MPATE-GE 2618:
C Programming for Music Technology

Unix, Commands and Compilation

Reading

- *Programming in C*, Chapters 1 and 2
- Optional: *Absolute Beginner's Guide to C*, Chapters 1-4, pp. 1-46

Course Goals

- Learn to
  - precisely express a problem
  - decompose a problem into smaller, easily solvable parts.
  - develop algorithms to solve problems or parts of a problem
- Learn to express algorithms in C (programming)
  - This is the "C programming" task
- Learn about
  - data structures
  - common algorithms (e.g. sort, FFT)
- Program audio applications

What is an algorithm?

- Many programs implement an *algorithm* for doing something
  - Sort, search (a list)
  - Transform (FFT), mix, scale, measure (audio)
- An *algorithm* is a precise, unambiguous procedure for producing certain results (outputs) from given data (inputs). It is a method for accomplishing a task or solving a problem.

Programming Platforms

- Assignments developed for Unix systems
  - in particular Mac OS X (which is based on Unix)
- All assignments (and libraries/APIs) are *cross-platform* (i.e. portable to other platforms)
- Quick Poll: how many students are using
  - Mac OS
  - Windows
  - Linux

Terminal

- Command-line interface to Unix
- Open a Terminal window
  - Finder: Applications/Utilities/Terminal.app
  - Launchpad: Other/Terminal.app
- You see Bash Shell Prompt
  - machine_name:directory_name user_name$
  - This is the Bash Shell "prompt" when it is ready to accept and process a "command line"
- "pwd"
  - Prints the current directory

What is the Bash Shell?

- It's a command line interface that lets you interact with the OS. That's what you're using when you type into your Terminal window.
- It is also a powerful interpreter language, and "scripts" can be written in Bash (just as with Matlab).
  - See Learning the bash Shell

History of Unix

- In 1960's MIT, GE and Bell Labs were collaborating to create a new operating system "Multics"
  - This project was mired in problems and was abandoned
- Bell Labs wanted to continue, but there was no funding
  - However, they could work on a typesetting program for publishing papers ("Troff")
  - To run Troff, Dennis Richie and Kent Thomson created a new operating system, UNIX, a pun on Multics

Unix and Compilers

- Bash commands permit you to use the Unix operating system.
- Mac OS X is built on a Unix (BSD) core
- C programs are compiled
  - create an executable (app)
- We will use GNU C/C++ compiler (gcc or g++)
  - Need to download Xcode on Mac
  - On Mac gcc/g++ actually runs the clang compiler
  - Learn how in first Lab session

Unix Commands

- Traversing the Unix file system – directory commands

cd	changes current directory to home directory
cd `<dir>`	changes current directory to specified directory
cd ..	go up one directory level
ls	list the contents of present working directory
pwd	show path to current directory
mkdir `<dir>`	make a new directory
rmdir `<dir>`	delete a directory

Commands continued

- File commands

cat <filename> prints contents of file

less <filename> same as cat but shows one screenful at a time

cp <src> <dest> copy file <src> to <dest>

mv <src> <dest> move or rename <src> to <dest>

touch <filename> creates an empty new file called <filename>

vi <filename> opens a new file in the vi editor. Also vim. (But never use this or you will instantly be a dinosaur).

Directories, Filename and Pathname

- Unix file system has nested Directories (or folders)
  - /Users/srq/Documents/NYU/Code
  - This is *path* or *pathname*
- Directories contain files with Filenames
  - main.c
- In Unix, files are designated by a Pathname
  - Full: /Users/srq/Documents/NYU/Code/main.c
  - Relative: ./main.c or ../Code/main.c
    - "." is "current directory"
    - ".." is "parent directory"

Pathnames or Filenames with spaces

- "Escape" with "\" as "\ "

/Users/srq/Documents/NYU/My\ C\ programming\ class

- I recommend:
  - Don't use spaces in pathnames!
  - Instead use "-" or "_"
  - My_C_programming_class

Some Unix questions

- How does the shell know where to find the programs you type?
  - See what your path is…
    ```
    echo $PATH
    ```
  - Typically "." is NOT in your PATH
    - Prevents spoofing
  - Hence, use full or relative pathname: ./a.out
- Why don't I see all my files when I type **`ls`**?
- What are command flags?
  ```
  ls -aF
  ```
- How do I find out more about a command?
  ```
  man <command name>
  ```

Stdin, Stdout and Pipes

- What is standard input and output?
  - Programs (and bash shell) read from stdin, write to stdout
- Unix supports concept of re-direction and pipes
- Read input from a file

