MNXB01 course - C++ module

Caterina Marcon

caterina.marcon@hep.lu.se

To recap: pointers

- C++ permits to know the position (the address) of the first bit allocated for each variable;
- the address of the variable can be saved in a new type named pointer that is able to hold the address of the variable it points to;

[C++]

```
1 int y = 5; // declare variable y
2 int *yPtr; // declare pointer variable yPtr
3 yPtr = &y; // this statement assigns address of y to yPtr
```

- the address operator (&) is a unitary operator that obtains the memory address of its operand.

To recap: pointers

- The * when appears in declaration is not an operator but it indicates that the variable being declared is a pointer:

[C++]

```
1 int * p1;
```

- The * operator commonly referred to as DEREFERENCING operator, returns a "synonym" for the object to which it points to:

[C++]

```
1 int *p1; //pointer declaration
2 int a = 7;//variable declaration
3 p1 = &a;
4 cout << *p1 <<endl;//the output is 7
5 cout << p1 <<endl;//the output is a memory address (e.g.0013F580)
```

To recap: Reference

- A reference is an alternative name or alias for an object/variable.
- Almost like a pointer, but with object syntax.
- References are declared by appending type with &.
- Unlike pointers, references must ALWAYS be initialized.

[C++] Reference

```
1 double b;
2 double& bref = b; // bref is declared as a reference to b
3 double& ref; // ref is not initialized so this will NOT compile
4 int i = 5; //assign value of 5 to i
5 int& r = i; //r is a reference to i
6 int x = r; //since r is reference to i assign value of 5 to x
7 r = 2; //since r is reference to i assign value of 2 to i
```

To recap: Reference

[C++] Exercise

```
1  #include <iostream>
2
3  int main()
4  {
5  int i = 2;
6  int* pi = &i;
7  int & ri = i;
8
9  std::cout << "i = " << i << std::endl;
10 std::cout << "&i = " << &i << std::endl;
11 std::cout << "pi = " << pi << std::endl;
12 std::cout << "*pi = " << *pi << std::endl;
13 std::cout << "&pi = " << &pi << std::endl;
14 std::cout << "ri = " << ri << std::endl;
15 std::cout << "&ri = " << &ri << std::endl;
16
17 return 0;
18 }
```

To recap: Reference

[C++] Solution

```
1 i = 2
2 & i = 0 x7fff508948fc
3 pi = 0 x7fff508948fc
4 * pi = 2
5 & pi = 0 x7fff50894900
6 ri = 2
7 & ri = 0 x7fff508948fc
```

object	i (variable)	pi (pointer)	ri (reference)
address	0x7fff508948fc	0x7fff50894900	0x7fff508948fc
value	2	0x7fff508948fc	2

To recap: pass by value, reference or pointer

- There are three ways in C++ to pass arguments to a function: by value, by reference with reference arguments, by reference with pointer arguments.
- When an argument is passed by value, a copy of the argument's value is made and passed to the called function.
- Changes to copy do not affect the original variable's value in the caller.
- With pass by reference (with reference arguments and with pointer arguments) the caller gives the called function the ability to access the caller's data directly and to modify the data if the called function chooses to do so.

To recap: pass by reference

- To indicate that a function parameter is passed by reference, put an ampersand (&) after the parameter's type in the function prototype (line 1).
- In the function call, write the variable name (line 5).
- As for the function prototype, add an ampersand (&) between the parameter's type and its name (line 10).

[C++] Example pass by reference

```
1  void squareByReference(int &); //function prototype
2  int main(){
3     int z = 4;
4     cout << "z= " << z << endl; //z = 4 before squareByReference call
5     squareByReference(z); //function call
6     cout << "z= " << z << endl; //z = 16 after squareByReference call
7     return 0;
8  }
9  void squareByReference(int &numberRef) { //function implementation
10    numberRef = numberRef * numberRef; //caller argument modified
11 }
```

To recap: pass by value

[C++] Example pass by value

```
1  int squareByValue( int ); //function prototype
2  int main(){
3  int z = 4;
4  cout << "z= " << z << endl; //z = 4 before squareByValue call
5  cout << "value returned by squareByValue: " << squareByValue(z) << endl;
6  //z = 4 after squareByValue call
7  return 0;
8  }
9  //function implementation
10 int squareByValue(int number)
11 {
12 return number = number * number; //caller argument not modified
13 }
```

To recap: pass by reference with pointer

- To indicate that a function parameter is passed by pointer, put an * after the parameter's type in the function prototype (line 1).
- In the function call, write the variable name preceded by & (line 5).
- In function declaration specify the type followed by the pointer (line 10).

