Appendix B

Solving Electromagnetic Problems Using C++ [1]

B.1 Introduction

The C++ programming language was developed by Bjarne Stroustrup in 1980, at Bell Labs (Murray Hill, New Jersey). It was considered to be an improvement on the C language by adding support for object-oriented programming. This appendix is meant to be a brief review of C++ as well as an introduction to using C++ to solve electromagnetic problems. Examples are offered to illustrate the points previously discussed. Style notes are also included to offer insight into what good programming practices are.

Most of the main features of the language are covered but the reader is encouraged to consider the references at the end of this appendix to obtain supplemental information on the more advanced topics of object-oriented programming. The reader is encouraged to pay close attention to the examples. The programs may contain subtle implementations of other concepts in addition to the main concept being covered at the time. (All the programs were developed using Microsoft Visual C++ 5.0.)

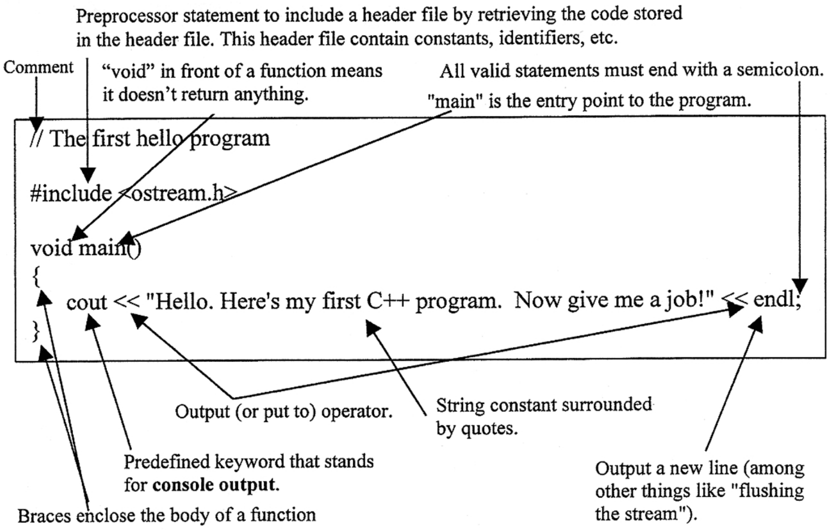
B.2 A Brief Description of C++

A. What Every New Programmer to a Language Should See First

Consider the obligatory "hello" program:

---

[1] Written by Raymond C. Garcia, Georgia Institute of Technology, Atlanta, GA 30332-0320.



```
// The first hello program

#include <ostream.h>

void main()
{
  cout << "Hello. Here's my first C++ program.  Now give me a job!" << endl;
}
```

Preprocessor statement to include a header file by retrieving the code stored in the header file. This header file contain constants, identifiers, etc.

Comment

“void” in front of a function means it doesn’t return anything.

All valid statements must end with a semicolon.

"main" is the entry point to the program.

Output (or put to) operator.

String constant surrounded by quotes.

Predefined keyword that stands for **console output.**

Output a new line (among other things like "flushing the stream").

Braces enclose the body of a function

The “hello” program simply prints a statement along with a carriage return.

One of the most notable statements in every C++ program is the “#include” statement. It is here where preprocessed code is included and incorporated with a program code. Table B.1 contains information on what header file to include based on what function is needed [1, 2].

B. Types and Declarations

The C++ language supports the commom intrinsic types as well as user defined types:

- Boolean – the keywords *true* and *false* are supported. All logical expressions evaluate to the type *bool.*

- Integer – the keyword *int* is used to represent integer values. Commom “unmodified” integers range from $-32,768$ to $32,767$.

- Floating-point – the keywords *float* and *double* are used to represent floating-point types. The commom “unmodified” ranges are 3.4e$-$38 to 3.4e$+$38 for floats and 1.7e$-$308 to 1.7e$+$308 for doubles.

- Character – the keyword ***char*** is used to represent character data. Most compilers implement a char as an 8-bit value which can hold 256 different values.

All of the types mentioned above except for *bool* can be modified by the following keywords:

- signed

- unsigned

Table B.1 Inherited C Include Files

Header File	Comments
assert.h	Used for debugging.
ctype.h	Allows you to check the case and type of char (i.e., is it upper/lower case or a digit, etc.).
errno.h	Several macros are defined to report error conditions.
float.h	Macros and constants used for serious numerical, floating-point programming.
iso646.h	Provides readable alternatives to certain operators or punctuators (i.e., “and” instead of &&).
limits.h	Goes hand-in-hand with float.h; offers the range limits for all data types.
locale.h	Formatting of numeric data; mostly informative.
math.h	Contains mathematical functions (sin, cos, etc.).
setjmp.h	Contains a macro, function, and type for bypassing normal function calls (similar to goto); be careful.
signal.h	Supports the handling of “signals” (conditions occurring at run-time).
stdarg.h	Macros contained here allow you to walk through a variable argument list.
stddef.h	As the name suggests, this headers contains “standard definitions.”
stdio.h	Contains functions that perform input and output (fopen, fclose, fprintf, etc.).
stdlib.h	Contains general utility functions.
string.h	Contains functions which facilitate the manupulation of array character data.
time.h	Provides time structures
wchar.h	Allows its user to perform input and output operations on wide streams or manipulate wide strings.
wctype.h	Similar to ctype.h except it acts on wide chars.

