Visual Computing and Multimedia

10504: Mestrado em Engenharia Informática

Lab. 2 — C++ Explained

Outline

Variables, pointers, and references

Functions

Variables, pointers, and references

VARIABLES, POINTERS, AND REFERENCES

Variables, pointers, and references

Variable:

- It is a name/identifier that represents a value stored in memory.

Pointer variable:

- It is a name/identifier that represents an address (of memory) stored in memory.

Reference variable:

- It is a pointer variable.
- But, it also works as an alias to the pointed variable, so that it can be used an usual variable.
- It must be initialized at the declaration stage.

Variables, pointers, and references (cont'd)

Variable:

- It is a name/identifier that represents a value stored in memory.

Pointer variable:

- It is a name/identifier that represents an address (of memory) stored in memory.

Reference variable:

- It is a pointer variable, but it also works as an alias of the pointed variable.
- It must be initialized at the declaration stage.

Example:

```
int b;              // usual variable
int& a = b;         // reference variable
a = 10;

int b;              // usual variable
int *a = &b;        // pointer variable
*a = 10;
```

FUNCTIONS

Function

Header:

- Specifies WHAT is done by the function.

Body:

- Describes HOW the function does the specified work.

Examples:

```
return-data-type function-name (parameter list)
{
   constant declarations
   variable declarations

   other C++ statements

   return value
}
```

value-returning function

```
void function-name (parameter list)
{
   constant declarations
   variable declarations

   other C++ statements

}
```

non value-returning function

Function’s formal parameters
Function’s prototype

Formal parameters:

- The argument names in the function header.

Example:

- x and y in the following function:

```
int FindMax(int x, int y)
{
   int maximum;

   if(x>=y)
    maximum = x;
   else
    maximum = y;


   return maximum
}
```

Prototype:

- The use of function prototypes permits *error checking* of data types by the compiler.
- It also ensures conversion of all arguments passed to the function to the declared argument data type when the function is called.
- It the function header followed by “;”. The argument names are not necessary.

Example:

- `int FindMax(int, int);`

Function’s actual parameters
Calling a function

Actual parameters:

- The argument names in the function call are referred to as *actual parameters.*

Example:

- firstnum and secnum in the following function:

```
#include <iostream.h>

int FindMax(int, int);              // function prototype

int main()
{
    int firstnum, secnum, max;

    cout << "\nEnter two numbers: ";
    cin >> firstnum >> secnum;
    max=FindMax(firstnum, secnum); // the function is called here
    cout << "The maximum is " << max << endl;

    return 0;
}
```

Calling a function by value

How does it work?:

- The function receives a copy of the actual parameter values
- The function *cannot* change the values of the actual parameters.

Example**:**

- The values of firstnum and secnum are copied into x and y arguments, respectively, of the FindMax function (see previous transparency).

Calling a function by reference

How does it work?:

- Very useful when we need a function which "returns more than one value”.
- The formal parameter becomes an *alias* for the actual parameter.
- The function *can* change the values of the actual parameters.

Example:

```
#include <iostream.h>

void newval(float&, float&); // function prototype

int main()
{
    float firstnum, secnum;

    cout << "Enter two numbers: ";
    cin >> firstnum >> secnum;
    newval(firstnum, secnum);
    cout << firstnum << secnum << endl;

    return 0;
}


void newval(float& xnum, float& ynum)
{
    xnum = 89.5;
    ynum = 99.5;
}
```

Differences between pointers and references in calling functions

Two differences:

- A reference parameter is a *constant pointer* (after initializing it, it can't be changed).
- References are *dereferenced automatically* (no need to use the dereferencing op. *).

Example:

```
#include <iostream.h>

void newval(float*, float*);

int main()
{
    float firstnum, secnum;

    cout << "Enter two numbers: ";
    cin >> firstnum >> secnum;
    newval(&firstnum, &secnum);
    cout << firstnum << secnum << endl;

    return 0;
}


void newval(float* xnum, float* ynum)
{
    *xnum = 89.5;
    *ynum = 99.5;
}
```

calling function with pointer arguments

```
#include <iostream.h>

void newval(float&, float&);

int main()
{
    float firstnum, secnum;

    cout << "Enter two numbers: ";
    cin >> firstnum >> secnum;
    newval(firstnum, secnum);
    cout << firstnum << secnum << endl;

    return 0;
}


void newval(float& xnum, float& ynum)
{
    xnum = 89.5;
    ynum = 99.5;
}
```

calling function with reference arguments

Calling a function by reference
The “const” modifier

How does it work?:

