CS240: Programming in C

Lecture 2: Hello World!

PURDUE UNIVERSITY

Introducing C

- High-level programming language
- Developed between 1969 and 1973 by Dennis Ritchie at the Bell Labs for use on UNIX OS
  - Dennis Ritchie received the Turing Award in 1983 with Ken Thomson for creating UNIX
- First standard for C, 1989, c89
- Current standard is c11, replacing c99

C at a glance: Types and functions

- **Types** (set of data with predefined characteristics):
  - Basic: Characters, integers, floating points, pointers
  - Hierarchies: structures, unions
- **Control-flow** expressions: if-else, switch, while, do, for, break, continue, goto
- **Functions** (a piece of code that pefroms operations on some input, returns an output and is reusable):
  - Can return basic types, structures, or pointers
  - Can be called recursively

C at a Glance: Variables

- Symbolic name for information
- Can be
  - internal to a function
  - external to a function
    - visible within the file
    - across files
- Can be passed to functions by value or by reference
- Can be declared in a block-structure mode which limits its visibility to the block
- Can have memory if declared static

C at a Glance: Preprocessing

- Macros substitutions
  - Macro: brief abbreviations for longer constructs
- Inclusion of other files
  - Usually header files
- Conditional compilation
  - Support for different architectures
  - Unique inclusion of a file
  - Support for debug

C at a Glance: `libc`

- No direct access to composite objects such as strings, sets, lists, arrays
- No storage facilities other than static and stack
- No input/output facilities

- **Developers get access to some of these because of a library made available with the language, `libc`**

How is C different from Java?

- Java is platform independent, compiled once and then runs on different platforms without being recompiled
  - There is a caveat … you need a JVM
- C is not platform independent, but the code can be written to be portable on different platforms, needs to be recompiled

Software portability

- **Ideal of portable program**: can be moved with little or no extra investment of effort to a different platform/computer that differs from the one on which the program was originally developed
- **Reality**: Writing a program in Standard C does not guarantee that it will be portable (because of differences among C implementations), but close enough

Some portability issues

- Representation issues
  - Endianess
  - Integer representation
  - Size
  - Alignment
- Standard libraries
  - Sometimes the functions that have the same functionality have different names/parameters on different platforms;
  - Include the right header files
  - Link with the right libraries

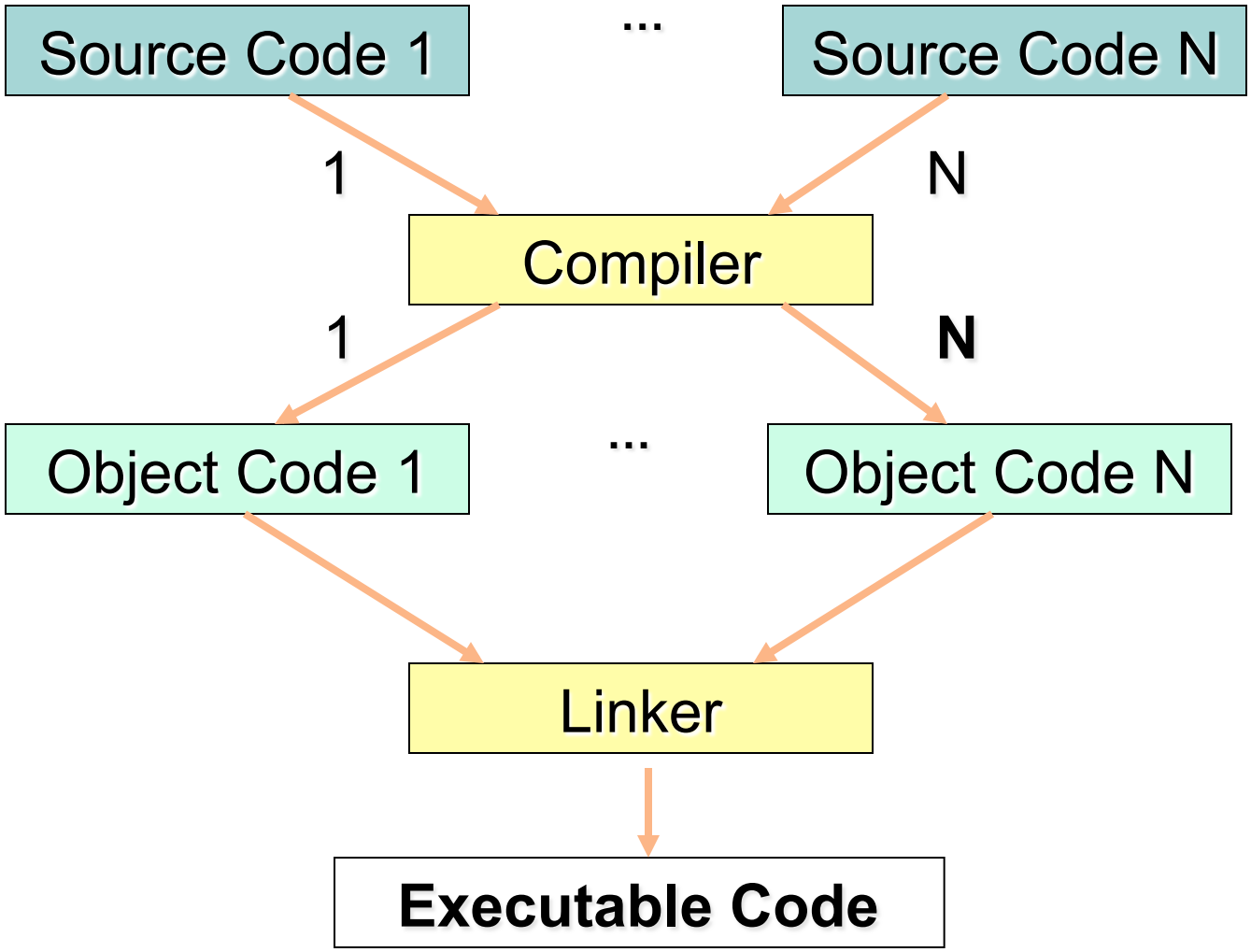
Source code

- Source files: **`file_name.c`**
  - Contain declarations and/or definitions of functions
- Header files: **`file_name.h`**
  - Contain declarations and macro definitions to be shared between several source files
  - C pre-processor transforms the program before compilation: replacing macros and including files
    - It is called automatically by the compiler

