Welcome to COS 110!

Lecturers:
- Marde Helbig (coordinator)
- Anna Rakitianskaia (me)
- Phlippie Bosman (Afrikaans)

Evaluation:
- **Continuous assessment**:
  - Semester tests count directly towards your final mark
  - Final exam: only 20% of the final mark
  - Supplementary: replaces the last exam opportunity only (20%)
Welcome to **COS 110**!

**Course Website:**

- [http://cs.up.ac.za/courses/COS110](http://cs.up.ac.za/courses/COS110)
- Study Guide is there

**Practicals & Tutorials:**

- We start next week!
- Check website for the latest tutorial date/time/venue changes
- Book for a practical session
- Get username/password from COSI team (CS reception)
Welcome to COS 110!

Class Representative

- Class representative will handle communication between students and lecturers

- We need a volunteer!
Chapter 13:
Introduction to Classes
**PROCEDURAL AND OBJECT-ORIENTED PROGRAMMING**

- **Procedural programming:**
  - Basic building block: *function*

- **Object-Oriented programming:**
  - Basic building block: *class*
    - A complex data type that binds data & related functions in the same structure
    - An object is an instance of a class
    - Modeling the world with classes is much more intuitive than modeling the world with functions
Let’s model a Wizard!

- Procedural way: data is separated from the functions

```c
struct Wizard;

- Properties of a wizard:
  - int mana
  - int health

void castSpell(Wizard &w);

- Cast a spell
- How do we determine who is affected by the spell?
- Return value, extra inputs?..
Let’s model a Wizard!

- Object-Oriented way: data is bundled with corresponding functions!

```cpp
class Wizard {
  // Properties of a wizard:
  int mana;
  int health;

  // Functions of a wizard:
  void castSpell(Enemy&);
  void heal();
};
```
OBJECT-ORIENTED PROGRAMMING TERMINOLOGY

- Any object (chair, table, student, wizard) can be described in terms of
  - Properties
  - Behaviours, or functions

- **class**: like a **struct** (allows bundling of related variables), but variables and **functions** in the class can have different properties than in a **struct**

- **object**: an instance of a **class**, in the same way that a variable can be an instance of a **struct**
**Classes and Objects**

- A Class is like a blueprint and objects are like houses built from the blueprint.

![Blueprint that describes a house.](image)

![Instances of the house described by the blueprint.](image)
OBJECT-ORIENTED PROGRAMMING TERMINOLOGY

- **attributes:**
  - member variables of a class
  - In the wizard example:
    - `int mana`
    - `int health`

- **methods or behaviours:**
  - member functions of a class
  - In the wizard example:
    - `void castSpell(Enemy&);`
    - `void heal();`
MORE ON OBJECTS

- **data hiding**: restricting access to certain members of an object
  - Only the wizard knows how much mana he/she still has
  - The wizard does not have to share this information with the enemies!

- **public interface**: members of an object that are available outside of the object
  - The spell functions are the public interface of the wizard
  - He/she interacts with the world (eg. enemies) via casting the spells
  - If you want to talk to NPCs, just add another function:
    - ```
    void converse(Wizard& anotherWizard);
    ```
ADVANTAGES OF A PUBLIC INTERFACE

- Create a class called `Car`.
- Define functions in class `Car` to break the car and to accelerate the car.
- Now, a user of class `Car` can simply press the “pedals” – i.e., call `accelerate()` and `break()` functions.
- He does not need to know how exactly the mechanics of the car work!
- It is enough to know that the car will break when you `break()`, and accelerate when you `accelerate()`.
- *What if the engine is upgraded?*
INTRODUCTION TO CLASSES

- Objects are created from a **class**
- Format:
  ```
  class ClassName
  {
    declaration;
    declaration;
  };
  ```
**CLASS EXAMPLE**

