CPSC 1070 Week 3 Notes

Micheal Merritt, Brendan McGuire, Daniela Arroyo

Table of Contents

C++ Expressions and Interactions

Streams

In C++, Streams represent a continuous flow of data from one place to another. The chief example of streams you see in C++ is the **Standard I/O**, represented by `std::cout`, `std::cerr`, and `std::cin` for standard output, error, and input respectively.

```
#include <iostream>

int main() {
     std::cout << "Hello World" << std::endl;
}
```

Why not use `print` like all the other languages?
By modeling the input and output as a stream of data, you're getting closer access to how the computer is actually dealing with those streams of data.

Right now, it's annoying to deal with, but it will prove useful in the future, promise.

The << and >> Operators
These are the stream insertion and stream extraction operators. They allow you to put and pull data from streams, respectively.

They allow us to add data or remove many types of data from the streams without having to join it all together into a single representation. To see why this is useful, try to create a single variable that is the concatenation of a string literal, a double type, and an integer (it's not easy or fun!).

You can achieve the same functionality by using the stream operators instead:

```
cout << months << " months ";
cout << " (" << years << " years)";
cout << " is required to pay off the loan.\n" << endl;
```

Stream Manipulators
Oftentimes, you want to customize how your data is pushed to the stream. This is achieved with stream manipulators, found in `<iomanip>`

Stream manipulators are added to your stream just by inserting them using the stream insertion operator, and they will then affect anything else downstream.

```
std::cout << fixed << setprecision(0) << months << " months";
```

Here are some common stream manipulators:

Manipulator	Description
`std::fixed`	Outputs floating-point numbers with a fixed decimal, instead of in scientific notation.
`std::setprecision(n)`	Sets the decimal precision for floating-point numbers to n decimal places
`std::endl`	Appends a newline to the stream, and then clears the buffer. In standard I/O this will display the contents of the stream.
`std::flush`	Does the same as above, but doesn't add a newline.

You can see a complete list of manipulators at
https://www.cplusplus.com/reference/library/manipulators/

Expressions

When you combine multiple variables, literals, together, you're creating an expression. At runtime, expressions are condensed to form a single final value. This is done in the order of operations:

Order of Operations

1. Parentheses
2. Unary Operators
3. Multiplication/Division/Modulo
4. Addition & Subtraction

Prefix & Postfix Incrementing and Decrementing

There are many use cases where you need to increase or decrease a variable by one, and the unary increment (++) or decrement (--) operators help you out here.

If the operator is placed before the variable, this is known as a prefix operation. The variable will be incremented/decremented, and the expression will return the new value.

If the operator is placed after the variable, this is known as a postfix operation. The variable will be incremented/decremented, and the expression will return the previous value.

```
int a = 1;
int b = ++a; // b = 2, a = 2

int c = 1;
int d = c++; // b =2 , d = 1
```

Type Casting & Coercion

Oftentimes, you need to convert from one type to another. If you want to display a number, you need to convert it to a string. If you want to convert from one type of number to another to perform a mathematical operation.

Casting with static_cast
A lot of this time, this can be done implicitly by the compiler. But when you need to do an explicit cast, you can use a `static_cast`

```
int main() {
    int number = 3455;
    double asDouble = static_cast<double>(number);
}
```

C-Style Casts
In older or more C focused codebases, you'll likely see an older C style cast. This is roughly the same as the cast above.

```
int main() {
    int number = 3455;
    double asDouble = (double)(number);
}
```

Overflow/Underflow

When casting numbers, you may run into *overflow (value too low)* or *overflow (value too high)* errors. Depending on your system, this may just issue a warning, or completely stop the execution of your program. **As a general rule of thumb, don't cast from a high information number (double, float, long) to a low information number (int, short)**

Control Statements & Scope

Control Statements

If/else
Lets your code branch based on the conditional expression inside the if statement.

```
#include <iostream>

using namespace std;

int main()
{
    double number;

    cout << "Enter a number: ";
    cin >> number;

    if (number > 100.0)
    {
        cout << "That number is greater than 100"
    }
    else
    {
        cout << "That number is less than or equal to 100"
    }
}
```

Logical Operators & Short Circuit Evaluation

If you need to modify or combine logical expressions, you need to use a *logical operator*.

Operator	Logical Equivalent	Short Circuit Evaluation
!a	NOT a (true <-> false)	n/a
a || b	a OR b	If a is true, then b is not evaluated.
a && b	a AND b	If a is false, then b is not evaluated

Switch

If you have a long series of similar if else calls, sometimes it can be easier to group them together as a switch statement.

```
switch (day) {
  case 0: { cout << "Sunday!" << endl; break; }
  case 1: { cout << "Monday!" << endl; break; }
  case 2: { cout << "Tuesday!" << endl; break; }
  case 3: { cout << "Wednesday!" << endl; break; }
  case 4: { cout << "Thursday!" << endl; break; }
  case 5: { cout << "Friday!" << endl; break; }
  case 6: { cout << "Saturday!" << endl; break; }
  default:  { cout << "Invalid Day Number!"}
}
```

A good use case for switch statements is enumerators. At compile time, enumerators collapse to another type, but while you are programming, they can be a useful construct to help you keep track of numbers

```
enum Suit { Diamonds, Hearts, Clubs, Spades };
Suit suit = Suit::Diamonds;

switch (suit) {
    case Suit::Diamonds:
    case Suit::Hearts: {
          cout << "Red cards!" << endl;
          break;
    }

    case Suit::Clubs:
    case Suit::Spades: {
```

```
        cout << "Black cards!" << endl;
    }

}
```

While Loops

While loops will repeat code as long as the condition at the top is true

```
int main()
{
    double number;

    cout << "Enter a number: ";
    cin >> number;

    int count = 0;
    while (count < number)
    {

        cout << count << endl;
        count++;
    };
}
```

For Loops

A special kind of while loop, usually used to repeat some code a number of times. The statement inside of the for loop has three components. The first section is run before the loop executes (usually to create a counter), the second section checks to make keep the loop running (usually checking the counter against a total), the last section runs after each loop (usually to increment a counter).

```
int main()
{
    double number;

    cout << "Enter a number: ";
    cin >> number;

    for (int count = 0; count < number; count++)
    {
```

```
        cout << count << endl;
    };
}
```

Break & Continue

When you are inside a loop, there may be times when you want to end a loop iteration early, or break from the loop entirely. This can be done with `continue` and `break` respectively.

