CSC360: Quick Guide to C

C development tools and syntax

Acknowledgement

- The content about C develop tool Chain is based on the materials provided by **Clark Zhao**
- The content about C syntax is based on the C handbook by **Flavio Copes**
- Thank **Robert Russell** and **Victor Kamel** a lot for giving a lot of constructive suggestions and comments on improving the slides

C Develop Tool Chains in Linux

Choose a text editor or IDE:

- Command line text editor: vim, nano, emacs, …
- GUI text editor: VSCode, Sublime text, …
- IDE: Jetbrains CLion, Visual Studio, …

C Compiler:

- gcc (GNU Compiler Collection)
- $ gcc hello.c -o hello
- $ ./hello

Debugger:

- gdb

1. Create and Edit Source Files

- Using your favorite editor mentioned before: vim or emacs etc.
- An example: $ vim hello.c

2. Compile Source Files

- $ gcc hello.c -o hello
- Preprocess -> compile -> assemble -> link
- Enable warnings: $ gcc hello.c –Wall –o hello
- Treat warning as error: $ gcc hello.c -Wall -Werror -o hello
- -Wshadow warn whenever a local variable shadows another local variable, parameter or global variable or whenever a built-in function is shadowed
- -Wpedantic stick more closely to the language standard

3. Execute Output

- $ ./hello

4. Make sure you code can compile, run and produce expected results on https://jhub.csc.uvic.ca/

Demo

Makefile

- Makefile is a way of automating software building procedure and other complex tasks with dependencies

Makefile Syntax

```
target … : prerequisites …
<TAB>recipe/command
        …
        …
```

Example

```
hello: hello.c
    gcc -o hello hello.c
```

```
$ cat hello.c
#include "hello.h"
int print_hello_world() {
    printf("Hello World 2\n");
    return 0;
}

$ cat main.c
#include "hello.h"
int main() {
    print_hello_world();
    return 0;
}

$ cat hello.h
#include <stdio.h>
int print_hello_world();
```

T01

Demo

```
$ cat Makefile

main: main.o hello.o
    gcc -o main main.o hello.o

main.o: main.c hello.h hello.c
    gcc -c main.c

hello.o: hello.c hello.h
    gcc -c hello.c

clean:
    rm -f main
    rm -f *.o
```

Compile with just typing `make <target>`

or just `make`

Debugging

- gdb - GNU Debugger

```
$ gcc -g hello.c -o hello
$ gdb hello
```

- A tutorial:
  https://www.cprogramming.com/gdb.html

- GDB documentation:
  https://www.sourceware.org/gdb/documentation/

Basic Syntax

```c
#include <stdio.h>

int main()
{
    printf("Hello World!");
    return 0;
}
```

Basic Syntax

```c
#include <stdio.h>

int main()
{
    printf("Hello World!");
    return 0;
}
```

The header file inclusion to use printf() function.

Basic Syntax

```c
#include <stdio.h>

int main()
{
    printf("Hello World!");
    return 0;
}
```

The main() function is the entry point of any C program

Basic Syntax

```c
#include <stdio.h>

int main()
{
    printf("Hello World!");
    return 0;
}
```

Function to print hello world.

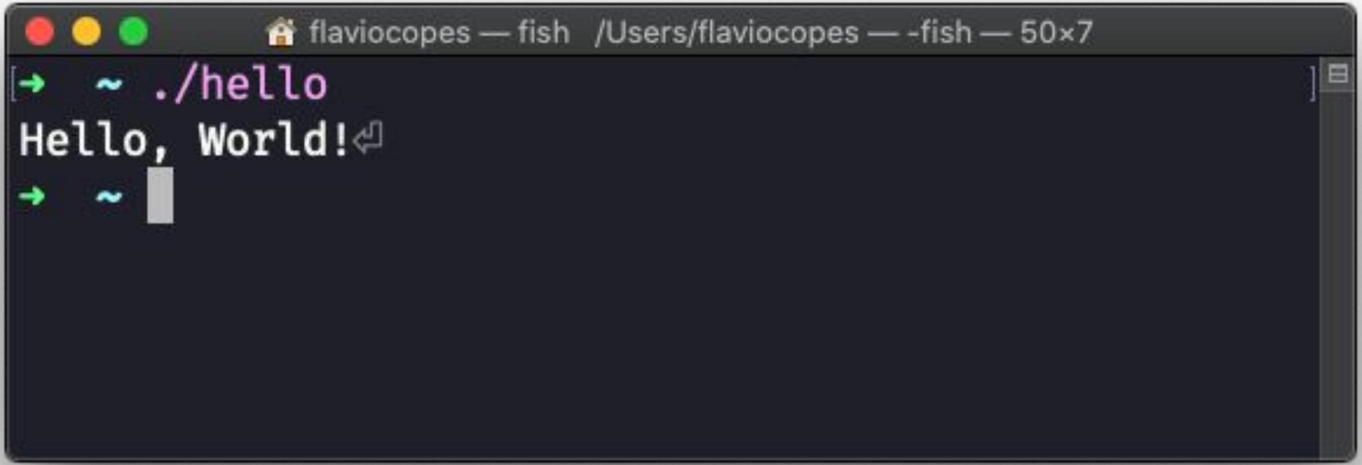
Basic Syntax

```
#include <stdio.h>

int main()
{
    printf("Hello World!");
    return 0;
}
```

