Project Part 1: Background Information

Overview – Your programming environment

Read the following discussion regarding C programming before you being Part 1 of your project. You will need this information to complete Part 1 of the project. Consult with your Open Learning Faculty Member (OLFM) if you have any questions.

For the project, you will need to use Linux.

- For Mac users, terminal windows can be used.
- For Windows users, you need to install a Linux operating system: e.g., Ubuntu (http://www.ubuntu.com/). There are two different ways to install Ubuntu system on a Windows computer system:
    i. You can install Ubuntu on your computer as a second operating system, or install Ubuntu like an application using "wubi" software. Then you can select either Ubuntu or your original Windows system at the bootup time. Please refer to the Ubuntu support pages for step-by-step instructions and videos on how to install Ubuntu.
    ii. Alternatively, you can install VirtualBox (http://www.virtualbox.org) on your computer, and install Ubuntu within VirtualBox. VirtualBox is a general-purpose virtualizer.
- **Note:** You do not have to use any Integrated Development Environment (IDE) or debugger.
- Use a text editor, e.g. nano (http://www.nano-editor.org/) or vi, to make programs.

Objective – Why do we use C programming language?

The purpose of Part 1 in this Project is to introduce and provide you with some experience and comfort in using C programming language. Most system programs are written in C, not C++, for fast execution. For example, the kernel of Linux is written in C.

For reference:

Holmes, S. (1995, January). C Programming [Online course notes]. Copyright 1995 by The University of Strathclyde, Glasgow, Scotland. Retrieved from http://www2.its.strath.ac.uk/courses/c/

Pros and cons of C programming language

Pros:

- Fast execution
- Easy to handle memory ⇨ Good for system programming
- Bit operation

Cons:

- Complex concepts of pointer, type conversion, and memory allocation
- **Note:** The project in this course does not rely on those complex concepts much. For the most part, **we will use common syntaxes of C and Java.**

Differences between C and Java

C	Java
Procedural: • No class • Common data: global variables • Abstract data type: struct, union	Object oriented: • Class • Common data: instance variables • Abstract data type: class
Micro approach: • Individual utility libraries	Macro approach: • Utilities included in language itself
Reference type variable: • Pointer	Generally no reference, but objects include the concept
Call by value, call by reference	Call by value mostly, call by reference for objects
Compiling: one file at a time; linking	Compiling: cross-reference

Exercise program: **`welcome.c`**

Each program consists of:

- include, ; import in Java
- define for constants and constants in Java ; final variable declaration for
- global variable declaration statements ; instance variables in Java
- main and other functions ; methods

```
// it is like import statement in Java
#include <stdio.h>      // standard i/o; this include statement is like import statement in Java.
#include <stdlib.h>     // standard library


#define SIZE 128        // constant definition; it is like final variable declaration in Java.


int test = 5;           // global variable; it is like a public instance variable in Java.


int main(int argc, char* argv[])    // or char **argv
                                    // char* may be used for strings.
                                    // argc - # of arguments including a.out
{
      int   i;
      char  name[32];


      printf("Welcome to C programming world!\n");


      printf("%d arguments:\n", argc);    /* %d means an integer
            */
      for (i = 0; i < argc; i++)
            printf("\t%s\n", argv[i]);    // %s means a string from the next argument


      name[0] = 'C'; name[1] = 'o'; name[2] = 'm'; name[3] = 'p'; name[4] = 'u';
```

```
name[5] = 't'; name[6] = 'e'; name[7] = 'e'; name[8] = '\0'; name[9] =
'r';
printf("The length of %s is %d.\n", name, strlen(name));    // what
will be printed?
}
```

Compiling

To compile the program, you will use the following statements:

```
$ gcc welcome.c                or    // it will create a.out
$ gcc welcome.c -o welcome           // it will create welcome instead
of a.out
```

Running

To run the program, use the following:

```
$ ./a.out
$ ./welcome
```

Basic data types

- `char` `unsigned char` `// unsigned variables use the left most bit`
- `short` `unsigned short` `// which is used as a sign (+/-) in signed variables`
- `int` `unsigned int`
- `long` `unsigned long`
- `float` `unsigned float`
- `double` `unsigned double`
- **There is no Boolean type in C. Any non-zero value is considered as TRUE, and zero value is considered as FALSE.**
- Array data type for any data type
- Pointer data type for any data type: E.g., `int*`, `char*`, …

Operators

The basic operators you will need include the following:

- `+, -, *, /, ++, --, >, <, >=, <=, =, ==, !=, &&, ||,` ...

  operand1 **&** operand2 — bitwise AND operation

  operand1 **|** operand2 — bitwise OR operation

  operand1 **^** operand2 — bitwise XOR operation

  operand1 **>>** *n* — shift rightward; the *n* left most bit[s] will be filled with 0

  operand1 **<<** *n* — shift leftward; the *n* right most bit[s] will be filled with 0

  ~operand — bitwise NOT operation; one's complement

  `&`operand — the address of the first byte for the variable in the main memory

  `*`operand — pointer to the address stored in the operand, i.e., the value stored in the address

- Implicit and explicit type conversion

Header files

- Include constants and data types
- For example, `/usr/include/stdio.h` includes constants and data types that are used for functions in the standard input/output library
- In a program,
  - `#include <stdio.h>  /* /usr/include/stdio.h */`
- `<...>` is searched from `/usr/include`
- `"..."` is searched from the current working directory
- **Note:** The include statement is like the import statement in Java.

