Object-Oriented Programming

CSCI-UA 0470-001
Class 21
Instructor: Randy Shepherd

Reference Vs Value Semantics

Java Vs C++

- Consider these simplified Point implementations in C++ and Java
- The look very similar….

```java
public class Point {
  private double x;
  private double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public void moveX(double dist) { x += dist; }
  // ...
}
```

```cpp
class Point {
private:
    double x;
    double y;

public:
    Point(double x, double y) : x(x), y(y) { }

    void moveX(double dist) { x += dist; }

    //...
};
```

Java Vs C++

- Now consider some code using Point objects.
- What gets printed from line 5 of the Java code?

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a.moveX(8);
5  System.out.print(
6    b.getX() + "," + b.getY()
7  );
```

```
1  Point a = Point(7,5),
2  Point b = Point(0,1);
3  b = a;
4  a.moveX(8);
5  cout << b.getX() << "," << b.getY();
```

Java Vs C++

- Now consider some code using Point objects.

- What gets printed from line 5 of the Java code?

- 15,5

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a.moveX(8);
5  System.out.print(
6    b.getX() + "," + b.getY()
7  );
```

```
1  Point a = Point(7,5),
2  Point b = Point(0,1);
3  b = a;
4  a.moveX(8);
5  cout << b.getX() << "," << b.getY();
```

Java Vs C++

- Now consider some code using Point objects.

- What gets printed from line 5 of the Java code?

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a.moveX(8);
5  System.out.print(
6    b.getX() + "," + b.getY()
7  );
```

  - 15,5

- What gets printed from line 5 of the C++ code?

```
1  Point a = Point(7,5),
2  Point b = Point(0,1);
3  b = a;
4  a.moveX(8);
5  cout << b.getX() << "," << b.getY();
```

Java Vs C++

- Now consider some code using Point objects.

- What gets printed from line 5 of the Java code?

  - 15,5

- What gets printed from line 5 of the C++ code?

  - 7,5

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a.moveX(8);
5  System.out.print(
6    b.getX() + "," + b.getY()
7  );
```

```
1  Point a = Point(7,5),
2  Point b = Point(0,1);
3  b = a;
4  a.moveX(8);
5  cout << b.getX() << "," << b.getY();
```

Java Vs C++

- Why? For objects..
  - Java has '*reference*' semantics
  - ('reference' in this case actually means pointer.)
- C++ has *value* semantics

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a.moveX(8);
5  System.out.print(
6    b.getX() + "," + b.getY()
7  );
```

```
1  Point a = Point(7,5),
2  Point b = Point(0,1);
3  b = a;
4  a.moveX(8);
5  cout << b.getX() << "," << b.getY();
```

Java Vs C++

- In the Java code, line 3 copies the pointer to ‘a’ to ‘b’.
- After the assignment, ‘a’ and ‘b’ refer to the same instance of point on the heap.
- Therefore, changes to either ‘a’ or ‘b’ will be visible by dereferencing either of them.

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a.moveX(8);
5  System.out.print(
6    b.getX() + "," + b.getY()
7  );
```

```
1  Point a = Point(7,5),
2  Point b = Point(0,1);
3  b = a;
4  a.moveX(8);
5  cout << b.getX() << "," << b.getY();
```

Java Vs C++

- In the C++ code, line 3 copies the *values* of ‘a’ into ‘b’
- So after the assignment, ‘a’ and ‘b’ hold two different copies of a Point that have the same values.
- Therefore changes to either are not reflected in the other.

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a.moveX(8);
5  System.out.print(
6    b.getX() + "," + b.getY()
7  );
```

```
1  Point a = Point(7,5),
2  Point b = Point(0,1);
3  b = a;
4  a.moveX(8);
5  cout << b.getX() << "," << b.getY();
```

Value Semantics

- If you are a Java programmer this should look familiar, primitive types in Java use value semantics.

- Moreover, modifying the value of one variable does not affect other.

- ex.

