ADVANCED CPP 4

L VALUES & R VALUES

* L value is an object that occupies some *identifiable location in memory*
* R value is any object that is *not* an L value

L value

*Most values in c++ are l values, including the following:*

Int i;

Int\* p = &i;

i = 2;

className name;

R value

*Below are some examples of r values*

Int x = 2; // ‘2’ is an r value (x is l value)

i + 2; // &(i + 2) = ERROR!

className = name(); // name() is the r value

REFERENCE (L value reference)

Int i;

Int& r = i; // i is left value – this is valud

Int& r = 5; // error! ‘5’ is an r value

There is an exception; *constant L value references can be assigned an R value (however, this is bad practice):*

*Const int& r = 5;*

Example (using a function):

Int square(int& x) { return x \* x; }

Square(i); // this is ok, the variable ‘i’ is a L value

Square(40); // not ok, 40 is an r value

// changing the function parameter to ‘*const int& x*’ would allow the second example to work.

R VALUE REFERENCE

R value reference was introduced in c++11 and are used for 2 things:

1. Moving semantics
2. Perfect forwarding

The syntax for an r value reference is to use 2 && ampersands: int&& c;

Int& num = 10; // error (L value reference)

Int&& num = 10; // ok

Example (using a function):

Int square(int&& num) { return x \* x; }

Square(40);

Move semantics

Move semantics is a way of moving resources around in an optimal way by avoiding unnecessary copies of temporary objects.

*Implementing a move constructor:*

Class Holder {

Public:

Holder(Holder&& other) {

M\_data = other.m\_data;

M\_size = other.m\_size;

Other.m\_data = nullptr; // no longer 2 copies

Other.m\_size = 0;

}

}

Move semantics provides a smarter way of passing heavy weight things around. You create your heavy-weight resource only once, then move it where needed in natural ways.

COPY CONSTRUCTOR

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

1. Initialize an object from another of the same type
2. Copy an object to pass it as an argument to a function
3. Copy an object to return it from a function

If a copy constructor is not defined in a class, the compiler itself defines one.

If the class has pointer variables and has some dynamic memory allocations, then it must have a copy constructor.

e.g.

class Point {

private:

int x, y;

public:

Point() {}

Point(const Point& p2) { x = p2.x; y = p2.y; }

Int getX() { return x; }

Int getY() { return y; }

}

Point p; // normal constructor called here

Point r = p; // copy constructor called here

*See below section copy constructors and assignment operators*

Rule of 3

If a class defines any of the following, then it should explicitly define all 3

* Destructor
* Copy constructor
* Copy assignment operator

Rule of 5

With the introduction of move semantics (r value references), in c++11, the rule of 3 was extended to include the following:

* Move constructor
* Move assignment operator

COPY ASSIGNMENT OPERATOR

The copy assignment operator lets you create a new object from an existing one by initialization. A copy assignment operator of class A is a non-static non-template member function that has one of the following forms:

A::operator=(A)

A::operator=(A&)

A::operator=(const A&)

If you do not declare a copy assignment operator for a class A, the compiler will implicitly declare one for you that is inline public.

e.g.

class Test {

public:

Test() {}

Test& operator= (const Test& t) {

return \*this;

}

}

Test p, r;

r = p; // note that the copy constructor will be called too due to the parameter that the assignment operator uses

COPY CONSTRUCTORS AND ASSIGNMENT OPERATORS

Assignment versus initialization

myClass one;

myClass two = one;

^^ here, the variable *two* is initialized to one because it is created as a copy of another variable. When two is created, it will go from containing garbage data directly to holding a copy of the value of *one* with no intermediate step.

myClass one, two;

two = one;

^^ here, *two* is assigned the value of *one*. *Two* has already been initialized on the line above *two = one*.

Copy constructors

Copy constructors are invoked whenever:

1. A newly-created object is initialized to the value of an existing object
2. An object is passed to a function as a non-reference parameter
3. An object is returned from a function

e.g. 1

myClass one;

myClass two = one;

myClass three = two;

^^ here, since *two* and *three* are being initialized to the values of *one* and *two*, respectively C++ will invoke the copy constructors to initialize their values. The above is equivalent to:

myClass one;

myClass two(one);

myClass three(two);

e.g. 2

void myFunction(myClass parameter) {}

myClass mc;

myFunction(mc);

^^ the variable *parameter* inside of myFunction will be initialized to a copy of *mc* using the copy constructor.

e.g. 3

MyClass myFunction() {

MyClass mc;

Return mc;

}

^^ if we call myFunction, then C++ will create a new myClass object that’s initialized to *mc* when myFunction returns.

Assignment operators

While the copy constructor is used to setup a new version of an object that’s a duplicate of another object, the assignment operator is used to overwrite the value of an already-created object with the contents of another class instance.

