COMP2017 / COMP9017 Week 7 Tutorial

Compiler pipeline, Signals, Makefile and Shared library

Take a little bit of time in this tutorial to fill out the mid semester feedback survey on Canvas

The C Compiler pipeline

Let's explore what the compiler does behind the scenes when we create a more complex program. The following is a make file which allows us to script our build system. Make is a simple utility that can determine when small pieces of our program require recompilation.

`Makefile` - builds the program

```
CC=gcc
CFLAGS=-g -std=c11 -Wall -Werror
TARGET=tasks

.PHONY: clean
all: $(TARGET)

clean:
	rm -f $(TARGET)
	rm -f *.o

list.o: list.c
	$(CC) -c $(CFLAGS) $^ -o $@

tasks.o: tasks.c
	$(CC) -c $(CFLAGS) $^ -o $@

tasks: tasks.o list.o
	$(CC) $(CFLAGS) $(LDFLAGS) $^ -o $@
```

`tasks.c` - the scaffold code for the task list application

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
```

```c
int main(void) {
        // ...
        return 0;
}
```

`list.c` - the implementation of the circular linked list

```c
#include "list.h"
// Initializes an empty circular linked list.
void list_init(node* head) {
        head->next = head;
        head->prev = head;
}
// ...
```

`list.h` - function prototypes for a circular linked list

```c
#ifndef LIST_H
#define LIST_H

typedef struct {
        void * data;
        node* next;
        node* prev;
} node;

// Initializes an empty circular linked list.
void list_init(node* head);

// Inserts given node before the head.
void list_push(node* head, node* n);

// Inserts given node after the head.
void list_append(node* head, node* n);

// Removes the given node from the list.
void list_delete(node* n);

// Returns whether the list is empty.
int list_empty(node* head);

#endif
```

Preprocessor

Your code is first processed through the C preprocessor. This executes all of the preprocessor directives.

You can examine the raw output of the preprocessor by calling it directly:

```
$ cpp tasks.c
```

Or by instructing the compiler to only perform the preprocessing step.

```
$ gcc -E tasks.c
```

This output is very helpful when debugging the problems related to macros and other preprocessor utilities.

Code Generation and Assembly

The -c flag on gcc asks the compiler to preprocess the C code, generate assembly and finally assemble the result into an object file. The object files contain machine code - assembly in binary format for the target CPU. We need to create an object file for every translation unit in our source code (every .c file is a translation unit). You can ask the compiler to stop after assembly generation with the following command:

```
gcc -S -g -std=c11 -Wall -Werror list.c
```

This command produces `list.s` - the assembly generated from `list.c`. `gcc` calls the assembler behind the scenes to turn this into machine code for object file.

You can also extract assembly from object files with objdump. Assembly files have two different syntax that are equivalent in functionality. `objdump` defaults to the AT&T syntax but can also output the Intel syntax.

```
gcc -c -g -std=c11 -Wall -Werror list.c
objdump -M intel -S list.o

list.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <list_init>:
...
```

Linker

Now we have two compiled object files, one for each translation unit. The linking stage merges these object files together to generate the executable. Behind the scenes, `clang` calls the `ld` linker to perform this task.

Since we often need to use variables and functions that are declared in another translation unit, C defines the concept of linkage. The job of the linker is to connect these translation units together.

- A variable or function has `internal` linkage if it is defined in the current translation unit.
- A variable or function has `external` linkage if it is defined in another translation unit.
- Any variable or function that is declared static has internal linkage, it is good practice to declare every variable or function as static unless it needs to be accessible from another translation unit.

`extern`, `static` and linkage

`extern`

When the `extern` keyword used within the a c source file, it notifies the compiler to link to a variable or function of the same name within the global memory space.

`source1.c`

```
int GLOBAL_VAL = 0;
```

`source2.c`

```
extern int GLOBAL_VAL;
```

In the example shown, source1.c will declare and initialise the variable while source2.c during the compilation step will link to the variable defined in source1.c .

`static`

static variables have file scope and internal linkage. These variables are not exposed to other files. Static modifier will also effect how the variable is used within scope as well.

```
int f() {
    static int i = 0;
    i++;
    printf("%d\n", i);
}
```

only accessible within the function if defined with static in a function

- If we were to call f() multiple times, what would be the output?

Pre-Tutorial Questions

1. -E preprocessor directive preprocessing
2. avoid the problem of double inclusion when dealling with the include directive

Question 1: C Preprocessor

- What does the `#include` directive do?
- What are include guards and when should they be used?

We have seen how the `#define` directive can be used to create compile time constants. The `#define` directive can also be used to create macros.

```
#define PI 3.14
#define NUM 42
#define STR "String"
#define MIN(a, b) ((a < b) ? (a) : (b))
#define MAX_BUFFER 1024
```

Similar to the `#define` directives, macros are substituted into their call site in a very similar manner to text search and replace. Why are the extra brackets around a, b and a < b necessary in the macro definition for `MIN`? For example what happens with `MIN(a++, 1))`

value of a will increase each round it is called

1. #include

The #include directive will include the contents of a file into the current source file.

2. #define

#define preprocessor will be a preprocessor variable that can be used within your program. It will be evaluated during the compilation process.

To build the executable, the linker must perform two main tasks:
1. Symbol resolution: The purpose of symbol resolution is to associate each symbol reference with exactly one symbol definition. (i.e. global variable, or a static variable)
2. Relocation: Compilers and assemblers generate code and dtat sections that start at address 0. The linker relocates these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location.

Question 2: Declarations, Definitions and linkage

