
C++

C++

Pre-process

Everything that starts # is a process statement.

The first thing a compiler does !

include < ... >
Find a file and just paste it.

int main ( ) { ... }
Entry point !

<<
Operators are just functions ! !

Basic compilation process

Headers	Headers are just included, not compiled
Cpp files	Cpp are compiled per file
Obj files	Multiple cpp are turned into obj files after compilation
exe	Obj files are stitched together by The linker into a exe

linker

Declaration
```
void Log(const char* msg)
```

Definition
```
void Log(const char* msg)
{
    //Actual Code
}
```

The Linker will found the definitions of every declaration in all the files.

compiler

Files don't mean anything (unlike Java)
Cpp files creates a single translation unit and obj file.

stages

Pre-process (#)
- include : File to include, reads file and copy pastes it to current file
- define : looks for a word and replaces for whatever follows
- if/endif : Include or exclude code based on condition

Obj File
Exports the instructions (asm and binary) for the CPU to run

static
Function is only declared for translation unit (not visible to other obj files)

inline
literally copies code body inside call

Linker : Obj files thrown together

Variables

int	32 bits	1 signed	31 bits (2^{31})
unsigned int	32 bits	(2^{32})	Goes for any type
char	1 byte	(character)	
short	2 bytes		
long	4 bytes		
long long	8 bytes		
float	4 bytes	Syntax:	3.3f
double	8 bytes		
bool	1 byte	boolean	

↑ We cannot address single bits, only byte

sizeof ( ) — size on bytes

type *
type &

functions

return_type name(param) {
return return_type name_return
}

header files

use for declarations and include on cpp
match funcs signatures

pragma once → only include file once per translation unit

For includes

include " " From current folder (work for everything)
< > From include directories

Cpp standard libraries usually have no end extension

debug

breakpoints ! Click on sidebar

→ conditions

if — The entire code is loaded, but the tree hierarchy jumps to certain code

operators — Compare memory functions (go to disassembly)

really fucking cool !

if (cvar) → is this value not 0

In memory 0 is false and 1* is true. (4 bytes)

boolean is just 9 bytes either all 0 (false) or not (true)

null ≡ 0

if (ptr) → (is not zero or null)

So evaluations or conditions are just functions that return a boolean (in memory)

else if

check if 1st fails

```
else {
    if (<eval>) {
    }
}
```

Loops

```
for (int i = 0 ; i < 5 ; i++) { ... }
```

declare · check if true · execute after body · body

Can contain anything ?

```
while (i < 5) { ... }
```

condition · body

```
do { ... } while (<cond>) { ... }
```

body

Runs at least once !

Control Flow

Works with all loops.

continue — Skip to next iteration

break — Get out of the loop

return — Get out of the function providing a value.

[NULL is just a #define null 0]

Pointers

Integer that stores a memory address.

```
int var = 3;
```

`var` — variable

```
void* ptr = &var;
```

`void*` — referencing; pointer no type

`&var` — get memory address from variable

Pointer types just define size, but ultimately are completely meaningless.

```
*ptr = 10;
```

*ptr — dereferencing; Set the value

```
ptr = 10
```

ptr — Sets the address

Create an address with 8 bytes of memory:

```
char* buffer = new char[8];
```

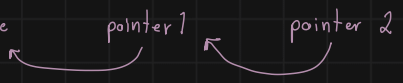
char* — We know char is 1 byte; This will reference the beginning of that block

new char[8] — Create 8 chars (8 bytes)

```
memset(buffer, 0, 8);
```

Fill block with 0

Pointer are variables that store a memory address, therefore you can have double or triple pointers.




```
char** pter = &buffer;
```

pter — stores an int with the memory address of the pointer

References

Sort of creating an alias for a variable.

```
int a = 3;
int& ref = a;
```

Creates a reference, not a new variable

```
void Increment(int& val){ val++; }
```

by using a reference as parameter you modify the actual variable

Syntax sugar for pointers

```
void Increment(int* val){ (*val)++; }
Increment(&a);
```

