Learning C++ Vol 1

Table of Contents

Topics	Page
Cpp Introduction, History and Features	1
Compilers, Archivers, linkers, toolsets, IDE	2-5
Constants, Variables, datatypes, Keywords	6-8
C/C++ standard library (std)	9-10
Commenting	11
Arithmetic operators	12
Increment and decrement operators	12
modulus operator and short-hand	12
Statements	12
if / else if / else conditional statements — Ternary Oprator	13
logical and comparison operators	14
bitwise logical operators	14
while and do while loops (while, do while)	15
for loops (for)	16
Arrays (a[])	17
Functions	18-20
Global Variable scopes	21
Break statement (break)	22
Continue statement (continue)	22
switch statement (switch)	23-24
Address operator (&)	25
Pointers (*)	25-26
Passing an array to a function	27
Relationships between Arrays and Pointers	28
Const Keyword	28-29
Array ranges in functions (pointer method)	30
Structures (struct)	30-31
Arrow Operator with pointers to Access Structure members	32
Pass structure to function by value, Pointer	32
Nested structures (pointer and value access)	33

need to change page 8, page 5, page 21

Table of Contents

Topics	Page
Size of operator	34
unions	35
dynamic memory allocation	36
New and delete operators	37
Dynamically allocating arrays	38
Avoiding Dangling Pointer Reference	39
Automatic type detection (auto keyword)	40
Range based for loop	40
Strings	41
Recursive function and Recursion	42
function overloading	43
Object oriented programming classes, objects, public access specifier	44
creating objects from a class (pointer method)	45
Private access specifier	46
class constructors	47
Overloading class constructors	48
Default class constructor parameters	49
Destructors in a class	50
Destructors to release resources	51
static variables	52
Static objects and functions	53-54
Friend functions	55
Inheritance properties	56
Single Inheritance	57
Inheritance continued	58
Protected Access Specifier	59
Public Inheritance	60
Private Inheritance	61
Changing Access level of base class members in derived class	62
Order of Execution of Constructors and Destructors	63
Multiple Inheritance	64 - 65
Calling and Passing to Base Class Constructor in Derived class	66

Table of contents

Topics	Pages
Overridding base class methods in derived class	67
this keyword	68-69
calling methods using base class type	70
Polymorphism	71
Virtual functions	72-73
pure virtual functions	74
virtual functions (extra)	75
Diamond problem	76
Nested classes / Inner classes	77
local classes	78
Operator overloading - (+, -, +=, -=, =+, =-, ++, --, [], (), ->)	79-87
Introduction to error handling / exception	88
Standard exception class / types	89
multiple catch blocks - error handling	90
Function throwing exceptions	91
Custom Exception class	92-93
Overload new and delete operators	94-95
malloc and free dynamic memory allocation	95
Overloading stream insertion, extraction operators << , >>	96
Copy constructors	97-98
IO stream - Introduction	99
Set and unset format flags of IO streams	99
IO stream - boolean values	99
IO stream - Presision width and fill	100
IO - manipulators	100
String class methods	101
getline keyword	101
File handling - opening, writing, reading, appending	102-104
String stream	105
preprocessor directives (#include, #define, #if, #else, #ifndef and many more	106-110
Templates / generics	111-112 (118)
multiple parameters and return values - templates	113

Topics	Pages
templates - additional concepts	114-118
namespaces	118-121
defining classes in separate files	122-123

C++ Introduction, History and Features

About C++

- C++ began as an expanded version of C
- was created by Bjarne Stroustrup in 1979 at Bell Lab
- Initially called C with classes, changed to C++ in 1983

Features of C++

- C++ is a middle level programming level
- object orientated program language (objects / classes ...)
- C++ is a block structured language (function in function)
- increases complexity of design
- C++ is just as efficient as C
- Also, normally the language of choice to create other languages

- used when you need fast and well performing systems
- Direct control of hardware – embedded dev, game dev
- wide platform support – windows, mac, linux, xbox, sony, pretty much everything
- produces machine code to run on the system, native language
- C#, Java – virtual machine languages

References

- Youtube series – C++ – learning Lad
- Youtube series – C++ – The Cherno Project
- Youtube series – Design Patterns – Christopher Okhravi

Documentation

- Text – The C++ Programming Language – Bjarne S.

- Online – msdn.microsoft.com
  - ↳ Documentation / Development tools and languages / Visual studio 2015 / Visual C++

Compilers, Archivers, linkers, toolsets, IDE

What is a tool set

- computer can only understand machine language code which is a combination of 0 and 1
- three basic tools to build C++ applications are the compiler, the linker, and the archiver
- a collection of the above programs and possibly other tools are called a toolset

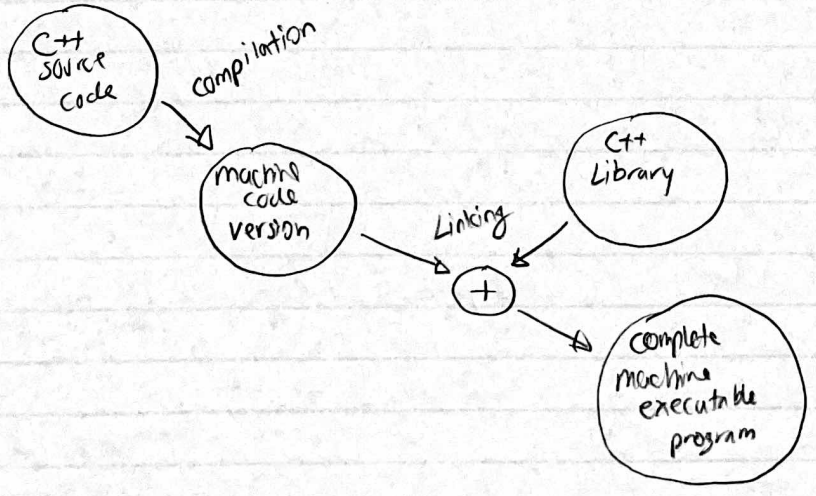
Compilers, Archivers and Linkers

- The compiler takes C++ source files as input and produces object files which contain a mixture of machine-executable code and symbolic references to functions and data
- The archiver takes a collection of object files as input and produces a static library, or archive, which is simply a collection of object files grouped for convenient use
- The linker takes a collection of object files and libraries and resolves their symbolic references to produce either an executable or dynamic library

Executable and dynamic library

- an executable or application is any program that can be executed by the operating system
- dynamic library (shared library) is like an exe but cant be run on its own (dll) - dynamic link libraries

How C++ Works



Compilers, Archivers, linkers

What are command line tools and toolsets

Toolset	Compiler	linker	Archiver
GCC	g++ [.exe]	g++	ar, ranlib [exe]
Visual C++	Cl.exe	link.exe	lib.exe

many more ↓

What is an IDE and why we use one

- the compiler, linker and archiver are command line tools
- names of input files and output files, with config information are passed to the compiler, linker, archiver as text to the command line.
- Invoking these tools are tedious - time consuming
- An IDE (Integrated Development Environment) provides a graphical interface for organizing a collection of source files and describing the binary files that should be generated from them
  - ↳ microsoft visual C++, eclipse, code blocks, and so on

Compilers, Archivers, linkers

How the compiler works

- we write our code (C++) in text form. Take text file and produce a object file (later handed to linker).
  - First preprocesses the code → all preprocessor statements are executed
    ↳ #include, #define, ...
  - then sorts out the text so it can figure out what things are
  → abstract syntax tree creation →
  - main job turn txt to constant data or instructions
  - no such thing as a file, it is all converted down
  - file extensions .cpp / .hpp tell the compiler how to treat that text when building / creating obj files

#include → preprocessor statement → basically opens up file and then pastes it into the current file - very simple

Prove it ↓

```
int multiply (int a, int b)
{
  int result = a * b;
  return result;
#include "EndBrace.h"
```

EndBrace.h file:

```
} //literally only an end brace
```

```
// will compile just fine now
// Preprocess to file → makes a preprocess file .i extension to check out
// You can also produce an assembly file which is a readable version of the binary machine code
```

Visual studio Optimize for speed → Properties / Optimization, debug config → maximize speed
↳ also need to go to code generation / runtime check / default

Compilers, Archivers, linkers

How the linker works

- the linker basically takes all the object files and the C++ library and resolves everything producing one final executable file
- errors will be shown for linker or compiler

example: - if you don't include the int main() { } then you will get a linker error - error LNK1561

- forget a ";" and you will get a compiler syntax error error C2143 - notice it starts with a C

Note: entry point doesn't need to be main, could be custom?

Header files

- traditional used to declare functions and easily connect the function to other files to use - Common place to store declarations (not definitions!)
  - ↳ place to say, yes this function exists!
- you need to copy and paste the declaration in each file that needs the function
  - ↳ use #include!!

Constants, Variables, Data types, Keywords:

Constants - entity that does not change

Integer - 15, 10

floating point - 12.23, 34.33

character - 'a' 'b' 'c'

string "Hello"

bit	bit	bit	bit	bit	bit	bit	bit
0 or 1							

8 bit = 1 byte

All stored as memory in a location - give it a name to make it easy

datatypes - kind of data stored, ie, integer, float, characters, strings, boolean, void

type modifiers - long, short - increase or decrease memory strage

keywords - predefined words for the compiler

int, char, bool, if, for, else, switch, and so on

↳ cant use for varible names

```
#include <iostream>    // input/output
using namespace std;
int main ()
{
  short int age = 23;
  float average;        // uninitialized will have garbage values
  char sex = 'm';
  bool isit = true;
  average = 22.52;
  cout << age << endl << average << endl << sex << endl << isit;
```

↑ print ↑ end line

Variables - primative types

- Data that we want to be able to change, read from and write to.
- int - integer - specific range - 4 bytes traditionally, depends on compiler though
  - ↳ signed - range - 2B to 2B about
    - ↳ 4 bytes = 4 * 8 bits = 32 bits
    - max = $(2^{(32-1)})$ = ~2.1 B one bit reserved if neg or not
- unsigned int - only positive number, no last bit for neg so you get all 32 bits
  - max = 2^{32} = ~4.29 B
- char - one byte of data, normally one character such as "a"
  - only 8 bits, can also be unsigned
- short - two bytes of data, can also be unsigned
- long - normally also 4 bytes of data, can also be unsigned
- long long - 8 bytes of data, can also be unsigned

⇒ at the end of the day characters are numbers 'A' = 65 from ascii

↳ but assumsions are made char will be a character not a number.

- Decimal point variables
- float - 4 bytes
- double - 8 bytes

```
float variable = 5.5   // will default to a double!!
float var = 5.5f   // will force it to a float
double varB = 5.5
```

Additional

- bool - 0 means false and any other number means true (default 1) 1 byte of data

```
bool var = true;   bool var = false;
```

↳ you only need one bit for 0/1 but you have to address it and that only uses bytes so min it could be is a byte!

↳ could store 8 bools in one byte to optimize!

