Unit 2 – C/C++ Language Programming

C PROGRAMMING

- General purpose programming language that features economy of expression, flow control, data structures, and a rich set of operators.
- Reference: B. W. Kernigham, D. M. Ritchie, *C Programming Language*, 2nd ed.

C BASICS

- **Data types**: each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it.
  - ✓ `size_t`: alias for a fundamental unsigned type. It is used as the return type by the `sizeof` operator. It is guaranteed to hold the size of the biggest object the host system can handle. Thus, the specific size is dependent on the compiler.

TABLE I. COMMON DATA TYPES IN C

Data Type	# of bytes	Comments
`char / unsigned char`	1	
`int / unsigned int`	2	16-bit machine
	4	32-bit machine
`long int / unsigned long int`	4	16-bit machine
	8	32-bit machine
`long long int`	8	
`size_t`	4	32-bit machine: unsigned int
	8	64-bit machine: unsigned long long int
`float`	4	32-bit floating point
`double`	8	64-bit floating point

- **Flow Control**:
  - ✓ Branching (what actions to take): `if statement, switch statement`
  - ✓ Looping (how many times to take a certain action): `while loop, for loop, do while loop`

- **Input/Output Devices**:
  - ✓ Standard Files: C treats all devices as files. Devices such as display are addressed in the same way as files. Table II lists the files that are automatically opened when a program executes in order to provide access to the keyboard and screen.

TABLE II. STANDARD FILES IN C. THE FILE POINTERS ARE THE MEANS TO ACCESS THE FIE FOR READING/WRITING PURPOSES

Standard File	File Pointer	Device
`Standard input`	`stdin`	Keyboard
`Standard output`	`stdout`	Screen
`Standard error`	`stderr`	Screen

  - ✓ Input: feed data into a program (file, command line)
    - `scanf`: It reads characters from the standard input stream `stdin` and scans them as per the provided format.
    - `sscanf`: It reads from a string instead of the standard input.
  - ✓ Output: display data on screen, printer, or file.
    - `printf`: Prints character stream of data on `stdout` console.
    - `sprintf`: Instead of printing on console, it stores stream on `char` buffer.

- **Input/Output Files**:
  - ✓ File: sequence of bytes. We can create, open, read, write, and close text as well as binary files.
  - ✓ Open file. `FILE* fopen (const char * filename, const char * mode);`
    - Opens a file and associates it with a stream that can be later identified by the `FILE` pointer returned.
  - ✓ Close File: `int fclose (FILE *stream);`
    - Closes the file associated with the stream. If the stream is successfully closed, a zero value is returned.

  - ✓ Writing on a text file:
    - `fputc`: writes a character to the stream. `int fputc (int character, FILE *stream)`
    - `fputs`: writes a string to stream. `int fputs (const char *str, FILE *stream)`
      - It copies from the address `str` until it reaches the terminating null character `\0`, which is not copied to the file.
    - `fprintf`: Writes formatted data to stream. `int fprintf (FILE *stream, const char * format, …)`
      - Instead of printing on console (`printf`), it prints stream in a file.

- ✓ Reading from a text file:
  - `fgetc`: gets character from stream. `int fgetc (FILE *stream)`
  - `fgets`: get string from stream. `char* fgets (char *str, int num, FILE *stream)`
    - It stores the characters in `str` until `num-1` characters have been read of a new line of end-of-file is reached (whichever happens first). A terminating null character is appended after characters are copied to `str`.
  - `fscanf`: Reads formatted data form stream. `int fscanf (FILE *stream, const char *format, ….)`
    - Instead of reading from `stdin` (`scanf`), it reads data from a file.

- ✓ Example (read/write text files):

```
int main () {
   char buff[33];
   FILE *file_i, *file_o;

   // Opening textfile with input matrix: raster scan fashion.
   file_i = fopen("input.txt","r");
   if (file_i == NULL) return -1; // check that the file was actually opened
   file_o = fopen("output.txt","r");
   if (file_o == NULL) return -1; // check that the file was actually opened

   fgets (buff, 32, file_i); // read 32 characters and place it on buff.
   fclose (file_i);

   fputs ("This is a test for fputs… \n");
   fclose (file_o);
   return 0;
}
```

- ✓ Reading/writing binary files:
  - `fread`: Read block of data from stream. It reads an array of `count` elements, of size `size`, and stores them on `ptr`.
    - Syntax: `size_t fread (void *ptr, size_t size, size_t count, FILE* stream);`
  - `fwrite`: Write block of data to stream. It writes an array of `count` elements of size `size` from `ptr` and places it in the current position in the stream.
    - Syntax: `size_t fwrite (const void * ptr, size_t size, size_t count, FILE* stream);`

POINTERS AND ARRAYS

- A pointer is a variable that contains the address of a variable. In a 32-bit system, it is represented by a 32-bit value (usually printed as a string of 8 hexadecimal characters).

- Example:

```
int main () {
   int x, y, z[10];

   x = 1; y = 2;
   int *ptr; // Declaration of pointer to an integer
             // type variable

   ptr = &x; // the address of x is assigned to ptr
   y = *ptr; // y = x = 1
   *ptr = 0; // x = 0

   ptr = &z[0]; // ptr points to z[0]

   printf ("%d", x);
   printf ("%p", ptr); // prints address of z[0]
   return 0;
}
```

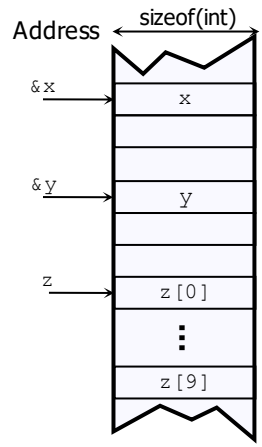

Figure 1. Pointer examples

- Example (arithmetic with pointers)

```
int main() {
   int val[3] = {10, 50, 90}; // array declaration
   int *ptr; // pointer to int declaration

   ptr = val; // Assign the address of val[0] to ptr

   for (int i = 0; i < 3; i++) {
      printf ("Value of *ptr = %d\n", *ptr);
      printf ("Value of ptr = %p\n\n", ptr);
      ptr++; // ptr = ptr + 1 (int)
    }
}
```

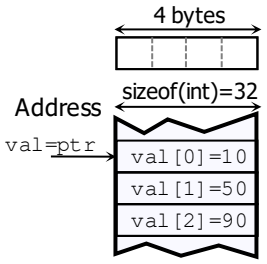

Figure 2. Array after the program executes. sizeof(int) = 32 bits.