- long
- short

Table B.3 lists some of the basic types along with the commom bit widths and ranges [3].

<STYLE NOTE>: It is good programming practice to use the appropriate data type and modifier whenever possible. For example, do not use a long double when a float is really needed. This habit will prove more valuable for large programs where memory is a concern. This is especially useful when precision is of interest.

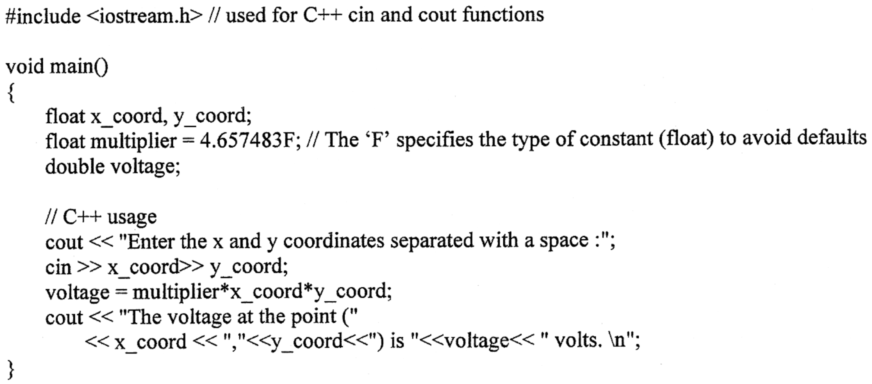
C. Input and Output with cin and cout

One of the major enhancements to C is the stream libraries which provide functionality for console/terminal and file input and output. This functionality is accessed through the keywords ***cin*** and ***cout***.

Table B.2 C++ Header Files [2]

	C++ Headers
Header File	**Comments**
bits	Provides a template and related function for manipulating a fixed-size sequences of bits.
bitstring	Similar to bits but functions operate on varying-length sequences of bits.
complex	Support functionality for representing and manipulating complex numbers.
defines	Defines a constant and several types that are used by many other headers.
dynarray	Defines a template which generically controls a sequence.
exception	Supports the handling of exceptions.
fstream	Provides stream buffers to assist the reading and writing of files.
iomanip	Provides template classes that provide extractors and inserters for information used in the class “ios.”
ios	Provides function signatures for controlling how to interpret input and output from a sequence of characters.
iostream	Include to perform input and output from a C++ program.
istream	Controls input from a stream buffer. “cin” is its most popular object.
new	Manages the allocation and storage of a program.
ostream	Controls output to a stream buffer.
ptrdynarray	Defines a template that supports a varying-size sequence of pointers to a generic object.
sstream	Used to define several template classes that support iostreams operations on sequences stored in an allocated array object.
streambuf	Defines macros and data types that control input from and output to char sequences.
string	Assists in the manipulation of varying-length sequences of chars.
strstream	Designed to assist in reading and writing char sequences stored in memory.
wstring	Similar to string but applies to wide characters.

Example B.1:



```
#include <iostream.h> // used for C++ cin and cout functions

void main()
{
     float x_coord, y_coord;
     float multiplier = 4.657483F; // The ‘F’ specifies the type of constant (float) to avoid defaults
     double voltage;

     // C++ usage
     cout << "Enter the x and y coordinates separated with a space :";
     cin >> x_coord>> y_coord;
     voltage = multiplier*x_coord*y_coord;
     cout << "The voltage at the point ("
          << x_coord << ","<<y_coord<<") is "<<voltage<< " volts. \n";
}
```

Table B.3 Modified and Unmodified Types Commonly Implemented

Data Type	Modifier	Width (# of Bits)	Range
char		8	−128 to 127
	signed	"	0 to 255
	unsigned	"	−128 to 127
int		16	−32, 768 to 32, 767
	unsigned	"	0 to 65, 535
	unsigned short	"	"
	signed	"	−32, 768 to 32, 767
	short	"	"
	signed short	"	"
	long	32	−2, 147, 483, 648 to 2, 147, 483, 647
	signed long	"	"
	unsigned long	"	0 to 4, 294, 967, 295
float		"	3.4e−38 to 3.4e+38
double		64	1.7e−308 to 1.7e+308
	long	80	3.4e−4932 to 1.1e+4932

Output:

```
Enter the x and y coordinates separated with a space :4.4 5.5
The voltage at the point (4.4,5.5) is 112.711 volts.
```

D. Pointers

Pointers appear to be one of the most confused concepts in C++. When used incorrectly, programs became unstable. A pointer is an item that holds the address of a defined variable. There are pointers to pointers and the like. When a pointer holds the address to the first element in an array, incrementing it will move it to the next address in the array. This may seem obvious but the main point is that different data types are different sizes in the machine they reside in. An int pointer will increment 2 bytes and a float pointer will increment 4 bytes if integers are represented with 2 bytes and floats are represented by 4 bytes. This is a machine dependent concept.

Some of the more popular uses of pointers are linked lists, dynamic memory allocation, and argument modification. Since structures and functions have not been discussed yet, the two examples of pointer usage given here are simple pointer assignment and dynamic memory allocation.

