Learn You a C++ for Great Good

Michael Bartling

October 13, 2016

University of Texas at Austin
Department of Electrical and Computer Engineering

C++ In a Nutshell

Basic difference between C and C++

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

---

Bjarne Stroustrup
The Creator of C++

3D $y = Mx + b$ in C

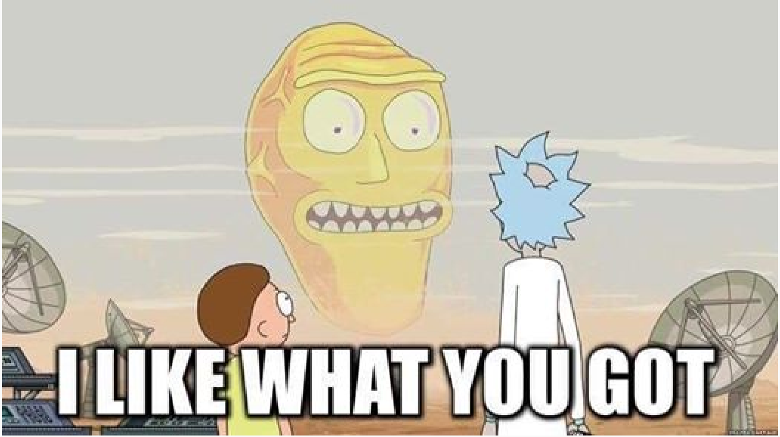
```c
// Add Two 3D Vectors
Vec3d add_vec3d_vec3d(Vec3d, Vec3d);

//Multiply 3x3 Matrix with Vec3d
Vec3d mult_mat3x3_vec3d(Mat3x3, Vec3d);

// Y = Mx+b
Vec3d y = add_vec3d_vec3d(
    mult_mat3x3_vec3d(M, x), b
    );
```

3D $y = Mx + b$ in C++

```cpp
Vec3d y = M*x + b;  //Obvious
```



Cool C++ Features

- New/Delete instead of Malloc/Free
- Function Overloading
- Objects
- More "control" over scope
  - Namespaces
  - Membership Resolution
- Template Metaprogramming
- Runtime Exceptions

Lets Dive in to C++

Function Overloading

In C, function names must be unique.

```c
int   add (int x, int y);
float add (float x, float y); //Compile time error
```

Developers conventionally encode the input and output types in the name. For example:

```c
int   add_i (int x, int y);
float add_f (float x, float y);
```

Function Overloading Cont.

C++ added the concept of **Name Mangling** to the compiler. Functions with the same "names" but different parameters get assigned unique identifiers at compile time!

Function Overloading Cont.

For example g++ will compile the following two functions

```
int   add (int x, int y);
float add (float x, float y);
```

And the output will look like this (note: using the `nm` command)

```
__Z3addff
__Z3addii
```

Here the __Z is a reserved label for the compiler (so we don't need to care about it). The 3 means the function "name" is the next 3 characters. Finally, the `ff` and `ii` indicate two float and two int input parameters respectively.

Objects

C style structs are really just "Plain Old Data" (POD). And any API using these structs operates on some reference to an object.

```
typedef struct Michael {
    int  age;
    bool isAwesome;
} Michael;
Michael* constructMichael(){
    Michael* m = (Michael*)malloc(sizeof(Michael));
    m->age = 24; m->isAwesome = true; return m}
void destroyMichael(Michael* thisMichael)
    { free(thisMichael); }
void isMichaelAwesome(Michael* thisMichael)
    { printf("true"); //Assumed }
```

Basic Objects in C++

C++ Allows you to embed all this crap directly "in" the object. The compiler will create functions with an implicit pointer to the current object (this).

```
class Michael{
  private:
    int  _age;
    bool _isAwesome; //only Michael can modify this
  public:
    Michael(void) { _age = 24; _isAwesome = true; }
    ~Michael(void) { /*Destructor not necessary*/ }
    bool isAwesome(void){ return _isAwesome; }
};
```

Basic Objects in C++

The previous C++ class is roughly equivalent to to following.

```
struct Michael{
    int  _age;
    bool _isAwesome;
};
Michael createMichael()
{ Michael m; m._age = 24; m._isAwesome = true; }
void destroyMichael(Michael& this)
{ /*Destructor not necessary*/ }
bool isAwesome(Michael& m){ return m._isAwesome; }
```

Basic Objects in C++

However, since the _age and _isAwesome are private to Michael the following is invalid.

```
Michael m;
m._isAwesome = false;   // compile time error
cout << m._age << endl; // compile time error
```

Only members of the Michael class have access to private fields, however everyone has access to the public space.

```
Michael m;
cout << m.isAwesome() << endl; //Valid and Assumed
```

Object Lifecycles

Conceptually, the object lifecycle is pretty simple.

1. Programmer creates the object by calling some constructor
2. Programmer uses object
3. Object destructor called automatically when object is out of scope or if manually deleted via `delete`.

Object Lifecycles: Constructors

Constructors literally construct types. For example, a constructor might initialize variables, set some sort of state, allocate space on the heap, or even copy objects.

