C++ for video games

•••

Why C++ ?

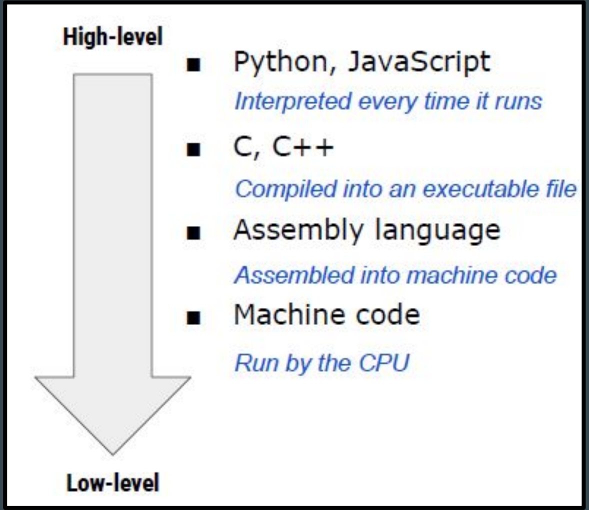
- Games are real time applications ⇒ **High performance**:
  - 3D graphics
  - Interactive
  - Network latency
- Games are **huge**:
  - Millions of lines of code
  - Pushes for good organization
  - QA madness

Why C++ ?

- Games development is **unpredictable**:
  - Code changes during production
  - Cycle of code -> play -> evaluate -> code again ...
- **Portability** between platforms
  - Supporting different platforms means supporting different system APIs for:
    - FileSystem
    - Graphics
    - Sound

Why C++ ?

- C and C++ are considered a middle level languages
- Isn't a low level language:
  - Independent of hardware
  - One statement produces multiple machine instructions
- Isn't a high level language:
  - Direct memory management
  - Faster performance
  - Harder to learn


High-level
Python, JavaScript
Interpreted every time it runs
C, C++
Compiled into an executable file
Assembly language
Assembled into machine code
Machine code
Run by the CPU
Low-level

Why C++ ?

- So, why C or C++
  - A combination of efficiency and language abstraction
  - Portability to platforms (OS API for system access are written in C/C++)
    - Win32, Win64
    - OpenGL, Vulkan, DirectX
- Why not C ?
  - C++ is higher level than C but can be as faster as C
  - C++ can access C APIs

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.”

- Bjarne Stroustrup

C++: What you should know

- **C++** is an object oriented language:
  - **Encapsulation**: private/protected/public/friend
  - **Inheritance**: private/protected/public … and multiple inheritance
  - **Polymorphism**: virtual methods and pure virtual/abstract classes
  - **Overloading**: functions and operators

C++: What you should know

- C++ is a huge language
  - C++ features has **implementation details** you should know.
  - C++ Has **multiple ways** to program the same but not all are efficient.
  - Video games are large and complex software. There are **practical guidelines** to write defensive code and prevent future bugs or inefficient code

C++: Implementation details

- Order of construction and destruction of a class
    - Order of construction is important when using inheritance
    - Virtual destructors are needed to destroy inherited classes when deleting a base class pointer.
    - Virtual table creation moment is undefined. Guide: Don’t call virtual methods in ctors/dtors
- Default methods automatically generated by the compiler
- The static keyword uses and initialization
- Why using ++i is better than i++ in C++ ?

C++ Implementation details

- STL:
    - string
    - list
    - vector
    - map / unordered_map
    - algorithms like sort
- When to use a list or a vector ?
- What is a hash table ?

C++ Implementation details

- **static_cast:** throws a compile error if there is no inheritance relationship between variable type and cast type. Note: The address of the object may change (when using multiple inheritance) for adjusting virtual table of casting type.
- **reinterpret_cast**: no instruction is generated. Just tells the compiler to interpret pointer as being of new type.

```
class A{};
class B : public A{};
class C{};

void func()
{
  B* b = new B;
  A* a1 = static_cast<A*>(b); // compile success
  C* c1 = static_cast<C*>(b); // compile error

  A* a2 = reinterpret_cast<A*>(b); // compile success
  C* c2 = reinterpret_cast<C*>(c); // compile error
}
```

C++ Implementation details

- **dynamic_cast:** for polymorphic types, checks in runtime (using RTTI info) whether the pointer is actually of casting type. If so adapts pointer (address may change), if it does not returns null.
  Note: don’t use it, RTTI is very expensive
- **const_cast :** Used to remove const from a variable.
  Note: avoid using it. Is considered a wrong design.

```
class A {virtual int f() {return 0;}};
class B : public A {virtual int f() {return 1;}};
class C : public A {virtual int f() {return 2;}};

void func() const
{
    A* a = new B;
    B* b = dynamic_cast<B*>(a); // b = valid ptr
    C* c = dynamic_cast<C*>(a); // c = nullptr
}
```

```
void student::func() const
{
  const_cast<student*>(this)->roll = 4
}
```

