


# Intro to C

CSE 2421

# The Joy of C

- C is procedural, not object-oriented
- C is fully compiled (to machine code), not to byte-code
- C allows direct manipulation of memory (via pointers)
- C does not have garbage collection; the software writer has to do explicit memory management when it is required, and failure to do so can result in significant problems (e.g., **memory leaks**, where the memory used by the program may grow potentially without bound)
- Many of the basic language constructs in C act similarly to the way they work in Java; nonetheless, there are sometimes important differences which need to be understood
- C has many nuanced, yet important details

# C does NOT...

- C does not support the notion of Classes or Objects
  - C does not itself support Encapsulation, as all memory is technically accessible and modifiable by any instruction of an executable
  - C does not itself support class Inheritance.
  - C is not an object oriented language, it is procedural.
- 

# ANSI C

- ▶ In this class, we will learn ANSI C, which was originally standardized in 1989 by the American National Standards Institute. This version of C is sometimes referred to as “C89” (also sometimes referred to as “Standard C”). The ISO, or International Standards Organization, also adopted an equivalent version of C in 1990, which is often referred to as C90. Therefore, C89 and C90 are in effect equivalent.
- ▶ There are later versions of C, for example, C99, which differ from ANSI C; for example, they may support features that ANSI C does not support (such as just-in-time declaration, for example).
- ▶ You are *required* to build (or compile) your C programs for this course using the gcc options `-ansi` and `-pedantic`, which will cause the compiler to enforce C89/C90 requirements for valid programs, and give warnings related to the ANSI standard. You also need the two `-W` flags as well. For example, if you are building your lab1 source code, you must use (assuming lab1 is the name of the executable to be produced by the build process):

```
% gcc -ansi -pedantic -Wreturn-type -Wimplicitfunction-declaration -o lab1 lab1.c
```

# Compile time versus run time

- ▶ In computer science, we distinguish events that can occur when a program is being compiled or built, called **compile time**, from events which can occur when the program is actually being executed, or running, called **run time**.
- ▶ For example, certain kinds of errors can be identified or occur at compile time, and others at run time.
- ▶ Errors which occur at compile/build time: ***syntax errors*** [these make a program *invalid*, i.e., in the case of C, *not a valid C program*], ***or possibly linkage errors – more on this in a few weeks***. In this case, the compiler will not even generate an executable.
- ▶ Compile/build time events are also often referred to as ***static***, and run time events as ***dynamic***.
- ▶ Errors which occur at run time: ***semantic errors*** [these do not make a program invalid, because only a valid program can actually be built and then executed]. An example of such an error would be division by zero. The compiler will not discover such an error, but when the program runs, an exception will be generated, and the operating system will terminate the program.

# The Bigger Picture

- ▶ Four General Categories of Statements in Languages
  - Declarations (optional in some languages like Python)
  - Data Movement
    - Memory to function variables
    - Function variables to function variables
    - Function variables to memory
  - Arithmetic/Logical Operations
    - Compare something
    - Calculate something
  - Control-Flow
    - Procedure/function calls
    - Looping
    - Conditionals

# First C program: Hello World

```
/* Version 1 */
```

```
#include <stdio.h>
```

```
void main (int argc, char **argv) {  
    printf("Hello, ");  
    printf("World!\n");  
}
```

```
/* Version 3 */
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main () {
```

```
    printf("Hello, World!\n");
```

```
    return (EXIT_SUCCESS);
```

```
}
```

```
/* Version 2 */
```

```
#include <stdio.h>
```

```
int main (void) {
```

```
    printf("Hello, World!\n");
```

```
    return (0);
```

```
}
```



# Hello programs – Preprocessor

- ▶ Notice the lines at the top which begin with the # character. These are known as **preprocessor directives**, and are used by a translation program called **the preprocessor**, which is the first program called when you build C source code (in our case, with gcc).
- ▶ *Pointers on C*, read Chapter 14, *The Preprocessor*
  - *The engineering library has a copy if you didn't purchase the book.*



# Preprocessor (continued)

- ▶ What the preprocessor does is find, in the area on disk where library files are kept, the file with the name `stdio.h` or `stdlib.h`. It **replaces** the directive in the source file with a copy the contents of the specified header file.
- ▶ The header file may contain various things, but one thing it usually contains is one or more **function prototypes (function declarations – see below)**, which tell the compiler information about a function (return type and parameters) that it needs to be able to do error checking in compiling the source code.
- ▶ The header file **does not** contain the code for each of the functions; it only contains the prototypes.


# Preprocessor (continued)

- ▶ The file accessed when `#include <stdio.h>` is referenced is `/usr/include/stdio.h`, similarly `#include <stdlib.h>` references the file `/usr/include/stdlib`.
- ▶ The preprocessor also removes comments from the source file, which must be enclosed between `/*` and `*/` [**Note: no single line comments with `//` are permitted in ANSI C**]. Also, the preprocessor for ANSI C is not written to find nested comments, so they are disallowed.
- ▶ When you write comments in code on exams, correct format is required.