```
1  int x = 1;
2  int y = x;
3  x++;
4  System.out.println(y);
5  // y still == 1
```

‘Reference’ Semantics

- Object types in Java use ‘reference’ semantics.
- Object variables do not store an object; they store the address of an object's location in memory.
- We can have this behavior in C++ also.
- ex.

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a->moveX(8);
5  cout << b->getX() << "," << b->getY();
```

‘Reference’ Semantics

- Whoops. There is a problem with this code.
- Consider carefully what happens on lines 2 and 3.

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a->moveX(8);
5  cout << b->getX() << "," << b->getY();
```

‘Reference’ Semantics

- Whoops. There is a problem with this code.
- Consider carefully what happens on lines 2 and 3.

```
1  Point a = new Point(7,5),
2  Point b = new Point(0,1);
3  b = a;
4  a->moveX(8);
5  cout << b->getX() << "," << b->getY();
```

- Line 2 creates an object and on line 3 we lose any reference to it without having deallocated!
- We’ve just leaked memory.

Why ‘Reference’ Semantics?

- Efficiency
    - Objects can have a relatively large memory footprint. Having to copy them every time they are passed as parameters would be slow.
- Sharing
    - Since objects can hold important program state, it is often desirable for them to be shared by parts of the program when they're passed as parameters.

Why Value Semantics?

- You can manipulate variables without worrying about other references to it.

- Operations are localized, which *always* leads to code that is easier to reason about.

- Some languages *only* have value semantics for this reason.

Strings

- Strings in many programming language have reference semantics.

- Since the program can’t know the size of a string, value the string cannot be allocated to stack.

- The String is put on heap, therefore a string is usually modeled as a reference type.

String with Value Semantics

- To get a deeper understanding, we'll write a C++ class that *encapsulates* a string with value semantics.

- First, though, we need to know the 'Rule of Three'.

Rule of Three

- The rule of three is a rule of thumb in C++ (prior to C++11) that claims that if a class defines one of the following it should define all three:
    - destructor
    - copy constructor
    - copy assignment operator

Rule of Three

- If one of the following is not declared by the programmer it will be created by the compiler with the default semantics.

- The default semantics are:
  - *Destructor* – Call the destructors of all the object's class-type members
  - *Copy constructor* – Construct all the object's members from the corresponding members of the copy constructor's argument.
  - *Copy assignment operator* – Assign all the object's members from the corresponding members of the assignment operator's argument.

Rule of Three

- If one of these had to be defined, it means that the compiler-generated versions likely do not fit the needs of the class.

- Because generated constructors and assignment operators simply copy all class data members, define explicit copy constructors and copy assignment operators for classes that have external references (without it only the ‘reference’ gets copied, not the object it points to).

stringval

- For our stringval we'll wrap a c-style string along with the length

- And in order to achieve value semantics we'll need to follow the 'Rule of Three'

```
1  class stringval {
2    // Encapsulate a c-style array
3    size_t len;
4    char *data;
5  public:
6    // Regular constructors
7    stringval(const char *s);
8    stringval(const size_t len);
9
10   // Copy constructor
11   stringval(const stringval &other);
12
13   // Destructor
14   ~stringval();
15
16   // Assignment operator
17   stringval &operator=(const stringval &other);
18 };
```

Copy Constructor

- Note the copy constructor at line 11
- This is required in order to pass stringval instances as parameters to a function.
- Moreover, when a stringval is passed as an argument, the copy constructor is used to copy the values of the object into the function parameters.

```
1  class stringval {
2    // Encapsulate a c-style array
3    size_t len;
4    char *data;
5  public:
6    // Regular constructors
7    stringval(const char *s);
8    stringval(const size_t len);
9
10   // Copy constructor
11   stringval(const stringval &other);
12
13   // Destructor
14   ~stringval();
15
16   // Assignment operator
17   stringval &operator=(const stringval &other);
18 };
```

```
1  // in stringval.cpp
2  stringval::stringval(const stringval &other)
3    : len(other.len), data(new char[len]) {
4    memcpy(data, other.data, len);
5  }
```

Destructor

- Note the destructor on line 14
- We need to clean up the heap when an instance is deallocated or the stack frame is popped.
- Specifically, we need to delete the c-string.
- (The compiler-generated destructor for this class would not call delete on the pointer!)

