ECE 220

Lecture x0005

Slides based on material originally by: Yuting Chen, Yih-Chun Hu & Thomas Moon

Recap

- **Last week:**
  - Stack ADT
  - Push/Pop routines
  - Uses for stack
  - MP2 material - RPN notation

- **Reminders/upcoming**
  - Mock Quiz on-going
  - Quiz 1 to be 02/10 - 02/12
  - Assigned reading for the week:
    - **Chapters 11, 12, 13** and **Appendix D** of P&P.

Lesson objectives

- Understand progression from machine code to assembly to higher level languages
- Understand compiled vs. interpreted languages
- Understand static vs. dynamic typing
- Understand basic data types in C and their flavors
- Understand scope, linkage and storage class associated to variables

Introduction to C programming

Why do we need C

- Type a response to the question below (answer need not be full sentences)

- What are three things you dislike about LC3 programming?

  - Unhelpful answer: All of it

  - Helpful answers: Shuffling registers, debugging, etc.

Why you don’t like LC3

- Type a response to the question below (answer need not be full sentences)

- What are three things you dislike about LC3 programming?

  - Unhelpful answer: All of it

  - Helpful answers: Shuffling registers, debugging, etc.

Generations of languages

- First generation: machine code, i.e. 1's and 0's
- Second generation: assembly language, e.g. LC3, x86 ISA, RISC
  - A little piece of history: https://github.com/chrislgarry/Apollo-11/tree/master
- **Third generation**: offering higher-level abstractions, e.g. early: C, FORTRAN, ALGOL and later: Java, Python, etc.
- Fourth generation: *no consensus*, tend to be highly domain specific.

Intro- C – High Level Language

- Developed in the early 1970s by Dennis Ritchie at Bell Laboratories
- Gives symbolic names to values
  - Don’t need to know which register or memory location
- Provides abstraction of underlying hardware
  - Operations do not depend on instruction set
  - Do not need to deal with low level implementations
  - E.g. We can write “`a=b*c`” in C language (in LC-3, there is no single instruction that performs an integer multiplication).

C – High Level Language

- Provides expressiveness
  - Use meaningful symbols that convey meaning
  - Simple expression for common control patterns (if-then-else)
- Enhances code readability
- Safeguard against bugs
  - Can enforce rules or conditions at compile-time or run-time

```
if (isItCloudy)
  get(Umbrella);
else
  get(SunGlasses);
```

Characteristics of C

- *Imperative* vs. *declarative* programming languages

  - In *imperative* programming, you describe the algorithm step-by-step

  - In *declarative* programming, you describe a result or a goal, and you get it via a "black box".

- **C is an ***imperative*** procedural language**

- C programs are ***compiled*** rather that *interpreted*

  - a _____ translates a C program into machine code that is directly executable on hardware

  - interpreted programs are executed by another program, called _____

- C programs are ***statically typed***

  - the *type* of each expression is checked at compile time

Example

In Python you can enter this function line by line into the REPL → interpreted.

```
def silly(a):
    if a > 0:
        print(“Hi”)
    else:
        print(“Hi “ + a)
```

```
>> silly(2)
   Hi
>> silly(-1)
   ERROR!!
```

Interpreter allows a to be whatever.

Error only raised if you hit this line.

This C snippet must be made into a complete program (more on that later) and then compiled using an invocation of a compiler like `gcc`.

```
void silly(int a){
  if (a>0){
    printf("Hi\n");
  }
  else{
    char *name = a;
    printf("Hi %s", name);
  }
}
```

a restricted to be an `int`.

