Introduction to C

Section 1.5 (Kinda)

C Standards

- Though C was originally developed by AT&T Bell Labs, various compilers and standards for the language have been created
  - Some compilers have their own extensions and variations
- **ANSI C** usually refers to **C89** Standard
- Newer revisions include **C99** and **C11**
  - https://en.wikipedia.org/wiki/ANSI_C

What We Already Know

- If know Java, already know a lot of C
- Java is actually based on C
- Syntax is quite similar
  - Blocks of code contained in **{ }**, lines end with **;**, etc.
- Most of the same basic data types (**char**, **int**, **float**, and **double**)

- Same arithmetic operators
  - **+**, **-**, *****, **/**, and **%**
- Same relational operators
  - **==**, **!=**, **<**, **>**, **<=**, and **>=**
- Same shortcut operators
  - **+=**, **-=**, ***=**, **/=**, **%=**, **++**, and **--**

- Mostly same logical operators
  - **&&**, **||**, and **!**
- Same selection constructs
  - **if**, **else**, **else if**, **switch**
- Same loops
  - **while**, **for**, **do while**

What's Different

- C does not have **classes** or **objects**
  - C is not object-oriented
  - C does have **structures** (**structs**)
- C does not have a **String** data type
  - **Arrays of char**s are used to represent strings

- C has **functions** instead of **methods**
  - A method is a **function** inside of an object / class
- Traditionally, C does not have a **Boolean** data type
  - **0** is used for **false** and **1** is used for **true**
  - Newer standards of C do have a sort of Boolean data type

Compiling a C Program

- Basic command for compiling is:
  - **gcc *filename*.c**
  - Produces a binary file called **a.out**
- To give the binary file a name:
  - **gcc -o *name filename*.c**
  - Produces a binary file named ***name***

Running a C Program

- To run a program, put **./** followed by the program's name:
  - ***./program_name***

Basic C Program

- **welcome.c**
  - Prints out a welcome message

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("Welcome to C!\n");