Classes

```
class Player {                Class is a type
public:                   } Private by default. Use public to external access
    int x,y;
    nt speed;

    void Move(int x, int y){...}  } Method  (Functions with hidden parameter sort of)
private:
    int m_xp;
};
Player player;            } Instance
```

Struct vs Class

Class private by default
Struct public by default

Structs exist for backwards compatability.
Use for only manipulating variables (no inheritance)

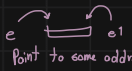
```
struct Player { ... }
```

How to write a class

m_variable } For private variables

Static

Inside a class:
Share memory with all instances of the class.



Static methods can only call static variables (can't access instance)

normal method: `int Sum(Player P, int x, int g){ }`

static method: `static int Sum(int x, int y){ }`

Outside of a class:
Internal
Only visible by a single translation unit.

Static Local

Static inside a class has similar behavior as static inside a function. (i.e. the var will be shared by all function calls)
You may use this inside a singleton class to return a single shared instance

eNum (enumorate)

Give name to value. Set of values. Works as a type (restriction). Not a namespace

```
enum Ex : unsigned char {      Type (Has to be integer) [Default 32 bits]
    A, B, C                } 0, 1, 2 by default  but you can A=9, B=10, C=0
}
Ex value = A;
```

You have to choose between one of them
~~Ex value2 = 5;~~
Incorrect !

Constructors

Special method that runs on every instance creation. (Inizialization)

```
class Entity {
    float x, y;
    Entity() {
        ...
    }
}
```

`float x, y;` } Declared but uninitialized memory

`Entity() { ... }` } Constructor : • No return • Has to be named as the class

Primitive types are not initialized ?
C++ has a default constructor

Destructor

Special method that runs on every instance deletion.

```
class Entity {
→   ~Entity() {
        ...
    }
}
```

`~Entity() { ... }` } Destructor

Inheritance

Define general classes (parent) that will share functionality with othe entities (child)

```
class Player : public Entity {
    ...
}
```

} Player extends from Entity

Virtual Functions

Methods that can be overwritten.

Methods declared normally will run method of type unless virtual functions.

```
Class Entity {
 public:
→    virtual std::string GetName()
        { ... }
 };
 class Player : public Entity {
  public:
      std::string GetName() override
          { ... }
 };
```

`virtual std::string GetName() { ... }` } This function can now be modified if inherited

`override` } Override keyword on modified method

Runtime costs :
- V Table loaded on memory
- Search function on V Table

Pure Virtual Functions (interfaces)

Define a function in base class without implementation and force sub classes to implement it.

```
class Entity {
public:
    virtual std::string GetName() = 0;
};
```

`= 0;` makes it pure

Cannot be instanciated (≈ template)

Visibility

Who can see / call methods.

Doesn't affect performance.

Class (private) Struct (public)

```
private:
    int X = 3;
```

Only parent class can access it.
(not even sub classes, only friends)

```
protected:
    int X = 6;
```

Parent class and other sub classes can access it

```
public:
    int X = 9;
```

Access for all.

Arrays

Collection of data

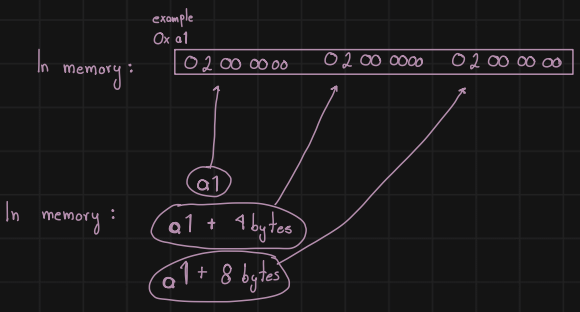
```
Certain type
int example[3];
```

Define array number

```
example[1];
```

index

In memory:



An array is really just a pointer

```
[*(int*)((char*)ptr + 8) = 3;]
```

Turns the a1+8 to value 3

Stack (↓ Destroys itself at the end) vs Heap (↳ with new, has to be destroyed) allocated arrays

String

Array of characters

char is a single byte

- **Raw string**

```
const   char*   name = "Halle";
```

const: strings are immutable (allocated memory)

char*: char pointer

In memory:

name 0x02	48	61	6C	6C	65	00
						↑ null termination char

Strings begin at memory address and end when a 0 is hit.

