CS590/CPE590

C++ Review

Kazi Lutful Kabir

Spring 2023

Outline

C++ Review:

- Token
- Variables
- Arrays
- Strings
- Command-line Arguments
- Functions
- Pointers
- Class

Tokens

- **Tokens** are the minimal chunk of program that have meaning to the compiler – the smallest meaningful symbols in the language.
- Let's see different kinds of tokens

Token Types	Description/Purpose	Examples
Keywords	Words with special meaning to the compiler	int, class
Identifiers	Name of things that are not built into the language	cout, std, variable-names
Literals	Basic constant values whose value is specified directly in the source code	"Hello World"
Operators	Mathematical or logical operations	+, and/or
Punctuation/Separators	Punctuation defining the structure of a program	{ }, ;
Whitespace	Spaces of various sorts; ignored by the compiler	space, tab, newline, comment

Basic Language Features

Values and Statements

- A **statement** is a unit of code that does something – a basic building block of a program.
- An **expression** is a statement that has a value – for instance, a number, a string, the sum of two numbers, etc.
- **Not** every statement is an expression. E.g., **#include** statement.

Operators

- Operators act on expressions to form a new expression.
  - Mathematical: +, -, *, /, %
  - Logical: and, or, etc.
  - Bitwise: manipulates the binary representation of numbers, e.g., |, ^, <<, etc.

Basic Language Features

Data Types

- Every expression has a type – integer, floating-point, string
- Data of different types take a different amounts of memory to store.
- An operation can be performed on compatible types and normally produces a value of the same type as its

Type Names	Description	Size (byte)	Range
char	Single text character or small integer. Indicated with single quotes ('a', '3').	1	signed: -128 to 127 unsigned: 0 to 255
int	Integer	4	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean (true/false). Indicated with the keywords true and false.	1	Just true (1) or false (0).
double	"Doubly" precise floating-point number.	8	+/- 1.7e +/- 308 (15 digits)

- A *signed* integer is one that can represent a negative number; an *unsigned* integer will never be interpreted as negative.
- There are 3 integer types: short, int, and long, in non-decreasing order of size.
  - memory usage or huge numbers.
- The sizes/ranges for each type are not fully standardized; those shown above are the ones used on most 32-bit computers.

Variable

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

int main () {
    int x ;
    x = 4 + 2;
    cout << x / 3 << ' ' << x * 2;
    return 0;
}
```

```
int main () {
    int x = 4 + 2;
    cout << x / 3 << ' ' << x * 2;
    return 0;
}
```

- Use ***variables*** to give a value a name so we can refer to it later.
- The name of a variable is an *identifier token*. Identifiers may contain numbers, letters, and underscores (_), and *may not start with a number*.
- The **declaration** of the variable x – must tell the compiler what type x will be so that it knows how much memory to reserve for it and what kinds of operations may be performed on it.
- The **initialization** of x – specify an initial value for it. This introduces a new operator: =, the assignment operator.
- A single statement does both declaration and initialization.

Arrays

- An array is a fixed number of elements of the same type stored sequentially in memory.
  type arrayName[dimension];
- The elements of an array can be accessed by using an index into the array.
- Arrays in C++ are **zero-indexed**, so the first element has an index of 0.
- Like normal variables, the elements of an array must be initialized before they can be used.
- The array be multidimensional array.
  type arrayName[dimension1][dimension2];
- Dimensions must always be provided when initializing multidimensional arrays.
- Multidimensional arrays are merely an abstraction for programmers, as all of the elements in the array are sequential in memory.

```
int arr[4];

arr[0] = 6;
arr[1] = 0;
arr[2] = 9;
arr[3] = 6;

int arr[4] = {6, 0, 9, 6};

int arr[] = {6, 0, 9, 6};
```

```
int twoDimArray[2][4] = { 6, 0, 9, 6, 2, 0, 1, 1 };
int twoDimArray[2][4] = { { 6, 0, 9, 6 } , { 2, 0, 1, 1 } };
```

Strings

- String literals such as “Hello, world!” are represented by C++ as a sequence of characters in memory. In other words, a string is simply a character array and can be manipulated as such.

```
char helloworld[] = { 'H', 'e', 'l', 'l', 'o', ',', ' ',
                      'w', 'o', 'r', 'l', 'd', '!', '\0' };
```

- The character array helloworld ends with a special character, ‘\0’, known as the **null** character.
- Character arrays can also be initialized using string literals.