    return 0;
}
```

Comments

- Comments in C are like the multiline comments in Java:
  - **/* … */**
- Traditionally, C does not support the single line comments of Java (**//**)
  - Some newer standards of C do allow **//** for single line comments

- **#include <stdio.h>** is a **preprocessor directive**
  - **#include** comparable to an **import** statement in Java
  - **Header (.h)** files contain references to functions, variables, and constants
  - Header files allow for sharing useful or common items between programs
  - Several standard header files part of **C Standard Library**
  - **stdio** stands for **standard input / output**

main() Function

- **int main(int argc, char* argv[])** begins **main** function
  - Program always starts running in the main function
  - Notice **main()** returns an integer
    - Returning zero means the program exited normally
    - Returning non-zero usually means the program exited abnormally
    - Not really standardized, usually just return 0

- **int argc** is a parameter that stands for **argument count**
  - It's the number of command line arguments passed into program when run
- **char* argv[]** stands for **argument vector**
  - Essentially an array of strings of the command line arguments
  - Note that sometimes this is written as **char** argv**, as we'll learn the two are equivalent

- In Java, signature for main method is quite specific
- In C, there's two signatures for main function specifically allowed:
  - **int main(int argc, char* argv[])**
  - **int main(void)**
- Other variations may be accepted as well, such as:
  - **void main(void)**

putchar()

- Most basic way to print to the screen is **putchar()**
- Simply prints a single character to the screen
  - **putchar('A');**
- Does not print strings

puts()

- Most basic way to print a string is **puts()**
  - Stands for put string
  - Similar to System.out.println() in Java
  - **puts("Hello, world!");**
- Note that puts() cannot be used to print variables (other than strings)

printf()

- In C, printing is primarily done with the **printf()** function
- printf() can be used to simply print a string, like puts()
- If want to print variables, need to use printf()
- printf() can also format the values being printed

- But printf() can also be used to
- printf()'s first argument is a string:
  - **printf("some text here");**
- Cannot directly print arithmetic values or variables:
  - **printf(5 + 4);**
  - **printf(x);**

- To print values, need to insert format specifiers into the string
  - Format specifiers act as placeholders and specify how to format the value when its printed
  - Basic format specifiers are comprised of **%** followed by a letter indicating the type
    - For example: **%d**, **%f**
- The values to be substituted into the string need to be listed after the string

- The values to be substituted into the string need to be listed after the string
  - **int x = 10;**
  - **float y = 19.99;**
  - **printf("x is %d and y is %f", x, y);**
    - Displays as: **x is 10 and y is 19.99**
- Note that both the number and types need to match up!

- Format specifiers can contain information between % and the letter to control how the value is formatted
- For example, putting an integer between % and the letter specifies a width in which to print the value