[Single quotes : Characters ' ' | Double quotes : char* " "]

- Should use **std::string** : Just an array of char with some built in functions

Standard string

```
#include <string>
std::string name = "Halle";

name.size();
bool contains = name.find("no") != std::string::npos;
```

} Include header file

} String is a proper class with methods.

String Literalls

"Isaac"; } 6 chars On read only memory. const char* name = "Isaac"

```
const wchar_t*    name1 = L"Isaac";
const char16_t*   name2 = u"Isaac";
const char32_t*   name3 = U"Isaac";
```

} Different memory sizes

```
const char* multi = R"line 1
line 2
line 3"
```

} Ignores Escape chars

Const

Promise that the variable won't change (you could break it)

```
const int c = 2;
```

```
const int * a = new int;
*a = 2;
a = 0x01;
```

Define constant pointer
You cannot change its value
You may change where it is pointing

```
int * const a = new int;
*a = 2;
a = 0x01;
```

Constant pointe
Can change value
Cannot change where is pointing

```
const int * const a = new int;
*a = 2;
a = 0x01;
```

Const const pointer
None

Read only method (only works on class)

```
class ...
private:
    int m_X;
public:
    int GetX() const {
        m_X = 1;
        return m_X;
    }
```

Cannot modify any data inside this only return

Unless mutable

```
mutable int b = 3;
```

You can now modify from a const method (inside a class)

```
int getX() const {
    b = 9;
}
```

Mutable

Inside a class

```
class Entity {
    int mutable x = 0;
    int y = 0;

    void GetXy() const {
        x++;
        y++;
    }
};
```

You can modify a variable inside a constant method

Inside a lambda

You can modify a var passed by value

Little throwaway func (like matlab)

```
int x = 3;
auto f = [=] () mutable {
    x++
};
//x = 3;
```

value

Lambda by value (copy)

Not modified

sort of weird
not that useful

Member Initializer Lists

Initialize class members in constructor

When initializing in constructor

```
class Entity () {
private:
    int m_X, m_Y;

public:
    Entity () {
        m_X = 0;
        m_Y = 0;
    }
    Entity (int x, int y) {
        m_X = x;
        m_Y = y;
    }
};
```

We want to initialize this vars

This initializes the "default" values

This initializes with given values

Better Way

```
class Entity () {
private:
    int m_X, m_Y;

public:
    Entity ()
        : m_X(0), m_Y(0)
    { // Code }

    Entity (int x, int y)
        : m_X(x), m_Y(y)
    { // code }
};
```

Remember that if a class is instanciated inside another class and then in the construtor, it will be called twice !
That's why it has to be initializated like this

Also remember it has to be in order !

Ternary Operators

Syntactic sugar for ifs when defining a variable (like js)

```
if (level > 5)
    speed = 10;
else
    speed = 0;
```

```
speed = level > 5 ? 10 : 0;
```

condition — true — false

Tecnically faster (return value optimization)

You may nest

```
speed = level > 5 ? level > 10 ? 15 : 10 : 0;
```

Create / Instanciate Objects

Instanciate classes need to occupy memory, there are 2 ways to do that

Where it goes ?

Stack

Automatic lifespan
[dies when out of scope]

```
Entity entity;          } Default constructer
Entity entity ("Hello"); } Special construtor
```

Always try to use this

Heap

Allocate object will always stay there [Not destroyed until you do it]

```
Entity* entity = new Entity ("Hello");
delete entity;
```

} Clean up

Try to avoid except if necessary

New Keyword

Allocate memory in heap. Very important!