Compiler knows this shouldn’t be permitted; will not compile

Translating HLL programs

Interpreter	Compiler
Program that executes instructions/statements	Program translates statements into machine language
Pros: Easy to debug, make changes, view intermediate results	**Pros:** Executes faster, memory efficient
Cons: Program takes longer to execute	**Cons:** Harder to debug, change requires recompilation
Languages: Python, Matlab	Languages: C, C++, Fortran

Static vs. dynamic typing

Static typing	Dynamic typing
Type of variables are known and/or constrained	Type of variables are associated to their runtime values
Pros: Bugs are caught earlier on, compiler can perform optimizations	**Pros:** Rapid prototyping is easier, more flexibility for programmer
Cons: Programs takes longer to type and require forethought	**Cons:** Errors not caught until runtime, typically slower
E.g languages: C, C++, Java	E.g. languages: Python, MATLAB, Ruby

A first look at C

```c
/* This program is the standard Hello-World in C
 and these lines showcase a 'multiline' comment.
*/

// The below is a macro or preprocessor directive
#include <stdio.h>

// The main function is the entry point to the program
int main(void){

// printf() displays the string inside quotation
    printf("Hello, World!\n");
    return 0;
}
```

Comments can be multiline or single line.

Macros always start with #

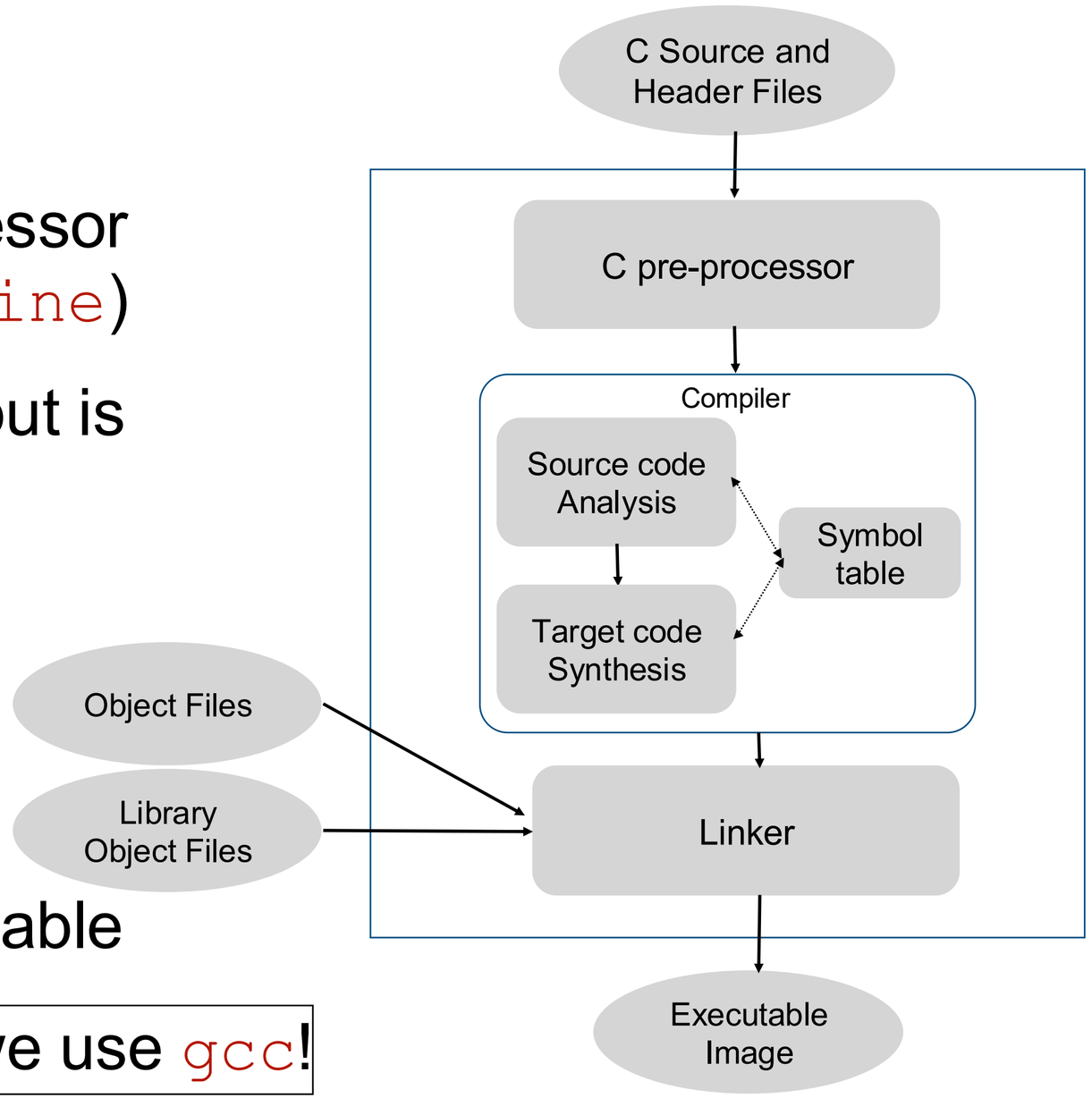
Main function always returns an `int`.

