CS211: Programing For Operating Systems

Robert Davidson

Contents

Useful headers

- `stdio.h` : Standard input/output library
- `stdlib.h` : Standard library
- `string.h` : String manipulation functions (`strncpy`, `strcat`, `strcmp`, `strstr`, `strlen`)
- `unistd.h` : Standard library for system calls (`fork`, `getpid`, `getppid`)

Code: My C Program

```
1     #include <stdio.h>
2
3     int main(void) {
4             // ...
5         }
```

1 Intro to C

It is a very small language and relies heavily on libraries The compiler must be told in advance how these functions should be used. So before the compilation process, the **preprocessor** is run to include the function prototypes The compiler then compiles the code into an object file.

1.1 Hello World

Listing 1: Hello World in C

```
1    #include <stdio.h>
2    int main(){
3        printf("Hello World\n");
4        return 0;
5    }
6
```

- **Line 1** : `#include <stdio.h>` is a preprocessor directive that tells the compiler to include the standard input/output library. This library contains the `printf` function.
- In C almost every line it either preprocessor directive, variable declaration, or a function call.
- C uses curly braces to delimit blocks of code and semicolons to terminate statements.
- **Line 4**: In our case, we assume main is called by the Operating System, so return 0 is used to indicate that the program has run successfully.

1.2 Variables

In C all variables must be declared before they are used. The declaration should have a type; telling the compiler what sort of data the variable will hold. The types of variables are:

- **int** : Integer (1, 2, 3, 4, 5, ...)
- **float** : Floating-point number (7 decimal digits)
- **double** : Double-precision floating-point number (15 decimal digits)
- **char** : Character (a, b, c, ...)
- **void** : No type (used for functions that do not return a value)

We can Also declare arrays as follows:

Listing 2: Declaring Arrays

```
1    int arr[5]; // Array of 5 integers
2    char name[10]; // Array of 10 characters
```

To access the first element of arr we can do `arr[0]`

1.3 An Example

Listing 3: Example of Variables

```
1    int d=-101;
2    float f=1.23456;
3    char c='a';
4    printf("Values of d, f, c are: %d, %f, %c\n", d, f ,c );
```

Explanation: In this case, `%d` is a placeholder for an integer, `%f` is a placeholder for a float, and `%c` is a placeholder for a character.

1.4 Operators

Operator	Description	Example
+	Addition	`a + b`
-	Subtraction	`a - b`
*	Multiplication	`a * b`
/	Division	`a / b`
%	Modulus	`a % b`

Table 1: Arithmetic Operators

Operator	Description	Example
=	Assignment	`a = b`
+=	Add and assign	`a += b`
-=	Subtract and assign	`a -= b`
*=	Multiply and assign	`a *= b`
/=	Divide and assign	`a /= b`
%=	Modulus and assign	`a %= b`
++	Increment	`a++`
–	Decrement	`a-`

Table 2: Assignment and Arithmetic Assignment Operators

Operator	Description	Example
==	Equal	`a == b`
!=	Not Equal	`a != b`
>	Greater	`a > b`
<	Less	`a < b`
>=	Greater or Equal	`a >= b`
<=	Less or Equal	`a <= b`

Table 3: Relational Operators

Operator	Description	Example
&&	Logical AND	`a && b`
||	Logical OR	`a || b`
!	Logical NOT	`!a`

Table 4: Logical Operators

1.5 Control Structure

Listing 4: If-Else

```
1     int a = 10;
2     if(a > 10){
3             printf("a is greater than 10\n");
4         }else if(a == 10){
5             printf("a is equal to 10\n");
6         }else{
7     printf("a is less than 10\n");
8     }
9
```

Logical opeators, `&&` and `||` can be used to make more complex conditions.

Listing 5: Complex If-Else

```
1     if(a > 10 && a < 20){
2         printf("a is between 10 and 20\n");
3     }
4
```

1.5.1 For loop

```
for(initial val; continuation condition; increment/decrement){...}
```

Listing 6: Print numbers from 0 to 9

```
1 for(int i = 0; i < 10; i++){
2     printf("i is %d\n", i);
3 }
```

1.5.2 While loop

```
while(expression){...}
```

Listing 7: Print numbers from 0 to 9

```
1 int i = 0;
2 while(i < 10){
3     printf("i is %d\n", i);
4     i++;
5 }
```

1.5.3 Do While loop

```
do{...}while(expression);
```

Listing 8: Print numbers from 0 to 9

```
1 int i = 0;
2 do{
3     printf("i is %d\n", i);
4     i++;
5 }while(i < 10);
```

1.6 Output

`printf()` is used to print formatted output to the screen. It is a variadic function, meaning it can take any number of arguments. The first argument is a format string, followed by the values to be printed.

