Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

b-it Bonn-Aachen International Center for Information Technology

Intermediate C++

MAS Foundations Course SS-19

September 19, 2019

Original Author: Sushant Vijay Chavan & Modifications By: Ethan Oswald Massey

Topics for today

We cover the following topics of C++ today:

- Arrays in C++
- Dynamic memory allocation.
- Writing basic classes and creating objects.
- Using object oriented programming concepts like Inheritance, Polymorphism.
- Optional: Using STL (Standard Template Library).
- Optional: Additional Development Tools

Arrays

- Arrays help in maintaining lists of elements of the same datatype.
- Helps us in writing compact and organized code.
- A syntax for creating an array involves using the `[]` after the name of the variable. For example:
  ```
  int costsOfProducts[5];
  ```
- Here, the number within `[]` represents the size (number of elements) of the array.
- After creating an array, any element of the array can be accessed using the `[]` operator as shown below.
  ```
  cout << costsOfProducts[2] << endl; // print cost of the 3rd product.
  ```
- **The index of elements always starts from 0**.

Arrays - Example/Exercise

- cd to the folder *arrays*.
- The program *arrays.cpp* shows a simple example of creating arrays of a few basic datatypes.
- The program *arraysWithFuctions.cpp* shows an example of how to use arrays with functions.
- **A function accepts an array as a pointer.**
- *Exercise:* Extend this program to initialize a character array with alphabets starting at `J` of **size 10**. Write a new function to print a char array and use it to print the array you created.

Creating and returning arrays from a function

- In the directory *arrays*, compile and run the program *returnArraysFromFunctions.cpp*.
- What is the output?

Creating and returning arrays from a function

Need for Dynamic Memory Allocation

- The memory (RAM) is divided into different segments and two important segments for writing a program are the **stack** and the **heap**.
- The segmentation fault in the previous program occurs because the array inside the function was created on the **stack** and as soon as the function completed, the memory allocated to the array was cleared.
- All local variables are created on the stack and are destroyed when they go out of scope.
- Heap is a large segment of memory where we can allocate any desired chunk of it for our use using the **new** operator.
- The memory allocated on heap is not deleted after the function goes out of scope. Only a **delete** operator can clear the memory.

Dynamic Memory Allocation - Example/Exercise

- cd to the folder *dma*.
- Open the program *arraysWithDMA.cpp*.
- Compile and run it.
- *Exercise:* Extend this program by adding a function to generate your name using DMA and another to print the generated name.
  - The name generation function will not require any inputs.
  - Initialize each element of the array with a character from your name.
  - The function to print the array requires two inputs: a pointer to your array and the size of the array.

Dynamic Memory Allocation

The Dangers of DMA

- As Uncle Ben tells SpiderMan, *With great power comes great responsibility*.
- Since you created the memory at your own discretion, it is your responsibility to clean it when you are done using it.
- Failing to clean up after yourself will result in memory leaks.
- Memory bugs can be quite insidious, so be careful.
- Compile, run, and then look at the code in *memoryWoes.cpp*
- There are a lot of problems with this code. See if you can find a few.

Object Oriented Programming

Basics

- C++ was developed with the primary aim of adding features to support object oriented programming (OOP) concepts to the C language.
- A **class** in OOP represents a template for creating an object. It defines all the properties and behaviors of every object that can be created using it.
  For example, a *student* class which defines the properties such as name, student ID etc. and behaviors such as study, play, etc.
- An **object** is an instance of a class.
  For example, we can create multiple instances of the above *student* class corresponding to different students belonging to a university.

