2. Extensions of the C language in the C++ language

The C++ language is a superset of the C language and it has been developed by Bjarne Stroustrup. It provides support for the procedural programming and for data abstraction, but the main purpose is the support provided for object oriented programming paradigm.

There are two types of extensions over the C language:

- adding some facilities which are not related to object oriented programming paradigm;
- adding the main elements in order to provide support for object oriented programming.

The first type of extensions refers to some elements as: reference type, in-line substitution of the functions, etc., while the second group of extensions refers to elements such as *class*, *inheritance*, *polymorphism*, etc.

In this chapter will be discussed only non object-oriented programming aspects related to the extensions of the C language.

2.1 Adding new data types

The C++ language adds two new data types: *bool* and *wchar_t*.

A) The *bool* data type represents the logical (boolean) values and it uses two built-in constants, *true* and *false*. There is a similarity with the Pascal language (which has the data type *Boolean*), and with the Java language (which has the *boolean* data type).

Unlike Pascal in the C++ language there is compatibility between the data type *bool* and arithmetic data types. The *bool* variables can be assigned with integer values, because a C++ compiler automatically converts an integer value to the *bool* value. For example, for the following sequence:

```
bool boolVar ;
int intVar ;
// ...
boolVar = intVar ;
```

the compiler generates an equivalent statement:

```
boolVar = intVar ? true : false ;
```

There is also an automatic conversion from the *bool* values to the integer values. For example:

```
intVar = boolVar ? 1 : 0 ;
```

In this way there is a compatibility with the C rules for evaluating conditional expressions appearing is some C statements such as *for*, *while*, *if*, etc.

Using the *bool* data type allows writing code with a more accurate meaning. For example the following function prototype:

```
bool Apartine(double x, double a, double b) ;
```

specifies that the function returns a logical value, while the function:

```
int Apartine(double x, double a, double b) ;
```
does not offers a such certitude.

B) **The *wchar_t (wide character)* data type** is an extension of the *char* data type, which allows the using characters represented internally on two bytes (for example the *Unicode* set of characters). For this type, `sizeof(wchar_t)` = 2, allowing to use sets of characters having more than 64000 characters.

2.2 Variable declaration and namespaces

Unlike the C language, where the local declarations must be located only at the beginning of a block, before the statements, in the C++ language the local declarations can be appear anywhere within a block. The scope of such local declared variables represents a part of the block where they have been declared, starting to the line of the respective declaration and ending the end of the current block.

Example 2.1.

```
void Processing()
{
  int k = 5 ;   //starts the scope of k variable
  // ...
  k = k + 3 ;
  // ...
  float x = 7 ; // starts the scope of x variable
  // ...
  //starts the scope of i variable
  for (int i=0 ; i<k ; i++)
    printf("%d", i) ;
  // ...
  //the scope of k, x, i variables ends
}
```

Another problem of the C language concerns the ***namespace*** of a program, making it difficult to write and to test large programs developed by several programming teams. All the variables used in different modules of a program are related to the whole program. So, the variables with the same name declared in different modules of a program, outside of any function, access the same memory zone and represent the same variables. A solution of this problem of the C language is to declare static variables in the respective files and to hide in this way these variables from outside.

The C++ language attaches the variables to a *namespace*, which allows the variables with the same name but in different modules to represent distinct variables.

All the variables declared in the standard libraries of the C++ language have a predefined namespace, denoted by *std*. For using a namespace different to the current compilation unit the directive *using* is used:

```
using namespace std;
```

For example, for using the standard functions and objects working with the input/output operations the following sequence should be used:

```
#include <iostream>
using namespace std;
```

Remark. The header files related to the standard library of the C++ language do not contain the suffix "*.h*" as in the C language. All the header files related to the standard library of the C language are rewritten in the C++ language, and their names have the character '*c*' as prefix. For example:

```
#include <alloc.h>
```

is equivalent with:

```
#include <calloc>
using namespace std;
```

However, in order to keep the compatibility with the C programs, the syntax for including the standard header files of the C language can be also used in the C++ programs.