Example B.2:

```
#include <iostream.h>
void main(){ // simple pointer assignment
    int *intptr;
    int intvar=10, array_size;
    int *int_array_ptr;
```

```
        intptr = &intvar; // set pointer equal to the intvar's address
        cout << "Contents of intptr: "<<*intptr<<endl; // here, the "*" is called a dereference operator

        cout<<"Enter the size of the array: ";
        cin>>array_size;
        if (array_size > 0){
              int_array_ptr = new int[array_size];  // dynamic memory allocation
              for (int x=0;x<array_size;x++)
                    int_array_ptr[x]=array_size - x;
              for (int y=0;y<array_size;y++)
                    cout<<" Element " << y << "= ("<< int_array_ptr[y]<<")";
              cout << endl;
        }
  }
```

Output:

```
        Contents of intptr: 10
        Enter the size of the array: 5
        Element 0= (5) Element 1= (4) Element 2= (3) Element 3= (2) Element 4= (1)
```

E. Arrays

Unspecified arrays take the following form:

```
array_item_type array_name[no_of_elements];
```

The statement above tells the compiler to reserve enough space to hold no_elements of array_item_type. These are valid array declarations:

```
int my_ints[10]; // simple one-dimensional array of ints
char buffer[80]; // simple one-dimensional array of chars
char screen[24][80]; // a two-dimensional array of chars
```

Arrays can also be declared where the size need not be indicated explicitly. For instance, the statement:

```
int my_ints[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

implicitly tells the compiler that my_ints is an array of ten integers. To specify a particular element of the my_ints array you type the following:

my_ints[2].

Here, my_ints[2] refers to the item '3' not '2' as most people who deal with non zero-based arrays would expect.

Multi-dimensional arrays are initialized as shown in the following example:

Example B.3:

```
#include <iostream.h>

void main()
{
    int nums[2][3]={1,2,3,4,5,6};  // initialized 2-d array


    // print out the contents; element-by-element
    cout << "nums[0][0] = "<<nums[0][0] << endl;
    cout << "nums[0][1] = "<<nums[0][1] << endl;
    cout << "nums[0][2] = "<<nums[0][2] << endl;
    cout << "nums[1][0] = "<<nums[1][0] << endl;
    cout << "nums[1][1] = "<<nums[1][1] << endl;
    cout << "nums[1][2] = "<<nums[1][2] << endl;

}
```

Output:

```
nums[0][0] = 1
nums[0][1] = 2
nums[0][2] = 3
nums[1][0] = 4
nums[1][1] = 5
nums[1][2] = 6
```

<**STYLE NOTE**>: This is more of a warning than a style issue. One of the virtues of C++ is the fact that it is fast and compact. In order for the language to be fast, a feature such as array bounds checking is not included. This places more responsability on the programmer to be careful when assigning and referencing arrays. The consequences can be devastating.

F. Control Constructs

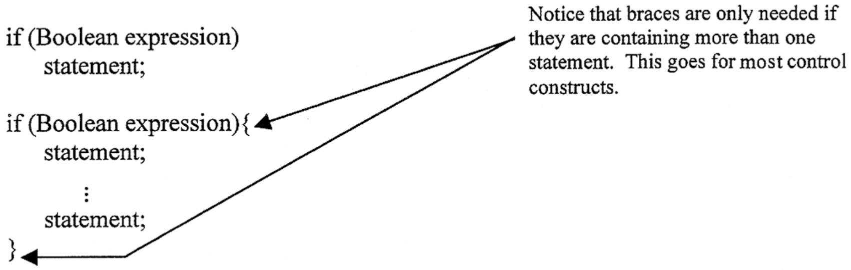
Relational operators, logical operators and control constructs go hand in hand. That is why they have been deferred until this section. Arithmetic operators are also an important part of control constructs. Table B.4 contains arithmetic, relational, and logical operators [4, 5].

The following control constructs are important.

Table B.4 Arithmetic, Relational, and Logical Operators

Operator	Type	Meaning	Comments
−−	Arithmetic	decrement	when placed in front a variable it is the decremented value that is used; when placed after a variable, the decrementing occurs after it is used
++	"	increment	prefix and postfix rules are the same as with the decrement operator
∗	"	multiplication	
/	"	division	remainder of integer division is truncated (e.g., $3/4 = 0$).
%	"	modulus	used for integer division
−	"	subtraction and unary minus	
+	"	addition	
>	relational	greater than	
>=	"	greater than or equal to	
<	"	less than	
<=	"	less than or equal to	
==	"	equal to	
! =	"	not equal to	
&&	Logical	AND	
||	"	OR	
!	"	NOT	

*The **if** statement:*



```
if (Boolean expression)
     statement;

if (Boolean expression){
     statement;
        ⋮
     statement;
}
```

Notice that braces are only needed if they are containing more than one statement. This goes for most control constructs.

*Nested **if**s:*

```
if (Boolean expression)
       if (Boolean expression)
              statement;
```

A ladder of ***if-else-if*** *statements:*

```
if (Boolean expression)
    statement;
else if (Boolean expression)
    statement;
else if (Boolean expression)
    statement;
else
    statement;
```

The ***for*** *loop:*

```
for (initialization; expression; increment)
    statement;

for (initialization; expression; increment)
{
    statement;
        ⋮
    statement;
}
```

where “initialization” initializes the variable(s) that control the loop as a counter, “expression” sets the condition on which the loop repeats, and “increment” defines how the control variable(s) are changed.

The ***switch*** *statement*

