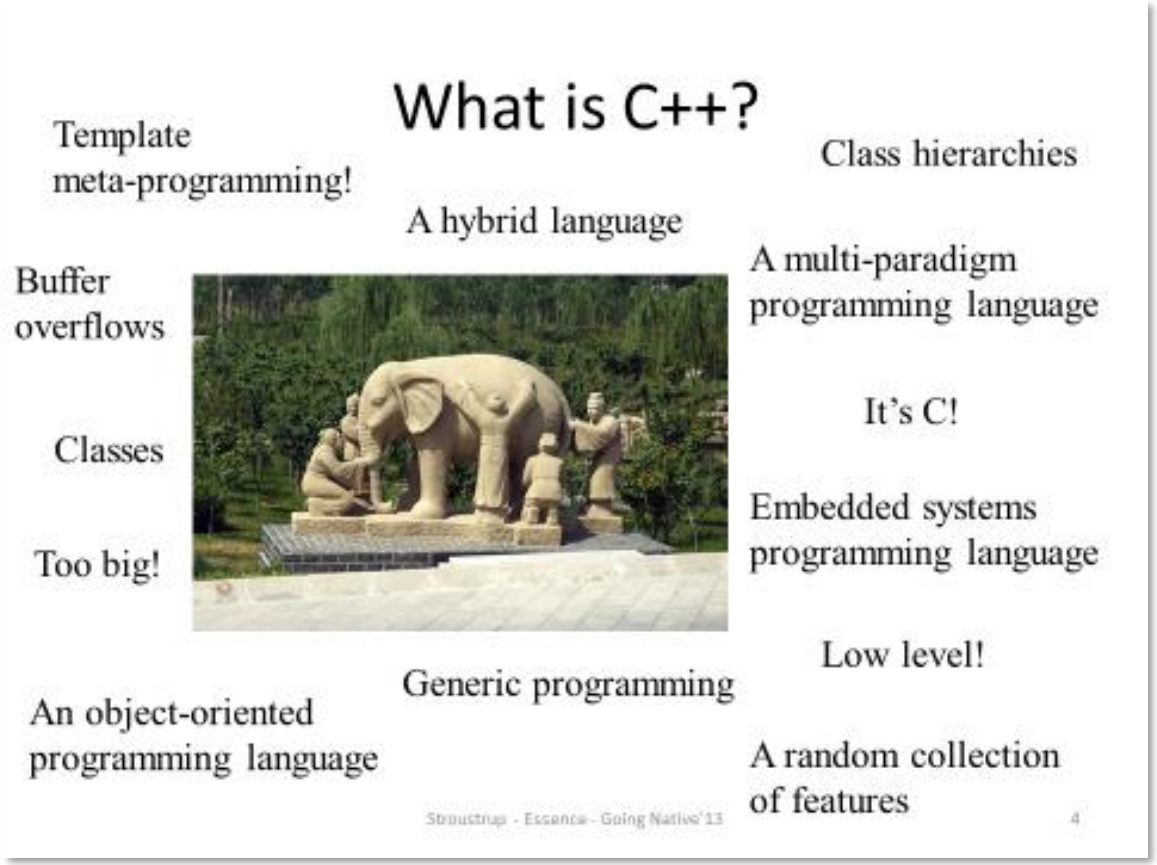
CISC 3142
Programming Paradigms in C++

Introduction

(What is C++, Key differences between Java and C++)

What is C++?

- C++ could be different things for different people
- It's a general purpose, multi-paradigm programming language that's best for:
  - System infrastructure (system programming , game engines, etc)
  - Resource constrained applications (embedded systems)



From: Stroustrup – "The Essence of C++", 2013

The Goals of the Course

- Expose students to ways of programming other than Java
- Introduction to C++ and its mix of programming paradigms
- Compare the differences in design, features of C/C++ and Java
- Likely taking this class alone won’t make you an expert in C++, but hopefully it gets you started on your way if you so desire.

What you're supposed to know already

- Elementary programming techniques
  - Types, variables, operations, expressions, control constructs, etc
- OOP related
  - Classes, objects, methods, constructors, static members
- I/O related
  - Exceptions, input/output streams, files
- Arrays
- Collections and generics
- Data structures in general and their implementations
  - Linked lists, trees, hash tables
- Basic programming techniques and algorithms
  - Sorting, searching, recursion

Evolution of Programming Languages

- Assembly Language (closest to machine code)
- Higher level, capturing domain-specific abstraction
  - Fortran (scientific computation)
  - Cobol (business transactions)
- Higher level, yet very efficient, comparable to Assembler -> C
- Combining C and Simula (classes – general-purpose abstraction) -> C++
- Java and C# were both influenced by C++