```
1  class stringval {
2    // Encapsulate a c-style array
3    size_t len;
4    char *data;
5  public:
6    // Regular constructors
7    stringval(const char *s);
8    stringval(const size_t len);
9
10   // Copy constructor
11   stringval(const stringval &other);
12
13   // Destructor
14   ~stringval();
15
16   // Assignment operator
17   stringval &operator=(const stringval &other);
18 };
```

```
1  // in stringval.cpp
2  stringval::~stringval() {
3    delete[] data;
4  }
```

Assignment Operator

- Note the assignment operator on line 17
- When we assign a value want to do a *deep copy* of any instance variables that have external containment (e.g. are pointers).
- (The compiler-generated assignment operator for this class would only do a *shallow copy* of pointer members)

```
1  class stringval {
2    // Encapsulate a c-style array
3    size_t len;
4    char *data;
5  public:
6    // Regular constructors
7    stringval(const char *s);
8    stringval(const size_t len);
9
10   // Copy constructor
11   stringval(const stringval &other);
12
13   // Destructor
14   ~stringval();
15
16   // Assignment operator
17   stringval &operator=(const stringval &other);
18 };
```

```
1  // in stringval.cpp
2  stringval &stringval::operator=(stringval &other) {
3    if (data != other.data) {
4      len = other.len;
5      data = new char[len];
6      memcpy(data, other.data, len);
7    }
8    return *this;
9  }
```

Assignment Operator

- Implementation:
  - Line 3: Check to make sure data and other.data are not identical. Otherwise the C-string will be deleted before it is copied.
  - Line 5: Allocate a new char[] of appropriate length before copying
  - Line 6: Copy contents of string into member

(I feel like I am forgetting something)

```
1  class stringval {
2    // Encapsulate a c-style array
3    size_t len;
4    char *data;
5  public:
6    // Regular constructors
7    stringval(const char *s);
8    stringval(const size_t len);
9
10   // Copy constructor
11   stringval(const stringval &other);
12
13   // Destructor
14   ~stringval();
15
16   // Assignment operator
17   stringval &operator=(const stringval &other);
18 };
```

```
1  // in stringval.cpp
2  stringval &stringval::operator=(stringval &other) {
3    if (data != other.data) {
4      len = other.len;
5      data = new char[len];
6      memcpy(data, other.data, len);
7    }
8    return *this;
9  }
```

Tracing

- Inside our implementation we've implemented some debugging output.
- Each method calls a macro TRACE that prints the line number and function name.
- ex lines 3, 8, 14 and 19

```
1  stringval::stringval(const size_t len)
2    : len(len), data(new char[len]) {
3    TRACE("Constructor(size_t)", "");
4  }
5
6  stringval::stringval(const char *s)
7    : len(strlen(s)), data(new char[len]) {
8    TRACE("Constructor(char*)", "");
9    memcpy(data, s, len);
10 }
11
12 stringval::stringval(const stringval &other)
13   : len(other.len), data(new char[len]) {
14   TRACE("Copy Constructor", "");
15   memcpy(data, other.data, len);
16 }
17
18 stringval::~stringval() {
19   TRACE("Destructor", "");
20   delete[] data;
21 }
```

Main Function

- We can now see how values are copied by reading the debug output.

- For example, for lines 2 & 3, the following would print

stringval:11 Constructor(char*)
main:3 s2 = World

```
1  // Instantiate a string with value semantics
2  stringval s2(" World");
3  TRACE("s2 = ", s2);
4
5  // Instantiate a string with value semantics
6  stringval s3(" ");
7  TRACE("s3 = ", s3);
8
9  // Copy the values of s2 into s3
10 s3 = s2;
11 TRACE("s3 = ", s3);
12
13 // Mutate s2, and print them both,
14 // changes not reflected in s3
15 s2[0] = '?';
16 TRACE("s2 = ", s2);
17 TRACE("s3 = ", s3);
18
19 // Pass s3 to a function
20 f(s3);
```

Main Function

- At lines 10 and 11, the following would print

operator=:27 Assignment operator
main:11 s3 = World

- At lines 16 and 17, the following would print

main:16 s2 = ?World
main:17 s3 = World