- ✓ Output (assuming address is 32-bits wide and `ptr` is assigned to `0x7ffcae3`)

```
Value of *ptr = 10
Value of ptr = 0x7ffcae0

Value of *ptr = 50
Value of ptr = 0x7ffcae4

Value of *ptr = 90
Value of ptr = 0x7ffcae8
```

- ✓ A pointer can be incremented and decremented. In computer systems, the minimum granularity for an address is one byte. Thus, these operations update the pointer value by as many bytes the variable it points to holds:
  - If `ptr` is a pointer to `int`, then `ptr++` means that the pointer `ptr` value increased by 4.
  - If `ptr` is a pointer to `char`, then `ptr++` means that the pointer `ptr` value increases by 1.

- **Relationship between arrays and pointers**: An array name acts like a pointer. The value of the pointer is the address of the first element of the array. In the previous program, `val` and `&val[0]` can be used interchangeably. Right after we execute `ptr = val` (and before the next instructions) we had that `ptr[0] = val[0]`, `ptr[1] = val[1]`, `ptr[2] = val[2]`.

MEMORY MANAGEMENT

- **Memory Layout of C programs**:
  - ✓ Text segment (or code segment): Area in memory that stores the instruction codes. It contains executable (machine code) instructions. Usually read-only, to prevent a program from accidentally modifying its instructions.
  - ✓ Data segment:
    - Initialized data segment (or data section): It contains data elements stored for the program (global, static, constant variables, as well as external variables). These are variables specifically initialized in the program.
    - Uninitialized data segment (or bss section). This is a read-write fixed-size segment, since the values of variables can change during run-time. It contains all global variables and static variables that are initialized to 0 or do not have explicit initialization in source code (e.g.: `static int i` within `main()`, `'char c'` as a global variable).

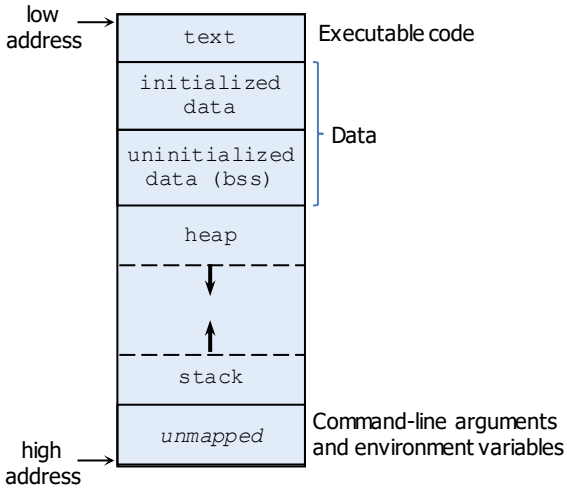

Figure 3. Typical memory layout in C

  - ✓ Stack: It contains the program stack (LIFO structure). A Stack Pointer (SP) register tracks the top of the Stack. The set of values pushed for a function call is termed a 'stack frame'. It stores all local variables and it is used for passing arguments to the functions along with the return address. At a minimum, a stack frame consists of a return address.
  - ✓ Heap: Dynamically allocated memory to a process during run-time. It is managed by `malloc`, `calloc`, `realloc`, and `free`. This area is shared by all threads, shared libraries, and dynamically loaded modules in a process.

- **Dynamic Memory allocation**: Procedure in which the size of a data structure (e.g.: array) is changed during run-time. There are four library functions defined under `<stdlib.h>` that facilitate this procedure:
  - ✓ `malloc()`: 'Memory allocation'. Dynamically allocates a single large block of memory with the specified size. Allocated memory is not initialized. If we try to access the content of memory block before initializing, we will get segmentation fault error (or garbage values). `malloc` is faster than `calloc`. Syntax: `ptr = (cast-type *) malloc (byte-size)`.
    - Example:

```
int *ip; // pointer declaration
ip = (int *) malloc (100*sizeof(int));
```

For 4-byte `int`, this statement allocates 400 bytes of memory.
If space in the heap is insufficient, it returns a NULL pointer

  - ✓ `calloc()`: 'Contiguous allocation'. Dynamically allocates the specified number of blocks in memory. It initializes each block with a default value '0'. Syntax: `ptr = (cast-type *) calloc (n, element-size)`
    - Example (4-byte `int`):

```
int *ip; // pointer declaration
ip = (int *) calloc (50,sizeof(int));
```

This statement allocates contiguous space in memory for 50 elements each with the size of `int`.
If insufficient space in heap, it returns a NULL pointer

  - ✓ `free()`: The memory allocated by `malloc/calloc` is not de-allocated on their own. Hence, the `free()` method is used to de-allocate the memory. This helps reduce wastage of memory. Syntax: `free (ptr)`.
    - If you create memory blocks in heap and forget the delete them, ***memory leaks*** occur. These can be serious issues on real-world programs that by definition never terminate. Thus, always keep track of allocated memory in order to free it when not needed.

- ✓ `realloc`: 'Re-allocation'. Dynamically changes the memory allocation of a previously allocated memory. It deallocates the old memory block and returns a pointer to a new memory block. It maintains the already present values and new blocks are initialized with garbage. Syntax: `ptr = realloc (ptr, newSize).`
  - Example (4-byte `int`):

```
int *ptr;
ptr = (int *) malloc (5*sizeof(int));

ptr = realloc (ptr, 10*sizeof(int));
```

A 20-byte memory block is initially allocated to `ptr`.

