Assorted Topics

ITP 435 – Spring 2016
Week 1, Lecture 2

Lecturer: Sanjay Madhav

Lab 1: RLE Compressor

- In class demonstration.

Unit Test

- Used to validate whether or not specific functionality works
- In theory, every feature should have a unit test that proves it functions correctly
- Many tools for this in C++…
  - Google Test
  - Boost Test Library
  - MiniCppUnit (Simple one to use)

Unit Testing Example

- WebKit HTML renderer

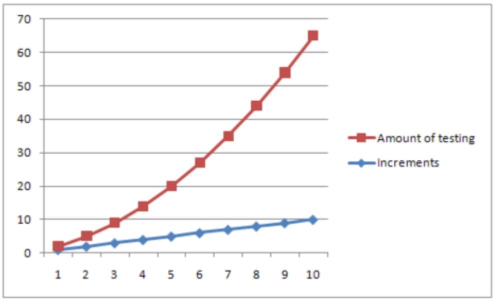
Safari

- The testing framework prints out the DOM (Document Object Model) and compares the actual output to the expected

- Thousands of unit tests (that take a few hours to run)

Regression

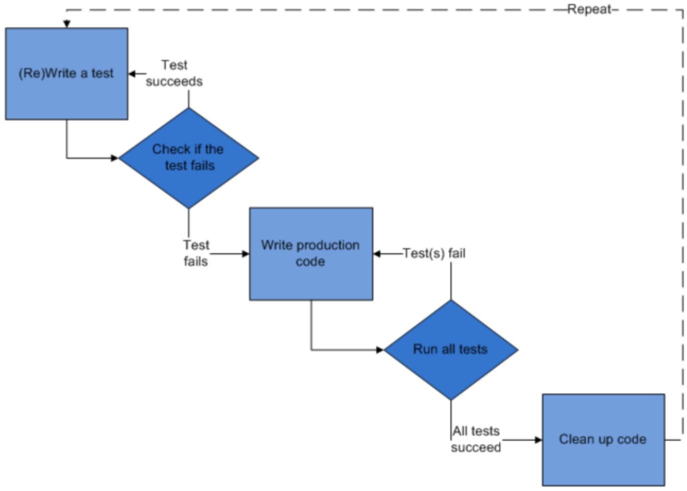
- A test that previously worked no longer works
- In a strict environment, unexpected regressions means the code cannot be committed



- Imperative that tests are automated for this to be efficient

Test-Driven Development

- Essentially means "write unit tests before writing the code"



Pointers (Review)

- A ***pointer*** is a type of variable that stores a memory address. Hopefully you remember this!
- Since memory addresses are binary numbers, from the perspective of the computer there really isn't much of a difference between an integer and a pointer
- However, when we use a pointer, we are telling C++ that we're using this number as a memory address

USC Viterbi School of Engineering
University of Southern California

More simply, sometimes people will say "pointers are integers" which is an accurate, though somewhat simplistic, way to look at it

Declaring a Pointer Example

```
int x = 0;
double y = 5.0;

int* z = &x;
```

	Memory	
Variable	**Address**	**Value**
x	0x04	0
y	0x08	5.0
z	0x10	??

USC Viterbi School of Engineering

University of Southern California

If we declare this pointer, what's "value" should be stored in it?

Declaring a Pointer Example

```
int x = 0;
double y = 5.0;

int* z = &x;
```

	Memory	
Variable	Address	Value
x	0x04	0
y	0x08	5.0
z	0x10	0x04

It’s 0x04, because we initialized it to the address of x, which is at 0x04.

Null Pointers

- Previously, null pointers were represented with 0 (or `NULL`, which is just a define as 0).
- This is not strongly typed...

```
void f1(int);
void f1(char *);

// 0 could mean int or it could mean a NULL pointer...
// Compiler always choose the “int” version.
f1(0);
```

- In C++11, there is a now a `nullptr` keyword, which is strongly typed

```
// nullptr is strongly-typed, so it calls the char* version
f1(nullptr);
```

Arrays and Pointers

```
int fib[] = {
   1,
   1,
   2,
   3,
   5,
};

int* ptr1 = &(fib[0]);
int* ptr2 = &(fib[4]);
```

