Modern_C++ Notes:

https://www.youtube.com/watch?v=F_vIB3yjxaM&list=PLgnQpQtFTOGR50iIOtO36nK6aNPtVq98C

Lecture 0:

LINUX:

- UNIX based OS
- By Linus Torvalds
- LINUX file directory
    - Starts with root: /
    - User can only use Home and folders under it
- Paths
    - Paths are case sensitive
    - Absolute path
        - /home/denimpatel/abc/def/
    - Relative path
        - abc/def
- Terminal commands
    - pwd: print working directory
    - cd: change directory
    - ls: list content of a directory
    - /: root folder
    - ~: home folder
    - .: current folder
    - ..: parent folder
- Structure of LINUX commands
    - $ command [options] [parameters]
    - TAB completion
- mkdir -p to create a chain of folders
- rm -r remove recursively
- cp -r <source> <destination>
- mv <source> <destination>
- Placeholders- can be used with most of the Linux commands
    - *: any set of characters
    - ?: any single character
    - [a-f]: characters range
    - [^a-c]: any characters not in range

- Chaining commands
  - Command 1; command 2: command 3
    - It won't stop if the command fails
  - Command 1 & command 2 & command 3
    - Stops if a command fails
- **htop** command for process manager
- Navigation
  - Use up-down key
  - Ctrl+R for backward search
  - less .bash_history

C++

- Comments
  - // - single line
  - /*   */ - multi-line comment
  - Simply discarded by compilers
- **Programs are meant to be read by humans and only incidentally for computers to execute**
- Clang_format to format the code
- cpplint to check the style

MAIN:
Primary thing and every c++ code contains it

#include<> : check for files in system folders
#include" " checks for files in current folders

I/O streams
stdin, stdout and stderr
std::cout<<" this is going into the output"<<std::endl;

Compilers
- Windows
- GCC
- Clang

To Compile: c++ -std=c++11 -o hello_world test.cpp

Lecture 1: Variables, types, control structure

Variables

- DO INITIALIZE YOUR VARIABLES
    - Hard to catch in bigger programs
- Naming
    - use snake_case - google style
    - Give meaningful variables
    - Don't use negation in names
    - Don't include type in names

Datatypes: https://en.cppreference.com/w/cpp/language/type

- Bool - 0 or 1
- Char
- Int
- Short
- Long
- Float
- Double
- Auto
    - Based on assignment

- Arithmatic operators: +-*/
- Bollean opearators: && and , || or, ! not
- String:
    - #include<string>
    - std::string message = “hello world”;
    - Concatenate with +
    - str.empty()
- Arrays:
    - #include<array>
    - std::array<float, 3> arr
    - Access with arr[0]
    - Number of stored iterms: arr.size()
    - Remove all elements: arr.clear()
    - arr.front()
    - arr.back()
- Vectors:
    - #include<vector>
    - Std::vector vec
    - Use when number of item is unknown beforewise

            - Vector is implemented as a dynamic table
            - vec.push_back(val) - add a new item
            - vec.emplace_back(val)- same as push_back - c++11 edition
            - Many push_back operations force vector to resize many times
                - vec.reserve(2000);
- Variables live in scopes- { }
- Constant variables
    - Use camelCase - google-style
    - const float importantVariable = 5;
- References to variables
    - &
    - float& ref = origional_variable
        - ref and origional_variable points to same location
    - Const with reference
        - const float& ref = origional_reference

Control Structures:

- If statement
    - if(statement){// will be executed if statement is true }
    - break exits
- Switch
    - switch(num){
      case 1:
            std::cout<<"is number 1"<<std::endl;
            Break;
      case 2:
            std::cout<<"is number 2"<<std::endl;
- While
  while(condition){
        //change condition
  }
    - Useful when unsure how many iterations needed
    - It's a curse too! Infinite loop
- For
  for(int i=0; i<max_num; i++){
  }

  for(float number:vec){
        std::cout<<number<<std::endl;
  }

A better way to do this:

```
for(const auto& number: vec){
    std::cout<<number<<std::endl;
}
```

- **USE GIT: https://rogerdudler.github.io/git-guide/**
  - git clone <repo_url> <local_folder>
  - In local folder
    - git add <files>
      - -a for all
    - git commit -m “description message”
    - Git push origin master

Lecture 2: Compilation, Debugging, Functions, Libraries, CMake

---

To make the file executable
- chmod +x filename
  +x means executable

Compilation flags
- std=c++1
- -o to rename the executable
- -Wall to show all the warnings
- -Werror make every warning the errors
- -O0 no optimization
- -O3 or -Ofast for full optimization
- -g for debugging ??