  - **printf("%5d", 100);**
  - Useful for aligning columns of text
- Can also use negative numbers to left justify
  - **printf("%-5d", 100);**

- When printing a real number, format specifier can also control how many decimal places to round to / display
  - **float x = 123456.7891234;**
  - **printf("%.2f", x);**
  - Prints rounded to two decimal places

Format Specifiers

- Quite a few different format specifiers
- A few primary ones used more than others
  - Most common **%d** for int, **%f** for float and double, and **%s** for string
- Some to display value in a particular way
  - **%x** to display a whole number in hexadecimal, **%e** to display a real number in scientific notation

Format Specifier	Use
%d	signed integer
%u	unsigned integer
%f	float, double
%x, %X	integer in hex
%c	char
%s	string
%p	pointer
%e	double

Escape Sequences

- printf() does not automatically move the cursor to the next line after printing the string
  - Often end the string with **\n** to start a **newline**
- **\n** is an escape sequence for the **newline character**
- Other escape sequences are **\t** for **tab** and **\"** for a **double quote character**

Example

- **printing.c**
  - Uses a variety of printf() formatting

Basic Data Types

- C has four basic data types:
  - **char** – holds a single character, can also be used to hold a small integer
  - **int** – whole number
  - **float** – real number
  - **double** – real number, greater precision than float

Data Type Qualifiers

- Additionally, C has four qualifiers that can be applied to some data types:
  - **short**
  - **long**
  - **signed**
  - **unsigned**

short and long

- **short** and **long** can be applied to int to create a smaller or larger integer type
  - **short int si = 5;**
  - **long int li = 400000000;**
- Or used on their own like a data type
  - **short s = 5;**
  - **long l = 400000000;**

signed and unsigned

- **signed** and **unsigned** can be applied to integer types to indicate to affect the range of values they can hold
- Specifically, a **signed** data type can hold negative number
- An **unsigned** data type can only hold non-negative values
- Data types without a qualifier are usually signed
  - char can be signed or unsigned (depends on hardware and system)

- Example:
  - **unsigned int ui = 4000000000;**
  - **signed int si = -30;**
  - **unsigned char uc = 255;**
- Can also be used in conjunction with **short** and **long**:
  - **unsigned long ul = 18000000000000000;**

Data Type Sizes

- In C, the size of a data type is system dependent
  - C standard says things like int must be at least 16 bits
- Minimum and maximum values of integer types are constants in the **limits.h** header file
- Minimum and maximum values for real number types are constants in **float.h** header file

sizeof() Operator

- **sizeof()** is an operator that tells us how many bytes of memory a particular data type uses
- Useful for telling how much memory a type uses on a particular system
- Syntax:
  - **sizeof(*data_type_name*)**
- Example:
    - **unsigned int size = sizeof(double);**

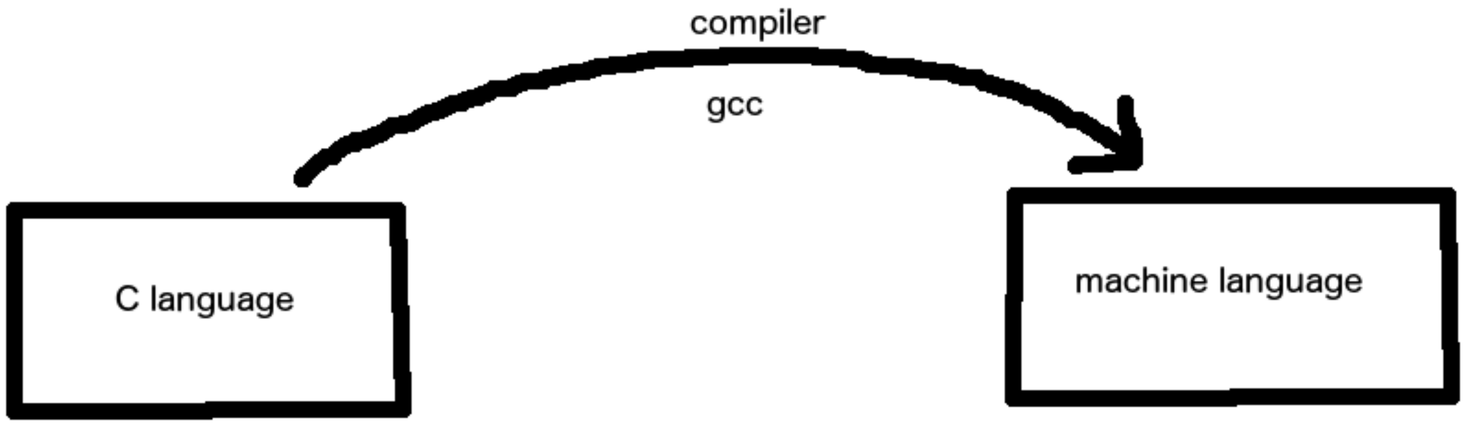
Example

- **type_sizes.c**
  - Prints out the sizeof(), min, and max values for each data type
  - Constants contained in limits.h and float.h