`ptr` is updated. The size of the memory block pointed by this new `ptr` changes from 20 to 40 bytes at run-time. If insufficient space in heap, it returns a NULL pointer

FUNCTIONS

- A function provides a convenient way to encapsulate some computation. It is then possible to ignore *how* a job is done; knowing *what* is done is sufficient.
- Function definition:

```
return_type function_name(parameter list) { // there might be no arguments to the function
   declarations
   statements
}
```

- Every function has a return type.
  - ✓ If a function does not return any value, then `void` is used as return type.
  - ✓ Functions can return any type except arrays and functions. A workaround this limitation is to return a pointer to an array or pointer to function.
- **Example**:

```
#include <stdio.h>
int power(int m, int n); // function prototype

/* test power function */
int main() { // main is a special function
   int i;
   for (i = 0; i < 10; ++i)
      printf("%d %d %d\n", i, power(2,i), power(-3,i)); // function power called twice
   return 0;
}

/* power: raise base to n-th power; n >= 0 */
int power(int base, int n) {
   int i, p;
   p = 1;
   for (i = 1; i <= n; ++i)  p = p * base;
   return p; // value computed by 'power function' and returned to main()
}
```

  - ✓ The function `power` is called twice by `main`. Each call passes two arguments to power, which each time returns an integer.
  - ✓ Generally, we use the term *parameter* for a variable named in the parenthesized list in a function.
    - The names used by `power` for its parameters are local to `power` and are not visible to any other function: other routines can use the same names without conflict. This is also true of the variables `i` and `p`.

- Function definitions can appear in any order, in one source file or several, although no function can be split between files. If the source program appears in several files, you may have to do more to compile and load it than if it all appears in one, but that is an operating system matter, not a language attribute.

PARAMETER PASSING TO A FUNCTION

Passing arguments by value

- The called function is given the values of its arguments in temporary variables rather than the originals. Parameters can be treated as conveniently initialized local variables in the called routine.
- Example:

```
int power(int base, int n) { /* power: raise base to n-th power; n >= 0; version 2 */
   int p;
   for (p = 1; n > 0; --n)
      p = p * base;
   return p;
}
```

  - ✓ The parameter `n` is a temporary variable and it is counted down until it becomes zero. However, whatever is done to `n` inside `power` has no effect on the argument that power was originally called with (i.e, in `main`, `n` keeps its value)

Passing arguments by reference

- Here, by passing arguments by reference, any changes made inside the function to a parameter are actually reflected in the caller routine.
- Parameters are always passed by value in C. However, we can use pointers to get the effect of pass by reference.

- **Example**:

```
void fun(int *ptr) {
    *ptr = 30;
}

int main() {
  int x = 20;
  fun(&x); // passing address of x to fun
  printf("x = %d", x);  // Output: 'x = 30'

  return 0;
}
```

  - ✓ The function `fun` expects a pointer `ptr` to an integer (i.e., an address to an integer). The function modifies the value located at the address `ptr` (it does not modify the value of the pointer `ptr`).

Main Function

- Special function included in every program. It serves as an entry point for the program.
- Two types:
  - ✓ Main function without parameters: `int main() { … }`
  - ✓ Main function with parameters: `int main (int argc, char *const argv[]) { … }`
    - The parameter option in the `main` function allows us to have input from the command line. Every group of characters (separated by a space) is saved as an array element of `argv`. The number of elements is given by `argc`.
- Note the `return` statement at the end of `main` in the examples in this section. The `main` function is like any other, it may return a value to its caller, which is in effect the environment in which the program was executed. Typically, a return value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions.

FUNCTION POINTERS

- They provide an efficient and elegant programming technique. You can use them to replace `switch/if` statements., and to implement *late binding*, which refers to deciding the proper function during runtime instead of compile time.
- Like pointers to fundamental types (`*int, *char, *float`), we can have pointers to functions.
- Function pointers are pointers (i.e., variables) which point to an address of a function. The running programs get a certain space in the main memory. Both the executable compiled program code and the used variables are placed in the memory. Thus, a function in the program code has an address.
- Function pointers allow a function to have functions as arguments. This is an elegant way to bind a function to a specific algorithm at runtime.
  - ✓ Example: We have a function that implements a sorting algorithm. At compilation time, we are not sure which sorting algorithm to use since we may not know the size of the data set n (provided by the user at runtime). If $n < 100$ we may use something like *insertion sort*, but if n is large, then we may decide to use something like *quicksort*.

Declaration and usage

- Standard function declaration: `int myfun (int);`
  - ✓ This function that returns `int` and takes an argument of `int` type.

- To declare a function pointer, the following looks as if it would work: `int *foo (int);`
  - The problem is that the operator `()` will take priority over operator `*`. Thus, it will declare a function `foo` with an argument of type `int` that returns a value of `int *`.
  - ✓ To properly declare a function pointer, we need to use the operator `()` around `*foo`. This binds the operator `*` with `foo`.
    - **`int (*foo) (int);`**

- To assign the address of a function to a function pointer, there are two alternatives:
  - ✓ `foo = &myfun;  // OR, we can just drop the '&'`
  - ✓ `foo =  myfun;`

    This is because function names are just like array names, they are pointers to the array they are referring to.

- To call the function via the function pointer, there are two alternatives:
  - ✓ `int x = (*foo)(50); // OR`
  - ✓ `int x = foo(50);`

    This is because the act of calling it (`foo(50)`) performs the dereference. There is no need to do the referencing yourself.

- **Example:**

```
#include <stdio.h>

void fun(int a) { // function that returns void and takes an argument of int type
    printf("Value of a is %d\n", a);
}
```

```
int main() {
   void (*fun_ptr)(int); // Declaration of a function pointer
   // If we just use void *fun_ptr (int), it would be declaring a function that returns a void pointe

   // Assigning the address of fun() to fun_ptr, i.e., fun_ptr is a pointer to function fun():
   fun_ptr = &fun;  // Note: fun_ptr = fun also works
   // void (*fun_ptr)(int) = &fun; // Declaration + assignment in one line

   // Invoking fun() using fun_ptr
   (*fun_ptr)(15); // Note: fun_ptr(15) also works.
    return 0;
}
```

✓ Program Output: `Value of a is 15`

- **Example:**

```
#include <stdio.h>

void add (int a, int b) {
     int sum;
     sum = a + b;
     printf ("The addition of %d and %d is: %d\n",a,b, sum);
}

int main ()
{
  void (*addPtr)(int, int); // Declaration of function pointer (no return value)

  addPtr = add;  // Alternative: addPtr = &add
  addPtr (13,65); // Alternative: (*addPtr)(13,65)
  return 0;
}
```

✓ Program Output: `The addition of 13 and 65 is: 78`

- Note that unlike normal pointers, a function pointer points to code, not data. And we do not allocate or de-allocate memory using function pointers.
- Like normal data pointers, we can have an array of function pointers.

