CSCI 350: Getting Started with C

Written by: Stephen Tsung-Han Sher
June 12, 2016

Introduction

As you have been informed, your work with *Pintos* will be almost exclusively with *C*. Since you have taken **CSCI-103** and **CSCI-104** in *C++*, *C* is not too foreign of a language for you to pick up. Nonetheless, there are still a few confusing differences when transitioning to *C*.

This guide serves to help you understand some differences between *C++* and *C* for you to start working on your project.

Contents

1 Printing to Console

The all time favorite function in *C++* is `std::cout`; unfortunately the `iostream` library is unavailable for *C*.

In *C*, you will need to use `printf` to print to the console. `printf` may be a bit confusing at first, but after a few practices you'll understand it without problem.

1.1 Simple `printf`

If you want to just print out text, simply call the `printf` function with a string as the parameter:

Source:

```
// stdio.h for printf
#include <stdio.h>
int main(void)
{
    printf("Hello␣World\n");
    return 0;
}
```

Output:

```
~$ Hello World
```

Note that `endl` is no longer available, so you'll need to make sure you use `\n` for an endline operation.

1.2 Advanced `printf`

Suppose you want to add in numbers, floats, pointers and whatnot, in *C++* you will do something similar to:

Source:

```
#include <iostream>
int main(void)
{
    float foo = 3.50;
    int* fooPtr = &foo;
    std::cout << "Hello␣World␣" << foo << "␣" << fooPtr << std::endl;
    return 0;
}
```

Output:

```
~$ Hello World 3.50 0x08048000
```

In *C* you will not be able to concatenate console outputs this easily. Instead you'll need to use a `%specifier` in your string and the specific variable you want to output as latter arguments in respective order. Formally the `%specifier` prototype is:

```
%[flags][width][.precision][length]specifier
```

However you usually won't need to worry about the flags, width, precision, nor length. Here is a short example:

Source:

```
// stdio.h for printf
#include <stdio.h>
int main(void)
{
    float foo = 3.50;
    int* fooPtr = &foo;
    printf("Hello␣World␣%f␣%p", bar, fooPtr);
    return 0;
}
```

Output:

```
~$ Hello World 3.50 0x08048000
```

Note that the order of the arguments after the string must match the order of your `%specifiers`. If you don't you'll get a compiler errors regarding some kind of mismatched variable type. Here's a list of the specifiers you'll most likely use in Pintos, a full list can be found at cplusplus.com.

Specifier	Output	Example
`d` or `i`	Signed decimal integer	`350`
`x`	Unsigned hexadecimal decimal	`15e`
`f`	Decimal floating point	`3.50`
`s`	String of characters	`CSCI350`
`p`	Pointer address	`0x08048000`

2 Dynamic Memory Allocation

You will not be able to use the keyword `new`; instead you will need to use `malloc`. `malloc` takes in the number of bytes of memory you want to request and returns you a pointer to the beginning of the section of memory you requested. `malloc` is often used in conjunction with `sizeof`, which returns the number of bytes of the data type you pass in as the argument:

```
struct myStruct* foo = malloc(sizeof(struct myStruct));
```

And instead of `delete`, you use `free`. Simply pass in the pointer of the block of memory you wish to free.

```
//Memory allocate
struct myStruct* foo = malloc(sizeof(struct myStruct));
//Free the memory allocation
free(foo);
```

To use `malloc`, make sure to `#include "threads/malloc.h"`.

3 Structs & Classes

Structs and classes are paramount to object-oriented programming languages such as *C* and *C++*. The key difference between *C++* and *C* is that classes are not supported by *C*.

3.1 Structs

Structs work effectively the exact same way in *C* as it does in *C++* except one syntactical difference. Consider the following struct (which is a struct you will get intimately familiar with in Pintos):

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list.
       */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif
```

```
    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
};
```

In *C*, whenever you want to use the `thread` struct, you will need to prepend the word `thread` with the word `struct` to indicate that it is a struct:

```
#include "threads/threads.h"
#include "threads/malloc.h"
#include <list.h>
// Defining a function that takes a struct as a parameter
void threadFunction(struct thread myThread, struct thread* threadPtr)
{
    // do something with myThread
}
int main(void)
{
    //Normal instantiation of a struct
    struct thread myThread;
    //Dynamic instantiation of a struct
    struct thread* newThread = malloc(sizeof(struct thread));
    //Passing a struct as an argument
    threadFunction(myThread, newThread);
    return 0;
}
```

It is also important to note that the order in which you define the members of a struct is the order in which the compiler will place these members on the memory. This is particularly important for the unsigned `magic` member at the bottom of the thread struct definition. The `magic` member is used to determine the boundaries of the struct, thus if you define members below the definition of `magic`, you will get a lot of page faults that you will not be happy with.

tl;dr: Do NOT define anything below `unsigned magic`.

3.2 Classes

As mentioned before, classes are not supported in *C*. Thus if you use a function associated with a struct, this function is called from a global scope. One quick workaround is to pass in the pointer of the struct you wish for the function to act on:

```
struct MyStruct
{
    int member;
}

void MyStruct_SetMember(struct MyStruct* ms, int m)
{
    ms->member = m;
}
```

Since the function exist in a global scope, it is usually a good practice to prepend the name of the struct on the function name in order to indicate which struct this function intends to act upon.

For example, look at the list implementation under `src/lib/kernel/list.h`; every function acts upon either the list struct itself or on one of the list's elements, all taken in as pointers. also note how every function begin with `list_`.

```
/* List traversal. */
struct list_elem *list_begin (struct list *);
struct list_elem *list_next (struct list_elem *);
struct list_elem *list_end (struct list *);

struct list_elem *list_rbegin (struct list *);
struct list_elem *list_prev (struct list_elem *);
struct list_elem *list_rend (struct list *);

struct list_elem *list_head (struct list *);
struct list_elem *list_tail (struct list *);
```

4 Use of Void & Aux for Function Parameters

4.1 Void Parameters

In *C++*, if you have a function with no arguments, you simply omit any declaration of arguments:

```
void MyFunction();
```

In *C*, you'll have to indicate that a function's arguments are void:

```
//Declaration
void MyFunction(void);
//Usage
int main(void)
{
    MyFunction();
    return 0;
}
```

Defining a function with no arguments indicates that this function takes an unspecified number of arguments, which can produce unpredictable behavior.

4.2 Auxiliary Parameters

In some functions you will use in Pintos, you will notice a `void* aux` as a parameter.

```
//In src/lib/kernel/list.h
struct list_elem *list_max (struct list *, list_less_func *, void *aux);
struct list_elem *list_min (struct list *, list_less_func *, void *aux);
```

The `aux` parameter is an extra parameter in which its usage varies depending on the function. The behavior of this extra parameter is usually defined in the function's documentation. If you do not want to use this extra parameter, you can pass in a `NULL`, which is usually the case.