Computer Science I

Basics

Dr. Chris Bourke
cbourke@cse.unl.edu

## Outline



# Part I: Introduction

Overview, Compiling, Elements of a Program

## Overview

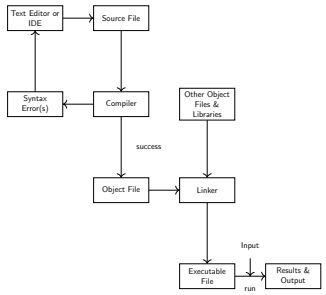
- A *computer program* is a collection of instructions that performs a specific task when executed by a computer
- A *programming language* is a formal language that specifies a set of instructions that can be used in a program
- We write programs in a code (text) editor or Integrated Development Environment (IDE)
- Source code is is *compiled* and run on a particular *operating system*

## Process

- *Source code* – plain text files containing computer code.
- *Assembly* – a low-level programming language closer to a processor's actual "language"
- *Machine Code* – set of instructions executed directly by a computer's central processing unit (CPU)
- *Binary* – a sequence of 0s and 1s

$$\text{source} \rightarrow \text{assembly} \rightarrow \text{machine}$$

## Compiling



## Demonstration

- Edit a source file, `hello.c`
- Assemble:
  `gcc -S hello.c`
- Compile:
  `gcc -c hello.c`
- View binary/hex:
  `hexdump -C hello.o`
- All in one: `gcc hello.c`
- Run: `./a.out`

## Example

```c
/**
 * Author: Chris Bourke
 * Date: November 2, 2016
 *
 * This program converts miles to kilometers
 */
#include<stdlib.h>
#include<stdio.h>

#define KMS_PER_MILE 1.60934

int main(int argc, char **argv) {

  double miles, kms;

  printf("Please enter miles: ");

  scanf("%lf", &miles);

  kms = KMS_PER_MILE * miles;

  printf("%f miles is equal to %f kilometers\n", miles, kms);

  return 0;
}
```

## Comments & Documentation

- Comments are human-readable messages embedded in code
- Single line comments: `// this is a comment`
- Multiple line comments:

```c
/* This is a multiline
   comment. */
```

## Comments & Documentation

- "Doc-style" comments:

```c
/**
 * This is a doc-style comment.  It is
 * a multiline comment commonly used for
 * large blocks of documentation comments
 */
```

- Comments should tell you the *what* and the *why*
- Code should be self-documenting, code itself should tell the *how*

## Preprocessor Directives

- Preprocessor directives tell the compiler to do certain things to the source code before compiling
- Begin with a hash `#`
- Macros: `#define a b`
- Macro usage: defining constants; avoiding "magic" numbers
- Including libraries:

```c
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
```

## Main Function

- The `main` function is the main entry point of a program
- Without a `main` a program is not *executable*
- Input/output functions: `scanf`, `printf`
- Math functions: `sqrt`, `sin`, `pow`

## Syntax & Punctuation

- In general, executable lines ends with a semicolon `;`
- *Blocks* of code are delimited by opening and closing curly brackets `{ ... }`
- Commas `,` are used to delimit arguments, variables, etc.
- In general, whitespace does not affect a program
- Keywords: `int`, `double`, `return`

# Part II: Variables & Operators

## Variables

- Variables are program elements that hold *values* that can be *assigned* and *reassigned*
- C is a *statically typed* language
- All variables have a name (*identifier*) and a fixed *type*
- Types: integers, floating-point (decimal) numbers, characters, etc.
- Variables must be *declared* before they can be used
- The *scope* of a variable is the section(s) of code in which it exists

## Basic Types

### Integers

- An integer variable: `int`
- A 32-bit signed two-s complement integer variable
- Can represent values:

$$-2^{31} = -2,147,483,648 \leq x \leq 2,147,483,647 = 2^{31} - 1$$

- Examples:

```
1  int x;
2  int numberOfStudents = 42;
3  int golfScore = -3;
```

## Basic Types

### Floating Point Numbers

- A `double` is a decimal number
- A 64-bit "floating" point number
- Similar to scientific notation:

$$3895.434 \rightarrow 3.895434 \times 10^{3}$$

$$-0.0043 \rightarrow -4.3 \times 10^{-3}$$

- Decimal values accurate to 16–17 digits
- Another: `float` (avoid, less precise)
- Examples:

```
1  double pi = 3.14159;
2  double totalCost = 3219.32;
```

## Basic Types

### Single Characters

- A `char` is a single character
- American Standard Code for Information Interchange (ASCII) character
- Examples:

```
1  char firstInitial = 'C';
2  char response = 'y';
```

## ASCII Table

# ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [END OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

## Variable Naming

Variable Name Rules:

- May contain `a-z`, `A-Z`, `0-9` or `_`
- May *not* begin with a number

Conventions:

- `under_score_casing`
- `UPPERCASE_UNDER_SCORE_CASING`
- `camelCasingConvention`
- Names should be short but *descriptive*
- Good: `initialValue`, `longitude`, `latitude`, `interestRate`
- Bad: `var1` `somevalue`

## Assignment Operator

- Assignment operator: single equal sign `=`
- "Place the value on the right-hand-side into the variable on the left-hand-side"
- Not an arithmetic equality
- Right hand side may be a:
  - Literal: hard-coded numerical value
  - Another variable (copy a variable's value)
  - An arithmetic expression