```
int x; definition, internal
extern int y;   declaration, external
static int z;   definition, external

int add(int a, int b); declaration, external
extern int multiply(int a, int b); declaration, external

int add(int a, int b) { definition, external
  return a + b;
}

static int subtract(int a, int b); declaration, internal

static int subtract(int a, int b) {definition, internal
  return a - b;
}

int main(void) {
  return 0;      definition, external
}
```

- Which of these are declarations and which are definitions?
- Classify the linkages in the above declarations as internal or external.
- Which definitions are accessible from another translation unit in the above C file?
- What happens if the linker can't find a function that has external linkage? undefined symbol error
- Header files often contain only declarations. There is nothing stopping us from putting definitions into the header as well. When would this be useful?

definition: allocated memory for the variable
declaration: internal linkage but can also be accessed out of this file

1. static defined locsal variables do not lose their value between function calls. In orther words, they are global variables, but scope to the local function they are defined in.
2. static global variables are not visible outside of the C file they are defined in.
3. static functions are not visible outside of the C file they are defined in.

Make Files

Construction

You are able to create a scriptable build system that allows for assembly and building of more complex C projects. The `make` command, coupled with a makefile allows our compilation and linking steps to be scripted and provides the build modules to be assembled later. Although make is commonly used for C programs, it can be use for a wide range of utilities.

This is coupled with a the `make` program which will by default look for a `makefile` or `Makefile` file within the directory you are in. Within a make file you will specify a make target. A make file will check files specified in the target (`dependencies`) for changes prior running the command. If a change has been detected, the command will be executed.

Defining a target

When using the make utility you will need to follow it with a target or by default it will utilise the first target defined. The first target defined is typically called `all`. Alternatively you can redefine this by modifying the `DEFAULT_GOAL` value.

Defining a target (pattern)

```
<target_name>: [dependencies]
  <command>
```

Example:

```
hello: hello.c
  gcc hello.c -o hello
```

Defining variables

Withing a make file we can create variables in a similar manner to `bash` variables. Common pattern with makefiles in C is to create a variables specifying the compiler command and compiler flags

Pattern:

```
<variable>=<value>
```

Example:

```
CC=gcc
CFLAGS=-Wall -Werror
DEBUG=-fsanitize -g
```

Automatic Variables

GNU Make has a specification of symbols that you can use within your make file. GNU Make Manual

Tutorial Questions

Question 3: Constructing a makefile

Your friend has started forgetting the compiler flags that are used and what files to bring together for their project. Your friend has uploaded the project on Ed for you to access. Construct a makefile for their project so they do not forget how to build their program with different configurations.

- Build their executable which is composed of `main.c`, `stack.c` and `stack.h`
- Produce a target that allows for debug (printf) statements, use a preprocessor option at compile time
- Create a target that produces a object file for `stack.c` and `stack.h`
- `clean` his project and remove all files that were compiled

You can refer to GCC's Preprocessor options

Question 4: Testing the project

Your friend seems to be making a lot of mistakes in their code and desperately requires some kind of unit tests for their stack. Construct a unit test module that will check for the following:

- Stack can be initialised
- Elements can be added to the stack by using the push function
- pop functionality works as intended by removing and returning the element at the top of the stack
- Stack can be deallocated
- Can retrieve the size of the stack
- Stack functions can handle null data

Afterwards, add another target within your make file for executing tests and enforce this check to be executed as part of your final build process.

Question 5: Creating your own collections library

Over time you will start utilising common patterns. Especially with C, instead of having to rewrite the same structures and functions repeatably you can write a library. In this instance you can get started creating your own collections library.

First Task is to write a ring-buffer queue.

```
struct rbuf_queue;

struct rbuf_queue* rbuf_new(size_t size);
void rbuf_enqueue(struct rbuf_queue* queue, void* element);
void* rbuf_deque(struct rbuf_queue*);
void rbuf_delete(struct rbuf_queue*);
```

Question 6: Using CMocka

Utilise the `cmocka` static library that is available as part of the additional resources. The resources should include a simple unit test and comments on how to write your own test cases with `cmocka`.

Use this library to write test cases for your collections and construct a makefile to help with constructing and testing of your collections.

You may refer to the API documentation here: CMocka API.

Question 7: Static Library

This is one kind of library that you can create. This doesn't differ much from `.o` object files. Extend your collections library with the linked list data structure you wrote earlier and combine the multiple object files into one library.

Hint: use `ar` command to create a library from multiple object files created from `gcc`.

```
$ gcc -c mylib.c -o mylib.o
$ ar rcs mylib.a mylib.o
```

./shared_library.sh

- What text file still needs to be exposed to our programs using this library? Without it, our program does not know what types and functions are declared.

Question 8: Creating a shared library (Extension)

Most of the code you have written has been utilising a shared library (.so) file located on your system. We are able to create our own share libraries that can be utilised on our system.

When compiling a shared library you **should** utilise PIC (Position Independent Code) which adds a layer of indirection of a process's memory for the shared library.

```
gcc -shared -fPIC -o <name>.so <files>
```

- Why is the `PIC` flag important for shared libraries? Discuss this with your tutors

Question 9: Dynamic Loading (Extension)

In the event you need to dynamically load a library you can utilise the functionality within `dlfcn.h`. The `dlopen` function allows loading of a library at run time.

Open usage

```
void* lib = dlopen("mylib", RTLD_LAZY);
...
```

- When and why would you use dynamic loading?
- What problems do you forsee with dynamic loading?