CPP Programming: Session 6

Anupama Chandrasekhar (September 6 2018)

Copy and Move

- When we need to transfer a value from **a** to **b**, we usually have two logically distinct options:
  - *Copy* is the conventional meaning of **x=y**; that is, the effect is that the values of **x** and **y** are both equal to **y**'s value before the assignment.
  - *Move* leaves **x** with **y**'s former value and **y** with some *moved-from state.*
- Typically a move cannot throw but a copy might since it might need to acquire memory
- Copy and Move have default definitions

Copy Constructor: Motivation

- In case of a function call of type **`int f(int a, char b)`**, the compiler pushes the argument to the stack and makes the function call, this is easy because for built in types the compiler knows exactly the what the sizes of the types are and how to copy them. And the return value is placed in a register. With built in types copying and object is equivalent to copying the bits of the value.
- Passing large objects is trickier because it is usually more than a bitwise copy, particularly if the data members are pointers and what you require from a copy is copying of the contents pointed by the pointer and not just copying the pointer. See example **1_howmany.cpp**

Copy Constructor : Explanation of HowMany.cpp

- After **h** is created the object count is 1 which is correct. But inside **f(),** where **h** is passed by value, the count it still 1, which looks wrong.
- Also, after call to **f()** and the **h2 = f(h),** we expect the object count to be 2 but it is 0. So something is off, further validated by the fact that the destructor is called twice.
- This problem occurs because the compiler incorrectly constructs a copy of the object, is assumes that it needs to do a bitwise copy.

Copy Constructor: Solution

- You can prevent the compiler from doing a bitwise copy by defining a copy constructor. Obviously we can't pass the source object by value, we need to pass it by reference. Often referred to as **`X(X&)`** **.**
- **`See HowMany2.cpp`**

Default Copy Constructor

- Because a copy constructor is need for pass by value and return values, the compiler creates one in case of simple structures, for simple primitive types this is a bit copy.
- In the case more complex types the complier will still automatically create a copy constructor, and it can do something more intelligent than a simple bit copy.
- See **DefaultCopyConstructor.cpp**
- The compiler recursively calls the copy-constructors for all the member objects and base classes. That is if the member object also contains another object its copy-constructor is also called. So in this case the compiler calls the copy-constructor for **WithCC**. The output shows this constructor being called. Because **WoCC** has no copy-constructor the compiler creates one for it that just performs a bitcopy and calls that inside the **Composite** copy-constructor. The call to **Composite::print( )** in main shows that this happens because the contents of **c2.wocc** are identical to the contents of **c.wocc**.

Lvalue and Rvalues recap

- What are Lvalues and rvalues?

```
x; // lvalue
*p; // lvalue
arr[0]; // lvalue

2; // rvalue
x + y; // rvalue
foo(); // rvalue
```

- **Rule of thumb**: Lvalue if you can take its address, if not rvalue

References

- References are names/aliases to objects

```
int& rx = x; //reference to x
const int& crx = x; // cannot modify x through crx
```

- Until C++11, references bound only to lvalues. Well, const references bind to rvalue references but you can't do much with them.
- C++11 introduces rvalue references that bind to rvalues


```
int&& rrx = foo();
```

- Use: to implement move semantics

```
int foo();

int main()
{
    int& rx = foo()
    int&& rrx = foo();
}
```

Move Constructors : Motivation and Implementation

```
string MakeFood()
{
    return "Noodles";
}

int main()
{
    string s{"Rice"};
    string s2(s); // Calls copy constructor
    string s3(makeFood()); // temp MakeFood, copy, destroy MakeFood()
}
```

- See MyVector.cpp

Operator Overloading

- Operator overloading is just another way to make a function call.
- The operators for built in data types cannot be overloaded, you can only overload operators for expressions involving user-defined datatypes.
- So if you need complex arithmetic, matrix algebra, string manipulation, classes are used to express these notions and defining operators for these classes gives a clean/shorthand way of expressing operations.

Operator Overloading : Example

- Overloading the **+** operator.

```
Complex a{1.2,1.3};
Complex b{2.1,3};
Complex c = a+b;
```

- For this to work the operator **+** needs to be overloaded.
- Equivalently:

```
Complex c = a.Add(b);
```

- See example 6_complex.h

Operator Overloading contd.

- Alternatively, we could have defined a global function

```
friend Complex operator+(Complex);

Complex operator+(const Complex  &num1, const Complex &num2)
{
    double result_real = num1.real + num2.real;
    double result_imaginary = num1.imag + num2.imag;
    return Complex( result_real, result_imaginary );
}
```

- When operator is a class member:

```
Complex a{1, 2 };
Complex b{2, 2 };
Complex c = a.operator=( b );
```

Copy Assignment

- Like copy constructors, copy assignments are used to make one object a copy of the other

- See **copyassignment.cpp**
- Revisiting RAII and Smart Pointers.

Function Templates

- Function templates are special functions that can operate with generic types. This allows for code reuse. A template parameter is used for passing type as a parameter.
- The format for declaring is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

- To use:

```
function_name<type>(parameters);
```

Class Templates

- Like function templates, but applied to classes.
- See example **9_classtemplates.cpp**
- When you do not want the default behavior for a certain type, you can specialize the function/class for that type.

- Just a glimpse into variadic templates : A way to write functions that take arbitrary number of arguments in a typesafe wayadn having all the argument handling done at compile time rather than runtime.
- See example **10_variadictemplates.cpp**