## Assignment Operator

Examples

```
1  //declare an int variable and set it to 10:
2  int a;
3  a = 10;
4  //or
5  int a = 10;
6
7  double c = 10.5 + a * 2 - 10 / 2;
```

## Arithmetic Operators

- Addition: $+ \rightarrow$ `+`
- Subtraction: $- \rightarrow$ `-`
- Multiplication: $\times \rightarrow$ `*`
- Division: $\div \rightarrow$ `/`
- Integer division: `%` gives the remainder
  - `5 % 2` results in `1`
  - `11 % 3` results in `2`
  - `12 % 6` results in `0`

## Order of Precedence

- Arithmetic follows the same basic order of operations
- Left-to-right
- Multiplication/division before addition/subtraction
- $5 + 12 \div 2 = 11$
- $5 + 12 \div 2 \neq (5 + 12)/2$
- Use parentheses to redefine order
- Best practice: use parentheses even when not necessary to indicate intent
- Examples:

```
1  (a + 10) * (b - c)
2  (a * b) + (c / d)
```

## Truncation

- Arithmetic with integers results in integers
- Arithmetic with `double` values results in floating-point numbers
- Not the same type of variables; sometimes they are *incompatible*
- Example:

```
//okay:
int a = 10;
double x = 10;

//in C, this results in truncation
int b = 3.14;  //b has the value 3
```

## Truncation

- *Truncation* is when the fractional part is "chopped off"
- Especially important with respect to division:

```
int a = 10;
int b = 20;

double c = a / b;  //results in 0.0

//explicit type cast:
double d = a / (double) b; //results in 0.5
```

# Part III: Basic Input/Output

## Basic Input/Output

- All systems support *standard streams*
- Standard input, standard output, standard error
- Standard input/output library: `stdio.h`
- Standard input: keyboard, use `scanf`
- Standard output: "console" or terminal, use `printf`

## Basic Input

- `scanf`: Scan Formatted
- First argument: a string containing a formatting *placeholder*:
  - `int`: use `%d`
  - `double`: use `%lf`
  - `char`: use `%c`
- Second argument: variable to store the value into
- *You must place an ampersand in front of the variable*

## Basic Input

Examples

```
int a;
double x;

//prompt the user for an integer
printf("enter an integer: ");
scanf("%d", &a);
printf("%d\n", a);

//prompt for a double:
printf("enter a value: ");
scanf("%lf", &x);
printf("%f\n", x);
```

## Basic Output

- `printf` : Print Formatted
- First argument: a string containing content and formatted *placeholder*:
  - `int` : use `%d`
  - `double` : use `%f`
  - `char` : use `%c`
- Multiple placeholders may be used
- Each subsequent argument will replace each placeholder

## Basic Output

Examples

```c
int a = 10;
double x = 3.14;
char initial = 'C';

printf("a = %d, x = %f and my initial is %c\n", a, x, initial);
```

## Formatting Floats

- Default formatting for floating point numbers: 6 decimals of accuracy
- Placeholder modifier: `%n.mf`
- `m` : number of decimals of accuracy
- `n` : *minimum* number of total columns to print including the decimal point
- A negation can be used to justify left: `%-10.2f`

## Formatting Floats

Examples

```c
double pi = 3.1415926;

printf("%f", pi);
printf("%.3f", pi);
printf("%.5f", pi);
printf("%10.2f", pi);
printf("%4.4f", pi);
printf("%-10.2f", pi);
```

## Good Code Style

- Code must be readable and easily understood
- Code must be self-documenting
- Use accurate, descriptive variable names
- Consistent naming convention
- Good use of whitespace
- Consistent indentation

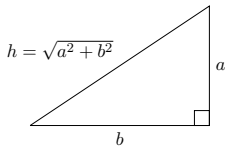
# Part IV: Exercises

## Exercise

Write a program to read in three values from the user and print their average.

## Exercise

Write a program that prompts the user for the length of two sides of a right triangle and outputs the length of its hypotenuse using the Pythagorean Theorem:




## Exercise

Write a program to calculate mileage reimbursement. Read in the beginning and ending odometer values as well as a per-mile rate from the user and display the total distance travelled as well as the total reimbursement.

## Exercise

Write a program to convert a number of days into years (assume 365), weeks, and days. For example, if the user enters 1,000 days, it would display "2 years, 38 weeks, 4 days."

# Part V: Noninteractive Input

Using Command Line Arguments

## Noninteractive Input

- Prompt/read (`printf` / `scanf`) is interactive
- Most real programs are *not* interactive
- Input can be provided as Command Line Arguments (CLAs)
- Specified when you invoke a program:
  `./a.out 10 20 3.5 hello`
- First argument is *always* the executable file name
- Programs can access them using the `argv` parameter
- `argv[0]`, `argv[1]`, `argv[2]`, etc.
- Number of arguments: `argc`
- Each argument is a string, not a number
- Conversion:
  - `atoi()` for `int`
  - `atof()` for `double`
- Demonstration

# Part VI: Toolkit Demo – Linters

## Linters

- Compilers only check for valid syntax
- *Static analysis* tools analyze the structure of code to detect *potential* bugs/errors
- A *linter* is a utility used to identify code that may be syntactically correct but that may indicate potential bugs
- GCC can be used as a basic linter by forcing it to identify all warnings
  `gcc -Wall foo.c`
- Demonstration