CSE 1325: Object-Oriented Programming
Lecture 18

From Java to C++

Mr. George F. Rice
george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

My English teacher demanded that I name two pronouns. I exclaimed, "Who, me?"

Today's Topics

- Variables
  - Stack, heap, reference, and pointer vs heap
- Array-like Collections
  - std::vector vs ArrayList
  - Stack vs Heap
- std::cin and std::getline
- Function overloading
- Exception handling
  - std::exception vs Exception
- make
  - Makefile vs build.xml

Functions and Globals in C++

- A C++ *global* is like a Java *public static* without class membership
  - Methods without classes are called “functions” (**`placeMines`** below)
  - Fields without classes are called “global variables” (**`board`**)
- Here’s the start of a MineSweeper game in C++

```
int WIDTH, HEIGHT, MINES;

bool[][] hasMine;  // Notice no “new” is required in C++
int[][] board;          Global variable

// Board codes
const int MINE_UNKNOWN = -1; // board code default (“.”)
const int MINE_MAYBE   = -2; // board code for a possible mine (“?”)
const int MINE_KNOWN   = -3; // board code for a suspected mine (“X”)

void placeMines() {          Function
    for(int mine=0; mine<MINES; ++mine) {
        int x = rand() % WIDTH;
        int y = rand() % HEIGHT;
        hasMine[x][y] = true;
    }
}

int main(int argc, char* argv[]) {
    if(argc > 1 && std::string(argv[1]) == “-h”)) {
```

Variables in Java

- All Java variables are either fields or local (stack) variables

Stack	Value (primitive)	Reference (object address on heap)	Object (on the heap)	Heap

```
class Coordinate {
    private int x, y;
    public Coordinate(int x, int y) {this.x = x; this.y = y;}
    public Coordinate() {this(0,0);}
    public void multiply(int by) {this.x *= by; this.y *= by;}
    @Override public String toString() {return "(" + x + "," + y + ")";}
}
public class Variables {
    public static void main(String[] args) {
        int x = 3;  // primitive - 3 is on the stack
        int y = 4;  // primitive - 4 is on the stack
        Coordinate c = new Coordinate(x, y);
            // object - address is on the stack
            //   the object itself is on the heap!
        System.out.println(x + " and " + y + " makes " + c);
    }
}
```

```
ricegf@antares:~/dev/202201/18/code_from_slides$ javac Variables.java
ricegf@antares:~/dev/202201/18/code_from_slides$ java Variables
3 and 4 makes (3,4)
ricegf@antares:~/dev/202201/18/code_from_slides$
```

Variables in C++

- C++ variables may be created on the stack OR heap

Stack

Value
(primitive or object)

Reference
(address on stack OR heap)

Const Reference
(immutable, address on stack OR heap)

Pointer
(address on stack OR heap)

Heap

Value
(primitive or object on the heap)

In C++, **new** always instances object on the heap and returns a *pointer* to it!

```
// Assume an equivalent C++ Coordinate class (on GitHub now and in Lecture 19!)
int main() {
    int x = 3;              // primitive - 3 is on the stack
    int y = 4;              // primitive - 4 is on the stack
    Coordinate c(x, y);     // object - (3,4) is on the stack (impossible in Java)
    Coordinate& cr = c;     // reference to (alias of) c - same object on the stack
    Coordinate* cp = &c;    // pointer to same object on the stack
    Coordinate* c2 = new Coordinate(4, 3); // point to new object on the heap
    std::cout << x << " and " << y << " makes " << c
        << " and also " << cr
        << " as well as " << *cp
        << " and " << *c2 << " on the heap!" << std::endl;
}
```

Dereferencing a *pointer*!

```
ricegf@antares:~/dev/202201/18/code_from_slides$ c17 variables.cpp
ricegf@antares:~/dev/202201/18/code_from_slides$ ./a.out
3 and 4 makes (3,4) and also (3,4) as well as (3,4) and (4,3) on the heap!
ricegf@antares:~/dev/202201/18/code_from_slides$
```

In C++, **new** always returns a *pointer*!

