Object-Oriented Programming

CSCI-UA 0470-001
Class 18
Instructor: Randy Shepherd

Translating Constructors

Constructors in the Target Language

- Translate constructor logic like into a regular method with some standardized name (e.g., _init).
- Moreover, introduce _init methods for each class corresponding to each constructor.
- Inside of each _init method, add an explicit call to the default _init method of the super class, before the actual body of _init is executed

Constructors in the Target Language

```
struct __A {
  __A_VT* __vptr;
  std::string a;

  __A();

  // impl should appear in cpp
  static void __init(A o) {
     o->a = "a";
  }

  static String toString(A);
  static Class __class();
  static __A_VT __vtable;
};
```

```
struct __B {
  __B_VT* __vptr;
  std::string a;

  __B();

  static void __init(B o) {
     __A::__init((A) o);
     o->a = "b";
  }

  static String toString(B);
  static Class __class();
  static __B_VT __vtable;
};
```

Constructors in the Target Language

- Every call to ‘super’ in a constructor is just replaced by a call to the corresponding _init method of the super class.

- Also, for 'new' statements, add an explicit call to _init after the allocation of the data layout for the new instance.

Overloading in C++

Function Overloading

- Like Java, C++ allows you to specify more than one definition for a function name in the same scope

- Functions may or may not be part of a class, so-called ‘free functions’ can be overloaded in the same manner as methods.

Function Overloading

- Like Java, functions must differ from each other by the types and arity of parameters.
- The compiler selects the implementation to use by comparing the argument types at the call site with the parameter types in the definitions.

```
int overloaded() { /* ... */ }

int overloaded(int a) { /* ... */ }

int overloaded(double a) { /* ... */ }

int overloaded(int a, double b) { /* ... */ }
```

Function Overloading

- Like Java, you can not overload function declarations that differ only by return type.

```
class Bar {
  Bar() { }

  // does not compile

  int bar() {
    return 1;
  }

  float bar() {
    return 1.0;
  }
};
```

Differences to Java

- So pretty similar to Java, except..

- Methods with the same name and the same parameter types cannot be overloaded if any of them is a static function.

```cpp
class Foo {
  Foo() { }

  // does not compile

  void foo() {
    cout << "foo" << endl;
  }

  static void foo() {
    cout << "static foo" << endl;
  }
};
```

Example code

- Code demonstrating method overloading in C++ can be found in the following repository

  https://github.com/nyu-oop/overloading-cpp

Operator Overloading

- operator overloading, like method overloading is a specific case of polymorphism, where different operators have different implementations depending on their arguments.
  - Ex. “foo” + “bar” *vs* 1 + 2
- In the simplest terms, operators are functions with special, symbolic function names.
  - Ex. a + b * c *vs* add(a, multiply (b,c))

Operator Overloading

- You can overload most of the built-in *operators* available in C++.
  - Some exceptions, ex. “::” and “?:”
- Thus you can use operators with user-defined types.
- Java does not support user operator overloading.

Example

- Lets suppose we are building graphics library.
- We have a number shape primitives and we'd like to be able to easily add two of them together to form a third that has the volume of the two combined.
- We define some type Box
- Wouldn't it be nice if we could write…

```
Box b1(1, 1, 1);
Box b2(1, 1, 1);
Box b3 = b1 + b2;
```

Example

- Note the method on line 14.

- Overloaded operators are functions with special names i.e. the keyword *operator* followed by the symbol for the operator being defined.

- Now we can do this..

```
Box b1(1, 1, 1);
Box b2(1, 1, 1);
Box b3 = b1 + b2;
```

```
1  class Box {
2  private:
3      double height;
4      double length;
5      double breadth;
6
7  public:
8      Box(double height, double length, double breadth) :
9        height(height), length(length), breadth(breadth) { }
10
11     // accessors....
12
13     // Overload + operator to add two Box objects.
14     Box operator+(const Box& b) {
15       double height = this->height + b.getHeight();
16       double length = this->length + b.getLength();
17       double breadth = this->breadth + b.getBreadth();
18
19       return Box(height, length, breadth);
20     }
21 };
```

Advantages

- Add the end of the day, operator overloading is *syntactic sugar.*

- It is used because it allows user-defined types to have a similar level of syntactic support as types built into the language.

- When used judiciously, it can lead to simpler, more expressive code.

Disadvantages

- Can be confusing if an operator is overloaded for multiple classes.
  - Same operator symbol would represent different things in different contexts.
  - This can lead to hard-to-read code.
- Operators could be overloaded to do something unintuitive (switching + and -)
  - What if the Box + actually subtracted the two? Seem crazy.. yet these things do happen.

3 Basic Rules of Operator Overloading

- *Whenever the meaning of an operator is not obviously clear and undisputed, it should not be overloaded.*
  - Basically, the first and foremost rule for overloading operators, at its very heart, says: Don’t do it.
  - Symbolic operators contain less information than a method name, except in very obvious cases (like addition)
  - Obvious cases are rare.

