C++ Review
Quick Notes

RajPat

2020
October

thanks to
Cherno, cppreference.com

Contents

Listings

1 Introduction

1.1 Header Files

```
1           #include <iostream>
2           // Include file is in one of the folders
3           // They have to be in one of the include directories.
4           // Relative path not needed.
5
6           #include "iostream"
7           // " " will also work in this case but is discouraged.
8           // " " are often used for user defined include files
    with relative paths.
9
10          #include "../Log.h"
11          // "../Log.h' -> relative to current file.
12
13          // C standard
14          #include <stdio.h>
15
16          // C++ standard
17          // C++ standard does not contain .h extension to
    differentiate with C standard library.
18          #include <iostream>
19
20          // Using C header files in C++ file
21          extern "C" {
22          #include <C_header_file.h>
23          }
24
```

Listing 1: Header file example

1.2 Compiler

1. C++ doesn't care about files → files have no meaning.
2. File → translation unit → object file.
3. If you include many cpp files in one cpp file, the compiler is going to generate one large cpp file and 1 translation unit
4. Remember, include is just pasting code in file.
5. Different types of C/C++ compilers → **CLANG, G++/GCC, MSVC, LLVM**, etc.
6. Generates machine code.
7. Can enable assembly output from compiler. This option generates readable data. Obj file is unreadable.
8. .asm will have machine code, mnemonics, etc. readable.

9. Developer can use compiler optimizations. Optimizations remove unused code (dead code).
10. Compiler has few modes/configurations, such as **Debug Release**.
11. Compiler does constant folding, such as replacing 5*2 to 10 at compile time.

1.2.1 Pre-processor

```
#include
#define
#ifdef
    ...
#endif
#if

```

Pre-processor output can be output to a file → *.i. This option in Visual studio does not generate obj file → useful for debug only.

1.3 Linker

1. Find where each symbol and function is and link them together.
2. C runtime library links the main function and knows where it is, first.
   (a) Default entry point is the main() function.
   (b) Can be customized to change entry point to something else other than main().
3. Compiler errors begin with "C" and linker errors with "L".
4. "Unresolved symbol" is a typical linking error.
   (a) Definition missing but declaration exists.
   (b) If the function is never called and is never defined, but declared, it won't generate error.
   (c) Static int func() → Only defined in this file. So if this func has linker error and is never used, linker can generate the error.
5. Multiple definitions of same function definitions is a linker error too.
   (a) Make static func so each cpp file has its own definition of func with same name → : no bueno
   (b) Make function inline
   (c) Solution → Header-file: declare func & Cpp-file: define func
6. Compiled obj files can be in library or platform APIs, and linker can point to these.

1.4 Libraries

1. Either pre-built binaries or build using source code.
2. Source code → static or dynamic library.
3. Usually 2 parts in a library → *include* and *libraries (lib)*.
4. *include* directories have header files, that we can use to find functions that are in binaries.
5. *lib* has pre-compiled binaries that can be used for linking.

1.4.1 Static Linking

1. Library is put into executable. Linking happens at compile-time.
2. Usually 2 parts in a library → *include* and *libraries (lib)*.

1.4.2 Dynamic Linking

1. Linking happens at run-time. i/e/ when executable is launched.
2. Load libraries dynamically. DLLs, etc. Libraries are in separate file at run-time and exe can look into this file and pull function pointers whenever needed.
3. Another case of dynamic linking is when application does not have any idea of any include/libraries. Developer can use some documentation to look into exact definitions and use them in application. (How? - Google it)
4. In addition to including include directory and dll.lib, exe must have access to .dll file (same folder as exe is good enough, can also provide search paths to executable).

1.5 Variables

1. Stored either in stack or heap
2. Primitive data types:
   (a) Char: 1 byte
   (b) Short: 2 bytes
   (c) int: 4 bytes
   (d) long: 4 bytes
   (e) long long: 8 bytes
   (f) float: 4 bytes (eg.float a=5.2f;)
   (g) double: 8 bytes (eg.double a=5.2;)

(h) bool: 1 bit (but occupies 1 byte)

3. To get size of variable, use **sizeof(boot)**; parenthesis option in Visual Studio.

4. Visual Studio in debug mode sets stack memory to "0xcc", which helps in debug. This is not done by the compiler in release mode as it would slow it up (obviously).

1.6 Conditions and Branches

1. Nothing to say, really → if, else.

```
                if (condition1) {
                  // condition1=true, condition2 and others
    are don't care
                } else if (condition2) {
                  // condition1=false & condition2=true
                } else {
                  // condition1=condition2=false
                }

```

Listing 2: If...else if...else example

2. **Ternary operator**

   result = (condition) ? (code-condition=true) : (code-condition=false)

1.7 Loops

```
            for (int i=0; i<5; i++)
            {
                printf("%d", i);
            }
            // same as
            int i=0;
            bool condition=true;
            for (;condition;)
            {
                printf("%d", i);
                i++;
                if !(i<5) condition=false;
            }

            // while
            while (condition)
            {
                printf("%d", i);
                i++;
            }

            //do while
            do
            {       }
```

```
25              while (condition)
26
```

Listing 3: Loops example

Control flows

1. Continue → only used inside loop → go to next iteration if there is one
2. Break → loops and switch statement → end loop
3. Return → get out of function