  `<command>` **<** `<inputfilename>`
- Save output to a file

  `<command>` **>** `<outputfilename>`
- Create a "chain" of commands

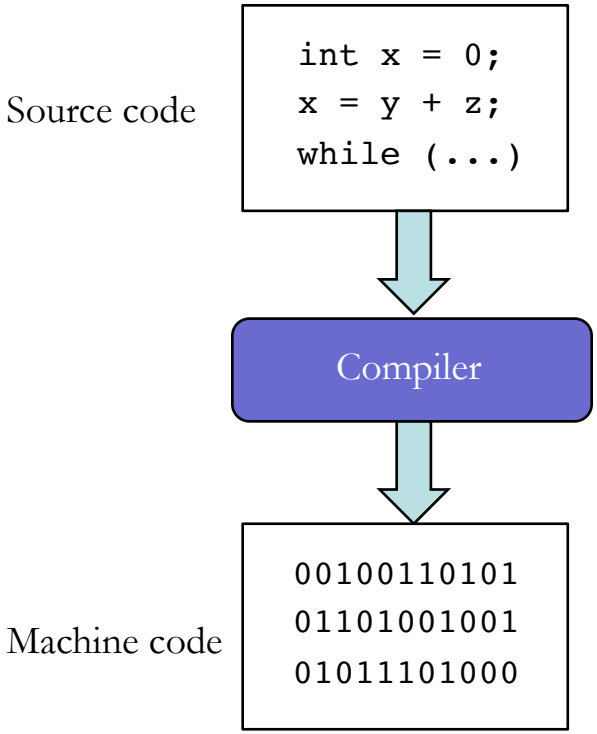
  `<command1> | <command1>` **>** `<outputfilename>`

Programming: Some history

- *Machine operation codes (opcodes or machine code)*
  - Early computers could only be programmed in terms of binary numbers that corresponded directly to specific *machine opcodes* and locations in the computer's memory.
  - Not very human friendly.
- *Assembly language*
  - Enabled the programmer to work on a higher level, using symbolic names for opcodes.
- High Level Language
  - Languages like FORTRAN (FORmula TRANslator) and "C" permit programmers to create code without being aware of the architecture of a particular computer.
  - Permits code to be portable!
  - The *compiler* translates the program into a form that a specific computer can use.

