2 - C++ Basics: Functions and Files

2.1 - Introduction to functions

- A **function** is a reusable sequence of statements designed to do a particular job.
- Functions provide for a way to split our program into modular chunks.
- The function initiated in the function call is called the **caller** and the function being called is the **callee**
- General format of a user defined function:

```
void identifier(){ //identifier replaced by the name of the function
//code goes here
}
```

- Calling functions more that once

```
void doPrint() //my user-defined function
{
    std::cout <<"HoHoHo christmas came early this year \n";
}

int main(){

    std::cout <<"Starting main()... \n";
    doPrint(); // function interrupted to call doPrint()
    doPrint();
    std::cout <<"Ending main().. \n";
    return 0;
}
```

- Functions calling function calling functions

```
# include <iostream>

void callB()
{
    std::cout <<"you my friend have come a long way \n";
}
void callA()
{
    std::cout <<"Starting callA() .. \n";
    callB();
    std::cout <<"Ending callA() .. \n";
}

int main()
{
    std::cout <<"Starting main()... \n";
```

```
    callA(); // function interrupted to call doPrint()
    std::cout <<"Ending main().. \n";
    return 0;
}
```

- Unlike other programming languages, in C++ **Nested functions** aren't allowed, i.e functions can't be defined within functions.
- In a function definition, the **function body** is the braces and statements in-between.

2.2 - Function return values

- Until now we have never returned values, we're going to do just this now.
- Two things are required to return value to caller:
  - Indicate the type of value the function will return.
  - use a return statement within the function.
- Below is a little example:

```
#include <iostream>

int get_digit(){

    std::cout << "Enter a digit please : ";
    int x{};
    std::cin >> x;
    return x;
}

int sums(int a, int b){
    return a+b;
}

int subs(int a, int b){
    return a-b;
}

int mults(int a, int b){
    return a*b;
}

int divs(int a, int b){
    return a/b;
}

int get_choice(){

    std::cout << "Please choose one of the following options : \n" ;
    std::cout << "1 - Add the two digits \n";
    std::cout << "2 - Substact the two digits \n";
    std::cout << "3 - Multiply the two digits \n";
    std::cout << "4 - Divide the two digits \n";

    int choice{};
    std::cin >> choice;

    return choice;
}


int main()
{
    std::cout << "Welcome to the calculator \n" ;

    // Intializations
    int a{};
    int b{};
    int c{};
    int result{};
    bool loop = true;
```

```
    // Get user inputs
    a = get_digit();
    b = get_digit();

    while (loop == true){

        c = get_choice();

        if (c == 1){
            result = sums(a, b);
        }
        else if (c == 2){
            result = subs(a, b);
        }
        else if (c == 3){
            result = mults(a, b);
        }
        else if (c == 4){
            result = divs(a, b);
        }
        else {
            std::cout << " Got invalid value of : " << c << "\n";
            std::cout << " Please choose values between 1 and 4 \n";
            continue;
        }

        std::cout << "Result : " << result <<"\n";
        std::cout << "Do you want to continue (Y/N) : ";
        std::string cont;
        std::cin >> cont;

        if (cont == "Y"){
            loop = true;
        }
        else {
            break;
        }
        a = get_digit();
        b = get_digit();
    }


    return 0;

}
```

- A value of 0 is returned after main function. This value is called a **status code**, and it tells the operating system (and any other programs that called yours) whether your program executed successfully or not.
- Always explicitly provide a return value for any function that has a non-void return type.
- When a return statement is executed, the function returns back to the caller immediately at that point. Any additional code in the function is ignored.
- A function can only return a **single** value back to the caller each time it is called.
- Does this program compile ?

```
#include <iostream>
void printA()
{
    std::cout << "A\n";
}

int main()
{
    std::cout << printA() << '\n';

    return 0;
}
```