Object code

- **Object code**:
  - Relocatable format machine code
  - Usually not directly executable
- **Library**:
  - File containing several object files used as a single entity in the linking phase of a program
  - It is indexed, so it is easy to find symbols (functions, variables) in them
- **Executable**: directly executable, launches a program, runs in its own space, contains a **`main`**
  - ELF for Linux
  - Mach-O for Mac
  - PE for Linux

From source code to executable code



Static libraries

- Libraries that are linked into the program during the linking phase of compilation
- Each process has its own copy of the static libraries is using, loaded in memory
- Executable files linked with static libraries are bigger

Examples:

Unix: XXX.a

Mac: XXX.a

Windows: XXX.lib

Shared (dynamic) libraries

- Linked into the program in two stages
  - **Compilation**: **`linker`** verifies that all the symbols (functions, variables) required by the program, are either linked into the program, or in one of its shared libraries. The object files from the dynamic library are not inserted into the executable file
  - **Runtime**: **`loader`** checks out which shared libraries were linked with the program, loads them to memory, and attaches them to the copy of the program in memory
- Only one copy of the library is stored in memory at any given time
- Use less memory to run our programs, the executable files are much smaller

Linking

- The linker needs to know how to find the specified libraries
- Default path usually /usr/lib
- Or use –L option to specify directory
- -l option needed for include files

- Windows: xxx.dll
- Mac: xxx.dylib
- Linux: xxx.so

Loader

- Loader needs to know how to find those dynamic libraries
- Environment variables

  LD_LIBRARY_PATH
- Loader looks in LD_LIBRARY_PATH *before* the compiled-in search path(s), and the standard locations (typically / lib, /usr/lib, …)

libc

- C Standard library – is an interface standard which describes a set of functions and their prototype used to implement common operations
- Libc – is the implementation of the C Standard library on UNIX systems

libc is linked in by default as a shared library

Let's speak C – Hello World

```
#include <stdio.h>
int main() {
    /* every program must have a main */
    printf("Hello world!\n");

    return 0;
}
```

```
gcc -c hello.c
```
means compile

```
gcc -o hello hello.o
```
means link

```
OR
gcc hello.c
```
compile and link

Main function

- Every C program has to have a main
- It has to be declared *int main* for portability
  - Returning 0 means the program exited OK
  - Return value is interpreted by the operating system
  - Without a return statement, undefined value is returned
- The complete signature for main()

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

Use define

```
#include <stdio.h>
#define HELLO "Hello World!\n”

int main() {
    printf(HELLO);
    return 0;
}
```

More C…

```
#include <stdio.h>

int main() {
    int c;

    c = getchar();
    while(c != EOF) {
      putchar(c);
      c = getchar();
    }
    return 0;
}
```

Practice : Exam questions

- Remember that at the exam you will not have a computer so try to answer these questions first with pen and paper and then try it by running the program
- How would the screen look like if you type enter abc enter
- Same question considering that line c = getchar(); was removed from the while loop.

Practice: Learn from mistakes

- Take the hello program and misspell the name of printf, observe what happens when you compile
- Remove the include file, observe what happens when you compile

Making our coding life easier…

- Complex projects have many source and header files compiled in object files and then linked in an executable, sometime linking with other external libraries.
  - How to compile/link in an organized and efficient way?
- **make** utility
  - Determines which pieces of a large program need to be recompiled and issues commands to recompile them.
  - Can be used with any programming language (not only C) whose compiler can be run with a shell command.
  - Not limited to programs: documentation, distribution.

Running **`make`**

- Uses a file, the default name is ‘’Makefile`` that describes the relationships among files in the program and provides commands for updating each file.

```
make
make –f Makefile_name
```

- **`make`** uses the Makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

An example of a Makefile

- **C files: main.c, command.c, display.c, utils.c**
- **H files: defs.h, command.h**

This is a tab

```
edit : main.o command.o display.o utils.o
	gcc -o edit main.o command.o display.o \
	utils.o
main.o : main.c defs.h
	gcc -c main.c
command.o : command.c defs.h command.h
	gcc -c command.c
display.o : display.c defs.h
	gcc -c display.c
utils.o : utils.c defs.h
	gcc -c utils.c
clean :
	rm edit main.o command.o display.o utils.o
```

Variables and implicit rules

- It is not necessary to spell out the commands for compiling the individual C source files, `make' can figure them out: it has an "implicit rule" for updating a `.o' file from a correspondingly named `.c' file using a `gcc -c' command.

- To simplify writing make files, one can define variables:

```
OBJS = main.o command.o display.o utils.o
…
edit : $(OBJS)
        gcc -o edit $(OBJS)
```

Example: Makefile for hello

```
all: hello

hello : helloworld.o
        gcc -o hello helloworld.o

helloworld.o : helloworld.c
        gcc -c helloworld.c

clean:
        rm hello helloworld.o
```

Readings for this lecture

K&R Chapter 1 and 2