Function pointers as arguments to functions

- In C, you cannot pass a function to another function as a parameter. But you can pass a function reference to another function using function pointers.
- Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function. In the following example, `wrapper()` receives a `void (*fun)()` as parameter and calls the passed function.
  - ✓ The declaration of the parameter in function `wrapper` amounts to the declaration of a function pointer.

```
#include <stdio.h>
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a function pointer as a parameter and calls the function
void wrapper(void (*fun)()) {
    fun(); // Alternative: *(fun)()
}

int main() {
    wrapper(fun1);  // Output: Fun1
    wrapper(fun2);  // Output: Fun2
    return 0;
}
```

  - ✓ `wrapper(fun1)`: We invoke `wrapper` with its parameter (function `fun1`). This is where the function is assigned to the function pointer (alternative: `wrapper(&fun1)`).

STRUCTURES

- A structure is a user-defined data type. It is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
- Examples:

```c
#include <stdio.h>
struct Point {
   int x, y;
};

int main() {
   struct Point p1 = {0, 1}; // Initialization of
                             // structure members

   // Accessing members of Point p1:
   p1.x = 20;
   printf ("x = %d, y = %d", p1.x, p1.y);
   return 0;
}
```

✓ Program Output: `x=20, y = 1`

```c
#include <stdio.h>
struct StudentData {
   char *stu_name;
   int stu_id;
   int stu_age;
};

int main() {
   struct StudentData st;
   // Accessing members of structure:
   st.stu_name = "Daniel";
   st.stu_id = 1618; st.stu_age = 30;

   printf("Student Name is: %s", st.stu_name);
   printf("\nStudent Id is: %d", st.stu_id);
   printf("\nStudent Age is: %d", st.stu_age);
   return 0;
}
```

✓ Program Output:

```
Student Name is: Daniel
Student Id is: 1618
Student Age is: 30
```

✓ Variables inside the structures can include very different types (arrays, pointers, even other structures).

- **Arrays of structures**: Just as with any fundamental types, we can create pointers for user-defined types.

```c
#include<stdio.h>
struct Point {
   int x, y;
};

int main() {
   struct Point arr[10]; // Create an array of structures

   // Access array members
   arr[0].x = 10;   arr[0].y = 20;

   printf("%d %d", arr[0].x, arr[0].y); // Output: 10 20
   return 0;
}
```

- **Pointers to structures**: Just as with any fundamental types, we can create arrays for user-defined types.

```c
#include <stdio.h>
struct Point {
   int x, y;
};

int main() {
   struct Point p1 = {15, 25};
   struct Point *p2 = &p1; // p2 is a pointer to structure p1

   // Accessing structure members using structure pointer
   printf("%d %d", p2->x, p2->y); // Output: 15 25
   return 0;
}
```

- C structures do not permit functions inside the structure or static members.
- You can use a structure inside another structure. This is called nested structures, as in this example:

```c
#include <stdio.h>
struct stu_address {
   int street;
   char *state;
   char *city;
   char *country;
}

struct stu_data {
   int stu_id;
   int stu_age;
   char *stu_name;
```

```
   struct stu_address stuAddress;
}

int main() {
   struct stu_data mydata;

   // Accessing members of structure:
   mydata.stu_name = "Daniel";
   mydata.stu_id = 1001;
   mydata.stu_age = 30;
   mydata.stuAddress.state = "UP";       //Nested struct assignment
   mydata.stuAddress.city = "Rochester"; //Nested struct assignment
   mydata.stuAddress.country = "USA";    //Nested struct assignment

   printf("Student Name is: %s", mydata.stu_name);
   printf("\nStudent Id is: %d", mydata.stu_id);
   printf("\nStudent Age is: %d", mydata.stu_age);
   printf("\nStudent City is: %d", mydata.stuAddress.city);

   return 0;
}
```

COMPILATION (LINUX)

- GCC (GNU Compiler Collection): Compiler system that supports various programming languages (C, C++, Objective-C, Objective-C++, etc.)
- Examples:
  - ✓ `gcc source.c`: It will compile the `source.c` file and generate an output file with the default name `a`.
    - To execute the output file, use `./a`.
  - ✓ `gcc source.c -Wall -o source`. It will compile the `source.c` file and generate an output file named `source`. The `-Wall` option will check for all kinds of warnings like unused variables (this is good practice).
    - To execute the output file, use `./source`.

COMPILATION WITH FUNCTIONS

- C allows to declare and define functions separately; this is especially needed in case of library functions. The library functions are declared in header files (`.h`) and defined in library files (`.c`).
- For example:
  - ✓ `example.c`. Main file. This file can use functions that are declared in a different file.

```
#include "func.h"

int main (void) {
   printmsg("world"); // It will append 'world' to 'Hello'
   return 0;
}
```

  - ✓ `func.c`. Function definitions

```
#include <stdio.h>
#include "func.h"

void printmsg (const char * name) {
   printf("Hello, %s!\n" , name);
}
```

  - ✓ `func.h`. Function declaration

```
#include <stdio.h>
void printmsg (const char * name);
```

  - ✓ Compilation: `gcc -Wall example.c func.c -o example`.
  - ✓ To execute the application: `./example` ↵

The make utility

- It can be cumbersome to compile lots of files in a big project. There is a solution: the **make** utility.
- If you run the command "make", then the program will look for a file named `Makefile` that contains information about program dependencies and what needs to be compiled.

`Makefile:`

```
# Compiler/linker setup
# Linux-specific flags.
PLATFORM = linux
```

```
CC       = gcc
CFLAGS   = -O3 -Wall
OSLIBS   =
LDFLAGS  =

# Example programs -------------------------------------------------------
OBJS = example
all: $(OBJS)

example: example.c func.o
   $(CC) $(CFLAGS) -o example example.c func.o

# library
func.o: func.c func.h
   $(CC) $(CFLAGS) -c func.c

# Maintenance and stuff -------------------------------------------------
clean:
     rm -f $(OBJS) *.o core
```

- Available directives:
  - ✓ `make example`: The program `example` is compiled and its executable file is available.
  - ✓ `make all`: all OBJS are compiled (there can be more than one), and the executable file is available.
  - ✓ `make clean`: it cleans indicated files in the Makefile (.o files and executables).

C++ PROGRAMMING

- General purpose programming language. It has object-oriented and generic programming features.
- C++ inherits most of C's syntax. C++: superset of C with additional implementation of object-oriented concepts.
- Reference: B. Stroustrup, *The C++ Programming Language*, 4th ed.

