C++ Guide

Do not distribute!

C++ Logic

Imports	`#include <iostream>`
	
	`#include <string>`
	
	`using namespace std;`
		• Put at the BEGINNING of each file.
		• **IOstream**: need in order to use cout / cin
		• **String**: need in order to use string
		• **Using namespace std;** need in order to shorten code (otherwise would be std::cout instead of just cout)

Main Function	`int main() {`
	
	`  // write code here !`
	
	`}`
		• What your program will actually execute.
		• You must have exactly **one** main function per project.
		• You only can write code **within functions** (with few exceptions), so make sure all your code is either within the main method or another function

Comments	`// this is a comment` `/* this is a multiline comment that takes up more space than a single-lined comment */`	• Your computer will ignore this code, so you can write whatever you want in here !
Output	`cout << “hello!”;` `cout << “waddup!” << endl;`	• Prints out to the screen for you to read • Think “C-out” as in printing **out** to the screen • endl; adds a new line after your print statement, so your code will be easier to read

Variables	`int num = 5;`
	`string name = “Snooze”;`
	`bool isCool = true;`
	`float pi = 3.14159265;`
		• Way to store information
		• Think of a variable as a **box** that stores certain information
		• Type: what kind of information the variable stores
		• Name: name of variable (think: name of box!)
		• Value: the information that the variable stores (think: what you put into the box)
		• In order to **access** the value of the variable, we call the variable’s **name** (eg: cout << num)

Operators	`+-*/`
	`%`
	`!`
	`++, --`
		• Same way as arithmetic operators
		• %: returns remainder, eg: 18 % 4 = 2, 21 % 5 = 1
		• ! returns not, or the opposite of
		• ++, --: equivalent of += 1, -= 1

Comparators	`>, >=, <, <=, ==, !=`	• > greater than • >= greater than or equal to • < less than • <= less than or equal to • == equals (note that there are **two** equal signs – remember, one equal sign means declare) • != does not equal

Variables (cont)	`int x = x + 1;`	• Reads from **right to left** • If x was initially 5, the right will be 5 + 1, and 6 will now be put into x
Input	`cin >> num;`	• User can set the value of the variable num • Think: “C-in” as in user puts information **into** the computer

Conditionals	`if (x < 5) {`
	`  cout << “so low!” << endl;`
	`} else if (x < 8) {`
	`  cout << “ok” << endl;`
	`} else {`
	`  cout << “so high!” << endl;`
	`}`
		• Use when you want the program to do different things depending on the conditions
		• Inside the parenthesis is a condition, has to return a true or false value

Switch statements	`int choice`
	`switch (choice) {`
	`case 1:`
	`  cout << “no” << endl;`
	`  break;`
	`case 2:`
	`  cout << “yes” << endl;`
	`  break;`
	`default:`
	`  cout << “maybe” << endl;`
	`  break;`
	`}`
		• Used to simplify if/else statements
		• Default case: equivalent of else case
		• Left code is the equivalent of:
		
		`if (choice == 1) {`
		`  cout << “no” << endl;`
		`} else if (choice == 2) {`
		`  cout << “yes” << endl;`
		`} else {`
		`  cout << “maybe” << endl;`
		`}`

While Loops	`while (condition) {`
	`  code;`
	`}`
		• Need to repeat a block of code for some amount of time, not always known

For Loops	`for (int i = 0; i < num; i++) {`
	`  code;`
	`}`
		• Repeat a set number of times
		• **Iterator**: declares a variable that counts number of iterations (times) the loop has run. Usually starts on 0
		• **Condition**: loop runs until this condition is false, contains the same variable as in iterator
		• **Incrementor**: after ea/iteration, the variable updates/increments, usually by 1 (++)

Jump Statements	`break;` `continue;`	• **Break**: exits the loop entirely • **Continue**: skips the code below it, but continues the loop

Opening / Closing Files	`#include <fstream>`
	
	`int main() {`
	`  fstream fs;`
	`  fs.open("save.txt",`
	`ios::out);`
	`  if (fs.is_open()) {`
	`  cout << "File opened`
		• Way to save information locally, since computer regularly deletes/restarts info on the actual compiler
		• `#include <fstream>` imports the necessary functions needed
		• `fs` is of type fstream and helps the compiler interact with local text files
		• `fs.open(“save.txt, ios::out)` opens up

	`successfully!" << endl;`
	`  }`
	`  fs.close();`
	`  return 0;`
	`}`
		the text file save.txt, or it creates a new txt file if save.txt doesn't exist
		○ Note ios::out means we're writing **out** to the file, so we can't read the file yet
		● `fs.is_open()` checks if file is open and returns a true/false value
		● `fs.close()` closes the file after opening it