C++ Implementation details

- Internal and External Linkage in C++
  - Difference between declaration and definition of symbols
    - Functions
    - Variables
  - Classes can be forward declared to be used in function declarations;

```
int f();   // declaration (usually in header)
int f()    // definition (usually in cpp)
{
  return 1;
}
```

```
extern int var; // declaration
int var = 45;   // definition
```

```
int f(MyClass a); // error undeclared MyClass
```

```
class MyClass;

Int f(MyClass a); // compiling ok
```

C++ Implementation details

- Internal and External Linkage in C++
  - Each cpp file in project is treated as a **compilation unit** (CU) and generates a .obj file
  - A symbol that is visible and can be shared between CUs has **external linkage**.
  - Symbols with external linkage must have **unique names** between different CUs

```
// explicit declare external linkage
extern int i;     // defined somewhere
extern int f();   // defined somewhere
```

```
// definitions have default external linkage

int i = 10;

int f()
{
  return 1;
}
```

C++ Implementation details

- Internal and External Linkage in C++
  - A symbol not visible to other CU has **internal linkage**.
  - internal linkage of symbols in headers means they are **duplicated** for each CU.
  - **const** variables has internal linkage
  - **anonymous namespaces** forces internal linkage for functions, variables and classes

```
// in header file → internal linkage

const float pi = 3.14159f;
```

```
// definitions with internal linkage

static int i = 10;

int f(){ return 1;}
```

```
// internal linkage

namespace // anomnymous namespace
{
  int i = 10;
  int f(){ return 1; }

  class A { };
}
```

C++ Practical guidelines

- References are your friends!
- When is better foo(int& i) than foo(int* i) ?
- Const-correctness:
    - When to return a const value ?
    - Make all arguments that you can const
    - Make all methods that you can always const

C++: Practical guidelines

- **Don’t waste time or space**
- Don’t do **extra allocations** if not needed
- Be descriptive, your code should not need comments
- Use descriptive names for classes and variables
- Everything public is very bad design!
- Avoid **multiple inheritance** and avoid **abusing inheritance**

C++: Practical guidelines

- Avoid passing classes **by value** (needs copying whole class). Use **references** instead:
  - Pass const references if object should not be modified.
  - Don’t use references for passing built-in types (int, float, char, long, etc).
  - Do not make built-in types const when passing args, no gain
- Initialize all variables
- Avoid **exception** handling ⇒ Never used in video games (disabled)
- Avoid **RTTI** ⇒ Never used in video games (disabled)

C++: Practical guidelines

- Prefer **forward declaring** classes to include headers ⇒ Saves compiling time
- Use inline for short functions (getters, setters)
  - note: inline has internal linkage
- Don’t leak memory or resources ⇒ Use memory leak detector!!!

C++: Practical guidelines

- Windows memory leak detector using crtdbg and cstdlib

```
#ifdef _DEBUG
#define MYDEBUG_NEW new( _NORMAL_BLOCK, __FILE__, __LINE__)
#endif // _DEBUG

#define _CRTDBG_MAP_ALLOC
#include <cstdlib>
#include <crtdbg.h>

#ifdef _DEBUG
#define new MYDEBUG_NEW
#endif

void DumpLeaks(void)
{
    _CrtDumpMemoryLeaks();  // show leaks with file and line where allocation was made
}

int main(int argc, char ** argv)
{
    atexit(DumpLeaks);
    int* a = new int;
    a = new int; // ouch, leak memory
    delete a;
}
```

C++: Practical guidelines

- Use always **assert** when possible to check preconditions and postconditions of functions. Use better assert than standard c++ assert:
    - Allow you to ignore this same assert forever
    - Have different importance levels for asserts
    - Allows you to turn them off / on depending on the type of build
    - Have messages with the asserts to be more descriptive
    - Capture the problem in your own LOG files
- Either build your own or use something along the lines of SDL assert (read this)

Bibliography

The C programming language (K&R)

The C++ programming language

Pragmatic programmer

Effective C++

Game Programming Patterns

Homework: Vector3 class

Vector3 class with x/y/z components using a template. You should be able to write code like:

```
Vec3 a()

Vec3 b(1,0,1)

Vec3 c(b)

Vec3 d = b + c
```

```
d.Normalize();

d.distance_to(b);

d.dot_product(b);

d.cross_product(b);

d.angle_between(b);
```

LET'S DO THIS!
imgflip.com