C Basics

CS61C Note 2

Cesar

1 Why C

C programming allows us to exploit certain computer architecture features like, CPU memory and developing system-level programming for operating systems.

CPU architecture means how processors execute instructions, data, and interact with dynamic memory.

With this understanding, this created the godfather of all operating systems, **Unix**.

Unix is a multi-tasking and multi-user computer operating system developed by AT(and)T in 1970 for current programmers to experiment new technological discoveries. Dennis Ritchie, the founder of C programming, was also a founder of Unix during this time.

Many modern operating systems are derivatives or unix-like. For example, MacOS is a derivative of Unix and Linux is unix-like.

2 Basics

Hello World

```
#include <stdio.h>

int main() {
   printf("hello world!\n");
   return 0;
}
```

C is a function-oriented program, meaning breaking down memory into function calls.

Compilers and Interpreters

Compiler refers to transforming readable code into usually machine code or assembly.

Characteristic of a compiler in C

1. Slow to develop because you have to edit the code, compile, worst case there is a bug, so you need to edit and fix.

2. Reasonable compilation time, meaning how fast (gcc file.c) compiles. This is because C programs are converted directly into architecture-specific machine code.

3. Pretty fast run-time, meaning how fast (./a.out) is executed.

- If compilation is fast $->$ you have less optimization in your code $->$ slower-runtime

- If compiliation is slow $->$ you have more optimization in your code $->$ faster run-time

How does the compiler find errors in a program? The compiler follows a sequence of steps before it is executed.

1. **Tokenizing(word analysis):** Breaks code into words (ex. "int", "x", "=", "10", ";")

2. **Syntax analysis:** Checks grammar rules in a given program language.

3. **Logic analysis:** Checks meaning of variables, types, and other logic operations in a given program language.

4. **Generate code:** Converts readable code into assembly or machine code.

If any of these stages finds a problem, the compiler reports and error.

Interpreter refers to executing readable code by a program that is not a machine. Python3 interprets and executes python source code as an example.

Characteristic of a interpreter.

1. Fast to develop because you edit and run the code without actually doing any compilation.

2. Compilation is generally slow becaues the interpreter need to convert an entire source code into machine code before executing.

3. Slow run-time because an interpreter may have dynamic methods, functions and classes that the program must keep track of before executing.

3 C Syntax

1. Language model: Function oriented
2. Compilation: gcc hello.c, creates machine language code
3. Execution: ./a.out, loads and executes the program
4. Memory Managment: Manual (Malloc, free)

Another way of programming the main function in C is using the command-line arguments.

```
#include <stdio.h>

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

 printf("Number of argc strings: %d\n", argc); // number of
 strings on the command line

 for(int i = 0 ; i <= argc; i++) {
// argv is a pointer variable that points to
each element in the array of strings
// array -> ./a.out Hello -> ["./a.out","Hello", null],
NULL tells us when the arrays is finished
  printf("Argc number %d: %s\n", i, argv[i]);
 }
 return 0;
```

So the main function has two parameters, argc and argv. Argc counts the number of argument passed and argv fetches its individual value. Here is an example

```
$ gcc file.c -o file // assigning the word
file as the output name associated with the file.c
$ ./file hello world
```

```
Result
Number of argc strings: 3
Argc number 0: ./file
Argc number 1: hello
Argc number 2: world
Argc number 3: null // null indicates the array is
done, no more inputs
```

4 C Variables

We have different C types:

1. Signed integer, positive and negative numbers (ex. -2, 120, 0, -300) = int x; (4 bytes)
2. Unsigned integer, non-negative numbers (ex. 0, 3, 33, 100, 192929) = unsigned int x; (4 bytes)
3. Double integers, precise decimal numbers (ex. -32.5, 4.4, 3.14) = double x; (8 bytes)
4. Float integers, normal decimal numbers (ex. -32.5, 4.4, 3.14) = float x; (4 bytes)
5. Characters, single characters (ex. 'h', 'e', 'o') = char letter; (1 byte)
6. Short integers, short signed numbers (ex. -2, 120, 0, -300) = short x; (2 bytes)
7. Long integers, long signed numbers (ex. -2, 120, 0, -300) = long x; (4 or 8 bytes)
8. Long long integers, longer signed numbers (ex. -2, 120, 0, -300) = long long x; (8 bytes)

Declaring C variables

In C programming, variables are not automatically declared and intialized for you. So, to declare and initialize a variable you:

```
#include <stdio.h>
int main(){
 int x; // declare a variable x with a type integer
 x = 10; // initialize variable x
 printf("value of x is %d\n", x);
 return 0;
}
```

This is also okay and faster for declaring variables

```
int x = 20; // faster (declaration + initialization)
```

C variables are typed, meaning you can type a specific condition for a variable. For example, if I want to declare a unsigned integer that stores a value using only 16 bits, I can type:

```
uint16_t z = 38;
```

So, the unsigned value 38 will be stored using 2 bytes. However this comes from the library $< stdint.h >$, that makes integers already have this specific condition.