	Memory	
Variable	Address	Value
fib[0]	0x10	1
fib[1]	0x14	1
fib[2]	0x18	2
fib[3]	0x1C	3
fib[4]	0x20	5
ptr1	0x24	0x10
ptr2	0x28	0x20

Pointer to a Pointer

```
double a = 10.0;

double* ptr1 = &a;
double* ptr2 = &a;

// ** because pointer to pointer!
double** ptrToPtr = &ptr1;

// Dereferencing...
// *ptrToPtr = ??
// **ptrToPtr = ??

// Address of...
// &ptrToPtr == ??
```

	Memory	
Variable	**Address**	**Value**
a	0x10	10.0
ptr1	0x18	0x10
ptr2	0x1C	0x10
ptrToPtr	0x20	0x18

USC Viterbi School of Engineering

University of Southern California

So ptrToPtr is 0x18 because that’s the memory address of ptr1

Pointer to a Pointer

```
double a = 10.0;

double* ptr1 = &a;
double* ptr2 = &a;

// ** because pointer to pointer!
double** ptrToPtr = &ptr1;

// Dereferencing...
// *ptrToPtr = 0x10
// **ptrToPtr = 10.0

// Address of...
// &ptrToPtr == 0x20
```

	Memory	
Variable	**Address**	**Value**
a	0x10	10.0
ptr1	0x18	0x10
ptr2	0x1C	0x10
ptrToPtr	0x20	0x18

USC Viterbi School of Engineering

University of Southern California

The many uses of const

- Global variables that never change:

```
const int ScreenWidth = 1024;
```

- You should use the above as opposed to a define:

```
#define SCREEN_WIDTH 1024
```

- If you have a pointer, it can either be constant itself, or point to a constant value:

```
// Name is pointer to a const char
const char* Name = "Sanjay";
// pWidth is a const pointer to an int
int* const pWidth = &ScreenWidth;
// Name is a const pointer to a const char
const char* const Name = "Sanjay";
```

Passing by Value

- Basic types are okay to pass by value:

```
// Declares add, which passes lhs and rhs by value
int Add(int lhs, int rhs);
```

- You should almost never pass classes by value:

```
// BAD!!! Makes unnecessary copies of lhs/rhs
Vector3 Add(Vector3 lhs, Vector3 rhs);
```

Passing with Pointers

- You could pass with pointers (you have to in plain old C). This guarantees that you don't copy over the class data.

```
// lhs and rhs are passed as pointers
Vector3 Add(Vector3* lhs, Vector3* rhs);
```

- But pointers do not have to have a valid address:

```
// This is a valid call, and probably will crash
Add(nullptr, nullptr);
```

Pass by Reference

- Like with pointers, you don't copy over class data when passing by reference:

```
// lhs and rhs are passed by reference
Vector3 Add(Vector3& lhs, Vector3& rhs);
```

- However, if you pass by regular reference, Add could change the values of lhs and rhs.

- Instead, you can pass by const reference:

```
// Add is disallowed from changing lhs or rhs
Vector3 Add(const Vector3& lhs, const Vector3& rhs);
```

Const Correctness

- Suppose Vector3 has a function Clear that sets x, y, and z to 0:

```
class Vector3
  {
public:
      void Clear(); // Resets to 0
};
```

- The compiler has to prevent you from running Clear on a const reference to a Vector3:

```
// This won't compile
void Calculate(const Vector3& v)
{
     v.Clear();
}
```

Const Member Functions

- If you have a function that does not change member data, you need to mark it as a const function:

```
class Vector3
{
public:
	float GetX() const; // GetX() can’t change member data
};
```

- You can then run GetX() on const references:

```
// Works
void Calculate(const Vector3& v)
{
	std::cout << v.GetX();
}
```

USC Viterbi School of Engineering

University of Southern California

Note that functions declared outside a class cannot be const.

Const Best Practices

- Any member functions which do not change member data in your class should be declared as const member functions.

  (If you don't do this, other programmers might get mad)

- Pass classes by const reference when you want to guarantee your function does not modify said class.

USC Viterbi School of Engineering

University of Southern California

C-style Casts

- In C, to cast from one type to another you typically do:

```
int i;
float f = (float) i;
```

- Problem 1: Searching for all casts in a file is impossible.

