# 5COSC001W - Object Oriented Programming, Week 1

Dr. Klaus Draeger

September 23, 2020

# Contents

This module:

- introduces the concepts of objects and classes;
- teaches applications of object oriented analysis and design in tackling programming problems;
- aims to extend your programming skills from year 1.

Since C++ is our language of choice, we will also:

- address topics specific to it, such as multiple inheritance.

# Contents

- Classes and objects.
- Object oriented principles and characteristics.
- Inheritance, polymorphism, abstraction, encapsulation.
- How to design classes. Introduction to UML.
- Concurrency and multithreading.
- Design patterns and principles.

# Assessment

- One coursework, one exam
- 50% of mark each
- Need to score 30% in each and 40% overall
- Coursework spec will be published in week 3, deadline is in week 10
- Exam is in January, date yet to be determined

# Books and Resources

Books:

- Programming: Principle and Practice Using C++, Bjarne Stroustrup, 2nd Edition

Further reading and resources:

- C++. The Complete Reference, Herbert Schildt
- The Object Oriented Thought Process, 4th Edition, M. Weisfeld
- Design Patterns: Elements of Reusable Object-Oriented Software, Gemma, Helm, Johnson, Vlissides
- www.hackerrank.com
  (Programming challenges to practise what you have learned)

# Recap: C++ basics

- **Variables** contain values of various **types**:
  - Basic types (`int`, `bool`, ...)
  - Arrays: `int numbers[10][10];`
  - Pointers: `int *numPointer;`
  - Also references, enums, structs, ...
- Basic operations:
  - Declaration introduces a new variable
  - Assignment sets a variable's value: `myNumber = 5;`
  - Other operations like comparisons (`myNumber <= 7`)

# Recap: C++ basics

- Program structure:
  - branching: `if`, `switch`
  - loops: `while`, `for`
  - functions: `bool compare(int x, int y);` declares a function `compare` with two arguments of type `int` and return type `bool`
- Functional decomposition:
  - Break the code down into reasonably small functions
  - Candidates are
    - Pieces of code which are used in multiple places
    - Pieces of code which have a clearly defined purpose
- Object orientation allows us to further structure code and data

# So What is "Object Oriented" About?

- Programs typically create models of part of the “real” world
- The parts of the model are objects, which can be
  - something tangible (a person, a vehicle, . . . )
  - something more abstract (a point in time, a quest, . . . )
  - auxiliary objects (e.g. a **factory** for creating other objects)

  Often a good starting point: **nouns** in the specification
- Objects are categorised into types called **classes**

```
class Quest{ /* details of the class go here */ };
```

- Objects belonging to a class (**instances** of the class) share the same **characteristics** and **behaviors.**

```
Quest q1, q2; // Creates two instances of the Quest class
```

# Properties of Classes

For each class we need to decide:

- its **attributes**, given by **variables** of various types
  - Example: Class `Robot` may have attributes
    - `xPosition, yPosition` of type `int`
    - `tool` of type `Tool`, which is another class
  - The current values of these attributes in an instance define its **state**, including relations with other objects.
- its **operations** (functions or **methods**), defining instances' possible behaviours and ways of changing state
  - Example: Class `Robot` may have a method
    `void moveTo(int newX, int newY)`
    to change position
- its **constructors**, for initialising new instances
- its **destructor**, for disposing of an instance

These are collectively known as **members** of the class.

# Choosing classes: example

Consider this snippet from a fictional game description:

## DigBots game

The player controls a number of bots which move around on a grid. Each bot has a tool with a power level (. . . )

- The **nouns** (or noun phrases) in this excerpt are: **player, number, bot(s), grid, tool, power level**.
- Not all of these represent in-game objects or entities:
  - The **player** is outside the game (but some games will have related classes like **player characters** or **profiles**)
  - The **number** in "a number of bots" is not itself an entity.
  - The **power level** is going to be just a number, an **attribute** of the Tool class.
- This leaves us with classes **Bot, Grid, Tool**.

# Example: Tool class

Header files contain **declarations**; in **tool.h**:

```
class Tool{
   private:
      int powerLevel = 1; // Attribute and initial value
   public:
      Tool();      // Default constructor
      Tool(int p); // Constructor with given power level
      void setPowerLevel(int p); // Setter method
      int getPowerLevel();       // Getter method
};
```

- The attribute is defined as **private** and cannot be directly accessed from outside the class
- The constructors and methods are **public**, i.e. accessible from the outside
- Constructors are declared similarly to methods, but
  - they have no return type
  - their name must equal the class name

# Example: Tool class

Source files contain **definitions**; in **tool.cpp**:

```
#include "tool.h"

Tool::Tool(){}

Tool::Tool(int p):powerLevel(p){}

void Tool::setPowerLevel(int p){
   powerLevel = p;
}

int Tool::getPowerLevel(){
   return powerLevel;
}
```