3 Types of Initialization

- C++ supports 3 types of initialization (!)
  - **Assignment** (OK for primitive and some common types like std::string)
    - int a = 0;
    - std::string s = “Hello”;
  - **Direct** (for some types, but disfavored)
    - int a(x); // Problem: Is this a variable definition or a function declaration?
    - std::string s(“Hello”);
  - **Uniform** or **Brace** (favored for *almost* all types)
    - int a{0};   // Rather uncommon
    - std::string s{"hello"};   // Rather uncommon
    - std::string s2{s};
    - std::vector <std::string> vs{"a", "b", "c"};
    - std::map<string, double> height{{"Superman", 1.92}, {"Batman", 1.88},{“Wonder Woman”, 1.82},{“Hulk”, 2.23}}; // in meters, obviously :-) OR use = before first {
    - double *pd = new double[3] {0.5, 1.2, 12.99};

Note: std::vector is very similar to a Java ArrayList

std::map is like a Java HashMap

New array of doubles on the heap.

<u>Always</u> use uniform initialization for non-primitive types
(except where it doesn’t work *sigh*)

The C++ Choice
`new` or Not **`new`**?

- Objects instanced *without* **`new`** will be stored on the **stack**
  - **`Foo foo{bar};`**
  - These have limited lifetimes – when the scope exits, the object is automatically destroyed
  - This is very useful for temporary instances
- Objects instanced *with* **`new`** will be stored on the **heap**
  - **`Foo* foo = new Foo{bar}; // the only option in Java`**
  - These have unlimited lifetimes – they are *only* destroyed by an explicit **`delete`** command
    - To delete an array from the heap, used **`delete[]`** instead
  - This is very useful for long-lived instances

Why No Garbage Collector?

- Java's garbage collector is very convenient for the programmer
  - Never worry about the most common memory leaks
    - Though a clever programmer can still create them, e.g., via **`static`**
  - Memory is reclaimed only when needed
- BUT we can't predict when (or if!) (or how long!!!) gc will run
  - The gc may run at a critical, inopportune moment
  - Runtime can be long, especially with large heap
  - We cannot easily assess free memory
- C++ offers "smart pointers" that offer reference-counted instant garbage collection for objects on the heap
  - **`shared_ptr`** when many pointer copies exist (general case)
  - **`unique_ptr`** when only one pointer will ever exist (optimized case)

C++ Smart Pointers

- std::unique_ptr<T> ensures the (single referenced) memory allocated from heap is released when the managing object goes out of scope

The Problem

```
{   T *p = new T{42, "meaning"};  // T is a class type that we defined earlier
    my_function(p);
    delete p;
}
```

my_function may throw an exception, skipping the delete[] and "leaking" memory

The Solution (1 pointer)

unique_ptr's constructor accepts a pointer & instances a "smart pointer"!

```
{
    std::unique_ptr<T> p{new T(42, "meaning")};
    my_function(p);
} // p's destructor is guaranteed to run "here", calling delete
```

Even if my_function throws an exception, memory is freed when local scope exits

- std::shared_ptr<T> implements a reference counter and releases the memory when the counter reaches 0

The Solution (2+ pointers)

```
{
    std::shared_ptr<T> p(new T(3.14, "pi"));
    my_object.may_add(p);
} // p's destructor will only delete the T if may_add didn't copy the smart pointer
```

may_add <u>may</u> keep a reference to p. If so, memory is freed only when <u>both</u> references are deleted.

Know that these exist, but we will NOT code with them in class or on exams.

Parameter Mutability in Java

- A primitive's *value* is copied. The original variable's value can't be modified.
- An object's *address* is copied. The object **can** be modified within a method.
  - But the *address* stored in the original variable **cannot** be modified.

```
public class Immutables {
    public static Coordinate multiply(Coordinate c, int by) {
        c.multiply(by);  // The object on the heap can be modified
        return c;
    }
    public static void changeTo(Coordinate c, int x, int y) {
        c = new Coordinate(x, y); // The address of the object is immutable
    }
    public static void main(String[] args) {
        Coordinate c = new Coordinate(3, 4);
        System.out.println("Created as " + c);
        multiply(c, 2);
        System.out.println("     x2 is " + c);
        changeTo(c, 4, 3);
        System.out.println("Changed to " + c);
    }
}
```

```
ricegf@antares:~/dev/202201/18/code_from_slides$ javac Immutables.java
ricegf@antares:~/dev/202201/18/code_from_slides$ java Immutables
Created as (3,4)
     x2 is (6,8)
Changed to (6,8)
ricegf@antares:~/dev/202201/18/code_from_slides$
```