```cpp
class Wizard {
    private:
        int mana;
        int health;
    public:
        void castSpell(Enemy&);
        void converse(Wizard&);
        void heal();
};
```
ACCESS SPECIFIERS

- Used to control access to members of the class

- **public**: can be accessed by functions outside of the class

- **private**: can only be called by or accessed by functions that are members of the class
**Class Example**

```cpp
class Rectangle {
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth();
        double getLength();
};
```
MORE ON ACCESS SPECIFIERS

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is `private`

```cpp
class Rectangle {
    // private by default!
    double width;
    double length;

    public:
    void setWidth(double);
    void setLength(double);
    double getWidth();
    double getLength();
};
```

```cpp
class Rectangle {
    private:
    double width;

    public:
    void setWidth(double);
    void setLength(double);
    double getWidth();
    double getLength();

    private:
    double length;
};
```
USING \texttt{const} WITH MEMBER FUNCTIONS

- \textbf{Mutator}: a member function that changes a private data member value in some way
- \textbf{_accessor}: function that retrieves a value from a private member variable without modifying it.
  \texttt{const} appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

\texttt{public}:
  \begin{verbatim}
  void setWidth(double);
  void setLength(double);
  double getWidth() const;
  double getHeight() const;
  \end{verbatim}
DEFINING A MEMBER FUNCTION

- Every class function belongs to the class
- It can only be called on an object of that class

- You can define member functions inside the class
- Functions defined in the class are called **inline** functions:

```cpp
// Example of defining a class with member functions

class Rectangle
{
  private:
    double width;
  public:
    void setWidth(double w) {
      width = w;
    }
    // However, this may not be a good idea! (Why?)
};
```
SEPARATING INTERFACE FROM IMPLEMENTATION

- **Rectangle.h**
  - Contains class definition and implementation

- **Main.cpp**
  - Includes the class and instantiates objects of the class
SEP ARATING INTERFACE FROM IMPLEMENTATION

Includes the class definition and instantiates objects of the class

Contains class definition only

Includes the class definition and implements the class
DEFINING A MEMBER FUNCTION

- Every class function belongs to the class
- It can only be called on an object of that class

When defining a member function outside the class:
- Put prototype in class declaration
- Define function using class name and scope resolution operator (::)

```cpp
int Rectangle::setWidth(double w)
{
    width = w;
}
```
SEPARATING INTERFACE FROM IMPLEMENTATION

- Member-function prototypes are placed in Rectangle.h
- Member-function definitions are placed in a source-code file Rectangle.cpp.
- Driver program (main) #includes prototypes only
- Main.cpp and Rectangle.cpp are compiled (-c) separately, and linked together by the linker (-o)
- Now our driver program (main) can see the public interface, but the implementation of the public interface is hidden
DEFINING AN INSTANCE OF A CLASS (OBJECT)

- An object is an instance of a class
- Defined like struct variables:
  ```
  Rectangle r;
  ```
- Access members using dot operator:
  ```
  r.setWidth(5.25);
  cout << r.getWidth();
  ```
- Not allowed to access `private` members using dot operator:
  ```
  r.width = 5.25; // compile error!
  ```
- Not allowed to call class functions without an object:
  ```
  getWidth(); // compile error!
  ```
POINTER TO AN OBJECT

- Can define a pointer to an object:
  ```
  Rectangle *rPtr = null;
  ```

- Can access public members via pointer:
  ```
  rPtr = &otherRectangle;
  rPtr->setLength(12.5);
  cout << rPtr->getLength() << endl;
  ```

- Recap: what is a pointer?
- What is the & operator?
Dynamically Allocating an Object

- We can also use a pointer to dynamically allocate an object.
- **Dynamically allocate**: i.e., decide how much memory to reserve at run time, not at compile time
- **How**: using the `new` keyword