In loops, break used to end the loop entirely. Continue is used to end this iteration, and start again at the loop body.

In a switch statement, break is used to prevent case fallthrough.

```
int findNumberSum() {

  int sum = 0;
  for (int i = 0; i < 100; i++) {
    // Ignore multiples of 5 and multiples of 7
    if (i % 7 == 0 || i % 5) continue;

    sum += i;

    if (sum > 30) {
      break;
    }
  }
  return i
}
```

An example of using break and continue in a loop

```
switch (suit) {
    case Suit::Diamonds:
    case Suit::Hearts: {
          cout << "Red cards!" << endl;
          break;
    }

    case Suit::Clubs:
    case Suit::Spades: {
        cout << "Black cards!" << endl;
     }
```

```
}
```

Break in switch statements is used to prevent fallthrough. Execution will pass through the next statement, allowing you to perform a single block with multiple cases, but this can lead to a lot of undefined behavior, so you should be careful

Scope

Most variables aren't accessible forever, they have a *scope*. There are many kinds of scope in C++, but the predominant one you'll have to think about is **block scope,** which means that the scope of a variable is the innermost set of curly braces.

Anytime you add a new set of curly braces, you are creating a scope. You can always access the variables inside the scope directly outside of you, but you can't access the scope inside you, nor can you access the scope beside you.

```
{ // scope created
int a = 24;

{ // new scope is created, but a is still accessible
int b = 32;
a = b;
} // b goes out of scope

{ // new scope, a is accessible, but b is not
b = 34; // This is a compiler error
}

} // a goes out of scope

a; // This is a compiler error.
```

Study Questions

1. What is the correct format to print in C++?
    a. system.out.print (“Hello World”);
    b. print(“Hello World”)
    c. cout << “Hello World” << endl;
    d. cout >> “Hello World” >> endl;

2. How would you create an explicit cast?
    a. `static_cast`
    b. `explicit_cast`
    c. `cast_static`
    d. `cast_explicit`

3. In what order should one follow when creating an expression?
    a. Parentheses, Unary Operators, Multiplication/Division/Modulo, Addition & Subtraction
    b. Addition & Subtraction, Unary Operators, Parentheses, Multiplication/Division/Modulo
    c. Parentheses, Multiplication/Division/Modulo, Addition & Subtraction, Unary Operators
    d. Unary Operators, Parentheses, Addition & Subtraction, Multiplication/Division/Modulo

4. When should a programmer use a switch statement?
    a. To replace the value of a variable with another set of values
    b. To compare the value of a variable against a set of other values
    c. To make your code easier to read

5. What is the purpose of the break keyword in a loop?
    a. A signal to exit out of the loop
    b. To break the current iteration and skip to the next one

6. What is the difference between = and ==?
    a. = is an equal sign and == is a double equal sign
    b. = is an assignment operator and == is a rational operator
    c. = is a rational operator and == is an assignment operator
    d. = is used in C and == is used in C++

7. What would be the output of this code?

```
#include <iostream>
    using namespace std;
    int main()
    {
        int a = 10;
        if (a < 15)
        {
            time:
            cout << a;
            goto time;
        }
        break;
        return 0;
    }
```

    a. 1010
    b. 10
    c. Infinitely print 10
    d. Compile time error

CPSC 1070 Week 5 Notes

Micheal Merritt, Brendan McGuire, Daniela Arroyo

Object-Oriented Programming

For some programs, it makes sense to model the structure of your code after several discrete objects, and the relationships between these constructs.

What are some use cases for an object-oriented style?

- ➔ Games or Simulations
- ➔ Modeling a real-world object
- ➔ Provide easy organization to a singular construct that your program will create many times

When might object-oriented programming be a bad choice for your program structure?

- ➔ Inheritance can lead to some unnecessary boilerplate. You should avoid creating abstractions that do not have logical parallels (watch out for Manager or Provider classes)
- ➔ If you find yourself using a lot of static methods, a namespace may better suit your use case

Abstraction

One of the core principles of OOP, and Computer Science in general, is abstraction. Abstraction occurs when specific implementation details of a program are hidden from the public interface.

Why is abstraction helpful?

- ➔ **Don't Sweat The Details**. You shouldn't need to worry about all of the particulars of vectors just to use it.
- ➔ **The programmers who made it probably know better**. By restricting your access to only a few public accessors/mutators, shooting yourself in the foot is a lot harder
- ➔ **Internal changes can be made more easily.** If the creator of the ADT has an optimization, they can safely make it without messing up your code.

Classes & OO Analysis (By Example)

Let's create a class to represent a character in a 2D game. Before we start programming, we need to figure out the shape of the class, and what our class will need to do.

OO Analysis: What does the Character Class need to do?

- ➔ Should keep track of the player's position, velocity, and acceleration
- ➔ Should provide accessors and mutators for these values
- ➔ Should have a public step function to process all the physics: adjust position & velocity
- ➔ Should have a public function to "render" the character to the screen

OO Analysis: What are the relationships with other classes?

