User Types

- Enum
- User defined types
- struct
- Class

Enum

An enumeration is a type that represents a set of named integer constants. In other words, an `enum` is a way to define a custom set of named values that can be assigned to a variable of that enum type.

```
enum name { list_of_constants };
```

Example:

In [ ]:
```
#include <iostream>

enum WEEKDAY {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
};

WEEKDAY someday = SUNDAY;
```

Question: Will it print "Sunday" ?

✔ ✗

In [ ]:
```
switch (someday) {
  case TUESDAY:   std::cout << "Tuesday!" << std::endl;   break;
  case WEDNESDAY: std::cout << "Wednesday!" << std::endl; break;
  case THURSDAY:  std::cout << "Thursday!" << std::endl;  break;
  case FRIDAY:    std::cout << "Friday!" << std::endl;    break;
  case SATURDAY:  std::cout << "Saturday!" << std::endl;  break;
  case SUNDAY:    std::cout << "Sunday!" << std::endl;    break;
  default:        std::cout << "I don't like Mondays!" << std::endl; break;
}
```

Question: Will it print "1" ?

✔ ✗

In [ ]:
```
// enum WEEKDAY {  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
std::cout << WEEKDAY::MONDAY << std::endl;
```

Question: Will it print "1" ?

✔ ✗

In [ ]:
```
// enum WEEKDAY { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
std::cout << TUESDAY << std::endl;
```

Using using :-) again

You can redefine a type with your own name. This is called a type alias.

```
using MathType = float;

typedef float MathType;
```

```
MathType variance;
MathType measurements[5];
MathType fooFunction(MathType x, MathType y);
```

Equivalent to:

```
float variance;
float measurements[5];
float fooFunction(float x, float y);
```

struct

Motivation

```
   double taxDeclaration(double eenergy, double heating, double water, double services, double
   income);
   double budgetNextYear(double eenergy, double heating, double water, double services, double
   inflation);
   double totalcosts(    double eenergy, double heating, double water, double services, double
   other);
```

A struct is a collection of one or more variables (known as members) grouped together under a single name.

In [ ]:
```
struct HouseCosts {
  double eenergy;
  double heating;
  double water;
  double services;
};


HouseCosts hc = {103.22, 344.34, 324.23, 987.99};
HouseCosts hc2023;
hc2023.eenergy = 103.22;
hc2023.heating = 344.34;
hc2023.water = 324.23;
hc2023.services = 987.99;
double taxDeclaration(HouseCosts &hc, double income);
double budgetNextYear(HouseCosts &hc, double inflation);
double totalcosts(HouseCosts &hc, double other);
```

The members of a struct can be of different data types and are accessed using the dot (.) operator.

In [ ]:
```
double totalcosts(HouseCosts &hc, double other) {
  auto sum = hc.eenergy + hc.heating + hc.water + hc.services;
  return sum;
}
```

Passing by value is not recommended as it copies the entire object (not efficient), but it is possible. Example:

In [ ]:
```
double budgetNextYear(HouseCosts hc, double inflation) {
  auto sum = hc.eenergy + hc.heating + hc.water + hc.services;
  return sum*(1+inflation);
}
```

To write on our struct, you need to pass it by reference:

In [ ]:
```
void setElectricCost(HouseCosts &hc, double newValue){
  hc.eenergy = newValue;
}
```

If you don't want to get the memory copied and also not modified, use `const`.

In [ ]:
```
double totalCosts(const HouseCosts &hc, double other) {
  auto sum = hc.eenergy + hc.heating + hc.water + hc.services;
  hc.eenergy = 0;
  return sum;
}
```

How do we use these functions?

In [ ]:
```
HouseCosts hc2024 = {103.22, 344.34, 324.23, 987.99};
auto total = totalcosts(hc2024, 45.65);
auto budget2025 = budgetNextYear(hc2024, 0.04);
setElectricCost(hc2024, 405.34);
```

One step further: functions which are typically associated with this data can be included inside the `struct`.

In [ ]:
```
double budgetNextYear(HouseCosts &hc, double inflation) {
  auto sum = hc.eenergy + hc.heating + hc.water + hc.services;
  return sum * (1 + inflation);
}
```

↓

In [ ]:
```
struct HouseCosts {
  double energy;
  double heating;
  double water;
  double services;

  double budgetNextYear(double inflation){
    auto sum = energy + heating + water + services;
    return sum * (1 + inflation);
  }
};
```

And how do we use this function inside a struct? Using the dot operator (.) as well.

In [ ]:
```
HouseCosts hc2023 = {103.22, 344.34, 324.23, 987.99};
auto budget2024 = hc2023.budgetNextYear(0.04);
```

But using functions in a `struct` is not really used due to software engineering principles. Usually we want to encapsulate the data, and a Class can help us do this.

