignment, be aware that the expression is evaluated first, and then the *result* is converted to the correct type.

## Type Conversion through Type Casting

C is flexible enough to allow the programmer to convert the type of an expression by placing the desired type in parentheses before the expression, an operation called a *type cast*.

Example: `(double)5 / 2` results in 2.5, not 2 as seen previously.

## Expressions with Multiple Operators

- ▶ In expressions, we often have multiple operators
  - ▶ Equations may not evaluate as we wish them to:
    - ▶ Is  $x/y * z$  the same as  $(x/y) * z$  or  $x/(y * z)$ ?
- ▶ **Unary operators** take only one operand: `-5`, `+3`, `-3.1415`, etc.
- ▶ **Binary operators** take two operands: `3 + 4`, `7 / 5`, `2 * 6`, `4 % 3`, etc.

## Rules for Evaluating Expressions

1. *Parenthese rule*: All expressions in parentheses must be evaluated separately. Nested parenthesized expressions must be evaluated from the inside out, with the innermost expression evaluated first.
2. *Operator precedence rule*: Operators in the same expression are evaluated in a fixed order (see Table 1 in your book)
  - Unary Operators    `+, -`
  - Binary Operators    `*, /, %`
  - Binary Arithmetic    `+, -`
3. *Associativity rule*:
  - ▶ Unary operators in the same subexpression and at the same precedence level are evaluated right to left.
  - ▶ Binary operators in the same subexpression and at the same precedence level are evaluated left to right.

## Exercise

### Problem

What are the results of the following expressions:

1.  $x = -15 * 4 / 2 * 3;$
2.  $y = 5 + 2 * 4;$
3.  $z = 4 - 5 * 2 - 4 + -10;$

1.  $x = -90$  (why not  $-10$ ?)
2.  $y = 13$
3.  $z = -20$

## Make Expressions Unambiguous

$x = (-15 * 4) / (2 * 3);$

## Writing Mathematical Formulas in C

### Pitfalls

Common misconception: Mathematical Formulas in algebra and in C are *not* the same.

- ▶ Algebra: multiplication is implied:  $xy$
- ▶ C: Operations must be explicit:  $x * y$
- ▶ Algebra: division is written  $\frac{a+b}{c+d}$
- ▶ C: Cannot use such conveniences, must write  $(a + b)/(c + d)$

## Formatting Numbers in Program Output

- ▶ Formatting Values of Type `int`
- ▶ Formatting Values of Type `double`
- ▶ Program Style

C displays all numbers in its default notation unless you instruct it to do otherwise.

## Formatting Integer Types

- ▶ Recall the placeholder in `printf/scanf` for integers: `%d`
- ▶ By default, the complete integer value is output with no leading space
- ▶ You can insert a number to specify the *minimum* number of columns to be printed: `%nd`
- ▶ If  $n$  is less than the number of digits, no effect
- ▶ Otherwise, leading blank spaces are inserted before the number so that  $n$  columns are printed

## Formatting Integer Types

### Example

```
1 int x = 2345;
2 printf("%6d\n", x);
3 printf("%2d\n", x);
```

Result:

```
1 2345
2 2345
```

## Formatting Values of Type double

- ▶ Recall the placeholder for doubles: `%f`
- ▶ Must specify both the total number of columns as well as the *precision* (number of decimal digits)
- ▶ Format: `%n.mf`
  - ▶  $n$  is the field width (minimum number of columns to be printed *including* the decimal)
  - ▶  $m$  is the number of digits after the decimal to be printed (may end up being padded with zeros)
- ▶ May or may not define both  $n, m$ .

## Formatting Values of Type double

### Example

```
1 double pi = 3.141592;
2 printf("%.2f\n", pi);
3 printf("%6.2f\n", pi);
4 printf("%15.10f\n", pi);
```

Result:

```
1 3.14
2 3.14
3 3.1415920000
```

## Programming Tip

Man pages

- ▶ The command `man` (short for manual) can be used to read documentation on standard library functions.
- ▶ Example: `:prompt>man printf` gives a detailed description on `printf`
- ▶ Contains additional information: how to print leading zeros instead of blanks, etc.

## Interactive Mode, Batch Mode, and Data Files

- ▶ Input Redirection
- ▶ Program Style
- ▶ Output Redirection
- ▶ Program-Controlled Input and Output Files

## Definitions

- ▶ Active mode - the program user interacts with the program and types in data while the program is running.
- ▶ Batch mode - the program scans its data from a data file prepared beforehand instead of interacting with its user.