Braces indicate scope

Statements always terminated with a ;

Compilation process

- Preprocessor
  - Macro substitution by C preprocessor directive (eg: `#include, #define`)
  - *Source level* transformation: output is still C code
- Compiler
  - Generates object files
- Linker
  - Combines object files into executable images



On EWS we use `gcc`!

From the textbook - Figure 11.3

```c
// The next two lines are preprocessor directives
#include <stdio.h>
#define STOP 0

/* Function : main
   Description : prompt for input, then countdown
*/
int main(void){
   // Variable declarations
   int counter;    // Holds intermediate count values
   int startPoint; // Starting point for count down

   // Prompt the user for input
   printf("===== Countdown Program =====\n");
   printf("Enter a positive integer: ");
   scanf("%d", &startPoint);

   // Count down from the input number to 0
   for (counter = startPoint; counter >= STOP; counter--)
      printf("%d\n", counter);
}
```

Before compilation copy content of header files into source code.

- <…> header files are standard and in a predefined directory
- “…” header files are in the same directory as the source C file

Before compiling replace all instances of the symbol `STOP` with the value 0.

- Used for values that won’t change during execution

Figure 11.3 – main function

```
// The next two lines are preprocessor directives
#include <stdio.h>
#define STOP 0

/* Function : main
   Description : prompt for input, then countdown
*/
int main(void){
   // Variable declarations
   int counter;    // Holds intermediate count values
   int startPoint; // Starting point for count down

   // Prompt the user for input
   printf("===== Countdown Program =====\n");
   printf("Enter a positive integer: ");
   scanf("%d", &startPoint);

   // Count down from the input number to 0
   for (counter = startPoint; counter >= STOP; counter--)
      printf("%d\n", counter);
}
```

Every C program has a (and only one) function called `main` that returns an integer

- This is the code that is executed when the program starts.

`void` indicates this `main` function takes no arguments

- Advanced usage: pass in *command-line arguments*.

```
int main(int argc, char *argv[])
```

- **Exercise**: In C, what is the difference between `int func()` and `int func(void)`?

Figure 11.3 – local variables

```
// The next two lines are preprocessor directives
#include <stdio.h>
#define STOP 0

/* Function : main
   Description : prompt for input, then countdown
*/
int main(void){
   // Variable declarations
   int counter;    // Holds intermediate count values
   int startPoint; // Starting point for count down

   // Prompt the user for input
   printf("===== Countdown Program =====\n");
   printf("Enter a positive integer: ");
   scanf("%d", &startPoint);

   // Count down from the input number to 0
   for (counter = startPoint; counter >= STOP; counter--)
      printf("%d\n", counter);
}
```

Variables are used as names for data items. Each variable has:

- ***type*** which indicates to the compiler how the data has to be interpreted and/or stored
- ***identifier***, i.e. the name of the variable (case-sensitive, cannot begin with number)
- ***scope***, the portion of code in which data held in memory is accessible via its identifier
- ***storage class***, the duration for which the data is held in memory

Figure 11.3 – I/O commands

```c
// The next two lines are preprocessor directives
#include <stdio.h>
#define STOP 0

/* Function : main
   Description : prompt for input, then countdown
*/
int main(void){

   // Variable declarations
   int counter;    // Holds intermediate count values
   int startPoint; // Starting point for count down

   // Prompt the user for input
   printf("===== Countdown Program =====\n");
   printf("Enter a positive integer: ");
   scanf("%d", &startPoint);

   // Count down from the input number to 0
   for (counter = startPoint; counter >= STOP; counter--)
      printf("%d\n", counter);
}
```

More on these I/O commands & program flow topics next lecture.