```
1  // Instantiate a string with value semantics
2  stringval s2(" World");
3  TRACE("s2 = ", s2);
4
5  // Instantiate a string with value semantics
6  stringval s3(" ");
7  TRACE("s3 = ", s3);
8
9  // Copy the values of s2 into s3
10 s3 = s2;
11 TRACE("s3 = ", s3);
12
13 // Mutate s2, and print them both,
14 // changes not reflected in s3
15 s2[0] = '?';
16 TRACE("s2 = ", s2);
17 TRACE("s3 = ", s3);
18
19 // Pass s3 to a function
20 f(s3);
```

Main Function

- At line 20, a function ‘f’ is invoked, it is defined:

```
void f(stringval s) {
  TRACE("s = ", s);
}
```

- What will be printed?

```
1  // Instantiate a string with value semantics
2  stringval s2(" World");
3  TRACE("s2 = ", s2);
4
5  // Instantiate a string with value semantics
6  stringval s3(" ");
7  TRACE("s3 = ", s3);
8
9  // Copy the values of s2 into s3
10 s3 = s2;
11 TRACE("s3 = ", s3);
12
13 // Mutate s2, and print them both,
14 // changes not reflected in s3
15 s2[0] = '?';
16 TRACE("s2 = ", s2);
17 TRACE("s3 = ", s3);
18
19 // Pass s3 to a function
20 f(s3);
```

Main Function

- At line 20, a function ‘f’ is called, it is defined:

```
void f(stringval s) {
  TRACE("s = ", s);
}
```

- What will be printed?

stringval:17 Copy Constructor
f:8 s = World
~stringval:22 Destructor

```
1  // Instantiate a string with value semantics
2  stringval s2(" World");
3  TRACE("s2 = ", s2);
4
5  // Instantiate a string with value semantics
6  stringval s3(" ");
7  TRACE("s3 = ", s3);
8
9  // Copy the values of s2 into s3
10 s3 = s2;
11 TRACE("s3 = ", s3);
12
13 // Mutate s2, and print them both,
14 // changes not reflected in s3
15 s2[0] = '?';
16 TRACE("s2 = ", s2);
17 TRACE("s3 = ", s3);
18
19 // Pass s3 to a function
20 f(s3);
```

Main Function

- Line 20 completes and the stack frame for main is popped, what prints then?

```
1  // Instantiate a string with value semantics
2  stringval s2(" World");
3  TRACE("s2 = ", s2);
4
5  // Instantiate a string with value semantics
6  stringval s3(" ");
7  TRACE("s3 = ", s3);
8
9  // Copy the values of s2 into s3
10 s3 = s2;
11 TRACE("s3 = ", s3);
12
13 // Mutate s2, and print them both,
14 // changes not reflected in s3
15 s2[0] = '?';
16 TRACE("s2 = ", s2);
17 TRACE("s3 = ", s3);
18
19 // Pass s3 to a function
20 f(s3);
```

Main Function

- Line 20 completes and the stack frame for main is popped, what prints then?

~stringval:22 Destructor
~stringval:22 Destructor
~stringval:22 Destructor
~stringval:22 Destructor

- Code can be found here
https://github.com/nyu-oop-fall16/value-vs-reference-cpp