This example only has one class, but let's consider how a character class might interact with other classes in a game

- ➔ Should talk to the renderer (access) to render the character onto the screen
- ➔ Has ownership over the items in the character's inventory
- ➔ Would probably inherit from a more general Entity class (inheritance)

character.h

Now that we have a general idea of the class shape and requirements, let's get coding! We'll start with the header file, which will define the general shape of the class.

```
#include <tuple>

/* These are header guards, so we can make sure the file never gets included more
than once. Not including header guards can lead to some really strange errors */
#ifndef CHARACTER_H
#define CHARACTER_H

using namespace std;

// This is the class definition, it should show the general shape of the class
you're creating, including a public and private definitions
class Character {

    // We're going to store the position and velocity of the player character in
    // private variables, and allow external code to access them using mutators
    // and accessors. This will allow us to change this internal code without
having
    // to mess up external code in the future
    private:
        double pos_x;
        double pos_y;

        double vel_x;
        double vel_y;
```

```
        double accel_x;
        double accel_y;

// The public shape of the class, that external code can access. You should be thoughtful of how and when you change this code because doing so requires you (or other people) to go back and change the
    // code that uses this method
    public:

        // Constructors, with a default constructor implementation to place the character at (0,0)
        Character(double x, double y);
        Character();

        // Destructor, to free memory when this object goes out of scope
       ~Character();

        // Position Accessors and Mutators
        void setPosition(double x, double y);
        pair<double, double> getPosition();

        // Velocity Accessors and Mutators
        void setVelocity(double x, double y);
        pair<double, double> getVelocity();

        // Acceleration Accessors and Mutators
        void setAcceleration(double x, double y);
        pair<double, double> getAcceleration();

        // Steps the computation for each frame, moving position a little, and reducing velocity a little
        void step();

        // "Renders" the character to the screen (for now just prints the position to the terminal)
        void render();

};

#endif
```

character.cpp

Now that we've outlined how our class looks, let's implement some of the specifics. Remember that when you're implementing a method in a class, use the scope resolution operator (::).

```
#include "character.h"

/**
 * Initializes the player at the given coordinate
 * @param x x coordinate
 * @param y y coordinate
 **/
Character::Character(double x, double y) {
    pos_x = x;
    pos_y = y;

    vel_x = 0;
    vel_y = 0;

    accel_x = 0;
    accel_y = 0;
};

/**
 * Default Constructor, which initializes the player at (0,0).
 * By using constructor delegation, we are able to easily provide a default
constructor
 **/
Character::Character() : Character(0, 0) {};
/**
 * Because we're not directly allocating any memory in the class, we shouldn't
need to put
 * anything here in the destructor
 **/
Character::~Character() {

}

/**
 * Mutator for the position
 * @param x x coordinate
 * @param y y coordinate
 **/
void Character::setPosition(double x, double y) {
    pos_x = x;
    pos_y = y;
```

```
}

/**
 * Gets (accessor) the position of the current player character
 **/
pair<double, double> Character::getPosition() {
    return { pos_x, pos_y };
}

/**
 * Mutator for the velocity
 * @param x x coordinate
 * @param y y coordinate
 **/
void Character::setVelocity(double x, double y) {
    vel_x = x;
    vel_y = y;
}

/**
 * Gets (accessor) the velocity of the current player character
 **/
pair<double, double> Character::getVelocity() {
    return { vel_x, vel_y };
}

/**
 * Mutator for the acceleration
 * @param x x coordinate
 * @param y y coordinate
 **/
void Character::setAcceleration(double x, double y) {
    accel_x = x;
    accel_y = y;
}

/**
 * Gets (accessor) the acceleration of the current player character
 **/
pair<double, double> Character::getAcceleration() {
    return { accel_x, accel_y };
}


/**
```

```
 * Steps through the kinematic calculations, updating the position and velocity
based on the acceleration
 **/
void Character::step() {
    pos_x += vel_x;
    pos_y += vel_y;

    vel_x += accel_x;
    vel_y += accel_y;
};

/**
 * "Renders" the character to screen, printing it out in the terminal
 **/
void Character::render() {

    pair<double, double> position = getPosition();

    cout << "Render Character at (" << pair.first << "," << pair.second << ")" <<
endl;
}
/**
 * In the main method, step through a basic example of instantiating and using a
Character Class
 **/
int main() {

    // Use the default constructor to initialize character at (0, 0)
    Character player;
    player.setAcceleration(2, 2);

    while (true) {
        player.step();
        player.render();

        // Slowly lower the players acceleration to simulate friction
        pair<double, double> accel = player.getAcceleration();
        player.setAcceleration(accel.first - 0.1, accel.second - 0.1);
    }
}
```

Enumerators

Enumerators are most useful when you're trying to represent a set number of categories, like months or Grade Level. You can use the class specifier to namespace the items in your enum.

```
enum class Year
{
    Freshman,
    Sophomore,
    Junior,
    Senior
};

Year student = Year::Freshman;
int a = (int)student; // a = 0
```

By default, enums are represented at runtime by integers, starting at 0, and counting up. However, this can be overridden if you so desire. By using a colon after the enum name, you can specify a type to represent the enum at runtime

```
enum class Year : char
{
    Freshman = 'F',
    Sophomore = 'S',
    Junior = 'J',
    Senior = 'E'
}

Year student = Year::Sophomore;
char a = (char)student; // a = 'E'
```

Compiler Internals

When you compile a program with g++, what is actually happening? The compiler will usually go through a few steps.

1. Compilation

In this step, the compiler will tokenize, parse, and validate your code file by file. The output of this step is an object code file, which contains a base level intermediate representation, which will be translated into the binary in the next step

```
g++ -c file.cpp
```
Produces an object file, file.o

2. Linking

In this step, the compiler will link together all of the object files that will comprise your binary. By looking at the symbols in each object file, and connecting everything together, including code from external sources, like the standard library.

The output of this step is a single binary .out file

Arrays & Vectors

Arrays and Vectors let you store many different values of the same type, in similar memory locations. Arrays are a holdover from C, but still, find widespread use in C++ codebases because of their low memory footprint and compatibility with the C Standard Library. Vectors are a new construct in C++ and let you define arrays of arbitrary size, automatically.