2.3 - Introduction to function parameters and arguments

- A function parameter is a variable used in function. Function parameters work almost identically to variables defined inside the function, but with one difference: they are always initialized with a value provided by the caller of the function.
- **Warning** about function argument order of evaluation (couldn't reproduce error)
  - The C++ specification does not define whether arguments are matched with parameters in left to right order or right to left order. When copying values, is of no consequence. However, if the arguments are function calls, then this can be problematic: `someFunction(a(), b()); // a() or b() may be called first`
  - So do this instead (you should explicitly define the order of executio)

```
int a{ a() }; // a() will always be called first
int b{ b() }; // b() will always be called second
someFunction(a, b); // it doesn't matter whether a or b are copied first because they are just values
```

2.4 - Introduction to local scope

- Local variable lifetime
  - Function parameters are created and initialized when the function is entered, and variables within the function body are created and initialized at the point of definition
  - Local variables are destroyed in the opposite order of creation at the end of the set of curly braces in which they are defined
  - An object's lifetime is defined to be the time between its creation and destruction.
  - Variable creation and destruction happen when the program is running (called runtime), not at compile time. Therefore, lifetime is a runtime property
- Local scope
  - An identifier's scope determines where the identifier can be accessed within the source code.
  - Lifetime is a runtime property and scope is a compile time property

```
#include <iostream>

int add(int x, int y) // x and y are created and enter scope here
{
    // x and y are visible/usable within this function only
    return x + y;
} // y and x go out of scope and are destroyed here

int main()
{
    int a{ 5 }; // a is created, initialized, and enters scope here
    int b{ 6 }; // b is created, initialized, and enters scope here

    // a and b are usable within this function only
    std::cout << add(a, b) << '\n'; // calls function add() with x=5 and y=6

    return 0;
} // b and a go out of scope and are destroyed here
```

  - Now an example where a is replaced by x and b by y

```
int add(int x, int y) // add's x and y are created and enter scope here
{
    // add's x and y are visible/usable within this function only
```

```
    return x + y;
} // add's y and x go out of scope and are destroyed here

int main()
{
    int x{ 5 }; // main's x is created, initialized, and enters scope here
    int y{ 6 }; // main's y is created, initialized, and enters scope here

    // main's x and y are usable within this function only
    std::cout << add(x, y) << '\n'; // calls function add() with x=5 and y=6

    return 0;
} // main's y and x go out of scope and are destroyed here
```

- Why does the program still work?
  - First, we need to recognize that even though functions main and add both have variables named x and y, these variables are distinct. The x and y in function main have nothing to do with the x and y in function add -- they just happen to share the same names
  - Second, when inside of function main, the names x and y refer to main's locally scoped variables x and y. Those variables can only be seen (and used) inside of main. Similarly, when inside function add, the names x and y refer to function parameters x and y, which can only be seen (and used) inside of add.
  - Neither add nor main know that the other function has variables with the same names. Because the scopes don't overlap, it's always clear to the compiler which x and y are being referred to at any time.

2.5 - Why use functions?

- For the following reasons:
  - Organization
  - Resuability
  - Testing
  - Extensibility
  - Abstraction
- Effectively using functions:
  - Statement that appear more than once are generally turned to functions.
  - Code that has a well defined set of inputs and outputs is a good candidate for a function.
  - A function should only perform one task.
  - If functions are too long and complex it's better it's split into multiple sub-functions : **Code Refactoring**

2.6 - Whitespace and basic formatting

- Whitespace is the term used for characters that are used for formatting purposes.
- C++ is a whitespace independent language, as its complier ignores whitespaces.
- Common C++ conventions - Best practices

2.7 - Forward declarations and definitions

```
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

- This code wouldn't compile as when the compiler reaches the function call to add on line 5 of main, it doesn't know what add is, because we haven't defined add until line 9!
- To overcome this we have two options :
  - Reorder function calls : define add() before main() ; but this wouldn't work if we have A() and B() and within A() a call is made to B() while the same occurs within B().
  - **Forwards declaration**
    - To do so a declaration statement called a **function prototype** is used. It consists of the function's return type, name, parameters, but no function body
    - The program will now compile

```
int add(int x, int y); // forward declaration of add() (using a function prototype)

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n'; // this works because we forward declared add() above
    return 0;
}

int add(int x, int y) // even though the body of add() isn't defined until here
{
    return x + y;
}
```

- Declarations vs Definitions
  - A **definition** actually implements (for function or types) or instantiates (for variable) the identifier. A definition is needed to satisfy the linker. If you use an identifier without providing a definition, the linker will error. The one definition rule (or ODR for short) is a well-known rule in C++

```
int add(int x, int y) // implements function add()
{
    int z{ x + y }; // instantiates variable z

    return z;
}
```

  - A **declaration** is a statement that tells the compiler the existence of an identifier and its type information .