```
char helloworld[] = "Hello, world!"
```

- The individual characters in a string can be manipulated either directly by the programmer or by using special functions provided by the C/C++ libraries. These can be included in a program using the **#include** directive:
  - cctype (ctype.h): character handling
  - cstdio (stdio.h): input/output operations
  - cstdlib (stdlib.h): general utilities
  - cstring (string.h): string manipulation

Strings

```
#include <iostream>
#include <cctype>
using namespace std;

int main() {
    char messyString[] = "t6H0I9s6.iS.999a9.STRING";
    char current = messyString[0];
    for(int i = 0; current != '\0'; current = messyString[++i]) {
        if(isalpha(current))
            cout << (char)(isupper(current) ? tolower(current) : current);
        else if(ispunct(current))
            cout << ' ';
    }
    cout << endl;
    return 0;
}
```

- The *isalpha* functions check whether a given character is an **alphabetic character**, an **uppercase letter**, or a **punctuation character**, respectively.
- These functions return a *Boolean* value of either true or false.
- The *tolower* function converts a given character to lowercase.

- The for loop takes each successive character from messyString until it reaches the *null character.*

Strings Functions

- strcpy(): copies one string into another
- strcat(): concatenates two functions
- strlen(): returns the length of a function
- strcmp(): compares two strings

Command Line Arguments

Let's look at the main function:

It contains parameters now!

```
#include <iostream>
#include <sstream>
using namespace std;

/*
 * Computes the max of two integers m and n.
 */
int max(int m, int n) {
  return m > n ? m : n;
}

int main(int argc, char* argv[]) {
  int m, n;
  stringstream iss; // input string stream variable

  if(argc != 3) {
    cerr << "Usage: " << argv[0] << " <integer m> <integer n>" << endl;
    return 1;
  }
  iss.str(argv[1]);
  if(!(iss >> m)) { // Read one integer from iss and check for failure too.
    cerr << "Error: the first argument is not a valid integer" << endl;
    return 1;
  }
  iss.clear(); // Remember to clear iss before using it with another string!
  iss.str(argv[2]);
  if(!(iss >> n)) {
    cerr << "Error: the second argument is not a valid integer" << endl;
    return 1;
  }
  cout << "m is: " << m << endl;
  cout << "n is: " << n << endl;
  cout << "max(" << m << ", " << n << ") is: " << max(m, n) << endl;
  return 0;
}
```

What is `#include <sstream>` ?

- **sstream**: stands for string stream in C++; it associates a string object with a string -- using this we can read from string as if it were a stream like **cin**

- **sstream** class is extremely useful in parsing input

- Basic Methods:
  -- **clear()** : clear the stream
  -- **str()** : get and set string object whose content is present in the stream
  -- **operator <<** : add a string to the stringstream object
  -- **operator >>** : read something from the stringstream object.

What is `argc` and `argv` ?

- Command-line arguments are provided after the name of the program in command-line shell of OS

- Passing command line arguments
  -- define **main(int argc, char *argv[])** with two arguments:
  1. the number of command line arguments
  2. the list of command-line arguments

- **argc** (ARGument Count): stores number of command-line arguments passed by the user including the name of the program

- **argv**(ARGument Vector) is array of character pointers listing all the arguments

argc and argv

argv is an array of pointers to strings:

```
        +-----+
        |     |      +---+---+---+---+---+---+---+
argv[0] |   -------->| m | a | x | t | w | o | \0|
        |     |      +---+---+---+---+---+---+---+
        +-----+
        |     |      +---+---+---+
argv[1] |   -------->| 2 | 3 | \0|
        |     |      +---+---+---+
        +-----+
        |     |      +---+---+---+
argv[2] |   -------->| 5 | 7 | \0|
        |     |      +---+---+---+
        +-----+
```

argc is equal to 3: 1 for the program name and 2 arguments given to the program on the comand line.

Functions

Advantages

- Readability: sqrt(5) is clearer than copy-pasting in an algorithm to compute the square root, e.g., 5^(0.5).
- Maintainability: To change the algorithm, just change the function (vs changing it everywhere you ever used it).
- Code reuse: Lets other people use algorithms you've implemented.

Function Declaration Syntax

Return Type

Function Name

Argument

Signature

Body

```
int raiseToPower(int base, int exponent)
{
 int result = 1;
 for (int i = 0; i < exponent; i = i + 1) {
 result = result * base;
 }
 return result;
}
```

Functions

Return Values

Return Type | Function Name | Argument | Signature | Body | Return

```
int raiseToPower(int base, int exponent)
{
 int result = 1;
 for (int i = 0; i < exponent; i = i + 1) {
 result = result * base;
 }
 return result;
}
```

- Up to one value may be returned; **it must be the same type as the return type**.
- If no values are returned, give the function a void return type
  - Note that you cannot declare a variable of type void
- Return statements don't necessarily need to be at the end.
- Function returns as soon as a return statement is executed.

```
void printNumberIfEven(int num) {
    if (num % 2 == 1) {
    cout << "odd number" << endl;
    return;
    }
    cout << "even number; number is " << num << endl;
}
```

Functions

```
int foo() {
    return bar()*2
}

int bar() {
    return 3;
}
```

```
int square(int z);
int cube(int x){
    return x*square(x);
}
int square(int x):{
    return x*x
}
```

- Function declarations need to occur **before invocations**.
  - Solution 1: reorder function declarations
  - Solution 2: use a function **prototype** to inform the compiler that you'll implement it later
- Function prototypes should match the signature of the method, though argument names don't matter

- Function prototypes are generally put into separate header files
  - Separates specification of the function from its implementation

Functions

Recursion

```
int fibonacci(int n){
    if (n==0 || n == 1){
    return 1;
    } else {
        return fibonacci(n-2) + fibonacci(n-1)
    }
}
```

- Functions can call themselves.
- Fib(n) = fib(n-1)+fin(n-2) can be easily expressed via a recursive implementation.

Functions

Scope

- the extent up to which something can be worked with.
  - Where a variable was declared, determines where it can be accessed from
  - **numCalls** has global scope – can be accessed from any function
  - **result** has function scope – each function can have its own separate variable named **result**

```
int raiseToPower(int base, int exponent){
 numCalls = numCalls + 1;
 int result = 1;
 for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
 }
 return result;
}

int max(int num1, int nume2){
    numCalls = numCalls +1;
    int result;
    if (num1 > num2) {
        result = num1;
    }
    else {
        result = num2;
    }
    return result;
}
```

Functions

Scope

```
double squareRoot(double num) {
    double low = 1.0;
    double high = num;
    for (int i = 0; i < 30; i = i + 1) {
        double estimate = (high + low) / 2;
        if (estimate*estimate > num) {
            double newHigh = estimate;
            high = newHigh;
        } else {
            double newLow = estimate;
            low = newLow;
        }
    }
  return estimate;
}
```

Cannot access variables that are out of scope

C++ Pointers

- Every memory cell has a value (the content of the cell) and an address (a location in memory)

- Since humans are not good at remembering numerical addresses, we prefer to use variable names instead

- **A pointer is a variable that stores as its value the address of another variable**

```
int x = 5;
int *z;
z = &x;
```

```
            RAM
        +---------+
      0 |         |
        +---------+
        |         |
           ...
        |         |
        +---------+
   1992 |         |
        +---------+
 x 2000 |    5    | <--+
        +---------+    |
   2008 |         |    | points at
        +---------+    |
 z 2016 |  2000   | ---+
        +---------+
   2024 |         |
        +---------+
```

Distinguish among

-- pass by value

-- pass by reference

-- pass by pointer

```
#include <iostream>
using namespace std;
void pass_by_value(int k) {
k = 10;
}
void pass_by_pointer(int *k) {
*k = 10;
}
// & is not the "address of" operator here, it is just a notation
// to indicate that k is passed by reference.
void pass_by_reference(int &k) {
k = 10;
}

int main() {
int x;  // Type: integer
x = 5;  // Store the integer 5 into x.
int *z; // Type: pointer to integer
z = &x; // Store the address of integer x into z.
cout << x << " " << &x << " " << z << " " << &z << endl;
cout << "*z is: " << *z << endl;
*z = 7; // Same as: x = 7
cout << x << " " << &x << " " << z << " " << &z << endl;
cout << "*z is: " << *z << endl;
x = 5;
pass_by_value(x);
cout << "x is: " << x << endl;
pass_by_pointer(&x);
cout << "x is: " << x << endl;
x = 5;
pass_by_reference(x);
cout << "x is: " << x << endl;
return 0;
}
```

Pass by Value

- In Pass By Value, the value of the **integer x** given as argument to the function call is copied into the **integer k** at the start of the function call. After that **x** and **k** are independent of each other so modifying k does not modify x.

```
                ...
          |            |
          +--------+
    0992  |            |
          +--------+
  k 1000  |     5      |  <--+
          +--------+         |
    1008  |            |     |
          +--------+         |
          |            |     |
                ...          |  copy
          |            |     |
          +--------+         |
    1992  |            |     |
          +--------+         |
  x 2000  |     5      |  ---+
          +--------+
    2008  |            |
          +--------+
          |            |
                ...
```

Pass by Pointer

- In Pass By Pointer, the **address of integer x** given as argument to the function call is copied into the pointer k at the start of the function call. Since k points **at x (k contains the address of x), *k is then the same as x itself, so changing *k changes x**, even though x and k are defined in different functions.
- **Modifying a variable defined in a different function** is one of the two main reasons why pointers are very useful.

```
              . . .
         |           |
         +-----------+
   0992  |           |
         +-----------+
k  1000  |   2000    |  ---+
         +-----------+     |
|  1008  |           |     |
|        +-----------+     |
|        |           |     |
|  copy     . . .          |  points at
|        |           |     |
|        +-----------+     |
\  1992  |           |     |
 \       +-----------+     |
x  2000  |     5     |  <--+
         +-----------+
   2008  |           |
         +-----------+
         |           |
              . . .
```

Pass by Reference

- In Pass By Reference, the **variable x given as argument to the function call is aliased with the new name k** at the start of the function call. Since **k is just another name for x**, changing k changes x, even though x and k are defined in different functions.
- Internally, the C++ compiler implements Pass By Reference by automatically rewriting your code to use Pass By Pointer, so Pass By Reference is just a nice notation provided to you by C++ for your convenience.

```
                 ...
            |          |
            +----------+
     1992   |          |
            +----------+
x    2000   |    5     |  k
            +----------+
     2008   |          |
            +----------+
            |          |
                 ...
```

Dynamic Memory

- The values pointer points at the first element of an array of integers which is dynamically allocated from using the "new" operator:

int *values = new int[x];

```
                    ...
                |         |
                +---------+
         1992   |         |
                +---------+
values   2000   |  34992  |  ---+
                +---------+     |
         2008   |         |     |
                    ...         |
                                | points at
                    ...         |
                |         |     |
                +---------+     |
        34992   |         | <--+
                +---------+
        35000   |         | <--- values + 1
                +---------+
        35008   |         | <--- values + 2
                +---------+
        35016   |         | <--- values + 3
                +---------+
        35024   |         | <--- values + 4
                +---------+
                |         |
                    ...
```

```cpp
#include <iostream>
using namespace std;
// The array parameter is in fact a pointer but we can use it
// as if it were the name of the array itself.
void display_array(int array[], int length) {
for(int i = 0; i < length; i++) {
cout << array[i] << " ";
}
cout << endl;
}
// The array parameter is a pointer and we use pointer arithmetic.
void display_array_ptr(int *array, int length) {
for(int *p = array; p < array + length; p++) {
cout << *p << " ";
}
cout << endl;
}
```

```cpp
int main() {
int x;
x = 15;
// Pointer to an anonymous variable-length array which is dynamically allocated
int *values = new int[x];
for(int i = 0; i < x; i++) {
// Using the pointer as if it were the name of the array:
values[i] = i;
// or using pointer arithmetic (which is what the CPU really does):
*(values + i) = i;
}
// Using the pointer as if it were the name of the array:
display_array(values, x);
display_array_ptr(values, x);
// Deleting the array (the pointer is not modified):
delete [] values;
return 0;
}
```

Dynamic Memory

- The array itself has no name so its elements can only be accessed through the values pointer. The array elements can then be accessed in two different (and equivalent) ways:

  1) either by using the values pointer as if it were the name of the array itself, plus using the usual array notation: **values[0], values[1], values[2]**, etc.

  2) by using pointer arithmetic: values, values + 1, values + 2, etc., are pointers pointing at the different elements of the array, and therefore ***values, *(values + 1), *(values + 2)**, etc., are the array elements themselves.

- In fact, internally the C++ compiler automatically transforms **values[i]** into ***(values + i)** which is what the computer's CPU then uses to access the array elements in memory.

- **Accessing dynamically allocated memory** is the other one of the two main reasons why pointers are very useful.

```
                    ...
                |         |
                +---------+
         1992   |         |
                +---------+
  values 2000   | 34992   |  ---+
                +---------+     |
         2008   |         |     |
                    ...         |
                                | points at
                    ...         |
                |         |     |
                +---------+     |
        34992   |         |  <--+
                +---------+
        35000   |         |  <--- values + 1
                +---------+
        35008   |         |  <--- values + 2
                +---------+
        35016   |         |  <--- values + 3
                +---------+
        35024   |         |  <--- values + 4
                +---------+
                |         |
                    ...
```

Class

A **user-defined datatype** which groups together related pieces of information.

```
class Employee {
    int id;
    string name;
    float salary;
};
```

Fields indicate what related pieces of information our datatype consists of – Another word for field is members.

Fields can have different types.

Access Modifier:

- Private: can be accessed within the class (default)
- Public: can be accessed from anywhere

Class Definition

As a rule of thumb, data members should be declared **private.** Member functions should be declared **public**.

```
class class_name {
 private:
        // data members
 public:
        // constructor and destructor
        // member functions
};
```

Objects

- Class definition and declaration
  - Once a class has been defined, it can be used as a type in object, array and pointer declarations
  - Example:

```
class Employee {
      int id;
      string name;
      float salary;
 };
```

```
Employee e1; //creating an object of Employee
Employee e2;
```

Constructors

- *Constructor:–* a function used to initialize the data of an object of a class
  - Same name as class itself
  - Cannot return anything, not even **void**
  - A class may define more than one *constructor*
    - With different parameter lists
    - Default constructor has no parameters
- Called automatically
  - When class object is declared as automatic variable
  - By **new** operator

Compiler provides one, if you do not!

Compiler's default simply calls constructors of data members of the class

Destructors

- *Destructor:*– a function used to clean up an object of a class prior to deleting that object
  - Class name preceded by **'~'**
  - No parameters, no result

Compiler provides one if you do not!

- Called automatically
  - When function exits scope of automatic class object

Compiler's default simply calls destructors of data members of the class.

Constructors and Destructors

- Constructors – Similar to Java
- Destructors – No counterpart in Java
- Purpose of Destructors
  - Free dynamic storage pointed to only by members of object
  - Reduce reference count when object disappears
  - Safely close things – e.g., files

Constructor Overloading (Polymorphism)

- Special case of function overloading
- Function overloading
  -- functions can have the same name
  but differ in number/type of arguments

```
void sum(int a, int b)
{
  cout << "Result = " << (a + b);
}

void sum(double a, double b)
{
    cout << endl << "Result = " << (a + b);
}

void sum(int a, int b, int c)
{
    cout << endl << "Result = " << (a + b + c);
}

// main function
int main()
{
    sum(10, 2);
    sum(5.3, 6.2);
    sum(1, 2, 3);

    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

class Point {
private:
  int x_, y_;
  float z_;

public:
    Point(){
    x_ = 0;
    y_ = 0;
    z_ = 0.1;
    }
  //Constructor that uses initializer list (constructor overload
  Point(int x, int y, float z){
     x_ = x;
     y_ = y;
     z_ = z;
  }
```

```cpp
  //Method that do not modify member variables
  void print_coords() const{
    cout << "(x,y,z)=(" << x_ << "," << y_ << "," << z_ << ")" << endl;
  }