```c
Rectangle *rPtr = new Rectangle;
// put in values:
rPtr->setWidth(10);
rPtr->setLength(5);
// NB! when you’re done: delete the pointer
delete rPtr;
rPtr = null;
```
CONSTRUCTORS

- A class defines a new type
- Objects are variables of type class
- How do you initialize objects??

Using a constructor!
- Constructor is a member function that is automatically called when an object is created
- Purpose is to construct (initialize) an object
- Constructor function name is class name
- Constructors have no return type
Constructors

Let's add a default constructor:

class Rectangle
{
    public:
        Rectangle();
        void setWidth(double);
        void setLength(double);
        double getWidth();
        double getLength();
    private:
        double width;
        double length;
};

Rectangle::Rectangle()
{
    width = 1;
    length = 1;
}

No return type

Same name as the class

Constructor called automatically whenever an object is created

Rectangle r;
Rectangle * rPtr = new Rectangle;
DEFAULT CONSTRUCTORS

- A default constructor is a constructor that takes no arguments.

- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.

- A simple instantiation of a class (with no arguments) calls the default constructor:
  
  ```
  Rectangle r;
  ```

- Try to always provide a default constructor for every class that you write.
PASSING ARGUMENTS TO CONSTRUCTORS

- Constructors can also take arguments:
  - Indicate parameters in prototype (Rectangle.h):
    
    ```cpp
    Rectangle(double, double);
    ```
  - Use parameters in the definition (Rectangle.cpp):
    
    ```cpp
    Rectangle::Rectangle(double w, double len)
    {
        width = w;
        length = len;
    }
    ```

- A class can have multiple constructors
PASSING ARGUMENTS TO CONSTRUCTORS

Now we can instantiate objects with initial values for the data members:

```cpp
Rectangle r(10, 5);
Rectangle * rPtr = new Rectangle(15, 4);
```
MORE ABOUT DEFAULT CONSTRUCTORS

- We can provide default values to the constructor parameters:

  \[ \text{Rectangle}(\text{double } = 0, \text{ double } = 0); \]

- If all of a constructor's parameters have default arguments, then it is a default constructor.

- Creating an object and passing no arguments will cause this constructor to execute:

  \[ \text{Rectangle } r; \quad // \quad r \ \text{instantiated with} \]
  \[ \quad // \quad \text{width } = 0, \ \text{length } = 0 \]
**Classes with No Default Constructor**

- When all of a class's constructors require arguments, then the class has **NO** default constructor.

  ```
  Rectangle(double, double); // no defaults
  ```

- When this is the case, you must pass the required arguments to the constructor when creating an object.

  ```
  Rectangle r; // compile error!
  // width = 0, length = 0
  ```
Practicals start next week
- Remember to book for a session on the course page

What 1st prac is going to be about:
- General procedural programming recap
- Finding bugs in the code
- Simple OOP (object-oriented programming)
- Make sure Chapter 13 makes sense to you

Tutorials also start next week
- Q/A
- Test at the end of every tutorial
- Tutorials and pracs are compulsory
QUICK RECAP

- What is a class?
- What is an object?
- What is a constructor?
- What is a default constructor?
DESTRUCTORS

- Opposite of a constructor:
  - Used to deallocate, or delete an object
- Member function **automatically called** when an object is **destroyed**
- Destructor name is ~classname, *e.g.*, ~Rectangle
- Has no return type; takes no arguments
- Only one destructor per class, *i.e.*, it cannot be overloaded
- **If constructor allocates dynamic memory**, destructor should **release it**
CONSTRUCTORS, DESTRUCTORS, AND DYNAMICALLY ALLOCATED OBJECTS

- When an object is dynamically allocated with the new operator, its constructor executes:

  ```cpp
  Rectangle *r = new Rectangle(10, 20);
  ```

- When the object is destroyed, its destructor executes:

  ```cpp
  delete r;
  ```
OVERLOADING CONSTRUCTORS

- A class can have more than one constructor

- Overloaded constructors in a class must have different parameter lists:
  - Rectangle();
  - Rectangle(double);
  - Rectangle(double, double);
ONLY ONE DEFAULT CONSTRUCTOR AND ONE DESTRUCTOR

- Do not provide more than one default constructor for a class