Value returned by the main() function

Compile and Execute


flaviocopes — fish   /Users/flaviocopes — -fish — 50×7
→ ~ ./hello
Hello, World!
→ ~

Variables

```
data_type variable_name;
data_type variable_name = initial_value;
```

```
char c = 'a';
int integer = 24;
float f = 24.32;
double d = 24.3435;
void v;
```

Integer Numbers

C provides us the following types to define integer values:

- `char` store letters of the ASCII chart, takes at least 1 byte
- `int` takes at least 2 bytes
- `short` takes at least 2 bytes
- `long` takes at least 4 bytes
- `long long` takes at least 8 bytes

Unsigned Integers

For all the above data types, we can prepend `unsigned` to start the range at 0, instead of a negative number. This might make sense in many cases.

- `unsigned char` will range from `0` to at least `255`
- `unsigned int` will range from `0` to at least `65,535`
- `unsigned short` will range from `0` to at least `65,535`
- `unsigned long` will range from `0` to at least `4,294,967,295`
- `Unsigned long long` will range from `0` to at least 18,446,744,073,709,551,615

stdint.h

- The exact numbers that can be stored in each data type depends on the implementation and the architecture
- E.g., char is signed on x86, while it is unsigned if using arm-linux-gcc
- stdint.h provides a consistent definition for integers
    - Exact-width integer types `e.g., int16_t, uint64_t`

The Problem of Overflow

If you have a `unsigned char` number at 255 and you add 10 to it, you'll get the number `9`:

```c
#include <stdio.h>

int main(void) {
  unsigned char j = 255;
  j = j + 10;
  printf("%u", j); /* 9 */
}
```

Operators

OPERATOR	NAME	EXAMPLE
=	Assignment	a = b
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulo	a % b

+	Unary plus	+a
-	Unary minus	-a
++	Increment	a++ or ++a
--	Decrement	a-- or --a

```c
int a = 2;
int b;
b = a++; /* b is 2, a is 3 */
b = ++a; /* b is 4, a is 4 */
```

Comparison Operators

==	Equal operator	a == b
!=	Not equal operator	a != b
>	Bigger than	a > b
<	Less than	a < b
>=	Bigger than or equal to	a >= b
<=	Less than or equal to	a <= b

Logical Operators

- `!` NOT (example: `!a`)
- `&&` AND (example: `a && b`)
- `||` OR (example: `a || b`)

The Ternary Operator

```
<condition> ? <expression> : <expression>
```

```
int a = 2;
int b;
b = a > 0? 1 : 0; /* b is 1/
```

Operator Precedence

In order from less precedence to more precedence, we have:

- the `=` assignment operator
- the `+` and `-` **binary** operators
- the `*` and `/` operators
- the `+` and `-` unary operators

```
int a = 2;
int b = 4;
int c = b + a * a / b - a;
```

Conditionals

```
int a = 1;

if (a == 1) {
  /* do something */
}
```

```
int a = 1;

if (a == 2) {
  /* do something */
} else {
  /* do something else */
}
```

```c
int a = 1;

if (a == 2) {
  /* do something */
} else if (a == 1) {
  /* do something else */
} else {
  /* do something else again */
}
```

```c
int a = 1;
switch (a) {
  case 0:
    /* do something */
    break;
  case 1:
    /* do something else */
    break;
  default:
    /* handle all the other cases */
    break;
}
```

for Loops

```
for (int i = 0; i <= 10; i++) {
  /* instructions to be repeated
*/
}
```

while loops

```
int i = 0;

while (i < 10) {
  /* do something */

  i++;
}
```

Do while loops

```
int i = 0;

do {
  /* do something */

  i++;
} while (i < 10);
```

Break

```
for (int i = 0; i <= 10; i++) {
  if (i == 4 && someVariable == 10) {
    break;
  }
}
```

Continue

```
for (int i = 0; i <= 10; i++) {
  if (i == 4 && someVariable == 10) {
    continue;
  }
  /* do something else */
}
```

Derived Data Types

Derived data types are derived from the basic data types. There are 2 derived data types in C:

1. Arrays
2. Pointers

Arrays

```
int prices[5];
```

You can initialize an array at definition time, like this:

```
int prices[5] = { 1, 2, 3, 4, 5 };
```

But you can also assign a value after the definition, in this way:

```c
int prices[5];

prices[0] = 1;
prices[1] = 2;
prices[2] = 3;
prices[3] = 4;
prices[4] = 5;
```

1. The variable name of the array, `prices` in the example, is a **pointer** to the first element of the array
2. A value of an array type can *decay* to a value of a pointer type when passed as an argument to a function (Demo)

Strings

In C, strings are one special kind of array: a string is an array of `char` values

```
char name[7];
```

A string can be initialized like you initialize a normal array:

```
char name[7] = { ‘F’, ‘l’, ‘a’, ‘v’, ‘i’, ‘o’ };
```

Or more conveniently with a string literal (also called string constant), a sequence of characters enclosed in double quotes:

```
char name[7] = "Flavio";
```

Do you notice how "Flavio" is 6 chars long, but I defined an array of length 7?

The last character in a string must be a `0` value, the string terminator, and we must make space for it

This is important to keep in mind especially when manipulating strings.

Speaking of manipulating strings, there's one important standard library that is provided by C: string.h.

Manipulating Strings

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

- `strcpy()` to copy a string over another string
- `strncpy()`to copy a group char of size n over another string **(doesn’t always null-terminate its output)**
- `strcat()` to append a string to another string, similar strncat()
- `strcmp()` to compare two strings for equality
- `strncmp()` to compare the first `n` characters of two strings
- `strlen()` to calculate the length of a string

Pointers

When you declare an integer number like this:

```c
int age = 37;
```

We can use the & operator to get the value of the address in memory of a variable:

```c
printf("%p", &age);

/* 0x7ffeef7dcb9c */
```

We can assign the address to a variable:

```c
int *address = &age;
```

Using `int *address` in the declaration, we are not declaring an integer variable, but rather a **pointer to an integer**.

We can use the pointer operator `*` to get the value of the variable an address is pointing to:

```c
int age = 37;
int *address = &age;
printf("%u", *address); /* 37 */
```

When you declare an array:

```c
int prices[3] = { 5, 4, 3 };
```

The `prices` variable is actually a pointer to the first item of the array. You can get the value of the first item using this `printf()` function in this case:

```c
printf("%u", *prices); /* 5 */
```

The cool thing is that we can get the second item by adding 1 to the `prices` pointer:

```
printf("%u", *(prices + 1)); /* 4 */
```

In fact, the operation prices[1] is the same as *(prices + 1).

Functions

```
void doSomething(int value) {
    printf("%u", value);
}
```

Functions have 4 important aspects:

1. they have a name, so we can invoke ("call") them later
2. they specify a return value
3. they can have arguments
4. they have a body, wrapped in curly braces

We can have multiple parameters, and if so we separate them using a comma, both in the declaration and in the invocation:

```
void doSomething(int value1, int value2) {
  /* ... */
}

doSomething(3, 4);
```

Parameters are passed by **copy**. This means that if you modify `value1`, its value is modified locally.

If you pass a **pointer** as a parameter, you can modify that variable value because you can now access it directly using its memory address.

Input and Output

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

This library provides us with, among many other functions:

- `printf()`
- `scanf()`
- `sscanf()`
- `fgets()`
- `fprintf()`

We have 3 kinds of I/O streams in C:

- `stdin` (standard input)
- `stdout` (standard output)
- `stderr` (standard error)

With I/O functions we always work with streams. A stream is a high level interface that can represent a device or a file. From the C standpoint, we don't have any difference in reading from a file or reading from the command line: it's an I/O stream in any case.

Some functions are designed to work with a specific stream, like `printf()`, which we use to print characters to `stdout`. Using its more general counterpart `fprintf()`, we can specify which stream to write to.

We can use escape characters in `printf()`, like `\n` which we can use to make the output create a new line.

printf()

```
int age_yesterday = 37;
int age_today = 36;

printf("Yesterday my age was %d and today is %d",
age_yesterday, age_today);
```

There are other format specifiers like `%d`:

- `%c` for a char
- `%s` for a string
- `%f` for floating point numbers
- `%p` for pointers

scanf()

This function is used to get a value from the user running the program, from the command line.

We must first define a variable that will hold the value we get from the input:

```c
char name[20];

scanf("%s", name);
```

Variable Scope

When you define a variable in a C program, depending on where you declare it, it will have a different **scope**.

This means that it will be available in some places, but not in others.

The position determines 2 types of variables:

- **global variables**
- **local variables**

Local Variable

A **local variable** is defined inside a function, and it's only available inside that function.

```c
#include <stdio.h>


int main(void) {
  char j = 0;
  j += 10;
  printf("%u", j); //10
}
```

Global Variable

A global variable is defined outside of any function and can be accessed by any function in the program. Access is not limited to reading the value: the variable can be updated by any function.

```c
#include <stdio.h>

char i = 0;

int main(void) {

  i += 10;

  printf("%u", i); //10

}
```

Static Variables

Inside a function, you can initialize a **static variable** using the `static` keyword.

A static variable is initialized to 0 if no initial value is specified, and it retains the value across function calls.

Consider this function

```
int incrementAge() {
  int age = 0;
  age++;
  return age;
}
```

If we call `incrementAge()` once, we'll get `1` as the return value. If we call it more than once, we'll always get 1 back, because `age` is a local variable and it's re-initialized to `0` on every single function call.

If we change the function to:

```
int incrementAge() {
  static int age = 0;
  age++;
  return age;
}
```

Now every time we call this function, we'll get an incremented value:

User-defined Data Types

The user-defined data types are the data types that are defined by the programmers in their code. There are 3 user-defined data types in C:

1. Enumeration
2. Structure
3. Union

Type Definitions

```c
typedef existingtype NEWTYPE;
```

For example

```c
typedef int NUMBER;

NUMBER one = 1;
```

`typedef` gets really useful when paired with two things: enumerated types and structures.

Enumeration

Using the `typedef` and `enum` keywords we can define a type that can have either one value or another.

```
typedef enum {
   //...values
  } TYPENAME;
```

For example

```c
typedef enum {
  monday,
  tuesday,
  wednesday,
  thursday,
  friday,
  saturday,
  sunday
} WEEKDAY;
```

Every item in the enum definition is paired to an integer, internally. So in this example `monday` is 0, `tuesday` is 1 and so on.

```c
#include <stdio.h>

typedef enum {
  monday,
  tuesday,
  wednesday,
  thursday,
  friday,
  saturday,
  sunday
} WEEKDAY;

int main(void) {
  WEEKDAY day = monday;

  if (day == monday) {
    printf("It's monday!");
  } else {
    printf("It's not monday");
  }
}
```

Structure

Using the `struct` keyword we can create complex data structures using basic C types.

This is the syntax of a structure:

```
struct <structname> {
    //...variables
};
```

You can declare variables that have as type that structure by adding them after the closing curly bracket, before the semicolon, like this:

```c
struct person {
  int age;
  char *name;
} flavio;
```

Or multiple ones, like this:

```c
struct person {
  int age;
  char *name;
} flavio, *people;
```

We can access members like: flavio.age or people -> age

We can also declare variables later on, using this syntax:

```c
struct person {
  int age;
  char *name;
};

struct person flavio;
```

and once we have a structure defined, we can access and change the values in it using a dot:

```
struct person {
  int age;
  char *name;
};

struct person flavio = { 37, "Flavio" };

flavio.age = 38;
```

Structures are very useful because we can pass them around as function parameters, or return values, embedding various variables within them. Each variable has a label.

It's important to note that structures are **passed by copy**, unless of course you pass a pointer to a struct, in which case it's passed by reference.

Using `typedef` we can simplify the code when working with structures.

Let's look at an example:

```
typedef struct {
  int age;
  char *name;
} PERSON;
```

Command Line Parameters

In your C programs, you might need to accept parameters from the command line when the command launches.

For simple needs, all you need to do to do so is change the `main()` function signature from

```
int main(void)
```

to

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

`argc` is an integer number that contains the number of parameters that were provided in the command line.

`argv` is an array of strings.

Note that there's always at least one item in the `argv` array: the name of the program

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  for (int i = 0; i < argc; i++) {
    printf("%s\n", argv[i]);
  }
}
```

If we pass some random parameters, like this: `./hello a b c` we'd get this output to the terminal:

```
./hello
a
b
c
```

Header Files

A header file looks like a normal C file, except it ends with `.h` instead of `.c`. Instead of the implementations of your functions and the other parts of a program, it holds the **declarations**.

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

`#include` is a preprocessor directive.

The preprocessor goes and looks up the `stdio.h` file in the standard library because you used brackets around it.

To include your own header files, you'll use quotes, like this:

```c
#include "myfile.h"
```

The above will look up `myfile.h` in the current folder.

You can also use a folder structure for libraries:

```c
#include "myfolder/myfile.h"
```

Macros

With `#define` we can also define a **macro**. The difference between a macro and a symbolic constant is that a macro can accept an argument and typically contains code, while a symbolic constant is a value:

```c
#define POWER(x) ((x) * (x))

printf("%u\n", POWER(4)); //16
```

If defined

We can check if a symbolic constant or a macro is defined using `#ifdef`:

```c
#include <stdio.h>
#define VALUE 1

int main(void) {
#ifdef VALUE
  printf("Value is defined\n");
#else
  printf("Value is not defined\n");
#endif
}
```