2.3 References

One of major disadvantage of the C language is the only way of passing parameters when calling a function (unlike languages like Pascal or Ada). The C language allows only *call by value*, which requires using pointers in the case when a function modifies the value of a certain parameter.

The C++ language adds the notion of ***reference***. A reference is an alternative name (alias) for a variable. Such a data type is a derived type, which is obtained from a base type by using the operator &. If *T* is a data type, then the notation *T*& represents the reference type derived representing the set of all the reference elements at the *T* type.

The values of reference types are similar to pointers, in the sense that a reference has as value the memory address of a variable belonging to a base type. But there are some important differences between pointers and references:

a) A reference must be always initialized at the declaration. For example, the declaration:

```
int k;
int &r = k;
```

declares the *r* reference variable as having a value equal with the memory address of the *k* variable. This initialization is not an assignment, it allowing only the base variable (*k*) to have a new name (*r*). In the block where two variables have been declared (*k* and *r* in our case), both names (*k* or *r*) can be used as referring the same object.

b) References are automatically dereferred when using them in a program, so we do not need the * operator. For example:

```
int k = 5;
int &r = k, *p;
p = &k;
r = r + 1 ;      // this means k = k + 1
*p = *p + 1 ;
```

The above two observations need some remarks:

- A reference value cannot be modified after the initialization, it referring always the same object to which it has been initialized.

- The operators do not perform their actions on the reference, but on the variable referred by the reference.

The main way to use the reference is the passing of the function parameters. In this case a formal parameter having a reference type represents another name for the actual parameter corresponding to a call. Any modification of the formal parameter value means the modification of the actual parameter value.

Example 2.2. Interchanging two values by using two different methods:

```
void Swap1(int *a, int *b)
{
  int c = *a ;
  *a = *b ;
  *b = c ;
}
void Swap2(int &a, int &b)
{
  int c = a ;
  a = b ;
  b = c ;
}
void Process()
{
  int x = 7, y = 5 ;
  Swap1(&x, &y) ;
  printf("%d%d", x, y) ;
  x = 7 ; y = 5 ;
  Swap2(x, y) ;
  printf("%d%d", x, y) ;
}
```

When a function is called the formal parameters of reference type are initialized with the address of the actual parameters, this fact implies that these actual parameters have to be names of variables with a data type equal with the base type of the corresponding references.

2.4 Inline functions

In the case of small functions, with small number of statements, the calling mechanism (creation of the frame on the stack, parameter passing, etc.) can become significant in respect with the execution time of the function, so the execution time of the program can increase and its efficiency decreases.

An alternative can be the use of *macrodefinitions*, which are replaced by the compiler in the processing part. For example:

```
#define minim(a, b) ((a < b) ? a : b)
void Processing()
{
  int x = 7, y = 5, z ;
  z = minim(x, y) ;
```

```
    // ...
}
```

The C++ language provides in addition the possibility of expanding ***inline*** the functions. The in-line expanding means the generation by the compiler of the function code, without generating a calling part.

The *inline* declaration of a function is realized by using the keyword *inline* before its definition, and in the case of a member function of a class, by including the implementation of the function block in the class declaration. For example, for the minim function can be writing also:

```
inline int minim(int a, int b)
{
  return ((a < b) ? a : b) ;
}
```

In the case of the inline functions the compiler tries to place an instance of the calling function in the same code segment as the called function, but this fact is generally not guaranteed. For complex functions (recursive functions, or functions having repetitive statements) the inline mechanism is not performed.

In general the using of the inline functions is more efficient than the usual functions, but it is less efficient than the using of macrodefinitions.