1.8 Pragma

```
1       #pragma once
2
3       // is same as single include definition as below
4       #ifndef __HEADER__
5       #define __HEADER__
6       ...
7       #endif
8
```

1.9 Operators

https://en.cppreference.com/w/cpp/language/operators

1.10 Functions

1. void func(void, void ...)
2. Returns only one value.
3. If values are passed to function as **values**, it creates a local copy in stack, which vanishes once the function is out of scope.
4. Can avoid copying by passing by reference.
5. Functions return a *single* value.
6. Multiple values can be returned by function using the following methods:
7. **Method-1:** Define a **struct** and return it
8. **Method-2:** Pass the return variable by reference and write into it in the function.
9. **Method-3:** If return types are same, define a std::array in the function and return the pointer to array. Example: *static std::array¡std::string, 2¿ function(int a, int b)*. Arrays are created on **stack**.

10. **Method-4:** If return types are same, define a std::vector and return it. Vectors are created on **heap**.

11. **Method-5:** std::tuple; example: std::tuple¡std::string, int, float¿ function(int a, int b);

2 Static

2.1 Static variables outside a class or struct

1. Linkage of symbols is internal to the translation unit.

2. No other translation unit is going to see this variable.

2.2 Static variables inside a function/scope

When a variable is declared as static, it is allocated for the **lifetime** of the program. Even though the function is called several times, space for it is allocated only once and the value of the variable in the previous call is preserved even in next function call.

2.3 Static variables inside a class or struct

1. Linkage of symbols is shared across all instances of class.

2. **Static variables in a class**: As the variables declared as static are initialized only once as they are allocated space in separate static storage, so the static variables in a class are shared by the objects. There can not be multiple copies of the same static variables for different objects. Also because of this reason **static variables can not be initialized using constructors**.

```
#include<iostream>
class GfG
{
public:
    static int i;
    GfG()
    {  // Do nothing
    };
};

// This is how you initiate static var in a class
int GfG::i = 1;

int main()
{
    GfG obj;
    // prints value of i
    std::cout << obj.i << std::endl;
```

```
}

```

Listing 4: Static example

2.4 Static methods inside a class or struct

1. Linkage of function is shared across all instances of class.
2. Static methods cannot access non static variables!
3. Static method does not have an associated instance. (This is like a method outside a class).

3 Pointers & References

3.1 Pointers

1. It is an integer that holds an address to a memory location.
2. Pointers: **bool***
3. References: **bool&**
4. References are extensions of pointers.
5. References do not occupy memory, they reference to existing variables.

```
#include <iostream>
#include "stdout_log.h"

#define LOG(x) std::cout << x << std::endl;

// call by value
int func1(int val)
{
  val++;
  return val;
}

// call by pointer
void func2(int* valptr)
{
  //valptr++; // increments pointer to next int address
  (*valptr)++;  // de-reference and then increment
}

// call by reference
void func3(int& val)
{
  val++;
}

```

```
26
27        int main()
28        {
29          LOG("\nhello world");
30            int a;
31          a = 5;
32          for (int i = 0; i < a; i++) {
33            stdout_log("hello world");
34          }
35
36          LOG("\npointers");
37          // pointers
38          {
39            int var = 10;
40            //void* ptr = nullptr;
41            //  0 is not a valid memory address. THis is null
42            //  this wont work: ptr = &var;
43
44            //void* ptr = &var;
45            //std::cout << *ptr << std::endl;
46            int* ptr = &var;
47            std::cout << *ptr << std::endl;
48          }
49          LOG("\nallocate to heap,  pointer to pointer");
50          // allocate to heap,  pointer to pointer
51          {
52            char* buffer = new char[8];
53            char** ptr = &buffer;
54            memset(buffer, 0, 8);
55            delete[] buffer;
56          }
57          LOG("\npointers and references");
58          // pointers and references
59          {
60            int a = 10;
61            int* ptr = &a;  // pointer
62            int& ref = a; // reference -> this variable doesn't
    exist
63            LOG(a);   // 10
64            LOG(ptr); // 009EF8BC
65            LOG(*ptr);  // 10
66            LOG(ref); // 10
67            ref++;
68            LOG(a);   // 11
69          }
70          LOG("\ncall by value and call by ref");
71          //call by value and call by ref
72          {
73            int a = 10;
74            LOG(a);          //10
75            LOG(func1(a));  //11
76            LOG(a);          //10
77            func2(&a);
78            LOG(a);          //11
79            func3(a);
80            LOG(a);          //12
81          }
```

```
82          }
83
```

Listing 5: Pointers example

3.2 Smart Pointers

Smart pointers automate **new** and **delete**. These are *wrappers* around raw pointers. These are good to keep track of memory used and make sure that "delete" is inherently called to prevent memory leak. These do not altogether replace *new* and *delete*. Use *std::unique_ptr* whenever needed, if not, use *std::shared_ptr*.