Today:

- Using `gcc` to compile on EWS machines
- Data types, scope/storage and basic operations

EWS and `gcc`

- Typical workflow (recommended but not necessary):
  - `ssh` (MacOS/*nix) or VSCode via ssh (Windows) into EWS Machine
  - Navigate to your project folder (use Linux commands like `cd`)
    - Use a text editor (like `vim`, `nano`, etc. but recommend `vim`, try running `vimtutor` to get started) to edit source files
    - Invoke `gcc` with the appropriate *flags* (`man gcc` is your friend)
    - Run/debug executable
- Let us run the previous program.

Demo time: EWS, ssh/FastX, gcc, manpages, linux commands, etc.

Basics of C programs

About variables: data types

- Integers: `short, int, long`
  - Flavors *signed* and *unsigned*
- Floating point: `float, double`
- `char`
- `bool`

Four basic types

Bits	1	8	16	32	64
Types	Bool	Char		Int	
				Float	

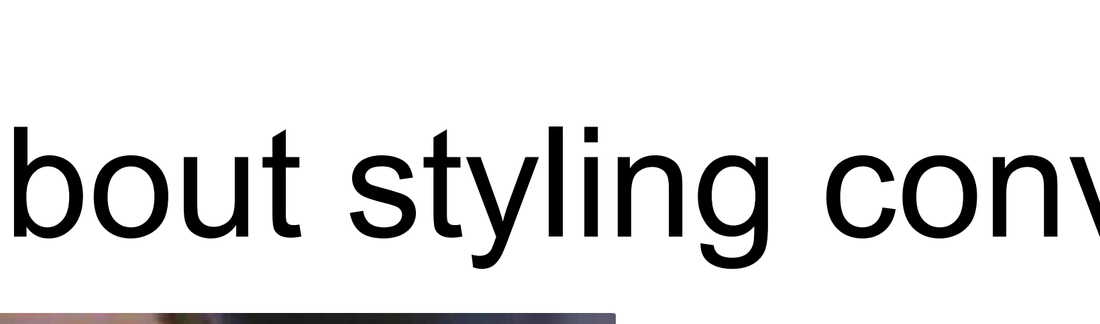
```
/* print different types*/

#include <stdio.h>
#define PI 3.1416

int main()
{
    int i = 3;
    float f = 3.14;
    char c = 'M';
    printf("value of i is %i\n", i);
    printf("value of f is %f\n", f);
    printf("value of c is %c\n", c);
    printf("value of PI is %f\n", PI);
    return 0;
}
```

Called *format specifiers;* more about them next lecture.

Single quote for char!

Note about styling conventions

Adopt one and stick to it:
[Course Website]



More links:

[GNU C Convention]

About variables: scope

- **Scope** of a variable is the duration or portion of the code within which the data it represents in memory is accessible via its identifier
  - Globally available vs. locally scoped

```
int itsGlobal = 0;

int main(){
   /* local to main */
   int itsLocal = 1;
   printf("Global %d Local %d\n", itsGlobal, itsLocal);
   {
      /* local to this block */
      int itsLocal = 2;
      /* change global variable */
      itsGlobal = 4;
      printf("Global %d Local %d\n", itsGlobal, itsLocal);
   }

printf("Global %d Local %d\n", itsGlobal, itsLocal);
   return 0;
}
```

Translation unit: Technical term for a C source file *just before* compilation, i.e. already preprocessed.

About variables: linkage

- **Linkage** describes how *identifiers* can or cannot refer to the same entity throughout the *whole program* **or** single *translation unit*.
  - **None** vs. internal vs. external
- Helps *linker* disambiguate identifiers between translation units.

None: The identifier can be referred to only from the scope it is in. All function parameters and all non-`extern` block-scope variables (including the ones declared `static`) have this linkage

Note: Some concepts in this & following slides are discussed in far more detail than in the textbook. The reason is two-fold: (a) if you ever go online and try reading material on C, you will inevitably run into some of these concepts and technical jargon and (b) while it is okay to sweep things under the rug for the average coder, a good programmer should be aware what exactly is going under the rug before doing the sweeping.