Writing to a File	`#include <fstream>`
	
	`int main() {`
	`  fstream fs;`
	`  fs.open("test.txt",`
	`ios::out);`
	
	`  if (fs.is_open()) {`
	`  fs << "test!" << endl;`
	`  }`
	`  fs.close();`
	`}`
		● Opens file test.txt if exists, or creates a test.txt if doesn't exist
		● Inputs `test!` into the text file
		 
		●

Reading a file	`#include <fstream>`
	
	`int main() {`
	`  fstream fs;`
	`  fs.open("test.txt", ios::in);`
	
	`  string readWord;`
	`  string readLine;`
	
	`  if (fs.is_open()) {`
	`  // read first word`
	`  fs >> readWord;`
	`  cout << readWord;`
	
	`  // read first line`
	`  getline(fs, readLine);`
	`  cout << readLine;`
	`  }`
	
	`  fs.close();`
	`}`
		● Opens file test.txt if exists, or creates a test.txt if doesn't exist
		● readWord will have value of the first word
		● readLine will have value of the first line
		● if the program above is run before this program is run, the output onto terminal will be `test!`

Random Number Generation	`#include <random>`
	
	`int main() {`
	`  random_device rd;`
	`  rd() % 6 + 1;`
	`}`
		● `rd()` returns a random integer
		● `rd() % 6 + 1` returns a random integer from 1-6

Functions	`type name(type1 p1, type2 p2) {`
	`  // insert code here`
	`  return var;`
	`}`
		● block of code that performs a specific action
		● useful so you don't have to repeat code
		● **type**: what type the function returns. eg: int, string, bool, void (used when function doesn't

	`int addOne(int a) {`
	`  return a + 1;`
	`}`
	
	`void repeatString(string a) {`
	`  cout << a << a << endl;`
	`}`
		return a value)
		• **name**: name of function
		• **parameters**: inputs into function, aka what the function takes in. as many or few as needed, and different/same types
		• **code**: the function performing the action
		• **var**: function has to return a variable of type **type**

Calling a Function	`funcName() { ... }`
	`addOne(int x) { ... }`
	`repeatString(string s) { ... }`
	
	`int main() {`
	`  funcName();`
	
	`  addOne(1); // returns 2`
	`  repeatString("hello");`
	`  // prints "hellohello"`
	`}`
		• Function only performs action when called
		• Call function in main or within another function
		• `a, b` within `()` must match function's original parameters

Arrays	`int array[size];`
	`int nums[4] = {1, 2, 3, 4};`
	`nums[2] = 100;`
		• Good for storing a collection of items of the same type
		• Fixed size (can't modify size of array after)
		• Element: individual value in array
		• **Declaring array**: `int list[size];`
		• **Initializing values**: `int list[2] = {1, 2};`
		• **Changing individual elements**: `list[0] = 0;`

Vectors	`#include <vector>`
	`vector<int> list;`
	`list = {1, 5};`
	`list.push_back(6);`
	
	`// list now contains: [1, 5, 6]`
	
	`list.size(); // returns 3`
	`list.resize(1);`
	`// list now contains [1]`
	
	`list.insert(0, 2);`
	`list.insert(2, 5);`
	`// list now contains [2, 1, 5]`
	
	`list.erase(list.begin());`
	`// list now contains [1, 5]`
		• Need to import vector library
		• Mutable size (can add or delete elements in vector)
		• **Initializing vector**: `vector<type> listName;`
		• **Declaring known values**: `listName = {1, 2, 3};`
		• **Adding value to end of vector**: `list.push_back(value)`
		• **Looping through vector**:
		• `for (int element : list) {`
		`  doSomething();`
		`}`
		• **Accessing specific element**: `list[0];`
		• **Size**: `list.size();`
		• **Reize**: `list.resize();`
		• **Delete**: `list.erase(list.begin() + 2);`
		• **Insert**: `list.insert(list.begin() + 1, 10);`

String Operators	`string place = "Berkeley";`
	`int index = place.find("ke");`
	`// index contains 3`
	
	`index = place.find("e");`
	`while (index != -1) {`
		• New line: `\n`
		• **Finding 1st index of where substring occurs:** `index = varName.find(substring);`
		• **Finding all indices of where substring occurs, using a while loop**:

	`cout << index << endl;`
	`place.replace(index, 1, “a”);`
	`index = place.find(“e”);`
	`}`
	`// prints 1, 4, 6`
	
	`place.replace(5, 2, “dogg”);`
	`cout << place << endl;`
	`// prints Barkadoggy`
		`while(index != -1) {`
		`place.replace(index, 1, “”);`
		`index = varName.find(substring);`
		`}`
		• **Replace k characters with newString starting from index:** `varName.replace(index, k, newString);`