Example 2.3. To illustrate this mechanism the same operation will be achieved through tree different methods: macrodefinition, inline function, and usual function. The system time will be used for determining the running time of each function (*f1*, *f2*, and *f3*).

```
#include <ctime>
using namespace std;

#define SQR(x) (x)*(x)

inline float Sqr (float x)
{
  return x*x;
}

float sqr (float x)
{
  return x*x;
}

void f1()
{
  float x = 2.5, y;
  for (long k=0; k<10000000; k++)
  {
    y = SQR(x);
  }
}
```

```
void f2()
{
  float x = 2.5, y;
  for (long k=0; k<10000000; k++)
  {
    y = Sqr(x);
  }
}

void f3()
{
  float x = 2.5, y;
  for (long k=0; k<10000000; k++)
  {
    y = sqr(x);
  }
}

int main()
{
  time_t t1, t2, t3, t4;
  time(&t1);
  f1();
  time(&t2);
  printf("t1 = %d\n", t2-t1);
  f2();
  time(&t3);
  printf("t2 = %d\n", t3-t2);
  f3();
  time(&t4);
  printf("t3 = %d\n", t4-t3);
  return 0;
}
```

The output of the above program running on a certain computer is the following:

```
t1 = 1
t2 = 14
t3 = 17
```

2.5 Default arguments for function parameters

Usually, a main rule for many programming languages imposes the same number of parameters both for the function definition and for the function call. The C language allows the definition (quite difficult) of some functions with variable number of parameters, with the help of the operator **…**. It is the task of programmers to treat the parameters, because the compiler does not perform any verification.

In addition to the C language, the C++ language provides a simpler and more efficient method for functions with a variable number of parameters: ***functions with default values for parameters***.

A parameter with a *default value* is declared as usually through a name and a data type, but in addition it is initialized with an appropriate value. If the function call contains an actual parameter with another value than the one specified in the initialization, the actual value is used as initialization; if the actual parameter is missing, the actual value is considered as the initialization value.

Example 2.4. The funcţia *Distance* can determines both the distance between two points in a plane, and the distance between a point and the origin of the axes.

```
double Distance(double x, double y,
           double x0 = 0, double y0 = 0)
{
  return sqrt((x-x0)*(x-x0)+(y-y0)*(y-y0)) ;
}

void Processing()
{
  double x1 = 3, y1 = 5, x2 = 4, y2 = 6, d1, d2 ;
  //distance between(x1,y1) and origin
  d1 = Distance(x1, y1);
  //distance between (x1, y1) and (x2, y2)
  d2 = Distance(x1, y1, x2, y2);
  // ...
}
```

Remarks:

a) A parameter with default value can be initialized only with a constant expression, which can be evaluated at the compilation phase.
b) A function can have more parameters with default values, but in this case they must take the last positions (because otherwise the current values of the parameters cannot be determined when calling the function).

A problem can occur in the case of the functions which are also declared and defined, because the heading of such a function can appear two times in the program. In this case, the C++ language imposes that the initializing values to be specified only once, at the definition or at the declaration. A solution which keeps the modular style for programming is that in which the implicit values are specified in the functions prototype from the header file, if it exists.

2.6 Function overloading

Overloading of the functions name means in fact the existence of two or more functions with the same name which perform different tasks. An example of overloading exists also in Pascal language, where some operators are used for different operations. For example, the + operator is used for adding the numbers, for the union of sets and also for string concatenation.

The C++ language allows the definition of overloaded functions.

Example 2.5. The definitions of four functions with the same name *add*:

```
int add(int a, int b) { return a + b ; }
```

```
double add(double a, double b) { return a + b ; }

char* add(char *a, char *b) { strcat(a, b) ; return a ; }

struct complex { double re ; double im ; } ;

complex add(complex a, complex b)
{
  complex c ;
  c.re = a.re + b.re ;
  c.im = a.im + b.im ;
  return c ;

}

void Processing()
{
  int k = add(5, 1) ;
  double s = add(1.5, 8.4) ;
  char *s1 = "abc", *s2 = "xyz", *s3 = add(s1, s2) ;
  // ...
}
```

The compiler determines the effective function which will be called depending of types of the actual parameters and their number.

