Modules and Makefiles

is_triangle function

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

bool is_triangle(double side_a, double side_b, double side_c);

int main(){

    return 0;
}

bool is_triangle(double side_a, double side_b, double side_c){
    if(side_a + side_b > side_c && side_b + side_c > side_a && side_c + side_a >side_b){
        return true;
    }
    else{
        return false;
    }
}
```

main function

```c
int main(){

    double side_a, side_b, side_c;
    printf("Enter side_a: ");
    scanf("%lf", side_a);
    printf("Enter side b: ");
    scanf("%lf", side_b);
    printf("Enter side c: ");
    scanf("%lf", side_c);

    if(is_triangle(side_a, side_b, side_c)){
        printf("Yes, this is a triangle.\n");
    }
    else{
        printf("No, this is not a triangle.\n");
    }

    return 0;
}
```

Removing redundancies

```
bool is_triangle(double side_a, double side_b, double side_c){
    if(side_a + side_b > side_c &&
       side_b + side_c > side_a &&
       side_c + side_a >side_b){
        return true;
    }
    else{
        return false;
    }
}
```

```
bool is_triangle(double side_a, double side_b, double side_c){
    return(side_a + side_b > side_c &&
       side_b + side_c > side_a &&
       side_c + side_a >side_b)
}
```

test_is_triangle function

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

bool is_triangle(double side_a, double side_b, double side_c);

void test_is_triangle(double side_a, double side_b, double side_c,
                      bool expected);
```

test_is_triangle function

```
void test_is_triangle(double side_a, double side_b, double side_c,
                      bool expected) {
    if (is_triangle(side_a, side_b, side_c) != expected) {
        printf("Tests failed for values %lf, %lf, %lf!\n", side_a, side_b,
               side_c);
    }
}
```

Multiple possible test case scenarios

- Useful for testing assignments
- Software testing

```
// all sides equal
test_is_triangle(1, 1, 1, true);
// a + b is slightly larger than c
test_is_triangle(1.001, 2, 3, true);
// a + c is slightly larger than b
test_is_triangle(2, 3, 1.001, true);
// a + b is slightly larger than c
test_is_triangle(2, 1.001, 3, true);
// a + b < c
test_is_triangle(1, 2, 10, false);
// a + b == c
test_is_triangle(1, 2, 3, false);
// a + c < b
test_is_triangle(1, 5, 1, false);
// a + c == b
test_is_triangle(1, 2, 1, false);
// b + c < a
test_is_triangle(3, 1, 1, false);
// b + c == a
test_is_triangle(3, 2, 1, false);
```

Dividing Program into modules: **triangle.c**

triangle.c > ...

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

bool is_triangle(double side_a, double side_b, double side_c);

bool is_triangle(double side_a, double side_b, double side_c) {
    return side_a + side_b > side_c &&
           side_a + side_c > side_b &&
           side_b + side_c > side_a;
}
```

Dividing Program into modules: **triangle.h**

C triangle.h > ...

```c
#include <stdbool.h>

bool is_triangle(double side_a, double side_b, double side_c);

```

Dividing Program into modules: **triangle.c**

C triangle.c > ...

```c
#include "triangle.h"

bool is_triangle(double side_a, double side_b, double side_c);

bool is_triangle(double side_a, double side_b, double side_c) {
    return side_a + side_b > side_c &&
           side_a + side_c > side_b &&
           side_b + side_c > side_a;
}
```

Dividing Program into modules: **triangle.c**

C triangle.c > ...

```c
#include "triangle.h"

bool is_triangle(double side_a, double side_b, double side_c) {
    return side_a + side_b > side_c &&
           side_a + side_c > side_b &&
           side_b + side_c > side_a;
}
```

Dividing Program into modules: test_triangle.c

test_triangle.c > ...

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

#include "triangle.h"

void test_is_triangle(double side_a, double side_b, double side_c,
                      bool expected);

int main() {
    // all sides equal
    test_is_triangle(1, 1, 1, true);
    // a + b is slightly larger than c
    test_is_triangle(1.001, 2, 3, true);
    // a + c is slightly larger than b
    test_is_triangle(2, 3, 1.001, true);
    // a + b is slightly larger than c
    test_is_triangle(2, 1.001, 3, true);

