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.

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

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)))