Debugging
- The best option is to use gdb
- Use GDBGUI - comes with a user-friendly interface
- C++ -std=c++11 -g first.cpp
- gdb a.out to start code in debug mode
- COMMANDS
  - print variable_name
  - s to step one line
- gdbgui a.out to open it with an interface

Functions
- Only a single return value
- As many parameters of any type
- Naming of function: Google-style CamelCase
  - for paramerts use snake_case
- Google-style: write small-sized functions
- Function declaration
  int sum (int a, int b);
- Function implementation
  Int sum(int a, int b){
  return a+b;
  }
- Passing big objects by reference

  void DoSomething(std::string& input_string)
    - but function implementation can change the value!!
    - To prevent this from happening, put const

  void DoSomething(const std::string& input_string)
    -

- Function overloading
    - The compiler infers a function from the arguments
    - Return-types plays no role at all
- Default arguments: int sum(int a, int b = 10)
- Don't reinvent the wheels
    - #include <algorithm>
    - Highly optimized ones than self-implemented
- Header/Source separation
    - Move all declaration to header files(utils.h)
        - #include “utils.h” for linking
    - Implementation goes to utils.cpp
    - Compiling multiple files is not trivial
        - Make object files from cpp files (utils.o)
        - c++ main.cpp utils.o -o main

Libraries

- Multiple object files that are logically connected
- Types
    - Static: becomes part of your code, fast, takes a lot of space, named as lib.o
    - Dynamic: slower, can be copied, named as lib.so
- Library is a binary object that contains compiled implementation of some methods
- **Linking** is a process of linking compiled implementation to function declaration

Use **CMake** to simplify the build

```
add_library(tools tools.cpp)
add_executable(main main.cpp)
target_link_library(main tools)
```

Typical project structure

- ❑ Project_name/
    - ❑ CMakeLists.txt
    - ❑ build/
    - ❑ bin/

- ❑ Tools_demo
- ❑ lib/
  - ❑ Libtools.a
- ❑ src/
  - ❑ CMakeLists.txt
  - ❑ Project_name/
    - ❑ CMakeLists.txt
    - ❑ Tools.h
    - ❑ Tools.cpp
    - ❑ Tools_demo.cpp
- ❑ tests/
  - ❑ Test_tools.cpp
  - ❑ CMakelists.txt
- ❑ Readme.md

Build process:

CMakeLists.txt files define the whole build

```
cd <project_folder>
mkdir build
cd build
cmake ..
make -j2
```

Working with CMakeLists.txt

```
#mendatory lines
project(first_project)
cmake_minimum_required(VERSION 3.1)

#to use c++11
set(CMAKE_CXX_STANDARD 11)

set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
```

Another useful commands:

```
#to set additional flags
set(CMAKE_CXX_FLAGS_DEBUG "-g -O0)
set(CMAKE_CXX_FLAGS_RELEASE "-O3)
```

To clean the build

```
cd project/build
make clean
rm -r *
```

Lecture 3: Google Test, Namespaces, classes

```
project(cmake_test_project)
cmake_minimum_required(VERSION 3.0)

# to use headers containing in specific folders
include_directories(${PROJECT_SOURCE_DIR}/src)

## to use Cmake of src folder
add_subdirectory($(PROJECT_SOURCE_DIR)/src)
```

In Terminal type **tree** to visualize the project structure

UNIT TEST - Google test:

- For every function write at least two tests
  - One for normal cases
  - One for extreme cases
- Make writing tests a habit
- Single dummy google test

  ```
  TEST(TestModule, FunctionName){
    EXPECT_EQ(4,FunctionName())
  }
  ```