More C Features

constant: We can have variables that don't change after initializing, using the keyword const (constant) that is assigned to a type value. That value can not be altered during the entire execution of a program.

```
int main() {
 const char name[] = "Cedar"; // can not be modified after initializing
 *name = 'L'; // trying to replace 'C' for 'L' ERROR
 printf("First letter of the string %c\n", *name);
 return 0;
}
```

In this example, after initializing the (const) keyword, you are no longer able to edit the orignal type value. So this snippet will return an error because of the constant assigned to the type char.

sizeof: A compile-time operator that measures the size of a type/variable in bytes. Does not work for functions because sizeof only works for expressions, types and variables.

```
int main() {
 int num = 100;
 long z = 9;
 double pi = 3.14;
 printf("Using sizeof operator for variables num, z, and pi:
   %lu bytes, %lu bytes, %lu\n bytes",
   sizeof(num), sizeof(z), sizeof(pi));
 return 0;
}
```

We have three different variables with three different types. Each type has its own byte size, for example, sizeof(num) will print 2 bytes, sizeof(z) 8 bytes and sizeof(pi) 4 bytes. The sizeof operator looks at the type associated with the variable.

5 struct and typedef Type

struct

We can implement a user-defined structure type to group different variables. This allows us to pass multiple values in a single function parameter and most importantly organizes memory from the variables it is storing.

Here is a simple example of a struct type in C

```
struct Student {
 char name[100];
 int age;
 double gpa;
};

int main() {
 struct Student x = {"Mark", 12, 3.3};
 printf("Student %s is %d years old and has a %f gpa\n", x.name, x.age, x.gpa);
 return 0;
}
```

We have a semi-colon at the end of the struct because C treats struct as a declaration. In the main function, (struct Student) becomes a type for variable x that stores three arguments, name, age, and gpa.

To illustrate this in more depth, here is a more advanced example of using a struct type in C. This example implements a struct type name student with a constructor like function that initalizes its variables from the struct function, and then prints it out.

```
#include <stdio.h>
#include <string.h>

struct Student {
 char name[20];
 int age;
 double gpa;
 char school[40];
```

```
};

struct Student makeStudent(const char n[20], int a, double g, const char s[40]) {

 struct Student p;
 // strcpy = "string copy", we use this because we our (char[]) has a fixed number
 of characters
 strcpy(p.name, n);
 p.age = a;
 p.gpa = g;
 strcpy(p.school, s);
 return p;
}

void printStudent(struct Student x) {
 // x.name, x.age,..etc, x is our parameter in this function.
 printf("Student information -> Name: %s, Age: %d, GPA: %f, School: %s.\n", x.name,
   x.age, x.gpa, x.school);
}

int main() {
 struct Student user = makeStudent("Jeff", 15, 3.4, "Wildcat High");
 printStudent(user);
 return 0;

}
```

Because struct is a declaration for a new type, we can use the sizeof operator to measure the size of its variables. For example,

```
#include <stdio.h>
#include <string.h>
struct Demo {
 int z; // -> sizeof(int) = 4 bytes
 long x; // -> sizeof(long) = 8 bytes
 char y[30]; // -> sizeof(char) = 1 byte * 30
 float pi; // -> sizeof(4 bytes)
};

int main() {
 struct Demo trial = {2010, 3, "hello world", 3.14};
 // sizeof the total amount of variables = 50 bytes
 printf("Using sizeof on the new type structg Demo: %lu\n", sizeof(trial));
 return 0;
}
```

However, you can not initialize the variables inside the struct because C treats struct as a declaration, which is why we have a semi-colon at the end of the struct.

typedef

typedef is a keyword that allows us to define a type using an alias to an existing type. The main difference is struct creates a new type and typedef makes a nickname to an exisiting type.

A simple example of typedef

```
int main() {
 typedef double dd;
 dd balance = 900.9;
 printf("Using typdef, here is my balance %f\n", balance);
 return 0;
}
```

We used the keyword typedef followed by the exisiting type double then we assign a nickname associated with the existing type double. Thus, the keyword (dd) becomes an alias to double.

Additionally we can combine struct and typedef to simplify its call when passing arguments. Here is an example

```
typedef struct {
 int x;
 int y;
}Coordinates;

int main() {
 Coordinates findMy = {20,11}; // no need to type struct, just the nickname of the struct
 printf("My coordinates is x = %d and y = %d\n", findMy.x, findMy.y);
 return 0;
}
```