C++ Parameters
Like Love, it's... Complicated

- You may choose to pass a variable by
  - **Value** – The object itself is copied and may be modified in the method, but the original object is unmodified.
    Perfect for primitives and small objects you don't want to modify.
  - **Reference** – The *address* of the object is copied, exactly as in Java. The original object CAN be modified, but not the reference.
    Perfect for objects you DO want to modify.
  - **Const Reference** – The address of the object is copied, but the compiler reports an error if the function or method tries to modify the original object.
    Perfect for large objects that you don't want to modify.
  - **Pointer** – The address is copied and passed; the address in the original variable is inaccessible. The object pointed to by the parameter *and the address stored in the pointer* may be modified.
    Perfect for generating seg faults (ahem) and maximum flexibility.

Parameter Mutability in C++

- The C++ code below attempts to pass and modify an object by all 4 types
  - What output do you expect?

The character after the type – , &, or * – and the optional `const` define how the parameter is passed!

```
// Assume an equivalent C++ Coordinate class (coming in Lecture 19!)
void pass_by_value           (      Coordinate  c) {c .multiply(2);}
void pass_by_reference       (      Coordinate& c) {c .multiply(2);}
void pass_by_const_reference(const Coordinate& c) {c .multiply(2);}
void pass_by_pointer         (      Coordinate* c) {c->multiply(2);}

int main() {
    Coordinate c(3, 4); // object - (3,4) is on the stack
    pass_by_value(c);
    std::cout << "Pass by value           results in " << c << std::endl;
    pass_by_reference(c);
    std::cout << "Pass by reference       results in " << c << std::endl;
    pass_by_const_reference(c);
    std::cout << "Pass by const reference results in " << c << std::endl;
    pass_by_pointer(&c);
    std::cout << "Pass by pointer         results in " << c << std::endl;
}
```

Parameter Mutability in C++

- The compiler refuses to build code that would modify a const reference parameter!

```
ricegf@antares:~/dev/202201/18/code_from_slides$ c17 immutables.cpp
immutables.cpp: In function 'void pass_by_const_reference(const Coordinate&)':
immutables.cpp:17:65: error: passing 'const Coordinate' as 'this' argument discards qualifiers [-fpermissive]
   17 | void pass_by_const_reference(const Coordinate& c) {c .multiply(2);}
      |                                                                 ^
immutables.cpp:8:10: note:   in call to 'void Coordinate::multiply(int)'
    8 |     void multiply(int by) {this->x *= by; this->y *= by;}
      |          ^~~~~~~~
ricegf@antares:~/dev/202201/18/code_from_slides$
```

```
void pass_by_reference      (      Coordinate& c) {c .multiply(2);}
void pass_by_const_reference(const Coordinate& c) {c .multiply(2);}
void pass_by_pointer        (      Coordinate* c) {c->multiply(2);}

int main() {
    Coordinate c(3, 4); // object - (3,4) is on the stack
    pass_by_value(c);
    std::cout << "Pass by value                results in " << c << std::endl;
    pass_by_reference(c);
    std::cout << "Pass by reference            results in " << c << std::endl;
    pass_by_const_reference(c);
    std::cout << "Pass by const reference results in " << c << std::endl;
    pass_by_pointer(&c);
    std::cout << "Pass by pointer              results in " << c << std::endl;
}
```

Parameter Mutability in C++

- Comment out that issue, and here's the result

```
ricegf@antares:~/dev/202201/18/code_from_slides$ c17 immutables.cpp
ricegf@antares:~/dev/202201/18/code_from_slides$ ./a.out
Pass by value           results in (3,4)
Pass by reference       results in (6,8)
Pass by pointer         results in (12,16)
ricegf@antares:~/dev/202201/18/code_from_slides$
```

No change with pass by value.
Changed with pass by reference
and pass by pointer.
Original value.

```cpp
// Assume an equivalent C++ Coordinate class (coming in Lecture 19!)
void pass_by_value          (      Coordinate  c) {c .multiply(2);}
void pass_by_reference      (      Coordinate& c) {c .multiply(2);}
void pass_by_const_reference(const Coordinate& c) {}// c .multiply(2);}
void pass_by_pointer        (      Coordinate* c) {c->multiply(2);}

int main() {
    Coordinate c(3, 4); // object - (3,4) is on the stack
    pass_by_value(c);
    std::cout << "Pass by value           results in " << c << std::endl;
    pass_by_reference(c);
    std::cout << "Pass by reference       results in " << c << std::endl;
    // pass_by_const_reference(c);
    // std::cout << "Pass by const reference results in " << c << std::endl;
    pass_by_pointer(&c);
    std::cout << "Pass by pointer         results in " << c << std::endl;
}
```