OBJECT ORIENTED DESIGN

- Objected Oriented Programming (OOP): It aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.
- This approach works very well in real-world applications because the state and behavior of these classes and objects map very well to real-world objects.

OBJECTS AND CLASSES

- **Class**: Extended concept of the structure in C. It is an abstract user-defined type which contains a set of members. The most common kinds of members are:
  - ✓ Data members: data variables
  - ✓ Member functions: functions to manipulate the data variables. They can also define the meaning of initialization (creation), copy, move, and cleanup (destruction).
- **Object**: Instance of a class. When created, memory is allocated. The data members and member functions define the properties and behavior of the objects in a Class. A class is like a blueprint for an object (no memory is allocated when a class is defined).
- Formal definition: "*Objects are a language construct that associate data with the code to act on and manage that data. Multiple functions may be associated with an object and these functions are called the methods or member functions of that object. Objects are considered to be members of a class of objects, and classes can be arranged in a hierarchy in which subclasses inherit and extend the features of superclasses. All instances of a class have the same methods but have different state. The state of an object may or may not be directly accessible; in many cases, access to an object's state may only be permitted through its methods*".

- Pillars of object-oriented programming:
  - ✓ **Encapsulation**: Binding together the data and the functions that manipulate them. In C++, the methods and variables are encapsulated in a Class.
  - ✓ **Abstraction (Data Hiding)**: This refers to providing only essential information about the data to the outside world, hiding the background details of implementation.
  - ✓ **Polymorphism**: It is the ability of a message to be displayed in more than one form. An operation may exhibit different behaviors in different instances depending on the data types used in the operation. As such, C++ supports operator overloading and function overloading.
  - ✓ **Inheritance**: Ability of a class to derive properties and characteristics from another class.
    - Sub Class: The class that inherits properties from another class.
    - Super Class: The class whose properties are inherited by a Sub Class.
    - Reusability: This is useful when we want to create a new class and there is already a class that includes some of the code that we want. By doing this, we are reusing the fields and methods of the existing class.

- Class definition basic structure:

```
class ClassName {
Access Specifier:   // private, public, protected
   Data members;    // variables to be used
   Member Functions () {...} // methods to access data members
};
```

  - ✓ Access Specifier: They set the accessibility of the class members. It sets some restrictions on the class members not to get directly accessed by outside functions.
    - Public: Available to everyone (even other classes). They can be accessed from anywhere in the program.
    - Private: They can also be accessed by functions inside the class. An object or function outside the class cannot access them (exception: member functions of *friend functions*). If we do not indicate the access specifier a data member of member function, it will default to private.
    - Protected: Similar to private, but a class member declared as Protected is inaccessible outside the class except by any subclass derived of that class.
  - ✓ The public members provide the class' interface and the private members provide implementation details.

- Each object contains data and code to manipulate the data. When a program executes, the objects interact by sending messages to each other. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted, and type of response returned by the objects.

- Example (public members):

```
using namespace std;
class person {
public:    // Access Specifier
   // Data members
   string name;
   int id;

   // Member function
   void get_details() {
       cout << "Name: " << name << "\t" << "id: " << id << "\n";
   }
};

int main() {
   // Declaring an object of class person
   person p1; // p1 is an object.

   // Accessing data members (allowed since they are 'public'):
   p1.name = "Daniel";   p1.id = 35032;

   // Accessing member function:
   p1.get_details(); // it will print "Name: Daniel      id: 35032"

   return 0;
}
```

- Example (private and public members):
  - ✓ We can access the private data members of a class indirectly using the public member functions of the class.

```
using namespace std;
class Circle {
private:
   float radius; // data member
public:
   // Member function
   void compute_area (float r) {
      radius = r; // radius is modified by the function
      float area = 3.14*radius*radius;
      cout << "Area is :" << area << endl;
   }
};

int main() {
   Circle myobj; // // Declaring an object 'myobj' of class Circle

   // Accessing private data member outside the class.
   myobj.compute_area(1.5); // it will print "Area is: 7.065"
   return 0;
}
```

CONSTRUCTORS

- Special member functions of a class which initialize objects in a class. They are automatically called by the compiler each time an object is created.
- Constructors have same name as the class and no return type. Maybe defined inside or outside the class definition.
- Types:
  - ✓ Default constructor: It doesn't take any argument. `Class_name () {…}`
  - ✓ Parameterized constructor: We pass arguments. Typically, these argument help initialize an object when created. `Class_name (`*`parameters`*`) { … }`
  - ✓ Copy constructor: It creates a new object, an exact copy of an existing object, by initializing it with an object of the same class. `Class_name (const Class_name &old_object) { … }`. `const`: optional.

- Example (default and parameterized constructor):

```
using namespace std;
class Grizzlies {
public:
    int id; // data member

    //Default Constructor
    Grizzlies() {
        cout << "Default Constructor called" << endl;
        id = -3;
    }
```

```
    //Parameterized Constructor
    Grizzlies (int x) {
        cout << "Parameterized Constructor called" << endl;
        id = x;
    }
};

int main() {
    Grizzlies obj1; // object obj1 will call Default Constructor
    cout << "Grizzly id is: " <<obj1.id << endl;

    Grizzlies obj2(21); // object obj2 will call Parameterized Constructor
    cout << "Grizzly id is: " <<obj2.id << endl;
    return 0;
}
```

- ✓ Program Output:

```
Default Constructor called
Grizzly id is: -3
Parametrized Constructor called
Grizzly id is: 21
```

- ✓ Parameterized constructor: If we do not include print messages and only initialize the object, there is a compact form. In the following class example, the parameterized constructor definition is equivalent.

```
class Example {
public:
   int x, y, s;

   // Parameterized constructor
   Example (int xa, int ya) {
      x = xa; y = ya; }

    int myoperate() {
      s = x*x + y*y;
      return s;
    }
};
```

```
class Example {
public:
   int x, y, s;

   // Parameterized constructor
   Example (int xa, int ya): x(xa), y(ya) {}
   // this means x = xa, y = ya

   int myoperate() {
      s = x*x + y*y;
      return s;
   }
};
```

- ✓ Note that we can have more than one constructor in a class. This is called *Constructor Overloading*.