- Calling by reference is the *preferred* way to pass a large structure or class instances to functions, simply because the entire structure need not be copied each time it is used!!
- C++ provides us with protection against accidentally changing the values of variables passed by reference with the *const* operator

Example (function prototype)**:**

```
int FindMax(const int&, const int&);
```

Example (function header)**:**

```
int FindMax(const int& x, const int& y)
```

Function overloading

How does it work?:

- C++ provides the capability of using the same function name for more than one function (*function overloading*)
- The compiler must be able to determine which function to use based on the number and data types of the parameters.
- *Warning*: creating overloaded functions with identical parameter lists and different return types is a syntax error!!

Example:

```
void cdabs(int x)
{
     if (x<0)
        x = -x;
     cout << "The abs value of the integer is " << x << endl;
}

void cdabs(float x)
{
     if (x<0)
        x = -x;
     cout << "The abs value of the float is " << x << endl;
}
```

STRUCTURES AND CLASSES

What is a structure?

Data type composition:

- It is an compound data type built using elements of other types.
- Declaring a structure requires declaring its *members* and their data types.

Example:

```
struct RECTANGLE
{
     float height;
     float width;
     int xpos;
     int ypos;
};
```

Declaration:

- They are declared like variables of any other type.

```
RECTANGLE R;

RECTANGLE &RRef = R;
RECTANGLE *RPtr = &R;
```

Accessing members of a structure

Dot operator (.):

- Applies to both variables and references.

Example:

```
R.height = 15.34;
RRef.height = 15.34;
```

Arrow operator (->):

- Applies to pointers.

Example:

```
RPtr->height = 15.34;
(*RPtr).height = 15.34;
```

Declaration of member functions/methods of a structure

Member functions:

- Functions which operate on the data of the structure.
- The prototype of a member function appears within the structure definition.
- Usually, the declaration of structs appears in a separate file .h

Example:

rectangle.h

```
struct RECTANGLE
{
     float height;
     float width;
     int xpos;
     int ypos;

     void draw();                  // draw member function
     void position(int,int);       // position member function
     void move(int,int);           // move member function
};
```

Implementation of member functions/methods of a structure

Member functions:

- Usually, they are implemented outside the structure.
- Usually, the implementation of member functions appears in a separate file .cpp
- The :: "scope resolution operator" is necessary for that.

Example**:**

rectangle.cpp

```
void RECTANGLE::draw()
{
    cout << "position is  " << xpos << ypos << endl;
}

void RECTANGLE::position(int x, int y)
{
    xpos = x;
    ypos = y;
}

void RECTANGLE::move(int dx, int dy)
{
    xpos += dx;
    ypos += dy;
}
```

Referring to a member function

Accessing to a member function:

- This is done in the same way as for struct variables.

Examples:

```
R.draw();
RRef.position(100,200);
RPtr->move(30,30);
```

Controlling access to members

Access specifiers:

- Most common member access specifiers are: public and private.
- The *private* keyword specifies that the structure members following it are private to the structure and can only be accessed by member functions (and by *friend* functions).

Examples:

rectangle.h

```
struct RECTANGLE
{
  private:
     float height;
     float width;
     int xpos;
     int ypos;

  public:
     void draw();                // draw member function
     void position(int,int);     // position member function
     void move(int,int);         // move member function
};
```

What is a class?

Definition:

- Practically, there are no differences between structures and classes.
  - Structures have all of their members public by default.
  - A class is a structure which has all of its members private by default.

Example:

rectangle.h

```
class RECTANGLE
{
  private:                          // only for clarity
     float height;
     float width;
     int xpos;
     int ypos;

  public:
     void draw();                   // draw member function
     void position(int,int);        // position member function
     void move(int,int);            // move member function
};
```

What is a constructor?

Definition:

- It is a member function which initializes every single class’ object.
- A constructor has:
  - the same name as the class itself,
  - no return type.