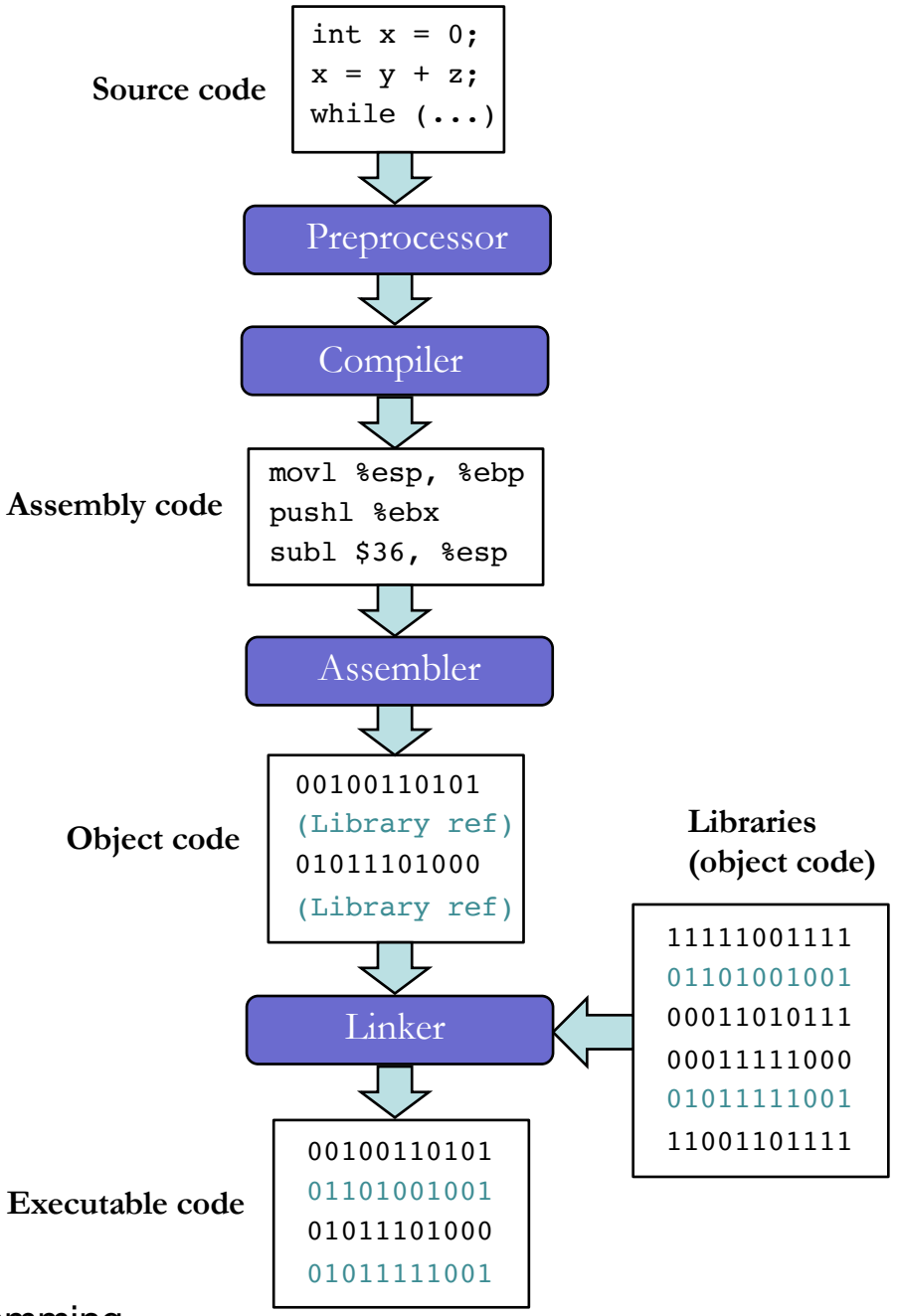
Compiling a C program: Basic outline

- Source code
  - Human-readable instructions (your program)
  - Format is plain text
- Source code is then translated by a *compiler* into machine code (binary)
  - You can then run this program on your specific machine.



Compiling Details

- Preprocessor
  - "*Includes*" code pointed to by "#include <stdio.h>
- Compiler
  - C-code is translated into *assembly code*
- Assembler
  - Assembly language produced by compiler is translates into *object code*
  - Assembly instructions are turned into *opcodes* (machine language codes that specifies a particular operation for the CPU)
- Object code
  - Machine code that is not directly executable (.o files). Addresses are relative.
- Linker
  - The linker takes object files and combines them into an executable
  - External libraries used by your program are linked in at this stage.
  - Relative addresses are resolved to absolute addresses
  - **`ld`** is the Unix linker or *loader*



Example C program

```
/* This is a comment */
// This is also a comment

#include <stdio.h>

int main (void)
{
   printf("Hello World!\n");
   return 0;
}
```

Structure of a C Progam

- Include statements #include <stdio.h>
- Function declaration int main()
- Function definition (the declaration and body)

Structure - 2

```
#include <stdio.h>

int main (void)
{
   printf("Hello World!\n");
   return 0;
}
```

include statement

function declaration and definition

Compiling a C program

- Create a file containing source code
  - "helloWorld.c"
- Compile it
- Run it

```
$ gcc helloWorld.c
$ ls -F
 a.out*   helloWorld.c
$ ./a.out
Hello World!
$
```

Variations

```
#include <stdio.h>
int main (void)
{
   printf("Hello World!\n");
   printf(”Programming is fun.\n");
   return 0;
}
```

Variations

```
#include <stdio.h>
int main (void)
{
   printf("Programming is %d times more fun.\n",
       100);
   return 0;
}
```

Variations

```
#include <stdio.h>
int main (void)
{
   int n;
   n = 100;
   printf("%s %d %s\n",
       "Programming is", n, "times more fun");
   return 0;
}
```

Useful gcc flags

- The `–Wall` flag directs gcc to output all warnings
- The `–o <filename>` flag indicates the executable filename
  - default executable name is `a.out`

```
gcc –Wall errors.c –o errors
```

Printing Output

- To print to standard output, use `printf()`
  - This function prints a formatted string (a series of text characters) and zero or more variables to stdout (the terminal).
- Include the header file stdio.h to use printf in your program.

printf()

- Using `printf()`

```
printf(formatString, variables...);
```

- The variables are printed in the order they are passed to `printf` and interpreted as the type specified in the string.
- The character % always precedes a type specifier (also known as a conversion specifier).
  - Note that type specifier and variable type must match!
  - `printf("%s %i\n", "The variable value is:", i);`
- If you want to print an actual percent sign, use "%%"
- http://en.cppreference.com/w/c/io/fprintf

printf() formatting

- Width and precision
  `%[-][#]<width>.<precision>`

  `%3.2f`
  `%-2.5f  etc.`
- '`-`' means left justify
- '`#`' can mean show the prefix (e.g. `0x` for a hex number) or show decimal point for floats.

Getting Started - 1

- Install gcc and other Unix tools on your computer
  - If you are using Windows, you must install Cygwin.
  - If you are using MacOS, you might have to install Xcode
  - If you are using Linux, you shouldn‘t have to install anything.
- Download and install an editor
  - Sublime http://www.sublimetext.com/
  - Atom https://atom.io/

Getting Started - 2

- Setup your computer
  - See the "Development Environment" document (to be covered in the first Lab session) to help you set up the shell environment on your computer
- Get acquainted with the Unix environment
  - Learn how to navigate the Unix directory structure; experiment with basic Unix commands
- Get acquainted with your editor
  - Create helloworld.c
  - Compile and run it