- Problem 2: C-style casts allow you to do crazy (unsafe) stuff like:

```
int random_address = 0x123456;
char* garbage = (char*)random_address;
(*garbage) = 'a'; // probably will crash
```

C++ Style Casts

- static_cast – Allows you to do implicit conversions (eg. int -> float), as well as within a class hierarchy (without any runtime checks)

```
int i;
float f = static_cast<float> (i);
```

- reinterpret_cast – Allows you to do any cast you can do in C.

```
int random_address = 0x123456;
char* garbage = reinterpret_cast<char*> (random_address);
```

- const_cast – Allows you to strip the "const" away from const references (you should stay away from this in most cases)

- dynamic_cast – Does a "down cast" from parent class to child class, while checking whether or not it is valid at runtime.

dynamic_cast

- A down cast allows you to take a parent class pointer, and at runtime try to cast it to a child class.
- Eg. If you have a “Shape” pointer, and want to find out if it’s a “Triangle” at runtime, you can do:

```
Shape* myShape;
Triangle* myTriangle = dynamic_cast<Triangle*> (myShape);
if (myTriangle) // dynamic_cast returns 0 if not a triangle
{
      // do something
}
```

- In most cases, dynamic_cast shouldn’t be necessary if we correctly used polymorphism.
- This requires runtime-type information to be enabled in the compiler.

Note, you can also do a down cast with `static_cast`, but it will do no check to ensure the cast is valid.

(A Little More) Complex Pointer Stuff

- A pointer that points to a struct/class often points to the first member:

```
struct MyStruct
{
   int m_Int;
};
MyStruct* s;
(void*) s == (void*) &(s->m_Int) // TRUE
```

- void* pointers can be cast to any other type:

```
void* v;
MyStruct* s = reinterpret_cast<MyStruct*> (v);
```

USC Viterbi School of Engineering — University of Southern California

Exception: If the class/struct has virtual functions, the pointer may not necessarily point to the first element.

Pointer Arithmetic

- Results of pointer arithmetic depend on the type, and don't work on void*:

```
int* i;
i++; // Increments memory address by 4 bytes (sizeof(int))
char* c;
c++; // Increments memory address by 1 byte (sizeof(char))
void* v;
v++; // Compile error
```

An Issue with Const

- In some instances, you may have a member variable which needs to be modifiable even in a const member function.
- Consider a multi-threaded file system:

```
class FileSystem
{
private:
    std::mutex Lock;
public:
    unsigned int GetBytesFree() const;
};
```

- We want `GetBytesFree` to be const, since it's not changing anything in the actual file system.
- However, if we are going to use our mutex to guarantee it's thread safe, we need `GetBytesFree` to be able to modify `Lock`.

Mutable to the rescue

- If you mark a variable as mutable, then it can be changed even in a const member function:

```
class FileSystem
{
private:
    mutable std::mutex Lock; // Lock can now be changed
public:
    unsigned int GetBytesFree() const;
};
```

- With great power comes great responsibility… you should only use mutable in very specific and unique cases such as the above.

Static

- A class can have a static variable, which means it's shared between all instances of the class.

```
// StaticTest.h
class StaticTest
{
private:
     static int num_instances;
}

// StaticTest.cpp
int StaticTest::num_instances = 0;
```

- In the above, I could make the constructor increment num_instances and the destructor decrement it. Then I know how many instances of my class are in use at any one time.

Static, Continued

- A static variable declared within a function retains state between subsequent calls to the function.

```
void MyFunction()
{
    // Track the number of times the function is called
    // num_calls is initialized on load of the program
    static int num_calls = 0;
    num_calls++;
}
```

Empty Class

- If you have this:

```
class Empty { };
```

- What code gets run when you do this?

```
Empty* test = new Empty();
delete test;
```

Compiler Generated Functions (in C++ 98)

- That empty class effectively is:

```
class Empty
{
public:
     Empty() { /*...*/ }
     Empty(const Empty& rhs) { /*...*/ }
     ~Empty() {}
     Empty& operator=(const Empty& rhs) { /*...*/ }
};
```

USC Viterbi School of Engineering

University of Southern California

Constructor, copy constructor, destructor, and assignment operator

In C++11, there are two more functions (which we'll talk about later)