It checks the necessary memory (type), looks for available space and returns a pointer.

Quite slow

new is an operator

```
int * b = new int; // 4 bytes
int* a = new int[50]; // 200 bytes
Entity* e = new Entity(); } Also calls the constructor
```

usually new calls malloc()

```
Entity* e = (Entity *)malloc(sizeof(Entity)); } This does NOT call the constructor.
                                                Probably shouldn't do this
```

Remember to use **delete** (also an operator)

```
delete e;    } Call C function free()
delete[] a;  } Call a different delete for arrays.
```

Implicit Conversion and explicit Keyword

↓ Automatic

```
class Entity
...
    Entity(string name, int age)
        : m_Name(name), m_Age(age)
...
void SomeFunction(const Entity& e){}

Entity e = "Isaac";  } Entity is not a string or int but it implicity converts it into an Entity class
Entity e2 = 21;      } because it has a constructor

Also
SomeFunction(22);    } Doesn't take int as parameter but implicitly converts to class instance
```

C++ can only do **1** implicit conversion

[From String to char*, for example]

explict disables implicit conversions

```
...
→ explicit Entity(string name, ...)   } Constructor
...
~~Entity a = 22;~~                    } Now it can't convert
```

Operator and Operator Overloading

↓

Symbol to represent function (+, *, +=, new, ...)

Overloading: Change behaviour of an operator

Operators are just functions

Example

We may overload the + operator to make it add vectors

```
struct Vector2 {
    int x, y;
    ...
    Vector2 Add()
    ...
}
```

```
vector2 pos(2, 2);
vector2 speed(3,3);
vector2 res = pos.Add(speed);
```

Instead, we overload →

```
struct Vector2 {
    int x, y;
    ...
    Vector2 operator+(...) const
    {
        ...
    }
}
```

Now, simply

```
Vector2 res = pos + speed
```

The same thing

```
vector2 res = pos.Add(speed);
```

=

```
Vector2 res = pos + speed
```

Some operators:

```
bool operator==()
operator*()
operator<<
```

The "this" Keyword

Pointer to current object instance (only accesible from a member function)

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

Same thing: Saving x to object instance .x

If inside a method we get

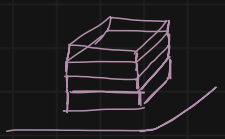
```
Entity* e = this;
```

If inside a const method we get

```
const Entity* e = this;
```

Object Lifetime

A variable on the stack will cease to exist when it goes out of scope.



```
{
    Entity e;   ← Created
}               ← Destroyed
```

```
int* CreateArray(){
    int array[50];
    return array;
}
```

Doesn't work (memory in stack is cleared)

This is the base for scoped-smart pointers or timers

```
class scopedP(){
    scopedP(Entity* e)
        : m_E(e) {}
    ~scopedP(){
        delete(m_E);
    }
```

Deletes automatically when out of scope

Smart Pointers

Automate new - delete (wrap up)

3 basic types

- Unique Pointer

  Cannot make copies of it, since it gets destroyed when out of scope.

  Very useful and no overhead.

```
std::unique_ptr<Entity> entity = std::make_unique<Entity>();
                        ^var name
```

We use this so that if the constructor returns nothing we don't get a random dangling pointer

Then simply:

```
entity->SomeMethod();
```

Cannot copy:

~~`std::unique_ptr<Entity> e1 = entity;`~~

Because you would have 2 pointers to the same location, if one gets destroyed, the others would point to nothing.