- How to check size of in bytes of data types

```
std::cout << sizeof(bool) << endl;
```

How to Debug in C++ Visual Studio 2017

- break points and memory → you set break points to debug memory
- 99% of the time, computer right and you messed up somewhere
- finding your mistake - can be hard
- break point - point in program where debugger will "pause"
  - ↳ then you check memory
- You can step through line by line and step into functions
- How to set break point in visual studio - F9 or click left most of line
  - ↳ make sure its on lines that are not blank
  - ↳ make sure in debug mode!
  - ↳ click run debug mode → will swap to different layout
    - yellow arrow where it paused
    - three buttons to check out code, step-in, step-over, step-out
    - yellow arrow is not yet executed
- bottom window auto, local, watch
  - type variable name and it will show variable
- View memory - Debug | windows | memory | memory 1
  - ↳ shows all memory, every two values in hex is a byte
  - ↳ find a variable use address: &a ← variable name, presents in hex
  - ↳ debug mode will make it easier like variable = cccccc = uninitialized stack memory
- Disassembly view - shows line and then assembly commands

C/C++ Standard Library (using namespace std)

- Math.h, Cmath (floating point math)

```
sin(x); cos(x); tan(x); // trig functions
```

- Iostream.h, iostream

```
cin >> x >> y; // read words x and y (any type)
cout << "x=" << 3 endl; write line to stdout
```

<< - stream insersion operator

```
int main () {
  int age;
  cin >> age;
  cout << "You have entered: " << age;
  return 0; }
```

C/C++ Standard Library (std)

Comments

- leaving notes for your furture self or other ppl
- Preprocessor
- // Comment to end of line (single line comment)
  /* Multi-line comment */

C++ Arithmetic Operators

Add - x+y

Subtract - x-y

Multiply - x*y

Devide - x/y ← make sure y can never be zero

C++ Increment and Decrement Operators (++, --)

x ++ // Add 1 to x, evaluates to original x
y -- // Subtract 1 from y, evaluates to original y } post fix

++ x // Add 1 to x, evaluates to new value
-- y // Subtract 1 from y, evaluates to new value } Prefix

C++ Modulus, Short-hand Operators (%, +=, -=, ...)

modulus - % - remainder operator ie. num1 % num2

num2) num1
Remainder

Short-hands

num1 = num1 + num2 instead num1 += num2;

Can also use -=, *=, /=, %=

Statements

x=y; // every expression is a statement

int x; // declaration are statements

; // empty statement

{ // A block is a statement

C++ if / else / else if conditional statements

```
if (condition)   // will either be true (1) or false (0)
{
    execute if condition is satisfied // brackets not needed if only one line statement
}
else
{
    execute this if condition is not met
}
```

Comparison operators for conditions (==, !=, <=, >=)

equal not equal less than greater than

Single line short version – Ternary Operator (? :)

```
( value == 5 ) ? (true) : (False);
```

condition if true execute else execute

↳ can be used in preprocessor macros

```
(value == 5) && (Tren expr); // only need than expression
(value == 5) || (Else expr); // only need else expression
```

if ( ) { }, else if ( ) { }, else { }

check for null pointer –

```
const char* ptr = "Hello";
if (ptr)
    Std::cout << "Exists" << ptr << Std::endl;
// instead of if (ptr != nullptr)
```

extra use

★ You can use debugging to check if it goes into if statements or not

– special bool case : bool comparisonResult = x == 5; // equals true if x=5

– logical Not can also be used (!)

```
if ( ! condition )
{
    if condition is a bool and equal to false it will execute
}
```

– you can also nest conditional statements inside eachother

C++ logical and comparison operator

Comparison	logical
x < y // less than	x & y - bitwise and
x > y // greater than	x ^ y - bitwise not
x <= y // less than or equal to	x | y - bitwise or
x >= y // greater than or equal to	x && y - x and y (evaluates if both (not 0))
x == y // equal to	x || y - x or y (evaluates if x or y is true)
x != y // not equal to	

logical - comparison operators example

```
if (date >= 1)
    if (data <= 31)
        cout << "date is valid";
```

instead use

```
if (date >= 1   && date <= 31)
    cout << "date is valid";
else
    cout << "date is invalid";
```

Bitwise logical operators (& (and), | (or), ^ (xor), ~ (complement)
(>> shift right, << shift left))

& (and)	| (or)	^ (xor)	~ (complement)
0 & 0 = 0	0 | 0 = 0	0 ^ 0 = 0	~ 010 = 101
0 & 1 = 0	0 | 1 = 1	0 ^ 1 = 1	(flips all bits)
1 & 0 = 0	1 | 0 = 1	1 ^ 0 = 1	
1 & 1 = 1	1 | 1 = 1	1 ^ 1 = 0	

>> (shift right)

0110 >> 1 = 0011

<< (shift left)

0110 << 1 = 1100

☆ note can also short-hand to

&=, |=, ^=, ~=, >>=, <<=

C++ while and do while loops

while loop

```
while (x) { expression }   // while x is true or =1, execute
                              the expression, x can be a conditional
```

```
int main () {
  int counter = 1;
  while ( counter <= 30 ) {   //while loop prints statement 30 times
      std::cout << "Hello" << std::endl;
      counter ++;
  }
  return 0;  // must return because main returns an int (0 means no error)
}
```

- be careful of infinite loops, program can get stuck
- to create a basic infinte loop — while (1) { }

do while loop

```
do { expressions } while ( expression );
```

```
int main () {
  char input;
  do {
    std::cout << " Hello " << std::endl;
    std::cin >> input;
  } while ( input != 'x' );
  return 0;
}
```

☆ If a condition already exists, use while loops

☆ Do while have a lot less uses then while and for loops

C++ for loops

for (x; y; z) { execute }

x - initialization

y - condition

z - updation

for (int i=0; i < 20; i++) { execute } // executes 20 times then kicks out

- make sure that system cant get stuck in for loop

for (variable declaration; condition; code called) { }

iteration

```
for (int i=0; i<5; i++)    //runs through 5 times
{
  std::cout << "Hello World!" << std::endl;
}
std::cin.get();
//Could even move things around
int i = 0;
for (  ; i<5;  )
{
  //print
  i++;
}
```

- vary useful for traversing arrays and executing a statement x many times

☆ use for arrays and if conditions dont exist

C++ Arrays

- storing many values in one place Single dimensional Array

- int array [6]; → Collection of similar elements of same data type

type name element number

- int array [6] = { 10, 20, 30, 40, 50, 60 }; // filling array of size 6

position 0 1 2 3 4 5 // note that indexed starting at 0

- accessing array - start at zero - Get

  array [2]; // will equal 30

- Individual setting - Set

  array [0] = 15; // change value 0 to 15

- when filling array when declaring, the size is not needed

  int array [] = { 10, 20, 30 }; // builds an array size 3

multidimensional array

- creating an array built of an array

  array [x][y] // x rows and y columns

  int array [2][6] = { 0 1 2 3 4 5

  0 { 10, 20, 30, 40, 50, 60 },

  1 { 12, 22, 33, 44, 55, 66 } };

  Get - accessing

  getArray = array [0][2]; // sets to 30

- you can create as many dimensions as you need

C++ Functions

```
void display ( ) {
  std::cout << "Hello" << std::endl; }

int main () {
  display ();  // function call
  return 0;
}
```

```
returnType functionName ( parameters ) { expressions; }
```

- function prototyping – if you declare a function / create it after main function then you need to say it exists

```
void display ();
int main () {        }
void display () {    }
```

- example

```
int addTwo (int , int);
int main () {
   int result;
   result = addTwo (20, 30);
   std::cout << result << std::endl;
   return 0;
}
int addTwo ( int a , int b ) {
   std::cout << a << b << std::endl;
   return a+b;
}
```

- If a parameter is initialized in function then not needed in the call
- default return type if not listed is int

C++ Functions

inline function: otimized for speed

- used for very short / simple functions to speed up process
- automatically places code in place of where it is called

```
inline void display (int){
    std::cout << "Hello" << std::endl;
```

- then call just like regular function

- increases size of executables, do not use for long functions

multiple return statements - voting age check example

→ can only return one value unless pointers are used

```
bool check (int);
int main() {
    if (check (18))
        cout << "you can vote";
  return 0; }
bool check (int age){
  if (age >= 18)
    return true;
  else
    return false;
```

C++ Functions

- blocks of codes that perform a specific task, maintainability, input and output.
- when program gets to a function it creates variable and result space in stack
  the jumps to where the function is, jumps back with value → takes time
  ↳ all avoided if compiler decides to make it inline though.
- only the main function is exempt to the "has to return a value" rule
  int main () → will default return 0; → do not need to write it

C++ Global variable Scopes

scope - region of a program

variable scope - region of a program where a variable is accessible

can be declared in three areas

↳ declared inside a function

↳ local scope

↳ declared outside a function

↳ global scope

global scope - can be accessed anywhere, outside all functions including (main)

local scope - can only be accessed inside a function

Control flow in C++ (continue, break, return)

Return

```
int multiply (int a , int b)
{
   result = a*b;
   return result;
}
```

- must return same type as outlined in function declaration
- you can put return wherever in function
- exits function once came accross

C++ Break statement - Control flow

- break statement will stop a loop and go back to the regular program
- example - looking for things - find it then break
- used in switch statements

```
int main ( ){
    int counter;
    for (counter = 1 ; counter <=10; counter ++ ) {
        cout << counter <<endl;
        if ( counter == 5 )
            break; // stops looping
    } return 0; }
```

output = 1, 2, 3, 4, 5 break occured and loop is stopped

C++ Continue statement - control flow

- skips the current iteration of the loop and forces it to continue again
- also used in switch statements

```
int main ( ) {
    int counter
    for ( counter = 1; counter <=10; counter ++ ) {
        if ( counter == 5 ){
            continue; // if counter=5 then it will skip all execution and go on looping
                                  remainder skipped
        }
        cout << counter << endl;
    }
```

→ output = 1, 2, 3, 4, 6, 7, 8, 9, 10

C++ switch statement

basically many if statements with a default case if non of the cases are entered

```
switch ( expression )
{
   case constant-expression :
          statements;
          break;
   case constant-expression :
          statements;
          break;
// any number of case statements
   default:
          statements;
}

int main () {
   char grade = 'A' ;
   switch ( input ) {
      case 'A' : {
            cout << "excellent";
            break; }
      case 'B' : {
            cout << " very good " ;
            break ; }
      : and so on
      :

      default : {
            cout << "invalid grade " ;
      }
```

Don't think you need brackets inside case

C++ switch statements

C++ Address Operator - & operator

address of operator & is used to get an address location a variable

```
int main () {
    int age = 100;
    int weight = 60;
    cout << age << endl; // outputs: 100
    cout << &age << endl; // Outputs: 0X23fefc  <- the address location
```

C++ Pointers

- variable that contains an address location for another varible

```
int main () {
    int age = 24;
    bool human = true;

    int *ageptr;
    bool *humanptr;

    ageptr = &age; // store age address location in ageptr pointer variable
    humanptr = &human;

    return 0;
}
```