Array-Like Collections
(or Containers, as C++ Calls Them)

- C++ equivalent to Java’s ArrayList is std::vector

```java
import java.util.ArrayList;

public class ArrayLike {
    public static void main(String[] args) {
        ArrayList<Integer> v = new ArrayList<>();
        // Remember, must be a class - NOT ArrayList<int> !!!
        v.add(42); v.add(17);v.add(255);  v.add(911); v.add(65535);
        for(var i : v) System.out.println(i);
    }
}
```

```cpp
#include <iostream>
#include <vector>

int main(int args, char* argv[]) {
    std::vector<int> v; // on stack - may be a primitive OR a class!
    v.push_back(42); v.push_back(17);v.push_back(255);
        v.push_back(911); v.push_back(65535);
    for(auto i : v) std::cout << i << std::endl;
}
```

Instead of Java’s `v.add`, in C++ use `v.push_back`!

Array-Like Collections

(or Containers, as C++ Calls Them)

- C++ equivalent to Java’s ArrayList is std::vector

```
import j

public c
    publ



    }
}
```

```
ricegf@antares:~/dev/202201/18/code_from_slides$ javac ArrayLike.java
ricegf@antares:~/dev/202201/18/code_from_slides$ java ArrayLike
42
17
255
911
65535
ricegf@antares:~/dev/202201/18/code_from_slides$ g++ --std=c++17 array_like.cpp
ricegf@antares:~/dev/202201/18/code_from_slides$ ./a.out
42
17
255
911
65535
ricegf@antares:~/dev/202201/18/code_from_slides$ 
```

```cpp
#include
#include

int main(int args, char* argv[]) {
    std::vector<int> v; // on stack
    v.push_back(42); v.push_back(17);v.push_back(255);
        v.push_back(911); v.push_back(65535);
    for(auto i : v) std::cout << i << std::endl;
}
```

Stack vs Heap

- C++ std::vector can be on the stack OR the heap

Stack

```
#include <iostream>
#include <vector>

int main(int args, char* argv[]) {
    std::vector<int> v; // on stack
    v.push_back(42); v.push_back(17);v.push_back(255);
        v.push_back(911); v.push_back(65535);
    for(auto i : v) std::cout << i << std::endl;
}
```

- In C++, **new** keyword allocates **heap** memory and returns a pointer
- **Access methods from pointer to object via -> rather than .**
- **Dereference accesses using ***

Heap

```
#include <iostream>
#include <vector>

int main(int args, char* argv[]) {
    std::vector<int>* v = new std::vector<int>; // on heap - requires pointer!
    v->push_back(42); v->push_back(17);v->push_back(255);
        v->push_back(911); v->push_back(65535);
    for(auto i : *v) std::cout << i << std::endl;
}
```

C++ Function Overloading

Exactly Like Java Method Overloading

```java
import java.util.ArrayList;

public class Overloading {
    public static void print(Integer i) {System.out.println(i);}
    public static void print(ArrayList<Integer> is) {
        for(Integer i : is) print(i);
    }
    public static void main(String[] args) {
        print((int) (Math.random() * 100));
        ArrayList<Integer> is = new ArrayList<>();
        for(int i=0; i<10; ++i) is.add((int) (Math.random() * 100));
        print(is);
    }
}
```

```cpp
#include <iostream>
#include <vector>

void print(int i) {std::cout << i << std::endl;}
void print(std::vector<int> is) {
    for(int i : is) print(i);
}
int main() {
    print(rand() % 100);
    std::vector<int> is;
    for(int i=0; i<10; ++i) is.push_back(rand() % 100);
    print(is);
}
```