- Shared Pointer

  Uses reference counting (counts everytime it is referenced [called])

  [If called twice, count = 2, if first out of scope count = 1
  if both out of scope count = 0 and it gets deleted]

```
std::shared_ptr<Entity> sharedEntity = std::make_shared<Entity>();
                        ^var name
```

You always call make_shared, since it has to make a control block (some memory to store reference count)

CAN copy:

```
std::shared_ptr<Entity> e1 = sharedEntity;
```

✓

When all references are gone, it frees the memory

- Weak Pointer

  Same as a shared pointer, but it doesn't increase the reference count,

  (Which means it may or may not point to something, since it doesn't keep the shared pointer alive

```
std::weak_ptr<Entity> e2 = sharedEntity;
```

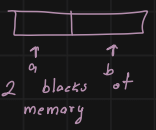
Some copy, but doesn't ++ the ref count.

You can ask if it is still alive or not.

Copying and Copy Constructors

```
int a = 6;
int b = 9;
b = 2;
```

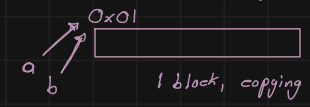
} Literal copy




```
Vector2* vA = new Vector2();
Vector2* vB = vA;
```

} This is copying the pointer

ie. the num where the memory block is



When copying a class, it will make a shallow copy of the internal variables.

which means if you have a pointer, the copy will have the same exact pointer.

We have to perform a deep copy with a

copy constructor

constructor that takes in a const reference of the same type

```
class Entity ...
→ Entity(const Entity& other){
  ...
}
```

If you don't want it to be able to copy:

```
Entity(const Entity& other) = delete;
```

Runs when

Copy

```
Entity e = Entity();
Entity e1 = e;
```

You always pass variables by const reference

```
void WhateverFunc(const Entity& e){
  Entity copyOfEntity = e;
  ...
}
```

} If you want to make a copy, you can

If you don't pass as reference, you are copying needlessly every time the function gets called.

Arrow Operator

Dereferences and calls a method

```
(*classPtr).someFunc();
classPtr->someFunc();
classPtr->x = 3;
```

} Shortcut, they do the same thing

You can also overload them (to make it function in your own classes)

Use to find offset of internal variables

```
struct Vector3 = { float x,y,z; }
int offset = (int)&((Vector3*)nullptr)->x;
```

Dynamic Arrays

Array that can resize.

```
#include <vector>
std::vector<type> name;
```

type: type of array item; name: var name

} You generally want to allocate objects (<classObj>) instead of pointers (<classPtr>)

```
name.push_back(...);
```

} Add to array

```
name.size();
```

} It has the size method

```
name[0];
```

} Overloaded index operator by default

```
for(classObj& v : name)
cout ... v ...
```

} Iterate each vector

```
name.erase(vertices.begin()+1);
```

} Erases second element

Optimizing std::vector

Avoid copying objects (there are tons of other optimizations)

Set an initial capacity

```
std::vector<Vertex> vertices;
vertices.reserve(3);
```

} Make it initialize with 3 "spaces" in memory

Construct vector directly into vector

```
vertices.emplace_back(1,2,3);
```

} Basically tell the constructor values and it construcs in place

```
vertices.push_back(Vertex(4,5,6));
```

} This creates first the object in the stack and then copies this inside the dynamic array. (Essentially making it twice)

[Or creating and copying - rather than directly creating]

Libraries (Static Linking)

Linking at compile time

Static

Directly linked part into executable (in exe)

Dynamic

Linked at runtime (for example, at application launch) (in dll, not in exe)

- Add files to include path (C/C++ - General - Additional Include Dir - "path")
- Add to program
  #include "path.h"
  Provide declarations (which functions do we have?)
- Include library definitions to the linker (Linker - Input - Additional Dependencies - "name.lib"; Linker - General - Additional Library Dir - "path")
  Provide definitions (what do the functions do?)

Allow optimization because the compiler/linker know it all

Dynamic Libraries

Linking at runtime

When you launch exe, it gets loaded to memory

- Add files to include path (C/C++ - General - Additional Include Dir - "path")
- Add to program
  #include "path.h"
- Include dynamic library definitions to the linker
- Place dll file into the executable directory

Making and working with Libraries

check video again probably

Multiple Return