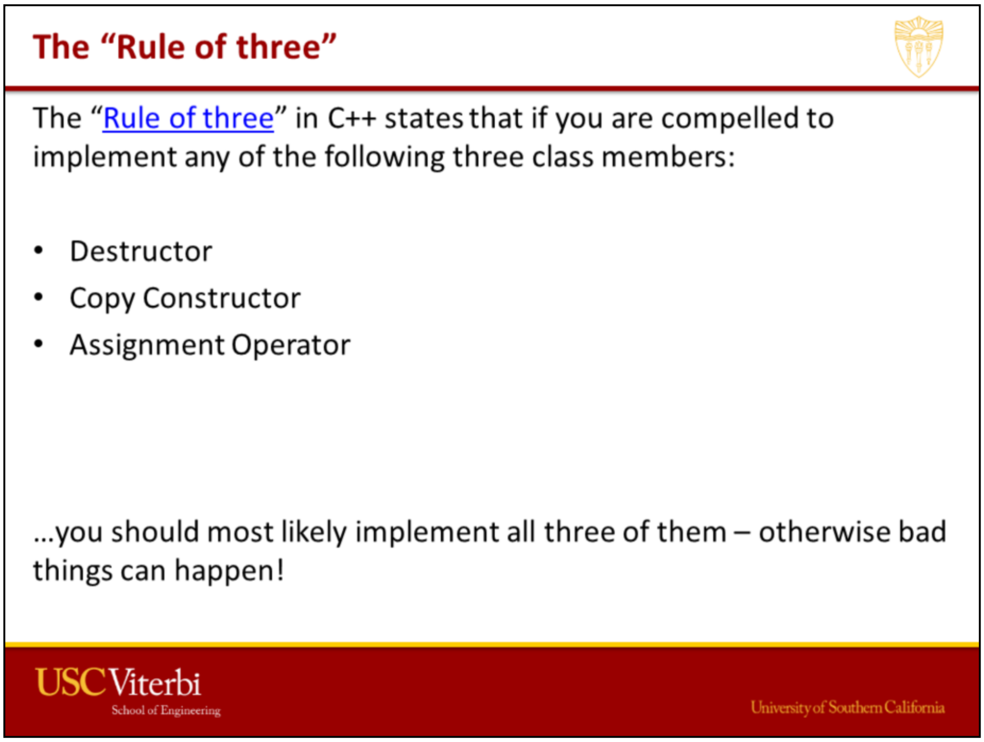
Default copy/assignment

- Default copy constructor and assignment operators are ***shallow*** copies
- So something like this:

```
template <class T>
class List<T>
{
/*...*/
private:
    Node<T>* pHead;
};
```

- Would not copy properly if you did this:

```
List<int> L1;
L1.Add(123);
List<int> L2(L1); // shallow copy bad :(
```



So for instance, if you are dynamically allocating stuff, that tells you that you need a destructor to prevent memory leaks. By the rule of three, this means you probably need copy constructor and assignment to ensure you are doing deep copies.

Disallowing Functions

- If you declare a normally auto-generated function as private, it's not usable by anyone:

```
class Test
{
private:
    // Prevent anyone from using these functions!!
    Test(const Test& rhs){ }
    Test& operator=(const Test& rhs) { }
};
```

Implicit Construction

- If you have this:

```
class A
{
public:
      A(int value);
};
void my_func(const A& my_A);
```

- The following is legal code:

```
my_func(12345); // Will construct an A with provided integer
```

Explicit Keyword

- To disallow previous scenario, use `explicit`:

```
class A
{
public:
    explicit A(int value);
};
```

- Then only this would be allowed:

```
my_func(A(1234));
```

Override Keyword

- Suppose there is a Class A:

```
class A
{
public:
   virtual void TakeDamage(int amount);
};
```

- Class B inherits from A, and overrides TakeDamage:

```
class B : public A
{
public:
   void TakeDamage();
};
```

- What’s wrong with this picture (yes, it compiles!)?

Class B intends to override TakeDamage, but it in fact implements a new take damage that does not override A’s, because the parameters are different.

Override Keyword, Cont'd

- The keyword `override` after the function name/parameters says "I guarantee this overrides a function from a parent class"

- So if we wrote the code as:

```
class B : public A
{
public:
   void TakeDamage() override;
};
```

- Error C3668: 'B::TakeDamage' : method with override specifier 'override' did not override any base class methods

USC Viterbi School of Engineering

University of Southern California

So using override can be a good way to prevent such errors.