- Example (copy constructor)

```
using namespace std;
class Point {
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; } // Parameterized constructor

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()            {  return x; }
    int getY()            {  return y; }
};

int main() {
    Point pa(10, 15); // Parameterized constructor is called here
    Point pb = pa;    // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "pa.x = " << pa.getX() << ", pa.y = " << pa.getY();   // it will print pa.x=10, pa.y=15
    cout << "\npb.x = " << pb.getX() << ", pb.y = " << pb.getY(); // it will print pb.x=10, pb.y=15

    return 0;
}
```

- ✓ If we execute `Point pb = pa` without defining the Copy Constructor, the C++ compiler creates a default copy constructor which does a member-wise copy between objects; this is called a *shallow copy* (pointers and references of copied object point to the same memory locations). This works fine in general. However, it is better to have a user-defined copy constructor if an object has pointers or if there is any run-time allocation of resources. A copy constructor implements a *deep copy* (pointers and references of copied object point to new memory locations).
- ✓ In a user-defined copy constructor, we pass an object by reference and we generally pass it as `const` reference. We should do this whenever possible so that objects are not accidentally modified.

DESTRUCTORS

- A destructor is a member function that destructs or deletes an object. It is automatically called when the scope of the object ends: the function ends, the program ends, a block containing local variable ends, a delete operator is called.
- Destructors have same name as the class preceded by a tilde (~)
- Destructors do not take any argument and don't return anything. Only one destructor can exist in a class.

- Example:

```
class A {
    A() {    // Constructor
       cout << "Constructor called"; }

    // Destructor
    ~A() {
       cout << "Destructor called"; }
};

int main() {
   A obj1;   // Constructor Called
   int x = 1;
   if(x) {
       A obj2;  // Constructor Called
   }   // Destructor Called for obj2
} //  Destructor called for obj1
```

  - ✓ Program Output

```
Constructor called // This happens when obj1 is created
Constructor called // This happens when obj2 is created
Destructor called  // This happens when obj2 scope ends
Destructor called  // This happens when the program ends (obj1 scope ends as well)
```

CONST OBJECTS AND CONST MEMBER FUNCTIONS

- Objects of a class can be declared as `const`. Syntax: `const Class_Name Object_name;`
  - ✓ An object declared as `const` cannot be modified, and hence it can only invoke `const` member functions. The `const` property of an object goes into effect after the constructor finishes executed and ends before the destructor executes. So, both constructor and destructor can modify the object, but other methods of the class cannot.
  - ✓ An object declared as `const` needs to be initialized at time of declaration. This is possible via constructors.
- Member functions and member function arguments can be declared as `const`.
  - ✓ A function declared as `const` can be called on any type of object: const and non-const.
  - ✓ Non-const functions can only be called by non-const objects.
  - ✓ A function declared as `const` cannot modify the object on which it is called.

- Examples (`const` member function and `const` object)

```
class Test {
   int value;
public:
   Test(int v = 0) {value = v;}

   int getValue() const {return value;}
};

int main() {
   Test t(20); // t.value = 20
   cout<<t.getValue(); // Output: '20'
   return 0;
}
```

  - ✓ The `const` function `getValue()` cannot modify `value`.
  - ✓ Parameterized constructor: Its default argument (`v=0`) is used if the argument is not specified when an object is instantiated.

```
class Demo {
   int value;
public:
   Demo(int v = 0) {value = v;}
   void showMessage() {
      cout<<"showMessage() Function"<<endl; }
   void display() const {
      cout<<"display() Function"<<endl; }
};
int main() {
   const Demo d1; // d1.value=0 via constructor
   d1.display(); // Output: 'display() Function'
   return(0); }
```

  - ✓ The `const` object `d1` can't be modified.
  - ✓ `const Demo d1`: The constructor can modify it (`d1.value=0`). This is also valid: `const Demo d1(20)`.
  - ✓ `d1.showMessage()` would return an error: this non-const function can only be called by non-const objects.
  - ✓ The `const` function `void display()` can be called by const and non-const objects.

POINTERS WITH OBJECTS

- As with pointers to normal variables and functions, we can have pointers to objects, class member variables and class member functions.
- To have pointer to data members and member functions, you need to make them public.

Pointers to objects and accessing Data members

- We can define a pointer of class type, which can be used to point to class objects. To access data members, we use the dot `.` operator with object and we use the `->` operator with pointer to object:

```
class Simple {
public:
   int a;
};

int main() {
   Simple obj; // Declaring object 'obj'
   obj.a = 20;
   Simple* ptr;  // Declaring pointer of class type
   ptr = &obj; // Assigning pointer 'ptr' to point to object 'obj'

   cout << obj.a;  // Accessing data member 'a' via the object
   cout << ptr->a; // Accessing member 'a' with pointer
}
```

Pointers to Data Members

- A pointer can point to individual class members:
  - ✓ Declaration: `datatype class_name:: *pointer_name;`
  - ✓ Assignment: `pointer_name = &class_name:: datamember_name;`
  - ✓ Declaration + Assignment: `datatype class_name:: *pointer_name = &class_name:: datamember_name`
- When we have a pointer to a data member, we have to dereference that pointer to get what it is pointing to:
  - ✓ `object.*pointer_name` : When accessing the pointer to data member via the object.
  - ✓ `objectPointer->*pointer_name` : When accessing the pointer to data member via the object pointer.
- Example (note: the syntax is convoluted and is only used under special circumstances):

```
class Data {
public:
   int a;
   void print() { cout << "a is "<< a; }
};

int main() {
   Data d, *dp; // d: object, dp: pointer to object
   dp = &d;     // pointer to object defined
    // Declaration and assignment. Pointer 'ptr' declared and assigned to point to data member 'a'
   int Data::*ptr = &Data::a;   // 'ptr': pointer to data member 'a' in an object of class 'Data'

   d.*ptr=10; // 'ptr' points to 'a' in object 'd'. Thus *ptr = a = 10
   d.print(); // printing data member 'a'

   dp->*ptr=20; // we access data member 'a' via the pointer to object 'd', i.e., 'dp'
   dp->print(); // printing data member 'a'
}
```