```

Modules in C

- Many programs are too complex to live in one file
  - Complex programs are organized into modules
  - A module containing main() is referred to as an entry point
  - A module in C that is not an entry point is typically divided into 2 files
    - Header file
      - Extension is .h
      - Contains the function prototypes for the module (the module’s interface)
      - Reading a module's header file is sometimes enough to figure out how to use the functions in the module
    - Implementation file
      - Extension is .c
      - Contains the implementations of each function declared in the header

Compiling multiple c files using gcc

```
(base) zwkbhowmiknb02:module kbhowmik$ gcc triangle.c test_triangle.c -o test_triangle
(base) zwkbhowmiknb02:module kbhowmik$ ./test_triangle
(base) zwkbhowmiknb02:module kbhowmik$
```

Separately compiling triangle.c

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
(base) zwkbhowmiknb02:module kbhowmik$ gcc triangle.c
Undefined symbols for architecture x86_64:
  "_main", referenced from:
     implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
(base) zwkbhowmiknb02:module kbhowmik$
```

Creating object files

```
(base) zwkbhowmiknb02:module kbhowmik$ gcc triangle.c -c
(base) zwkbhowmiknb02:module kbhowmik$
```

Creating object files

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
(base) zwkbhowmiknb02:module kbhowmik$ gcc -c test_triangle.c
(base) zwkbhowmiknb02:module kbhowmik$
```

Linking object files

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE bash

```
(base) zwkbhowmiknb02:module kbhowmik$ gcc test_triangle.o triangle.o -o test_triangle
(base) zwkbhowmiknb02:module kbhowmik$
```

Compliling Modules

- Typically each module is independently compiled into an object file
- gcc can link object files together to form an executable
- Compiling object files separately is advantageous because when a single implementation file is edited, only one object file needs to be re-compiled

Makefile

```
# A Makefile builds targets. A target has the following syntax:
# <target name>: <target dependencies>
#       <command>
#       <optionally more commands>
#
```

Makefile

```
test_triangle: triangle.o test_triangle.o
    gcc triangle.o test_triangle.o -o test_triangle

triangle.o: triangle.c triangle.h
    gcc -c triangle.c

test_triangle.o: test_triangle.c triangle.h
    gcc -c test_triangle.c

clean:
    rm -f test_triangle test_triangle.o triangle.o
```

Using Makefile to compile program

```
(base) zwkbhowmiknb02:module kbhowmik$ make
gcc -c triangle.c
gcc -c test_triangle.c
gcc triangle.o test_triangle.o -o test_triangle
```

Why use Makefile

```
void test_is_triangle(double side_a, double side_b, double side_c,
                      bool expected) {
    if (is_triangle(side_a, side_b, side_c) != expected) {
        printf("Tests failed for values %lf, %lf, %lf.\n", side_a, side_b,
               side_c);
    }
}
```

```
(base) zwkbhowmiknb02:module kbhowmik$ make
gcc -c test_triangle.c
gcc triangle.o test_triangle.o -o test_triangle
(base) zwkbhowmiknb02:module kbhowmik$
```

```
CFLAGS=-std=c99 -Wall

test_triangle: triangle.o test_triangle.o
    gcc $(CFLAGS) triangle.o test_triangle.o -o test_triangle

triangle.o: triangle.c triangle.h
    gcc $(CFLAGS) -c triangle.c

test_triangle.o: test_triangle.c triangle.h
    gcc $(CFLAGS) -c test_triangle.c
```

- CFLAGS is a variable which stores gcc compiler options for the project.
- To use the variable, you need to use $(CFLAGS) as part of the command

```
clean:
    rm -f test_triangle test_triangle.o triangle.o
```

```
(base) zwkbhowmiknb02:module kbhowmik$ make clean
rm -f test_triangle test_triangle.o triangle.o
(base) zwkbhowmiknb02:module kbhowmik$
```

clean

- clean is a special target because it does not build anything. Instead, it
- removes the executable and the object file. Run "make clean" to clean up the compiled files.

make

- Tool to automate compiling
- Saves time and makes the compiling process less error prone
- Rules for compiling are defined in a file named Makefile
- The entire project can be compiled by typing make