Arrays	Vectors
- Simpler memory model - Must be static (number of elements initialized at compile time) - Degrades to a pointer when passed to a function (pass by reference)	- C++ Class - Automatically allocates/frees memory - Can have an arbitrary (and changing) number of elements - C++ copy semantics apply when being passed to function (pass by copy)

```
int numbers[6] = {1, 2, 3, 4, 5};
```

```
#include <vector>
vector<int> numbers({1, 2, 3, 4, 5});
```

Element Access

What good is a list of elements if you can't access the individual elements. In C++, the primary way of accessing elements in a group is using the [] operator. It's important to remember that both of the following methods return *references* to the data in question.

Square Bracket Operators

You can access contiguous elements by putting the desired index in square brackets. This works for both Arrays and Vectors.

```
int num = numbers[2]; // num = 3, works for both the Array and Vector above
numbers[2] = 10; // Updates the location in memory
```

Things to be aware of when using square brackets

- Range bounds are not verified in either the array or the vector. This can lead to some strange side effects if you do not bound check yourself
- Returns a raw pointer, so if you're not careful, can lead to use-after-free errors

.at()

In addition to the [] operator, a C++ vector supports an additional method .at() which returns a std::reference to the element at index i.

```
int num = numbers.at(2); // num = 3, works for only Vector
numbers.at(2) = 10; // Updates the location in memory
```

- Does ranging checking for you, and throws an exception if out of bounds
- Same return type as square brackets, so use-after-free errors are still possible.

Iteration

There are several ways to iterate through items in arrays and vectors in C++, but the primary methods are index-based loops and range-based loops

Index-Based Loops

These are the regular loops you are thinking about: define a starting point, and ending point, and how much to increase by each time.

- Simple, work for almost everything
- Bound checks can be really frustrating (you need to make sure you start and stop in the right place)
- Easy to change iteration order: can skip elements easily, can go backward easily

Plain Arrays

```
int SIZE = 5;
int numbers[] = {1, 2, 3, 4, 5};

for (int i = 0; i < SIZE; i++) {
  cout << size[i] << endl;
  size[i]++;
}
```

Vectors

```
#include <vector>

vector<int> numbers({1, 2, 3, 4, 5});

for (int i = 0; i < numbers.size(); i++) {
  cout << size[i] << endl;
  size[i]++;
}
```

Range-Based Loops

Using the colon syntax, we can iterate through all the elements in an array without worrying about out of bounds errors

- Less freedom than index-based (need to iterate through every item, from start to finish)
- The same syntax for many different containers
- Don't need to worry about out of bounds errors.

Plain Arrays

```
int numbers[] = {1, 2, 3, 4, 5};


for (int &num : numbers) {
  cout << num << endl;
  num++;
}
```

Vectors

```
#include <vector>

vector<int> numbers({1, 2, 3, 4, 5});

for (int &num : numbers) {
  cout << num << endl;
  num++;
}
```

Study Questions

1) Which of the following is the most accurate way to declare a getSide accessor method for a Square class.

```
int getSide()
{
    return side;
}
(A)
------------------------------
int Square::getSide(int)
{
    return side;
}
(B)
------------------------------
int Square::getSide()
{
    return side;
}
(C)
```

    a) Option A.
    b) Option B.
    c) Option C.
    d) None of these are correct

2) What is the output of this code?

```
3)  enum Color
4)      {
5)          red,
6)          green = 3,
7)          blue,
8)          yellow = 1,
9)          brown
10)     };
11)
12)     Color r = static_cast<Color>(blue);
13)     cout << r;
```

a) blue
b) 4
c) 1
d) 2
e) Compiler Error

3.) Which of the following Data types if any, can change its number of elements after declaration

a) Vectors
b) Arrays
c) Neither

4)What will this code output?

```
int sizes[] = {1, 2, 3, 4, 5};


    for (int i : sizes)
    {
        if (i % 2 == 0)
        {
            continue;
        }
        cout << i++ << " ";
    }
```

a) 2 4 6
b) 1 2 3 4 5
c) 1 3 5
d) 2 3 4 5 6 7

Answer Key

1.) Answer choice C is correct. Since this is an accessor method, it begins with the data type of what's going to be returned, an int. In C++ when writing a method for a class, you must state the class name followed by a scope resolution operator(::) before you state the name of the function.
2.) Answer Choice B is correct. Whenever you define the exact value of an enum type and leave the next one undefined, it defaults to being the previous value + 1. Therefore blue is 4 and brown is two.
3.) Answer choice A is correct. When working with arrays, you must either explicitly or implicitly define their size ex. Int foo[5]; or int foo[] = {1,2,3}; Vectors are special in that you can use the push back(ads element at end)/ pop back(deletes element at end) methods to dynamically adjust its size
4.) Answer choice C is correct. The for-each loop will check each value in the sizes integer array. If the remainder when dividing that value by 2 is equal to zero, the loop will continue to the next iteration, otherwise, it'll cout that value. Since we used the POSTFIX(i++) operator, it'll print out 1,3,5 instead of 2,3,6.

CPSC 1070 Week 7 Notes

Micheal Merritt, Brendan McGuire

Advanced Objects

This Pointer

```
void File::setFileName(string file) {
  this->file = file;
}

string File::getFileName() const {
  return this->file
}
```

What does it do?
In a class member, this is a pointer to the instantiated class object.

Why is it useful?
When you have a local variable that's the same as a member variable and need to distinguish them

Constant Member Function

```
void Model::setValues(const double a,
const double b) {
 this->a = a;
 this->b = b;
}

double Model::getComputedValue() const {
  return this->a + this->b
}
```

What does it do?
When you place const **before a parameter**, the function cannot modify that parameter

When you place const **after the parameter list** the function cannot modify the object state

Why is it useful?
These are useful annotations to limit how you are able to modify state, which is a leading cause of errors.