  - ✓ Program Output:

```
a is 10
a is 20
```

Pointer to Member Functions

- Syntax: `return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name;`
- Example:

```
class X {
public:
   int a;
   void f (int b) { cout << "The value of b is "<< b << endl; }
};

int main() {
  int X::*ptiptr = &X::a;   // Pointer declaration and assignment to point to data member 'a'

  // Pointer Declaration and assignment to member function 'f'
  void (X::* ptfptr) (int) = &X::f;

  X xobject;  // create an object of class type X
  xobject.*ptiptr = 10;   // initialize data member using the pointer 'ptiptr'
  cout << "The value of a is " << xobject.*ptiptr << endl;

  // Call member function using the pointer to function 'ptfptr' and with parameter '20'
  (xobject.*ptfptr) (20);
}
```

  - ✓ Program Output:

```
The value of a is 10
The value of b is 20
```

FUNCTION OVERLOADING

- We can use the same name for different functions, to perform, either same or different functions in the same class. Two different ways to overload a function:
  - ✓ By changing the number of arguments.
  - ✓ By having different types of arguments.
- This enhances the readability of the program. If you have to perform a single operation but with different number or types or arguments, then you can simply overload the function.
- A class with multiple functions with same name but different parameters is said to be overloaded.

- Example (overloading: different number of arguments):

```
// First definition: 2 parameters
int sum (int x, int y) { cout << x+y; }

// Second overloaded definition: 3 parameters
int sum(int x, int y, int z) { cout << x+y+z; }

int main() {
   sum (10, 20);           // sum() with 2 parameter will be called. Output: 30
   sum (10, 20, 30);       // sum() with 3 parameter will be called. Output: 60
}
```

  - ✓ The overloaded `sum()` function has two definitions: one with accepts two arguments, and another which accepts three arguments. Which `sum()` function will be called? It depends on the number of arguments.

- Example (overloading: different types of arguments):

```
// First definition: 2 parameters of type int
int sum(int x, int y) { cout<< x+y; }

// Second overloaded definition: 2 parameters of type float
float sum(float x, float y) { cout << x+y; }

int main() {
    sum (10,20);     // sum() with 2 parameters of type int will be called. Output: 30
    sum(10.5,20.5);  // sum() with 2 parameters of type float will be called. Output: 31.0
}
```

  - ✓ The overloaded `sum()` function has two definitions: one with accepts two arguments of type `int`, and another which accepts two arguments of type `float`. Which `sum()` function will be called? It depends on the types of arguments.

- Example (function overloading in a class):

```
class printData {
public:
   void print(int i)    { cout << "Printing int: " << i << endl; }
   void print(double f) { cout << "Printing float: " << f << endl; }
   void print(char* c)  { cout << "Printing character: " << c << endl; }
};

int main(void) {
   printData pd;

   pd.print(7);           // Call print to print integer
   pd.print(500.618);     // Call print to print float
   pd.print("Hello C++"); // Call print to print string
   return 0;
}
```

  - ✓ Program Output:

```
Printing int: 7
Printing float: 500.618
Printing character: Hello C++
```

CONST KEYWORD

- C++ allows member methods to be overloaded on the basis of `const` type.
- Example: using a `const` member function and non-const member function

```
class Test {
protected:
    int x;
public:
    Test (int i):x(i) { } // Parameterized Constructor. x = i
    void fun() const { cout << "fun() const called " << endl; }
    void fun() { cout << "fun() called " << endl; }
};

int main() {
    Test t1 (10);  // t1 is created with t1.x = 10
```

```
    const Test t2 (20); // t2 is created with t2.x = 20
    t1.fun();
    t2.fun();
    return 0;
}
```

✓ Program Output:
```
Fun() called
Fun() const called
```

✓ The `const` object `t2` calls `void fun() const`, while the non-const object `t1` calls `void fun()` (non-constant function)

Constant parameters

- C++ allows functions to be overloaded on the basis on const-ness of parameters only if the `const` parameter is a pointer or reference (alias: another name for an existing variable. It can be thought of as a constant pointer).
- Example:

```
void fun(char *a) {
  cout << "non-const fun() " << a;
}

void fun(const char *a) {
  cout << "const fun() " << a;
}

int main() {
  const char *ptr = "ECE4900 - Fall 2020";
  fun(ptr);
  return 0;
}
```

✓ Program Output: `const fun() ECE4900 - Fall 2020.`

```
void fun(const int i) {
    cout << "fun(const int) called ";
}

void fun(int i) {
    cout << "fun(int ) called " ;
}

int main() {
    const int i = 10;
    fun(i);
    return 0;
}
```

✓ This will generate a compiler error as the `const` parameter `i` is not a pointer or reference.

OPERATOR OVERLOADING (`OPERATOR` KEYWORD)

- We can specify more than one definition for an operator in the same scope, this is called *operator overloading*.
- Most of the basic built-in operators in C++ can be redefined or overloaded.
- For user-defined types, we can redefine the way the operator works. To overload an operator, you need to define a special operator function inside a class, using the keyword **`operator`** followed by the symbol of the operator being defined. For example, `operator+()` overloads the `+` operator.
  - ✓ The overloaded operator must have at least one operand of the user-defined types. You cannot have an operator working on fundamental types (e.g.: `int, float, double`). For example, you can't overload the `+` operator for two `int`s to perform a different operation.

```
class MyClassName {
   ...
public:
   returnType operator symbol (arguments) {
      ...
   }
   ...
};
```

`returnType`: return type of the function

`symbol`: operator symbol you want to overload. Example: `+, <, -, ++, &, ~, /, ()`, etc.

`arguments`: You can pass arguments to the operator function in a similar way as functions

- Example:

```
class Test {
private:
   int count;
public:
   // Test(): count(5){} // This is compact way for the default constructor
   Test () { count = 5; }

   void operator ++ () { // in this example, this operator function definition has no parameters
      count = count+1;
   }
   void Display() { cout << "Count: "<< count; }
};

int main() {
    Test t; // created object 't' will call the default constructor, making t.count = 5

    ++t; // this calls "function void operator ++()" function
    t.Display(); // invoking member function
    return 0;
}
```

✓ Program Output: `Count: 6`

✓ The function `void operator++()` is called when the operator `++` operates on the object of `Test` class (`t`).