Remarks:

a) For defining two different overloaded functions they must have different number of parameters number or at least the data type of one of the parameters.
b) Two overloaded functions can not differ only by the type of the returned value, because the type of the returned value is not verified by the compiler.

2.7 Operators for memory handling

The creation and the destruction of the dynamic objects in the C language mean the memory allocation and deallocation. Usually there are used standard functions, such as *malloc* and *free*.

The C++ language has in addition two operators represented by the keywords ***new*** and ***delete***. The used syntax is:

```
<pointer name> = new ['('] <type> [')'][(<expression>)] ;
delete <pointer name> ;
```

Example :

```
double *p = new double ;
double *p = new(double) ;
```

One can observe the superiority of the *new* operator over the *malloc* function, because it can determine both the amount of memory necessary to be allocated, and the pointer type which will be returned.

The ***new*** operator is used also for memory allocation for composed elements. In the case of arrays the length of the array must be explicitly specified. The used syntax is:

```
<pointer name> = new <type> '[' <dimension> ']' ;
delete '[' [<dimension>] ']' <pointer name> ;
```

For example:

```
struct point { double x, y ; } ;
// a structure with 2 components is allocated
struct point *p = new struct point(2, 4) ;
// an array with 10 components is allocated
double *q = new double[10] ;
// an array with 10 pointers to int is allocated
int **r = new int*[10] ;
```

Remarks:

a) The *new* operator, as *malloc* function, allocates memory in the heap part of the memory image of the program.
b) This operator calls by default a constructor the class if the data type is an instance of certain class.

The complementary for the operator *new* is the operator *delete*. The action made by this operator is similarly with the function *free*: it de-allocates the memory area where a certain pointer points to. In addition to the function *free*, it allows protection in the case of trying to free the memory for a NULL pointer.

The use of *delete* is an advantage in the case of the deallocation of an array.
For example:

```
int *p = new int[10];
delete p ;
```

In this case only the part occupied by the first element of the array is deallocated. To de-allocate all the area of the associated array, the number of components of the array must be specified. This number will be specified at the end of the key word delete between parentheses, as in the following example:

```
delete[10] p ;
```

For avoiding the errors which might occur if there are de-allocated a different number of components than the number used by its associated *new* operator a simpler syntax can be used:

```
delete[] p ;
```

In this case the number of deallocated elements is automatically determined by the compiler.

The operator *new* can be used in addition for the creation of multi-dimensional arrays. In this case all the dimensions of the array must be specified. For example, the following expression:

```
new int[2][3][4]
```

allocates the memory for two arrays of the type:

```
int [3][4]
```

and it returns a pointer to the first array, that is a pointer of the following type:

```
int (*)[3][4]
```

Example 2.6. Allocation and deallocation of an array:

```
int a[2][4] = {1, 2, 3, 4}, (*p)[4];
```

```
p = new int[2][4];
for (int i=0; i<2; i++)
{
  for (int j=0; j<2; j++)
  {
    p[i][j] = a[i][j];
  }
}
// ...
delete[] p;
```

Remark. Regardless the number of the dimensions of an array that is allocated by the operator *new*, the syntax for deallocation of this array by using the operator *delete* is the same (only one pair of brackets).

The operator *delete* (as the *new* operator) by default calls the class destructor, if the pointer indicates an instance of a certain class.

2.8 Template functions

The C++ language offers support for data abstraction and parameterization. The main notions added in this context are the ***template functions*** and ***template classes***. The using of template functions and classes, as the ***generic programming*** paradigm will be discussed in a further chapter. In this chapter only some basic elements concerning generic functions will be presented.

A ***template function*** contains at least a generic (unspecified) data type, in this way the generality of the function will be increased. The syntax for defining a template function imposes the presence of the following construction:

```
template '<' class ⟨name⟩ '>'
```

before the header of the function. In this construction, *⟨name⟩* represents the name of the data type (or the name of the class), which is a parameter for the template function (that is different to the formal parameters of the function) and it can be used inside the block of the function.