```
int add(int x, int y); // tells the compiler about a function named "add" that
                       // takes two int parameters and returns an int.  No body!
int x; // tells the compiler about an integer variable named x
```

    A declaration is all that is needed to satisfy the compiler. This is why we can use a forward declaration to tell the compiler about an identifier that isn't actually defined until later. But if a function definition is not present this would lead to a linker error.

- In C++, all definitions also serve as declarations. This is why int x appears in our examples for both definitions and declarations. Since int x is a definition, it's a declaration too.
- While it is true that all definitions are declarations, the converse is not true: all declarations are not definitions.

2.9 - Naming conflicts and std namespaces

- The code below will compile but, when the linker begins to link all files a error will be thrown:

```
a.cpp file
#include <iostream>

void myFcn(int x)
{
    std::cout << x;
}


main.cpp file
void myFcn(int x)
{
    std::cout << 2 * x;
}

int main()
{
    return 0;
}
```

- **The std namespace**
  - A namespace is a grouping of identifiers that is used of reduce the possibility of naming collisions. It turns out that std::cout's name isn't really std::cout. It's actually just cout, and std is the name of the namespace that identifier cout is part of.
  - Identifiers defined inside a namespace won't conflict with identically name identifiers defined outside of the namespace.
    - std::cout wouldn't conflict with some foo::cout that was defined inside a namespace named foo

2.10 - Introduction to the preprocessor

- Prior to code compilation in C++ the code goes through a phase called translation
- Many things happen in **translation** phase to get the code ready to be compiled
- Preprocessor is the most noteworthy of all phases during translation.
- The **preprocessor** is best thought of as a separate program that manipulates the text in each code file.
- When the preprocessor runs it scans the code for preprocessor **directives.**
  - These directives perform specific particular text manipulation tasks.
  - The output of the preprocessor goes through several more translation phases, and then is compiled
- The **include directive (# include <file>)**
  - When one # includes a file, the preprocessor replaces the **# include directive** with the preprocessed contents of the included file, which is then complied.
- **Macro defines**
  - A macro is a rule that defines how input text is converted to replacement text.

- There are 2 basic types of macros : object-like and function-like macros.
- **Function-like macros** : act like functions and have a similar purpose
- **Object-like macros** : can be defined in two ways:

```
# define identifier // they don't end with a semi colon
# define indentifier substitution_text

// object-like macros with substitution text
#define MY_NAME "Alex"

int main()
{
    std::cout << "My name is: " << MY_NAME;
    return 0;
}
```

  - object like macros can also be defined without substitution text, so, any further occurrence of the identifier is removed and **replaced by nothing!**
- **Conditional compilation #ifdef, #ifndef, and #endif**
  - They allow one to specify under what conditions something will or won't compile.
  - **#ifdef** : this preprocessor derivative allows the preprocessor to check if an #identifier has been previously defined. If so, the code between the #ifdef and **matching** #endif is **compiled.**

```
#include <iostream>

#define PRINT_JOE

int main()
{
#ifdef PRINT_JOE // if PRINT_JOE is defined
    std::cout << "Joe\n"; // execute this code
#endif

#ifndef PRINT_BOB // if PRINT_BOB is not defined
    std::cout << "Bob\n"; // execute this code
#endif

#if 0 // Don't compile anything starting here
    std::cout << "Bob\n";
    std::cout << "Steve\n";
#endif // until this point

    return 0;
}
```

  - **#ifndef** : opposite of #ifdef, in that it allows you to check whether an identifier has NOT been #defined yet
  - **#if 0** : excludes a block of code from being compiled
- Macros only cause text substitution for normal code. Other preprocessor commands are ignored

- Directives are resolved before compilation, from top to bottom on a file-by-file basis.