- Note the `Tool::` prefixes: We are outside the class declaration and need to specify which scope these members belong to.
- The second constructor uses an **initializer list**, about which more in a bit

# Overloading and Defaults

- The `Tool` class has several **overloaded** constructors with different parameters.
  - The constructor used will be chosen based on the parameters provided

```
class Tool{
      /* ... */
      Tool();      // Default constructor
      Tool(int p); // Constructor with given power level
};

Tool tool1;        // No parameters provided, uses the default constructor
Tool tool2(15);    // in parameter provided, uses the other constructor
```

  - The same can be done with methods.
- In this case, the same effect can be achieved using **default parameters**:

```
class Tool
{
      /* ... */
      Tool(int p = 1);  // Uses parameter p if provided, default 1 otherwise
};
```

# Example: Bot class

## In **bot.h**:

```
#include "tool.h" // Need definition of Tool

class Bot{
   private:       // Attributes defining the state
      int xPosition, yPosition;
      Tool *tool; // Pointer to an instance of the Tool class
   public:
      Bot(int x, int y);
      ~Bot();     // Destructor
      void move(int dx, int dy);
};
```

## In **bot.cpp**:

```
#include "bot.h"

Bot::Bot(int x, int y):xPosition(x),yPosition(y){
   tool = new Tool();
}

Bot::~Bot(){ // Delete our tool to prevent memory leak
   delete tool;
}

void Bot::move(int dx, int dy){
   xPosition += dx;
   yPosition += dy;
}
```

# Object construction

- The attributes are initialized in the order in which they are declared, using these steps for each:
  1. If it occurs in the initializer list, this initialization is used
  2. Otherwise, if its declaration comes with an in-class initializer (as in `int powerLevel = 1;`), this value is used
     Example, in **`tool.h`**:

```
class Tool{
   private:
      int powerLevel = 1; // value 1 is used unless overridden by initializer list
/ ... /
};
```

  3. Otherwise, it gets default-initialized if possible
     For example, we could modify the **`Bot`** class like this:

```
class Bot{
      /* ... */
      Tool tool; // Will be initialized using default constructor Tool::Tool()
   public:
      Bot(int x, int y);
};
```

  4. The constructor body may then do additional operations.

# Reminder: Object Lifetimes

- A variable only exists in the scope in which it is declared

```
void shortProject() {
   Robot scrappy;
   /* do stuff with scrappy */
}  /* scrappy gets deleted at the end of the project*/
```

- But:

```
void badProject() {
   Robot* scrapPtr = new Robot();
   /* do stuff with scrapPtr */
}  /* scrapPtr gets deleted, object still on the heap! */
```

- This is OK if the object is still needed – but make sure to
  - keep it accessible (e.g. by returning or storing the pointer)
  - `delete` it when no longer needed
  - Otherwise, you get memory leaks.
- As of C++11, there are **smart pointers** which help with this. We will see them in a later lecture.

# Objects as Parameters

- Objects can be parameters to a method
- But note that the method receives a **copy**:

```
class Bot{
   /* ... */
   void disarm(Bot victim) {
      victim.tool = nullptr;
   }
};
Bot marvin;
Bot mephisto;
mephisto.disarm(marvin); // disarms a copy, marvin is still armed
```

- If this is not wanted:
  - either make the parameter a pointer:

    ```
    void Robot::disarm(Robot *victim){ victim->tool = null; }
    ```

  - or pass the object is **by reference**:

    ```
    void Robot::disarm(Robot &victim){ victim.tool = null; }
    ```

# Encapsulation

- Each member in a class has an access modifier
- **public**: accessible outside the class
- **private**: accessible only within the class itself
- Defined by blocks (preceded by "`private:`" and "`public:`" keywords)
  Default (if not preceded by either keyword) is `private`
- Often attributes will be private, and methods and constructors will be public
  - indirect/restricted access to attributes through methods
  - this facilitates **encapsulation (data hiding)** between classes

# Encapsulation

- Idea: wrap the data (attributes) and code acting on the data (methods) together as a single unit.
- Attributes of a class will be hidden from other classes, to be accessed only through methods.
- This allows us to:
  - limit how they can be changed
  - keep track of changes
  - modify class internals without breaking other code.
- In the Robot example:
  - limit movement to possible speeds, empty terrain, ...
  - maintain a list of visited locations
  - change internal representation of coordinates.

# Encapsulation: Example

```
class Robot{
   // ...
   // list of visited locations
   std::list<std::pair<int, int>> visited;

   bool move(int dx, int dy){
      // cannot move too fast
      if(dx < -1 || dx > 1 || dy < -1 || dy > 1)
         return false;
      xPosition += dx;
      yPosition += dy;
      visited.push_back(pair<int, int>(xPosition, yPosition));
      return true;
   }
};
```

- Position cannot be changed directly
- Method ensures speed limit and proper history tracking

# Summary – What you should know so far

- Objects and Classes
- Class declaration
- Objects as instances of a class
- Constructors, initializer lists
- Destructors
- Encapsulation