Getting Started with C

Writing and Running Your First C Program

Example: Hello World.

1. Save the code below as hello.c:

```c
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

2. Compile the code using GCC:

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

3. Run the program:

```
./hello
```

The GCC Compiler

gcc stands for GNU Compiler Collection. It is the standard C compiler on Linux and many other systems.

gcc takes your human-readable C source code (like hello.c) and compiles it into a program your computer can run (an “executable” file).

General usage:

```
gcc [options] source_file.c -o output_executable
```

Common options:

- -O2 : Optimization level 2
- -Wall : Show all warnings
- -c : Compile only (do not link)
- -g : Include debug info
- -lm: Link the math library (required for math functions like sqrt, sin, etc.).

Make File [Outside the scope]

If you have more than one .c file, compiling each one manually is repetitive. A Makefile automates building your project.

A Makefile saves your compilation steps in a file, so can just type make to build your program.

It also automatically recompiles only what changed!

Example:

```c
// File: main.c
#include <stdio.h>
#include "helper.h"

int main() {
    say_hello();
    printf("The sum of 2 + 3 is %d\n", add(2, 3));
    return 0;
}
```

```c
// File: helper.c
#include <stdio.h>
#include "helper.h"

void say_hello() {
    printf("Hello from helper.c!\n");
}

int add(int a, int b) {
    return a + b;
}
```

```c
// File: helper.h
#ifndef HELPER_H
#define HELPER_H

void say_hello();
int add(int a, int b);

#endif
```

Compile without a Makefile:

```
gcc -c helper.c -o helper.o
gcc -c main.c -o main.o
gcc main.o helper.o -o program
```

Compile with a Makefile:

```
# Compiler and flags
CC = gcc
CFLAGS = -Wall -g

# Source files and output
SRCS = main.c helper.c
OBJS = $(SRCS:.c=.o)
TARGET = program

# Default target
all: $(TARGET)

# Link object files to create executable
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $(TARGET)

# Compile .c to .o
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Clean target to remove build files
clean:
    rm -f $(OBJS) $(TARGET)
```

```
make
./program

make clean
```

C Data Types

Type	Typical Size	Range (Signed)	Format Specifier	Example
`char`	1 byte	–128 to 127	`%c`	`'A'`
`unsigned char`	1 byte	0 to 255	`%c` / `%u`	`'A'`
`short`	2 bytes	–32,768 to 32,767	`%hd`	`12345`
`unsigned short`	2 bytes	0 to 65,535	`%hu`	`12345`
`int`	4 bytes	–2,147,483,648 to 2,147,483,647	`%d`	`42`

Type	Typical Size	Range (Signed)	Format Specifier	Example
`unsigned int`	4 bytes	0 to 4,294,967,295	`%u`	`42`
`long`	8 bytes*	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	`%ld`	`123456789L`
`unsigned long`	8 bytes*	0 to 18,446,744,073,709,551,615	`%lu`	`123456789UL`
`float`	4 bytes	~1.2E-38 to ~3.4E+38	`%f`	`3.14f`
`double`	8 bytes	~2.2E-308 to ~1.8E+308	`%lf`	`3.141592`
`long double`	16 bytes**	~3.4E-4932 to ~1.1E+4932	`%Lf`	`3.14L`

```c
#include <stdio.h>

int main(void) {
    int age = 22;
    char grade = 'A';
    float pi = 3.14f;
    double e = 2.718281828;
    unsigned int count = 100;
    long big_number = 1234567890L;
    short small_number = 42;

    printf("Age (int): %d\n", age);
    printf("Grade (char): %c\n", grade);
    printf("Pi (float): %.2f\n", pi);
    printf("e (double): %.6f\n", e);
    printf("Count (unsigned int): %u\n", count);
    printf("Big number (long): %ld\n", big_number);
    printf("Small number (short): %d\n", small_number);

    return 0;
}
```

Standard Input/Output Library Functions

#include <stdio.h> is the Standard Input/Output header in C.

Function	Purpose	Example Usage	Notes
`printf`	Print formatted output to screen	`printf("Value: %d", x);`	Use format specifiers: `%d`, `%f`, `%s`

Function	Purpose	Example Usage	Notes
`scanf`	Read formatted input from keyboard	`scanf("%d", &x);`	Use `&` to pass variable address
`putchar`	Print single character to screen	`putchar('A');`	
`getchar`	Read single character from keyboard	`char c = getchar();`	
`puts`	Print string to screen	`puts("Hello");`	Adds newline at end
`gets` (unsafe)*	Read string from keyboard	`gets(str);`	**Avoid**—unsafe, use `fgets` instead
`fopen`	Open a file	`FILE *fp = fopen("file.txt", "r");`	Returns `NULL` on failure
`fclose`	Close a file	`fclose(fp);`	
`fgets`	Read line from file/keyboard	`fgets(buf, 100, fp);`	Reads at most n–1 chars, adds `\0`
`fputs`	Write string to file/screen	`fputs("text", fp);`	
`fprintf`	Print formatted output to file	`fprintf(fp, "%d", x);`	Like `printf`, but to a file
`fscanf`	Read formatted input from file	`fscanf(fp, "%d", &x);`	Like `scanf`, but from a file