Static Members

```
class ShoppingCart {
  private:
    vector<double> prices;
    static double taxRate;
  public:
    double computeTotal() const;
    static double calculateTax(double a) {
      return a * taxRate;
    }
}

double ShoppingCart::computeTotal() {
  double total = 0;
  for (double price : prices) {
    total += price;
  }
  return total + calculateTax(total)
}
```

What does it do?
Static members are shared across all instances of a class, allowing you to share common behavior.

Shared static variables can be accessed in any instance of the class

Shared static members can be called from any member of the class

Why is it useful?

- ➔ Shared state and behavior across instances
- ➔ Localized constants

Keep in mind: you should only have a few static members per class. If you find yourself having a lot of static members, a namespace may be better suited to your needs.

Friends of Classes

```
int getPerimeter(Square sq) {
    return sq.side * 4;
};

class Square
{

private:
    int side;
    friend int getPerimeter(Square sq);
    Square(int s) : side(s){};

public:
    int getArea();
}
```

What does it do?
Allows specific functions or classes to access the private (or protected) members of a class by specifying their status in the class declaration.

Why is it useful?
Friend classes should be used sparingly!

Declaring a class or function a friend breaks encapsulation by allowing external code to access your internals. This can be useful in a few cases and when prototyping early on, but as your code gets more complex it can lead to some undefined behavior.

Copy Constructors and Memberwise Agreement

```
class RemoteFile
{
private:
    string url;
    bool fetched;
    string contents;
public:
    RemoteFile(string u) : url(u) {
        fetched = false;
    }
    // Copy Constructor: Copy the URL
    RemoteFile(const RemoteFile &old) {
        fetched = false;
        url = old.url;
    }
    void get() {
        // Make the request
        string response = request(url);
        contents = response;
        fetched = true;
    }

}

RemoteFile a("http://example.com");
RemoteFile b = a;
```

What does it do?
Copy Constructors let you override the behavior when you assign one object to another. This can be incredibly useful for enforcing C++ copy semantics.

By default, assigning an object to another will copy all of its member variables to each other.

Why is it useful?
Memberwise Agreement and Copy Constructors make it much easier to copy objects around.

As a general rule, C++ prefers copying data over changing data ownership, and copy constructors allow you to control this behavior to make your class more predictable and easy to work with.

Operator Overloading

```
class JoinableArray
{

private:
    vector<double> values;

public:
    // Take an initializer list to pass to
the internal vector
    JoinableArray(initializer_list<double>
inital) : values(inital){};

    // Copy constructor to copy values
    JoinableArray(const JoinableArray &old)
: values(old.values){};

    JoinableArray operator+(JoinableArray
right)
    {
        // Use the copy constructor to copy
all the new values to the output array
        JoinableArray combined = *this;

        // Insert the right hand vector into
the combined vector

combined.values.insert(combined.values.end()
, right.values.begin(), right.values.end());

        return combined;
    }

}


JoinableArray a({3.93, 2.34, 244.0});
JoinableArray b({1.0, 2.0, 3.0});

JoinableArray c = a + b;
// C should have values of { 3.93, 2.34,
244.0, 1.0, 2.0, 3.0 }
```

What does it do?
By declaring a specific operator method on a class, you can redefine how that operator works with your specific class.

Why is it useful?
This can reduce the number of concepts that the programmer needs to know when they are interacting with your class: instead of needing to remember that it has an add function, you can simply use the regular add. This can also help your code feel more natural and be easier to read.

Be aware though: you are overloading standard behavior, which can lead to developer confusion

Questions

1. Which of the following is not true about copy constructor
    a. They're initialized by default
    b. They must must be defined in the program before being used
    c. They can modify the member variables of the object being copied
2. True or False: When using an object as a parameter in a friend function, you must use the indirection operator.
    a. True
    b. False
3. Which of the following is the correct copy constructor for the RemoteFile class with member variables fetched and url?

```
RemoteFile(const RemoteFile &old) {
        fetched = false;
        url = old.url;
    }(A)

RemoteFile(const RemoteFile old) {
        fetched = false;
        url = old.url;
    }(B)
```

Answer Key

1. B. If you don’t declare a copy constructor then one will be initialized by default. However, problems can arise with memory addresses between copied objects and the originals, so a declared copy constructor can help to avoid them
2. B. The friend function needs the memory address of an object. You could enter in an object pointer or use the & symbol to get the address.
3. A. You need the memory address of the file you’re trying to copy.

CPSC 1070 Week 8 Notes

Micheal Merritt, Brendan McGuire

Copy Semantics & Operator Overloading

6Copy Semantics

By default, C++ uses **Copy Semantics**. This means that when you assign one variable to another, by default, C++ will copy that data to a different portion of memory instead of creating two references to the same data.

Why does C++ do this?

Copying data, instead of managing references, significantly reduces the programmer overhead in managing data lifetimes. It's a tradeoff between safety and memory

However, when you have class managing their own memory, this default behavior can introduce some weird undefined behavior. Copying a vector, for example, without properly duplicating the data inside would break the programmer's expectation of copy semantics. That is to say, when a programmer assigns a vector to another value, C++ has taught them that this creates a copy.

```
vector<int> a({ 1, 2, 4, 2, 4 });
vector<int> b = a;
```

So how do we live up to the programmer's expectations?

Copy Constructors

Copy constructors let you define how your class handles that operation above. For example, in a vector, you would want to copy all of the data into a new region of memory.

Below is the actual implementation of the copy constructor for the `std::vector` class, stripped of some preprocessor detail. I know it looks intimidating, but give it a look!

```
vector(const vector& __x) : _Base(__x.size()) {
  this->_M_impl._M_finish =
      std::__uninitialized_copy_a(__x.begin(), __x.end(),
      this->_M_impl._M_start,
      _M_get_Tp_allocator());
}
```

You can see how this copy constructor calls a function to copy the contents of the passed vector into the current vector's memory. This helps to preserve the C++ Copy Semantics, and keep the programmer's expectations consistent.