The format string may contain a number of escape characters, represented by a backslash. Some of the most common escape characters are:

Sequence	Description
`\a`	Produces a beep or flash
`\b`	Moves cursor to last column of previous line
`\f`	Moves cursor to start of next page
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\\`	Prints a backslash
`\'`	Prints a single quote

A conversion character is a letter that follows a `%` and tells `printf()` to display the value stored in the corresponding variable. Some of the most common conversion characters are:

Specifier	Description
`%c`	Single character (char)
`%d` or `%i`	Decimal integer (int)
`%e` or `%E`	Floating-point (scientific notation)
`%f`	Floating-point value (float)
`%g` or `%G`	Same as %e/%E or %f, whichever is shorter
`%s`	String (char array)
`%u`	Unsigned int
`%x`	Hexadecimal integer
`%p`	Pointer (memory address)
`%%`	Prints the % character

1.7 Input

`scanf()` reads input from standat input, format it, as directed by a conversion character and store the address of a specified variable.

Listing 9: Reading an integer

```
1     int number;
2     char letter;
3     printf("Enter a number and a char: ");
4     scanf("%d %c", &number, &letter);
5
6     printf("You entered: %d and %c\n", number, letter);
```

- The scan `scanf()` returns an integer equal to the number of successfull conversions made.
- There is related function `fscanf()` that reads from a file. `scanf()` is really just a wrapper for `fscanf()` that treats the keyboard as a file.
- There are other useful functions for readint the standard input stream: `getchar()` and `gets()`.

Listing 10: Check for no input

```
1     int number;
2     printf("Enter a number between 1 and 30: ");
3     scanf("%d", &number);
4
5     while ((number<1) || (number>30))
6     {
7         printf("Invalid number. Please enter a number between 1 and 30: ");
8         scanf("%d", &number);
9     }
```

1.8 Functions

***Def.* Protype and Definition**

Protype : A function prototype is a declaration of a function that tells the compiler what the function looks like. It includes the function name, return type, and parameter types. The prototype must be declared before the function is called.

Definition : A function definition is the actual implementation of the function. It includes the function name, return type, parameter types, and the body of the function. The definition must be declared after the function is called.

1.9 Call-by-Value and Pointers

In C is it important to distuinguish between a variable and the value stored in it. A variable has a location in memory. The value of the variable is stored in that location. For example:

```
1 int i = 10;
```

tells the system to allocate a location in memory to store the value 10. The variable `i` is a pointer to that location in memory. One of the distuinguihing features of C is it we can manipulate the memory address of the variable almost as easily as we can manipulate the value stored in it.

***Concept:* Pointers**

- if `i` is a variable, then `&i` is a pointer to the location in memory where the value of `i` is stored.
- The declaration `int *p;` creates a variable `p` that can store the memory address of an integer. The `*` indicates that `p` is a pointer to an integer.
- If a memory address is stored in the variable `p`, then `*p` is the value stored at that address.

1.10 Characters

Concept: Character representation

In C, a character is just an unsigned integer. Each character is represented by an integer between 0 and 127.

Printing characters:

- `printf("&c", c)`
- `putchar(c)`

Reading characters:

- `scanf("&c", c)`
- `c = getchar()`

1.11 Strings

Concept: String representation

C does not have a string data type. Instead it uses arrays of type char to represent strings. For example if we make a declaration:

```
char greeting[20]="Hello.  How are you?";
```

the system stores each character as an element of the array greeting.

1.11.1 String Functions

Useful functions defined in `string.h`:

```
char *strncp(char *dest, char *src, int n)
```

Copies at most `n` characters from `src` to `dest`. The advantage of this is that we don't copy more chatavyers to `dest` than it can hold (prevents buffer overflow). `strncpy()` does not append a null character to the end of the string, so we need to do that manually.

```
char *strcat(char *dest, char *src)
```

Concatenates the string `src` to the end of the string `dest`. The destination string must be large enough to hold the concatenated result. `strcat()` appends a null character to the end of the string.

```
int strcmp(char *str1, char *str2)
```

Compares two strings annd returns an intger:

- 0 if the strings are equal
- A negative integer if `str1` comes first alphabetically
- A negative integer if `str2` comes first alphabetically

```
char *strstr(char *haystack, char *needle)
```

Searches for the first occurrence of the string `needle` in the string `haystack`. If found, it returns a pointer to the first occurrence of `needle` in `haystack`. If not found, it returns NULL.

```
int strlen(char *str)
```

Returns the length of the string `str` (not including the null character).