Classes	`class Dog {`
	`public:`
	
	`string name;`
	`int age;`
	`static const int numLegs = 4;`
	
	`Dog(string n, int a) {`
	`name = n;`
	`age = a;`
	`}`
	
	`static void bark() {`
	`cout << “Woof!” << endl;`
	`}`
	
	`string getName() {`
	`return name;`
	`}`
	`};`
		• **Class**: classification or type, eg: Dog, Student, Car, etc.
		• **Dog**: class name, class declaration
		• **Specifier**: privacy of class. Can be public, private, or protected
		• **Instance Variables**: properties of this class. can be different for each individual (non-static), or the same for all instances of this class (static)
		• **Constructor**: how the Dog object is created. Each undeclared instance variable **must** be defined in the constructor
		• **Methods**: Functions for the class. Can be non-static (different for every instance in class, such as name) or static (same for every instance in class, such as bark)

Objects	`class Dog {`
	`...`
	`}`
	
	`int main() {`
	`Dog d = Dog(“Spots”, 5);`
	`cout << d.getName() << endl;`
	`// prints “Spots”`
	
	`d.bark();`
	`// prints “Woof!”`
	`}`
		• **Object**: individual instance of class, eg: Spots (type Dog), Joe (type Student), myCar (type Car)
		• Create an object in the main class by calling the **constructor** function (see above)
		  ○ Make sure the arguments within your call match the parameters in the function
		• Use instance variables to call both static and non-static methods

Scrolling & Time	`#include <thread>`
	`#include <chrono>`
	
	`void delay(int delayMS) {`
	`this_thread::sleep_for(chrono`
	`::milliseconds(delayMS));`
	`}`
		• Import chrono and thread libraries
		• use delay(#milliseconds you want to delay) to pause the screen temporarily
		• you can write a delay scroll function by using repeated cout << endl;, with delay() in between
		`void delayScroll(int delayMS, int lineCount) {`

		`for (int i = 0; i < lineCount; i++) {`
		`  cout << "" << endl;`
		`  delay(delayMS);`
		`  }`
		`}`

Pointers and References	`int a = 5;`
	`cout << “a:” << &a << endl;`
	`// a`
	`// prints “10234013” or some other 8-digit memory space`
	
	`int* a_pointer = &a;`
	`cout << a_pointer << endl;`
	`// a_pointer, aka &a`
	`// prints “10234013” or some other 8-digit memory space`
	
	`cout << *a_pointer << endl;`
	`// *a_pointer, aka a`
	`// prints 5`
		• Use to find memory (where in computer) each variable or object is stored
		• & before a variable will return its memory location
		• * after a variable type indicates the variable is a **pointer**, which will point to a variable’s **memory location** instead of its value
		  ○ **NOTE**: different than * below, which is used for dereferencing
		  ○ pointers always take in type &variable, since it stores a memory location
		• **Dereferencing**: * before a **pointer** variable **dereferences** it, or finds the actual value of whatever its pointing to.
		  ○ eg. if you have `int a`, `int* a_pointer = &a`, then `*a_pointer` is equivalent to `*&a`, which is equivalent to `a`

Functions: Extra	`int addOne(int x) {`
	`  return x + 1;`
	`}`
	
	`int main() {`
	`  int x = 5;`
	`  addOne(x);`
	`  cout << x << endl;`
	`  // prints out 5, not 6`
	`}`
		• Parameters vs. Arguments
		  ○ **Parameters**: in the function initialization
		  ○ **Arguments**: in the function call
		• Arguments are passed in by value if it is a primitive type– in the eg. to the left, the **value 5** is passed into the function, not the actual variable **x**. So, x remains unchanged and the function

C++ Graphics (SFML)

Setup	`#include <SFML/Graphics.hpp>`
	`using namespace sf;`
	
	`RenderWindow window(VideoMode(400, 200), "Blank Canvas");`
	
	`while (window.isOpen()) {`
	`  // code`
	`}`
		• Imports SFML library
		• using namespace sf; ← allows you to write SFML code without having to write sf:: in front of everything
		• renders window of width 400, height 200
		• Window titled “Blank Canvas”
		• **while loop**: Opens window so window doesn’t close immediately

Window Events	`Event e;`
	`// initialization of Event e`
	
	`Event::KeyPressed`
	`Event::KeyReleased`
		• **Events**: stuff that happens within program that act as a “trigger” for another action happening
		• **KeyPressed**: when user presses a key
		• **KeyReleased**: when user releases a key
		• **Closed**: when user closes the window