[C++] Example pass by reference with pointer

```
1  void squareByPointer( int *); //function prototype
2  int main(){
3  int z = 4;
4  cout << "z= " << z << endl; //z = 4 before squareByPointer call
5  squareByPointer( &z);//pass z address to the function
6  //z = 16 after squareByPointer call
7  return 0;
8  }
9  //The func. dereferences the pointer and squares the value zPoint points to
10 void squareByPointer( int *zPoint )
11 {
12 *zPoint = (*zPoint) * (*zPoint); //caller argument modified
13 }
```

In summary

- In pass by reference method the function multiplies numberRef by itself and stores the result in the variable to which numberRef refers in main function (z).
- In pass by value method the function multiplies the variable number by itself, store the result in number and returns the new value of number.
- In pass by reference with pointer method, the function deferences the pointer and squares the value to which zPoint points to modifying the z variable in the caller.

Exercise

Write a program that prints out the sum of two integer numbers. For the sum calculation you have to implement a function which requires input arguments passed by pointer. The function should return an integer by value.

Exercise

[C++] Solution

```
1  #include <iostream>
2
3  int SumNumbers(int *a, int *b); //function prototype
4
5  int main() {
6
7  int m = 5;
8  int n = 6;
9  std::cout << "Sum: " << SumNumbers(&m,&n) << std::endl;
10 return 0;
11 }
12 int SumNumbers(int *a, int *b) {
13 return (*a) + (*b);
14 }
```

Exercise

Write a program that prints out the sum of two integer numbers. For the sum calculation you have to implement a function which requires input arguments passed by reference. The function should return an integer by value.

Exercise

[C++] Solution

```
1  #include <iostream>
2
3  int SumNumbers(int & a, int & b); //function prototype
4
5  int main() {
6
7  int m = 5;
8  int n = 6;
9
10 std::cout << "Sum: " << SumNumbers(m,n) << std::endl;
11
12 return 0;
13 }
14
15 int SumNumbers(int &a, int &b) {
16 return a + b;
17 }
```

Heap and stack

- The memory available for a program to use is made up of two areas called stack and heap.
- The stack is small (Megabytes), fixed size memory for local variables.
- When a variable on the stack falls out of scope it is deallocated; we don't have to worry about memory management with the stack.
- The stack is small, so it can overflow (this typically happens due to bugs like infinite loops).

Heap and stack

- The heap is a large memory and it can grow dynamically.
- To put a variable on the heap, it is necessary to use the operator new (already used in slide 30 lecture 6). This operator returns a pointer through which the variable is accessed.
- Variable on the heap are never deallocated automatically. The memory must be freed manually using the delete operator.
- A pointer itself is an integer (it holds a memory address); it is on the stack and it is deallocated automatically.

New and delete operators

[C++] Basic pointer manipulations

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int *p1, *p2;
6  p1 = new int; //heap memory allocation
7  *p1 = 42;
8  p2 = p1;
9  cout << "*p1 == " << *p1 << endl;
10 cout << "*p2 == " << *p2 << endl;
11 *p2 = 20;
12 cout << "*p1 == " << *p1 << endl;
13 cout << "*p2 == " << *p2 << endl;
14 p1 = new int;
15 *p1 = 100; //heap memory allocation
16 cout << "*p1 == " << *p1 << endl;
17 cout << "*p2 == " << *p2 << endl;
18 delete p1; //heap memory deallocation
19 return 0;
20 }
```

C++ class Basics

A class is a user-defined type

Used to make abstractions for easier code understanding and extension

A class consists of members

Members are data members and member functions

Data members defines the state of the object

Member functions defines the behaviour of the object

In C++ class members are accessed with . (dot)

Special functions: Constructors and destructors

Classes in C++

In C++ classes are declared with keyword `class`

[C++]

```
1  class name_class
2  {
3    private: // access specifier
4      int m; // declare member m (data)
5
6    public: // access specifier
7      name_class(int i=0): m(i) // constructor, will initialize data member m
8      {
9      }
10
11     int mf(int n) // a member function
12     {
13       int old = m;
14       m = n;
15       return old;
16     }
17 }; // remember to end classes with semicolon
```

public , private and protected

public , private and protected determines the access attributes of the members following the keyword

A public member can be accessed from outside the class and anywhere within the scope of the class object

A private member can only be accessed from within the class scope

A protected member can be accessed from other members of the same class but also from members of derived classes.

Data members

Data members usually define the internal state of the class

[C++]

```
1 class Employee
2 {
3   // ... some code here ...
4   private: // data members are usually private!
5     string name_; // name of employee
6     int age_;    // the age of employee
7     double salary_; // the salary the employee gets
8   // .. some more code here ..
9 };
```

Member functions

Member functions are functions that can manipulate member data

[C++]

```
1  class Employee
2  {
3    private:
4      void IncreaseAge(); // member functions can also be private
5    public:
6      void TotalEmpl(int);
7      void GiveRaise()
8      {
9        salary_ *= 10;
10     }
11 };
12
13 // calls member function
14 void Employee::TotalEmpl(int x){
15 //...function implementation
16 }
17 employee.GiveRaise();
```

Constructors

A constructor is a function with the explicit purpose of initializing the object and its data members

They are recognized as having the same name as the class itself

[C++] Constructors

```
1  class Employee
2  {
3    public: // constructors are usually public
4      Employee(); // default constructor
5  };
6
7  // ...
8
9  Employee tk; // default construction
10
11 Employee yw("Wangy", 28, 100); // create an employee
12 Employee ig("Ian", 28, 1000000000000000); // create another employee
```

Destructors

Destructors are called when an object of class type goes out of scope

Its purpose is to clean up after the object, e.g. by de-allocating memory

[C++] Destructor

```
1 class Employee
2 {
3   public:
4     ~Employee(); // destructor is declared using name of class prepended with ~
5 };
```

[C++] Example

```
1 int main()
2 {
3   { // start scope
4     Employee employee; // declare employee as type Employee
5   }// employee goes out of scope and its destructor gets called
6   return 0;
7 }
```

Example 1: rectangle class

[C++] Rectangle class - part 1

```
1  //class rectangle
2  #include <iostream>
3  using namespace std;
4
5  //class definition
6  class rectangle {
7   private:
8   int width, height;
9   public:
10  rectangle (int, int); //constructor
11  ~rectangle() {}; //destructor
12  int area() {return (width*height);}
13 };
14
15 //constructor implementation
16 //This function takes as inputs two integers and copies these numbers in the privat
17 rectangle::rectangle (int a, int b){
18 width = a;
19 height = b;
20 }
```

Example 1: rectangle class

[C++] Rectangle class - part 2

```
1
2  int main()
3  {
4   //Fist instance of the class rectangle
5   rectangle rect1 (3,4);
6   //Second instance of the class rectangle
7   rectangle rect2 (5,6);
8   cout << "First rectangle area: " << rect1.area() << endl;
9   cout << "Second rectangle area: " << rect2.area() << endl;
10 return 0;
11 }
```

- The instances of the class are called `objects` . It is possible to create many instances of a class.

Exercise

Rewrite the rectangle class and adapt it to a triangle.

Triangle class

[C++] Triangle class - part 1

```
1  //class triangle
2  #include <iostream>
3  using namespace std;
4
5  //class definition
6  class triangle {
7   private:
8   int base, height;
9   public:
10  triangle (int, int); //constructor
11  ~triangle() {}; //destructor
12  int area() {return (base*height)/2;}
13 };
14
15 //constructor implementation
16 //This function takes as inputs two integers and copies these numbers in the privat
17 triangle::triangle (int a, int b){
18 base = a;
19 height = b;
20 }
```

Triangle class

[C++] Triangle class - part 2

```
1
2  int main()
3  {
4   //Fist instance of the class triangle
5   triangle tri1 (3,4);
6   //Second instance of the class triangle
7   triangle tri2 (5,6);
8   cout << "First triangle area: " << tri1.area() << endl;
9   cout << "Second triangle area: " << tri2.area() << endl;
10 return 0;
11 }
```

Multiple source files

- A program can be split into multiple files.
- Makes it easier to understand and change.
- Allows individual parts to be compiled separately.
- If I make a change in one Compilation Unit only that Compilation Unit. must be recompiled, and NOT the whole program: faster compilation times.

Rectangle class declaration and implementation

- Declaration (.h files): list of data and function members.
- Definition (.cpp files): implementation of functions.

[C++] rectangle.h

```
1 class rectangle {
2  private:
3  double width, height;
4  public:
5  rectangle (double a, double b); //constructor
6  ~rectangle(); //destructor
7  double area();
8 };
```

Rectangle class declaration and implementation

[C++] rectangle.cpp

```
1  #include <iostream>
2  #include "rectangle.h"
3  using namespace std;
4
5  //contructor implementation
6  rectangle::rectangle (double a, double b){
7  width = a;
8  height = b;
9  }
10
11 //destructor implementation
12 rectangle::~rectangle(){}
13
14 //member function area implementation
15 double rectangle::area(){
16 return width*height;
17 }
```

Rectangle class declaration and implementation

[C++] main.cpp

```
1  //class rectangle
2  #include <iostream>
3  #include "rectangle.h"
4  using namespace std;
5  int main()
6  {
7   //Fist instance of the class
8   rectangle rect1 (3,4);
9   //Second instance of the class
10  rectangle rect2 (5,6);
11  cout << "First rectangle area: " << rect1.area() << endl;
12  cout << "Second rectangle area: " << rect2.area() << endl;
13 return 0;
14 }
```

[C++] results

```
1 First rectangle area: 12
2 Second rectangle area: 30
```

Rectangle class declaration and implementation

- In the previous example we have two translation units that are compiled separately.
- To compile the units:

[C++] To compile

```
1 g++ main.cpp rectangle.cpp -o rectangle
2 //"rectangle" is the executable name
```

Exercise

Rewrite the rectangle class and adapt it to a triangle.

Triangle class

[C++] triangle.h

```
1 class triangle {
2  private:
3  double base, height;
4  public:
5  triangle (double a, double b); //constructor
6  ~triangle(); //destructor
7  double area();
8 };
```

Triangle class

[C++] triangle.cpp

```
1  #include <iostream>
2  #include "triangle.h"
3  using namespace std;
4
5  //contructor implementation
6  triangle::triangle (double a, double b){
7  base = a;
8  height = b;
9  }
10 //destructor implementation
11 triangle::~triangle(){}
12
13 //member function area implementation
14 double triangle::area(){
15 return (base*height)/2.;
16 }
```

Triangle class

[C++] main.cpp

```
1  //class triangle
2  #include <iostream>
3  #include "triangle.h"
4  using namespace std;
5  int main()
6  {
7   //Fist instance of the class
8   triangle tri1 (3,4);
9   //Second instance of the class
10  triangle tri2 (5,6);
11  cout << "First triangle area: " << tri1.area() << endl;
12  cout << "Second triangle area: " << tri2.area() << endl;
13 return 0;
14 }
```

[C++] results

```
1 First triangle area: 6
2 Second triangle area: 15
```

Some good programming conventions

- All member variables should in general be private to facilitate encapsulation.
- it is good practice to use a convention to name all private variables. For instance, we could prepend their name with "f". So, if we see in the code "fNumber" we know that it is a private variable.
- Getters and Setters functions are used to handle private info.

Class rectangle

[C++] rectangle.h

```
1  #include <cmath> //for fabs
2  class rectangle {
3   private:
4   double fwidth, fheight;
5   public:
6   rectangle (double a = 0., double b = 0.); //constructor
7   ~rectangle(); //destructor
8   double area();
9   double getWidth(){return fwidth;}
10  double getHeight() {return fheight;}
11  //These funcs. make sure that width and heigh are positive
12  void setWidth(double a) {fwidth = fabs(a);}
13  void setHeight(double b){fheight = fabs(b);}
14
15 };
```

Class rectangle

[C++] rectangle.cpp

```
1  #include <iostream>
2  #include "rectangle.h"
3  using namespace std;
4
5  rectangle::rectangle (double a, double b){
6  fwidth = a;
7  fheight = b;
8  }
9  rectangle::~rectangle(){}
10
11 double rectangle::area(){
12
13 return fwidth*fheight;
14
15 }
```

Class rectangle

[C++] main.cpp

```
1  //class rectangle
2  #include <iostream>
3  #include "rectangle.h"
4  using namespace std;
5
6  int main()
7  {
8   //Fist instance of the class
9  rectangle rect1 (3,4);
10 cout << "Width = " << rect1.getWidth() << " Height = " << rect1.getHeight() << endl
11 rect1.setWidth(12);
12 rect1.setHeight(11.5);
13 cout << "Width = " << rect1.getWidth() << " Height = " << rect1.getHeight() << endl
14  //Second instance of the class
15  rectangle rect2 (5,6);
16  cout << "First rectangle area: " << rect1.area() << endl;
17  cout << "Second rectangle area: " << rect2.area() << endl;
18 return 0;
19 }
```

Class rectangle

[C++] output

```
1 Width = 3 Height = 4
2 Width = 12 Height = 11.5
3 First rectangle area: 138
4 Second rectangle area: 30
```

Inheritance

- We'd to repeat a lot of code to write the `triangle` and `rectangle` .
- Since `triangle` and `rectangle` have many features in common, we can share characteristics among similar types using inheritance.
- Both `triangle` and `rectangle` can be thought as "special" cases of something more general (e.g. class shape).

Inheritance

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

Inheritance

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

Inheritance

- A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes.
- In order to derive a class, you have to write:

[C++]

```
1 class derived-class: access-specifier base-class
```

- Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Inheritance

[C++] Class shape (shape.h)

```
1  #include <cmath>
2  //This class has the common characteristics of triangles and rectangles
3  class shape {
4  public:
5   shape (double a = 0., double b = 0.); //constructor
6   ~shape(); //destructor
7   double getWidth(){return fwidth;}
8   double getHeight() {return fheight;}
9   //These funcs. make sure that fwidth and fheigh are positive
10  void setWidth(double a) {fwidth = fabs(a);}
11  void setHeight(double b){fheight = fabs(b);}
12
13 //Protected members can be accessed by this and whatever inherits from this
14 protected:
15 double fwidth, fheight;
16 };
```

Inheritance

[C++] Class shape (shape.cpp)

```
1 #include "shape.h"
2
3 //constructor implementation
4 shape::shape(double a, double b){
5 setWidth(a);
6 setWidth(b);
7 }
8
9 shape::~shape() {}
```

Inheritance

- Now we'll make triangle inherit from shape. The only new code we have to write is whatever is specific to triangle (in this case the area function).
- The class shape is the parent class, while triangle is the derived class.

[C++] Derived class triangle (triangle.h)

```
1  //Include the class that we want to inherit from
2  #include "shape.h"
3  class triangle : public shape {
4  public:
5  //A ctor and dtor must still be provided
6  triangle(double a = 0., double b = 0.);
7  ~triangle();
8  //This function is specific to triangle
9  double area() {( fwidth*fheight)/2. ;}
10 };
```

Inheritance

[C++] Derived class triangle (triangle.cpp)

```
1 #include "triangle.h"
2 //constructor implementation
3 triangle::triangle(double a, double b) : shape(a,b){
4 //:shape(a,b) Parent object initialization
5 }
6 //Destructor implementation
7 triangle::~triangle(){
8 //At the end of this destructor, the parent destructor is automatically called
9 }
```

Inheritance: exercise

Write a program that defines a base class named Shape and a derived class named Rectangle or Triangle. The derived class must provide a method to calculate the area according to the shape chosen. The methods for setting width and height must check the input from the user and make sure that it is positive.

[C++]

```
1 //To compile for example:
2 g++ main.cpp triangle.cpp shape.cpp -o triangle
```

Inheritance: solution

[C++] Class shape (shape.h)

```
1  // Base class shape
2  #include <cmath>
3  class Shape {
4     public:
5        Shape (double a = 0., double b =0.);
6        ~Shape();
7
8     void setWidth(double a) {width = fabs(a);}
9     void setHeight(double b) {height = fabs(b);}
10
11    protected:
12       double width;
13       double height;
14 };
```

Inheritance: solution

[C++] Class shape (shape.cpp)

```
1  #include <iostream>
2  #include "shape.h"
3  using namespace std;
4
5  Shape::Shape (double a, double b){
6  width=a;
7  height=b;
8
9  }
10
11 Shape::~Shape (){}
```

Inheritance: solution

[C++] Class shape (Triagle.h)

```
1 #include "shape.h"
2 class triangle : public Shape {
3 public:
4 //A ctor and dtor must still be provided
5 triangle(double a = 0., double b = 0.);
6 ~triangle();
7 //This function is specific to triangle
8 double area() { return ( width*height)/2. ;}
9 };
```

Inheritance: solution

[C++] Class shape (triagle.cpp)

```
1  #include "Triangle.h"
2
3  //constructor implementation
4  triangle::triangle(double a, double b) : Shape(a,b){
5  //:shape(a,b) Parent object initialization
6  }
7  //Destructor implementation
8  triangle::~triangle(){
9  //At the end of this destructor, the parent destructor is automatically called
10 }
```

Inheritance: solution

[C++] Class shape (main.cpp)

```
1  #include <iostream>
2  #include "Triangle.h"
3  using namespace std;
4
5  int main() {
6     triangle tri1;
7
8     tri1.setWidth(5.8);
9     tri1.setHeight(1);
10
11    // Print the area of the object.
12    cout << "Total area: " << tri1.area() << endl;
13
14    return 0;
15 }
```

Pointers and classes

- When I create an instance of a class, I allocate a memory space to save the class data members.
- I can then access this memory space in two ways:
  - Instantiating the class as in the previous examples:

[C++]

```
1       triangle tria1 (5.5,4);
2       cout << "Triangle 1 area:" << tria1.area() << endl;
3       //members are accessed using a '.'
```

  - Using `new` operator (pointer):

[C++]

```
1       triangle* tria2 = new triangle(5,4);
2       cout << "Triangle 2 area:" << tria2->area() << endl;
3       //members are accessed using a '->'
```

Pointers and classes: exercise

Consider the triangle class you wrote before and use `new` in order to create an instance of the class.

Pointers and classes: solution

[C++] Triangle.h

```
1 class triangle {
2 private:
3 double width, height;
4 public:
5 triangle (double a, double b);
6 ~triangle();
7 double area();
8 };
```

Pointers and classes: solution

[C++] Triangle.cpp

```
1  #include <iostream>
2  #include "Triangle.h"
3  using namespace std;
4
5  triangle::triangle (double a, double b){
6  width=a;
7  height=b;
8  }
9
10 triangle::~triangle (){}
11
12 double triangle::area(){
13 return (width*height)/2;
14 }
```

Pointers and classes: solution

[C++] main.cpp

```
1  #include <iostream>
2  #include <cmath>
3  #include "Triangle.h"
4  using namespace std;
5
6  int main(){
7
8  triangle tria1 (5.5,4);
9  //IMPORTANT
10 triangle* tria2 = new triangle(5,4);
11
12 cout << "Triangle 1 area:" << tria1.area() << endl;
13 //IMPORTANT
14 cout << "Triangle 2 area:" << tria2->area() << endl;
15
16 //IMPORTANT
17 delete (tria2);
18 return 0;
19 }
```

Coding style

- Use meaningful (intention-revealing) names;
- Keep functions small (no more than 20 lines);
- A single function should do one thing;
- Use comments in order to explain why you do something (and not how).