## Input Redirection

- ▶ Recall miles-to-kilometers conversion program: active mode prompted user for input
- ▶ If expected formatting of input/output is known, you can put it in a plain text file and use *input/output redirection* on the command line
- ▶ Example:

```
prompt:> conversion < mydata
```

where `mydata` is a plain text file containing a single double-formatted number.

Recall the mile-to-kilometer program.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     double miles, kilometers;
5     printf("How many miles do you have?");
6
7     scanf("%lf",&miles);
8
9     kilometers = miles * 1.609;
10    printf("You have %f kilometers\n",kilometers);
11
12    return 0;
13 }
```

## Echo Prints vs. Prompts

- ▶ `scanf` gets a value for `miles` from the first (and only) line of the data file
- ▶ If we will only run the program in batch mode, there is no need for the prompting message
- ▶ We do need to output the answer though:  
`printf("The distance in miles is %.2f.\n",miles);`
- ▶ However, we can also redirect the output to a file:  
`prompt:> conversion < mydata > result.txt`
- ▶ Its enough to echo only the number: `printf("%.2f.\n",miles);`

## Program-Controlled Input and Output Files

As an alternative to input/output redirection, C allows a program to explicitly name a file from which the program will take input and a file to which the program will send output. The steps needed to do this are:

1. Include `stdio.h`
2. Declare a variable of type `FILE *`.
3. Open the file for reading, writing or both.
4. Read/write to/from the file.
5. Close the file.

## Program Example

### File Input/Output

```
1 #include <stdio.h>
2 #define KMS_PER_MILE 1.609
3
4 int main(void) {
5     double kms, miles;
6     FILE *inp, *outp;
7
8     inp = fopen("distance.dat","r");
9     outp = fopen("distance.out","w");
10    fscanf(inp, "%lf", &miles);
11    fprintf(outp, "The distance in miles is %.2f.\n", miles);
12
13    kms = KMS_PER_MILES * miles;
14    fprintf(outp, "That equals %.2f kilometers.\n", miles);
15    fclose(inp);
16    fclose(outp);
17    return 0;
18 }
```

## Common Programming Errors

- ▶ Syntax Errors
- ▶ Run-Time Errors
- ▶ Undetected Errors
- ▶ Logic Errors

## Errors

- ▶ Bugs - Errors in a programs code.
- ▶ Debugging - Finding and removing errors in the program.
- ▶ When the compiler detects an error, it will output an error message.
  - ▶ Difficult to interpret
  - ▶ Often misleading
- ▶ Three types of errors
  - ▶ Syntax error
  - ▶ Run-time error
  - ▶ Undetected error
  - ▶ Logic error

## Syntax Errors

A syntax error occurs when your code violates one or more grammar rules of C and is detected by the compiler at it attempts to translate your program. If a statement has a syntax error, it cannot be translated and your program will not be executed.

Common syntax errors:

- ▶ Missing semicolon
- ▶ Undeclared variable
- ▶ Last comment is not closed
- ▶ A grouping character not closed ('(', '{', '[')

## Run-Time Errors

- ▶ Detected and displayed by the computer during the *execution* of a program
- ▶ Occurs when the program directs the computer to perform an illegal operation. Example: dividing a number by zero or opening a file that does not exist
- ▶ When a run-time errors occurs, the computer will stop executing your program
- ▶ May display a useful (or not) error message
- ▶ Core dump, bus error, etc

## Undetected Errors

- ▶ Code was correct and logical, executed fine, *but* lead to incorrect results.
- ▶ Essential that you test your program on known correct input/outputs.
- ▶ Common formatting errors with `scanf/printf`: keep in mind the correct placeholders and syntax.

## Logic Errors

- ▶ Logic errors occur when a program follows a faulty algorithm.
- ▶ Usually do not cause run-time errors and do not display error messages—difficult to detect
- ▶ Must rigorously test your program with various inputs/outputs
- ▶ Pre-planning your algorithm with pseudocode or flow-charts will also help you avoid logic errors

## Questions

Questions?

## Exercise

Debug the following program:

```
1  /*
2  * Calculate and display the difference of two input values
3  */
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int a, b; /* inputs
9      integer sum; /* sum of inputs */
10     printf("Input the first number: %d", &A);
11     printf("Input the second number: ");
12     scanf("%d", b);
13     a + b = sum;
14     printf("%d + %d = %d\n", a, b, sum);
15     return 0;
```

## Exercise: Answer

```
1  /*
2  * Calculate and display the difference of two input values
3  */
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int a, b; /* inputs */
9      int sum; /* sum of inputs */
10     printf("Input the first number: ");
11     scanf("%d", &a);
12     printf("Input the second number: ");
13     scanf("%d", &b);
14     sum = a + b;
15     printf("%d + %d = %d\n", a, b, sum);
16     return 0;
17 }
```