3 Basic Rules of Operator Overloading

- *Always stick to the operator's well-known semantics.*
    - The compiler will happily accept code that implements the '+' operator to subtract.
    - However, the users of such an operator would never suspect the expression a + b to subtract a from b.

3 Basic Rules of Operator Overloading

- *Always provide a set of related operations.*
    - Operators are related to each other and to other operations. If your type supports a + b, generally speaking, it should support a - b as well.

Example code

- Code demonstrating method overloading in C++ can be found in the following repository

  https://github.com/nyu-oop/overloading-cpp

Operator Overloading in Translator

- So far in our translator, we need to check to see if the index is within the bounds of the array. Then we can access a specific element in the array

```
__rt::checkIndex(a, 2);
std::cout << "a[2]  : " << a->__data[2] << std::endl;
```

Array Dereference Operator

- What if we can check if an index is within the bounds of the array while accessing the element in the array. Just like in Java.

- We can overload the array subscript operator to provide this additional functionality.

```
T& operator[](int32_t index) {
  if (0 > index || index >= length)
    throw ArrayIndexOutOfBoundsException();
  return __data[index];
}
```

Array Dereference Operator

- Now we can access an element in an array by using the syntax.

```
(*a)[2];
```

- Note that the [] operator must return a reference to an array element to support modification of the array

- Also note that the [] operator always takes an index, so it can check whether the index is in bounds

Array Dereference Operator

- By convention, when you define the [] operator, you also need to define a const [] operator.

- The const [] operator does not allow modification of a const array.

- If the array is const it will return a const element.

```
const T& operator[](int32_t index) const {
  if (0 > index || index >= length)
    throw ArrayIndexOutOfBoundsException();
  return __data[index];
}
```

Left Shift Operator

- Every time we want to print a string, we have to use the code:

```
cout << k->__vptr->getName(k)->data // k.getName()
```

- Wouldn't it be nice if we didn't have to use '->data' every time we were printing a string object?

- We can do that by overloading the << operator (left shift operator)

- But wait a minute.. << is part of the standard library! Actually that not a problem.

Left Shift Operator

- We can overload the left-shift operator like so..

```
std::ostream& operator<<(std::ostream& out, String s){
  out << s->data;
  return out;
}
```

- This overloads the left-shift operator *for the ostream* when our String object is passed in.

- Returning *out* enables chaining. If we had void return type, then chaining would be impossible.

```
String s1("1");
String s2("2");
cout << s1 << s2 << endl;
```

Implementations

- Overloading the [] operator is an example of an operator that is a method of a class (our array template)

- Overloading the << operator is an example of overloading an operator that is not a member of a class.

- The target library is updated with these operators overloads in the java_lang_4 repo.

- You can feel free to add more if the need arises.

Wrapper Class for Pointers

- To illustrate the power of operator overloading we implement a wrapper class for the built-in pointers of C++.
- For now, the wrapper class provides the same functionality as a regular pointer.
- However, later on we will extend it with functionality for automatic memory management (smart pointers).

Wrapper Class for Pointers

- We made a new template class: Ptr<T> where T is some type to which we point.
- Ex. Ptr<int> syntax is more intuitive than int*
- A Ptr instance contains an address of a T

Wrapper Class for Pointers

- We have a new template class Ptr<T> where T is some type to which we point.

- Ptr<int> syntax is more intuitive than int*

- (trace is a function that prints the string and line number to std out)

```cpp
template<typename T>
  class Ptr {
    T* addr;

  public:
    Ptr(T* addr) : addr(addr) {
      trace("constructor");
    }

    Ptr(const Ptr& other) : addr(other.addr) {
      trace("copy constructor");
    }

    ~Ptr() {
      trace("destructor");
    }

    // ...
```

Wrapper Class for Pointers

- A Ptr instance contains an address of a T
    - i.e. the pointer we are wrapping
- Constructor on line 6 takes pointer as argument
- Also a ‘copy constructor’ on line 10 to make a copy of another instance

```
1  template<typename T>
2    class Ptr {
3      T* addr;
4
5    public:
6      Ptr(T* addr) : addr(addr) {
7        trace("constructor");
8      }
9
10     Ptr(const Ptr& other) : addr(other.addr) {
11       trace("copy constructor");
12     }
13
14     ~Ptr() {
15       trace("destructor");
16     }
17
18     // ...
```

Wrapper Class for Pointers

- Note line 14
- This ~ClassName syntax indicates that this is a destructor
- Intuitively you can think of it as the opposite of a constructor

```
1  template<typename T>
2    class Ptr {
3      T* addr;
4
5    public:
6      Ptr(T* addr) : addr(addr) {
7        trace("constructor");
8      }
9
10     Ptr(const Ptr& other) : addr(other.addr) {
11       trace("copy constructor");
12     }
13
14     ~Ptr() {
15       trace("destructor");
16     }
17
18     // ...
```

Destructors

- A destructor is a method which is automatically invoked when the object is destroyed. i.e. when the memory is to be freed.

- It can be called when its lifetime is bound to scope and the execution leaves the scope or when it was allocated dynamically and is released explicitly.