- used to more easily complete tasks, some functions in C++ they are needed such as dynamic memory allocation

☆ a pointer is an interger that stores a memory address

☆ pointers are used to manipulate memory

C++ Pointers

A Pointer is just an interger that holds a memory address

```
int main()
{
  //void* ptr = nullptr;   or could do =0 or =NULL which is littrally #define 0
    int var = 8;
    void* ptr = &var;
```

(C++11)

pointer to store address — what is memory address of var

- Just a variable holding an integer which is a address
- types really dont matter much

```
*ptr
```

dereferencing pointer — read or write to

```
int main()
{
  int var = 8;
  int* ptr = &var;
  *ptr = 10; // changes value stored at this memory location to 10, var is this memory data
             // therefore var will now equal 10
```

- used good for allocation of memory of the heap (like below)

```
int main()
{
  char* buffer = new char[8]
  memset(buffer, 0, 8);
  delete[] buffer;
}
```

Refrences

* Double pointers

```
int main()
{
  int var = 8;
  int* ptr = &var;
  int** dptr = &ptr;
}
```

Passing an Array to a function

example -

```
void show (int [] ,int );
int main ( ) {
    int numbers [] = { 5, 10, 15, 20};
    int length = 4;

    show (numbers, length);

    return 0; }
void show (int numbers [] , int length ) {
    for (int counter = 0; counter < length; counter++ ) {
        cout << numbers [counter]; }
```

Pass by Address - Pass by reference (uses pointers)

```
void display (int *ptr );
int main ( ) {
    int age = 24;
    display (& age ); // pass in address of age
    return 0; }
void display (int *ptr ) {    // function takes a pointer
    cout << *ptr << endl;
    *ptr = 100; // will change age value to 100 instead of 24
```

Relationships between Arrays and Pointers

```
int main() {
  int number[5] = {29, 55, 44, 88, 89};
  cout << number << endl;  // prints memory location of number[0] value (29)
  cout << *number << endl; // prints 29 as it is the first element in the array
```

→ we can access an array location two ways index or pointer

number[2] ⟶ value of 44

*(number + 2) ⟶ value of 44

Const keyword

- make a element read only, can not change it
- many different uses
- safer code / protection

```
int main() {
  const int pi = 3.142; // set varible when you declare it
```

functions example

```
void display (const int num[], int limit)
 {  }
// makes the array read only and the function can not modify the values at all
```

Const keyword

Array Ranges in Functions

example

```
int numbers [] = {10, 12, 13, 44, 11, 12, 17, 18, 19};
display (numbers, numbers+9); // passing addresses

void display (const int *start, const int *end) {
    const int *ptr;
    for (ptr = start; ptr != end; ptr++) {
        cout << *ptr << endl; //prints all array elements
    }
```

Introduction to Structures (struct)

- Collection of elements of different type, basically an array that excepts different data types

```
struct student {
    int idnum;
    char sex;
} eric, mike; // creates a struct for eric and mike globally

int main () {
    student evan, kyle; // local declaration of struct for evan and kyle.

//How to add contents to the struct
    student kyle = {1111, 'm'}; // set during init
    eric.idnum = 2222; // added after init with dot operator
    eric.sex = 'm';

    cout << eric.idnum << eric.sex << endl;
```

Introduction to structures (struct)

Arrow operator with pointers to access structure members

- Can be used to access struct variables through a pointer

Example

```
struct student {
    int idnum;
    char sex;
};
int main() {
    student eric;
    student *ericptr;
    ericptr = &eric;
    eric.idnum = 1234; // dot operator used to access through variable
    ericptr -> sex = 'm'; // arrow operator use
    cout << ericptr -> idnum << endl; // arrow used to access through pointer
    cout << eric.sex << endl;
```

- Passing structure to function by value, pointer

```
struct student {
    int idnum;
    char sex;
    age;
};
void display (student s) {
    cout << s.idnum << endl;
    s.idnum = 0000; // will not change value in main() because passed by value
void show (student *s) {
    cout << s -> idnum << endl;
    s.idnum = 0000; // will change value in main() because passed by pointer
```

call within

```
student eric = {1234, 'm', 24};
display (eric);
show (&eric);
```

- make sure to define structures before function prototyping

Nested Structures and C++ Dot Operator

Example

```
#include <iostream>
# include <string>
Using namespace std;
 Struct address {
    int house_no;
    string street_name; }; //note need std::string if no std namespace
 struct student {
    string name;
    int rollno;
    address addr; }; // nested structure //What if address *addr;
int main() {
    student eric;
    eric.name = "eric";
    eric.rollnum = 1234;
    eric.addr.house_no = 68;
    eric.addr.street_name = "m g road";

    cout << eric.name << endl;
    cout << eric.addr.house_no << endl;
    return 0; }
```

need to set up in main()

```
    address addr = {50, "test"};
    ericptr -> addr = &addr;
```

change this up as well

OR with pointers

Nested structures and C++ arrow operator (pointers method)

```
    student eric; // create struct eric
    student *ericptr; // create struct pointer
    ericptr = &eric; // fill pointer with eric struct address
    ericptr -> name = "eric"
    ericptr -> rollno = 1234;
    ericptr -> addr . house_number = 68; // ericptr -> addr -> house_no
```

not a pointer so use dot operator

```
    cout << ericptr -> name << endl;
    cout << ericptr -> addr. house_no << endl;
```

C++ sizeof operator with variables, data types, structures, unions

Keyword — sizeof is a compile time operator that determines the size in bytes of variable or data type.

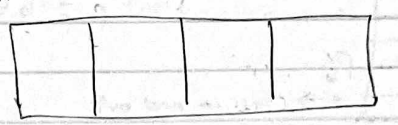
```
int main() {
  cout << sizeof (int) << endl;
  "      size of (short int)    "
  "      sizeof (char)          "
  "      sizeof (float)         "

  int age;
  cout << sizeof (age) << endl;
  // can use on many other things such as structs

// sizeof used with arrays
  std:: uint32_t array[10];
  sizeof ( std:: uint32_t ); // 4
  sizeof (array); // 10 * sizeofInt = 40
  sizeof (array [0]); // sizeofInt = 4
  sizeof (array) / sizeof (array[0]); // numOfBytes / sizeofElement = 40/4 = 10
                                      // number of elements in an array
for ( int i = 0; i < sizeof(arr) / size (arr[0]); i++ ) { }
  ↳ loop through array size amounts
```

C++ Unions

Unions are user defined data type similar to a struct but in unions, you can only store the value in only one member at a time because the memory is shared between the members of the unions. Union memory is shared between all members

```
union unionName {
    datatype data;
    datatype data2; }
```

shared memory

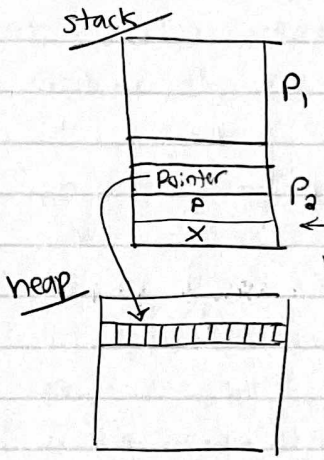


```
union employee {
    int id;
    float salary;
} eric, evan; // can declare here

int main () {
    employee mike, todd;
    todd.id = 25;
    cout << todd.id << endl; //25
    eric.salary = 10;
    cout << eric.salary << endl; //10
    cout << todd.id << endl; //10
```

↳ the value of todd.id was lost when eric.salary was set

C++ New and delete operators | <u>Dynamic memory allocation</u>

dynamic memory allocation is the process of allocating memory while the program is running. The system will allocate space to store variables and data called the stack



int x = 10;

float p = 6.5;

↔ comes in and out

but if you have too much memory to store, it may over right other things

heap is additional memory seperate from the stack memory to store large things to keep the stack safe

→ create a pointer in stack that points to memory location in the heap

C++ New and delete operators

new is used to allocate memory, C++ will allocate from heap and return address

delete is used to deallocate memory

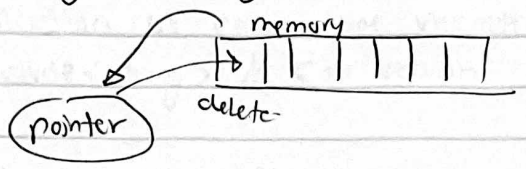
new dataType

```
int main ( )
{
  int *pointer;
  pointer = new int; //will allocate heap memory to store int and address stored in
                     // the pointer created
  cout << pointer; // pointer address
  *pointer = 24; // storing the value of 24

  delete pointer; // must free memory after use
```

Dynamically Allocating Arrays

```
int main ( ) {
   int *pointer = NULL; // nothing
   cout << "how many items you going to enter" << endl;
   int input;
   cin << input;
   pointer = new int [input];
   int temp;
   for (int counter = 0; counter < input; counter++ ) {
      cout << " enter item " << counter+1 << endl;
      cin << temp ;
      * (pointer + counter ) = temp ;
  cout << " you have entered" <<endl;
   for (int counter =0; counter < input; counter ++ ) {
      cout << counter + 1<<" item is " << *(pointer + counter) << endl;
   }
   delete [] pointer;
   return 0;
}
```

Avoiding Dangling Pointer Reference



Say you have a pointer created and used dynamic memory allocation, now you use delete the allocation and refer to the pointer after the fact – you have a dangling pointer

- Note if new fails to allocate memory, it will return a NULL pointer instead
- Dangling pointer – pointer that refers to a once allocated memory location but has been deleted

```
int main() {
  int *pointer = nullptr; // keyword to set pointers to NULL
  pointer = new int;
  if (pointer != nullptr) {  //make sure memory was allocated
    *pointer = 10;
    cout << *pointer << endl;
    delete pointer;
    pointer = nullptr; }
  else
    cout << "memory not allocated";
  return 0; }
```

→ Steps to avoiding dangling pointers

1. delete only at end of functions when you are sure you dont need them anymore
2. set pointer to nullptr to be extra safe otherwise

Automatic type Detection (C++ 11 or higher)

example

```
int age = 26;
// instead you can use
auto age = 26;
```

↳ compiler will deduce what type to use for it based on your initialization → <u>Must initialize it for it to work</u>!?

```
auto age = sum(); // will set to return type of sum() function
```

Range based for loop (C++ 11 or higher)

```
for ( variable : collection/range ) { }   // format

int grades [] = {66, 55, 88, 89, 90, 40, 22}
    for ( int var : grades ) {
        cout << var << endl;
    }
// will print 66, 55, 88, 89, 90, 40, 22
```

Introduction to strings

- Collection of char type variables

```
#include <string> // C++ version

using namespace std;
int main() {
    string name = "eric"; // note: std::string if not std namespace!!
    string lastname = "moser";
    cout << name << lastname << endl;
    string fullName;
    fullName = name + " " + lastName;
    if (name == "eric")
        cout << "name matched" << endl;
    else
        cout << name mismatch" << endl;
```