Example:

rectangle.h

```
class RECTANGLE
{
  private:
     float height;
     float width;
     int xpos;
     int ypos;

  public:
     void RECTANGLE(float,float);   // constructor
     void draw();
     void position(int,int);
     void move(int,int);
};
```

What is a constructor? (cont'd)

rectangle.cpp

```
void RECTANGLE::RECTANGLE(float h, float w)
{
     height = h;
     width = w;
     xpos = 0;
     ypos = 0;
}
```

How does a constructor work?:

- A constructor is called automatically whenever a new instance of a class is created.
- You must supply the arguments to the constructor when a new instance is created.
- If you do not specify a constructor, the compiler generates a default constructor for you (expects no parameters and has an empty body).
- *Warning*: attempting to initialize a data member of a class explicitly in the class definition is a syntax error. It is up to constructors to initialize member variables.

Example:

main.cpp

```
void main()
{
     RECTANGLE R(20.0,30.0);

     R.position(100,100);
     R.draw();
}
```

Overloading a constructor

rectangle.cpp

```
void RECTANGLE::RECTANGLE()
{

    height = 0;
    width = 0;
    xpos = 0;
    ypos = 0;
}
```

Multiple constructors:

– You can have more than one constructor in a class, as long as each has a different list of arguments.

Example:

rectangle.h

```
class RECTANGLE
{
  private:
    float height;
    float width;
    int xpos;
    int ypos;

  public:
    void RECTANGLE();              // constructor
    void RECTANGLE(float,float);   // constructor
    void draw();
    void position(int,int);
    void move(int,int);
};
```

main.cpp

```
void main()
{

    RECTANGLE R1(20.0,30.0);
    RECTANGLE R2();

    R1.draw();
    R2.draw();
}
```

Object composition in classes

Definition:

- A class may have objects of other classes as members.

Example:

rectangle.h

```
class RECTANGLE
{
  private:
     float height;
     float width;
     int xpos;
     int ypos;
     COLOR c;

  public:
     void RECTANGLE(float,float,int,int,int);
     void draw();
     void position(int,int);
     void move(int,int);
};
```

color.h

```
class COLOR
{
  private:
     int R;
     int G;
     int B;

  public:
     void COLOR(int,int,int);
};
```

Object composition in classes (cont'd)

rectangle.cpp

```
void RECTANGLE::RECTANGLE(float h,float w,int r,int g,int b):c(r,g,b)
{
    height = h;
    width = w;
    xpos = 0;
    ypos = 0;
}
```

color.cpp

```
void COLOR::COLOR(int r,int g,int b)
{
    R = r;   G;= g;   B = b;
};
```

main.cpp

```
void main()
{
    RECTANGLE R(20.0,30.0,1,0,1);

    R.draw();
}
```

What is a destructor?

Definition:

- Function that deletes an object.
- A destructor function is called automatically when the object goes out of scope:
  - the function ends;
  - the program ends;
  - a block containing temporary variables ends;
  - a *delete* operator is called.
- A constructor has:
  - the same name as the class itself, but is preceded by a tilde (~),
  - no arguments and return no values.

Example:

string.h

```
class STRING
{
  private:
     char *s;
     int size;

  public:
     STRING(char*);     // constructor
     ~STRING();         //  destructor

};
```

string.cpp

```
STRING::STRING(char *c)
{
     size = strlen(c);
     s = new char[size+1];
     strcpy(s,c);
}

STRING::~STRING ()
{
     delete []s;
}
```

What is a copy constructor?

Definition:

- It is a member function which initializes an object using another object of the same class.
- In the absence of a copy constructor, the C++ compiler builds a default copy constructor for each class which is doing a memberwise copy between objects.
- Default copy constructors work fine unless the class contains pointer data members ... Why?

Example:

string.h

```
class STRING
{
  private:
     char *s;
     int size;

  public:
     STRING(char*);
     ~STRING();
     STRING(const STRING&); // copy constructor

     void print();
     void copy(char*);
};
```

string.cpp

```
STRING::STRING(const STRING& aString)
{
    size = aString.size;
    s = new char[size+1];
    strcpy(s,aString.s);
}
```

main.cpp

```
void main()
{
 string str1("George");
 string str2 = str1;

 str1.print();    // what is printed ?
 str2.print();

 str2.copy("Mary");

 str1.print();   // what is printed now ?
 str2.print();
}
```

Summary

Variables, pointers, and references

Functions

Structures and classes