- Its main purpose is to free the resources (memory allocations, open files etc.) which were acquired by the object along its life cycle

Destructors

- If the instance is allocated on the stack, the destructor is called automatically when the instance goes out of scope.
- If the instance is allocated on the heap, the destructor is called when the instance is explicitly deallocated with a delete statement.
- The destructor implicitly calls the destructor of the super class (assuming there is one) as well as the destructors of all members which have them.
- If you do not provide a destructor, the compiler provides a default destructor, which does nothing accept for the above implicit destructor calls.

Overloading Assignment

- On line 3 we overload the assignment operator.

```
1  // ...
2
3  Ptr& operator=(const Ptr& right) {
4    trace("assignment operator");
5    if (addr != right.addr) {
6      addr = right.addr;
7    }
8    return *this;
9  }
10
11 T& operator*() const {
12   trace("dereference operator");
13   return *addr;
14 }
15
16 T* operator->() const {
17   trace("arrow operator");
18   return addr;
19 }
```

Overloading Assignment

- This is done to make sure that the stored address in the instance is reassigned to a new value when our pointer appears in the left hand side of an assignment expression.

- This will be necessary later. (We will be doing "reference counting")

- When overloading the assignment operator, you must protect against self assignment. (line 5)

```
1  // ...
2
3  Ptr& operator=(const Ptr& right) {
4    trace("assignment operator");
5    if (addr != right.addr) {
6      addr = right.addr;
7    }
8    return *this;
9  }
10
11 T& operator*() const {
12   trace("dereference operator");
13   return *addr;
14 }
15
16 T* operator->() const {
17   trace("arrow operator");
18   return addr;
19 }
```

Guarding Self-Assignment

- The typical sequence of operations within an assignment operator is something like this.

```
MyClass& MyClass::operator=(const MyClass &rhs) {
  // 1.  Deallocate any memory that MyClass is using internally
  // 2.  Allocate some memory to hold the contents of rhs
  // 3.  Copy the values from rhs into this instance
  // 4.  Return *this
}
```

- So what happens if you then do something like this…

```
MyClass mc;
mc = mc; // oops
```

Guarding Self-Assignment

- Fortunately it is easy to avoid this.
- A correct version of an assignment overload would be something like as follows…

```
MyClass& MyClass::operator=(const MyClass &rhs) {
  // Only do assignment if RHS is a different object from this.
  if (this != &rhs) {
    ... // Deallocate, allocate new space, copy values...
  }
  return *this;
}
```

Overloading Dereferencing

- On line 11 we overload the dereference operator.
- Just dereferences the wrapped pointer and returns the value.
- A convenience.

```
1  // ...
2
3  Ptr& operator=(const Ptr& right) {
4    trace("assignment operator");
5    if (addr != right.addr) {
6      addr = right.addr;
7    }
8    return *this;
9  }
10
11 T& operator*() const {
12   trace("dereference operator");
13   return *addr;
14 }
15
16 T* operator->() const {
17   trace("arrow operator");
18   return addr;
19 }
```

Overloading Arrow

- On line 16 we overload the arrow operator.

```
1  // ...
2
3  Ptr& operator=(const Ptr& right) {
4    trace("assignment operator");
5    if (addr != right.addr) {
6      addr = right.addr;
7    }
8    return *this;
9  }
10
11 T& operator*() const {
12   trace("dereference operator");
13   return *addr;
14 }
15
16 T* operator->() const {
17   trace("arrow operator");
18   return addr;
19 }
```

Overloading Arrow

- Kind of odd that it just returns the pointer.
- Doesn't that mean we will require two deferences?
- Something like….

```
__rt::Ptr<String> foo(new String("foo"));
cout << (foo->)->data << endl;
```

```
// ...

Ptr& operator=(const Ptr& right) {
  trace("assignment operator");
  if (addr != right.addr) {
    addr = right.addr;
  }
  return *this;
}

T& operator*() const {
  trace("dereference operator");
  return *addr;
}

T* operator->() const {
  trace("arrow operator");
  return addr;
}
```

Overloading Arrow

- The operator-> has special semantics in that, when overloaded, it reapplies itself to the result.
- While the rest of the operators are applied only once, operator-> will be applied by the compiler as many times as needed to get to a raw pointer and once more to access the memory referred by that pointer.
- Don’t you just love C++?

```
// ...

Ptr& operator=(const Ptr& right) {
  trace("assignment operator");
  if (addr != right.addr) {
    addr = right.addr;
  }
  return *this;
}

T& operator*() const {
  trace("dereference operator");
  return *addr;
}

T* operator->() const {
  trace("arrow operator");
  return addr;
}
```

As usual..

- Repo here
  https://github.com/nyu-oop/java-lang-4
- You must port this code to your translator.
- I have included a CHANGES.md file that points out the specific additions. Just copy and paste those bits.

Translator Timeline

- About 6 weeks until final project presentations
- Features to implement:
    - Constructor translation
    - Array translation
    - Method Overloading
    - Memory management

- Right now you should be wrapping up the midterm requirements and starting arrays.
- More inputs coming in the next few days, but nothing stopping you from making your own.