Translation unit: Technical term for a C source file *just before* compilation, i.e. already preprocessed.

Variables: linkage types

- **Linkage** describes how identifiers can or cannot refer to the same entity throughout the *whole program* **or** single *translation unit*.
  - None vs. **internal** vs. external
- Helps *linker* disambiguate identifiers between translation units.

Internal: The identifier can be referred to from all scopes in the current translation unit.

All `static` file-scope identifiers (both functions and variables) have this linkage.

Translation unit: Technical term for a C source file *just before* compilation, i.e. already preprocessed.

Variables: External linkage

- **Linkage** describes how identifiers can or cannot refer to the same entity throughout the *whole program* **or** single *translation unit*.
  - None vs. internal vs. **external**
- Helps *linker* disambiguate identifiers between translation units.

External: The identifier can be referred to from any translation units in the entire program. All non-`static` functions, all `extern` variables (unless earlier declared `static`), and all file-scope non-`static` variables have this linkage.

Variables: No linkage

- **Linkage** describes how identifiers can or cannot refer to the same entity throughout the *whole program* **or** single *translation unit*.
  - **None** vs. internal vs. **external**
- Helps *linker* disambiguate identifiers between translation units.
- Linkage is external by default unless `static` (functions) or `const` (variables) or block scoped.

```c
/* This is prog_part.c */
#include <stdio.h>

void foo(int my_num){ // foo has extern linkage
 int a=10;            // a has no linkage
 printf("Foo got %d", my_num);
}
```

```c
/* This is prog_main.c */
#include <stdio.h>

void foo(int my_num);

int main(void){
    int a_value = 10;
    printf("Main value is: %d\n", a_value);
    printf("Calling foo with %d \n", ++a_value);
    foo(a_value);
}
```

Variables: Internal linkage

- **Linkage** describes how identifiers can or cannot refer to the same entity throughout the *whole program* **or** single *translation unit*.
  - None vs. **internal** vs. **external**
- Helps *linker* disambiguate identifiers between translation units.
- Linkage is external by default unless `static` (functions) or `const` (variables) or block scoped.

`foo` no longer available outside `prog_part.c`

```c
/* This is prog_part.c */
#include <stdio.h>

int a=10;           // A has external linkage now
static void foo(int my_num){ // Internal linkage
 printf("Foo got %d", my_num);
}
```

```c
/* This is prog_main.c */
#include <stdio.h>

void foo(int my_num);

int main(void){
    int a_value = 10;
    extern int a;
    printf("Main value is: %d\n", a_value);
    printf("Calling foo with %d \n", ++a_value);
    foo(a_value);   // Will raise error
    printf("Value of a is: %d\n", a);
}
```

Tells linker `a` is defined elsewhere

Variables: Automatic storage

- A variables *storage class/duration* determines how long data is maintained in memory
  - Can be ***automatic****, static* or *dynamic* (advanced)
- ***Automatic****:* The storage is allocated when the block in which the object was declared is entered and deallocated when it is exited by any means (`goto`, `return`, reaching the end).

Variables: static storage

- A variables *storage class/duration* determines how long data is maintained in memory
  - Can be *automatic,* ***static*** or *dynamic* (advanced)

Yes it is unfortunate. A good reference is available here.

- ***Static****:* The storage duration is the entire execution of the program, and the value stored in the object is initialized only once, prior to the `main` function. All objects declared `static` and all objects with either internal or external linkage have this storage duration.

About variables: storage class

Compare

```c
#include <stdio.h>

void printx(){
   static int x = 0;
   x++;
   printf("value of x is %d \n",x);
}

int main(){
   printx();
   printx();
   printx();
   printx();

   return 0;
}
```

```c
#include <stdio.h>

void printx(){
   int x = 0;
   x++;
   printf("value of x is %d \n",x);
}

int main(){
   printx();
   printx();
   printx();
   printx();

   return 0;
}
```

Next time

- Operators in C
- Basic I/O functions
- Control structures in C
- Debugging with GDB