  - sizeof() tells how many bytes for each type
  - Some less common format specifier types used

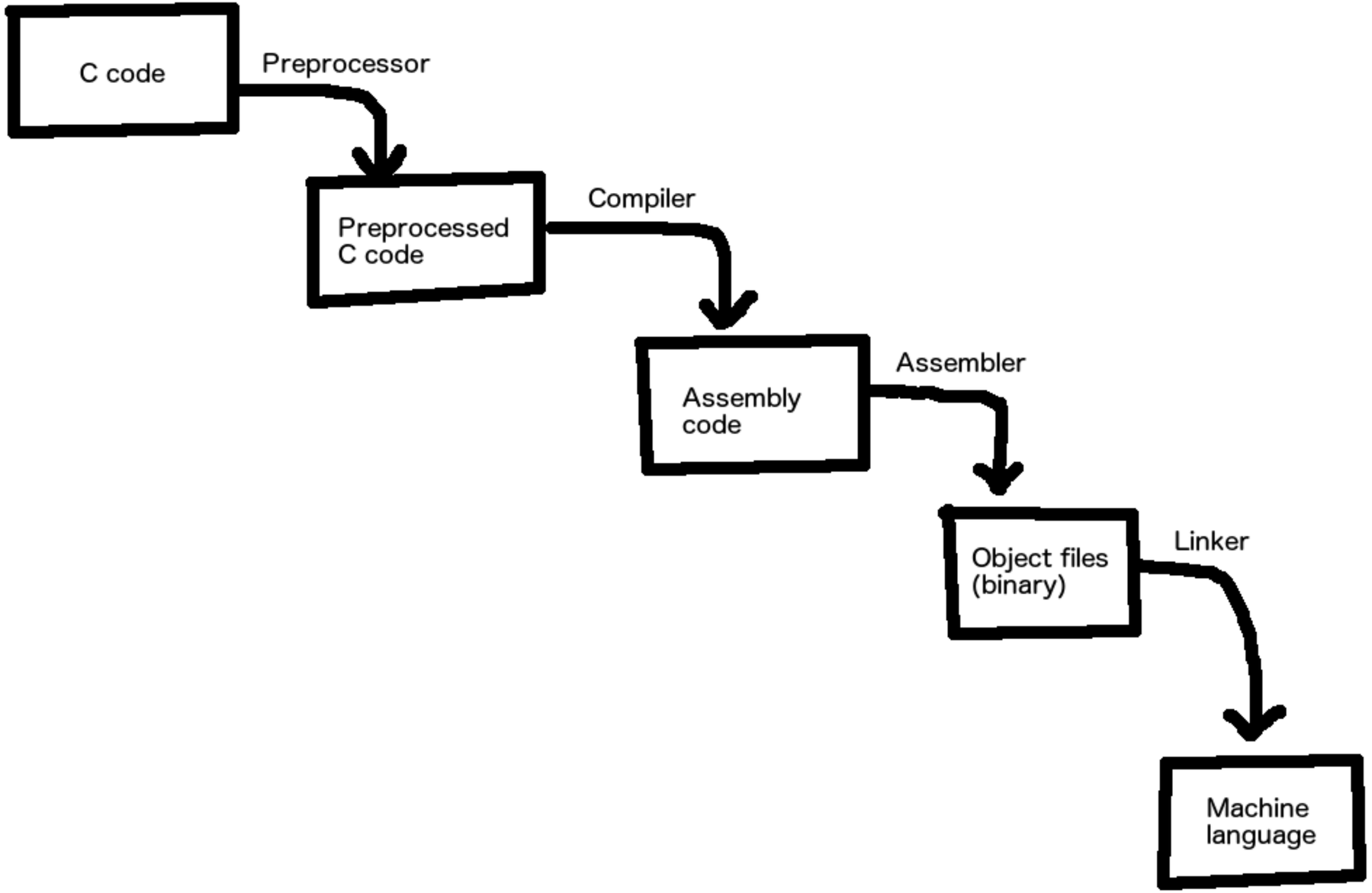
C Compilation Process

- When a C program is compiled, it's translated from **C language** to **machine language** by the compiler

- There's actually several internal intermediary steps going on this process
  - Exact steps can vary depending on the specific compiler


compiler
gcc
C language
machine language

- **Preprocessor** handles preprocessor directives (commands that start with #) in the C code
- **Compiler** translates preprocessed C code into **assembly language**
- **Assembler** translates **assembly code** into **machine language** in **object files**
- **Linker** links together **object files** into the completed **executable file**


C code
Preprocessor
Preprocessed C code
Compiler
Assembly code
Assembler
Object files (binary)
Linker
Machine language

Getting User Input

- In Java, we use the Scanner and nextInt(), nextDouble(), etc. to get input from the user
- C does not have a Scanner class
- Instead, several different functions can be used to read input from the user

getchar()

- Most basic way to get input is **getchar()**
- Reads in a single character from the user
  - **int c = getchar();**
- Useful at times, but not what we will be using most of the time

gets()

- A simple way to read a string from the user is **gets()**
  - **gets(*stringVar*);**
  - Waits for user to type something in and fills *stringVar* with the string

scanf()

- To get input from the user, often use **scanf()**
- **scanf()**'s first parameter is a string containing a pattern that it's scanning for
- String contains format specifiers much like **printf()**'s string
- Variables to be filled with input values are listed after the string

- Example:
  - **int number = 0;**
  - **scanf("%d", &number);**
- Note that for basic variable types, **&** needs to be put in front of the variable name when being filled
  - Learn why and what **&** means later on
- Format specifier needs to match type being gotten!

- Can also be used to get multiple values at once
  - **int x, y, z;**
  - **scanf("%d\t%d\t%d\n", &x, &y, &z);**
  - Useful for processing lines of a file that have some pattern (separated by tabs, commas, etc.)

- scanf() has many format specifiers as well
- Some the same as with printf()
- Important to note that the scanf() format specifier for doubles is **%lf**

Format Specifier	Use
%d	int
%u	unsigned int
%hu	unsigned short
%hhu	unsigned char
%lu	unsigned long
%f	float
%lf	double
%c	char
%s	string

C with scanf()	Java with Scanner
scanf("%d", &*var*)	*var* = input.nextInt()
scanf("%lf", &*var*)	*var* = input.nextDouble()
scanf("%s", *var*)	*var* = input.next()

- **basic_input.c**
  - Gets two integers from the user
  - Adds them together
  - Prints the sum

- **multi_input.c**
  - Gets three integers from the user with one scanf() call
  - Adds them together
  - Prints the sum

Symbolic Constants

- A common way to create constants is with **#define**
  - Another example of a **preprocessor directive**
- Preprocessor replaces the constant with the associated value during preprocessing
- Syntax:
  - **#define *CONSTANT value***

- Example:
  - **#define PI 3.1415926**
  - **#define MAX 100**

if Statements

```
if (boolean_expression)
{
    /* code that runs */
}
```

- Beware of classic error in C is to use **=** when **==** is intended
  - **if (x = 100) { /* ... */ }**
- Usually compile time error in Java
  - But C code generally compiles, but doesn't behave as expected, treated as putting 100 into x

else Statements

```
if (boolean_expression)
{
    /* code runs when true */
}
else
{
    /* code runs when false */
}
```

else if Statements

```
if (boolean_expression0)
{
    /* code runs when be0 true */
}
else if (boolean_expression1)
{
    /* code runs when be1 true */
}
else
{
    /* code runs when both false */
}
```

- **selection.c**
  - Gets an integer from the user
  - Uses if / else if / else to print whether zero, positive, or negative

while Loops

```
while (boolean_expression)
{
    /* code that repeats */
}
```

do while Loops

```
do
{
    /* code that repeats */
}
while (boolean_expression);
```

for Loops

```
int i;

/* ... */

for (i = initial_vale; i < final_value; i++)
{
    /* code that repeats */
}
```

- Traditionally, C required that variables in a function be declared at the beginning of the function
- Furthermore, variables were not allowed to be declared in for loop headers
  - **for (int i = 0; i < 100; i++)**
  - **int i;**
  - **for (i = 0; i < 100; i++)**
- Newer C standards neither of these are an issue

math.h

- If want to use math functions in C, need to include **math.h**
  - **#include <math.h>**
  - On Linux, may need to use -lm switch when compiling as well
    - **gcc -lm *file*.c**

- Contains functions comparable to methods in Java's Math class
  - Exponents: **pow()**, **sqrt()**
  - Transcendental: **log()**, **log2()**, **log10()**, **exp()**, etc.
  - Trigonometric: **sin()**, **cos()**, etc.
  - Rounding: **ceil()**, **floor()**, etc.

- **math_functions.c**
  - Includes math.h
  - Uses a few math functions

Random Numbers

- Random numbers are generated with the **rand()** function
  - Add **#include <stdlib.h>** when using **rand()**
- **rand()** returns a random integer between **0** and **RAND_MAX**
  - **RAND_MAX** is a constant defined in **stdlib.h**

- Before **rand()** can be used, **srand()** needs to be called
- **srand()** seeds the random number generator
  - Just needs to be called once before use of **rand()** in a program
- **srand()** takes as input an unsigned integer

- Commonly, the current time is used as the seed to **srand()**
- **time(NULL)** returns back the current time
  - To use **time(),** need to include **time.h**
    - **#include <time.h>**

- So to generate a random number:
  - **srand(time(NULL));**
  - **int r = rand();**
- Often mod **rand()** to control its range:
  - **int r = rand() % 100;**
    - Gives a range between 0 and 99, inclusive

- **random.c**
  - Includes stdlib.h and time.h
  - Seeds the random number generator with time(NULL)
  - Creates some random numbers

Making Functions

- To make our own functions, simply create them above the **main()** function
- Traditionally in C, required to create the function above any attempted use of the function
  - Newer standards may allow function to made after **main()**

```
#include <stdio.h>

int function(int x)
{
    /* code */
}

int main(int argc, char* argv[])
{
    /* code */
    function(5);
}
```

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    /* code */
    function(5);
}

int function(int x)
{
    /* code */
}
```

```
returnType functionName(dataType parameter)
{
    /* code */
}
```

- Valid return types are the data types and **void**
- For functions with no parameters, put the **void** keyword between the parentheses:
  - **int functionName(void) { /* ... */ }**

- **function.c**
  - Creates a function
  - Uses the function in the main method

Function Prototypes

- If we need to refer to or use a function before it has been defined, we can use a **function prototype**
  - Kinda like declaring a function, but not yet defining what it does
  - Similar to how we have to declare a variable before we try to use the variable
- Sometimes put **function prototypes** before **main()** and then function definitions below **main()**

```
#include <stdio.h>

int do_something(int x);

int main(int argc, char* argv[])
{
    /* code */
    do_something(5);
}

int do_something(int x)
{
    /* code */
}
```

- A **function prototype** is basically the first line of a function followed by a semicolon
  - ***returnType functionName(dataType parameter);***
- Often put function prototypes in header files as well
  - Allows function to be shared between multiple programs

- **prototype.c**
  - Makes a function prototype
  - Uses function in main()
  - Defines function after main()

Arrays

- Syntax for creating an array:
  - ***dataType arrayName*[*size*];**
  - This allocates the memory
    - Notice there's no new like in Java

- Traditionally, C requires the size be a constant known at compile time (i.e., not a variable or calculation)
  - Newer standards might not require this
  - C does have ways to do dynamic memory allocation, more details later in course
- Can also use initializer lists as well

- Examples:
  - **int numbers[100];**
  - **double data[10];**
  - **int array[] = {5, 2, 3, 4, 9};**

Header Files

- Header files often contain constants and function prototypes
  - Useful for sharing functionality amongst multiple programs
- Several standard header files, but we can make our own

- **circle.h**, **circle.c**, and **circle_prog.c**
  - **circle.h** is header with constant and function prototypes
  - **circle.c** contains definitions of functions
  - **circle_prog.c** includes the header and uses the functions
  - To compile multiple files, just list them to gcc:
    - **gcc circle.c circle_prog.c**