3.2.1 std::unique_ptr

This is a scoped pointer. Cannot copy a unique pointer as copy constructor of this class is deleted.

```
1           #include <iostream>
2           #define Log(x) std::cout << x << std::endl;
3
4           class Entity
5           {
6           public:
7             Entity()
8             {
9               Log("Created entity");
10            }
11
12            ~Entity()
13            {
14              Log("Destroyed entity");
15            }
16            void print_info(void)
17            {
18              Log("inside entity");
19            }
20          };
21
22          int main()
23          {
24            // std::unique_ptr -> stack allocated object
25            {
26              //method 1
27              //std::unique_ptr<Entity> entity(new Entity());
28
29              //method 2 - preferred way
30              //if constructor throws exception, would not result
     in dangling pointer
31              std::unique_ptr<Entity> entity = std::make_unique<
    Entity>();
32              entity->print_info();
33
34            }
35          }
```

```
Output:
Created entity
inside entity
Destroyed entity

```

Listing 6: std::unique_ptr example

3.2.2 std::shared_ptr

Compiler maintains a *ref count* to keep track of how many pointers point to object. Only when all of them are out of scope is when then the underlying object gets destroyed.

```
#include <iostream>
#define Log(x) std::cout << x << std::endl;

class Entity
{
public:
  Entity()
  {
    Log("Created entity");
  }

  ~Entity()
  {
    Log("Destroyed entity");
  }
  void print_info(void)
  {
    Log("inside entity");
  }
};

int main()
{
  {
    std::shared_ptr<Entity> e0;
    {
      std::unique_ptr<Entity> entity = std::make_unique<Entity>();
      // this is how you can create a shared pointer
      std::shared_ptr<Entity> sharedEntity = std::make_shared<Entity>();

      e0 = sharedEntity;
      // entity gets destroyed here
    }
    // sharedEntity and e0 are pointing to same object. so, even though sharedEntity
    // is no more, since e0 is still refernced to pointer, the destroyer isn't called.
  }
  // e0 is destroyed now after this scope
  // when all references are gone, that's when the underlying object is destroyed
```

```
            }
```

Listing 7: std::shared_ptr example

3.2.3 std::weak_ptr

std::weak_ptr¡Entity¿ weakEntity = sharedEntity;
This does not increase *ref count*.

3.3 Function pointers

```
            #include <iostream>
            #include <vector>
            #define Log(x) std::cout << x << std::endl

            void func(std::string msg)
            {
              Log(msg);
            }

            int val_scale(int value)
            {
              auto x =  value * 2;
              Log(x);
              return x;
            }

            int val_add(int value)
            {
              auto x = value + 2;
              Log(x);
              return x;
            }

            void for_each(const std::vector<int>& values, int(*
    funcPtr)(int))
            {
              for (int value : values)
                funcPtr(value);
            }

            int main()
            {
              // call by value
              func("message");

              // method 1
              // auto f = func("hello");  // doesn't work - can't
    be of type void
              auto f = &func; // & is optional
              f("method1"); // prints method1

              // method 2
              void(*var)(std::string txt) = func;
```

```
    var("method2"); // prints method2

    // method3
    typedef void(*varFunc)(std::string txt);
    varFunc ff = func;
    ff("method3");  // prints method3

    // example2
    // we have a vector of variables and we want a func
    // to iterate and perform some action
    std::vector<int> values = { 1,2,3,4,5,6,7 };
    for_each(values, val_scale);
    // prints 2, 4, 6, 8, 10, 12 14
}
```

Listing 8: function pointers example

3.4 Lambdas

Wherever we use function pointers to functions, we can use lambdas.

4 Arrays

4.1 Raw arrays, C++ ¡array¿

1. Arrays are collection of elements, i.e. contiguous segment of memory.
2. Arrays created in **heap** with *new* keyword result in memory indirection and maybe a performance hit depending on access frequency.
3. In C++11, there is an inbuilt *standard array* data structure, different than the raw array shown in the below example.

```
#include <iostream>
#include <array> // C++11 standard array
#define Log(x) std::cout << x << std::endl;

int main()
{
    // created on stack - will be destroyed after scope
    // size has to be a compile time constant
    int array1[5];

    // initialize
    for (int i = 0; i < 5; i++)
        array1[i] = i;

    // prints address of first element
    Log(array1);

    Log(sizeof(array1));  // prints 20 = 5*4
```

```
19
20              // prints 0, 1, 2, 3, 4
21              for (int i = 0; i < 5; i++)
22                Log(array1[i]);
23
24
25              // prints 0, 1, 2, 3, 4
26              char* ptr = (char *)array1;
27              for (int i = 0; i < 5; i++)
28              {
29                Log(*(int*)(ptr));
30                ptr = ptr + 4;
31              }
32
33              // dynamic memory allocation
34              // will be created on heap - will not be destroyed
    after scope
35              // must delete manually
36              int* another = new int[5];
37
38              // initialize
39              for (int i = 0; i < 5; i++)
40                another[i] = array1[i];
41
42              Log(sizeof(another)); // prints 4 as another is a
    pointer
43
44              // free the heap before you leave!
45              delete [] another;
46
47              // c++11 standard array
48              // this is static array - size doesn't change
49              std::array<int, 5> another1;
50              Log(sizeof(another1));  // prints 20 = 5*4
51            }
52
```