Class in C++

Format

```
// The class keywork is used to declare the start of a class definition
class Student {
  public:
    Student(string name, int ID) // This is a constructor. It is used to initialize the properties of the class
    {
        name_ = name;
        studentID_ = ID;
    }

    ~Student() // This is a destructor. It is used to perform any actions during the destruction of the object.
    {
    }

    void play() // This is a method (behaviors)
    {
        cout << "Student " << name_ << " is playing." << endl;
    }

  private:
    // These are the member variables (properties) of this class
    string name_;
    int studentID_;
}; // Class definitions end with a semicolon.
```

Class - Example/Exercise

- An example is provided in the folder *studentInfo*
- *Exercise:* Write a program with a Bird class.
- The class should have the following properties: name, color, abilityToFly (a boolean).
- The class should have two methods (or behaviors): fly() and sing().
- When the method fly is called, we first check if the bird can fly and then print if that the bird is flying. Otherwise print the bird cannot fly. For example:
  `The Eagle is flying` or `A Penguin cannot fly`.
- When the method sing is called, simply print out the the brd is singing. For example: `The Nightingale is singing`.

Inheritance in C++

- Inheritance defines an **is-a** relation between classes.
- A class can inherit properties from another class during its class declaration. For example,
  `class Student : public Person`
- The access specifier determines the mode of inheritance.
- This is followed by the name of the class from which we need inherit methods and variables.
- Open the example *animals.cpp* in the folder *inheritance*.

Polymorphism in C++

- There are two type of polymorphisms:
  - **Static** - *Function overloading* (function with same name, but different parameters in the same/derived class). For example:
    ```
    int sum(int a, int b) { return a + b; }
    int sum(float a, float b) { return a + b; }
    ```
  - **Dynamic** - *Function overriding* (function with same name and same parameters in the derived class). For example, assume class Bird derives from Animal:
    In class Animal `void move() { cout << "Animal runs" << endl; }`
    In class Bird, `void move() { cout << "Bird flies" << endl; }`
    Notice that the move method is doing different things in the base and derived classes.

Polymorphism in C++ - Examples/Exercise

- cd to the folder *polymorphism*.
- The program in *functionOverloading.cpp* demonstrates the static polymorphism.
- The program in *functionOverriding.cpp* demonstrates the dynamic polymorphism.
- *Exercise:* Write a simple calculator class that can add and subtract numbers. The calculator provides a *feedInput* method to feed integers or floats to the class.
- There should be two separate functions to add (one to add integers and another for floats). Similarly for subtraction too.
- Derive from this class and write an inverse calculator which adds when we call the subtract methods and vice-versa.

Polymorphism in C++

The need for virtual functions

- The inheritance from classes can result in a long chain and it will be difficult for us to keep track of all the different classes that are a part of the chain.
- We desire to have a set of basic functions that are applicable to all the derived classes. The derived classes can override the functions as per their wish.
- With this structure, it should be possible to just use the base class name as the datatype that can represent all the derived classes too.
- Virtual functions help us achieve this.
- Open the file *animals.cpp* from the folder *virtualFunctions*.
- What happens if you do not make the *die()* function virtual, but still overload it in the *Phoenix* class?

Virtual functions - Exercise

- Write a simple program to compute the area and perimeter of different types of Quad's.
- Use a base class called *Quad* which has two pure virtual methods called *computeArea()* and *computePerimeter()*.
- Derive two classes called *Square* and *Rectangle* and implement the pure virtual functions of the *Quad* class.
- Create objects of the Square and Rectangle classes and store them using the Quad class pointer.
- Test if your code compute's the correct area and perimeter.

STL

- C++ provides standard libraries that implement some basic data structure's. These are called the standard template libraries (or STL).
- Some important data-structure's often used are:
  - vector - An alternative to the C++ arrays
  - map - This stores data as a key-value pair.
  - set - Similar to vector, but only stores unique values.
- These libraries provide a set of utility functions that help in managing large data structures with ease.
- Since STL's are used widely in C++, students are encouraged to learn them and understand how to use atleast the three structures mentioned above.

Additional Content

- make - a way to organize and build C/C++ projects
- cmake - a more modern, poweful, and somewhat easier to use take on make that will make make files for you
- gdb - used for command line debugging
- valgrind - profiling, detecting and debugging memory issues
- smart pointers - wraps around C++ pointers for additional functionality and safety