Add GTest with CMake

To install

```
sudo apt install libgtest-dev
```

```
enable_testing()
add_subdirectory(tests)
```

Run your tests

```
cd <Project_folder>
mkdir build
cd build
cmake ..
make
```

```
ctest -VV (very verbose)
```

Namespaces

- Helps avoid name conflicts
- To group the project into logical modules
- **Avoid: using namespace name in the header files**
- Nested namespaces are possible

```
namespace fun{
	int GetMeaningOfLife(){
		return 42;
	}
}

namespace boring{
	int GetMeaningOfLife(){
		Return 0;
	}
}

int main(){
std::cout<<"fun   way"   <<fun::GetMeaningOfLife<<"voring   way"   <<
boring::GetMeaningOfLife <<std::endl;
}
```

- Nameless namespaces
  - To define constants only available within current cpp file

```
Namespace{
const int localVariable = 10;
}
```

Create new types with classes and structs

```
Class ClassName{
Public: //anybody can use this part
	Image(const std::string& file_name);
	Void Draw();

private: //nobody can access this part
	int rows_ = 0;
	int cols_ = 0;
}; //semicolon in the end

int main(){
```

```
    Image image(some-image.jpg);
    image.Draw();
    return 0;
}
```

- Access modifiers
  - Private //by default
  - Protected
  - Public

Structs:

```
struct NamedInt{
int num;
string name;
}

void PrintStruct(const NamedInt& s){
cout<<s.name<<""<<s.num<<endl;
}

int main(){
// initializing stucts using braced initialization
PrintStruct({10, "world"})
retutn 0;
}
```

- Classes can store data of any type
- Google-style all data must be private
- Use snake_case with a trailing "_" for private members
- Set data in the constructor - classes have at least one constructor
- Clean up data in destructor- classes have only one destructor
- Use initializer list to initialize data

Const correctness

- Const after function name state that it won't change the object variables

Use classes as modules and use header file for class declaration CPP files for the class implementation.

Lecture 4: Move Semantics, Classes

- These are relatively advanced topics

Move semantics

The intuition of lvalue and rvalue:

- lvalues can be written on the left of assignment operator(=)
- rest all are an rvalue
- It is possible to convert lvalue to rvalue
- int&& a =2; //now a can only be used as rvalue
- int b = std::move(a); //never access values after move

Classes:

- Operator overloading

```
Class Human{
public:
      bool operator < (const Humnan& other) const{
      return height_< other.height_;
      }

};
```

Copy constructor:

- Initialize object with new object
- MyClass b(a);
- MyClass c = a;

Copy assignment operator

- NO NEW OBJECT IS CREATED, JUST VALUES ARE PASSED
- b = a
- Myclass& operator = (const MyClass& other) const{...}

Move Constructor

- MyClass operator = (MyClass&& other) {...}
- B = std::move(c);
- MyClass a = std::move(c);

Rule of all or nothing

- Define all of constructors or none.
- use =default to use default implementation
- MyClass() = default
- MyClass(MyClass&& var) = default
- MyClass(const MyClass& var) = default
- MyClas operator =(const Myclass& other) = default

Deleted functions

- void SomeFunction(...) = delete;
- Calling such functions will result in compilation error
- EX: if a class has a constant data member, the copy/move constructors and assignment operators are implicitly deleted.

Inheritance:

- Classes and structs can inherit data and functions from other classes
- 3 types of inheritance in c++:
  - Public
  - Protected
  - Private

Public Inheritance:

- Public inheritance keeps all access specifiers of the base class
- Public inheritance stands for “is a” relationship.
- Derived is kind of a Base class
- Allows Derived to use all public and protected members of a base
- Derived gets its own special functions
  - Constructors
  - Destructors
  - Assignment operators

Initializer list:

```
Class Rectangle{
public:
	Rectangle(int w, int h): width_(w), height_(h){}
Protected:
	int width_ = 0;
	int height_ = 0;
}
```

Function Overriding

- A function can be declared *virtual*
- If a function is *virtual* in Base class it can be overridden in Derived class
- Base can force all Derived classes to override a function by making it *pure virtual*
  - Such Base class is called interface

Function overloading <> Function overriding
overloading:

- Pick from all functions with the same name but different params
- Pick function at compile time
- No class implementation is necessary

Overriding:

- Pick a function of same name and params but from a class hierarchy
- Pick at runtime

Abstract Classes: classes that have at least one pure virtual function

Interface: class that has only pure virtual functions and no data members

How virtual works: (runtime polymorphism)

- A class with virtual functions has a virtual table
- When calling a function the class checks which of the virtual functions that match the signature should be called

Lecture 5: Polymorphism, IO, stringstreams, CMake find_package

Polymorphism:

- Allows morphing derived classes into their base class type:
- Const Base& base = Derived(...)

When to decide you need a hierarchical structure?
Do “is a” and “has a” test.

Square is a shape: can inherit from shape
Student is a human: can inherit form human
Car has a wheel: should not inherit each other

Explicit: to stop the implicit conversion
explicit Square(int size): Rect(size, size){ }
Now square will ONLY accept integers not even float.

Interface:
If you decide every class should be printable in your codebase. So make pure virtual function and force all classes to implement it.

Reading and writing to files:

```
#include<fstream>
using std::string;
using Mode = std::ios_base::openmode;

std::ifstream f_in(string& file_name, Mode mode);
Std::ofstream f_out(string& file_name, Mode mode);
Std::fstream f_in_out(string& file_name, Mode mode);
```

Stringstream
combine int, double, string into a single string
Breakup strings into int, double, string etc.

CMake find_path and find_library

To use header files, use find_path

To find libraries, use find_library

```
find_library(SOME_LIB
NAMES <lib_name>
PATHS <path1> <path2>)
```

Find_package - calls multiple find_path and find_library functions

Chapter 6: Static, Numbers, Arrays, Non-owning pointers, classes

- **Static member variables** of a class
  - Exist exactly once per class, not per object
  - This value is equal across all instances
  - Must be defined in *.cpp file
- **Static member function** of a class
  - Do not need an object to call the function
  - className::MethodName(<params>)

EX: static variable

```
#include<iostream>
using std::cout; using std::endl;
struct Counted{
      Counted(){Counted::count++;}
      ~Counted(){Counted::count--;}
      static int count;
};
int Counted::count = 0; //static variables must be initialized this way
int main(){
      Counted a,b;
      cout<<"Count: "<<Counted::count<< endl;
      Counted c;
      cout<<"Count: "<<Counted::count<< endl;
      return 0;
}
```

EX: static functions

```
#include <math.h>
#include <iostream>
using std::cout; using std::endl;
Class Point{
      public:
            Point(int x, int y): x_{x} y_{y} {}
            static float dist(const Points& a, const Points& b) {
```

```
                int diff_x = a.x_ -b.x_ ;
                int diff_y = a.y_ -b.y_ ;
                return sqrt(diff_x + diff_x + diff_y * diff_y);
            }
    private:
            int x_ = 0; int y_ = 0;
};
int main(){
    Point a(2,2), b(1,1);
    cout<<"Dist is "<<Point::dist(a,b)<<endl;
    return 0;
}
```

P.S.: Not discussed static usage outside the class

Variable Declaration:

- Number representation
    - Alphanumeric - char - 1 byte
    - Numerical
        - Floating Point
            - Single precision - float - 4 byte
            - Double precision - double - 8 byte
        - Integers - int 4 byte

unsigned short Integer (2 byte) representation in memory

0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1

Which represents 37 in memory

Floating point number representation - 4 bytes
Using sign, exponent, mantissa

C style arrays

- The base for std::array, std::vector, std::string
- Length of array is fixed
- Stored in Continuous memory
- sizeof(arrayName)

Pointer

<TYPE>* defines a pointer of type <TYPE>
always initialize pointers to an address or a nullptr

```
int* a = nullptr;
```

Non-owning pointers:
The memory pointed to by a raw pointer is not removed when pointer goes out of scope

Address operator&:
Operator & returns the address of the variable in memory.

```
int a = 42;
int* a_ptr = &a;
```

Pointer dereferencing:
Operator * returns the value of the variable to which the pointer points

```
int a = 42;
int* a_ptr = &a;

int b = *a_ptr;
*a_ptr = 13;
```

- Dereferencing of nullptr: segmentation fault
- Dereferencing of uninitialized pointer: undefined behaviour

Storing function in memory:
- Usually, all are stored sequentially but the compiler can optimize it.
- The concept is called “padding and packing”
- I I I I 0 0 0 0 D D D D D D D D

Lecture 7: Pointers, const with pointers, Stack and Heap, Memory Leaks, Smart Pointers

Pointers can be used with classes

```
Myclass* obj_ptr = &obj;
obj_ptr->MyFunc();
```

Use pointers for polymorphism

- Derived derived;
- Base* base = &derived;

Pointers are just like references, but have additional useful properties:

- Can be reassigned
- Can point to nothing
- Can be stored in a vector or an array

```
#include<iostream>
#include<vector>\
Using std::cout;
struct AbstractShape{
      virtual void Print() const = 0;
}
struct Square: public AbstractShape{
      void Print() const override { cout<<"square \n"}
}
struct Triangle: public AbstractShape{
      void Print() const override { cout<<"triangle \n"}
}
int main(){
      std::vector<AbstractShape*> shapes;
      Square square;
      Triangle triangle;
      shapes.push_back(&square);
      shapes.push_back(&triangle);
      for(auto shape:shapes){
      shape->print();
      }
      return 0;
}
```

NOTE:* or & after the datatype represents the type

```
int a=10;
int* a_ptr = &a;       //a_ptr pointing to location where a is stored
int& a_ref = a; // a_ref and a are pointing to same memory

cout<<a_ref; //prints 10
cout<<*a_ptr; // prints 10
cout<<&a_ref; // prints address of variable a_ref
```

***this* pointer**

- Every object of a class or a struct holds a pointer to itself

>> Using const with pointers:
HINT: read from right to left to see which const refers to what

- Pointer can point to a const variables
  ```
  const MyType* const_var_ptr = &var;
  const_var_ptr = &other_var;

  //But you can not dereference and change the value as
  *const_var_ptr.some_variable = 10 //error
  ```

- pointers can be const:
  ```
  MyType* const var_const_ptr = &var;
  var_const_ptr->a = 10 //this is allowed
  ```

- Pointers can do both at the same time
  ```
  const MyType* const const_var_const_ptr = &var;
  ```

Memory management structure:

Working memory is divided into two parts:
Stack and Heap

Stack:

- Static memory
- Available for short term usage(scope)
- small/limited
- Memory allocation is fast
- LIFO
- Items can be added to the top of the stack with push

- Items can be removed from the top with pop

Heap:

- Dynamic memory
- Available for long time
- Raw modifications possible with new and delete (but usually encapsulated with class)
- Allocation is slower than stack allocation

>> Operators *new* and *delete*:

- User controls memory allocation
- Use new to allocate the data

new operator:

```
int* int_ptr = nullptr;
int_ptr = new int;

float * float_ptr = nullptr;
Float_ptr = new float[6]
```

new returns an address of the variable on heap

P.S.: Prefer using smart pointers!!

delete operator:

- User has to free the memory allocated by new
- Use delete delete[] to free memory

```
int* int_ptr = nullptr;
int_ptr = new int;
delete int_ptr;

{
float * float_ptr = nullptr;
float_ptr = new float[6]
}//heap memory doesn't go out of scope
delete[] float_ptr;
```

Memory Leak:

- Lost access to data available on the heap

Dangling pointer:

```
int* ptr_1 = some_heap_address;
int* ptr_2 = some_heap_address;
```

```
delete ptr_1;
ptr_1 = nullptr;
```

Yet, the ptr_2 has the address, accessing through ptr_2 would result in an error.

Also, you can not delete the same memory twice

Memory leak if nobody has freed the memory, and **Dangling pointer** if somebody has freed the memory in function.

RAII
Resource Allocation IS Initialization
New object -> allocate memory
Remove object -> free memory
Object **OWN** their data

```
class MyClass{
public:
      Myclass(){
      data_ = new someType;
      }
      ~MyClass(){
      delete data_;
      Data_ = nullptr;
      }
private:
      someType* data_;
};
```

With this approach still, you can make a copy of the object!!

shallow vs deep copy

- shallow copy: just copy pointers and not data
- Deep copy: copy data and creates new pointers
- Default copy constructor and assignment operator implement shallow copying

P.S.: use smart pointers!!

Smart Pointers:

- Wraps a raw pointer into a class and manage its lifetime(RAII)
- All about ownership (for heap memory, should not be used for stack)

- #include<memory>
- Std::unique_ptr;
- Std::shared_ptr;

Lecture 9: smart-unique/shared pointers, associative con., Enumeration

Smart pointers:

- They behave exactly as raw pointers
- Can be set to nullptr
- Use *ptr to dereference ptr
- Use ptr-> to access methods
- Smart pointers are polymorphic
- Additional functionalities
    - ptr.get() returns a raw pointer that the smart pointer manages
    - ptr.reset(raw_ptr) stops using currently managed pointer, freeing its memory if needed, sets ptr to raw_ptr

Unique pointer(confusing for beginners):

- No runtime overhead over a raw pointer

```
#include<memory>
auto p = std::unique_ptr<Type> (new Type);
Auto p = std::unique_ptr<Type> (new Type(<params))
```

From c++14 on:

```
auto p = std::make_unique<Type> (<params>) //RECOMMENDED
```

What makes it unique?

- Unique pointer has no copy constructor
    - Intentionally to “make it unique”
- Can not be copied, can be moved
- Guarantees that memory is always owned by a single unique pointer

```
#include<iostream>
#include<memory>

Struct A{
	int a = 10;
};
int main(){
	auto a_ptr = std::unique_ptr<A>(new A);
	std::cout<<a_ptr->a <<std::endl;
```

```
    auto b_ptr = std::move(a_ptr); // now responsibility of managing memory is  os b_ptr and even if a_ptr goes out of scope it won't free memory
    std::cout<<b_ptr->a<<std::endl;
    return 0;
}
```

Shared pointer: (overused)

- Constructed just like a unique_ptr
- Can be copied
- Store a usage counter and a raw pointer
    - Increases usage counter when copied
    - Decreases usage counter when destructed
- Frees memory when the counter reaches 0
- Can be initialized from a unique_ptr

```
#include<memory>
auto p = std::share_ptr<Type> (new Type);
auto p = std::shared_ptr<Type> ();   //RECOMMMENDED WAY

auto p = std::shared_ptr<Type> (new Type(<Parmas>));
auto p = std::make_shared<Type> (<Params>);
```

Ex:

```
#include<iosteam>
#include<memory>

class A{
    A(int a) { std::cout<< "I am alive! \n";  }
    ~A() { std::cout<<"I am dead!! \n";
};

int main(){
    auto a_ptr = std::make_shared<A>(10);
    std::cout<<a_ptr.use_count()<<std::endl;
    {
        auto b_ptr = a_ptr;
        std::cout<<a_ptr.use_count()<<std::endl;
    }
    std::cout<<"Back to main scope!! \n"<<std::endl;
    std::cout<<a_ptr.use_count()<<std::endl;
    return 0;
```

```
} // pointer is deleted here automatically!!
```

When to use what?

- By default use unique_ptr
- If multiple objects must share ownership over something, use share_ptr
- Using smart pointers allows avoiding having destructors in your own class
- Think of any freestanding *new* or *delete* as a memory leak or a dangling pointer.
  - Don't use delete
  - Allocate memory with make_unique, make_shared
  - Only use new in smart pointer constructor if cannot use the functions above

```
#include<iostream>
#include<vector>
#include<memory>

Using std::cout; using std::unique_ptr;
struct AbstractShape{
     virtual void Print() const =0;
};

struct Triangle: public AbstractShape{
     void Print() const override { cout<<"Triangle \n"; }
};

struct Square: public AbstractShape{
     void Print() const override { cout<<"Square \n"; }
};

int main(){
     std::vector<unique_ptr<AbstractShape>> shapes;
     shapes.emplace_back(new Square);
     auto triangle = unique_ptr<Triangle> (new Triangle);
     shapes.emplace_back(std::move(triangle));
     for(const auto& shape: shapes)
     {
          shape->Print();
     }
```

return 0;
}

Associative containers:

Std::map

- #include<map>
    - std::map
- Store items under unique keys
- Implemented usually as red-black tree
- Create from data:
    - std::map<key T, Value T> m = {{key, value}, {key, value},{key, value}};
- Add items to map: m.emplace(key, value)
- Modify or add item m[key] = value
- Get ref to an item m.at(key)
- Check if key present m.count(key) >0;
- Check size m.size();

Std::unordered_map
#include<unordered_map> to use std::unordered_map

- Serves the same purpose as std::map
- Implemented as hash table
- Key types have to be hashable
- Typically used with int, string as a key
- Exactly same interface as std::map

Iterating over maps

```
for(const auto &kv: m){
      const auto & key = kv.first;
      const auto & value = kv.second;
}
```

Typecasting:

- Every variable has a type
- Types can be converted from one to another
- Type conversion is called type casting
- There are 3 ways of typecasting
    - static_cast
    - reinterpret_cast
    - Dynamic_cast

Static_cast:

- static_cast<New Type> (variable);
- Conversion happens compile time
- Rarely needed explicitly
- Pointer to an object of a derived class can be upcast to a pointer of a base class

reinterpret_cast:
- reinterpret_cast<new Type> ()variable;
- Reinterpret the bytes of a variable as another type
- We must know what we are doing
- Mostly used when writing binary data

Dynamic_cast:
- dynamic_cast<Base*> (derived_ptr);
- Used to convert a pointer to a variable of Derived type to a pointer of a base type
- Conversion happens runtime
- If derived_ptr can not be converted to Base* returns a nullptr
- Google-style: Avoid using dynamic casting

Enumeration classes:
enum class EnumType { OPTION_1, OPTION_2};

Use values as EnumType::OPTION_1

```
#include<iostream>
#include<string>
using namespace std;
enum class Channel {STDCOUT, STDERR};

Void Print(Channel print_style, const string& msg){
      switch(print_style){
      case Channel::STDOUT;
            cout<<msg<<endl;
            break;
      }
      case Channel::STDERR;
            cerr<<msg<<endl;
            Break;
      Default:
            cerr<<”skipping”<<endl;
}
```

```
int main(){
      Print(Channel::STDOUT,"hello");
      Print(Channel::STDERR,"world");
      return 0;
}
```

Enum can be defined with explicit values

```
enum class EnumType{
OPTION_1 =10,
OPTION_2 = 13
};
```

Read/write binary files:

- Write a sequence of bytes
- Must document a structure well
- writing/reading is fast
- No precision loss for floating-point types
- Substantially smaller than ASCII files

```
file.write(reinterpret_cast(char*)(&a), sizeof(a))
```

```
//writing to a file in byte format
#include <fstream > // for the file streams
#include <vector >
using namespace std;
int main () {
      string file_name = "image.dat";
      ofstream file(file_name ,
      ios_base :: out | ios_base :: binary);
      if (! file) { return EXIT_FAILURE ; }
      int r = 2; int c = 3;
      vector <float > vec(r * c, 42);
      file.write(reinterpret_cast <char*>(&r), sizeof(r));
      file.write(reinterpret_cast <char*>(&c), sizeof(c));
      file.write(reinterpret_cast <char*>(& vec.front ()),
      vec.size () * sizeof(vec.front ()));
      return 0;
}
```

```
//reading from a file
```

```
#include <fstream >
#include <iostream >
#include <vector >
using namespace std;
int main () {
     string file_name = "image.dat";
     int r = 0, c = 0;
     ifstream in(file_name ,
     ios_base ::in | ios_base :: binary);
     if (!in) { return EXIT_FAILURE ; }
     in.read(reinterpret_cast <char*>(&r), sizeof(r));
     in.read(reinterpret_cast <char*>(&c), sizeof(c));
     cout << "Dim: " << r << " x " << c << endl;
     vector <float > data(r * c, 0);
     in.read(reinterpret_cast <char*>(& data.front ()),
     data.size () * sizeof(data.front ()));
     for (float d : data) { cout << d << endl; }
return 0;
}
```