Recursize Function and Recursion

- a function that calls itself
- example - calculate factorials $4! = 4*3*2*1$

  $n = n \times (n-1)!$
- be careful not to create an infinite loop

```
int factorial (int n) {
   if (n==1)
      return 1;
   else
      return n* factorial (n-1);
```

Function Overloading in C++

- creating a function with the same name but different set of parameters
- <u>have to have the same return type</u> - otherwise error same name

```
void display ( ) {
    cout < "Hi" << endl;

void display (std::string name) {
    cout << "Hi" << name << endl;

display ();  // calls first
display ("eric"); // calls second
```

Object Oriented Programming - Classes and Objects

- example : human

A Class is a blueprint or template from which objects are created.
Object is an instance of a class

```
class HumanBeing {
  public :
      void display () {
          cout << "Hello" ;
      }
};

int main () {
    int a ;
    HumanBeing eric ;  // object of the class HumanBeing
    eric . display () ;  // call a function
}
```

Public access specifier
↳ can be accessed by everyone, other functions, classes
visable everywhere

Class properties, methods, members

```
class HumanBeing {
  public :
      std:: string name ;
      void introduce () {
          std:: cout << "Hi, I am" << name << std:: endl;
      }
};
int main () {
  HumanBeing eric ;
  eric . name = "eric" ;
  eric . introduce () ;
  return 0;
}
```

Creating Objects from a Class (pointer method)

```
class Human {
  public:
    string name;
    void introduce () {
        cout << "Hi " << name << endl;
    }
};
int main() {
   Human eric;  // stack
   Human *evan = new Human(); // dynamically allocate to heap
   eric.name = "eric";
   eric.introduce();
   evan -> name = "evan";
   evan -> introduce();
   return 0;
}
```

Defining methods outside class definition

```
Class Human {
   public:
      string name;
      void introduce();
};
void Human :: introduce ( ) {
   cout << "Hi " << Human::name << endl; // scope not needed if using .hpp
}                                       // and .cpp method to set up
int main() {
   Human eric;
   eric.name = "eric";
   eric.introduce();
   return 0;
}
```

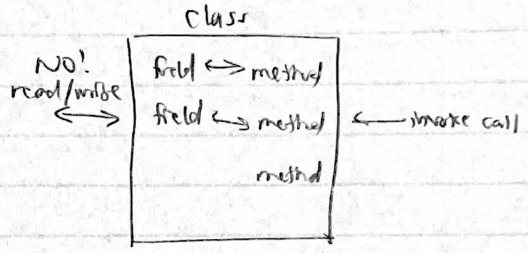
Scope resolution operator (::)
Used to define method outside of class

Private Access Specifer

```
Class Human {
  private:
      int age;
  public:
    void displayAge() {
        cout<< age << endl; }
    void setAge (int value) {
        age = value; }
};
int main() {
  Human eric;
  eric.setAge(26);
  eric.displayAge();
```

private - visable only to methods of own type aka only the methods of the class can access the variable

→ all fields/variables should be private



- keeps all access to the class only

→ can also create private methods that can only be used by methods of the class to hide/protect inner working of a system

→ <u>Default access specifier if you do not use any keywords!</u>

Class Constructors

- function with the same name as a class with no return type and will be executed upon creation of a object.
- can not be called manually only automatically

```
class Human {
  public:
      Human () {
          cout << " Constructor " << endl;    //or defined outside with scope operator
      }
};
int main () {
  Human eric;
  return 0;
}
```

- used to initialize properties of a class such as variables
- setting defaults/initial values of private variables
- must be public to work properly outside the class

Overloading Class Constructors

```
class Human {
  private:
    string name;
    int age;
  public:
    Human ( ) {
      age = 0;
      name = "noname";
    }
    Human (int iage) {
      age = iage;
      name = "noname"; }
    Human (string iname) {
      age = 0;
      name = iname; }
};
int main () {
  Human eric (26);    // uses constructor B
  Human eric;    // uses A
  Human eric ("eric");    // uses C
```

A

B

C

Default class constructor parameters

```
class Human {
  private:
      string name;
      int age;
  public:
      Human (string iname = "noname", int iage = 0) {
          cout << " constructor " << endl;
          name = iname;
          age = iage; }
      void speakUp () {
          cout << name << age << endl;
      }
};

int main () {
    Human eric ();  // can call like this because defaults are set
    Human evan ("evan", 22);// can also pass like this
    eric . speakup ();
    return 0;
}
// if default values are set then you dont need to pass
    a value with the object creation
```

Destructors in a class

A special method that is executed upon distruction of a class at the end of a program. When the program goes out of script, it will be called

```
Class Human {
  public:
    Human () {
      cout << "constructor called" << endl;
    }
    ~Human () {          // How to create a distructor!
      cout << "destructor called" << endl;
    }
};
int main () {
   Human *eric;
   eric = new Human ();  // Constructor called
   delete eric;  // this is when destructor is called
   return 0;
}
```

When you need delete to call a destructor?

If you create a object with Human eric; then the destructor will automatically be called and no need for delete keyword. delete keyword is only used if you dynamically allocated memory, delete must be used in that case to call the destructor

new used, then deleted needed to call destructor
otherwise it gets auto called at end of program

Destructors to release resources

↳ deallocation of memory

```
class Human {
  public:
    Human ( string iname , int iage ) {   // Constructor - allocates memory
      name = new string;
      age = new int;
      *name = iname;
      *age = iage;
    }
    ~Human () {              // de-allocate memory
      delete name;
      delete age; }
    void display () {
      cout << *name << *age << endl; }
};
int main() {
  Human *eric = new Human ("eric", 26); // one line creation
  eric -> display();
  delete eric; // destructor will be called here - must delete object
}
```

You can use constructors to dynamically allocate memory and destructors to deallocate the memory and free the heap back up

Static Variables

Static variables in a function: When a variable is declared as static, space for it gets allocated for the lifetime of the program on the heap. Even if the function is called multiple times, space for the static variable is allocated only once and the variable in the previous call gets carried through the next function call. This is useful for implementing coroutines in C/C++ or any application where the previous state of a function needs to be stored.

```
int main () {
    display();  // prints 1
    display();  // prints 2
    display();  // prints 3
}
void display () {
  static int counter = 0;  //set to zero first call
  cout << ++counter << endl;  // adds 1 to counter
//Not Java doesn't allow static local variables in functions
```

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

```
class Human {
  public:
    static int human_count;  // has class scope
    Human () { human_count++; }  // every new object, human_count is incremented
};
int Human:: human_count = 0;  // set value outside of class
int main () {
  cout << Human:: human_count;  // has class scope so you don't need an object first
  Human eric;  // count = 1
  Human evan;  // count = 2
```

Static Objects and functions

<u>Static objects</u> : just like variables, objects also when declared as static have a scope until the lifetime of the program.

<u>Note</u>: The destructor of a static object is only called on the exit of the main() function.!

```
class Human {
    public:
        //whatever you want in your class
};
int main() {
    static Human eric;
    return 0;
}
```

Static functions: Just like the static data members or static variables inside the class, static members functions also doesnt depend on object of class we are allowed to invoke a static member function using the '.' dot operator but it is recommended to invoke using the class name and scope resolution operator Static member functions are allowed to access only the static data members or other static member functions - not static → cant access!!

```
class Human {
    public
        static int human_count;
        Human () { human_count++; }
        static void humanTotal () {    // can only use static memoers / variables!!
int Human cout << human-count << endl;
    }
};
int Human :: human_count = 0;
int main () {
    Human eric;
    Human evan;
    Human :: humanTotal();  // use scope resolution operator to call because it is
                                object independent!
```

Static keyword

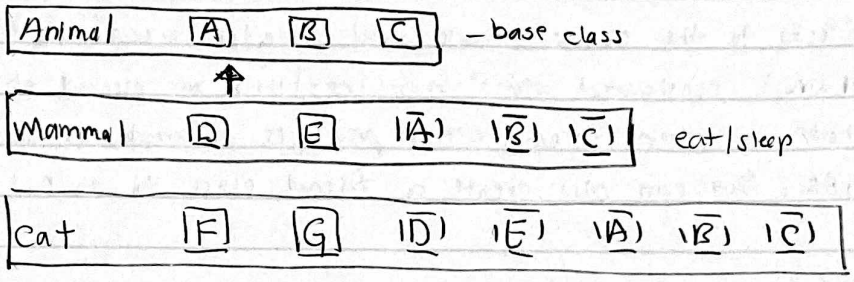
Friend functions

A friend function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges. Information is shared between friends. You can also create a friend class to another class

```
class Human {
  private:
    string name;
    int age;
  public:
    Human (string iname, int iage) {     // constructor
      name = iname;
      age = iage;
    }
    void tellMe () {
      cout << name << age << endl;
    }
  friend void display (Human man);   // prototype to create friend access
};
void display (Human man) {
  cout << man.name << man.age << endl;
}
int main () {
  Human eric ("eric", 26);
  display (eric);
  return 0;
}
// friend class
friend class classname;  // goes in place of the function prototype
in the class description.
```

Inheritance, Polymorphism

Single inheritance - a class which inherits variables and methods of another class



Cat is inheriting from mammal and mammal inherits from animal. Cat will have all fields and methods from mammal and animal.

→ Re-use code to not have to redefine everything

Sup-type : Cat is a subtype of mammal and animal (Dependent) (sub class)

Super-type : Animal is a supertype of mammal and cat (Ancestor)

child : direct decendent

parent : direct ancestor

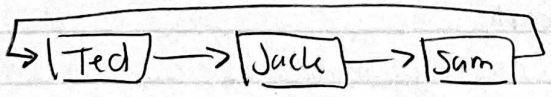
<u>Is-a vs has-a</u> - Think before using inheritance

A cat is-a mammal which is-a animal

<u>But</u> A car has-a steering wheel and has-a engine

↳ No inheritance for this

→ watch out for circular inheritance / dependencies



→ undetermined states - BAD

→ Overriding - you can override what functions do when you inherit for such cases when a cat eats from mammal but might have a more specific way to eat.