Systems Written in C++

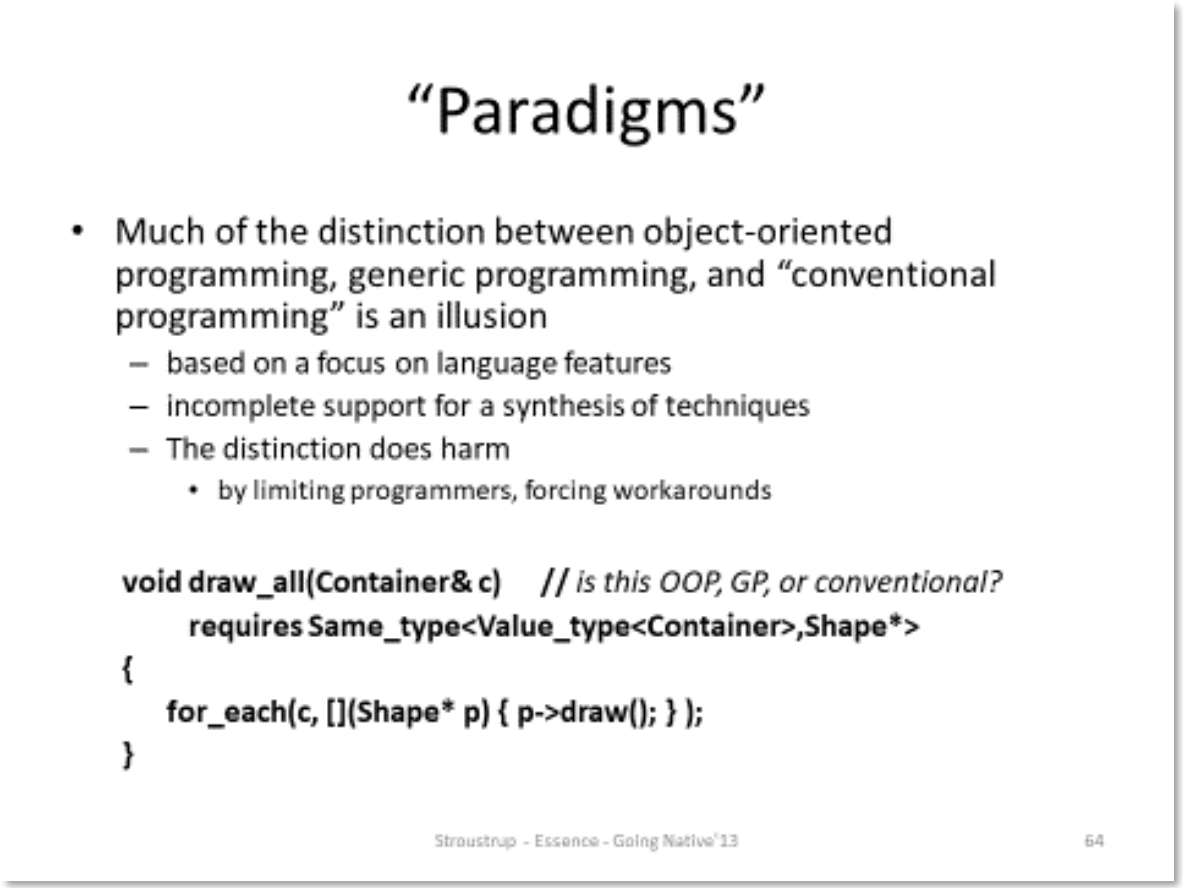
- Games
- GUI based apps (adobe suites)
- MYSQL
- OS (OS X, Windows, in part)
- Browsers
- Embedded Systems (smartwatches, medical equipment, robots, low-level function calls)
- Compilers
- Libraries

Programming Paradigms

- Programming Paradigms are about the unique ways a programming language works based on its features and styles
- Most common choices are
  - Imperative – the program instructs the machine explicitly on how to change its state. Side effects are allowed (e.g., global variables modified by any unit of code)
    - Procedural paradigm – instructions are grouped into procedures
    - Object-oriented paradigm – instructions are grouped together with part of the state they can operate on
  - Declarative – the program states what problem to solve in the form of properties of the desired result, but not how to solve them explicitly
    - Functional paradigm – blocks of code intended to behave like mathematical functions without side effects
    - Logic paradigm – the program does auto reasoning using inference based on a set of rules and facts

Criticism?