Typically, a `struct` is used without functions and is used to quickly access data:
`hc2023.energy`

Classes

A struct is almost a class... So you already know in principle how to do a class, what it is and how to use it. The difference is: the default access level of the member data.

In [ ]:
```
class HouseCost {
  private:
    double energy;
    double heating;
    double water;
    double services;
  public:
    double budgetNextYear(double inflation);
};
```

In [ ]:
```
double HouseCost::budgetNextYear(double inflation) {
    auto sum = energy + heating + water + services;
    return sum * (1 + inflation);
}
```

Some definitions that are useful to know:

Member or member variable: the variables within a class. In our example: `energy`, `heating`, `water`, `services` in our class `HouseCost`.
Member function: the functions within a class. In our example: `budgetNextYear`.
Accessibility level: this tells you, who can see your data members and your member functions. The keywords are: `private`, `protected` and `public` **Objects**: the instances of a class at runtime. For example: If `HouseCost myhousecosts;` is declared somewhere, then `myhousecosts` at runtime is an object (of the class `HouseCost`). Every object in memory has its own copy of members and functions.
Class: it does not define any data, but defines what type of variables an object encompasses, and what operations can be performed on such an object.

Class:	**Object:**
Programs code within a class scope.	Instance of a class in runtime.
Datatype	Variable

In [ ]:
```
class Y {
  public:
    int a;
    double b;
    void c(int x) {
        a = a + x;
    }
};

Y v;
v.a = 10;
v.c(40);
```

Question: in the previous example is `a` an object?
✔ ✗

Question: in the previous example is `Y` an object?
✔ ✗

Question: in the previous example is `c` a member function?

✔ ✗

Question: in the previous example is `v` an object?

✔ ✗

Visibility (private, protected, public)

Data hiding is one of the important features of OOP. It revents direct access to internal representation of the class.
A class can have multiple visibilities: Each section remains in effect until: another specifier or the closing right brace of the class body.

- Public:
  - Accessible from everywhere outside the class but within a program.
- Private:
  - Members cannot be accessed or viewed from outside the class.
  - Only the class and friend functions can access private members.
  - By default all the members and functions of a class are private.
- Protected:
  - Similar to private but members can be accessed in child classes (derived classes).

In [ ]:
```
#include <iostream>
class A {
  public:
    int x,y,z;
  public:
    int getX() { return x;}
  private:
    int getY() { return y;}
  protected:
    int getZ() { return z;}
};
```

Question: will a.x and a.getX() get printed?

✔ ✗

In [ ]:
```
A a;
std::cout << "a.x has value " << a.x << std::endl;
std::cout << "a.x has value " << a.getX() << std::endl;
```

Question: will a.getY() get printed?

✔ ✗

In [ ]:
```
std::cout << "a.y has value " << a.getY() << std::endl;
```

In [ ]:
```
std::cout << "a.z has value " << a.getZ() << std::endl;
```

Inside the class:

- "Normal" function members
- Getters and setters
- Constructors and destructors

Getters and setters

Getters and setters are methods or functions used to access and modify the private member variables (data) of a class from outside the class.

In [ ]:
```cpp
#include <iostream>

class B {
  private:
    int x_,y_;
  public:
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);
};
```

In [ ]:
```cpp
int B::getX() {
    return x_;
}
```

In [ ]:
```cpp
int B::getY() {
    return y_;
}
```

In [ ]:
```cpp
void B::setX(int x) {
    x_=x;
}
```

In [ ]:
```cpp
void B::setY(int y) {
    y_=y;
}
```

In [ ]:
```cpp
B myb;
myb.setX(3);
myb.setY(6);
std::cout << "x=" << myb.getX() << ", y=" << myb.getY() << std::endl;
```

Constructors and destructors

Constructors are useful to initialize the member variables, and destructors are useful to free resources.

Constructors

- There are different types of constructors:
  - default constructor,
  - init constructor (parametrized),
  - copy constructor,
  - move constructor, and
  - explicit constructor.
- The constructor is executed whenever new objects of that class are created.
- Has exact the same name as the class.
- Does not have any return type at all, not even void.
- It may or not receive arguments.

Default contructor: is called when an object is created without any arguments. It initializes the object with default values or performs any necessary setup. If you don't define a default constructor explicitly, the compiler will provide a default constructor for you if no other constructors are defined. It is used to allocate the space in memory to store the object with `new`.

In [ ]:
```cpp
#include <iostream>

class Point2D {
  private:
    double x_;
    double y_;
  public:
    //getters and setters
    Point2D();
};
```

In [ ]:
```
Point2D::Point2D(){
    std::cout << "Constructor called" << std::endl;
    x_=0;
    y_=0;
}
```

In [ ]:
```
Point2D mypoint;
```

Another way to initialize with the constructor:

In [ ]:
```
Point2D::Point2D(): x_(0), y_(0){
    std::cout << "Constructor called" << std::endl;
}
```

Initialization at declaration is also possible: C++11

In [ ]:
```
// restart kernel to run
#include <iostream>

class Point2D {
  private:
    double x_=0.5;
    double y_=0.6;
  public:
    //getters and setters
    void print (){
        std::cout << "x=" << x_ << " y=" << y_ << std::endl;
    }
};
```

In [ ]:
```
Point2D mypoint;
mypoint.print();
```

Init constructor: (a.k.a *parameterized constructor*) accepts one or more parameters, allowing you to initialize the object with specific values provided during object creation, in order to set the initial state of the object based on the values passed as arguments to the constructor.