Operator Overloading

Operator overloading lets you change how specific operators work for your class. By declaring a specific operator method on a class, you can redefine how that operator works with your code.

This can reduce the number of concepts that the programmer needs to know when they are interacting with your class: instead of needing to remember that it has an add function, you can simply use the regular add. This can also help your code feel more natural and be easier to read.

*Be aware though: you are overloading standard behavior, which can lead to developer confusion. Be mindful of the **Principle of Least Surprise**; your code should strive to be as straightforward as possible.*

```
class JoinableArray {

private:
    vector<double> values;

public:
    // Take an initializer list to pass to the internal vector
    JoinableArray(initializer_list<double> initial) : values(initial){};

    // Copy constructor to copy values
    JoinableArray(const JoinableArray &old) : values(old.values){};

    JoinableArray operator(JoinableArray right)
    {
        // Use the copy constructor to copy all the new values to the output array
        JoinableArray combined = *this;

        // Insert the right hand vector into the combined vector
        combined.values.insert(combined.values.end(), right.values.begin(),
right.values.end());

        return combined;
    }

}


JoinableArray a({3.93, 2.34, 244.0});
JoinableArray b({1.0, 2.0, 3.0});

JoinableArray c = a + b;
// C should have values of { 3.93, 2.34, 244.0, 1.0, 2.0, 3.0
```

File Streams

One of the concepts that can make C++ more powerful is the concept of streams. A single abstract idea is able to represent a wide variety of different I/O tasks: reading/writing to standard output, strings, and now, files!

The `fstream` header allows you to parse files as streams, so you can use all the same techniques you used to manipulate standard I/O!

```
#include <fstream>
using namespace std;

ifstream inFile("parse.txt"); // Creates a stream to read a file
ofstream outFile("run.log"); // Creates a stream to write to a file
fstream file("complete.log", ios::out | ios::app); // Creates a stream to append
to a file
```

File Open Methods

The second argument in the constructor for the file stream classes includes a number of flags you can set to customize the behavior of the class in a few specific situations. To include multiple of these fields together, you can use the pipe operator (|) to create a bitmask of all the options you want.

Flag	Meaning
`ios::app`	Creates a file if it doesn't exist, appends to the end of the file if it does
`ios::ate`	Same as append, but you are not allowed to seek to previous parts of the file
`ios::binary`	Opens the file in binary mode
`ios::in`	Opens the file to read
`ios::out`	Opens the file to write
`ios::trunc`	Erases/truncates a file if it exists already

Stream Operators

Like Standard I/O, the best way to interact with `fstream` is using the Stream Insertion (<<) and Stream Extraction operators, because they handle whitespace, serialization, and parsing for you.

Stream Insertion

```
double value;

outFile << "PROGRAM START " << time << endl;
outFile << "value = " << value << endl;
```

Stream Extraction

```
double amount, date, balance;
inFile >> amount >> date >> balance;
```

Other Data Manipulation Methods

Sometimes, the stream manipulators won't work for your use case, so the fstream objects contain some additional methods to help manipulate the file. *These functions are less "magical" than the steam manipulators, which is both a blessing and a curse. You will need to handle serialization, parsing, and whitespace yourself, but you get more granular control about what bytes exactly you are reading and writing.*

`getline(char* s, streamsize n, char delim)`

(Note, if you do not specify delim, it defaults to \n)
Fills the passed buffer from the file, until either it reaches the max number characters n, or it encounters the delimiter (defaults to newline)

```
ifstream logFile("log.txt");
```

`get()`

Retrieves a single character from the file, advancing the file pointer.

```
ifstream logFile("log.txt");
vector<char> chars;
while (!logFile.eof()) {
  chars.push_back(logFile.get());
}
```

peek()

Similar to get, except it does not advance the file pointer, allowing you to “peek” what the next value is, without consuming it

```
ifstream logFile("log.txt");
char start = logFile.peek(); // Gets the first char in the file
```

put(int ch)

Writes a single character to the stream.

```
ofstream logFile("log.txt");
logFile.put('H');
```

seekg(streamoff offset, ios_base::seekdir place)

Moves the file pointer to a specific location, relative to the start, end or current position of the pointer, plus an offset.

```
ifstream logFile("log.txt");
logFile.seekg(0, ios::beg); // Start of file
logFile.seekg(10, ios::cur); // Seek forward 10 bytes
logFile.seekg(-10, ios::end); // 10 bytes from the end of the file
```

read(char *buffer, int numBytes)

Reads a binary chunk of data from ifstream. Need to specify a buffer to fill and a number of bytes to read.

```
ifstream imageFile("config.png", ios::binary);
char header[31];
imageFile.read(&header, 30);
```

write(char *buffer, int numBytes)

Writes a binary chunk to an ofstream. Need to specify a buffer, and the number of bytes to write.

```
ofstream secondFile("second.png", ios::binary);
secondFile.write(&header, 30);
```

Handling Errors

After you perform an operation on a stream class, you can evaluate the truthiness of the stream class to determine whether the operation was successful.
To get more information, you can use the resultant bitmask to determine the exact error. You'll want to use the various getter methods on fstream to determine the exact cause of error

Method	Associated Bit(s)	Explanation
`eof()`	`ios::eofbit`	Bit is set, and method returns true when the End Of File is reached
`fail()`	`ios::failbit` `ios::hardfail`	Returns true when an error (failbit) or an irrecoverable error (hardfail) occurs. This is for a logical I/O error
`bad()`	`ios::badbit`	Returns true when ios::badbit (read/write error) is set
`good()`	`ios::goodbit`	Returns true when ios::goodbit (no errors) is set
`clear()`	`N/a`	Clears set flags

Questions

1. Given that log.txt is an empty file, What will this code block output?

```
fstream logfile;
    logfile.open("log.txt", ios::out | ios::in);

    logfile.put('B');
    char ch = logfile.get();
    cout << ch << endl;
```

   A) Compile Error
   B) -1
   C) B
   D) Nothing.