Listing 9: Raw arrays - Stack vs Heap example

4.2 Dynamic Arrays: std::vector

When it comes to re-sizing, the entire data structure has to be copied by compiler, so it would be a performance hit.

```
1             #include <iostream>
2             #include <vector>
3
4             #define Log(x) std::cout << x << std::endl;
5
6             int main()
7             {
8               // dynamic arrays
9               std::vector<int> ints;
10
11              struct Vertex {
12                int x, y, z;
```

```
            };
            std::vector<Vertex> vars;
            vars.push_back({ 1,2,3 });  // add item
            // this constructs vertex in main and then copies it
  to struct/class
            vars.push_back({ 4,5,6 });
            Log(vars[0].y); // can iterate - 2
            Log(vars.size()); // 2

            // iterate through dynamic array
            for (Vertex v : vars) // copies the data
              Log(v.z); // 3, 6

            for (Vertex& v : vars) // reference - no data copy
              Log(v.z); // 3, 6

            vars.erase(vars.begin()+1);
            Log(vars.size()); // 1

            for (Vertex& v : vars)
              Log(v.z); // 3

            vars.clear();
          }

```

Listing 10: std::vector example

1. Can use **vars.reserve(2);** to reserve memory for 2 objects and avoid resizing. This improves perf.

2. Use **vars.emplace_back()** if vars is class and thus calls constructor in place. This avoids copying from main to object

5 Strings

5.1 C Style - char array

1. Strings are array of characters.Characters are representations of symbols (letters, numbers, etc.).

2. Numerous types of character encoding: utf8 has 256 characters, utf16 has 2î6.

```
          #include <iostream>
          #define Log(x) std::cout << x << std::endl;

          int main()
          {
            // This is C type
            // const as strings are immutable
            // these are not allocated on heap
            const char* name = "raj"; // char * if " "
```

```
            Log(name);  // raj

            // technically possible to use just char
            char* newName = (char *)"apple";
            Log(newName); // apple

            // end of string is null termination character
            char eg[6] = {'R', 'a', 'j'};
            Log(eg);  // Raj

            char eg1[6] = { 'a', 'p', 'p', 'l', 'e', 't' };
            Log(eg1); // applet8903725098:random data until it
hits null term
            // without null character, it doesn't know where to
stop printing
            // null termination is '\0'.

          // strlen to find length of string
            Log(strlen(eg));  // 3
            Log(strlen(eg1)); // 19

            // strcopy_s to copy strings
            // strcopy is unsafe as it can reult in buffer
overflow
            strcpy_s(eg1, eg);
            Log(eg1);     // Raj
          }

```

Listing 11: C style char array

5.2 C++ style

1. C++ has a class called Basicstring, which is a template class.
2. std::String is a templated specialization of BasicString class with char as underlying parameter.
3. std::String is also a char array.
4. http://www.cplusplus.com/reference/string/string/

```
                #include <iostream>
                #include <string>

                #define Log(x) std::cout << x << std::endl;

                // This is C++ style
                int main()
                {
                  std::string str = "raj";
                  Log(str);     // raj
                  Log(str.size());  // 3
                  char eg[20] = { 't', 'e', 'a', '\0' };
                  std::string test ("new string");

```

```
15                  Log(test);  // new string
16                  Log(eg);  // tea
17                  test.copy(eg, 2, 4);
18                  Log(test);  // new string
19                  Log(eg);  // sta
20
21                  // appending strings
22                  // this does not work as we cannot append 2
    const char arrays
23                  // std::string str = "raj" + "tea";
24                  str += "tea";
25                  Log(str); //rajtea
26
27                  Log(str.find("ea"));  // 4
28                  Log(str.find("lot")); // 4294967295 (not
    correct)
29              }
30
```

Listing 12: C++ style strings

5.3 String literals

1. "asas" is a string literal.
2. String literals are stored in "read-only" locations of memory.

```
1               ...
2               ; COMDAT ??_C@_08JOKHDEJH@readonly@
3               CONST SEGMENT
4               ??_C@_08JOKHDEJH@readonly@ DB 'readonly', 00H
      ; `string'
5               CONST ENDS
6               ; COMDAT ??_C@_03KHJDFILH@lot@
7               CONST SEGMENT
8               ??_C@_03KHJDFILH@lot@ DB 'lot', 00H     ; `
    string'
9               ...
10
```

Listing 13: String literals in asm file

6 Const

1. **char*** is a mutable pointer to a mutable character/string.
2. **const** is a *promise* to not change the contents.
3. **const char*** is a mutable pointer to an immutable character/string (same as **char const***). You cannot change the contents of the location this pointer points to. Some compilers give error messages when we try to do so. For the same reason, conversion from **const char*** to **char*** is deprecated. Pointer can point to another immutable string/character, but not a good practise.

4. **char* const** is an immutable pointer (it cannot point to any other location) but the contents of location at which it points are mutable.

5. **const char* const** is an immutable pointer to an immutable character or string.

6. const after method name is possible in a Class. This defines that the method does not modify the contents of Class. Can be in "getters".

```
const int* const GetX() const
{
    return m_X;
}

```

Listing 14: Const in Class method

6.1 Mutable keyword

1. Mutable enables const method to modify a non-const variable.

2. Mutable can be used with **const** and **lambda**.

```
#include <iostream>
#include <string>

#define Log(x) std::cout << x << std::endl;

class MyClass
{
private:
  std::string m_Name;
  mutable int m_DebugCount = 0;
public:
  const std::string& getName() const
  {
    m_DebugCount++;
    return m_Name;
  }
};

int main()
{
  const MyClass obj;
  Log(obj.getName());
}

```

Listing 15: Mutable example with const method

7 Stack vs Heap

1. Creating in stack: **className obj;**.

2. Variables/objects instantiated in a scope are allocated in **stack** and disappear after scope is exited.

3. If the object/variable is needed outside the scope, we need it to be allocated on heap.

4. Stack is usually smaller in size.

5. To allocate on heap, use **new** keyword.

```
            // heap allocation
            Entity* obj = new Entity("myName");
            int* b = new int[50]; // 200 bytes

            // user MUST free the memory
            // delete calls destructor
            delete obj;
            // if allocation using new is with [], then use []
    in delete as well
            delete[] b;

```

Listing 16: Heap allocation using "new" example

6. Allocating on heap takes longer, has performance downside compared to stack allocation. User is also responsible to free the memory once the memory is not needed.

7. Allocating memory on stack is one CPU instruction (mostly) vs allocating on heap, it is quite a it of book-keeping, i.e. → malloc, OS checks free-list, etc.

```
            #include <iostream>
            #define Log(x) std::cout << x << std::endl;

            int main()
            {
              // allocate on stack
              int a = 5;
              int arr[10];

              // heap allocation
              // new calls malloc()
              int* b = new int;
              *b = 5;
              int* harr = new int[5];
              // must delete
              delete b;
              delete[] harr;

              // smart pointers will do new and delete without
              // programmer havign to use new/delete
            }


```

Listing 17: Stack vs Heap example

7.1 New keyword

1. Main purpose is to allocate memory on heap. **int* a = new int;**. This would request OS for 4 contiguous bytes of memory.
2. **new** also calls constructor for the object instantiation.This is unlike using **malloc**.
3. Using **delete e;** is similar to using **free(e);** function.
4. **new(ptr)** can be used to create a new object at a specific location. This is called **placement new**.

8 Enum vs Enum class

8.1 Enum

Enums are represented in integers. Usually, starting is 0, or can be user specified.

```
enum Example
{
  // enums are stores as integers
  A, B, C
};

enum Example2: char
{
  A2=5, B2, C2
};

int main()
{
  Example val = A;
  Log(val); // prints 0
  Log(B);   // prints 1

  Example2 val2 = A2;
  Log(val2);  // some symbol
  Log(B2);  // some symbol
}

```

Listing 18: Enum example

8.2 Enum Class

```
#include <iostream>

class Log
{
public:
  /*enum LogLevel
  {
```

```
    LogLevelError, LogLevelWarn, LogLevelInfo
  };
  */

  enum class LogLevel
  {
    LogLevelError, LogLevelWarn, LogLevelInfo
  };

private:
  LogLevel m_LogLevel = LogLevel::LogLevelInfo;

public:
  void SetLevel(LogLevel level)
  {
    m_LogLevel = level;
  }

  void Error(const char* message)
  {
    if (m_LogLevel >= LogLevel::LogLevelError)
      std::cout << "[ERROR]:"<<message<<std::endl;
  }

  void Warn(const char* message)
  {
    if (m_LogLevel >= LogLevel::LogLevelWarn)
      std::cout << "[WARN]:"<<message<<std::endl;
  }

  void Info(const char* message)
  {
    if (m_LogLevel >= LogLevel::LogLevelInfo)
      std::cout << "[INFO]:"<<message<<std::endl;
  }
};

int main()
{
  Log log;
  // using enums and not enum class
  //log.SetLevel(Log::LogLevelError);
  //log.SetLevel(Log::LogLevelWarn);
  //log.SetLevel(Log::LogLevelInfo);

  log.SetLevel(Log::LogLevel::LogLevelError);
  log.SetLevel(Log::LogLevel::LogLevelWarn);
  //log.SetLevel(Log::LogLevel::LogLevelInfo);

  log.Info(" Just info... ");
  log.Warn(" I am warning you... ");
  log.Error(" This is error... ");
}

```

Listing 19: Enum class example

9 ”auto” keyword

1. C++ can be used as a weakly typed language. Using **auto**, compiler can understand the underlying type.
2. One used is when a function return, host code in main() need not change, unless the returned value is used for specific type processing.
3. if type is long name, such as std::vector¡std::string¿, etc., using auto makes code readable.

```
#include <iostream>
#define Log(x) std::cout << x << std::endl

int main()
{
  auto a = 5;    // int
  auto b = "test"; // const char*
  auto c = 5.5f;   // float
  Log(a); // 5
  Log(b); // test
  Log(c); // 5.5
}

```

Listing 20: auto example

10 Object Oriented Programming

OOP binds together data and methods to transform data (functions) in a way that is encapsulated and hidden from other unrelated methods.

10.1 OOP Concepts

Object oriented programming comprises of following concepts

1. **Classes**

   User defined datatype, similar to struct.

2. **Objects**

   An object is instance of a class.

3. **Encapsulation**

   Encapsulation is defined as binding of data and methods processing the data within a class.

4. **Abstraction**

   Ability to control what is visible outside of class. Also, any downstream method using the function need not know underlying implementation of the said method in order to use it.

5. **Inheritance**

   Derive properties from another class. Facilitate abstraction and re usability.

6. **Polymorphism**

   Operator overloading and **function overloading** are 2 types of polymorphism in C++. Example: Same person can be a student, athlete at different situations.

10.2 Class vs Struct

1. Except visibility, nothing else.
2. Backward compatibility to C.
3. Class → default is private
4. struct → default is public.
5. One common practise if this is only for representation of data, use struct. If the encapsulation has some sense of functionality (i.e. some function to modify the data), then use class to encapsulate the functions along with data.

```
#include <iostream>

class Particle
{
public:
  float x, y, z, m;
  void get_info(void)
  {
    std::cout << "x : " << x << std::endl;
    std::cout << "y : " << y << std::endl;
    std::cout << "z : " << z << std::endl;
    std::cout << "m : " << m << std::endl;
  }
};

struct ParticleStruct
{
  float x, y, z, m;
  void get_info(void)
  {
    std::cout << "x : " << x << std::endl;
    std::cout << "y : " << y << std::endl;
    std::cout << "z : " << z << std::endl;
    std::cout << "m : " << m << std::endl;
  }
};

int main(void)
{
```

```
            Particle p;
            p.x = 1.35f;

            ParticleStruct ps;
            ps.x = 1.75f;

            p.get_info();
            ps.get_info();

        }

        Output:
        x : 1.35
        y : -1.07374e+08
        z : -1.07374e+08
        m : -1.07374e+08
        x : 1.75
        y : -1.07374e+08
        z : -1.07374e+08
        m : -1.07374e+08


```

Listing 21: Class vs Struct example

10.3 Constructors

1. Special method that runs every time an object is instantiated. (Similar to __init__ in python)
2. Used to initialization of variables, etc. inside the class.
3. Same name as class. Entity()
4. C++ provides a default constructor for the class which is empty. Hence, when we have a class with only static variables and static methods, it is important to delete the constructor.

```
        class Log
        {
            Public:
            Log() = delete;    // <- ThIs is to delete default constructor
            Static void Write()
            {
            }
        };

```

Listing 22: Delete constructor example

5. **explicit** keyword can be added to constructor to prevent implicit conversions by compiler. C++ compiler can make 1 implicit conversion.

```
            explicit Entity(int age)
                : m_Name("unknown'), m_Age(age) {}

            // This constructor will prevent the following implicit conversion by compiler
            // If explicit keyword is not used in constructor, this will work
            Entity b = 22;
            //.. but this will work
            Entity b(22);

```

Listing 23: Explicit keyword

10.3.1 Copy constructor

Default provided by compiler.

Creates a new object, exact same copy of existing object.

Copy constructor is called when new object is created from existing object while assignment operator is called when already initialized object is assigned a new value from different object.

```
            #include <iostream>
            #define Log(x) std::cout << x << std::endl;

            class Entity
            {
            private:
              int id;
            public:
              int x, y;
              Entity()
              {
                x = 0; y = 0; id = 0;
              }
              //function overloading for constructor - parameterized
              Entity(int vx, int vy)
              {
                x = vx;
                y = vy; id = 0;
              }
              // user-define copy constructor
              // if this is commented out, copy will still work with same syntax as shown in last line in main (e3 definition), but compiler does a shallow copy and not a deep copy.
              Entity(Entity& obj)
              {
                x = obj.x;  y = obj.y;  id = obj.id;
              }
              // polymorphism
              virtual void get_info(void)
              {
                Log(x);
                Log(y);
              }
```

```
32            };
33
34            class Player : public Entity
35            {
36              // anything that is not private in entity class is
    accessible by player
37            public:
38              const char* name; // 4 bytes
39
40              Player(const char* nName)
41              {
42                name = nName;
43              }
44
45              void get_info(void) override
46              {
47                Log(name);
48              }
49            };
50
51
52            int main()
53            {
54              Player player("Raj");
55              player.get_info();  // Raj
56
57              Entity* entity = &player;
58              entity->get_info(); // Raj
59
60              Entity e1(5, 6);
61              Entity e2;
62              e1.get_info();  // 5, 6
63              e2.get_info();  // 0, 0
64              Entity e3(e1);
65
66              e3.get_info();  // 5, 6 - deep copy
67            }
68
69
70
```

Listing 24: Copy constructor example

Can delete copy constructor if you want to prevent copying, such as *Entity() = delete;*.

10.3.2 Constructor initializer list

Creates only the instance needed instead of dual copies in derived classes. If you aren't using them, it affects performance.

```
1             Entity()  // constructor
2         : x(0), y(0), id(55) // member initializer list
3           {}
4
```

Listing 25: Member initializer list example

10.4 Destructors

1. Special method to destroy an object.
2. Stack and heap cleanup.
3. Same name as class with ~ at beginning ~Entity()

```
1
2           #include <iostream>
3           #define Log(x) std::cout << x << std::endl;
4           class Entity
5           {
6           public:
7             int x, y;
8             Entity()  // constructor
9             {
10              x = 0; y = 0;
11            }
12
13              // constructor - function overloading
14            Entity(int vx, int vy)
15            {
16              x = vx; y = vy;
17            }
18
19            ~Entity() // destructor
20            {
21              // nothing to destroy here as ints are local and
    scope is limited.
22              Log(x);
23              Log("destroyed");
24            }
25
26            void get_info(void)
27            {
28              Log(x);
29              Log(y);
30            }
31          };
32
33          void func(void)
34          {
35            Entity e;
36            e.get_info();
37
38            Entity e1(5, 6);
39            e1.get_info();
40          }
41
42
43          int main()
44          {
45            func();
46          }
47
48          Output:
```

```
0
0
5
6
5
destroyed
0
destroyed

```

Listing 26: Constructor & Destructor example with overloading

10.5 Singleton Class

10.6 ”this” keyword

It is available inside the member function of the class.

```
#include <iostream>
#define Log(x) std::cout << x << std::endl;

class Entity
{
  public:
    int x, y;
    Entity(int x, int y)
    {
      this->x = x;
      this->y = y;
    }

    int GetX() const
    {
      return this->x;
    }
};

int main()
{
  Entity e(5, 6);
  Log(e.GetX());  // prints 5
}

```

Listing 27: ’this” keyword example

10.7 Inheritance

Allows us to have relationships between classes.

```
#include <iostream>

#define Log(x) std::cout << x << std::endl;

class Entity
```

```
6           {
7           private:
8             int id; // cant access this outside of Entity class
    objects
9           public:
10            int x, y;
11
12            Entity()  // constructor
13            {
14              x = 0; y = 0; id = 55;
15            }
16
17            Entity(int vx, int vy, int vid) // constructor -
    function overloading
18            {
19              x = vx; y = vy; id = vid;
20            }
21
22            ~Entity() // destructor
23            {
24              // nothing to destroy here as ints are local and
    scope is limited.
25              Log("destroyed");
26            }
27
28            void get_info(void)
29            {
30              Log(x);
31              Log(y);
32            }
33
34            void move(int xa, int ya)
35            {
36              x += xa;
37              y += ya;
38            }
39
40          };
41
42          class Player : public Entity
43          {
44            // anything that is not private in entity class is
    accessible by player
45          public:
46            const char* name; // 4 bytes
47
48            Player(const char* nName)
49            {
50              name = nName;
51            }
52
53            void get_name(void)
54            {
55              Log(name);
56            }
57          };
58
```

```
59
60              void func(void)
61              {
62                Entity e;
63                e.get_info();
64
65                Entity e1(5, 6, 7);
66                e1.get_info();
67              }
68
69
70              int main()
71              {
72                //func();
73
74                Player player("Raj");
75                player.get_info();
76                player.move(1, 2);
77                player.get_info();
78                player.get_name();
79
80                Log(sizeof(Entity));
81                Log(sizeof(Player));
82                Log(player.x);
83                //Log(player.id); // can't access as id is private in
      entity
84
85              }
86
```

Listing 28: Inheritance example

10.7.1 Multiple inheritance

class C : public A, public B
Order of initalization: A → B → C

10.8 Arrow operator

1. For Entity* ptr;, to access methods of the object, we have to de-reference the pointer as follows: (*ptr).method().
2. Instead, could use ptr-¿method() and this would make our life easier as we do not have to de-reference pointer.
3. Can also use in struct for example, as shown below.

```
1         #include <iostream>
2         #define Log(x) std::cout << x << std::endl;
3         struct Vector3
4         {
5           float x, y, z;
6         };
7         int main()
```

```
        {
            Vector3 a;
          a.x = 3.0f; a.y = 4.1f; a.z = 17.2f;
          Log(a.x); //prints 3

          Vector3* b = new Vector3;
          b->x = 4.2f;
          Log(b->x);  // prints 4.2
          Log(b->y);  // prints random float
      }

```

10.9 Friend

10.9.1 Friend Class

```
            class Parent
            {
            private:
                int x, y;

                //class SomeClass can access private members of
    Parent class.
                friend class SomeClass;
            };

```

Listing 29: Friend Class

10.9.2 Friend Function

```
            class Parent
            {
            private:
                int x, y;

                // function() of SomeClass can access private
    members of Parent.
                friend int SomeClass::function();
            };

```

Listing 30: Friend Function

10.10 Polymorphism

Polymorphism → Compile-time & Run-time
    Compile-time → Function overloading and operator overloading
    Run-time → Virtual functions

10.10.1 Compile-time: Function & operator overloading

https://en.cppreference.com/w/cpp/language/operators is a good place to look at all C++ operators.

```
#include <iostream>
#define Log(x) std::cout << x << std::endl;

class Entity
{
private:
  int id;
public:
  int x, y;
  Entity()
  {
    x = 0; y = 0; id = 0;
  }
  //function overloading for constructor - parameterized
  Entity(int vx, int vy, int vid)
  {
    x = vx; y = vy; id = vid;
  }
  //function overloading
  void func(int vx)
  {
    x = vx; y = 2 * x; id = 2 * y;
  }
  //function overloading
  void func(float vx)
  {
    x = x*vx; y = x; id = id/2;
  }
  // operator overloading
  Entity operator + (Entity& obj)
  {
    Entity result;
    result.x = x + obj.x;
    result.y = y + obj.y;
    result.id = id + obj.id;
    return result;
  }
  void get_info(void)
  {
    Log(x); Log(y); Log(id);
  }
};

// overloading << operator
std::ostream& operator<<(std::ostream& stream, const Entity& obj)
{
  stream << obj.x << ", " << obj.y;
  return stream;
}

int main()
```

```
{
  Entity e1(5, 6, 2);
  Entity e2(3, 1, 4);
  e1.get_info();  // 5, 6, 2
  e2.get_info();  // 3, 1, 4
  Entity e3 = e2 + e1;
  e3.get_info();  // 8, 7, 6

  e1.func(2);
  e1.get_info();  // 2, 4, 8
  e1.func(2.1f);
  e1.get_info();  // 4, 4, 4

  std::cout << e1 << std::endl; // prints 4, 4
}

```

Listing 31: Function & operator overloading example

10.10.2 Run-time: Virtual Functions

1. Function overriding, as opposed to function overloading in compile-time.
2. Virtual functions allow us to override methods in subclasses.
3. B is subclass of A. If object of A is needed to call method of B (same method name), then method in A must be defined as virtual function. The method in B overrides method in A.
4. Penalty 1: Virtual functions need additional memory to store the "V table".
5. Penalty 2: Secondly, every time we call the function, it adds penalty to look into which virtual function needs to be used.
6. A class can have a virtual destructor but not a virtual constructor.
7. Virtual function cannot be static and friend.

```
#include <iostream>
#define Log(x) std::cout << x << std::endl;

class Entity
{
private:
  int id;
public:
  int x, y;
  void get_info(void)
  {
    Log(x);
    Log(y);
  }
};
```

```
16
17          class Player : public Entity
18          {
19            // anything that is not private in entity class is
    accessible by player
20          public:
21            const char* name; // 4 bytes
22
23            Player(const char* nName)
24            {
25              name = nName;
26            }
27
28            void get_info(void)
29            {
30              Log(name);
31            }
32          };
33
34
35          int main()
36          {
37            Player player("Raj");
38            player.get_info();
39
40            Entity* entity = &player;
41            entity->get_info();
42          }
43
44          Output:
45          Raj
46          0
47          0
48
```

Listing 32: Without virtual function example

```
1           #include <iostream>
2           #define Log(x) std::cout << x << std::endl;
3
4           class Entity
5           {
6           private:
7             int id;
8           public:
9             int x, y;
10            virtual void get_info(void)
11            {
12              Log(x);
13              Log(y);
14            }
15          };
16
17          class Player : public Entity
18          {
19            // anything that is not private in entity class is
    accessible by player
20          public:
```

```
21              const char* name; // 4 bytes
22
23              Player(const char* nName)
24              {
25                name = nName;
26              }
27
28              void get_info(void) override
29              {
30                Log(name);
31              }
32            };
33
34            int main()
35            {
36              Player player("Raj");
37              player.get_info();
38
39              Entity* entity = &player;
40              entity->get_info();
41            }
42
43            Output:
44            Raj
45            Raj
46
```

Listing 33: Virtual function example

10.10.3 Run-time: Pure virtual functions

1. This is a specific type of virtual function, which has no implementation in base function, but have implementation in the subclass.

2. A class with only virtual functions is called as interface. Since there are no implementations, we cannot instantiate this class.

10.11 Visibility - 3P's

1. **Private**

   Default visibility of a class is private. Only the class has access to it (other than friend functions/classes).

2. **Protected**

   Class and all subclasses in hierarchy can access it. But cannot be accessed in main().

3. **Public**

   Everyone has access to members.

11 Templates

1. Templates are macros on steroids.
2. Compiler writes code based on rules provided in template.
3. Example: Template function.

```
#include <iostream>

// print by macro
#define Log(x) std::cout << x << std::endl;

// print by function
void PrintMsg(std::string msg)
{
  std::cout << msg << std::endl;
}

// print by function - overload
void PrintMsg(int msg)
{
  std::cout << msg << std::endl;
}

// template - avoid overloading and code duplication by using single function
template<typename T>
void templatedPrintMsg(T msg)
{
  std::cout << msg << std::endl;
}

int main()
{
  Log("hello world"); // prints hello world
  PrintMsg("hello world");  // prints hello world
  PrintMsg(4);  // prints 4

  // implicit usage
  const char msg[20] = "hello world";
  templatedPrintMsg(msg);  // prints hello world
  templatedPrintMsg(4.4f); // prints 4.4
  templatedPrintMsg(3);  // prints 3

  // explicit usage
  templatedPrintMsg<int>(8);       // prints 8
  templatedPrintMsg<std::string>(msg); // prints hello world

  // if the main has no call to templatedPrintMsg,
  // the function definition doesn't exist in the code
  // this maybe compiler dependent - eg. CLANG vs MSCV
```

```
}

```

Listing 34: Template function example

4. Templates can be used to define multiple items, such as type and size in the following example. Templates provide *meta-programming* in C++.

5. Example: Template Class

```
#include <iostream>

// template is evaluated at compile time
template<typename T, int N>
class Array
{
private:
  T m_Array[N];
public:
  int GetSize() const
  {
    return N;
  }
};

int main()
{
  Array<float, 5> array0;
  Array<std::string, 6> array1;
  Log(array0.GetSize());  // prints 5
  Log(array1.GetSize());  // prints 6
}


```