  //Mutator
  void set_x(int x){
    x_ = x;
  }

  void set_y(int y){
    y_ = y;
  }

  void set_z(float z){
    z_ = z;
  }

  //Accessor
  int get_x() const {
    return x_;
  }
  int get_y() const {
    return y_;
  }
  float get_z() const {
    return z_;
  }
};
```

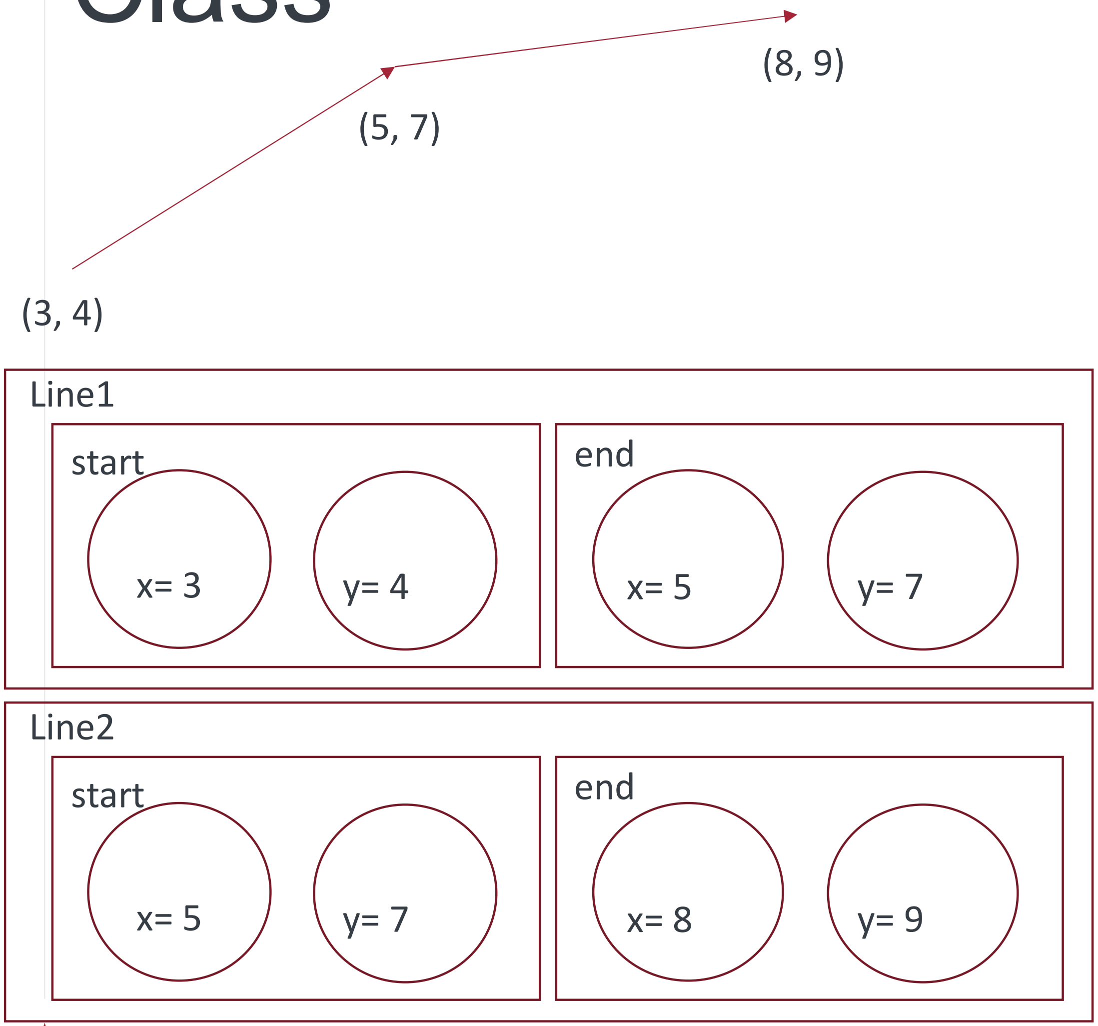
```cpp
int main(){
  Point point1(5,7, 1.985);
  Point point2(1,2,3);
  Point point3;

  point2.set_x(10);
  point3.set_x(15);

  point1.print_coords();
  point2.print_coords();
  point3.print_coords();

  return 0;
}
```

Class



Practice:

- A point consists of an x and y coordinate
- A line consists of 2 points: a start and a finish
- Assigning instances for fields

Class

```
class Point{
 public:
    double x, y;
};

class Line{
 public:
    Point start, end;
};

int main(){
    Line l1, l2;
    l1.start.x = 3.0;
    l1.start.y = 4.0;
    l1.end.x = 5.0;
    l1.end.y = 7.0;
    l2.start = l1.end;
    l2.end.x = 8.0;
    l2.end.y = 9.0;
    return 0;
}
```

Object-Oriented Programming (OOP) and Inheritance

Classic "procedural" programming languages before C++ structure programs as

1. Split it up into a set of tasks and subtasks
2. Make functions for the tasks
3. Instruct the computer to perform them in sequence

With large amounts of data and/or large numbers of tasks, this makes for complex and unmaintainable programs

OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more abstractly and focus on the interactions between them.

- **Encapsulation**: grouping related data and functions together as objects and defining an interface to those objects
- **Inheritance**: allowing code to be reused between related types
- **Polymorphism**: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type

Conclusion

C++ Review
- A good starting point if you did not have exposure to C++ before.

Math Review
- Knowing general properties of series, summations, exponents, and logarithms will be helpful.

Next Week
- Sorting and Complexity
- First Homework/Programming Assignment

Contents of this presentation are partially adapted from
Prof. In Suk Jang CS590 (Summer 2021 Lecture-2)
And from
My CS385 (Spring 2023 Lecture-4)

STEVENS
INSTITUTE OF TECHNOLOGY
1870

THANK **YOU**

Stevens Institute of Technology