Constructors are functions with the same name as the class. A few special case constructors:

- Default constructor has no input parameters
- Copy constructors take some reference to an object of the same type
- Move constructors (noteworthy, but not important for this class :))

Object Lifecycles: Constructors

Imagine we have a Pokemon class:

```
Pokemon() //Default Constructor
 { type = nullptr; moveSet.append(Tackle(pp=25));
   health = 10; ...}

Pokemon(string mType, int mHealth, ...)
 { type = new PokeType(mType); ... }
//Pass by const lvalue reference
Pokemon(const Pokemon& that) // Copy Constructor
 { copy(that); return this; }
```

Object Lifecycles: Destructors

Destructors literally destroy objects, and by default they do nothing. Destructors are only necessary to help clean up after the object. For example, use destructors to:

- delete (free) any memory dynamically allocated by the class
- notify engine of object destruction (Advanced and rarely used)

Destructors are called when objects go out of scope (automatically added by compiler) or when they are deleted. Think about why this is awesome (hint: containers of objects).

Object Lifecycles: Destructors

So for our Pokemon example, we need a destructor to free up the Pokemon type data.

```
~Pokemon(){
    if(type)
        delete type;
    ...
}
```

Object Lifecycles: Example 1

```cpp
#include <iostream>
#include <string>
using namespace std;
class Human{ //private by default
  string name;
  public:
    Human(string name): name(name)
    { cout <<"Constructing: " << name << endl; }
    ~Human(){ cout << "RIP: " << name << endl; }
};
int main(){
    Human m = Human("Ash Ketchum");
    cout << "Hello" << endl;
    return 0;
}
```

Object Lifecycles: Example 1

This Outputs

```
Constructing: Ash Ketchum
Hello
RIP: Ash Ketchum
```

Object Lifecycles: Example 2

```cpp
void foo(){ Human m("Jim-Bob-Joe"); }

int main(){
    Human m = Human("Ash Ketchum");
    foo();
    cout << "Hello" << endl;
    return 0;
}
```

Object Lifecycles: Example 2

This Outputs

```
Constructing: Ash Ketchum
Constructing: Jim-Bob-Joe
RIP: Jim-Bob-Joe
Hello
RIP: Ash Ketchum
```

Object Lifecycles: Example 3

```cpp
int main(){
    Human* m =  new Human("Ash Ketchum");
    delete m;
    cout << "Hello" << endl;
    return 0;
}
```

Object Lifecycles: Example 3

This Outputs

```
Constructing: Ash Ketchum
RIP: Ash Ketchum
Hello
```

Object Operator Overloading

You can also define operators the same way you define functions!

```
Vec2d operator + (Vec2d a, Vec2d b)
 { return Vec2d(a.x + b.x, a.y + b.y); }
Vec2d operator + (Vec2d a, double b)
 { return Vec2d(a.x + b, a.y + b); }
Vec2d operator + (double b, Vec2d a)
 { return Vec2d(a.x + b, a.y + b); }
```

Namespaces and Scope Resolution

- Namespaces == named scope/block of code
  - Useful for big projects/libraries
- Scope resolution operator, `Scope::Qualified_Name`

```cpp
namespace Vallath{
  int magicMathFunction(...); //Probably has bugs
}
namespace Michael{
  int magicMathFunction(...); //Always Perfect
}
...
//Use the function in the Michael namespace
int byteMe = Michael::magicMathFunction(...);
```

Namespaces and Scope Resolution

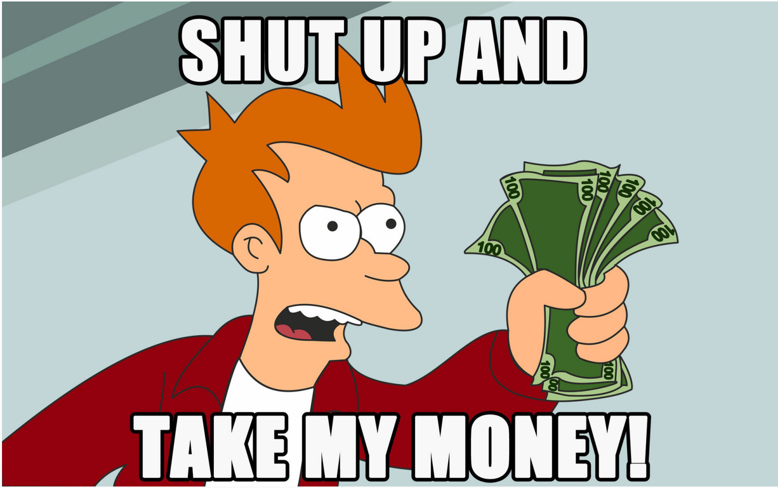
Scope resolution super useful for embedded types

```
class Vector{
    ...
    class iterator{...};
    iterator begin(void){...}
    iterator end(void){...}
};
...
// Iterate through the Vector elements
for(Vector::iterator i = mVec.begin();
    i != mVec.end(); i++){
        ...
}
```

Basic Takeaways

- Proper use of constructors/destructors makes it easier to debug memory leaks
- Objects encapsulate both logic and variables meaning code is smaller, is cleaner, and, generally, does more with less
- Language basics help expose more bugs at compile time
- C++ super scalable as compared to C
- Despite being more generic and reusable, many of the C++ STL datastructures and algorithms outperform hand tuned C code!

Basic Takeaways



Templates: Make Magic by Exploiting Types