Inheritance - example - single inheritance

```
class Person {
  public:
    string name;
    int age;
    void setName (string iname) { name = iname; }
    void setAge ( int age ) { age = iage; }
};
class Student : public Person {        // can use all public fields/methods from Person
  public:
    int id;
    void setID ( int iID ) { id = iID; }
    void Introduce() {
      cout << name << age << id << endl;
    }
};
int main() {
  Student eric;
  eric.setName ("eric");
  eric.setAge (26);
  eric.setID (12345);
  eric.introduce();
  return 0;
}
```

Inheritance

Protected Access specifier

protected - the members declared as protected are accessible from outside the class <u>BUT</u> only in a class derived from it. Similar to private but has this additional ability. Can not touch protected outside of the defined such as in a main function is not allowed.

	public	protected	private
same class	✓	✓	✓
derived class	✓	✓	X
outside classes	✓	X	X

```
class Person {
  protected:
      string name;
  public:
      void setName (string iname) {
          name = iname;
      }
};
class Student : public Person {
  public:
      void display () {
          cout << name << endl; // OKAY to access Here!
      }
};
int main () {
    Person eric ;
    //eric . name = "eric";  // Not allowed to do this!!
    eric . setName ("eric");
    eric . display ();
```

Public Inheritance

Using the public keyword to create an inheritance class. All public members will act public in the new class and all protected will act like protected in the new class. Private will not be inherited.

```
Class Person {
  protected:
      string name;
      int age;
  public:
      void setName (string iname) { name = iname; }
};
class Student : public Person {    // public inheritance!!
    // code
};
```

Protected Inheritance

Using protected keyword to create an inherited class. All public and protected members / variables of the base class will act a protected members in the derived class.

```
class Student : protected Person {
    // code
};
```

↳ Base class members can only be used within the derived class or further derived classes. Int main() or any other function can not touch any of the base class members

Private Inheritance

Using the private keyword to create an inherited class. All public and protected members of the base class will act as private members in the new derived class. Base inherited members can only be used within the new class members.

```
class student : private Person {
    //code
};
```

Changing Access Level of Base Class' members in Derived class

using access declaration to change access level of base class members in derived class. This also works for changing method access levels

```
class Person {
  protected:
    string name;
  public:
    void setName (string iname) { name = iname; }
};
class Student : private Person {
  public:
    Person :: name ;  // will force the inherited name variable to now be public
    void display() { cout << name << endl; }
    void studentSetName (string iname) { setName (iname); }
    Person :: setName ;  // this method is now public!!
int main() {
  Student eric;
  eric.name = "eric";  // this is now public access so its okay
```

Order of Execution of Constructors and Destructors

```
class Person {
  public:
    Person () { cout << "Person Constructor" << endl; }
    ~Person () { cout << "Person destructor" << endl; }
};
class Student : public Person {
  public:
    Student () { cout << "Student Constructor" << endl; }
    ~Student () { cout << "Student Destructor" << endl; }
};
int main () {
  student eric;
  return 0;
}
```

<u>Output</u>

```
Person Constructor
Student Constructor
Student Destructor
Person Destructor
```

Multiple Inheritance

The process of inheriting from two or more classes

```
class Father {
  public:
      int height;
      void askFather () {
          cout << "I am father, what do you want" << endl; }
};
class mother {
   public:
       string skincolor;
       void askMother () {
           cout << "I am mother, what do you want" << endl; }
};
Class Child : public Father , public Mother {
    public:
       void askParents ( ) {
           cout << "I am asking parents" << endl; }
       void setColorandHeight ( string icolor , int iheight) {
           skinColor = icolor;
           height = iheight;
       }
       void display () {
           cout << height << skincolor << endl;
};
int main () {
  Child eric;
  eric.setColorandHeight ("white", 5.10);
  eric.display ();
  eric.askFather ();
  eric.askMother ();
  return 0;
}
```

Multiple Inheritance

Calling and Passing to Base Class Constructor in Derived class

How to pass values from derived class to base class constructor. AND works with multiple Inheritance

```
Class Father {
  protected:
    int height;
  public:
    Father (int h) {
      cout << "Father constructor called" << endl;
      height = h;
    }
};
Class child : Father {
  public:
    child (int x ): Father (x) {
      cout << "child constructor" << endl;
      height = x ;  } // don't think you need this, but initializes value
    void display () {
      cout << "height is" << height << endl;
    }
};
int main () {
  Child eric (26); // will end up setting a value in the base class through constructors
  eric . display ();
  return 0;
}
// print out:
//  Father constructor called
//  child constructor
//  height is 26
//
```

sets value

passed value to base class

Overriding base class methods in Derived Class

You can change what functions do in the derived class and then all derived classes from there will inherit the new operation of the function.

```
class Person {
    public:
        void introduce () {
            cout << "hi i am a person" << endl;
        }
};
class Student : public Person {
    public:
        void introduce () {          //same name, so it is overriden
            cout << "hi I am a student" << endl;
        }
};
int main() {
    Student eric;
    eric.introduce();
    eric.Person::introduce();  //How to call original function before it was overritten
```

extra → `eric.Person::introduce();`

// use Person::introduce();
to call original method in derived class

extra

this keyword in C++

The this pointer accessible only within the nonstatic member functions of a class, struct or union type. It points to the object for which the member function is called. Static members do not have a this pointer.

```
this -> member-identifier
```

example:

```
void Date :: setMonth (int mn) {
    month = mn;              // These
    this -> month = mn;      // are all
    (*this). month = mn;     // equivalent!!
}
```

Also commonly used to return the current object from a member function

```
return *this;
```

Can also be used as a safe-guard against self-reference

```
if ( & object != this ) {
    // do not execute for case of self-reference
```

this keyword in C++

```
Class Person {
  private :
    int age;
  public :
    void setAge ( int age ) {
      this -> age = age;   // can be used when reusing variable names
    }
    void showAge ( ) {
      cout << this-> age << age << endl;
                    // same thing
    void setAge2 ( int iage ) {
      age = iage;          // same
      this-> age = iage;   // execution
    }
```

this keyword is used within a method to point to the class variable

Calling methods using Base class type

```
class Person {
  public:
    void introduce () {
        cout << "Hey from person " << endl); }
};
class Student : public Person {
  public:
    void introduce () {                    // overrides introduce function
        cout << "Hey from student " << endl);
};
void whosThis ( Person p ) {               // defined outside of classes
   p. introduce ();
}
int main () {
   Student eric ;
   eric . introduce ();  // prints Hey from student
   whosThis (eric) ;  // prints Hey from student

      // will take a student because it is derived from Person
        // will dig down to Person class and use that introduce version
   return 0;
}
```

Polymorphism

Polymorphism - means having many forms.

One key feature of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.

Polymorphism is the art of taking advantage of this simple but powerful and versatile feature

Virtual Functions

The virtual function provides the ability to define a function in a base class and have a function of the same name and type in a derived class called when a user calls the base class function. [Prevent "all the way back" base calls like on page 70 and call overwritten method instead]. Used in extending classes for testing so it calls overwritten "test version" instead of real version.

☆ ↳ Perfect Polymorphism example

```
Class Person {
   public:
      virtual void introduce() {
         cout << "Hey from Person" << endl; } };
class Student : public Person {
   public:
      void introduce() {
         cout << "Hey from Student" << endl; } };
class Farmer : public Person {
   public:
      void introduce() {
         cout << "Hey from Farmer" << endl; } };
void whosthis (Person &p) {
   p.introduce(); }
int main() {
   Farmer evan;
   Student eric;
   whosthis(evan);
   whosthis(eric);
   return 0;
}
// Output is now "Hey from Farmer" then "Hey from Student"
// instead of walking all the way back to the base class introduce
// implementation
// If introduce was never overwritten then it will just invoke base class
//
```

Virtual functions / Inherited Attributes / Hierarchical Nature

If you make a base class function virtual then it will always be overwritten with new implentation without having to redecare a derived class as virtual.

```
Class Person {
  public:
    virtual void introduce(){
        cout << "Hey from Person" << endl; } };
class Student : public Person {
  public:
    void introduce(){
        cout << "Hey from student" << endl; } };
class GStudent : public Student {
  public:
    void introduce () {
        cout << "Hey from GStudent" << endl; } };
void whosthis ( Person &p ) {
  p. introduce (); }
int main () {
  Person eric;
  Student evan;
  GStudent mike;
                                  // Output will be as follows
  whosthis ( eric );              // "Hey from Person"
  whosthis (evan);                // "Hey from Student"
  whos this (mike);               // "Hey from GStudent"
                                  //->Notice that Students introduce was not
  return 0;                       // virtual and GStudent still acted as
}                                 // if it was there
                                  //->If GStudent didnt overwrite introduce then
                                  // Students implentation would have happened
                                  //
```

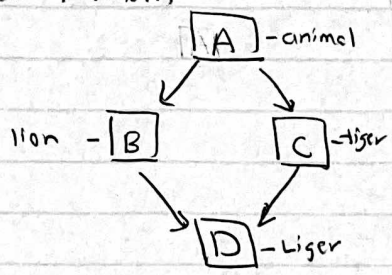
Pure virtual functions

Abstract classes cannot be instantiated (class with one or more pure virtual methods)
Derived classes need to overwrite / implement all inherited pure virtual functions.
If they do not, they too will become abstract. You can also define the pure virtual function outside of the class with the scope resolution operator.

```
class Person {
  public:
    virtual void introduce () = 0;   // =0 sets as pure virtual, all derived have to
};                                   // implement it then
void Person:: introduce () {
    cout << "Hey from Person" << endl; }    // How to define a pure virtual

class Student : public Person {
  public:
    void introduce () {          // Must create a introduce!!
      cout << "Hey from Student" << endl;
      Person :: introduce(); }   //How to call implemented pure virtual
};

int main() {
  Student eric;
  eric.introduce();
// Person evan;  //-can't implement because of pure virtual, this will error!!
```

Virtual functions

Diamond Problem



A common problem in object orientated languages. D doesn't know if it should use A through B or C

↳ solve with virtual inheritance

```
Class Animal {
 public:
   void walk () {
      cout << "animal walks" << endl;
   Animal () {
      cout << " Arimal constructor " << endl; 3;
Class Tiger : virtual public Animal {      // virtual added so that there is
  public:                                  // only one animal object otherwise
    Tiger () { cout << "Tiger const" << endl; 3; // there would be two different animal
Class Lion : virtual public Animal {       // objects
  public:
    Lion () { cout << " Lion const " << endl; 3 3;
Class Liger : public Tiger, public Lion {
   public:
      Liger () { cout << " Liger const " << endl; 3 3;
int main() {
   Liger eric;
   eric . walk (); // will result in diamond problem and ambigeous error
                   // solved with virtual keyword
// Prints out the following
// animal constructor                  // without the virtual keyword, animal
// tiger const                         // constructor would have be called twice
// lion const
// liger const
// animal walks
```

Nested Classes or inner classes in C++

```
using namespace std;
class Person {
    public:
        string name;

        class Address {
        public:
            string country;
            string stName;
            int houseNum;
        };
        Address addr;
        void AddressPlease () {
            cout << name << addr.country << addr.stName << addr.houseNum << endl);
        }
};

int main () {
    Person eric;                    // can't declare Address addr;
    eric.name = "eric";             // in here
    eric.addr.country = "USA";      // instead call Person::Address ad;
    eric.addr.stName = "James";     // use scope operator
    eric.addr.houseNum = 158;
    eric.AddressPlease ();
    return 0;
}
```

Nested Address class with in a person class

Local Classes (C++ only)

A local class is defined within a function definition. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

```
using namespace std;
void studentList();  //prototype
int main() {
    studentList();
    return 0;
}
void studentList() {
    class student {
    public:
        int age;
        void display() {
            cout << name << age << endl;
        }
    };
    Student eric;
    eric.name = "eric";
    eric.age = 26;
    eric.display();
}
// can not access outside of the function, function scope only
```

Operator Overloading

The operator keyword declares a function specifying what operator-symbol means when applied to instances of a class. This gives the operator more then one meaning and "overloads" it. The compiler distinguishes between the different meanings of an operator by examining the types of its operand.