```c++
Square();
Square(int = 0);  // will not compile
```

- Since a destructor takes no arguments, there can only be one destructor for a class
MEMBER FUNCTION OVERLOADING

- Non-constructor member functions can also be overloaded:

```plaintext
void heal(int x); // increase health by x
void heal(); // just increment health
```

- Must have unique parameter lists as for constructors
USING PRIVATE MEMBER FUNCTIONS

- A private member function can only be called by another member function
- It is used for internal processing by the class, not for use outside of the class
- Modular code is easier to write and maintain: when implementing a class, create helper functions
- Helper functions are “for your eyes only” – the user does not need to know about them
Arrays of Objects

Objects can be the elements of an array:

```cpp
Wizard army[100]; // create 100 wizards!
```

Default constructor for object is used when array is defined

Must use initializer list to invoke constructor that takes arguments:

```cpp
Wizard wizardFamily[3] =
    { "Elrond", "Galadriel", "Arwen" };
```
If the constructor requires more than one argument, the initializer must take the form of a function call:

```java
Wizard wizardFamily[3] = {
    Wizard("Elrond", 25, 50),
    Wizard("Galadriel", 50, 30),
    Wizard("Arwen", 40, 25)
};
```

The constructor is called explicitly here.
ARRAYS OF OBJECTS

- It isn't necessary to call the same constructor for each object in an array:

```java
Wizard wizardFamily[3] = {
    "Elrond",
    Wizard("Galadriel", 50),
    Wizard("Arwen", 40, 25)
};
```
ACCESSING OBJECTS IN AN ARRAY

- Objects in an array are referenced using subscripts

- Member functions are referenced using dot notation:

```cpp
wizardFamily[2].castSpell(wizardFamily[0]);
wizardFamily[0].heal();
```
THE UNIFIED MODELING LANGUAGE

- **UML** stands for *Unified Modeling Language*.

- The **UML** provides a set of standard diagrams for graphically depicting object-oriented systems.

  **Why do we need something like this?**
  - Because you will be working with other programmers in the industry!
  - UML is a means of describing/explaining your program without opening the code.
A **UML** diagram for a class has **three main sections**.

- Class name goes here
- Member variables are listed here
- Member functions are listed here
EXAMPLE: A RECTANGLE CLASS

```cpp
class Rectangle {
    private:
        double width;
        double length;
    public:
        bool setWidth(double);
        bool setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```
In **UML** you indicate a **private** member with a minus (-) and a **public** member with a plus (+).

These member variables are private.

These member functions are public.
UML Data Type Notation

To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable.

- width : double
- length : double
To indicate the data type of a function’s parameter variable, place a colon followed by the name of the data type after the name of the variable.

```
+ setWidth(w : double)
```
UML Function Return Type Notation

- To indicate the data type of a function’s return value, place a colon followed by the name of the data type after the function’s parameter list.

```
+ setWidth(w : double) : void
```
THE RECTANGLE CLASS

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>- width : double</td>
</tr>
<tr>
<td>- length : double</td>
</tr>
<tr>
<td>+ setWidth(w : double) : bool</td>
</tr>
<tr>
<td>+ setLength(len : double) : bool</td>
</tr>
<tr>
<td>+ getWidth() : double</td>
</tr>
<tr>
<td>+ getLength() : double</td>
</tr>
<tr>
<td>+ getArea() : double</td>
</tr>
</tbody>
</table>
Showing Constructors and Destructors

No return type listed for constructors or destructors

Constructors

Destructor

Wizard

- name : char*
- mana : int
- health : int

+ Wizard() :
+ Wizard(n : char*) :
+ Wizard(n : char*, m : int, h : int) :
+ ~Wizard() :
+ heal() : void
+ heal(x : int) : void
+ castSpell(otherWizard : Wizard&) : void
- initName(n : char*) : void