2. Which of the following must be included for string formatting
   A) <string>
   B) <sstream>
   C) <std>
   D) <fstream>

3. What will the following code block print out

```
int b = strcmp("ABc", "abC");



   if (b < 0)
   {
       cout << "str1 less than str2!" << endl; (A)
   }
   else if (b > 0)
   {
       cout << "str 1 greater than str2!" << endl; (B)
   }
   else
   {
       cout << "str 1 equal to str2!" << endl; (C)
   }
```

   A) A
   B) B
   C) C
   D) Compiler Error

Answer Key

1. D). After using point the “pointer” looking at the while goes past the B character. Thus it reads white space and prints nothing to the terminal.
2. B). Sstream is the correct class to include
3. A) This will not be a compiler error, the result will be a number stored in the b integer. Strcmp looks at the ASCII code rather than comparing string length/characters/cases. ABc is a lower ASCII value than abC (check ch. 12 in the book for a chart).

CPSC 1070 Week 10 Notes

Micheal Merritt, Brendan McGuire

Stacks & Queues

Sometimes, arrays and vectors just don't do it. Stacks and Queues are new data structures for working with data sets, with a focus on how the items are added and removed. A basic explanation of their concept and some common use cases are included.

Stacks (LIFO)

A stack is a last in first out linear collection of elements.

- ➔ Linear collection means it stores a bunch of items in order (like an array)
- ➔ Last-In First-Out means that the most recent item to go in is the first one to come out.
  - ◆ Return address for function calls
  - ◆ Plates in cafeteria

You can find a nice implementation of a stack on here

Queues (FIFO)

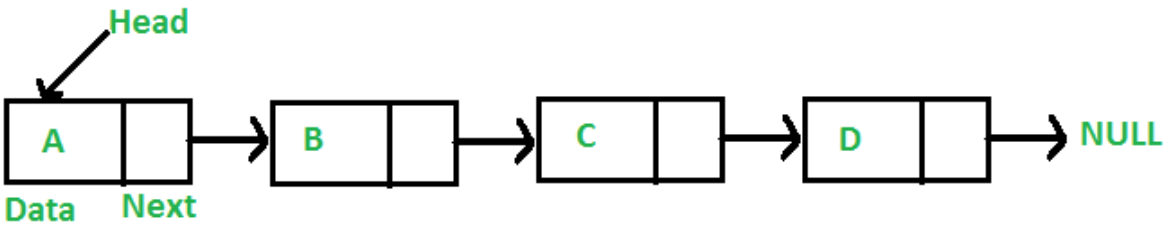
A queue the inverse of a stack, a first in, first out

- ➔ Linear collection means it stores a bunch of items in order (like an array)
- ➔ First-In First-Out means that the earliest item to go in is the first one to come out.
  - ◆ A line in the dining hall
  - ◆ Checklist

An example implementation of a queue can be found here

Linked Lists

A linked list is a way to achieve dynamic-length, iterable lists without need to have a contiguous block of memory. This allows you to quickly expand linked lists without the need for massive block allocations.



This occurs by having a number of elements, with each one pointing to the next (this is a singularly linked list)

Polymorphism

One of the hardest parts of software development is having to predict future requirements, and structure your code to be able to accommodate them effectively.

Polymorphism is a powerful tool for this, because it lets you write code once, and have it work for many different forms (types). In C++, there are a whole bunch of language features that let you write code polymorphically.

Function Overriding & Overloading

Some of the most basic polymorphism in C++ occurs with function overloading and overriding. These concepts allow you to have multiple function definitions for functions with the same name.

Overloading

In many codebases, you will want a function to be able to take many different types of arguments, and potentially have differing behavior to handle each of them.

Function Overloading *lets you define several functions with the same name, but different arguments, and different function bodies!*

```
void debug(double value) {
    cout << "DOUBLE " << value << endl;
};

void debug(int value) {
    cout << "INT " << value << endl;
};

void debug(vector<int> value) {
    cout << "VECTOR" << endl;
    for (auto &item : value) {
        cout << "   " << item << endl;
    };
};
```

In the above example, each function definition has a different argument list, and different implementations. This can be an incredibly useful tool for letting your code be more general & compatible.

Overriding

Overriding is a slightly different concept in C++, having to do with how derived classes deal with inheritance.

Function Overriding *occurs when a derived class redefines how the base class implements a specific method.*

```
class Rectangle {

    private:
        int width;
        int height;

    public:
        Rectangle(int w, int h) : width(w), height(h) {};
        Rectangle() : width(1), height(1) {}

        int area() {
            return width * height;
        }

};

class Square : public Rectangle {

    private:
        int side;

    public:
        Square(int s) : side(s) {};

        int area() {
            return side * side
        };

};
```

In the above example, the derived class Square is redefining what the area method means for Squares. Notice how Rectangle::area() and Square::area() have the same signature. This is what distinguishes overriding from overloading.

Final Keyword

If you do not want a derived class to override a class method, you can final at the end of a prototype to make it inextensible.

```
int Rectangle::area() final;
```

Templates

One thing you may have noticed in the previous example for overloading is that both of the function implementations for int and double were the same.

In fact, there are very many situations where you want your single function body to be able to work with many different types.

- ➔ **Reduce code repeat.** If you ever need to change the debug function, perhaps not to include a newline, having to change it in multiple places can be annoying and error-prone
- ➔ **Handle New or Custom Types**. If you have to write a bespoke implementation of your function for every type that you want to be able to use it, your code base will be extremely long and repetitive, but also quite fragile. Imagine if a new type is introduced or created in your codebase. You want your code to just work with this new object without having to make changes if possible.