Lecture 9: Templates, iterators, exception, program input parameters, OpenCV

Generic Programming:

- Datatype agnostic programming

Template functions:

Generic programming uses keyword template

```
Template <typename T, typename S>
T awesome_function(const T& var_t, const S& var_s){
      T result = var_t;
      return result;
}


template<typename T>
T DummyFunction(){
      T result;
      result T;
}

int main(){
      DummyFunction<int>();
      DummyFunction<double>();
      return 0;
}
```

Template meta-programming

- The compiler will generate concrete instances of generic classes based on the classes we want to use
- If we create MyClass<int> MyClass<flaot> the compiler will generate two different classes with appropriate types instead of template parameter

Iterators:

STL uses iterators to access data in containers

- Iterators are similar to pointers
- Allow quick navigation through containers
- The most algorithm in STL uses iterators
- Access current element with *iter

- Accepts -> alike to pointer
- Move to next element in container iter++
- Prefer range based for loops
- Compare iterator with ==, !=, <
- Pre-defined iterators:
    - obj.begin()
    - obj.end()

```
#include <iostream >
#include <map >
#include <vector >
using namespace std;
int main () {
      // Vector iterator.
      vector <double > x = {{1, 2, 3}};
      for (auto it = x.begin (); it != x.end (); ++it) {
            cout << *it << endl;
      }
      // Map iterators
      map <int, string > m = {{1, "hello"}, {2, "world"}};
      map <int, string >:: iterator m_it = m.find (1);
      cout << m_it ->first << ":" << m_it ->second << endl;
      if (m.find (3) == m.end ()) {
            cout << "Key 3 was not found\n";
      }
      return 0;
}
```

Error handling with exceptions

- We can throw an exception if there is an error
- STL defines classes that represent exceptions.
    - Base class: exception
- To use exceptions: #include<stdexcept>
- An exception can be “caught” at any point of the program and even “thrown” further
- The constructor of an exception receives a string error message as a parameter
- This string can be called through a member function what()

Google-style: Don’t use them!!

Program input parameter

```
int main(int argv, char const *argv[]);
```

- argc defines number of input params
- argv is an array of string params

By default

```
Argc =1
argv = "<binary path>"
```

Type aliasing

```
#include <array >
#include <memory >
template <class T, int SIZE >
struct Image {
	// Can be used in classes.
	using Ptr = std :: unique_ptr <Image <T, SIZE >>;
	std ::array <T, SIZE > data;
};
// Can be combined with "template".
template <int SIZE >
using Imagef = Image <float , SIZE >;
int main () {
	// Can be used in a function for type aliasing.
	using Image3f = Imagef <3>;
	auto image_ptr = Image3f :: Ptr(new Image3f);
	return 0;
}
```

OPENCV:

- OpenCV uses own types
- OpenCV trusts you to pick the correct type
- Names of types follow pattern
- CV_<bit_count><identifier><num_of_channel>
- Ex: CV_8UC3: 8-bit unsigned char with 3 channel for RGB

Basic Matix type

- Every image is a cv::Mat
- Mat image(rows, cols, datatype, value)
- Mat_ <T> image (rows, cols, value)
- Initialize with zeros

```
cv::Mat image = cv::Mat::zeros(10, 10, CV_8UC3);
using Matf = cv::Mat_<float>;
```

```
Matf image_float = Matf::zeros(10, 10);
```

- Get type identifiers with image.type();
- Get size with image.rows, image.cols
- I/O:
    - Read image with imread
    - Write image with imwrite
    - Show image with imshow
    - Detects I/O methods from extension
- cv::Mat is a shared pointer
    - Performs Shallow copy for fast operations

Ex: imread

```
#include <opencv2/opencv.hpp >
#include <iostream >
using namespace cv;
int main () {
     Mat i1 = imread("logo_opencv.png",CV_LOAD_IMAGE_GRAYSCALE );
     Mat_ <uint8_t > i2 = imread("logo_opencv.png",
     CV_LOAD_IMAGE_GRAYSCALE );
     std :: cout << (i1.type () == i2.type ()) << std :: endl;
     return 0;
}
```