For example, the following will invoke the assignment operator, not the copy constructor:

myClass one, two;

two = one;

what does C++ does for you

C++ will automatically provide objects a basic copy constructor and assignment operator for you. In many cases with only primitive data this is fine, however, consider the following code:

Class CString {

Public:

CString();

~Cstring(); // deallocates the stored string

// note, no copy constructor or assignment operator

Private:

Char\* theString; // a pointer!

}

^^ here, if we rely on C++ default copy constructor and assignment operator, we will run in to trouble. Consider the following code:

CString one;

Cstring two = one;

Because we haven’t provided a copy constructor, C++ will initialize two.theString to one.theString. Since *theString* is a char\* (char pointer), instead of getting a deep copy of the string, we’ll end up with two pointers to the same CString. Thus, changes to *one* will show up in *two* and vice versa*.* This is dangerous, especially when the destructors for both *one* and *two* try to deallocate the memory for *theString*

TEMPLATES

*Example 1:*

Template<int N>

Class Array {

Private:

Int m\_array[N];

Public:

Int getSize() { return N }

}

Array<5> array;

Array.getSize(); // 5

*Example 2:*

Template<typename T, int N>

Void add(T num) {

Std::cout << num + N;

}

Add<int, 10>(5); // 15

Variadic template

We can use variadic functions to write functions that accept arbitrary number of arguments. Variable templates are templates that take a variable number of arguments.

Template<typename… Ts> // declare a template parameter pack

Void ignore(Ts… ts) {} // this function accepts a bag of parameters

Ignore<int, double, bool>(1, 2.0, true);

// this is equivalent to

Template<typename T1, typename T2, …, typename Tn>

Void ignore(T1 t1, T2 t2, …, Tn tn) {}

When it comes to handling variadic functions, you can’t think in the standard ‘iterative c++ style’. You need to write such functions recursively; with a *base* case, and a *recursive* case, that reduces, eventually, in to a *base* case. This implies a separate function for each case.

*Example:*

// base case

Template<typename T>

Double sum(T t) { return t; }

// recursive case

Template<typename T, typename… Ts>

Double sum(T t, Ts… ts) {

Return t + sum(ts…);

}

Sum(1.0, 2.0, 3.0);

Sum<int, float, double>(1, 2.0, 3.0);

// a template parameter pack can also be empty.

*Example 2:*

Template<typename T>

T square(T t) { return t \* t; }

Template<typename T>

Double power\_sum(T t) { return t; }

Template<typename T, typename… rest>

Double power\_sum(T t, Rest… rest) {

Return t + power\_sum(square(rest)…);

}

Double result = power\_sum(4, 3, 2); // 29

// this narrows down to:

// 4 + (3 \* 3) + ((2 \* 2)(2\*2)) or

// 4 + (square(3)) + (square(square(2)))

BITWISE OPERATORS

Binary Representation

Binary digits (bits), are multiples of power of 2

8 bits = 11111111 = 255

16 bits = 1111111111111111 = 65,535

Operators

1. Left shit << // 1101101 << 11011010
2. Right shift >> // 11101101 >> 01110110

Unsigned int num = 8;

Std::cout << (num >> 1); // from 00001000 shifted once to the right 00000100 = 4

Std::cout << (num << 1); // from 00001000 to 00010000 = 16

Std::cout << (num << 3); // shift 3 times to the left = 64

Std::cout << (num << 0); // not shifted = 8

Unsigned int num = 7;

Std::cout << (num << 1); // from 00000111 to 00001110 = 14

1. AND &
2. OR |
3. XOR ^ // exclusive or
4. NOT ~

(5 & 3) = 1

0101 (5)

0011 (3)

0001

(5 | 3) = 7

0101

0011

0111

(5 ^ 3) = 6

0101

0011

0110

To understand NOT (~), one must understand ***two’s complement.*** Twos complement is a clever way of storing integers:

1. For zero – use all 0s, e.g. 0000
2. For positive, count up to max number of bits, e.g. 0111 (0 is used for positive)
3. For negative, reverse the roles of 0 and 1, e.g. 1000 (1 is used for negative)

e.g.

0000 = zero

0001 = one

0010 = two

0011 = three

0100 to 0111 = four to seven

1111 = negative one

1110 = negative two

1101 = negative three

1100 to 1000 = negative four to negative 8

~5 = 6

0101 (5)

1**0**1**0** = -6 // first 1 means negative. 0 and 1 are reversed so we count the **bold** (with the initial negative 1)

**Test**

1. What is the difference between L value and R value?
2. Give an example of a function which uses a L value reference in the function signature
3. Give an example of a function which uses a R value reference in the function signature
4. How can a R value reference be legally passed to a function whose signature only accepts a L value reference?
5. What is meant by move semantics?
6. Implement a move constructor
7. Implement a copy constructor? What is it?
8. Give an example of a template with 2 arguments
9. Give an example of a variadic template function which sums up the arguments passed