`main()` function

There should be one main function like main method in Java programs.

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

`argc` contains the number of arguments that are given to the program when it is executed, including the name of the program itself

argv[] array of `char*`

`char*` points to an address which contains a **string** that is a list of characters ending with `'\0'` character.

very similar to a **`char` array**

`argv[0], ...` are strings for the argument on the command line when the program is executed.

e.g., `$ nano welcome.c`

Input and output (i/o)

If you retrieve the manual for `printf`, e.g., `$ man –S 3 printf`,

```
#include <stdio.h>      // necessary header file for printf()
int printf(const char *format, …)
```

format includes all information how to print; `%d, %c, %f, %s, %u, %o, %x, …`

For example:

```
printf("%d %c %f %u %s", i, c, x, &x, buf);     // int i, char c, float
x, address of x, char* buf or char buf[]
```

`getchar()` a character from stdin

`getline()` a line from a FILE

Note: We will not use these functions in this course if possible.

For file input and output (i/o), use the following:

```
open()                  $ man -S 2 open
      int open(...);                          // returns the
file descriptor
read()                  $ man -S 2 read
      int read(int fd, char* buf, int length);   // fd is
the return value from open()
// buf could be declared as char[...] in
//  the function that calls read().
   // This function will read length bytes
   //  and store them into buf, and will
   //  return the number of bytes read.
                                                  // what if the
return value is < length?
write()           $ man -S 2 write
      int write(int fd, char* buf, int length);  // fd is
the return value from open()
// buf could be declared as char[...] in
//  the function that calls write().
   // This function will write length bytes
   //  stored in buf into the file, and will
   //  return the number of bytes written.
close()           $ man -S 2 close
      int close(fd);
```

Note: All the system functions return a negative value when there is an error.

Predefined file descriptors – no need to open again; already opened

0	standard input
1	standard output
2	standard error

Exercise Program: file_copy.c

Review the following sample code combining the above elements:

```
/*
 * copy a file to another file
 */

#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>      // These four header files are necessary to
use file i/o functions.
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

/* argv[1]: from */
/* argv[2]: to */

int main(int argc, char* argv[])
{
              int in, out;

              in = open(argv[1], O_RDONLY);            // O_RDONLY is
              defined in /usr/include/sys/types.h
              if (in < 0) {
                        printf("Cannot open the file %s\n", argv[1]);
                        exit(1);
              }

              out = open(argv[2], O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR
              | S_IWUSR);
              if (out < 0) {
                        printf("Cannot create the file %s\n", argv[2]);
                        exit(1);
              }
```

```
        ... // read some bytes from in and write them into out.
        ... // print those bytes onto the screen while copying.

        close(in);
        close(out);

        ... // test if the two files have the exactly same
        contents, and print the result
}
```

Global variables

- Variable declared not inside a function
- Common data for all functions in the file
- It is like public instance variables in Java.
- In order to access a global variable declared in other file,
    - `extern data_type variable_name;`

User defined data type

- **define** – macro and constant definition; like final variable declaration in Java

```
#define     BUF_SIZE    512
```

- **struct** – it is like a class definition in Java, which has only instant variables

```
struct record {
        int   number;
        char  name;
};

struct record kildong_record;
```

- **union**

```
union weight {
        int   kilo;
        int   pound;
};

union weight kildong_weight;
```

- **typedef**

```
typedef struct record record;
record kildong_record;
```

Note: We will not use union and typedef in this course if possible.

Linking and libraries

- `$ gcc –c main.c` compile `main.c` into `main.o`
- `$ gcc main.o student.o –o test –lm` `-lm` means library `libm.a` that is in `/usr/lib/`

Handling strings

A string in C is a consecutive list of bytes (or characters), usually in an array, ending with `'\0'` (0 value). When we use general data do not use these functions because even `'\0'` can be a part of the data.

- `<string.h>` related header file
- **`strcpy()`** string copy; e.g., `strcpy(dest_str, src_str);` // two arguments could be `char` arrays
  - **`strcpy(buf, "Welcome to ...\n");`** **// buf is a char array of enough room**
- **`strlen()`** string length; e.g., len = strlen(argv[0]);
- `strcat()` string concatenation; e.g., `strcat(dest_str, src_str);`
- `strcmp()` string compare; e.g., `i = strcmp(first_str, second_str);`

Note: We will not use `strcat()` and `strcmp()` in this course if possible.

- **`atoi()`** convert to integer; e.g., `i = atoi(argv[2]);`
- `atof()` convert to float; e.g., `f = atof(argv[1]);`

Handling memory

- `malloc()` allocate dynamic memory
- `free()` purge an allocated dynamic memory

Note: We will not use them in this course if possible.