Syntax

type operator operator-symbol (parameter-list)

Redefinable operators

, ! != % %= & && &= ( ) * *= + ++ += - --

-= -> ->* / /= << <<= = == > >> >>= [] ^ ^=

~ delete new and more

nonredefinable operators

. .* :: ?: # ##

Operator Overloading

```
using namespace std;
class Marks {
int intmark;
int extmark;
public:
    Marks () {        // default constructor
     intmark = 0;
     extmark = 0;
    }
   Marks (int im, int em) {
     intmark = im;
     extmark = em;
   }
   void display () {
     cout << intmark << endl << ext mark << endl;
   }
   Marks operator+ (Marks m) {        // Overload the + operator to
     Marks temp;                      // be able to add two Marks
     temp.intmark = intmark + m.intmark;   // classes together
     temp.extmark = extmark + m.extmark;
     return temp;
   }
};
int main () {
  Marks m1 (10, 20), m2 (30, 40);
  Marks m3 = m1 + m2;  // only doable because of new overloaded + operator
  m3.display();
  return 0;
}
// displays  40  60
             10+30  20+40
```

Operator Overloading

You can also declare the overloaded operator outside of the class with the scope operator. Using page 80 example except "-" version

inside class add:

```
    Marks operator- (Marks m);          // overloading '-' operator
```

outside of class:

```
    Marks Marks :: operator- (Marks m) {
        Marks temp;
        temp.intmark = intmark - m.intmark;
        temp.extmark = extmark - m.extmark;
        return temp;
    }
```

// This is done when creating h file with class and cpp file with
// implementation

Operator Overloading - Short hand operators

You can also overload short hand operators such as += and -=

```
Class marks {
int mark;
public:
    marks () {
     mark = 0;
     }
    marks (int m) {
     mark = m;
     }
   void YourMarkPlease () {
       cout << "your mark is" << mark << endl; }
   void operator+= (int bonusmark) {
     mark = mark + bonusmark;
   }
   friend void operator-= (marks &cur Obj, int redmark);
};
int main() {
   marks ericsmark (70);
   ericsmark. YourMarkPlease ();  //prints 70

   ericsmark += 20;
   ericsmark. YourMarkPlease ();  //prints 50

   ericsmark -= 20;
   ericsmark. YourMarkPlease();  //prints 90
```

Overloading Increment and decrement Operators (Prefix)

```
class marks {
  int mark;
  public:
    Marks ( ) { mark = 0; }
    Marks (int m) { mark = m ; }
    void yourMarkPlease ( ) { cout << "your mark is" << mark << endl; }

    void operator++ ( ) { mark += 1; }
    friend void operator-- (Marks &);                    // Both in class and out of
}                                                        // class with friend function
void operator-- (marks &m) { m. mark -= 1; }             // examples
                                                         //

int main ( ) {
  Marks ericsmark (75);
  ericsmark . yourMarkPlease ( );

  ++ ericsmark ;
  ericsmark . yourMarkPlease ( );
//++ ericsmark . yourMarkPlease ( ); -> need Marks return type to be able to do this
  -- ericsmark ;
  ericsmark . yourMarkPlease ( );
}
```

Overloading Operators increment and decrement (postfix) x++, x--

Using the same example on page 83

within the marks class:

// means we are talking about Postfix form!

```
Marks operator++ (int) {
  marks duplicate (*this);    // create a duplicate
  mark += 1;                  // add 1 to mark
  return duplicate;           // return new object
}
friend marks operator-- (marks&, int);
```

Outside of class:

```
Marks operator-- (Marks &m, int) {
    Marks duplicate(m);
    m.marks -= 1;
    return duplicate;
}
```

//similar int main() {} as page 83

```
(cris mark++).YourMarkPlease();
(cris mark--).YourMarkPlease();
```

Operator Overloading special [ ]

Have to be non static and cannot use friend functions to overload these guys [], (), ->

example: overload special [] to work with our class

```
class Marks {
int subjects[3];
public:
    Marks (int sub1, int sub2, int sub3){
        subjects[0] = sub1;     //stores in array
        subjects[1] = sub2;
        subjects[2] = sub3;}
    int operator[] (int position) {
        return subjects[position];
    }

int main() {
    Marks eric(22, 25, 55);
    cout << eric[0] << endl;
    cout << eric[1] << endl;
    cout << eric[2] << endl;
```

Overloading Special Function Call Operator ()

```
class marks {
int mark;   // automatically private
public:
  Marks (int m) {
    mark = m;
  }
  void YourMark() { cout << mark << endl; }
  Marks operator() (int mk) {
    mark = mk;
    return *this;
  }

int main() {
  Marks eric (85);  // constructor call
  eric.YourMark();  // output 85

  eric(49);  // new overloaded operator call
  eric.YourMark();  // output 49
// Not a new way to call but a new custom function that is easy to call!!
```

Overloading Special arrow operator ->

unary operator when overloading. Not allowed friend functions because special

must return a pointer

example - similar to page 86 marks class

inside marks class:

```
Marks *operator -> () {
    return this;
}
```

outside class and in int main()

```
Marks eric (65);
eric.whatsYourmark();
eric -> whatsYour mark(); // new overloaded arrow operator
```

Rules and Restrictions for operator overloading

- you cant change the presidence (ie * before +)
- you cant not change the number of operands a operator takes
- you cant overload ? :: .* . operators
- except = operator, all are inherited in derived classes
- make sure meaning doesnt change (like make a * add instead)

Introduction to Error / exception handling

Keywords : throw, catch, try, except

handle

```
#include <iostream>
#include <exception>

using namespace std;

int main () {
   int a = 10, b= 0;
   int c;
//  c = a/b;  // will cause an error and crash! divide by zero
  try {
  if ( b == 0)
      throw "divide by zero error";
  c= a/b;
  cout << c << endl;
  } catch ( const char *e) {
    cout << "exception occured" << endl << e;
  }
//output will print  exception occured
                     divide by zero error
```

① Use try block for code that might throw an exception

② use throw keyword to throw the exception once it occurs

③ use the catch block to handle if the exception occurs

Standard Exception Classes / types

→ already defined in C++ with #include <exception>

the lesser <stdexcept> can also be used

Exexception

std:: bad_alloc

std :: bad_cast

std :: logic_failure → std:: domain_error , std:: invalid_argument , std:: length_error
std:: out_of_range

std :: bad_typeid

std :: bad_exception

std :: runtime_error → std:: overflow_error , std:: range_error , std:: underflow_error

```
#include <iostream>
#include <stdexcept>
using namespace std;
int main() {
  int a=10, b=0, c;
  try {
    if (b==0)
      throw runtime_error ("devide by zero error");
    c=a/b;
    cout<< c << endl;
  } catch (runtime_error &error) {
      cout<< "exception occured" << endl;
      cout<< error.what();
  }
  return 0;
}
```

Multiple Catch blocks - Exception handling

```
int main () {
  try {
    throw 20; // which is an integer
  } catch (const char *err) {
      cout << err << endl;
  }
  catch (int error) {          ← this one will be executed
    cout << error << endl;
  }
  catch (runtime_error e) {
    cout << e.what() << endl;
  }
```

```
int main () {
  try {
    throw "error";
  } catch (...) {      //will handle any kind of exceptions (int, char, runtime_error, etc.)
    cout << "some exception occured" << endl;
  }
```

You can also add the catch (...) to the end of multiple catch block to handle all exceptions if the individual catches don't work

↳ If none match then catch in catch (...)

Functions Throwing Exceptions

what kind can be called

```
void test () throw (int, char, runtime_error) {
  throw 20;
  throw 'a';
  throw runtime_error ("error");
}

int main () {
  try {
   test();
  } catch (int e) {
     cout << "integer type" << e << endl;
  }
  catch (char c) {
    cout << "char type" << c << endl;
  }
  catch (runtime_error r) {
    cout << "runtime" << r.what() << endl;
  }
  return 0;
}
```

Bonus

Nesting Exceptions

- You can also nest try blocks with catches

```
try {
    try {
        throw
    } catch () { }
} catch () { }
catch () { }
```

* → If you use keyword throw; without any type to throw the exception to the outer block to handle

Creating Custom, User defined exception Class episode 100

```
#include <iostream>
#include <exception>
using namespace std;

class Overspeed : public exception {
int speed;
public :
    const char* what() {
     return "check your speed";
     }
    void getspeed() {
       cout << "your speed is" << speed << endl; }
    void setspeed (int speed) {
       this->speed = speed;
    }
};
class car {
int speed;
public :
    car() {
      speed = 0;
      cout << "speed is zero" << endl;
    }
    void accelerate () {
     for (;;) {
        speed += 10;
        cout << "speed is " << speed << endl;
        if (speed <= 250) {
           Overspeed s;
           s.setspeed (speed);
           throw s;
```

92

```
};
```

Creating Custom Exception Class (cont.)

```
int main() {
   Car nissan;
   try {
      nissan.accelerate();
   } catch (Overspeed s) {
      cout << s.what() << endl;
      s.setspeed();
      }
      return 0;
}
// prints out 0 to 240 by 10's and then error at 250 mph
```

How to overload new and delete operators

→ C++ does not have any garbage collection like others

```
#include <iostream>
#include <cstdlib>
#include <stdexcept>
class Student {
string name;
int age;
public :
    Student () {
        name = "noname";
        age = 0; }
    Student ( string name , int age ) {
        this -> name = name
        this -> age = age }
    void whoAreYou () { "Hi my name is" << name << "I am" << age << endl; }
    void *operator new (size_t size) {
        void *pointer;
        cout << "overloaded new" << endl;
        pointer = malloc (size);
        if ( ! pointer ) {
            bad_alloc ba;
            throw ba; }
        return pointer;
    }
    void operator delete (void *pointer) {
        cout << "overloaded delete" << endl;
        free (pointer)
    }
```

keyword type

keyword - dynamic memory allocation

keyword

// delete dynamic memory allocation

```
int main() {
  Student *ericpointer;
  try {
    ericpointer = new Student ("eric", 26);
    ericpointer -> whoAreYou();
    delete ericpointer;
    } catch (bad_alloc b) {
        cout << b.what() << endl;
    }
```

Dynamic memory Allocation

↳ New and delete vs malloc and free

→ C++ can use all but C only uses malloc and free

The most relevant difference is that the new operator allocates memory then calls the constructor, and delete calls the destructor then deallocates the memory. new calls the ctor of the object, delete calls the dtor. malloc & free just allocate and release raw memory. malloc needs specific size values for allocation as well.

