Object-Oriented Programming

CSCI-UA 0470-001
Class 10
Instructor: Randy Shepherd

In-class exercise

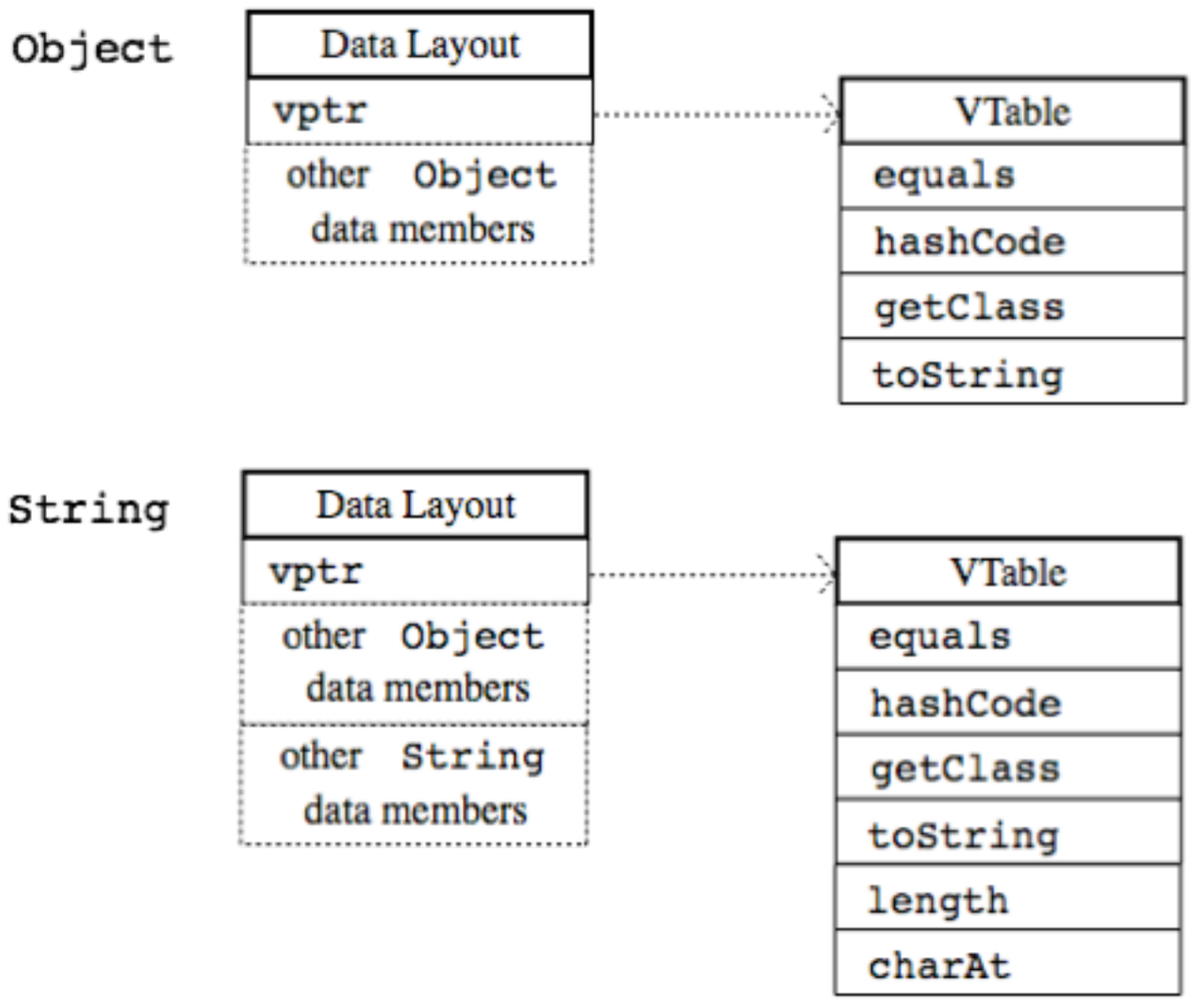
- Translated "by-hand" some Java to C++
- We had a ‘library’ of code to to build upon to implement our classes A and B (*java-lang.*)*
- Demonstrated how we can implement inheritance and virtual method dispatch *without* using the OOP features of C++
- The point of this approach is to understand how a compiler might do the same thing.

```java
package inputs.javalang;

class A {
    public int method() { return 12345;}
    public String toString() {
        return "A";
    }
}

class B extends A {
    public String toString() { return "B"; }
}

public class Input {
    public static void main(String[] args) {
        B b = new B();
        A a1 = new A();
        A a2 = b;

        System.out.println(a1.toString());
        System.out.println(a2.toString());

        System.out.println(a1.method());
        System.out.println(b.method());

        Class ca1 = a1.getClass();
        System.out.println(ca1.toString());

        Class ca2 = a2.getClass();
        System.out.println(ca2.toString());
    }
}
```

Review

Method Layout of Class Inheritance

- Every object has a pointer pointing to a vtable
- Every ‘subclass’ either overrides existing methods or introduces new methods in its *vtable*.
- New methods added by a ‘subclass’ add to the *vtable* of the ‘superclass’, adding new methods at the end.
- Any ‘overriden' methods go into an existing slot.



Solution

Our Types

```
#include "java_lang.h"

using namespace java::lang;

namespace inputs {
  namespace javalang {

    struct __A;
    struct __A_VT;

    struct __B;
    struct __B_VT;

    typedef __A* A;
    typedef __B* B;

    // ....
```

Declaration of A

```
1  struct __A {
2    __A_VT* __vptr;
3
4    __A();
5
6    static String toString(A);
7    static int32_t method(A);
8
9    static Class __class();
10
11   static __A_VT __vtable;
12 };
13
14 struct __A_VT {
15   Class __isa;
16
17   int32_t (*hashCode)(A);
18   bool (*equals)(A, Object);
19   Class (*getClass)(A);
20   String (*toString)(A);
21   int32_t (*method)(A);
22
23   __A_VT()
24   : __isa(__A::__class()),
25     hashCode((int32_t(*)(A)) &__Object::hashCode),
26     equals((bool(*)(A,Object)) &__Object::equals),
27     getClass((Class(*)(A)) &__Object::getClass),
28     toString(&__A::toString),
29     method(&__A::method) {
30   }
31 };
```

Declaration of B

```
1  struct __B {
2    __B_VT* __vptr;
3
4    __B();
5
6    static String toString(B);
7
8    static Class __class();
9
10   static __B_VT __vtable;
11 };
12
13 struct __B_VT {
14   Class __isa;
15
16   int32_t (*hashCode)(B);
17   bool (*equals)(B, Object);
18   Class (*getClass)(B);
19   String (*toString)(B);
20   int32_t (*method)(B);
21
22   __B_VT()
23   : __isa(__B::__class()),
24     hashCode((int32_t(*)(B)) &__Object::hashCode),
25     equals((bool(*)(B,Object)) &__Object::equals),
26     getClass((Class(*)(B)) &__Object::getClass),
27     toString(&__B::toString),
28     method((int32_t(*)(B)) &__A::method) {
29   }
30 };
```

Implementation of A

```
1  __A::__A() : __vptr(&__vtable) {}
2
3  String __A::toString(A __this) {
4    return new __String("A");
5  }
6
7  int32_t __A::method(A __this) {
8    return (int32_t) 12345;
9  }
10
11 Class __A::__class() {
12   static Class k =
13     new __Class(__rt::literal("class inputs.javalang.A"),
14       (Class) __rt::null());
15   return k;
16 }
17
18 __A_VT __A::__vtable;
```

Implementation of B

```
1  __B::__B() : __vptr(&__vtable) {}
2
3  String __B::toString(B __this) {
4    return new __String("B");
5  }
6
7  Class __B::__class() {
8    static Class k =
9      new __Class(__rt::literal("class inputs.javalang.B"),
10       (Class) __rt::null());
11   return k;
12 }
13
14 __B_VT __B::__vtable;
```

Main Class

```
#include <iostream>

#include "java_lang.h"

#include "output.h"

using namespace java::lang;
using namespace inputs::javalang;
using namespace std;

int main(void)
{
  B b = new __B();
  A a1 = new __A();
  A a2 = (A) b;

  cout << a1->__vptr->toString(a1)->data << endl;
  cout << a2->__vptr->toString(a2)->data << endl;

  cout << a1->__vptr->method(a1) << endl;
  cout << b->__vptr->method(b) << endl;

  Class ca1 = a1->__vptr->getClass(a1);
  cout << ca1->__vptr->getName(ca1)->data << endl;

  Class ca2 = a1->__vptr->getClass(a2);
  cout << ca2->__vptr->getName(ca2)->data << endl;
}
```

Key to success

- Understand the vtable concept
- Understand how a class can reuse methods from other classes without actually inheriting
- Read the java-lang-1 repository's main.cpp and see how we can use the classes and the table
- Apply these understandings to hw3

Follow-up Questions

- How would you implement the case where a subclass method accesses a member of the superclass?
- Why does ‘upcasting’ work?
  - i.e. Object obj = new __String(“?”);
- How would you implement static and private methods?

Xtc Features

Xtc has many…

- A few things which we will want to familiarize ourself with..
  - Tool.java & Runtime.java (in xtc.util)
  - Visitor.java (in xtc.tree)
    - Node.java and GNode.java (in xtc.tree)
    - Printer.java (in xtc.tree)
    - CPrinter.java (in xtc.lang)
    - SymbolTable.java (in xtc.util)
- All of these can be found in the repo https://github.com/nyu-oop-fall16/xtc

Tool & Runtime

- **Tool**
  - We've seen Tool in use in our TeamExerciseVisitor in-class exercise and hw2.
  - It is an entry point to a program that uses Xtc.
  - It configures the user interface, defining the set of valid commands and provides feedback to the user about their inputs
- **Runtime**
  - A helper class that processes command line options, prints errors and warnings, and *manages console output*.
  - *Note*: Don't forget to flush the buffer on the console!

Node & GNode

- Node
  - A node in an abstract syntax tree.
- GNode
  - A generic node in an abstract syntax tree.
- We mostly concern ourselves with GNodes. Though we use Nodes in our recursive visit() fallback method.
- We must familiarize ourselves with how to create them as well.
  - https://github.com/nyu-oop/xtc/blob/master/src/xtc/tree/GNode.java#L939

Visitors

- The Xtc mechanism for traversing an Ast (Nodes)
- The key method of the visitor class is “dispatch"
- The contract is as follows:
  - if the dispatched node is a dynamically typed GNode with name NAME then find the method "visitNAME" and call it with the dispatched node.
  - if the dispatched node has not method to handle it and is some type that extends Node, then find the "visit" method that takes that node and calls dispatch on all its children (depth first behavior!)

Printer

- A built-in mechanism for printing source code while traversing an Ast.
- Tightly coupled with Visitor.
- You can find demo code the xtc-demo repository https://github.com/nyu-oop-fall16/xtc-demo/blob/master/src/main/java/edu/nyu/oop/CppFilePrinter.java

Construction & Registration

```
1  public class PrinterExample extends Visitor {
2      private xtc.tree.Printer printer;
3
4      public PrinterExample() {
5          FileOutputStream fos = new FileOutputStream("output/main.cc");
6          OutputStreamWriter ows = new OutputStreamWriter(fos, "utf-8");
7          Writer w = new BufferedWriter(ows);
8          this.printer = new Printer(w);
9          printer.register(this);
10     }
11
12     //...
13 }
```

- Lines 5-7: Initialize the java.io.Writer for the target file
- Line 8: Instantiate the Printer with the Writer
- Line 9: Register Printer with Visitor. Important.

Basic Usage

- Can then be used inside that visitor to print target source.
- Here we simply print what we know the beginning of each main c++ file must have.
- This particular example would be done prior to visiting nodes in the Ast.

```
1  printer.pln("#include <iostream>");
2  printer.pln("#include \"java_lang.h\"");
3  printer.pln();
4  printer.pln("using namespace java::lang;");
5  printer.pln("using namespace std;");
6  printer.pln();
7  printer.pln("int main(void) {");
```

Usage in Visit Methods

```
1  public void visit(Node n) {
2      printer.incr().indent().pln("cout << \"" + n.getName() + "\" << endl;").decr();
3      for (Object o : n) if (o instanceof Node) dispatch((Node) o);
4  }
```

- Can be used inside any visit method.
- Structure of tree is maintained w.r.t. to output.
- Note the incr() and indent() methods.
    - These will come in handy, but *astyle* with the *format* sbt command will be even more effective.
- You'll want to explore what other methods it has.
- What does this code do?

Printer Output

- Emits valid and executable C++ code
- Note that the order of the names of the nodes correlates with what appears to the be order they appear in the Java source file.

```cpp
#include <iostream>
#include "java_lang.h"

using namespace java::lang;
using namespace std;

int main(void)
{
    cout << "CompilationUnit" << endl;
    cout << "PackageDeclaration" << endl;
    cout << "QualifiedIdentifier" << endl;
    cout << "ClassDeclaration" << endl;
    cout << "Modifiers" << endl;
    cout << "Modifier" << endl;
    cout << "ClassBody" << endl;
    cout << "MethodDeclaration" << endl;
    cout << "Modifiers" << endl;
    cout << "Modifier" << endl;
    cout << "Modifier" << endl;
    cout << "VoidType" << endl;
    cout << "FormalParameters" << endl;
    cout << "FormalParameter" << endl;
    cout << "Modifiers" << endl;
    cout << "Type" << endl;
    cout << "QualifiedIdentifier" << endl;
    cout << "Dimensions" << endl;
    cout << "Block" << endl;
    cout << "ExpressionStatement" << endl;
    cout << "CallExpression" << endl;
    cout << "SelectionExpression" << endl;
    cout << "PrimaryIdentifier" << endl;
    cout << "Arguments" << endl;
    cout << "StringLiteral" << endl;
    return 0;
}
```

CPrinter

- A pretty printer for the C programming language
- Supports entire language, 2000 lines long(!!)
- Offers insight as to how to do printing
  - Ex. If-else printing for C https://github.com/nyu-oop-fall16/xtc/blob/master/src/xtc/lang/CPrinter.java#L1219

Symbol Table

- A symbol table is a data structure used by a language translator where each identifier in a program's source code is associated with information relating to its type and scope.

- For example…

```
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double  sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1;  i <= count;  i++)
        sum += bar((double) i);
    return sum;
}
```

Symbol name	Type	Scope
`bar`	function, double	extern
`x`	double	function parameter
`foo`	function, double	global
`count`	int	function parameter
`sum`	double	block local
`i`	int	for-loop statement

Symbol Table

- You can get away without one for a while.
- Once we introduce method overloading, it will absolutely be necessary.
- Xtc provides one, but again, you'll need to learn how to use it.
- This is just getting the idea on your radar. We will talk more about this in a future class.

Code Style

Correctness is not sufficient

- In some types of work, its simply important to get the right answer.
- In software engineering, this is not enough. Writing software is part science and part craft!
- Part of the motivation for OOP
- This is especially true when working in teams.

Programming languages are for humans

- The reason high-level programming languages were developed was to make working with computers easier for *humans*.
- Not only when writing, but, more importantly when reading!
- Programs are ‘write once, read many’

Think of your readers

- “Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.”
  - *Jeff Atwood* http://www.codinghorror.com/blog/2008/06/coding-for-violent-psychopaths.html

- *Style* has a significant impact on the readability and maintainability of your code

What is 'Good Style'?

- It is somewhat subjective, and is therefore difficult to define succinctly.

- The following are usually considered as part of style:

→ layout & indentation

→ thoughtful, consistent naming

→ useful comments

→ sensible modules

Layout & Indentation

- Which is easier to read?

- Simply indenting your code in a sensible and consistent way can improve readability immensely

- IntelliJ and Sbt provide ways for you to quickly and painlessly format your code.

- Easy win!

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
} else {
    return false;
}
```

```
if  ( hours   < 24
   && minutes < 60
   && seconds < 60
)
{return    true
;}         else
{return   false
;}
```

Naming

- First and foremost, for our class we use camelCase for all names.
- Bad names

  → int tmp;

  → getInfo()

  → DataClass
- Good names

  → int methodCounter;

  → findChildNode()

  → VirtualTableGenerator

Comments

- Good comments aid maintainability, what are good comments?
  - None needed. (Your code is self-documenting!)
  - Don't write 'what' code is doing. Write 'why'.
  - Get to the point. Don't write your life story.
  - Alert readers to important code blocks.

Sensible Modules

- What are 'sensible modules’?
- Each language is different. As an example, in Java:

  → If a class file exceeds 200 lines of code your terror alert status should be at ‘elevated’.

  → As it passes 350, it should raise to ‘high'.

  → Past 500 the terror alert is ‘severe’.

- As the number of lines grows, the probability that you are conflating responsibilities of what should be separate classes does too.

Key Points

- Code should easy to read and neat.
- Code should be consistently formatted and organized intelligently.
- Things should be named thoughtfully.
- Code should ideally be self-documenting.

Key Points

- Code should be
  - easy to read and neat
  - consistently formatted
  - organized intelligently
  - named thoughtfully
  - self-documenting

Speaker Meeting