- Example:

```
class CVector {
public:
   int x,y;
   CVector () {}; // Default empty constructor

   CVector (int a, int b) { // Parameterized constructor
      x = a;
      y = b;
   }

   CVector operator + (CVector param) { // here, this operator function definition has 1 parameter
      CVector temp;
      temp.x = x + param.x;
      temp.y = y + param.y;
      return (temp);
   }
};

int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c; // object 'c' created with no initialization (default empty constructor)
  c = a + b; // Alternative (explicit function operator call): c a.operator+ (b);
  cout << c.x << "," << c.y;
  return 0;
}
```

  - ✓ Program Output: `4,3`
  - ✓ Note that there is more than one constructor in the class (constructor overloading)
  - ✓ The function `Cvector operator +(CVector param)` is called when the operator `+` operates two objects of `Cvector` class: `c=a+b`. Here, the operator function definition needs one parameter (`param`), which is the one to the right of `+`.

OVERLOADING THE PARENTHESIS '`()`' OPERATOR (FUNCTION CALL OPERATOR)

- Normal function (or member function in a class) call: `returnType myfunction (parameters)`. Hence, `()` is referred to as the function call operator.
- We can overload this function call operator '`()`' by defining a function for the operator (via the keyword `operator ()`).
  - ✓ The function called **`operator()`** implements the 'function call', 'call', or 'application' operator **`()`**.
  - ✓ Syntax for the function definition: `returnType operator() (parameters) { ... }`
  - ✓ The function call operator '`()`' can accept an arbitrary number of parameters of various types and may return any type. It is the only overloaded operator that can do this (all other overloaded operators have a fixed number of arguments). Also, unlike other operators, we can use fundamental types.
- The function call operator, when overloaded, does not modify how functions are called. Rather, it modifies how the operator is to be interpreted when applied to objects of a given type.
- In a class, there might be several (or just one) definitions for the function call operator. We say this operator is overloaded because usually we can use the object with the parenthesis to call different members (functions, constructors) based on the arguments. This applies even if there is only one member function `operator ()`.
- Example:

```
class Distance {
private:
    int feet;
    int inches;
public:
    Distance () { feet = 0; inches = 0; } // default constructor (no parameters)
    Distance (int f, int i) { feet = f; inches = i; } // parameterized constructor

    // overload function call: the result is of type Distance.
    Distance operator () (int a, int b, int c) {
      Distance D;
      D.feet = a + c + 10;
      D.inches = b + c + 100;
      return D;
    }

    // method to display distance
    void displayDistance () {
      cout << "F: " << feet << " I: " << inches << endl;
    }
};

int main() {
   Distance D1(11,10), D2; // initializing: D1.feet = 11, D1.inches = 10, D2.feet = 0, D2.inches = 0
   D1.displayDistance(); // It will output: F: 11 I: 10
```

```
    D2 = D1(10,10,10); // With 3 parameters, due to overloading, this is reinterpreted as invoking
                       // operator() whose result is of type Distance, hence D2 is the output.
    D2.displayDistance(); // It will output: F: 30 I: 120
    return 0;
}
```

- ✓ The `()` operator is overloaded, as the parameterized constructor also uses the parenthesis. The function defined in `operator()` is used only when we invoke the object with the 3 parameters (here, the object is treated like a function).
- ✓ `D2 = D1(10,10,10)`: It will output an object `D2`, where `D2.feet = 10 + 10 + 10, D2.inches = 10 + 10 + 100`.

- Usually, we overload the function call operator `()` to create classes that have a primary operation. But more commonly, the overloaded operator `()` is used to create objects that behave like functions, called ***functors***.

FUNCTORS (FUNCTION OBJECTS)

- Functors are objects that can be treated as though they are a function or function pointer. You can write code like this:

```
My_FunctorClass my_functor(4); // object definition and initialization
my_functor(3,2,1);             // treats the object as a function
```

- A functor is a class which defines the operator `()`. It lets you create objects that behave like functions. Since it overloads the function call operator, code can call its major method using the same syntax it would for a function.
- A functor is like a function, but it has the advantage that is stateful, i.e., it can keep data reflecting its state between calls.
- Unlike functions, we first need to create an object of functor (this might include initialization)
  - ✓ You can use a functor's constructor to embed information that is later used inside the implementation of `operator()`.
- Once the object is created, we can call the object as if it were a function. It is here where object's `operator()` is invoked.

- **Example:** Functor class (`MyFunctorClass`) with a constructor that takes an integer argument and saves it. When objects of the class are "called", it will return the result of adding the saved value (`xt`) and the argument to the functor (`y`).

```
class MyFunctorClass {
public:
   MyFunctorClass (int x): xt(x) {} // Parameterized constructor. xt = x

   int operator() (int y) {
      return xt + y; }

private: // can only be accessed by functions inside the class
   int xt; // Data member (private)
};

int main() {
  MyFunctorClass myfunctor(5); // 'myfunctor': object. Note that myfunctor.xt=5 is assigned
  int b = myfunctor(6); // it will call operator() and return b = 6 + 5
  cout << "Result: " << b << "\n";
  return 0;
}
```

- ✓ `MyFunctorClass myfunctor(5)`: Object declaration. The constructor initializes the data member `xt` to 5.
  - The act of constructing an object allows you to give the functor information that it can use inside the implementation of its function-like behavior (when the functor is called through `operator()`).
- ✓ `myfunctor(6)`: we treat this object of type `MyFunctorClass` like a function and pass it one `int` argument. Thus, it will invoke `operator()`, that will operate and return the result of type `int`. This is why functors are called function objects because we can call class `MyFunctorClass` as if it were a function.

- **Example:** Functor class (`Multiplier`) with a parameterized constructor that takes an integer argument and saves it. When objects of the class are "called", it will return the result of multiplying the saved value (`multiplier`) and the argument to the functor (`x`).

```
class Multiplier {
public:
   Multiplier(int m): multiplier(m) {} // multiplier = m
   int operator()(int x) { return multiplier * x; }
private:
   int multiplier;
};

Multiplier m(5); // declaration of object m of class Multiplier and initialization: m.multiplier = 5
cout << m(4) << endl; // Output: '20'
```

- ✓ `Multiplier m(5)`: Object declaration. The constructor initializes the data member `multiplier` to 5.
  - The act of constructing an object allows you to give the functor information that it can use inside the implementation of its function-like behavior (when the functor is called through `operator()`).
- ✓ `m(4)`: when passing the parameter 4 (not in the declaration), the call is reinterpreted as invoking `operator()`, that will operate and return the result of type `int`.