Overloading stream insertion, Extraction Operators << >>

```
#include <iostream>                    // overloading stream insertion operators
#include <string>                      // to allow for input / output to a class
using namespace std;

class Person {
String name;
int age;
public:
    Person () {
      name = "noname";
      age = 0; }
    friend ostream &operator << (ostream &output, Person &p);
    friend istream &operator >> (istream &input, Person &p);
};
ostream &operator << (ostream &output, Person &p) {
    output << "Hey " << endl;
    output << p.name << p.age << endl;
    return output; }
istream &operator >> (istream &input, Person &p) {
   input >> p.name >> p.age;
   return input; }
int main () {
   cout << "enter the name and age " << endl;
   Person eric;
   cin>> eric;                         // Output
   cout << eric;                       // enter name and age
   return 0;                           // ↳type : eric
}                                      // ↳type : 26
                                       // Hey
                                       // eric 26
```

Copy Constructor

define: It is a overloaded constructor. Used when you want to create a copy of a class and stop memory loc sincs. Called whenever we try to copy a class. If you use pointers, address locations fallin with copy.

Problem example:

```
int main() {
  Person eric ("eric", 26);
  eric.introduce(); // prints eric, 26
  Person duplicateeric = eric;
  duplicateeric.introduce(); //prints eric 26
  eric.changeNameandAge("Tim", 20);
  eric.introduce(); //prints Tim 20
  duplicateeric.introduce(); //prints eric 26 -> DID NOT UPDATE AT ALL!! GOOD!
  duplicateeric.introduce(); //prints Tim 20 if name and age are dynamically allocated BAD!
// If you use pointers for name and age instead then the memory location
// is copied and the values stay up to date? Dynamic memory allocation
// caused this issue and now copied object can change indirectly and you
// lost control of it.
```

Copy Constructor

```
class Person {
  string *name;
  int age;
  public:
    Person (string name, int age) {
      this->name = new string (name);
      this->age = age; }
    Person (const Person &p) {        // Copy constructor
      name = new string (*p.name);
      age = p.age;
    void changeNameAge (string name, int age) {
      *(this->name) = name;
      this->age = age; }
    void Introduce () { cout << name << age << endl; }
};
int main () {
  Person eric ("eric", 26);
  eric.introduce();  // print eric 26
  Person duplicateeric = eric;
  duplicateeric.introduce();  // print eric 26
  eric.changenameage ("Tim", 20);
  eric.introduce();  // print tim 20
  duplicateeric.introduce();  // print eric 26 -> it stayed
  return 0;                   // but no copy constructor it prints Tim 26
}
```

stayed as it should

name was dynamically allocated so it followed along!

C++ IO stream Introduction

File handling, printing, display

Text stream and Binary stream

headers #includes

<iostream> - input/output such as cout

<fstream> - file handling

<sstream> - string stream

Build in streams - cin, cout, cerror, clog (buffer version of error)

Set and unset format Flags of IO streams

```
#include <iostream>

using namespace std;

int main() {
  cout.setf(ios::showpos); //turn on show positive/negitives
  cout << 100 << endl; //prints +100
  // there are many format flags
  // How to set more than one flag
  cout.setf(ios::showpos | ios::showpoint);
  // How to unset a flag
  cout.unsetf(ios::showpos);
  // Another method to setting many flags
  auto flag = ios::showpos | ios::showbase; // auto and flag are keywords
  cout.flags(flag)
```

Boolean Values

```
bool boolvalue;
cin.setf(ios::boolalpha); // you can type true / false
cout.setf(ios::boolalpha); // prints true / false
```

Precision fill width parameters – IO stream

int main() {

cout.precision(4); // sets cout to 4 values of length

cout << 123.456 << endl; //prints 123.5 ← notice rounding?

cout << 123456 << endl; //prints 123456 → no effect on non floating point values

cout.width(10);

cout << "Hi" << endl; // prints _____ Hi
8 spaces

cout.fill(*); //will fill spaces with * char

IO manipulators

- endl; is a manipulator for example

cout << hex << 100 << endl; // prints 64 (0x64)

cout << setw(10) << setfill('*') << "erc" << endl;

// only have effect on one line, do not set like previous example

Writing your own manipulator function

learningcad youtube # 109

String Class methods and more

```
#include <iostream>
using namespace std;
int main() {
  string data ("Eric");
  cout << data << endl;
  cout << data.append ("rocks") << data.size << endl;

  // there are many methods associated with std::string
  // such as append, size, insert, substr, erase, ---
```

getline function

```
int main() {
  string data;
  cin >> data; // input Hello its Eric
  cout << data; // only outputs Hello
// instead use getline
  getline (cin, data); //use this instead of cin
```

File Handling – opening

fstream – file stream read/write

ifstream – input stream – read only

ofstream – output stream – write only

```
#include <fstream>   // file handling
#include <iostream>
using namespace std;
int main() {
  fstream file;
  file.open("eric.txt", ios::in | ios::out);  //open with read/write
```

add ios::trunc
↳ will create a new file if it doesn't exist
→ if it exists at the name, contents is deleted

//default of fstream is this

```
  if (!file.is_open()) {
      cout << "error while opening the file";
  } else {
  // read and write
   file.close();
  }
```

File handling – writing to a file

```
int main() {
  ofstream file ("eric.txt");   //open only to write to
  if (!file.is-open()) {
      cout << "file error" << endl;
  } else {
  file << "Hi, I am Eric" << endl;   // How to write to the open file
  file << "I am awesome" << endl;
  file.close();
  cout << "file written to" << endl;
  }
  return 0;
}
```

File Handling – Reading from files

```
int main() {
  ifstream file;
  file.open ("eric.txt");
  if (!file.is_open()) {
      cout << "file error" << endl;
  } else {
    cout << "Reading file" << endl;
    string line;
    while (file.good()) {      // Read each line and print it out
      getline (file, line);
      cout << line << endl;
    }
  }
  return 0;
}
```

keyword

File Handling – appending to a file

```
int main() {
  fstream file ("eric.txt", ios::in | ios::out | ios::app);
  if (!file.is_open()) { cout << "error opening file" << endl; }
  else { cout << "file open" << endl; }
  file << "learning rocks" << endl;
  file.seekg (0); // navigate to line 0
  string line;
  while (file.good()) {
    getline (file, line);
    cout << line << endl;
```

read write append

File Handling - extras

File position indicators Get / Put

Get

```
ifstream file ("error.txt", ios::in);
if(!file.is_open) {
    cout<<"error";
else
  cout<< file.tellg() <<endl;  // find location
```

Put

```
ofstream file ("error.txt", ios::out);
 // file error check
 else
  cout<< file.tellp() <<endl;
  file.seekp(5); // go to position 5
  file << "hello world" << endl;
```

Binary Files

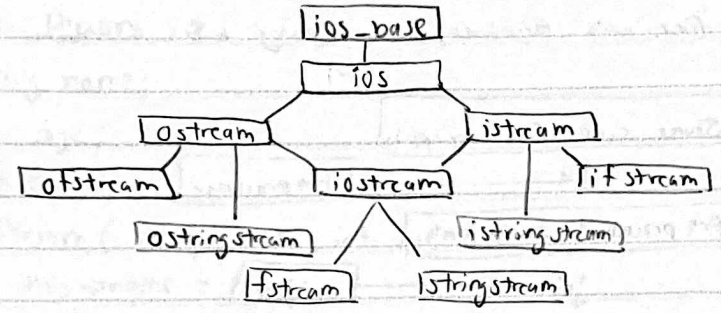
```
char input [100];
strcpy (input, "learning rocks");
fstream file ("error.bin", ios::binary | ios::in | ios::out | ios::trunc);
if (!file.is_open() { //print error }
else {
  int length = strlen (input);
  for (int counter = 0; counter <= input; counter ++) { file.put (input [counter]);
}
file.seekg (0)
char ch;
while (file.good ()) {
  file.get (ch);
  cout<< ch;
```

Abs

Binary Files — Read(), write() methods episode 118 learning lad

String Stream

stringstream allows you to treat a string option like a stream and use all stream functions and operators on it

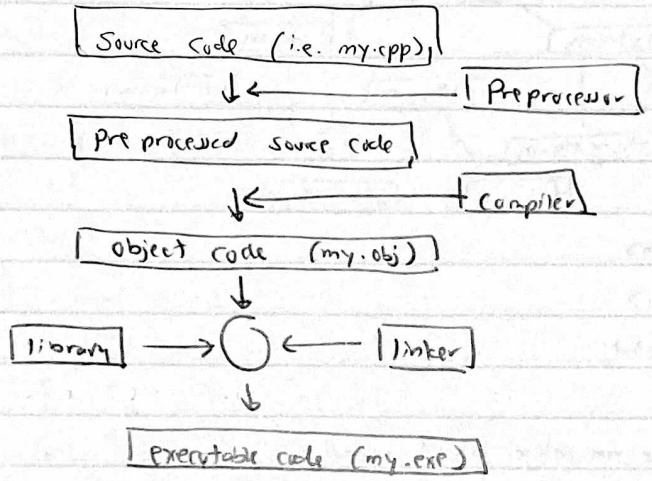



```
#include <iostream>
#include <sstream>
using namespace std;
int main () {
  cout << "Enter an integer value" << endl;
  int input;
  cin >> input;
  stringstream sso;
  sso << input;
  string strinput;
  sso >> strinput;
```

```
int main () {
  string number = "44";
  stringstream sso;
  sso << number;
  int input;
  sso >> input;
  cout << input << endl;
```

Preprocessor Directive #define

pre processor – The C pre processor modifies a source file before handing it off to the compiler, allowing conditional compilation with #ifdef, defining constants with #define, including header files with #include, and using built in macros such as __FILE__



For example #define PI 3.14159 // replaces "Pi" with 3.14159 before compiling
name replacement // object type macro

```
#include <iostream>
#define COUNTRY "USA"
using namespace std;
int main() {
  cout << "i am living in the country << COUNTRY << endl;
  return 0;
}
// additional
#define ONE 1
#define TWO ONE + ONE
#define THREE TWO + ONE

cout << ONE << TWO << THREE

#define WELCOME "Hello everyone \
welcome to the c++ notes"
```

used to continue line

Make sure to always use all CAPS to define a # define

Pre Processor Directives #include

include a header file or other source files with #include

- file named person-class.cpp

```
class Person {
 string name;
 int age;
 public:
   Person ( string name , int age ) {
     this->name = name;
     this-> age = age;
   }
   void display () {
     cout<< "I am" << name << "and I am " << age << "years old"<< endl;
   }
};
```

- file main.cpp

```
#include <iostream>
#include "person-class.cpp" // file include
using namespace std; // move above file because file needs std or will cause errors
int main() {
  Person eric ("eric ", 26);
  eric .display ();
  return 0;
}
```

☆ Very simple -> finds file and pastes it into this file when compiling obj file!