A *template function* describes a set of functions having similar code but different data types. Such a template function can be instantiated; each instance of a template function is a usual function. The syntax to instantiate a template function is similar to a function call; in addition the actual name of the used data type must be specified in angle brackets.

Example 2.7. The function *Swapp* will swap the values of two parameters whose data types are generic. In function *main* two instances of this template function will be called, where the generic data type *T* is instantiated to *int* and to *double* respectively.

```
#include <iostream>
using namespace std;

template <class T>
void Swap(T& a, T& b)
{
  T temp;
  temp = b;
```

```
  b = a;
  a = temp;
}

int main()
{
  int a=3, b=5;
  double x=33, y=55;
  Swap<int>(a,b);
  cout<<a<<" "<<b<<endl;
  Swap<double>(x,y);
  cout<<x<<" "<<y<<endl;
  return 0;
}
```

Remark. In the above example the two calls of *Swapp* can be replaced also by the following sequence:

```
Swap(a,b);
Swap(x,y);
```

because the compiler can detect automatically the data types *int* and *double* to which *T* will be instantiated.

2.9 Input/Output operators

The C++ language, as the C language, does not contain specific statements for I/O operations. In addition to C, the C++ language has a specific hierarchy of classes. The main concept in this hierarchy is that of ***stream***, which is similar to a data flow between the memory associated to a program and a peripheral device.

The specific I/O operations in the C++ language will be later described; in this chapter only some principal aspects related to the data reading and writing will be presented. In the I/O hierarchy there are two important classes used for the I/O operations: *istream and ostream*, and also the *iostrem* class (which is derived both from *istream and ostream*) used for both types of operations. In the header file *iostream* there exist the declarations of the principal classes, constants and objects used for this kind of operations. The most used objects are the followings:

- *cin* – used for data reading from the standard input device;
- *cout* – used for data writing at the standard output device;
- *cerr* – used for error message display;
- *clog* – used as *cerr*, but the buffer does not become empty for each error message.

For performing the I/O operations some operators were defined, which are the overloading of some C operators. For example, for *reading* the values from the keyboard was overloaded the right shift operator (>>), while for the *writing* operation was overloaded the << operator.

For example, for reading a value from the keyboard and assigned it to an int *n*, it can be write:

```
cin >> n ;
```

and for displaying the value of *n*:

```
cout << n ;
```

These operators have been overloaded for all built-in data types, also for strings. So the operators can have as parameters any built-in data type.

These operators can be concatenated because they return data, so there can be read and write many data with the same operator. For example:

```
int n = 7 ;
double x = 4.5 ;
cout << n << x ;
```

Example 2.8. A simple program for determining the sum of the elements of an array whose values are read from the keyboard:

```
#include <iostream>
using namespace std ;
int main()
{
  int n = 10 ;
  double s = 0, x[10] ;
  cout << ”Give the range elements : ” ;
  for (int i = 0 ; i < n ; i++)
  {
    cout << ”x[” << i << ”]= ” ;
    cin >> x[i] ;
    s += x[i] ;
  }
  cout << endl << ”s=” << s << endl ;
  return 0;
}
```

Remark. The reserved word *endl* represents the character ’\n’.

The C++ language has some member functions of the I/O classes that can be used for data reading and writing. For example, the function *get* (inserts a character in the output stream and returns the stream at the output), and *put* (extracts the next character from the input stream and returns the input stream) have the following declaration:

```
ostream& put(char c) ;
istream& put(signed char &c) ;
```

Example 2.8. A program that copy of the standard input file at the standard output file:

```
#include <iostream>
using namespace std ;
int main()
{
  int c ;
  while ( (c = cin.get()) != EOF)
    cout.put(c) ;
  return 0;
}
```