1.11 Strings (Continued)

1.11.2 String Output

We know how to use `printf()` to print strings:

- `printf("%s%s\n, "Good Morning ", name);`
- `printf("%s%8s\n", "Good Morning ", name);`

The second example, the field width specifier is given. This causes the string to be padded so it takes up a total of 8 spaces. If the string is shorter than 8 characters, it is padded with spaces on the left. If the string is longer than 8 characters, it is printed as is.

1.11.3 String Input

- `scanf("%s", name);` reads the next "word" from th input stream and stores it in the array `name`. A "word" is defined as a sequence of characters separated without a space, tab, or newline. The string is terminated with a null character.
- `getchar(name);` to get more control of the input in a loop
- `gets(string);` reads a line input and stores it all except the newline character. The string is terminated with a null character. `gets()` is not safe to use because it does not check the length of the input string. If the input string is longer than the array, it will cause a buffer overflow and overwrite other data in memory.
- `fgets(string, n, stdin);` reads in a line of text from the keyboard (standard input stream) and stores at most `n` characters in the array `string`. The string is terminated with a null character. If the input string is longer than `n`, it will be truncated and the rest will be discarded. `fgets()` is safer to use than `gets()` because it checks the length of the input string.

2 Arrays

3 Arrays

To declare a 3×4 matrix of floats, we write `float a[3][4];`. So:

$$\begin{bmatrix} a[0][0] & a[0][1] & a[0][2] & a[0][3] \\ a[1][0] & a[1][1] & a[1][2] & a[1][3] \\ a[2][0] & a[2][1] & a[2][2] & a[2][3] \end{bmatrix}$$

In general, an $n \times m$ array is declared as `float a[n][m];`. The first index is the row number and the second index is the column number. The first element of the array is `a[0][0]` and the last element is `a[n-1][m-1]`.

If a has the line `int [a][4];` then the system creats three arrays, each of length four. More precisely, it:

- declares 3 pointers to type `int`: `a[0]`, `a[1]`, and `a[2]`
- space for storing an integer is allocated to each of addresses `a[0], a[0]+1, a[0]+2, a[0]+3, a[1], a[1]+1`, ..., `a[2]+3`

This mean is if `a[][]` is declared as a two-dimensional 3×4 array, then the following are equivalent:

- `a[1][2]`
- `*(a[1]+2)`
- `*(*(a+1)+2)`
- `*(&a[0][0]+4+2)`

3.1 Explanation of Array Element Access in C

`a[1][2]`

This is the standard way to access a two-dimensional array element. It directly fetches the element in the second row (index 1) and the third column (index 2).

`*(a[1]+2)`

- `a[1]` yields the second row, which decays to a pointer to its first element (i.e., equivalent to `&a[1][0]`).
- Adding `2` moves the pointer two elements forward in that row.
- The dereference operator `*` then accesses the element at that position, which is `a[1][2]`.

`*(*(a+1)+2)`

- `a+1` moves the pointer from the first row to the second row.
- `*(a+1)` dereferences that pointer to yield the address of the first element of the second row (again, equivalent to `a[1]`).
- Adding `2` moves to the third element in that row, and the outer `*` fetches its value—again, `a[1][2]`.

`*(&a[0][0]+4+2)`

- `&a[0][0]` gets the address of the very first element of the array.
- Since the array is stored in contiguous memory, pointer arithmetic treats it as a flat sequence. Adding `4+2` (i.e., 6) moves the pointer to the 7th element in that sequence.
- If the layout of the array is such that the element `a[1][2]` is the 7th element (this is true, for example, if the row length is at least 3), then dereferencing this pointer retrieves `a[1][2]`.

4 Files

Taking an input from a file is not much different than taking input from a keyboard. All we do is:

- Declare an identifier of type `FILE` to hold the file pointer.
- Open the file (`fopen()`)
- Read the file
- Close the file (`fclose()`)

4.1 Declaring a file indentifier:

```
FILE *datafile;
```

The datafile is now a pointer we can associate with a file. The `FILE` type is defined in the `stdio.h` library.

4.2 Opening a file:

```
fileptr = fopen(char *filename, char *mode);
```

The `fopen()` is a function that is use for file opening. It takes two arguments: the name of a file and the mode it will operrate in. A file pointer is returned. The mode can be:

- `r` : read (open an existing file for reading)
- `w` : write (overrwite the file or create a new one)
- `a` : append (add to the end of the file)

4.3 Closing a File

```
fclose(fileptr);
```

Once a file is "closed" we can no longer read from it or write to it, unless we open it again. If we don't do this, the file will still be closed when the program terminates. But until then, no other program on the same node (computer) can work with the file, and it might but be fully written to storage