C++ Function Overloading

Exactly Like Java Method Overloading

```
ricegf@antares:~/dev/202201/18/code_from_slides$ javac Overloading.java
ricegf@antares:~/dev/202201/18/code_from_slides$ java Overloading
71
15
44
40
60
94
37
88
2
18
62
ricegf@antares:~/dev/202201/18/code_from_slides$ c17 overloading.cpp
ricegf@antares:~/dev/202201/18/code_from_slides$ ./a.out
83
86
77
15
93
35
86
92
49
21
62
ricegf@antares:~/dev/202201/18/code_from_slides$
```

std::cin and std::getline functions

- Operators << and >> are overloaded by type (more soon)
- In addition to >>, C++ also has a version of C's getline() (like Java Scanner.readLine) that fills a C++ std::string instead of a C char*

```
#include <string>
#include <iostream>

int main() {
  std::string s1;
  std::cout << "Enter your name: ";
  std::cin >> s1;
  std::cout << "Your name is " << s1
            << std::endl;
}
```

```
ricegf@pluto:~/dev/cpp/201808/02$ make cin
g++ --std=c++17 -o cin cin.cpp
Now run './cin' to execute the result!
ricegf@pluto:~/dev/cpp/201808/02$ ./cin
Enter your name (including spaces): George F Rice
Your name is George
```

```
#include <string>
#include <iostream>

int main() {
  std::string s1;
  std::cout << "Enter your name: ";
  std::getline(std::cin, s1);
  std::cout << "Your name is " << s1
            << std::endl;
}
```

```
ricegf@pluto:~/dev/cpp/201808/02$ make getline
g++ --std=c++17 -o getline getline.cpp
Now run './getline' to execute the result!
ricegf@pluto:~/dev/cpp/201808/02$ ./getline
Enter your name (including spaces): George F Rice
Your name is George F Rice
ricegf@pluto:~/dev/cpp/201808/02$
```

Note that cin >> reads a whitespace-separated *word* while getline reads an entire \n-terminated *line*. Thus, getline consumes the \n, while cin >> does not.

The -o specifies the name of the executable to build.

Mixing cin and getline

- Mixing “cin >>” with getline requires care

```
#include <string>
#include <iostream>

int main() {
  std::string first, full;
  while(1) {
    std::cout << "Enter first name: ";
    std::cin >> first;
    std::cout << "Enter full name: ";
    getline(std::cin, full);
    std::cout << first
              << ", your full name is "
              << full << std::endl;
  }
}
```

Lines of code may be broken into several physical lines in the file.

```
ricegf@pluto:~/dev/cpp/201808/02$ make mixed_wrong
g++ --std=c++17 -o mixed_wrong mixed_wrong.cpp
Now run './mixed_wrong' to execute the result!
ricegf@pluto:~/dev/cpp/201808/02$ ./mixed_wrong
Enter your first name: George
Enter your full name: George, your full name is
Enter your first name:
```

getline picks up the \n left by cin. We need to ignore() it!
http://cplusplus.com/reference/istream/istream/ignore/

```
#include <string>
#include <iostream>

int main() {
  std::string first, full;
  while(true) {
    std::cout << "Enter first name: ";
    std::cin >> first;
    std::cin.ignore();
    std::cout << "Enter full name: ";
    std::getline(std::cin, full);
    std::cout << first
              << ", your full name is "
              << full << std::endl;
  }
}
```

ignore() is a *method* of the cin *object*.

```
ricegf@pluto:~/dev/cpp/201808/02$ make mixed_right
g++ --std=c++17 -o mixed_right mixed_right.cpp
Now run './mixed_right' to execute the result!
ricegf@pluto:~/dev/cpp/201808/02$ ./mixed_right
Enter your first name: George
Enter your full name: George F. Rice
George, your full name is George F. Rice
Enter your first name:
```

Exceptions in C++ vs Java

- All C++ exceptions are unchecked (C++ uses no **`throws`**)
- Exception hierarchies and multiple catch supported
  - But no try-with-resources or **`finally`**
  - C++ relies on its destructors for clean up – no garbage collector
- Custom exceptions are fine – in fact, C++ will throw *anything* as an exception, even primitives (!)
  - Catch *anything* with **`catch(...)`**
  - But play nice and always subclass **`std::exception`**
  - Reminder: In Java, only objects implementing the **`Throwable`** interface can be thrown (but we usually extend **`Exception`**)
- Get exception message with **`e.what()`**, not **`e.getMessage()`**

What?!?

Roughly Equivalent Exceptions

Java	C++
Exception (superclass of Exception types)	std::exception (superclass of exceptions) std::runtime_error (general purpose)
ArrayIndexOutOfBoundsException	std::out_of_range
IllegalArgumentException	std::invalid_argument or std::logic_error
ArithmeticException	None - poll errno (from <cerrno>) Boost* supports exceptions, though
ClassCastException	std::bad_cast
NullPointerException	segfault
ClassNotFoundException (checked)	std::bad_function_call
InterruptedException (checked)	No equivalent – uses SIGINT signal and non-exception mechanism
OutOfMemoryError (technically an error)	std::bad_alloc

* Boost is a separate but more comprehensive C++ library

Example: ROT13 (again)

- ROT13 (“rotate 13”) is a very simple encryption cypher with a very useful property – encrypting a string again decrypts it!
  - Simple rotate each char ahead by 13 chars
    - A becomes N, B becomes O, X becomes K
    - N becomes A, O becomes B, K becomes X (!)
  - This only works for English, of course
- Non-letter chars (in our example) will throw an exception
- Very popular on the pre-web Internet
  - Usenet newsgroup readers all had ROT13 functions
  - Allowed “hiding” the solution to riddles and jokes in messages

Throwing a std::runtime_error

```cpp
#include <iostream>

void rot13(std::string& s) {
    std::string key = "nopqrstuvwxyzabcdefghijklm";

    for(char& c : s) {
        if(c == ' ') continue;
        if('a' > c || c > 'z')
            throw std::runtime_error{"Invalid char: " + std::string{c}};
        c = key[c-'a'];
    }
}

int main() {
    std::string s;
    std::cout << "Enter a string: ";
    std::getline(std::cin, s);
    try {
        rot13(s);
        std::cout << s << std::endl;
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }
}
```

```
student@cse1325:/media/sf_dev/23$ ./10_rot13
Enter a string: hello World
Exception: Invalid char: W
student@cse1325:/media/sf_dev/23$
```

Defining a Custom C++ Exception

- Custom exceptions inherit from std::exception
  - We can add whatever additional fields and methods that are helpful to us
  - Optionally, we can override **`what()`** so that catching **`std::exception`** provides a better message

```
class Bad_char : public std::exception {
  public:
    Bad_char(std::string s, char c) { Custom, more useful, constructor!
        msg =  "Bad character " + std::string{c} + " in " + s;
    }                                  Override the what() method
    const char* what() const noexcept override { // Don't worry about these yet...
        return msg.c_str();
    }
  private:             c_str method converts std::string to char* (remember those?)
    std::string msg;
};
```

Sneak preview of C++ classes (from Lecture 19)

Throwing a Custom Exception

```
void rot13(std::string& s) {
    std::string key = "nopqrstuvwxyzabcdefghijklm";

    for(char& c : s) {
       if(c == ' ') continue;
       if('a' > c || c > 'z') throw Bad_char{s, c};
       c = key[c-'a'];
    }
}

int main() {
    std::string s;
    std::cout << "Enter a string: ";
    std::getline(std::cin, s);
    try {
        rot13(s);
        std::cout << s << std::endl;
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }
}
```

Notice missing **`new`**.
C++ passes exceptions by *value* rather than by reference!

```
student@cse1325:/media/sf_dev/23$ ./11_custom_except
Enter a string: hello World
Exception: Bad character W in uryyb World
student@cse1325:/media/sf_dev/23$
```

No changes to main – it doesn't care!

Example: Timers

- Let's create a stopwatch program in Java and C++ that calculates lap and overall time
  - Accepts a name for each lap (or name or event)
  - EOF (Control-d on Linux / Mac, Control-z on Windows) when the timing is complete
  - Prints a nicely formatted table of lap and overall elapsed times

Timers Timing

```
ricegf@antares:~/dev/202201/18/code_from_slides$ java Timer
Enter some event names, and I'll time them!
Press Control-d to exit
Wilma
Fred
Dino
   Event Description       From Start    From Previous
               Wilma            4.207            4.207 seconds
                Fred            7.048            2.841 seconds
                Dino            8.864            1.815 seconds
ricegf@antares:~/dev/202201/18/code_from_slides$ ./timer
Enter some event names, and I'll time them!
Press Control-d to exit
Wilma
Fred
Dino
   Event Description      From Start   From Previous
               Wilma           4.167           4.167 seconds
                Fred           5.239           1.071 seconds
                Dino           6.871           1.633 seconds
                               8.543           1.672 seconds
ricegf@antares:~/dev/202201/18/code_from_slides$ 
```

Java Stopwatch

```java
class Timer {
    private static double elapsedTime(Instant start, Instant finish) {
        return ((double) Duration.between(start, finish).toMillis()) / 1000.0;
    }
    public static void main(String[] args) {
        System.out.println("Enter some event names, and I'll time them!");
        System.out.println("Press Control-d to exit");

        Scanner in = new Scanner(System.in);
        String line = "";

        ArrayList<Instant> times = new ArrayList<>();
        ArrayList<String> events = new ArrayList<>();
        times.add(Instant.now());
        events.add("Begin");

        while(in.hasNextLine()) {
            line = in.nextLine();
            times.add(Instant.now());
            events.add((line != null) ? line : "End");
        }

        System.out.printf("%20s %15s %15s\n",
                          "Event Description", "From Start", "From Previous");
        for(int i=1; i<times.size(); ++i) {
            System.out.printf("%20s %15.3f %15.3f seconds\n", events.get(i),
                elapsedTime(times.get(0),   times.get(i)),
                elapsedTime(times.get(i-1), times.get(i)));
        }
    }
}
```

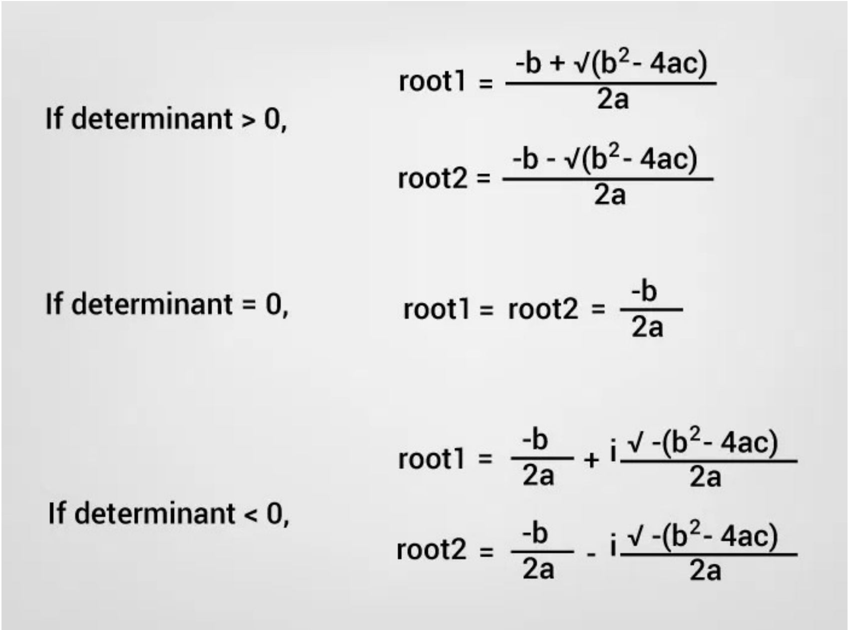
C++ Stopwatch

```
using time_point = std::chrono::steady_clock::time_point;
constexpr auto now = &std::chrono::steady_clock::now;

double elapsed_time(time_point start, time_point finish) {
    return (finish - start).count() / 1000000000.0;
}
int main() {
    std::cout << "Enter some event names, and I'll time them!\n"
              << "Press Control-d to exit" << std::endl;

    std::vector<time_point> times;    std::vector<std::string> events;
    times.push_back(now());           events.push_back("Begin");

    std::string line;
    while(std::cin) {
        std::getline(std::cin, line);
        times.push_back(now());
        events.push_back(line);
    }

    std::cout << std::setw(20) << "Event Description"
              << std::setw(15) << "From Start"
              << std::setw(15) << "From Previous" << std::endl;

    std::cout << std::fixed << std::setprecision(3);
    for(int i=1; i<times.size(); ++i) {
        std::cout << std::setw(20) << events[i]
                  << std::setw(15) << elapsed_time(times[0],   times[i])
                  << std::setw(15) << elapsed_time(times[i-1], times[i])
                  << " seconds" << std::endl;
    }
}
```

System.out.printf would be shorter but less instructive.

Example: Quadratics

- A quadratic equation is an equation that can be rearranged in standard form as ax^2+bx+c=0;
  - The solution can be determined directly by formula
- We'll write a C++ program from scratch that solves any quadratic given a, b, and c as command line arguments
  - And by "we" I mean "you"!



Step 1: Includes and main

- We'll need iostream for std::cout and cmath for square roots

Step 1: Includes and main

- We'll need iostream for std::cout and cmath for square roots

```
#include <iostream>
#include <cmath>

int main(int argc, char* argv[]) {
```

Step 2: Verify arguments

- If we didn't get 3 arguments (a, b, and c), show a usage statement and exit with error code -1

Step 1: Verify arguments

- If we didn't get 3 arguments (a, b, and c), show a usage statement and exit with error code -1

```
if(argc != 4) {
    std::cerr << "usage: " << argv[0] << " <a> <b> <c>\n"
              << "       for ax^2 + bx + c" << std::endl;
    return -1;
}
```

Step 3: Convert args to doubles

- Create variables a, b, and c and set them to the program arguments. Print error is not doubles.

Step 3: Convert args to doubles

- Create variables a, b, and c and set them to the program arguments. Print error is not doubles.

```
double a, b, c;
try {
    a = std::stod(std::string(argv[1]));
    b = std::stod(std::string(argv[2]));
    c = std::stod(std::string(argv[3]));
} catch(std::exception e) {
    std::cerr << "arguments must be 3 doubles: " << e.what() << std::endl;
    return -2;
}
```

Step 4: Real root(s)

- Calculate the discriminant (b*b – 4*a*c) and print the roots if they are real

Step 4: Real root(s)

- Calculate the discriminant (b*b – 4*a*c) and print the roots if they are real

```
double discriminant = b*b – 4*a*c;

if (discriminant > 0) {
    double x1 = (-b + sqrt(discriminant)) / (2*a);
    double x2 = (-b - sqrt(discriminant)) / (2*a);
    std::cout << "Roots are " << x1 << " and " << x2 << std::endl;
} else if (discriminant == 0) {
    double x = -b/(2*a);
    std::cout << "Roots are both " << x << std::endl;
}
```

Step 5: Imaginary roots

- If imaginary, print the imaginary roots, showing both + and – signs when printing doubles.

Step 5: Imaginary roots

- If imaginary, print the imaginary roots, showing both + and – signs when printing doubles.

```
else {
        double real = -b/(2*a);
        double imag = sqrt(-discriminant)/(2*a);
        std::cout << std::showpos
                  << "Roots are " << real <<  imag << "i and "
                                  << real << -imag << "i" << std::endl;
    }
}
```

Complete Quadratic Program

```
int main(int argc, char* argv[]) {
    if(argc != 4) {
        std::cerr << "usage: " << argv[0] << " <a> <b> <c>\n"
                  << "       for ax^2 + bx + c" << std::endl;
        return -1;
    }
    double a, b, c;
    try {
        a = std::stod(std::string(argv[1]));
        b = std::stod(std::string(argv[2]));
        c = std::stod(std::string(argv[3]));
    } catch(std::exception e) {
        std::cerr << "arguments must be 3 doubles: " << e.what() << std::endl;
        return -2;
    }
    double discriminant = b*b - 4*a*c;
    if (discriminant > 0) {
        double x1 = (-b + sqrt(discriminant)) / (2*a);
        double x2 = (-b - sqrt(discriminant)) / (2*a);
        std::cout << "Roots are " << x1 << " and " << x2 << std::endl;
    } else if (discriminant == 0) {
        double x = -b/(2*a);
        std::cout << "Roots are both " << x << std::endl;
    } else {
        double real = -b/(2*a);
        double imag = sqrt(-discriminant)/(2*a);
        std::cout << std::showpos
                  << "Roots are " << real <<  imag << "i and "
                                  << real << -imag << "i" << std::endl;
    }
}
```

What We Learned Today

- Unlike Java, C++ objects and primitives are flexible
  - Both objects and primitives may be on the stack or the heap – your choice!
  - Variables may hold a value, or be a reference, const reference, or pointer
  - The **`new`** keyword allocates memory on the heap and returns a pointer to it
- Two common ways to initialize variables (parentheses are obsolete)
  - Via assignment (**`int i=3; std::string s="Ok";`**)
  - Via braces (**`std::vector<std::string> vs{"a", "b", "c"}; double *pd = new double[3] {0.5, 1.2, 12.99};`**)
- Parameters may be passed by value, reference, const reference, or pointer – your choice!
- Throw exceptions by value (**`throw std::runtime_error;`**) and get a caught exception's message with **`e.what()`**