```c
#include <stdio.h>

int main() {
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);      // Reads input into x

    printf("You entered: %d\n", x);  // Prints x

    char name[50];  // a character array to store a string with null terminator
    scanf("%s", name);           // Read a string from input
    printf("Hello, %s!\n", name); // Print the string

    char buf[100];
    printf("Enter a line: ");
    fgets(buf, 100, stdin); // Reads a line of input (including spaces)
    printf("You entered: %s", buf);


    FILE *fp = fopen("output.txt", "w"); // Opens file for writing
    if (fp) {
```

```
        fprintf(fp, "Number: %d\n", x);  // Writes to file
        fclose(fp);
    }
    return 0;
}
```

Input Redirection (<) [Assignment1 - part1]

Input redirection means feeding the contents of a file to your program as if the user was typing them into the terminal.

You do this with < in the terminal:

```
./myprog < input.txt
```

In your C code, you just use scanf() or fgets() to read from standard input (stdin); you don't need to open the file yourself!

`scanf()`

Source Code:

```
#include <stdio.h>

int main() {
    int num;
    // Reads one integer at a time from standard input
    while (scanf("%d", &num) == 1) {
        printf("Read: %d\n", num);
    }
    return 0;
}
```

How to run:

```
./myprog < input.txt
```

input.txt:

```
7 8 9
10 11
12
```

Output:

```
Read: 7
Read: 8
Read: 9
Read: 10
Read: 11
Read: 12
```

fgets

Source Code:

```c
#include <stdio.h>

int main() {
    char line[256];

    // Reads one line at a time from standard input (including spaces and the
newline character at the end)
    while (fgets(line, sizeof(line), stdin)) {
        printf("Line read: %s", line);
    }
    return 0;
}
```

How to run:

```
./myprog < input.txt
```

input.txt:

```
Hello world!
This is a test.
123 456
```

Output:

```
Line read: Hello world!
Line read: This is a test.
Line read: 123 456
```

Command-Line Arguments: argc and argv [Assignemnt1- part2]

Command-line arguments allow users to provide input to your program when running it from the terminal or command prompt.

This lets your program accept different inputs each time it runs, without changing the code or using scanf().

argc (Argument Count): An integer that tells you how many arguments were passed to the program, including the program name.

argv: An array of strings containing all the arguments passed to the program, where each string represents one argument. The first element (argv[0]) is always the program name.

```
int main(int argc, char *argv[]) {
    printf("First argument: %s\n", argv[1]);
}
```

```
./myprog cpsc os 457
```

```
First argument: cpsc
```

Pointers

A pointer is a variable that stores the memory address of another variable.

Pointers let you work with memory directly—this is powerful for efficient code, sharing data between functions, and dynamic memory management.

```
#include <stdio.h>

int main(void) {
    // ====\Pointer Basics====

    int x = 10;
    int *p; // Declare a pointer to int

    p = &x; // Set p to hold the address of x

    printf("Value of x: %d\n", x);  // x: 10
    printf("Address of x: %p\n", &x);   // 0x7ffeefbff56c
    printf("Value of p (address): %p\n", p);    // 0x7ffeefbff56c
    printf("Value pointed to by p: %d\n", *p);  // 10

    // ====Multiple Pointers====

    int num = 25;
```

```c
    int *p1 = &num;  // p1 points to num
    int *p2 = &num;  // p2 also points to num

    printf("Value via p1: %d\n", *p1); // 25
    printf("Value via p2: %d\n", *p2); // 25

    // Change value using p1
    *p1 = 77;

    printf("Value via p1: %d\n", *p1); // 77
    printf("Value via p2: %d\n", *p2); // 77
    printf("Value of num: %d\n", num); // 77

    // ====Null Pointer========

    int *ptr = NULL;   // Declare a null pointer
    if (ptr == NULL) {
        printf("Pointer is NULL. It does not point to any value.\n");
    }

    // Safe: Do not use *ptr because it's NULL (would cause a crash)
    // *ptr = 10; // <-- This line would crash the program if uncommented!

    int number = 100;
    ptr = &number;         // Now ptr points to variable number
    printf("Pointer now points to x, value: %d\n", *ptr); // 100100

    // ====Array Pointers====

    int arr[3] = {10, 20, 30};
    int *pArr = arr;            // pArr points to the first element of arr

    // Print values using array notation
    for (int i = 0; i < 3; i++) {
        printf("Value at arr[%d]: %d\n", i, arr[i]);
    }

    // Print values using pointer arithmetic
    for (int i = 0; i < 3; i++) {
        printf("Value at arr[%d]: %d\n", i, *(pArr + i));
    }

    *(pArr + 2) = 1000;         // Set the third element of the array to 1000
    printf("Value at arr[2]: %d\n", arr[2]);    // 1000

    return 0;
}
```