THE UNIVERSITY OF SYDNEY

COMP2129 Week 5 Tutorial

The compiler pipeline, linked lists.

Notices

We welcome your feedback and thoughts about the course. Please complete the mid-semester survey, available on eLearning.

Exercise 1: The C Preprocessor

The C preprocessor is the first pass of the compiler over source code, and provides utility as a macro language, conditional compilation, and inclusion of header files. A good way to think of the C preprocessor is smart text-replacement which runs over the C code before the compiler sees it.

The most important directives of the preprocessor are:

- `#define mymacro value` means that every occurance of `mymacro` in the C program is replaced with `value`.
- `#define mymacro(a, b) value` replaces every occurance of `mymacro(v1, v2)` in the text with `value`, such that parameters `a` and `b` in the `value` are replaced with `v1` and `v2` respectively.
- The directive `#include <file.h>` includes a header file stored in specific locations (such as `/usr/include`). Instead of using `<...>`, you can use `#include "file.h"` if you want that a header file should be included from the current directory. (This will search the include path if no such file is found in the current directory)

Create your own header file `myheader.h` that contains the following macro defintiions:

```
1 #define mymin(a, b) ((a < b) ? (a) : (b))
2 #define LOOPBOUND (4)
```

These are just like what would be written in a C file. In fact, what the preprocessor does whenever it sees an `#include` directive is (more or less) copies the contents of the included file into the C file.

Create a C program, **myprog.c** which uses your header file:

```
#include <stdio.h>
#include "myheader.h"

int main(int argc, char **argv) {
  int i, j;
  for (i = 1; i <= LOOPBOUND; i++) {
    for (j = 1; j <= LOOPBOUND; j++) {
      printf("%d ", mymin(i, j));
    }
    printf("\n");
  }
  return 0;
}
```

Finally, create a **Makefile** which expresses the dependency of **myprog** on the files **myheader.h** and **myprog.c**:

```
.PHONY: all clean

all: myprog

clean:
	rm -f myprog

myprog: myheader.h myprog.c
	gcc -Wall -W -o myprog myprog.c
```

myprog.c will be used in the following exercise.

Exercise 2: The C Compiler

A C-compiler has several compilation steps, i.e., preprocessor, compiler, assembler, and linker. The C-Language itself consists of two different language levels. The first level is the pre-processing language which is a text-language and is implemented by the command line tool **cpp**. The text-language permits the definition and use of text-macros, including text files, and conditional compilation. The other level is the actual C-Language which is a simple imperative lanugage that has "no" garbage collection nor object-oriented programming features. Note that a C-program is a collection of functions and global variables (i.e. imperative programming paradigma) and it is compiled "statically", (i.e. before the execution of a program in contrast to Python that is interpreted whilst it is executing). The first invoked function of a program is the **main()** function.

Compile your program with the pre-processor only to see the manipulations of the text-macros, i.e., type one of the following two lines (they do the same thing):

```
$ cpp myprog.c
$ gcc -E myprog.c
```

After invoking the pre-processor for **myprog.c**, the result will show up on standard-output. As you can see, the defines in the program will be resolved and the include directives for **stdio.h** and **stdlib.h** paste the header files verbatim into the program. Note that a header file is a file that store function definitions, constant definitions and external variable definitions.

In the next step we compile the C-program to an assembly program by using following options

```
$ gcc -Wall -W -pedantic -std=c99 -S myprog.c
```

which produces an assembly program **myprog.s** in the current directory. The option **-S** triggers the compilation from C to assembly rather than producing an object with the assembler. Have a look at the assembly code of your program that is stored in **myprog.s**. The assembly code of the Intel processor has simple arithmetic and jump instructions and registers that can store temporary values.

With the option **-c**,

```
$ gcc -Wall -W -std=c99 -c -o myprog.o myprog.s
```

the assembler is invoked after the actual C-compiler. The assembler produces an object file. Note that the object file cannot be executed. To execute the object file, we need to link it with standard libraries to generate an executable.

With the commandline tool **objdump** the object file can be queried. Note that object files are binary files and cannot be directly inspected by a text editor. For example,

```
$ objdump -s myprog.o
```

gives the machine code of the program, plus some symbol tables.

The final step in the compilation is the linking, i.e., the object files of the program are linked with the standard library to an executable. With option **-o**,

```
$ gcc -Wall -W -std=c99 -o myprog myprog.c
```

the executable myprog is generated. The pre-processor, actual C-compiler, assembler and linker is automatically invoked behind the scene. After generating the executable, the program can be executed

in the current directory by typing, **`./myprog`**.