In [ ]:
```
// restart kernel to run
#include <iostream>

class Point2D {
  private:
    double x_;
    double y_;
  public:
    //getters and setters
    Point2D(double x, double y);
    void print (){
        std::cout << "x=" << x_ << " y=" << y_ << std::endl;
    }
};
```

In [ ]:
```
Point2D::Point2D(double x, double y):x_(x),y_(y) {
    //often the case, there is no need to code here
}
```

In [ ]:
```
Point2D p1(2.3,2.5);
p1.print();
```

Copy Constructor: is used to create a new object as a copy of an existing object of the same class. It initializes the new object with the values of the existing object of the same type. The copy constructor is typically invoked when objects are passed by value, returned by value, or explicitly created as copies.

Other situations when the copy constructor is called:

- Passing the object as an argument to a function.
- Returning it from a function.

In [ ]:
```
// restart kernel to run
#include <iostream>

class Point2D {
  private:
    double x_;
    double y_;
  public:
    //getters and setters
    Point2D(double x, double y);
    Point2D(const Point2D & rhs);
    void print (){
        std::cout << "x=" << x_ << " y=" << y_ << std::endl;
    }
};
```

In [ ]:
```
Point2D::Point2D(const Point2D & rhs): x_(rhs.x_), y_(rhs.y_){
    std::cout << "Copy constructor called" << std::endl;
}
```

In [ ]:
```
Point2D::Point2D(double x, double y):x_(x),y_(y) {
}
```

In [ ]:
```
Point2D p1(1.0, 3.0);
Point2D p2 = p1;
```

Destructor

It is a special member function executed:

1. when an object of the class goes out of scope
2. when the `delete` expression is applied to a *pointer* to an object

- It is unique for the class.
- It has exactly the same name as the class, prefixed with a tilde (~)
- Should always be defined as `virtual`
- Neither returns a value nor can take parameters
- Should not throw exceptions.

In [ ]:
```
// restart kernel to run
#include <iostream>

class Point2D {
  private:
    double x_;
    double y_;
  public:
    //getters and setters
    Point2D()=default;
    virtual ~Point2D();
};
```

In [ ]:
```
Point2D::~Point2D(){
    std::cout << "Destructor called" << std::endl;
}
```

In [ ]:
```
{ //some scope
    Point2D p1;
    std::cout << "Statements..." << std::endl;
}
```

You can also explicitely tell the compiler you dont want a default constructor to be generated by the compiler:

In [ ]:
```
// restart kernel to run
#include <iostream>

class Point2D {
  private:
    double x_;
    double y_;
  public:
    //getters and setters
    Point2D()=delete; // no default constructor available!! only init constructor
    Point2D(double a, double b);
    virtual ~Point2D();
};
```

Summary: a class implementatiom

Let's put together the entire class

`Point2D.h` :

In [ ]:

```
// restart kernel to run
#ifndef POINT2D_H
#define POINT2D_H

class Point2D {
  private:
    double x_;
    double y_;
    // other private functions

  public:
    Point2D();
    Point2D(double x, double y);
    Point2D(const Point2D & rhs);
    virtual ~Point2D();
    double getX();
    double getY();
    void setX(const double x);
    void setY(const double y);
    //other public functions
};

#endif
```

Point2D.cpp :

In [ ]:

```cpp
#include "Point2D.h"

Point2D::Point2D(): x_(0), y_(0){
}

Point2D::Point2D(double x, double y): x_(x), y_(y){
}

Point2D::Point2D(const Point2D & rhs): x_(rhs.x_), y_(rhs.y_){
}

Point2D::~Point2D(){
}

double Point2D::getX(){
    return x_;
}

double Point2D::getY(){
    return y_;
}

void Point2D::setX(const double x){
    x_=x;
}

void Point2D::setY(const double y){
    y_=y;
}
```

Question: is `Point2D(const Point2D & rhs);` an init constructor?

✔ ✗

Question: is it possible to have this destructor: `~Point2D(int x);`

✔ ✗

Question: is `Point2D();` a default constructor?

✔ ✗

Question: is it possible to do this: `~Point2D()=default;` ?

✔ ✗

Question: is `Point2D(double x, double y);` an init constructor?

✔ ✗

Question: can we do this `Point2D(double x, double y)=default;` ?

✔ ✗

In [ ]: