MEDIS: A Methodology for the Formation of Highly Qualified Engineers at Masters Level in the Design and Development of Advanced Industrial Informatics Systems

WP2.1 Chapter 4: Modular Programming

Co-funded by the
Tempus Programme
of the European Union

544490-TEMPUS-1-2013-1-ES-TEMPUS-JPCR

Authors: Busquets J.V., Albaladejo, J.

MEDIS: A Methodology for the Formation of Highly Qualified Engineers at Masters Level in the Design and Development of Advanced Industrial Informatics Systems

WP2.1 Chapter 4: Modular Programming

Contract Number: 544490-TEMPUS-1-2013-1-ES-TEMPUS-JPCR

Starting date: 01/12/2013 **Ending date:** 30/11/2016

Deliverable Number: 2.1

Title of the Deliverable: AIISM teaching resources - Industrial Computers

Task/WP related to the Deliverable: Development of the AIISM teaching resources - Industrial Computers

Type (Internal or Restricted or Public): Public

Author(s): Busquets J.V., Albaladejo, J.

Contractual Date of Delivery to the CEC: 30/09/2014

Actual Date of Delivery to the CEC: 30/09/2014

Project Co-ordinator

Company name :	Universitat Politecnica de Valencia (UPV)
Name of representative :	Houcine Hassan
Address :	Camino de Vera, s/n. 46022-Valencia (Spain)
Phone number :	+34 96 387 7578
Fax number :	+34 963877579
E-mail :	husein@upv.es
Project WEB site address :	https://www.medis-tempus.eu

Context

WP 2	Design of the AIISM-PBL methodology
WPLeader	Universitat Politècnica deValència (UPV)
Task 2.1	Development of the AIISM teaching resources - Industrial Computers
Task Leader	UPV
Dependencies	MDU, TUSofia, USTUTT, UP

Author(s)	Albaladejo, J., Busquets J.
Contributor(s)	Martínez, J.M., Hassan, H.

History

Version	Date	Author	Comments
0.1	01/03/2014	UPV Team	Initial draft
1.0	19/09/2014	UPV Team	Final version

Table of Contents

1 Executive summary

WP 2.1 details the learning materials of the Advanced Industrial Informatics Specialization Modules (AIISM) related to the Industrial Computers Module.

The contents of this package follows the guidelines presented in the UPV's documentation of the WP 1 (Industrial Computers Module)

- The PBL methodology was presented in WP 1.1
- The list of the module's chapters and the temporal scheduling in WP 1.2
- The required human and material resources in WP 1.3
- The evaluation in WP 1.4

During the development of this WP a separate document has been created for each of the chapters of the Industrial Computers Module (list of chapters in WP1.2).

In each of these documents, section 2 introduces the chapter; sections 3, 4, 5 and 6 details the Lecture, Laboratory, Seminar and Mini-project of the chapter; section 7 lists the bibliography and the references.

2 Introduction

This chapter introduces the modular programming techniques. The objective is to give the student the basic concepts to divide the project in separate modules, the make the entire project more robust, reliable and secure.

Object oriented programming is also explained, and some examples are listed. This programming methodology may be used or not, but the authors encourage their use for the benefits that may produce.

The practice develops a modular programming in the Qt C++ environment, to exercise the concepts obtained in the lecture.

3 Lecture

The historical evolution of imperative programming techniques ranges from sequential programming, using languages such as Basic and Fortran, through functional programming using Pascal, modular programming using C language and reaching the current techniques for object-oriented programming in languages as C++, Delphi ObjectPascal and VisualBasic.

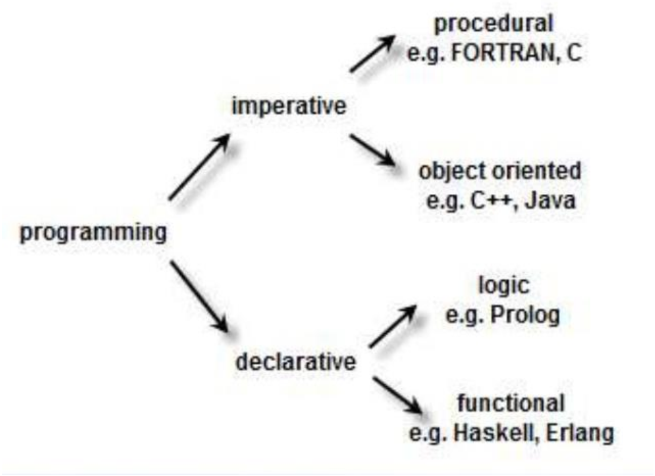


Figure 1. Programming techniques evolution

Next sections describe the more actual paradigms applied to II, as modular programming and object oriented programming.

3.1 Functions

The use of functions is the first step in the programming evolution. The functions group a set of sentences of code to perform a task. The programmer may call the function by its name to perform the task anywhere in the program file. Usually, functions are located in the file before its call. Usually functions pack meaningful pieces of code that are prone to be executed more than one time. The function declaration format is:

```
return_type function_name(parameter1, parameter2, …)

// each parameter is a type followed by an identifier

{

 // set of sentences (body)

}
```

Example a function to obtain the average of two values:

```
int average(int a, int b)

{

    return((a+b)/2);

}
```

The prototype of the function does not include the body and is followed by semicolon, it is usually placed in the header file to make the function available to other files:

```
int average(int a, int b);
```

The call to the function may be:

```
Result = average(6, 8);        // 7 is stored in variable result
```

Next example, commonly used in II, is a bit manipulation function `writeBit` to change the *value* of a given *bit* of a byte, not affecting the rest:

```
void writeBit(byte &DIO, byte bit, byte value)
{
        if (value == 1)
                DIO = DIO | (1<<bit);
        else
                DIO = DIO & ~(1<<bit);
}
```

Function `writeBit`, returns nothing (void), and has three `byte`-type parameters. The `byte` type is not C++ standard, it is previously defined as:

```
typedef unsigned char byte;       // values from 0x00 to 0xFF
```

The three parameters are: `&DIO`, `bit`, y `value`. DIO stands for Digital Input Output, and is a variable representing a register for digital control (one byte size). The second is the bit position (from 0 to 7) and the third is the new value for such a bit (either 0 or 1).

The parameters and any new variable declared in the body of the functions have a local scope. Their live finish as soon as the function finishes execution, either because a return or the last sentence of its body is reached. The contents of parameters `bit and value` are no worth once the function has done its work. However, the value resulting in DIO is populated outside the function because it was passed by reference (using `&` symbol). The variable used in the call, will kept its changes, for instance:

```
typedef unsigned char byte;// New type: unsigned char by byte

void writeBit(byte &DIO, byte bit, byte value)// Function implementation
{
        if (value == 1)
                DIO = DIO | (1<<bit); // Shift and OR mask with 1
        else
                DIO = DIO & ~(1<<bit); // Shift and AND mask with 0
}
void main(void)   // main function, called at program startup
{
byte a; // local variable declaration
a = 0x05; // data asigment DIO = 0x05 (0000 0101)
writeBit(a, 3, 1); // DIO = a, bit = 3 and value = 1
```

```
            // Now, a contains result in DIO = 0x0D (0000 1101)
        }
```

Identifiers only can be used once there are declared. This rule includes functions. Before a function is called, the compiler must know its details, either because there is the function implementation or a function prototype earlier in the file.

The prototypes of library C functions like `printf, pow`, etc are placed in header`.h` files. The #include directive is used to "read" this files. For instance:

```
        #include <math.h>

        #include <stdio.h>
```

3.2 Modular programming

Modular programming was a step forward from functional programming. As the programs are getting more complex and increase the number of functions, it become necessary to structure them in separate files (modules), organized related to their functionality.

A module in C or C++, comprises two files: header (a file with .h extension) and source (file with .c or .cpp extension). The header includes function prototypes, types and classes definitions and constants. The source includes the implementation itself of the functions.

Regarding the former example, we can group `writeBit` and `ReadBit` in a separate module, with files `mask.c` (implementation) and `mask.h` (function prototypes). The main file, for instance `main.c` will contain `main()` function and `#include <mask.h>` to be able to use functions implemented in `mask.c`.

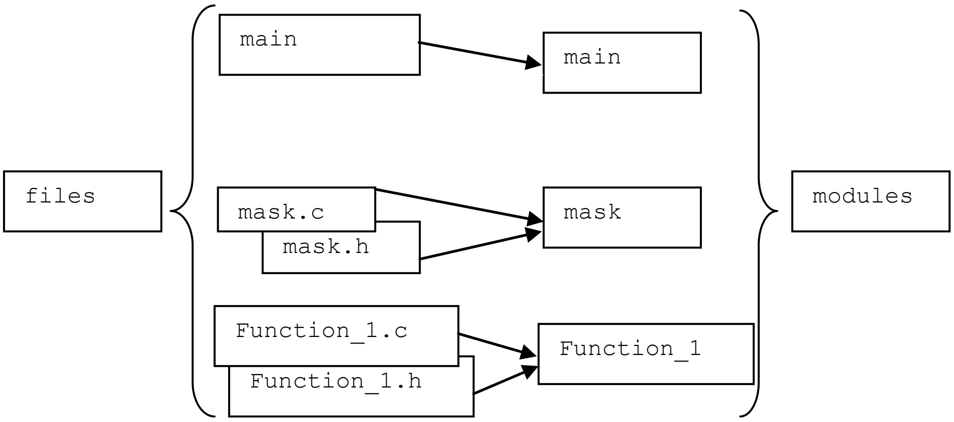


Figure 2 Source files (.c), headers (.h), and modules.

The file `mask.c`, contain the definition/implementation of many functions related to bit manipulation (in addition to `writeBit` and `readBit`). For instance, we can add `writeField` and `readField` to change several bits at once, based on a given mask.

```
#include "mask.h"    // Module Header file

void writeBit(byte &DIO, byte bit, byte value) {
    // Code here
}

byte readBit(byte &DIO, byte bit) {
    // Code here
    return(...)
}

void writeField(byte &DIO, byte mask, byte value){
    // Code here
}


byte readField(byte &DIO, byte mask) {
    // Code here
    return(...)
}
```

In the header file `mask.h`, there are the function prototypes, type definitions (`typedef`), constants, and classes if any. Any C program that includes this header, and adds to the project the `mask.c` file, will be able to use the bit manipulation functions. The definition of the files that comprises the entire project depends on the compiler or IDE being used.

```
//-------------------------------------------------------

// mask.h header file for mask.c


#ifndef MaskH

#define MaskH


typedef unsigned char byte; // 8 bit data (from 0x00 to 0xFF)

//--------------------------------------------------------

// Prototypes


void writeBit(byte &DIO, byte bit, byte value);

byte readBit(byte DIO, byte bit);

void writeField(byte &DIO, byte mask, byte value);

byte readField(byte DIO, byte mask);


#endif

//--------------------------------------------------------
```

The main program source file `main.c` (or any other name) contain the function `main()` where the program starts execution. This function is in the top of the hierarchy, thus no `main.h` is required, since no one will call to function `main()`. Sometimes, a `main.h` is written to contain some constants, type definitions, or enumerates.

Main file will use the required `#include` directives for the use of the functionality contained in other modules or libraries. For instance, our `main.c` uses `stdio.h` for `sprintf` and our `mask.h` for bit manipulation. This file may be as follows:

```
#include <stdio.h>  // System header file for sprintf()function
#include "mask.h"   // User header file for bit manipulation module

void printResult(byte DIO); // Previous prototype declaration

void main(void)// main function
{
    byte a;   // local variable
    a = 0x05;               // data assignment 0x05 (0000 0101)
    writeBit(a, 3, 1);      // Use of writeBit() function
     printResult(a);        // Use of printResult() function
}

void printResult(byte DIO) // Later function implementation
{
    char bytestring[10];
    sprintf(bytestring, "0x%X", DIO); // System function (stdio.h)
}
```

Exercise

- Build a C program with only one source file `myprogram.c`.
- Declare the prototypes and implement the following functions:
  - Function to read and return the value, either 0 or 1, of the bit at position `bit`:

    ```
    byte readBit(byte DIO, byte bit);
    ```

  - Function to write a continuous set of bits over a given data (DIO), based on the ones of the mask parameter. For example, `mask=0x1C` and `value=3`, will force the bits 2 and 3 to one, leaving the rest unchanged.

    ```
    void writeField(byte &DIO, byte mask, byte value);
    ```

- Function to read the data contained in a field of bits (mask) of a given data (DIO):

```
byte readField(byte DIO, byte mask);
```

- Now, move the functions to a module called mask.c and mask.h
- Next, if we have to add a new function to switch a field of bits (with XOR), would you add a new module?
- ¿What are the advantages of modular programming?

3.3 Object oriented programming

Structure the programs as functions contained in different files is not enough once the program become complex. It is difficult to know exactly the mission of each function and their relationships to global data.

The object oriented programming solves this problem, making easier to know the object functionality and enhance the reuse of code. As stated before, an object is a collection of member functions and variables. Most of them are encapsulated and hidden, unless there are declared as public.

Classes are a step forward of compound types, like structures, but they also can contain functions in addition to data. They are complex and powerful and triggers by itself a new programming paradigm: object oriented programming.

An object is an instantiation of a given class. We may instantiate any number of objects of a class, the same way we can instantiate many variables of a particular type. The following example instantiates objects a, b and c, and instantiates variables `i, j` and `k` of type `int`.

```
class MyClass {  // Class definition (or declaration)
   int x, y;     // variable members, so called properties

   int average();// function members, so called methods
}

int MyClass::average() { // Code of the class methods
   return( (x+y) /2);    // Access to properties x, y inside the class
}

main() {
   MyClass a, b, c;  // object instantiation of class MyClass
   int i, j, k;      // local variables

   a.x = 6;          // access to object a properties
   a.y = 8;          // access to object a properties
   i = a.average();  // access to object a method, i becomes 7
}
```

Classes introduce new syntax, like scope operator (::) to implement their methods outside the class itself. Also introduces access specifiers: `private`, `protected` and `public`. We can state with them which parts of the class are visible from the outside. This allows an important concept: the encapsulation and hiding of irrelevant information to the class user. The class user only can access to the public members of a class. Other advanced key aspects of a class are the inheritance and polymorphism.

To understand the power of a class, if we program a class to perform a PID controller, the proportional, integral and derivative gains are properties, and the calculation of the PID output is a method. Then, we can instantiate any number of objects of such a class, each one with its particular gains. Conversely, in functional programming, we would be forced to use a global array which contain the gains of each instance of the PID.

This section only presents the basics of classes, and their strong points for II. For deeper information, we suggest reading [1].

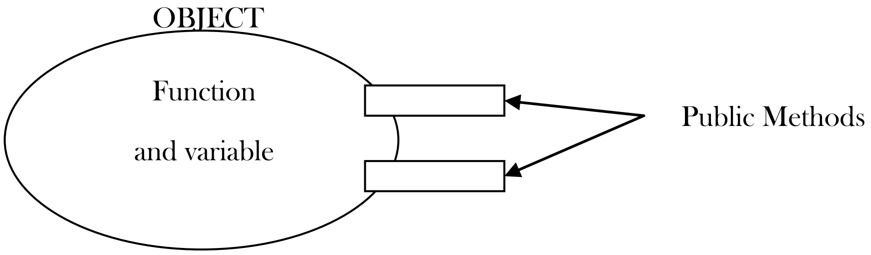


Figure 3. Graphic representation of an object

3.4 Class definition

In addition to properties and methods, a class may include two special methods: the constructor and destructor. These member functions (methods) must follow a given rules: must be named like the class itself, and they are not allowed to return values. The constructor are automatically called whenever an object of this class is created, allowing the class to initialize data and allocate memory if needed. The destructor is called when a class is destroyed, either because its living scope is finished (local object for instance), or the programs performs a `delete` . It may de-allocate memory, and perform safe actions, to return the system in a safe estate.

The definition of the class should not include any code and must be placed in a header file (`*.h`). The class definition is as follows:

```
class myClass
{
public:
    myClass(parametros); //constructor
    ~myClass();          //destructor
    int var_myProperty;
    double myMethod (int valor);

private:
    …

protected:
    …

}
```

Properties and methods accesible from class user.

Properties and methods only accessible from inside the class code

Properties and methods only accessible from the code of derived class or the class itself

The following example implements a class called `mask` to perform bit manipulation functions. It may be useful in the context of digital input/output in a SII.

```
//----------------------------mask.h----------------------------------------
     #ifndef MaskH
     #define MaskH
     //--------------------------------------------------------------
     typedef unsigned char byte;   // de 0x00 a 0xFF

     // Prototypes

     class Mask {

     private: //Private methods
        byte bitShift(byte mask);
     public: //Public methods
```

MEDIS: A Methodology for the Formation of Highly Qualified Engineers at Masters Level in the Design and Development of Advanced Industrial Informatics Systems

```
            void writeBit(byte &DIO, byte bit, byte value);
            byte readBit(byte DIO, byte bit);
            void writeField(byte &DIO, byte mask, byte value);
            byte readField(byte DIO, byte mask);
        };

        #endif
//-----------------------End mask.h---------------------------------------
```

The class definition is usually included in a header file where other prototypes, definitions and types take place.

The implementation of the class methods are usually placed in the source code file `*.cpp`. This file must include the header, `mask.h` in our example to take into account of the class definition. The body of the methods will follow this syntax, using scope operator "::"

```
myClass::myClass (parametros)
{
       ...
}
```

contructor body, if any

```
myClass::~myClass()
{
       ...
}
```

destructor body, if any

```
double myClass::myFunction(int value)
{
       ...
}
```

method body

Notice that each method is preceded by the class name, following the scope operator in between. The method body behaves like in a normal function, but it can refer to any private or public member (either method or attribute). In this case, the scope operator is not needed, since the code and the methods belong to the same class. The member variables (attributed) are accessed as they were global variables from the point of view of the class code. However, these "global" variables are independently repeated for each instance of the class (for each object).

Example:

The implementation of the mask member functions are:

```
//-------------------------------mask.cpp--------------------------------
#pragma hdrstop
#include "Mask.h"
//------------------------------------------------------------------------
#pragma package(smart_init)

/************ PRIVATE FUNCTIONS ******************/

byte Mask::bitShift(byte mask)
{
        byte count=0;
        while (readBit(mask, count)==0 && count<8) count++;
        return(count);
}
/************ END PRIVATE FUNCTIONS **************/



/************ BIT FUNCTIONS **********************/

void Mask::writeBit(byte &DIO, byte bit, byte value)
{
        if (value==1)
                DIO = DIO | (1<<bit);
        else
                DIO = DIO & ~(1<<bit);
}

byte Mask::readBit(byte DIO, byte bit)
{
        int result;


        result = DIO & (1<<bit);
        result = result >> bit;
        return(result);
}

/************ FIELD FUNCTIONS *******************/

void Mask::writeField(byte &DIO, byte mask, byte value)
{
        DIO = DIO & ~mask;
        DIO = DIO | (value << bitShift(mask));
}

byte Mask::readField(byte DIO, byte mask)
{
        byte result;
        result = DIO & mask;
        result = result >> bitShift(mask);
        return(result);
}
/************ END FIELD FUNCTIONS ***************/
```

3.5 Creating objects

The instantiation or creation of an object is merely the same as instantiate or create a variable. Like any other data type, objects may be global or local. They can be created statically at start-up of the program, or dynamically at run time whenever it is needed, by the use of pointer and the new statement. For instance, the following sentence creates two objects of the class `myClass`:

```
myClass one, two;
```

The main difference between objects and variables are that in the object instantiation, there is a call to the object constructor (if any), thus there is some code that may be executed at that precise instant. In addition, the constructor may require some parameters:

```
myClass one("object one"), two("object two");
```

In this example, the constructor requires a string. In fact, thanks to the overload capabilities of C++, several constructors may be defined, varying their parameters, and only one will be invoked during the instantiation depending on them.

The second way of object creation is dynamically at run time. This allow us to create the only the required objects during executions. It makes also easier to manage variable matrix of objects. To do that, we use the C++ sentences new and delete, that are equivalent to function `malloc` and `free` of C. We can proceed as follows:

```
myClass *onePtr;
```

Pointer declaration for later use

The memory is allocated to the object, and the address is assigned to the pointer onePtr

```
onePtr = new myClass;

onePtr = new myClass(parameters);
```

The memory used by the object is released

```
delete onePtr;
```

Pointer to the object in memory

The class constructor is called during the executions of the new statement, not at the pointer declaration. The same is for delete, where the destructor is called prior the object memory is released.

Emample:

```
myClass *onePtr, *twoPtr;

onePtr = new myClass(parameters);
```

```
twoPtr = new myClass(parameters);

…

delete onePtr;

delete twoPtr;
```

This way, the following scenario is created:

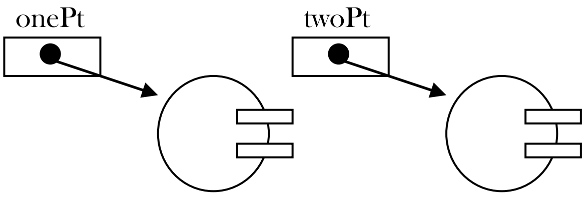


Figure 5. Two objects and their pointers.

3.6 Using objects

Once the class is instantiated in one or many objects, the program may call the object public methods. To do that, we have to write the object name, followed by scope operator dot "." or arrow "->" in case we use pointers to objects. For accessing the public attributes, the syntax is the same, but we use in addition the assignment operator "=". For instance:

```
double a;

myClass one(parameters), *twoPtr;

twoPtr = new myClass(parameters);

…

a = one.myMethod(3);// call to a method

one.myProperty = 5;        // Property assignment

…

a = twoPtr->myMethod(3);   // call to a method

two->myProperty = 5;// Property assignment
```

The following example uses the Mask class, to perform some bit manipulation. It gathers the operands from the visual environment, makes the mask operations and shows the results in a user interface (widget.ui).

```
#include "mask.h"

/************ BIT Functions ************/
byte Mask::bitShift(byte mask)
{
        byte count=0;
        while (getBit(mask, count)==0&&count<8)count++;
        return(count);
}
/**************************/

void Mask::putBit(byte &DIO, byte bit, byte value)
{
        if (value==1)
                DIO = DIO | (1<<bit);
        else
                DIO = DIO & ~(1<<bit);
}
/**************************/

byte Mask::getBit(byte DIO, byte bit)
{
        int result;

        result = DIO & (1<<bit);
        result = result >> bit;
        return(result);
}
/**************************/
/*** End of BIT Functions ***********/
/**************************/
/************ BYTE Functions **************/

void Mask:: putField(byte &DIO, byte mask, byte value)
{
        DIO = DIO & ~mask;
        DIO = DIO | (value << bitShift(mask));
}
/**************************/
byte Mask::getField(byte DIO, byte mask)
{
        int result;
        result = DIO & mask;
        result = result >> bitShift(mask);
        return(result);
}
```

4 Lab: Functions, classes and objects

4.1 Objective

The goal is to implement the code for handling:

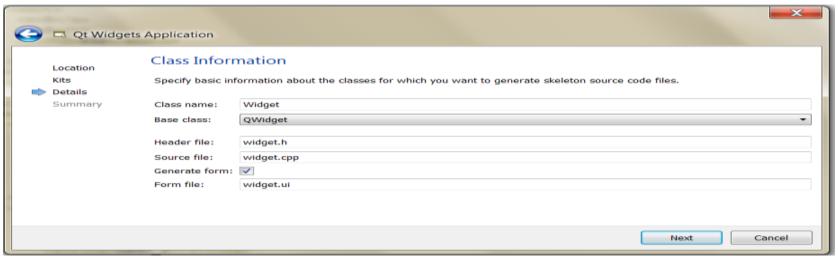
- Functions.
- Edit lines and labels
- Class example and
- Using objects

4.2 Equipment

- Personal Computer with Microsoft Windows 7 or superior operating system.
- Open source version of Qt framework for Microsoft Windows.

4.3 Departing point

Now, create a new fresh Qt application, and proceed for a Qt Widgets application and build the code to verify the absence of errors.



Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Class name: Widget
Base class: QWidget
Header file: widget.h
Source file: widget.cpp
Generate form: ☑
Form file: widget.ui

Figure 6: Qt Widgets application.

In the WidgetForm, add a button, a lineedit and two labels as follow, and arrange all objects with a gridlayout, like in the figure 7:

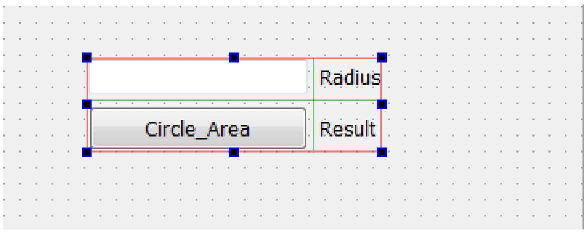


Figure 7: Widget user interface.

Change the values to the new ones showed in the table below:

Object	Property	Value
Push Button	Name	BtCircleArea
	Text	Circle_Area
Line Edit	Name	ERadius
	Text	0.0
Label	Name	LRadius
	Text	Radius
Label	Name	LResult
	Text	Result
gridLayout	Layout	gridLayout

Table 1: Objects, properties and values.

Add to the project two new files: A source (Functions.cpp); and the header file (Functions.h). Compile the code to verify the absence of errors.

Edit de header file and copy this code (in **bold**):

```
#ifndef FUNCTIONS_H

#define FUNCTIONS_H

#define PI 3.14159265358979323846264338327950

double circle_area(double radius);

#endif // FUNCTIONS_H
```

Edit the source file, functions.cpp, and add the following code:

```
#include "functions.h"


double circle_area(double v)
{
    return(PI*v*v);
}
```

Edit the widget source file (widget.cpp), and copy the **bold** text:

```
#include "widget.h"
#include "ui_widget.h"
#include "functions.h"

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
}

Widget::~Widget()
{
    delete ui;
}
```

In the widget form, select the button and make a slot when is “clicked”, and copy the following code in the body of the function:

```
void Widget::on_pushButton_clicked()

{

    double rad = 0.0,are = 0.0;


    rad = ui->lineEdit->text().toDouble();

    are = circle_area(rad);

    ui -> LResult->setText(QString::number(are));

}
```

4.4 Lab Activity

Add in the widget form all the remaining objects listed in the table below, and make a widget user interface like in the figure 8,

Object	Property	Value
OK Push Button	Name	PB_RectArea
	Text	Rectangle_Area
AB| Line Edit	Name	LE_Side_1
	Text	0.0
AB| Line Edit	Name	LE_Side_1
	Text	0.0

Label	Name	L_Result
	Text	Result
Label	Name	L_Side_1
	Text	Side_1
Label	Name	L_Side_2
	Text	Side_2
▦	Layout	gridLayout

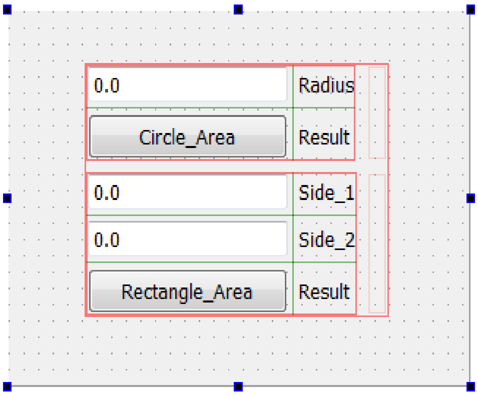


Figure 8: Objects, properties, values, and Widget user interface.

Add to the functions source code, functions.cpp, the new function to calculate the rectangle area:

```
double rectangle_area (double l1, double l2){
/* Enter your code here */
}
```

Write the appropriate code inside the function. And remember include the new function prototype in the header file: fucntions.h.

Associate to the new push button, the “slot” when is “clicked”, and provide a code similar to the following one:

```
void Widget::on_PB_RectArea_clicked()
{
    double l1,l2,are=0.0;
    char string [20];

    l1 = ui->LE_Side_1->text().toDouble() ;
    l2 = ui->LE_Side_2->text().toDouble() ;
    are = rectangle_area(l1,l2);
    sprintf(string,"%.3lf",are);
    ui -> L_Result_2 ->setText(string);
}
```

Take care, if you need some of the different format to adequate the result, because you must include the header “<stdio.h”>, in the header file “functions.h” in order to use the “sprint” function.

Run it to check its operation.

This could be the aspect of the app, like the figure 9:

5 Seminar

In addition to the definition of properties and methods, classes support a number of extended features: overloading operators, friendship and inheritance, etc. Also, there are some naming and style conventions that the programmer must take into account. For the seminar, the student must search the web for:

- Extended features for classes.
- Naming conventions in C++ for classes, properties and methods.
- Find a C++ application like Bitwise Calculator in source repositories available on the web.

6 Mini-project

The student must design and implement a QT module to manage the digital inputs of the project. The module must include functions of class methods to initialize and read the IO board.

Validate the developed functions creating the appropriate test module that exercises the functionality of the previous one. The user of this module will be able to read digital inputs by an appropriate graphical interface.

7 References

Online compiler and tutorials for many programming languages

http://www.tutorialspoint.com/compile_c_online.php

Programmer’s style guide:

http://geosoft.no/development/cppstyle.html

C++ detailed tutorials, reference, forum, and more

http://www.cplusplus.com/

Source code general repository

https://github.com/

General respository for projects and applications

https://sourceforge.net/