```
1  // Instantiate a string with value semantics
2  stringval s2(" World");
3  TRACE("s2 = ", s2);
4
5  // Instantiate a string with value semantics
6  stringval s3(" ");
7  TRACE("s3 = ", s3);
8
9  // Copy the values of s2 into s3
10 s3 = s2;
11 TRACE("s3 = ", s3);
12
13 // Mutate s2, and print them both,
14 // changes not reflected in s3
15 s2[0] = '?';
16 TRACE("s2 = ", s2);
17 TRACE("s3 = ", s3);
18
19 // Pass s3 to a function
20 f(s3);
```

Take-away

- Due to C++'s value semantics of objects, we often pass around pointers.
- This means we need to make sure to clean up after ourselves.
- In order to do that you must have a clear understanding of the rules of the language.

Manual Memory Management

- We had a memory leak in that code, did you notice? (slide 26, in the implementation of the assignment operator)

- Memory management can be tricky when done ‘by hand’.

- There’s got to be a better way!

Garbage Collection

Memory Management Strategies

- The heap is finite, we must clean up after ourselves.
- Three options
  - Manual deallocation e.g. free, delete (C, Pascal)
  - Semi-automatic deallocation, using destructors (C++)
  - Automatic deallocation via garbage collection (Java, Ruby)
- Manual and semi-automatic deallocation are dangerous
  - not all current references to an object may be visible
  - high potential for human error

Basic Garbage Collection Algorithms

- *mark/sweep* – ‘automatic’, needs run-time support
- *copying* – ‘automatic’, needs run-time support
  - variant: generational
- *reference counting* – ‘semi-automatic’, usually done by programmer

Automatic Garbage Collection

What is Automatic Garbage Collection?

- Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

- An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

‘Live’ Objects

- An Object x is *live* (i.e., can be referenced) if x is:
  - pointed to by some variable located on the stack
  - in static memory
  - referred to by a value present in a register
  - is referred to by another object on the heap that is live

‘Live’ Objects

- All live objects in the heap can be found by a graph traversal
  - Traversal start at the roots; variables on the stack, static memory, registers.
  - Any object not reachable from the roots is dead and can be collected.

Mark & Sweep

- Each object has an extra bit called the 'mark bit'
- 'mark phase'
  - the collector traverses the heap object graph and sets the mark bit of each object encountered
- 'sweep phase'
  - each object whose mark bit is not set is collected

Copying

- The heap is split into two parts: FROM space and TO space
- Objects are allocated in the FROM space
- When FROM is full, collection begins
    - During traversal, each encountered object is copied to TO space
    - When traversal is done, all live objects in are in TO space
- Now the spaces are flipped, FROM becomes TO and vice versa

Generational GC

- A variant of copying. The JVM uses this technique (among others).
- Observation: the older an object gets the longer it is expected to stay around. Why?
    - Objects that live for a long time tend to make up core program data structures and will probably live until the end of the programs life.
- It turns out that the vast majority of objects in typical object-oriented programs (between 92 and 98 percent according to various studies), die young.

Generational GC

- Generational GCs exploit this ‘generational hypothesis'
- Instead of just two heaps (FROM and TO), we have several signifying ‘generations’ of objects.
  - younger generations collected more frequently than older generations (because younger generations will have more garbage to collect)
  - when a generation is traversed, live objects are copied to the next-older generation
  - when a generation fills up, it is garbage collected

Urban Performance Legends

- "Garbage collection will never be as efficient as direct memory management." - Snooty C++ Programmer

- In a way, those statements are right -- automatic memory management is not *as* fast -- it's often considerably *faster*.

- The new/delete approach deals with blocks of memory one at a time, whereas the garbage collection approach tends to deal with memory management in large batches, yielding more opportunities for optimization.

Urban Performance Legends

- Allocation in JVMs was not always so fast -- early JVMs indeed had poor allocation and garbage collection performance.

- A lot has happened since the JDK 1.0 days; the introduction of generational collectors in JDK 1.2 has greatly improved performance.

- As a result, for most objects, the garbage collection cost is -- zero. This is because a copying collector does not need to visit or copy dead objects, only live ones. So objects that become garbage shortly after allocation contribute no workload to the collection cycle.

Semi-automatic Garbage Collection

Reference Counting

- The problem:
  - we have several references to some data on the heap
    - we want to release memory when there are no more references to it
    - we do not have automatic garbage collection
- Sound familiar?

Reference Counting

- The solution: keep track of how many references point to the object and free it when there are no more.
  - Set reference count to 1 for newly created objects.
  - Increment reference count whenever we copy the pointer to the object.
  - Decrement count when a point to the object goes out of scope or stops pointing to the object.
  - When the count gets to 0, we can free the memory.

Reference Counting

- Advantages:
  - Memory can be reclaimed as soon as no longer needed.
  - Simple, can be done by the programmer for languages not supporting GC.
- Disadvantages:
  - Additional space needed for the reference count.
  - Will not reclaim circular references.

Reference Counting

- Next class we will look at an implementation of a reference counting ‘smart pointer’

- This is how we will manage memory in the translator.