- Not everyone is a fan of classifying programs into distinctive paradigms. That includes the author of C++.
  - His view was reflected in his talk on C++ in 2013
  - Many languages have a mix of these programming paradigms, including C++ and Java
- We will look into some of the representative paradigms of C++ - it's in the title of the course after all



"Paradigms"

- Much of the distinction between object-oriented programming, generic programming, and "conventional programming" is an illusion
  - based on a focus on language features
  - incomplete support for a synthesis of techniques
  - The distinction does harm
    - by limiting programmers, forcing workarounds

```
void draw_all(Container& c)    // is this OOP, GP, or conventional?
    requires Same_type<Value_type<Container>,Shape*>
{
    for_each(c, [](Shape* p) { p->draw(); } );
}
```

Stroustrup - Essence - Going Native'13 64

From: Stroustrup – "The Essence of C++", 2013

Fundamental differences between Java/C++

	Java	C++
Compile or interpreted?	Both, compiled into bytecode, then interpreted by JVM	Compiled to object code, then linked into a single executable
Platform independence	Independent, byte code can be run everywhere – written once, compiled once, run everywhere	Dependent, Source code needs recompiling – written once, compiled anywhere
Memory management	System controlled (garbage collection)	Manually via new/delete (triggers dtor)
Multiple inheritance	No, multiple interfaces allowed	Yes, need to avoid the Diamond Problem
Overloading (static polymorphism)	Only for methods, not for operators	Both for methods and operators
Virtual function (dynamic polymorphism)	Implied (default)	Explicit
Pointers/References	References	Both, references must be initialized
Root object?	Pure OOP, all objects derived from Object	OOP was added to C, no root object.
Values vs References: `MyClass myobj;`	Creates a reference on stack. When followed by `new`, the object is created on heap memory	Creates a value, i.e. with memory layout, on stack memory (assuming inside a function)

Difference Examples

- “Hello World” Comparison

Java:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

C++:

```
#include <iostream>
int main(int argc, char *argv[]) {
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

Points worth noting

1) `public` access modifier for class: In java, only one allowed in each source file; in C++, class name has external linkage.
2) Everything in Java is an object (except for primitive data types), thus even `main()` has to be wrapped inside a main class. `main()` is `static` (tied up with the class) as no objects were created yet when it's run
3) `void` return value in Java vs `int` return in C++, for `main()`
4) `args` (Java arrays are objects with bounds (`.length`), and each `String` has `.length()`. In C/C++, arrays are not aware of bounds, so size must be passed in. C-style strings are ended with `'\0'`. C++ `strings` (from class `string`) are objects and have the `.length()` function
5) In Java, class libraries other than `java.lang.*` need importing. Every identifier in C/C++ needs to be declared first, thus even basic I/O objects `cout/endl` need the `#include` (pre-processor directive).

Difference Examples (cont')

- A swap example (swapping two objects)

Java:

```
class Thing {
  String name;
  Thing(String nm) {name = nm;}
}
public class SwapTest {
  public static void swap(Thing tt1, Thing tt2) {
    String temp = tt1.name;
    tt1.name = tt2.name;
    tt2.name = temp;
  }
  public static void main(String[] args) {
    Thing t1 = new Thing("No.1");
    Thing t2 = new Thing("No.2");
    swap(t1, t2);
    System.out.println("t1's name: " + t1.name);
    System.out.println("t2's name: " + t2.name);
  }
}
```

C++:

```
#include <iostream>
#include <string>
using namespace std;
class Thing {
public:
  string name;
  Thing(const string& nm):name(nm){}
};
void swap(Thing& tt1, Thing& tt2) {
  Thing temp = tt1;
  tt1 = tt2;
  tt2 = temp;
}
int main() {
  Thing t1 = Thing("No.1");
  Thing t2 = Thing("No.2");
  swap(t1, t2);
  cout << "t1's name: " << t1.name << endl;
  cout << "t2's name: " << t2.name << endl;
}
```

Points worth noting

- Java: `class {}` definition not ended with a semicolon
- Java: `String name`: `name` is a reference (address). C++: `string name`: `name` is a value (holds all memory occupied by the string). To declare a reference in C++, it's "`string& name`". For a pointer: "`string* pname`"
- Parameter passing in parameterized constructors
  - "`String nm`" vs "`const string& nm`": similar in behavior. In C++, passing `string&` is more efficient than passing `string` – only address is copied, and the `const` keyword allows string literals to be passed
  - C++ can use a member initialization list for a variety of reasons
- In Java, `Things t1/t2`'s contents are created in heap, but local variables `t1/t2` live on stack. In C++, both are entirely created on stack.
- Java really can't swap objects with a method without knowing the internals of `Thing`! This won't work inside `swap()`:

```
Thing temp = tt1; // only makes an alias of the reference, tt1 , so temp now points to where tt1 is pointing at.
tt1 = tt2;
tt2 = temp;
```

- Why?
- What it does is, since parameters are passed by value (value of reference), only the parameters `tt1/tt2` get swapped, arguments `t1/t2` are intact (still pointing to their original data). But in C++, the statement "`Thing temp = tt1;`" actually copies `tt1's` content into `temp` – same to the two ensuing statements.
- Key take away: in C++, for a reference variable: once it's initialized (i.e. getting the address), the ensuing assignments change its content!

Development Steps and Environment

- With Unix/Linux/MacOS
  - Launch terminal
  - Use any text-editor to create .cpp files
  - `$g++ prog.cpp -o prog`
  - `$./prog`
- Under Windows
  - Microsoft Visual C++
    (Visual Studio Community is free)
  - Code::Blocks (open-source IDE)
  - Use your favorite text editor + MinGW
  - Use Cygwin to simulate a Unix working environment

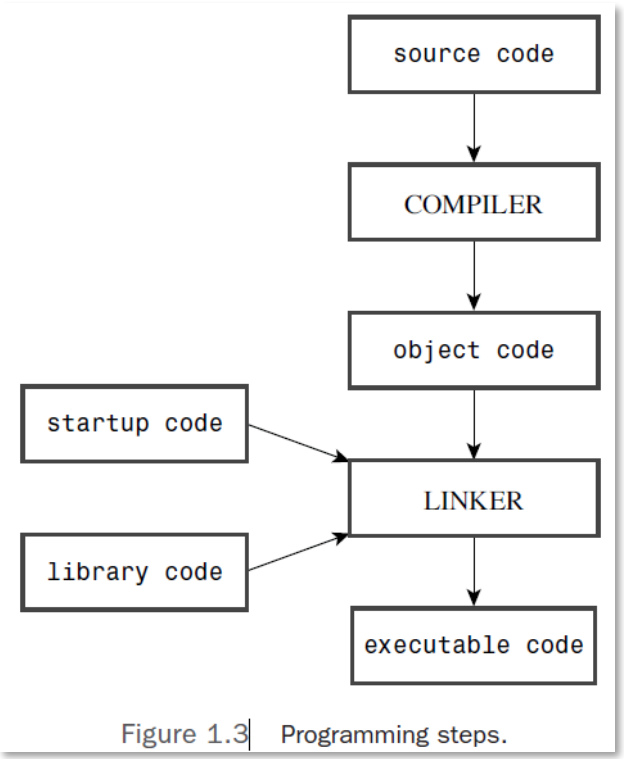


Figure 1.3 Programming steps.

From: Prata – "C++ Primer Plus", 2013

C++ Example – get factors other than 1 & self

```
#include <iostream>
using namespace std;
void reverseArray(int a[], int n);
int main() {
    int num;
    cout << "Enter your integer: ";
    cin >> num;
    int factors[num];// VLA: variable length array
    int j = 0;
    cout << "Factors other than 1 and itself: ";
    bool isFirst = true;
    for (int i=2; i < num; i++) {
        if (num % i == 0) {
            cout << (isFirst ? "" : ", ") << i;
            factors[j++] = i;
            if (isFirst) isFirst = false;
        }
    }
    cout << (isFirst ? "None" : "") << endl;
    if (j>0) {
        reverseArray(factors, j);
        cout << "In descending order: ";
        for (int i=0; i<j; i++)
            cout << factors[i] << (i==j-1 ? "" :
                ", ");
        cout << endl;
    }
}

void reverseArray(int a[], int n) {
    for (int i=0; i< n/2; i++) {
        int temp = a[i];
        a[i] = a[n-i-1];
        a[n-i-1] = temp;
    }
}
```

```
Enter your integer: 48
Factors other than 1 and itself: 2, 3, 4, 6, 8, 12, 16, 24
In descending order: 24, 16, 12, 8, 6, 4, 3, 2
```

Note: this program works but is very inefficient, you'll improve it in hw-1

C++ Example – basic file operations

Read a sequence of numbers from a file, write their square roots in another file

```
#include <iostream>
#include <string>
#include <cmath>
#include <fstream>
using namespace std;
int main() {
    try {
        ifstream inFile("input.txt");
        if (!inFile.good())
          throw string("Failure opening input.txt");
        ofstream outFile("output.txt");
        if (!outFile)
          throw string("Failure opening output.txt");
            int num;
            while (inFile >> num) {
                double sqrtNum = sqrt(num);
                outFile << num << " -> "
                    << sqrtNum << endl;
            }
            inFile.close();
            outFile.close();
            return 0;
        } catch(string message) {
            cerr << message << endl;
            exit(1);
        }
}
```

input.txt:

```
36
96
49
27
```

output.txt:

```
36 -> 6
96 -> 9.79796
49 -> 7
27 -> 5.19615
```

Other important differences

- C++ has global and namespace scope
- C++ has pointers which allow pointer-arithmetic to arbitrarily refer to memory locations (both powerful and dangerous, could lead to *dangling pointers* and *memory leaks*). Modern C++ has remedies to avoid using naked pointers
- C++: assigning a reference results in a copy. In Java only an alias is created
- C++: `std::string` is mutable, Java `String` is immutable
- C++: other than global/static variables, there is no default zero-initialization for local and instance variables. For Java, instance variables are zero-initialized