4.4 Reading from a file

```
fgets(char *str, int n, FILE *fileptr);
```

reads in a line of text from the fileptr stream and stores at most n characters in array sting. The mew line character is stored. If the string can't be read, because we have reached the end of the file, then NULL is returned.

```
fgetc(FILE *fileptr);
```

reads the next character in the file and stores it in the char variable c. If the end of the file has been reached, EOF is returned.

```
fscanf(FILE *fileptr, char *format, ...);
```

reads formatted input from the file. The format string is similar to the one used in `printf()`

4.5 Navigating a file

Each time a character is reader from the input stream, a counter associated with the stream is incremented.

```
rewind(FILE *fileptr);
```

sets the indicator to the start of the file

```
ftell(FILE *fileptr);
```

is used to check the current value of the file position indivator in the form of a long int.

```
fseek(FILE *fileptr, long offset, int place);
```

the value of offset is the amount the indicator will be changed by, while placce is one of:

- `SEEK_SET(0)` : the start of the file
- `SEEK_CUR(1)` : the current position in the file
- `SEEK_END(2)` : the end of the file

4.6 Writing to a file

To write to a file, we declare a file pointer:

```
FILE *outfile;
```

and open a new file in write mode:

```
outfile = fopen("myList.txt", "w");
```

to write to the file, we use one of:

- `fprintf(outfile, char *format, ...);` : works like `printf()` except its first argument is a file pointer
- `fputs(char *str, FILE *fileptr);` : writes the string str to the file pointed to by fileptr, without its trailing '\0' character. The string is not formatted.
- `fputc(int c, FILE *fileptr);` : writes the character c to the file pointed to by fileptr. The character is not formatted.

4.7 Issues concerining the use of files in C, we should be aware of:

- There are 6 modes a file can have: `r, w, a, r+, w+, a+`
- To open a binary file, also add a `b` to the mode. For example, `rb` opens a file for reading in binary mode.
- `freopen()` attaches a new file to an existing file pointer. It is used to redirect the standard input or output to a file. For example, `freopen("myList.txt", "w", stdout);` redirects the standard output to the file `myList.txt`. This means that any output that would normally go to the screen will now go to the file.
- `tmpfile()` open a tempory file in binary read/write and is automatically deleted when closed or when the program terminates
- `fflush(fileptr);` flushes the output buffer of the file pointer. This means that any data that has been buffered but not yet written to the file will be written to the file.
- `remove("myList.txt");` deletes the file `myList.txt` from the disk.
- `rename("myList.txt", "myList2.txt");` renames the file `myList.txt` to `myList2.txt`. If the new name already exists, it will be overwritten.
- `int feof(FILE *fileptr);` returns a non-zero value if the end of the file has been reached. It is used to check if we have reached the end of the file while reading it.

5 The Process

"A process is a running program."

The operating system gives the impression many programs are running at the same time, but in reality, only one program is running at a time. This is made possible by abstracting the the concept of a running program as a process.

Concept: **Every process has**

- **The process text:** the program code
- **The program counter:** the address of the next instruction to be executed
- **The process stack:** temporary data, local variables, function parameters, return addresses
- **The data section:** global variables

A process is not just a program - if two users run the same program at the same time, they create different processes. Each process has its own memory space, so they do not interfere with each other. A program is a passive entity, while a process is an active / dynamic entity.

Concept: **Operations OS must perform on a process**

- **Create** a new process (open a new program)
- **Terminate / Destory** a process (close a program)
- **Wait** or pause a process until some event occurs (e.g. waiting for user input)
- **Supsend and Resume** similar to wait but more explicit
- **Status** report info about a process (e.g. how much memory it is using, how long it has been running, etc.)

Concept: **States of a process**

- **New** : the process is being created
- **Running** : instructions being executed
- **Block / Waiting** : waiting for some event to occur (e.g. I/O operation)
- **Ready** : waiting to be assigned to a processor
- **Terminated** : the process has finished executing

5.1 Process Creation

A parent creates a child process. The child process can create its own child, forming a tree of processes. After a parent creates a child process it may:

- **Execute** concurrently with the child process
- **Wait** until the child process terminates to continue

The parent may share all, none or some of its resources with the child process (memory space, open files, etc.)

5.2 Process Identifcation Number

ll processes have a unique identifier called a **PID**. If we create a child process in C, using a `fork()` a new process is created:

- The new process runs **concurrently** with the parent, unless we instruct it to `wait()`.
- The subproc (child) is given a **copy of the parents memory space**
- At the time of creation, the two processes are identical, except the `fork()` returns te child process' PID to the parent and 0 to the child.

In order to use `fork()` , we must include the `unistd.h` library, which includes:

- `fork()` - creates a new process
- `getpid()` - returns the PID of the calling process
- `getppid()` - returns the PID of the parent process