# Hello programs – EXIT\_SUCCESS macro

- ▶ Another thing which the preprocessor does is to replace *macros*, which are fragments of code, defined in the source file, or in a header file, with the code which they are defined to represent (Keep in mind that code in a source file is just text, so macros are just chunks of text).
- ▶ These macros, or code fragments, are defined with the define preprocessor directive, as follows:
  - `#define string1 string2`
    - `string1` cannot contain any white space characters (spaces, tabs, new lines), but `string2` can.
- ▶ An example of this is in Version 3 of the Hello program, where you see `EXIT_SUCCESS` in the return statement at the end of the function. This macro is defined in the `stdlib.h` file, and that is why it has been included, using a preprocessor directive, in this version of the program.

# Hello programs – statements

- ▶ In C, as in Java, statements must generally end with a semi-colon.
  - ▶ Remember, though, that preprocessor directives are NOT statements, and are not terminated with a semi-colon!
  - ▶ C is also *case sensitive*, so if we have two variables, named num and Num in a C program, the compiler will treat them as two distinct variables. (This is not a good programming practice).
- 

# More on C programs – declarations and definitions

- ▶ C programs consist of zero or more preprocessor directives, and **declarations** and **definitions** (see explanation on a slide below) of:
  - One or more functions (which may contain variable declarations or definitions), and
  - Zero or more variables declared or defined outside of any function

# More on C programs – main and other functions

- ▶ Notice that all three versions of the Hello program have a main() function. Every C program must have exactly one main function, and program execution always begins in this function. main() does not necessarily have to be the first function defined in the program.
- ▶ Also, notice that, although Java also has a main method, in Java, it is in a class, but C has no classes!
- ▶ In C, technically, we have *no methods*, but only ***functions***. ***Please pay attention to this distinction***. It makes you look bad to other computer science professionals to talk about a “method” in a C program (remember **the correct programming model and terminology** for the language being discussed)! One of the fastest ways to lose the respect of your work–life peers is to use the wrong terminology. It tells them you don’t know what you are doing.

# Functions

- ▶ A number of statements grouped into a single logical unit are called a function
- ▶ REMEMBER: It is required to have a single function 'main' in every C program
- ▶ A function prototype is a function declaration or definition which includes:
  - Information about the number of arguments
  - Information about the types of arguments
  - What type of value the function returns
- ▶ Although you are allowed *not to specify* any information about a function's arguments in a declaration, it is purely because of backwards compatibility with Old C and should be avoided (poor coding style).
  - A declaration without any information about the arguments is *not* a prototype.



# Functions

- ▶ In all cases, C passes arguments to functions by value
  - For example, `int mult_values ( int a, int b);`
  - This insures that the values passed to the function can not be changed in the calling program by the called function.
- ▶ But, what if that's what we would like to have happen?
  - That's when pointers get involved and we use what is called "pass by reference". (Though to be clear – we pass the reference by value.)
  - We'll look at this when we look at pointers in a week or so.

# Declarations and definitions (examples below)

- ▶ **Declaration (*type information only, no value(s)*):** For a variable, it tells the compiler the type of a variable, but not its value. For a function, it tells the compiler the return type, and the number and types of its parameters (parameter names are optional).
  - A variable can only be declared once in a given block in a C program; a function can be declared multiple times in a C program, as long as all of the declarations are consistent (that is, identical with respect to types).
- ▶ **Definition (*type information and value*):** For a variable, it tells the compiler the type of the variable **and the initial value**. For a function, it tells the compiler the return type, parameter types, and **the code** that should be executed (i.e., the statements) when the function is called.
  - A given variable or function can only be defined once in a C program.
  - Note that a definition is *also* a declaration, since it contains type information.

# Declarations and definitions (more)

## ► In C:

- A variable must be declared (but not necessarily defined) before it can be referenced in a non-declarative statement.
- A function must be declared (but not necessarily defined) before it can be referenced in a non-declarative statement (that is, before it can be *called or invoked*) .

# main and other functions (more)

- ▶ Functions consist of one or more blocks (blocks can legally be empty).
- ▶ A block in ANSI C has this form:
  - { /\* left curly brace \*/
  - Zero or more variable declarations
  - Zero or more non-declarative statements
  - } /\* right curly brace \*/
- ▶ **IMPORTANT:** all declarations in the block must precede the first non-declarative statement (i.e., no just-in-time declaration in ANSI C).
- ▶ Nested blocks are valid in C, but nested functions are *not valid* (that is, the compiler will generate errors, and will not produce an executable, if your source code contains one or more nested functions).
- ▶ In C, all functions have “**file scope.**” This means that any function declared in a file can be called from anywhere in the same file, after the point at which it is declared.

# Another example C program

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int total = 0;          /* variable declaration and definition */
    int i;                  /* variable declaration */
    int values[4] = {12, 14, 18, 20}; /* array variable
                                         declaration and definition */
    for (i = 0; i < 4; i++) { /* nested block */
        total = total + values[i];
    }
    printf("The total is: %i\n", total);
    return (EXIT_SUCCESS);
}
```

# Demo

>> and >

.vs files turned into .h files