	`Event::Closed`
	`Event::Resized`
	`sf::Event::TextEntered`
	`sf::Event::LostFocus`
	`sf::Event::GainedFocus`
	`sf::Event::MouseWheelScrolled`
	`sf::Event::MouseMoved`
	`sf::Event::MouseButtonPressed`
	`sf::Event::MouseButtonReleased`
	`sf::Event::MouseEntered`
	`sf::Event::MouseLeft`
		• **Resized**: when user resizes window
		• **TextEntered**: catch input in a text field
		• **LostFocus**: when user switches out of window
		• **GainedFocus**: when user switches back into window
		• **MouseWheelScrolled**: when mouse wheel moves
		• **MouseMoved**: when mouse moves within the window
		• **MouseButtonPressed**: when mouse button is pressed
		• **MouseButtonReleased**: when mouse button is released
		• **MouseEntered**: when mouse enters window
		• **MouseLeft**: when mouse leaves window (good for needing to pause the game)
		• More documentation here: https://www.sfml-dev.org/tutorials/2.5/window-events.php

Event Key Codes	`e.key.code`
	
	`sf::Keyboard::A`
	`sf::Keyboard::5`
	`sf::Keyboard::shift`
	`sf::Keyboard::Escape`
	
	`if (e.key.code ==`
	`sf::Keyboard::A) {`
	`  // insert code`
	`}`
		• **e.key.code**: which key is pressed
		• **sf::Keyboard::X**: the key X. can replace X with any other key (eg. A-Z, 0-9, etc.)

Event Functions / Set-up Code	`while (window.isOpen()) {`
	`  Event e;`
	
	`  while (window.pollEvent(e)) {`
	`  // insert code`
	`  }`
	
	`  window.clear();`
	`  // insert drawing code`
	`  window.display();`
	`}`
		• **isOpen()**: whether or not the window is open
		• **clear()**: clears everything on window
		• **display()**: displays whatever user draws
		• **close()**: closes the window
		• **e.type**: the type of event e is

Polling Events	`while (window.pollEvent(e)) {`
	`  switch (e.type) {`
	`  case sf::Event::Closed:`
	`  window.close();`
	`  break;`
	
	`  case sf::Event::KeyPressed:`
	`  // insert code here`
	`  break;`
		• Events are mutually exclusive (only ONE event can be happening at a time)
		• Usually use a switch statement to match event to what's happening

	`  default:`
	`  break;`
	`}`
	
	`}`	

Drawing Circles	`sf::CircleShape c;`
	`c.setRadius(50);`
	`c.setPosition(200, 100);`
	`c.setFillColor(sf::Color::Red);`
	`c.setOutlineThickness(10);`
	`c.setOutlineColor(sf::Color(0, 0, 0));`
	
	`window.draw(c);`
		• Creates object c of type CircleShape
		• **setRadius**: sets size of circle
		• **setPosition**: sets x, y position of circle respectively
		  ○ **NOTE**: position is **NOT** the center of the circle, but rather the top left corner of the imaginary box that encloses circle
		• **setFillColor**: sets color of circle
		• **setOutlineThickness**: changes thickness of outline
		• **setOutlineColor**: changes color of outline
		• **draw(c)**: put inside window function so window will draw a circle

Drawing Shapes	`sf::CircleShape triangle(80, 3);`
	`sf::CircleShape square(80, 4);`
	`sf::CircleShape octagon(80, 8);`
	`sf::RectangleShape line(sf::Vector2f(150, 5));`
		• Similar to drawing CircleShape
		• Different possible shapes:
		  ○ CircleShape → can also draw triangles, squares, and any regular polygon w/them
		  ○ ConvexShape
		  ○ RectangleShape
		• Try playing around with the different shapes you made to draw a picture!

Textures	`sf::Texture tex;` `tex.loadFromFile("Textures/player_ship.png");`	• Part of what makes up a sprite

Sprites	`sf::Sprite sp;`
	`sp.setTexture(tex);`
	`window.draw(sp);`
	
	`sp.setScale(1.5, 1.5);`
	`sp.setPosition(100, 100);`
	`sp.setOrigin(32, 32);`
		• **Creating sprite**: initialize, set texture, & draw
		• **Scaling Sprite**: changes size, in eg. by 1.5x of original size
		• **setPosition**: change location of sprite from top-left corner
		• **setOrigin**: changes the center of the sprite

Movement Vectors	`sf::Vector2f movement(0, 0);`
	`movement.y -= 0.1f;`
	`movement.x += 0.1f;`
	
	`sp.move(movement);`
		• Vector: keeps track of movement of sprite
		• **movement.y**: the y component of movement vector
		• **movement.x**: the x component of movement vector