< > - relative reference

" " - everything else can even use instead of < >

#include <iostream> // C++ standard library

#include <stdlib.h> // C standard library

C++ Function like macros

```
#include <iostream>
// macro that takes number and makes sure it is always positive
#define ABS (number) (number) < 0 ? -(number) : (number)
```

if — then execute — else

```
using namespace std;
int main () {
   cout << ABS (-2); // displays 2
   cout << ABS (4); // displays 4
   return 0;
}
```

Conditional Compilation macros #if, #endif, #elif, #else

```
#include <iostream>
#define WINDOWS 1
#define MAC 2
#define LINUX 3
#define OS LINUX
using namespace std;
int main () {
#if OS == WINDOWS          // will only compile based on conditional
   cout << "windows" << endl;
#elif OS = MAC
   cout << "mac" << endl;
#else
   cout << "linux" << endl;
#endif
   cout << "hello world" << endl;   // will then compile remaining code
   return 0;
}
```

Conditional Compilation macros - ifdef and ifndef

ifdef - if defined

ifndef - if not defined

```
#include <iostream>
#define ERIC 1
Using namespace std;
int main () {
 #ifdef ERIC
     cout<< "Hey eric " << endl;
 #else
     cout << " Hey" << endl;
 #endif

 #ifndef SAM
     cout << "No sam here" << endl;
```

Pre processor directive #undef

- used to undefine a macro that has been #define

```
#define ERIC 1
int main () {
  #undef ERIC
  #ifndef ERIC
     cout<< "not defined" << endl;
  #endif
  return 0;
}
```

Predefined Macros

__LINE__	integer line number in the current source code
__FILE__	name of current source file
__DATE__	the compile date of the current source file mm:dd:yy
__TIME__	time of translation
__STDC__	defined as 1 when compiled as C
__cplusplus	

```
6  int main () {
7      cout << __LINE__ << endl;   // prints 7 cus its on line 7
8      cout << __FILE__ << endl;   // full file extension
9      cout << __DATE__ << endl;   // current date at compile
10     cout << __TIME__ << endl;   // time when compiled
11     cout << __STDC__ << endl;   // return 1 if standard C code
12     cout << __cplusplus << endl;   // conforms to standard C++ -> 6 digit number if not = less digits
13     #line 1000   // moves to line 1000
14     cout << __LINE__ << endl;   // prints 1000
```

Preprocessor directive

#pragma once

↳ make sure this is only included once in a file

↳ used for header files

↳ #ifndef is used also for the same thing

Generic Programming and Templates

what if you have the below methods

```
int max (int x , int y ) {
   return (x>y )? x : y ;  // returns greatest value of two values
}
float max ( float x , float y ) {  // overloaded to take different data type
   return (x>y )? x : y ;
}
```

You can use templates to reduce this down to one function!

```
template < class myType >
```

OR

```
template <typename identifier >
```

<u>example</u>:

place holder for data type

```
template <typename myType > void display ( myType x )
{
   cout << "you have passed" << x << endl;
}
int main () {
   display (10) ;
   display (1.234) ;
   display ("Eric rocks") ;
```

Basically you add the template as the return type and use the identifier as the type it takes

Templates

Multiple Parameters and Return Values

```
#include <iostream>
using namespace std;

template < typename T > void display (T, T);
template < class T > T maxi (T x, T y);
int main () {
  cout << maxi (20, 30) << endl;   //prints 30
  cout << maxi (20.256, 3.897) << endl;   //prints 20.256

  display (20, 30)   //prints 20 and 30          // HAS TO BE OF SAME TYPE!
  display ("anil", "anjali");   // prints anil and anjali
  display (12.36, 56.78);   //prints 12.36 and 56.78

}
template < typename T > void display (T x, T y) {
  cout << x << "and" << y << endl;   //
}
template < class T > T maxi (T x, T y) {
  return (x >= y) ? x : y;
}
```

Passing Standard Parameters to Generic Functions

```
#include <iostream>

using namespace std;

template < typename T > void display (T x ,int y);
int main () {
  display ("eric moser", 8); // prints "eric moser" 8 times
  return 0;
}

template < typename T > void display (T x , int y) {
  for (int counter = 1; counter <= y; counter ++ ) {
      cout << x << endl; } }
```

can pass any type of data

only handles an int value

Generic Functions with multiple Generic Types

```
template < typename T1 , typename T2 > void display (T1 x , T2 y);
int main () {
  display ("eric", 26);
  display (10 , 12);
  display (10.1 , 12.2);
  display ("eric", 26.5);
  return 0;
}
template < typename T1 , typename T2 > void display (T1 x , T2 y);
{
 cout << x << "and" << y << endl;
}
```

Explicitly overloading Generic Functions with Normal Functions

```
template < typename T > void whatYouGot (T x);
void whatYouGot (int x);
int main () {
   whatYouGot (23.4); // uses generic because float
   whatYouGot (22);  // uses normal
   whatYouGot ("eric"); // uses generic
   return 0;
}

template < typename T> void whatYouGot (T x) {
   cout << "generic function" << endl;
   cout << x << endl;
}

void whatYouGot (int x) {
   cout << "normal function" << endl;
   cout << x << endl;
```

Overloading Generic Function template

You can overload the Generic Function to handle different amounts of parameters.

```
template < typename T > void whatyouGot (T x ) {
   cout << "generic function" << endl;
   cout << x << endl;
}

template < typename T > void whatyouGot (T x , T y ) {
   cout << "overloaded function" << endl;
   cout << x << y << endl;
}

// Calls could be
whatyouGot (55)
   "        (55.5)
   "        ("eric", 25) // can handle two values
   and so on
```

Generic Class

```
template < class MYTYPE > class MYCLASS {
  MYTYPE p1;
  MYTYPE p2;
  public:
    MYCLASS (MYTYPE x, MYTYPE y) {
      p1 = x;
      p2 = y;
    }
    void whatYouGot () {
      cout << "You have " << "p1" << p1 << "p2" << p2 << endl;

int main () {
  MYCLASS <int> intObject (22,55);
  MYCLASS <string> stringObj ("eric", "eorn");

  intObject . whatYouGot();
  stringObj . whatYou Got();

  return 0;
}
```

Generic class with more than one Generic Type

```
template < class Type1 , class Type2 > class myclass {
 Type1 p1;
 Type2 p2;
 int counter; 100;
 public:
   myclass (Type1 x, Type2 y) {
    p1 = x;
    p2 = y;
    counter = 100; }
   void whatyougot () { cout << p1 << p2 << counter << endl; }
int main() {
  myclass <int, string> obj1 (26, "Erz");
  myclass <Float, Float> obj2 (26.26, 66.24);
  obj1. whatyougot();
  obj2. whatyougot();
  return 0; }
```

Default parameters as datatypes Generic Class

```
template < class Type1 = string , class Type2 = int > class myclass {
 Type1 p1; Type2 p2;
 public:
   myclass (Type1 x, Type2 y) { p1 = x; p2 = y; }
   void whatyougot () { cout << p1 << p2 << endl; }
}
int main() {
 myclass < > obj1 ("anil", 24); // using the defaults
 myclass <Float> obj2 (22.22, 55); // overrides first string to float
 myclass <int, string> obj3 ( 10, "Erz" ); // overrides both

 //calls for whatyougot func
 return 0;
}
```

Defaults are set

Explicit Specialisation of Generic Class

```
template < class Type > class myClass {
 Type1 p1;
 public:
   MyClass (Type p) { p1 = p; }
   void whatyouGot () { cout << p1 << endl; }
};
template < > class MyClass <int> {
 int p1;
 public:
   MyClass (int p) { p1 = p; }
   void whatyouGot () { cout << p1 << endl; }
// Calls
 MyClass <string> ob1 ("exe");
 MyClass <int> ob2 ( 10 );
```

Namespaces

A namespace is a declaration region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur expecilly when your code base includes multiple libraries.

for example: you might take in two files and both have a method of the same name.

file 1:

```
 namespace ONE {
   Display () { cout << "Hi from One" << endl; }  }
```

file 2:

```
 namespace TWO {
   Display () { cout << "Hi from two" << endl; }  }
```

Main file:

```
#include "file one"
#include "file two"
int main ()
{
  ONE :: display ();
  TWO :: display ();
  return 0;
  //using namespace ONE; -> would get namespace
}
```

pick namespaces

Namespaces - Nested

```
namespace one {
  int i;
}

namespace two {     // can define multiple namespaces
  int j;
}

namespace one {     // can add more to a namespace
  int k;
}

int main() {
   one :: i = 10;
   two :: j = 20;
   one :: k = 30;
   using namespace one;
    cout << i << j << k;
   return 0; }
```

```
// nesting
namespace one {
   int i;
   namespace two {
     int j;
}}
int main() {
   one :: two :: j = 10;
   one :: i = 15;
   return 0;
}
```

Namespaces - Anonymous

→ a namespace without a name is anonymous

```
namespace {
  int x;
  display () {
    cout << x << endl;
  }
}
int main () {
  x = 25;      // do not need to specify the namespace name first
  display ();
```

→ Does not carry through files like the named one will. local to a particular file.

Namespaces - nested anonymous

```
namespace One {
  int x;
  namespace {
    void display () { cout << x << endl; }
    namespace {
      int y;
    }
  }
}
```

Namespace - Aliases - giving a new name to a namespace

```
namespace verylargespace {
  int x;
  namespace nestedspace {
    int y; }}
namespace One = verylargespace;   // you can rename the space
int main () {
  One :: x = 100;
  One :: nestedspace :: y = 50;
  return 0;
}
```

20

Namespaces – inline nested

→ defined in new C++ 11 standard.

```
namespace AppVersion {
  namespace version1 {
    void display() { cout << "one" << endl; }
  }
  namespace version2 {
    void display() { cout << "two" << endl; }
  }
  inline namespace version3 {
    void display() { cout << "three" << endl; }
  }
}
int main() {
  AppVersion :: display(); // calls version 3
  AppVersion :: version2 :: display();
```

→ used here to pull only the latest version of the code and you can still access older versions with a longer call.

→ if you add more to version3 later, the inline keyword is not needed

Defining Classes in separate files

→ used for very large code bases

Sources

↳ main.cpp

↳ Person.cpp

Headers

↳ Person.hpp

Person.hpp

```
#ifndef PERSON_H        // if not included yet
                        // used to make sure that the class is only included once
#define PERSON_H        // include it
class Person {
  public:
      Person ()
      void whatyou got ();   // only prototype
  protected:
  private:
};
```

Person.cpp

```
#include "Person.hpp"    #include <iostream>    Using namespace std;
Person :: Person ()
{
  cout << "Person constructor" << endl;
}
Person :: whatyougot ()
{
  cout << "Hello" << endl;
}
```

Defining a class in separate files

main.cpp

```
#include <iostream>
#include "Person.hpp"   // only headers need to be included here
using namespace std;

int main () {
    Person eric;
    eric.whatyourgot();
    return 0;
}
```