```
#include <iostream>

void foo()
{
#define MY_NAME "Alex"
}

int main()
{
  std::cout << "My name is: " << MY_NAME;
```

```
  return 0;
}
/*Even though it looks like #define MY_NAME “Alex” is defined inside
function foo, the preprocessor won’t notice, as it doesn’t understand
C++ concepts like functions*/


function.cpp
#include <iostream>

void doSomething()
{
#ifdef PRINT
    std::cout << "Printing!";
#endif
#ifndef PRINT
    std::cout << "Not printing!";
#endif
}


main.cpp
void doSomething(); // forward declaration for function doSomething()
#define PRINT

int main()
{
    doSomething();
    return 0;
}
/*
The above program will print: not printing

Even though PRINT was defined in main.cpp, that doesn’t have
any impact on any of the code in function.cpp (PRINT is only
#defined from the point of definition to the end of main.cpp)
*/
```

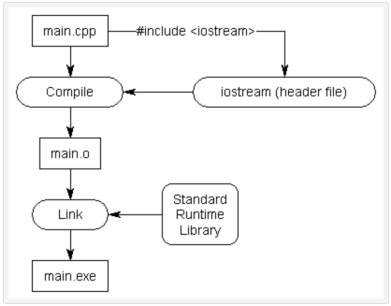
2.11 - Header Files

- Header files allows for forward declaration of multiple function within a .h or .hpp file, which can then be imported by the main function, supplanting all the hassle concerning forward declaring all functions at the beginning of the main.cpp file.
- Check out the code below:

```
# include <iostream>

int main()
{
    std::cout << "Hello, world!";
    return 0;
}
```

std::cout has been forward declared in the “iostream” header file. When we #include <iostream>, we’re requesting that the preprocessor copy all of the content (including forward declarations for std::cout) from the file named “iostream” into the file doing the #include.



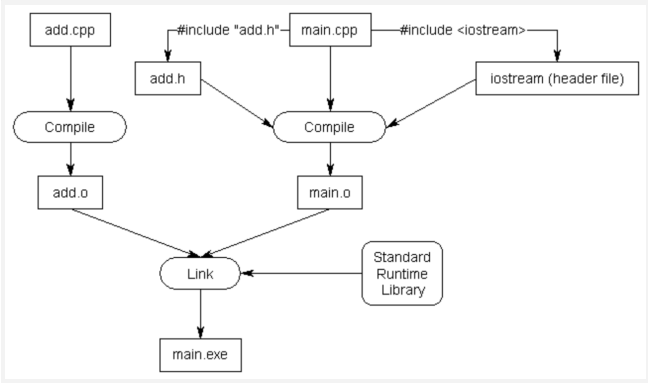
- Header files are paired with code files, with header files providing forward declarations for the corresponding code files.

```
add.hpp
int add(int x, int y);

add.cpp
int add(int x, int y)
{
  return x + y
}

main.cpp
# include <iostream> // here contents of the file iostream is added
# include "add.hpp" //here contents of file add.hpp is added
int main()
{
  std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
  return 0;
}
```

  - As seen from the code above, the introduction of the .hpp file header allows the main.cpp file to access the content of the .hpp fil



  - Why angled brackets for iostream and quotes for .hpp

    Angled brackets are used to tell the preprocessor that we are including a header file that was included with the compiler, so it should look for that header file in the system directories. The double-quotes tell the preprocessor that this is a user-defined header file we are supplying, so it should look for that header file in the current directory containing our source code firs

- Header files best practices:
  - Always include header guards.
  - Do not define variables and functions in header files.

- Give your header files the same name as the source files they're associated with (e.g. *grades.hpp* is paired with *grades.cpp*).
- Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h.
- Be mindful of which headers you need to explicitly include for the functionality that you are using in your code files
- Every header you write should compile on its own.
- Only #include what you need (don't include everything just because you can).
- Do not #include .cpp files.
- Order your #includes as follow: your own user-defined headers first, then 3rd party library headers, then standard library headers

2.12 - Header Guards

- When there's redundancy in object and function declarations we often end up with compiler errors:

```
int main()
{
    int x; // this is a definition for variable x
    int x; // compile error: duplicate definition

    return 0;
}

#include <iostream>

int foo() // this is a definition for function foo
{
    return 5;
}

int foo() // compile error: duplicate definition
{
    return 5;
}

int main()
{
    std::cout << foo();
    return 0;
}
```

- Similarly, following is the issue when we declare headers which contain functions with similar identifiers

```
square.hpp
// We shouldn't be including function definitions in header files
// But for the sake of this example, we will
int getSquareSides()
{
    return 4;
}

geometric.hpp
#include "square.hpp"

main.cpp
#include "square.hpp"
#include "geometry.hpp"
int main()
{
    return 0;
}
```

```
//after resolving all of the #includes, main.cpp ends up looking like this:
int getSquareSides()  // from square.h
{
    return 4;
}
int getSquareSides() // from geometry.h (via square.h)
{
    return 4;
}
int main()
{
    return 0;
```

First, main.cpp #includes square.hpp, which copies the definition for function getSquareSides into main.cpp. Then main.cpp #includes geometry.hpp, which #includes square.hpp itself. This copies contents of square.h (including the definition for function getSquareSides) into geometry.hpp which then gets copied into main.cpp.

- The above problem can be supplanted using header guards. They are conditional compilation directives that take the following form:

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations (and certain types of definitions) here

#endif
```

When this header is #included, the preprocessor check whether SOME_UNIQUE_NAME_HERE has been previously defined.
- If this is the first time we've included the header, SOME_UNIQUE_NAME_HERE will not have been defined.
- Consequently, it #defines SOME_UNIQUE_NAME_HERE and includes the contents of the file. If the header is included again into the same file, SOME_UNIQUE_NAME_HERE will already have been defined from thew first time the contents of the header were included, and the contents of the header will be ignored (thanks to the #ifndef).

- This how we overcome the build error faced when executing the above program

```
square.h
#ifndef SQUARE_H
#define SQUARE_H
int getSquareSides()
{
    return 4;
}
#endif

geometry.h
#ifndef GEOMETRY_H
#define GEOMETRY_H
#include "square.h"
#endif

main.h
#include "square.h"
#include "geometry.h"
int main()
{
    return 0;
}

main.cpp
#ifndef SQUARE_H // square.h included from main.cpp,
#define SQUARE_H // SQUARE_H gets defined here
// and all this content gets included
int getSquareSides()
{
    return 4;
}
#endif // SQUARE_H
```

```
#ifndef GEOMETRY_H // geometry.h included from main.cpp
#define GEOMETRY_H
#ifndef SQUARE_H // square.h included from geometry.h, SQUARE_H is already defined from above
#define SQUARE_H // so none of this content gets included

int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H
#endif // GEOMETRY_H

int main()
{
    return 0;
}
```

2.13 - How to design your first program ?

- **Define your goal** : a few sentences
  - Allow the user to organize a list of names and associated phone numbers.
  - Generate randomized dungeons that will produce interesting looking caverns.
  - Generate a list of stock recommendations.
  - Model how long it takes for a ball dropped off a tower to hit the ground.
- **Define requirements** : constraints (e.g. budget, timeline, space, memory, etc…) that the solution needs to abide by and capabilities (focused on the “what”, not the “how”) the program must exhibit.
- **Define your tools, targets and backup plan**:
  - Defining what target architecture and/or OS your program will run on.
  - Determining what set of tools you will be using.
  - Determining whether you will write your program alone or as part of a team.
  - Defining your testing/feedback/release strategy.
  - Determining how you will back up your code.
- **Break hard problems into easy ones**
  - Get from bed to work
    - Bedroom things
      - Get out of bed
      - Pick out clothes
      - Get dressed
    - Bathroom things
      - Take a shower
      - Brush your teeth
    - Breakfast things
      - Prepare cereal
      - Eat cereal
    - Transportation things

        - Get in your car
        - Drive to work
- **Figure out sequence of events** : determine the sequence of events that will be performed
- **Implementations steps :**
    - Step 1: outline main function

```
int main()
{
//    doBedroomThings();
//    doBathroomThings();
//    doBreakfastThings();
//    doTransportationThings();

    return 0;
}
```

    - Step 2: implement each function
    - Step 3: Final testing (check if program works as envisioned