Exercise 3: Linked Lists: Queue

This exercise is to implement a queue data structure using a linked list. Since all of the linked lists nodes will be dynamically allocated, the queue will be able to grow arbitrarily large.

Recall that the *queue abstract data type* is a collection of objects which is accessed in a FIFO (first in, first out) manner. The main operations on the queue are *push*, which adds an element to the queue, and *pop*, which removes and returns an element from the queue. A third method which will allow the safe use of pop is the *isempty* method, which returns a boolean. (C has no boolean type, so returning an `int` is fine).

The following struct definitions and function prototypes are available from Piazza and Blackboard as **`queue.h`**. Download them now, and start working on implementing the required functions. Your implementation should go in a file **`queue.c`** which does not contain a **`main()`** function, so that other code that you write can include this queue implementation.

A description of the queue follows.

The queue elements are stored as a doubly-linked list, in a series of **`Node`** structs. The Queue itself is a pair of pointers, to the front and back of the doubly-linked list.

```
1  struct Node {
2    int data;
3    struct Node *next, *prev;
4  };
5
6  struct Queue {
7    struct Node *front, *back;
8  };
9
10 // Alias the struct Queue * to a short type (Queue) so that
11 // the caller is presented with a simple interface.
12 typedef struct Queue * Queue;
```

Notice that the type **`struct Queue *`** is typedef'd to **`Queue`**, which allows the code to present a simpler interface to the caller. They can pass the queue around as if it were a reference, and not worry about the fact it might be a struct or a pointer. It also means the caller won't accidentally be copying-by-value the **`struct Queue`**, which would cause the queue to become inconsistent.

Now we have five functions to implement, in order to get a bare-minimum queue. The are listed below.

`Queue queue_create(void)` allocates space for a new, empty queue, and hands back a reference to the caller. You should make sure to set the **`front`** and **`back`** pointers here to **`NULL`**, since the queue does not contain anything at this point.

`int queue_isempty(Queue)` returns 1 if the queue is empty, and 0 otherwise. This is used by the caller to know when to stop calling **`queue_pop`**. Since the **`front`** and **`back`** pointers are used to keep track of queue elements, this should just check if either is **`NULL`**.

`void queue_push(Queue, int)` allocates space for a new queue element, stores the **`int`** argument inside it, and inserts it onto the back of the queue. Be careful that you treat the case where the queue is empty differently from the case where the queue already contains elements.

`int queue_pop(Queue)` "pops" the first element from the queue, and returns the value of the data stored inside it. It should also free the memory used for the relevant **`struct Node`**. Be careful that you treat the case where the queue contains one element differently from the case where the queue contains many elements.

`void queue_delete(Queue)` frees the queue, as well as any elements left inside the queue at the time of deletion.

Once you have implemeted these functions in the file **`queue.c`**, make a new file called **`queue_test.c`** and try creating, pushing, popping, and deleting the queue. You can compile the files together using a single command:

```
1    $ gcc queue.c queue_test.c -o queue_test
```

or by using a **`Makefile`** with a dependency, which will compile **`queue.c`** and **`queue_test.c`** both to object files before linking them together.

Finally, use the **`valgrind`** utility on your testing code to make sure there are no memory leaks.

Exercise 4: Extension: Deque

Deques, or *double-ended queues* are queues which can insert or remove from both ends. These are often used in work queues, where jobs can fail. For example, imagine a server farm somewhere like YouTube which converts videos. When new videos come in, they get added to the back of a work queue, and workers pop jobs off the front of the queue and work on them. Sometimes, workers die, and a supervisor program notices, and pushes the job they were working on back onto the *front*, rather than the *back*, of the queue.

Extend the queue from the previous exercise into a **`Deque`**, with functions **`push_back`**, **`pop_back`**, **`push_front`**, and **`pop_front`**. Thoroughly test your code to make sure it works.

Exercise 5: Extension: Generics

C has no generics. The best that C has is the *void pointer* (**`void *`**), which is a pointer to an area of memory which doesn't make any claims about what type that memory could hold. Callers of functions cast whatever they want to store to a **`void *`** before storing it, then cast it back whenever they recieve a value back.

Modify the queue or the deque to store void pointers as the datatype, rather than integers. Then make testing code which gets the new queue to store strings. Make sure you turn on the **`-W -Wall -std=c99`** options in the compiler, and get rid of all compiler warnings by using the right typecasting.

Exercise 6: Extension: Size

Occasionally queues need to be asked what size they are. The current implementation has no way of knowing the size of a queue. Add a size function by either iterating over all queue elements to determine the size, or by storing the size as an extra field in the **`struct Queue`**, and keeping it updated during push and pop operations.

What are the benefits and drawbacks of each strategy? How much extra memory do you use by storing the size? Is it significant?