The tool that lets you solve this problem is C++ templates. Templates allow you to declare a single function implementation, and have it work for any number of types. Let's take a look at how we can refactor the above example to work for any type that can be outputted to stdout, not just ints and doubles.

```
template <typename T>
void debug(T value) {
   cout << "T " << value << endl;
};

template <typename T>
void debug(vector<T> value) {
   cout << "VECTOR" << endl;
   for (auto &item : value) {
      cout << "   " << item << endl;
   };
};
```

(Slight note, we're using the C++11 style typename specifier as opposed to the older class specifier. For the most part these are equivalent, however there are some slight differences, which you can read about here if you're interested)

We've upgraded our debug function to work with any type that can be printed, as well as all vectors of types that can be printed!

Virtual Functions & Function Binding

Again keeping with this theme of code reuse, let's look at pointers to objects. In C++, you are allowed to use a pointer to a derived class in place of the pointer to the base class. For example, if you have a function that takes a pointer to a Rectangle (above), passing a Square pointer shouldn't be an issue.

Why is this ok?

- ➔ Because of inheritance rules, we know that anything public in the base class is present on the derived class
- ➔ We want to be able to extend the functionality of a class by deriving from it, and still be able to use functions which take the original class

```
void printArea(Rectangle *rect) {
    cout << rect->area() << endl;
};
int main() {
    Square sq(5);
    printArea(&sq); // This is ok!
}
```

However, when you start to override class methods, you can start to run into some weird undefined behavior: you will end up calling the base class method, instead of the derived one! (For more information on why this happens, check out the Monomorphization section at the end of this topic)

```
int Rectangle::area() {
    cout << "RECTANGLE AREA" << endl;
    return width * height;
}

int Square::area() {
    cout << "SQUARE AREA" << endl;
    return side * side;
}
```

```
void printArea(Rectangle *rect) {
    cout << rect->area() << endl;
};

int main() {
    Square sq(5);
    printArea(&sq);
}
```

If you run this example, you will see that “RECTANGLE AREA” will be printed to the console, because printArea takes a pointer to a Rectangle.

So what if we want to use the new Square area method when we pass a Square? We should make area a virtual function to ensure that the function is bound correctly.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call. Note that ensuring correct polymorphism at runtime does incur a slight performance overhead, because the function call is resolved at runtime. For nearly all applications, however, this performance consideration will be inconsequential.

```
virtual int Rectangle::area() {
    cout << "RECTANGLE AREA" << endl;
    return width * height;
}

virtual int Square::area() {
    cout << "SQUARE AREA" << endl;
    return side * side;
}

void printArea(Rectangle *rect) {
    cout << rect->area() << endl;
};

int main() {
    Square sq(5);
    printArea(&sq);
}
```

Pure Virtual Functions & Abstract Classes

Oftentimes in the base class, you will want to define some sort of interface that all derived classes must implement. In C++ this is possible with pure virtual functions. These are class methods which any class deriving must implement themselves, and are defined by writing =0 after the function prototype.

```
class Animal {    public: virtual void id()=0; };
```

Now, any class that derives Animal must implement id. Defining at least one pure virtual function makes Animal an abstract class. These classes define the general interface that all animals must have, however, Animal itself should never be instantiated.

This makes sense: there are no just animals in nature, there are kangaroos, or dogs, or elephants, or birds, or whatever.

Exceptions

Old C-style Exception Handling

To help motivate how C++ style extensions are useful, let's first take a look at how errors in C are handled. Specifically, let's look at File IO.

After every function call, the programmer is expected to check against a global variable (or global function) to see if the function they just called caused an error, handle it as best they could, and then continue function execution if possible. This sounds easy, but can get real verbose real fast.

```
#include <cstdio>

int main() {

    // Open a file
    FILE *handle = fopen("file.txt", "r");

    // wait, gotta check if it opened successfully
    if (ferror(handle)) {
        printf("Something went wrong when opening the file?!");
    }

    // Okay, now we can read from the file
    char ch;
    while ((ch=fgetc(handle)) != WEOF)
        printf("%x ", ch);


};
```

Having to constantly check for errors in this way can make your code pretty hard to read, so C++ introduced a new feature to lump this error checking code together a little more.

C++ Try/Catch

Try catch is a more expressive and automatic means of detecting errors. Inside a try block, you can throw any type, and C++ will unwind the stack, attempting to find an appropriate catch block to handle the type. If C++ cannot find an appropriate try-catch block, then the program may terminate.

This more easily allows for graceful failure than the old C-style system, which is quite brittle.

```
void doFn() {

    try {

        // This would catch to doFn() catch block
        throw 4.0;

        // This would catch to main block
        throw 2;

        // There's no catch block to handle it, program immediately terminates
        throw "o";

    } catch (double d) {
        cout << "DOFN" << d << endl;
    };

};

int main() {

    try {
        doFn();
    } catch (int i) {
        cout << "ERROR " << i << endl;
    }

};
```

Command Line Arguments

When writing programs that users interact with over the command line, accepting program arguments is an important method of input.

In C and C++, command line information is passed to the main function, like so

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

Where argc refers to the number of command line arguments, and argv refers to the array of is character pointers listing all the arguments.

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

    for (int i = 0; i < argc; i++) {
        printf("%s", argv[i]);
    };

};
```

Questions.

1. What occurs when an unhandled exception is present in the program
    a. Compiler Error
    b. The program terminates (Segmentation fault)
    c. Program rewinds to find try statements with matching handles.
2. When does a template function use memory
    a. When the program starts
    b. When it is called
3. Which of the following cases would classify a function as being an **Overridden Function**
    a. Same class, different signatures
    b. Subclass, same signatures
    c. Subclass, different signatures
    d. Same class, same signatures

Answer Key

1. C. The program will trace back through main to find a compatible try handle, if it reaches the end the program will terminate
2. B. When the template function is called
3. B. An overridden function is a subclass specific function that **Overrides** the base classes implementation. That's why it has the same signatures(parameters)