```
switch(expression){
    case CONSTANT:
        statement;
           ⋮
        statement;
        break;
        case CONSTANT:
                statement;
                      ⋮
                statement;
                break;
        default:
                statement;
                      ⋮
                statement;
}
```

The expression being switched must evaluate to a character or integer value.

The ***while*** *loop*

```
while (Boolean expression)
{
      statement;
            ⋮
      statement;
}
```

The ***do-while*** *loop*

```
do {
          statement;
                ⋮
          statement;
   } while (Boolean expression);
```

Example B.4:

```
#include <iostream.h>

void main()
{
      int choice = 0;  // variable used to store the selection
```

```
        // This sets up the menu
        cout << "Enter the Monte Carlo Method to Run:\n"
             << endl
             << "     (1) Floating Random Walk\n"
             << "     (2) Fixed Random Walk\n"
             << "     (3) Fixed-Radius Floating Random Walk\n"
             << "   Choose: ";

        cin >> choice; // get the user's choice

        // switch-case statements
        switch (choice){
        case 1:
            cout << "\nRunning the Floating Random Walk...\n";
            break;
        case 2:
            cout << "\nRunning the Fixed Random Walk...\n";
            break;
        case 3:
            cout << "\nRunning the Fixed-Radius Floating Random Walk...\n";
            break;
        default:
            cout << "Invalid choice\n";

        }
    }
```

Output:

```
Enter the Monte Carlo Method to Run:

     (1) Floating Random Walk
     (2) Fixed Random Walk
     (3) Fixed-Radius Floating Random Walk
  Choose: 2

Running the Fixed Random Walk...
```

The effect of using the ***continue*** *and* ***break*** *keywords*

- A ***continue*** statement skips the statements following it without exiting the loop.

- A ***break*** statement will exit from the inner-most loop or the current ***switch*** statement it is in.

G. Structures and Unions

Structures form the building blocks for writing programs that deal with data. Whether dealing with simple things like a recipe in a cook book to defining boundary-value problems to perform Monte Carlo simulations, data structures are all around us.

Syntax:

```
struct struct_tag{
        type member1;
        type member2;
        type member3;
} variable declarations;

union union_tag{
        type member1;
        type member2;
                  :
        type memberN;
} variable declarations;
```

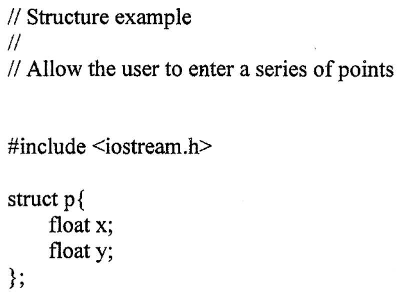
Members of unions and structures are accessed using “dot” notation. For example, to access *member1* of the union *myunion,* the following syntax is used:

myunion.member1.

If a pointer to a union was defined, an arrow would replace the period. For example,

myunion->member1.

Example B.5:



```
// Structure example
//
// Allow the user to enter a series of points


#include <iostream.h>

struct p{
      float x;
      float y;
};
```

```
void main()
{
    int num=0,i=0, c=1;

    p *points, *index;

    while (num <= 0){
        cout << "Enter the number of points to consider: ";
        cin >> num;
    }

    points = new p[num]; // allocate enough memory for the points
    index = points;

    for (i=0;i<num;i++){

        cout << "\nEnter the x value of point number "<<i+1<<": ";
        cin >> index->x;

        cout << "Enter the y value of point number "<<i+1<<": ";
        cin >> index->y;

        index++;
    }

    index = points; // reset the index

    cout << "\n\nThis is what you've entered: \n\n";

    do{
        cout << "Point number " << c++ << " is (" << index->x<<","<<index->y<<")"<< endl;
        index++;
    } while (i-- > 1);

    delete [] points;  // free up the memory
}
```

Output:

```
Enter the number of points to consider: 3

Enter the x value of point number 1: 1.1
Enter the y value of point number 1: 1

Enter the x value of point number 2: 22.2
Enter the y value of point number 2: 2

Enter the x value of point number 3: 3
Enter the y value of point number 3: 3.33
        This is what you've entered:

        Point number 1 is (1.1,1)
        Point number 2 is (22.2,2)
        Point number 3 is (3,3.33)
```

A union allows several variables to share the same memory location. For example, if 4 bytes are used to represent a float and 1 byte is used to represent a char, the union of the two would occupy 4 bytes. This allows a union to be considered many things. The main point is that this example of a union can be used to hold either the float or the char. Table B.5 should help to illustrate this point

Table B.5 Byte Storage within a Union

The float uses all four bytes			
1st Byte	2nd Byte	3rd Byte	4th Byte
1 byte for the char	1 byte of float	1 byte of float	1 byte for the char

H. Functions

The role of functions is important to program development in C++. All executable statements exist within functions. Offering effective ways to implement numerical techniques in C++ is impossible to do without describing how to use functions.

Part of defining a function is through the use of a prototype. A function prototype is a program statement which indicates to the compiler the type and number of arguments that a function requires. Type checking is improved with the use of prototyping by allowing the compiler to accurately check for type mismatches.

A function prototype takes the following form:

```
return_type function_name(argument_type argument_name, argument_type argument_name, etc.);
```

For example, to tell the compiler that you have a function called "add_them" that takes two integer arguments which will be called val1 and val2, and returns an integer value, you would type the following:

```
int add_them(int val1, int val2);
```

<STYLE NOTE>: C++ allows a programmer to prototype without specifying all arguments of the function with the use of ellipsis (...). It is considered good style to have the prototype appear exactly as the function's declaration.

Here is the code for an example that uses a function that adds two numbers together.

Example B.6:

```
#include <ostream.h>

int add_them(int val1, int val2);

void main()
{
    int x = 1, y=2;
    int z;
    z = add_them(x,y);
    cout << "The sum is " << z ;
}

int add_them(int val1, int val2)
{
    int sum;
    sum = val1 + val2;
    return (sum);
}
```

Output:

```
The sum is 3
```

Call-By-Value

The process in which the compiler creates a copy of the variable's value that is being passed in the function call is called *call-by-value.* This is the default calling convention for C++. In the example above, copies of x and y were used by the sum function. The major point here is that call-by-value cannot allow a function to modify the arguments used in the function call.

Call-By-Reference

When call-by-reference is used, the address of the argument is used as opposed to a copy of it. Call-by-reference is more efficient and faster than call-by-value. Less program memory is needed and the return statement is not necessarily needed to modify the variable. In C++, pointers as well as the reference type is used to implement call-by-reference.

As the following example illustrates, call-by-reference is used to modify a variable. Pay particular attention to the need of the dereference operator (*) in the function 'will_change_with_ptr' but other than the address operator (&), the code in the function 'will_change_with_ref' does not require the added consideration when working with pointers.

<STYLE NOTE>: It is advantageous to use reference types when implementing call-by-reference. It becomes unnecessary to pay attention as to whether the argument being used is a pointer or not. A major source of bugs exists in a code that inappropriately uses a variable that should be de-referenced first.

Example B.7:

```
#include <ostream.h>
// function prototypes
void will_not_change(int a);
void will_change_with_ptr(int *a);
void will_change_with_ref(int &a);
void main(){
    int x = 1;
    cout << "x originally is " << x << ".\n";
    will_not_change(x);  // call-by-value
    cout << "x after will_not_change is " << x << ".\n";
    will_change_with_ptr(&x);  // call-by-reference
    cout << "x after will_change_with_ptr is " << x << ".\n";
    will_change_with_ref(x);  // call-by-reference
    cout << "x after will_change_with_ref is " << x << ".\n";}
void will_not_change(int a){
    a+=1; // increment it by 1}
void will_change_with_ptr(int *a){
    *a+=1; // increment it by 1}
void will_change_with_ref(int &a){
    a+=1; // increment it by 1}
```

Output:

```
x originally is 1.
x after will_not_change is 1.
x after will_change_with_ptr is 2.
x after will_change_with_ref is 3.
```

Array Arguments

A common necessity in programming is the need to pass arrays to functions. The following code example shows two ways to use arrays as arguments. The first function has a pointer to the first array element. The second example uses an unsized array.

Example B.8:

```
                    #include <ostream.h>
                    // define the structure that is used in the prototypes
                    struct point{
                         float x;
                         float y;};
        // function prototypes
        void print_points_using_ptr(point *p);
        void print_points_using_unsized_array(point p[]);

        void main(){
             // initialize the array of points
             point bunch_of_points[5]={{1,1},{2,2},{3,3},{4,4},{5,5}};
             // print them using address of the first element
             print_points_using_ptr(bunch_of_points);
             cout << endl; // blank line in between lists of points

             // print them using an unsized array
             print_points_using_unsized_array(bunch_of_points);}
        void print_points_using_ptr(point *p){
             for (int j=0;j<5;j++)
                  cout << "point[" << j << "]: (" << p[j].x << "," << p[j].y << ")\n";}
        void print_points_using_unsized_array(point p[]){
             for (int j=0;j<5;j++)
                  cout << "point[" << j << "]: (" << p[j].x << "," << p[j].y << ")\n";}
```

Output:

```
point[0]: (1,1)
point[1]: (2,2)
point[2]: (3,3)
point[3]: (4,4)
point[4]: (5,5)

point[0]: (1,1)
point[1]: (2,2)
point[2]: (3,3)
point[3]: (4,4)
point[4]: (5,5)
```

Three more things are left to conclude our discussion about functions:

First, the keyword ***inline*** instructs the compiler to directly place the function code at the point in which it is invoked as opposed to making a function call. This is most useful for time-saving situations with short functions that are called many times. The compiler may ignore this directive in cases where the function has loops, switch, or goto statements.

Syntax:

```
inline type function_name(argument list){
       code statements
}
```

Second, the scope resolution operator "::" is used when there are two variables with the same name at different scopes. Here is some example code that illustrates its use:

Example B.9:

```
#include <ostream.h>

// file scope variable
int x=4;

// function prototypes
int times2(int c);
int times4(int c);

void main(){
      int z=1;
      int y;

      y = times2(z); // call function that uses local variable

      cout << "y is " << y << endl;
      cout << endl;

      y = times4(z); // call function that uses global variable
      cout << "y is " << y << endl;
}
    int times2(int c){
          int x=2;  // local variable
          return (c*x);
    }

    int times4(int c){
          return(c*(::x));  // could have used 'return(c*::x)'
    }
```

Output:

```
y is 2

y is 4
```

Third, passing arguments on the command line. This is an important feature that allows programmers to call their program and pass arguments to it. The following example shows how this is done by calling the function "main" with a month string followed by a year.

Example B.10:

```
#include <iostream.h>
void main(int argc, char *argv[]){
    if (argc <3)
        cout << "You must enter the month and year!" << endl;
    else
        cout << "The month is: "<<argv[1]<< endl;
        cout << "The year is: "<< argv[2] <<endl;
}
```

Output:

```
Enter: exd12 “September” “1999”

The month is: September
The year is: 1999
```

B.3 Object-Orientation

In order to discuss object orientation, we must first define what an object is. For most readers familiar with imperative language programming, an object is looked upon as a variable. But in the object-oriented world, an object is something that not only represents a value but also its behavior. For example, we can have an object called list. This list would not only contain the items within it but it would also represent the "behavior" of maintaining itself. In particular, this list object can contain methods that would add and delete items.

Another example of an object is a random walk. This random walk can have data that indicates its current position as well as how many steps it is currently taken. This random walk object can also have methods that perform the steps as well as functionality used to determine when a border is reached.

Why Object-Oriented Programming?

Object-oriented programming emerged out of the need to compensate for the problems associated with *procedural* languages. Procedural languages, as defined in [6] are designed such that "...programs are organized around control structures such as iteration and procedure invocation." What this means is that procedural language programs are simply a list of instructions that have the following principles [7]:

- single entry, single exit
- clearly defined inputs and outputs
- top-down hierarchical decomposition
- modular design for easy modification and reuse
- only sequence, selection, case, and iteration constructs.

For most small programs, procedural languages are suitable but as programs become larger and more complex, procedural programs become more difficult to change, debug, and adapt. Take a look at Fig. B.1 which describes the procedural paradigm [8].

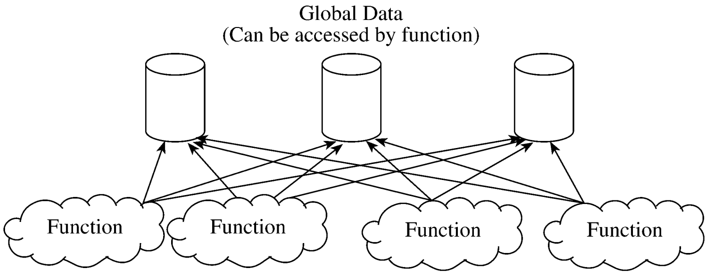


Figure B.1
Procedural programming paradigm.

As Fig. B.1 indicates, a procedural program is based on functions operating on data. The larger the program, the more functions there are that operate on the data. The problem here is that it becomes quite easy to have a function that corrupts the data shared by other functions. This can happen in several ways. A new programmer on a programming team for a large project may not be fully aware of the impact of a new function written, thereby inadvertently causing undesirable results. Another way of corrupting the data would be for a data structure to change without completely updating the rest of the program. For large programs, this is a daunting task. Every function that accessed the data structure needs to be modified to reflect the new changes.

Continuing our discussion on the limitations of procedural programming, take notice of Fig. B.2 which describes the object-oriented programming paradigm. The first point to make is that programs are organized in objects expressed as classes. Each class is composed of data and member functions that operate on the data. Given this object-centric approach to programming, both cases which would be problematic in a procedural language are inherently addressed in C++. For instance, any additional functions or data structure changes will benefit from the organized objects; the impact of the additional function is foreseen based on deciding which class it is a member of; and updates of objects due to data structure changes are tractable since it is known which members have access to it.

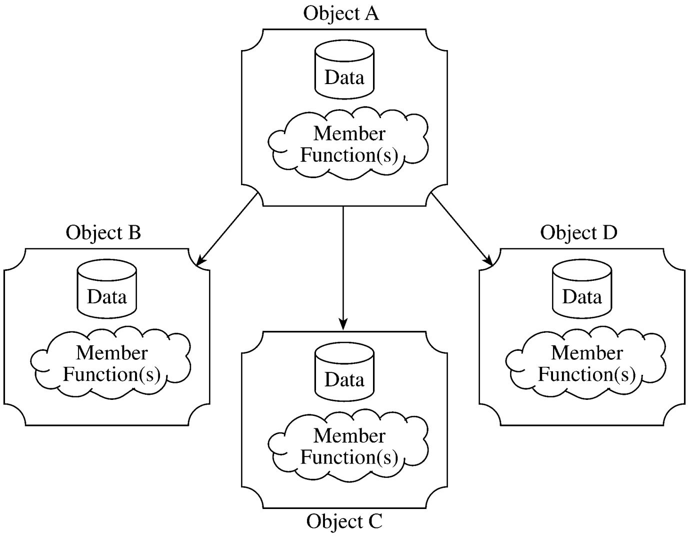


Figure B.2
Object-Oriented programming paradigm.

A. Inheritance

Inheritance is the process of having one object "inherit" data, functionality, etc. from another object. Taking another look at Fig. B.2, objects B, C, and D inherit from object A. Object A is known as the *base class* and objects B, C, and D are called *derived classes*.

Taking our random walk object described earlier, suppose we call this our "parent" object. Examples of "child" objects are floating random walk and fixed random walk. This parent-child relationship is such that both the floating random walk and fixed random walk objects inherit the mechanisms that represent their current location and amount of steps. Another inherited feature is the boundary checking functionality. Figure B.3 should help to drive this point home.

B. Polymorphism

Polymorphism is best described using functions. Recall our random walk object. In particular, the method used to perform steps. What if this method was aptly called "step." Now, since there are various types of walks (floating, fixed, etc.) with each differing by how they get to the border. The step method would need to act differently based on the type of object it was. For example, if it were a floating random walk object, it would have to take a step by picking a random point on a circle with radius set to the shortest distance to the border; but if it were a fixed random walk object, it would randomly choose one of four directions to step (simply north, south, east, or west). It is how the step method acts based on the object that invoked it that exemplifies

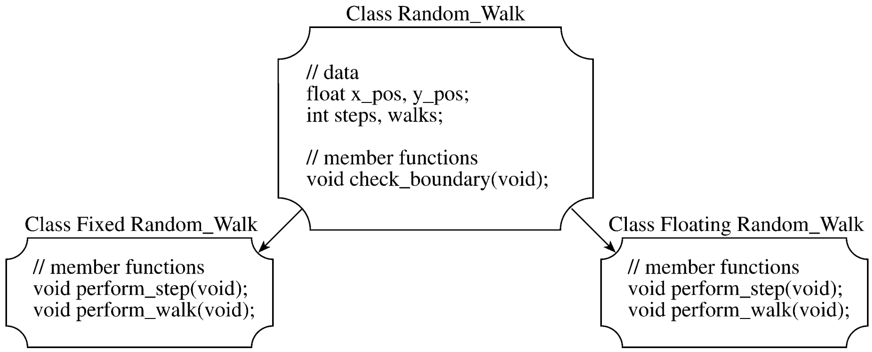


Figure B.3
Example illustrating inheritance.

polymorphism. The process of how the compiler knows which step member function to run is called *dynamic binding.*

C. Data Abstraction

Data abstraction is the concept of having data and methods that act on that data exist within one conceptual unit (namely a class). This concept of having data and "datacentric" methods existing within a single syntactic entity has at least two advantages:

- program modification is localized to this single class
- code changes pertaining to the encapsulated pieces can be changed in one area without being visible to the rest of the program and without affecting its behavior.

The simplest example to give about data abstraction is the common ***float*** data type. Here, the C++ language allows you to declare a float variable but it does not allow you to modify its floating-point representation. The language also allows you to add floating-point numbers but there is no way for you to write a program to change *how* the addition is performed.

D. Encapsulation

Encapsulation (also called data hiding) is the act of hiding object information details with various levels of access. Using our ***float*** example again, the floating-point representation as well as the mathematical operation details are *encapsulated.*

Using the class example in Fig. B.3. The encapsulation language feature allows an object to have members that are exclusive to just itself, derived objects, or the "public." Using the random walk example, the floating random walk object uses a data member “walk” from the class Random_Walk (see Fig. B.3) that specifies the amount of times it has reached a border. Since the class Floating_Random_Walk inherits from the class Random_Walk, it now has an int variable named “walk” that it can use in its function “perform_walk.” The data member “walk” is inherited. If the variable “walk” was made ***private***, derived classes would not be able to access it.

E. Overloading

Operator overloading is the process of changing the meaning of an operator. For instance, the add operator “+” naturally adds two numbers. This operator can be “overloaded” to work (or operate) on any type of object. Suppose we want to add two matrix objects. There is no built in functionality for adding two matrices. What needs to be done is to overload the “+” operator such that an element-by-element addition is performed. A more detailed look here requires looking at the matrix object as having a “+” method that operates on similar matrix objects such that the following expression:

```
matrixA + matrixB
```

calls the ‘+’ method of matrixA with matrixB as a parameter.

B.4 C++ Object-Oriented Language Features

The C++ language has built-in features which support object-oriented programming. Namely classes, operator overloading, virtual functions, templates, and exception handling. What follows is a brief description of each language feature along with supporting examples.

A. Classes

The use of classes is a major language feature that supports object-oriented programming. For a solid concise look at classes the reader is referred to [5]. A class can be looked at as a more powerful ***struct***. This is what a class declaration looks like:

```
class class_name{
private:
    private data and methods
public:
    public data and methods
protected:
    protected data and methods
} object names;
```

where,

private- indicates that these data items are to be accessible to the object itself; no children or any other part of the program. ***Private*** is the ***default*** access if none are indicated.

public- the data items are accessible to all derived classes.

protected- it is an otherwise private member that can be used only by itself and other objects that inherit them.

Example B.11: This example illustrates the relationship between public and private members of a class.

```
#include<iostream.h>
#include<math.h>

class myclass{
      double num; // all unspecified sections are assumed by the compiler to be private
public:
      void set_num(double);      // sets the value of num
      void print_num(void);   // prints the value of num
      void get_tan(void);    // computes num = tan(num)
      void times2(void);     // computes num = 2*num
};

// now it's time to define the functions
void myclass::set_num(double val){
      num = val;
}
void myclass::print_num(void){
      cout << "The number is: " << num << endl;
}
void myclass::get_tan(void){
      num = tan(num);
}
void myclass::times2(void){
      num = 2*num;
            }
            void main(){
                  myclass C;              // instantiate a myclass object named C
                  C.set_num(3.1415);  // num = 3.1415
                  C.print_num();
                  C.get_tan();            // num = -92.65E-6
                  C.print_num();
                  C.times2();             // num = -185.3E-6
                  C.print_num();
            }
```

Output:

```
                    The number is: 3.1415
                    The number is: -9.26536e-005
                    The number is: -0.000185307
```

Notice that in this example all member functions have access to the variable “num.”

Constructors and Destructors

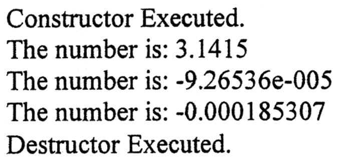
Within every class, member functions called a *constructor* and *destructor* exist. A constructor is used to initialize class variables or allocate memory storage. A destructor is used to return memory allocated. Constructors can accept arguments and can be overloaded while destructors cannot. The constructor takes on the same name as the class in which it exists while the destructor has the class name preceded with a “ ˜ ” character. The compiler automatically creates a constructor and destructor whenever they are not defined.

Example B.12:

```
#include<iostream.h>
#include<math.h>

class myclass{
    double num; // all unspecified sections are assumed by the compiler to be private
public:
    myclass(double val) { cout << "Constructor Executed.\n"; num=val;};
    ~myclass(){cout << "Destructor Executed.\n";};
    void print_num(void);   // prints the value of num
    void get_tan(void);     // computes num = tan(num)
    void times2(void);      // computes num = 2*num
};
void myclass::print_num(void){
    cout << "The number is: " << num << endl;
}
void myclass::get_tan(void){
    num = tan(num);
}
void myclass::times2(void){
    num = 2*num;
}
void main(){
    myclass C(3.1415);  // instantiate a myclass object named C setting num to 3.1415
    C.print_num();
    C.get_tan();        // num = -92.65E-6
    C.print_num();
    C.times2();         // num = -185.3E-6
    C.print_num();
}
```

Output:



```
Constructor Executed.
The number is: 3.1415
The number is: -9.26536e-005
The number is: -0.000185307
Destructor Executed.
```

As illustrated in the example, the constructor and destructor are executed automatically every time the class is instantiated. This example also illustrates how a constructor is overloaded and takes a parameter to initialize the variable "num."

B. Operator Overloading

As described earlier, operator overloading takes place when a standard operator is redefined to mean something else. The syntax for overloading an operator is:

type ***operator*** overloaded_operator(parameter list)

Operator overloading can only occur from within the object that the method exists.

Member functions can also be overloaded. It is possible to have several member functions with the same name behaving differently. The one condition is that they must differ at least in the arguments they each take.

The following example is a class which converts from polar form to rectangular form and the reverse.

Example B.13:

```
#include <ostream.h>
#include <math.h>
// Create a Cartesian and polar structure
struct cartesian{ double x_pos,y_pos;}; struct polar{ double angle, magnitude;};
class myclass{
       polar pval; cartesian cval;
public:
       // constructor is overloaded
       myclass(cartesian cart){cval.x_pos=cart.x_pos;cval.y_pos=cart.y_pos;      pval.magnitude=0;pval.angle=0;};
       myclass(polar pol){pval.angle=pol.angle;pval.magnitude=pol.magnitude;  cval.x_pos=0;cval.y_pos=0;};
       // convert is overloaded; one takes polar the other takes cartesian
       void convert(polar);       void convert(cartesian);  void print_vals(void);};
void myclass::convert(polar pol){
       cval.x_pos=pol.magnitude*cos(pol.angle);   cval.y_pos=pol.magnitude*sin(pol.angle);}
void myclass::convert(cartesian cart){
       pval.angle = atan(cart.y_pos/cart.x_pos);      pval.magnitude = sqrt(pow(cart.y_pos,2)+pow(cart.x_pos,2));}
void myclass::print_vals(void){
       cout << "  x_pos: " << cval.x_pos << "  y_pos: "<< cval.y_pos << endl;
       cout << "  magnitude: " << pval.magnitude << "  angle: "<< pval.angle << endl;}
void main(){
       polar p;   p.magnitude = 10;   p.angle = 45;
       myclass X(p);
       cout << "Before Polar Conversion:\n";
       X.print_vals();X.convert(p); // convert the polar point to cartesian
       cout << "\nAfter Polar Conversion:\n";
       X.print_vals();
       cartesian c; // C++ is case sensitive so c is different from C
       c.x_pos = 15.5;       c.y_pos = 22.5;
       myclass Y(c);
       cout << "\nBefore Cartesian Conversion:\n";
       Y.print_vals();Y.convert(c); // convert the cartesian point to polar
       cout << "\nAfter Cartesian Conversion:\n";
       Y.print_vals();
}
```

Output:

```
Before Polar Conversion:
  x_pos: 0  y_pos: 0
  magnitude: 10  angle: 45

After Polar Conversion:
  x_pos: 5.25322  y_pos: 8.50904
  magnitude: 10  angle: 45

Before Cartesian Conversion:
  x_pos: 15.5  y_pos: 22.5
  magnitude: 0  angle: 0

After Cartesian Conversion:
  x_pos: 15.5  y_pos: 22.5
  magnitude: 27.3222  angle: 0.967566
```

C. Templates

For action that is the same for different data types, two options exist for implementing this case scenario. One option is to write a different class or function for each data type. Another option is to use templates to create generic functions and classes.

The following example illustrates this point. Suppose we want to swap two numbers. There has to be a different function for each type of number (an int, float, double, etc.). What template functions allow us to do is create one function that acts independently of the data type it works on.

Example B.14:

```
// the use of a template function to
// swap values of any type
#include <ostream.h>

template <class type1>              // no line can exist between the template keyword
void swapper(type1 &x, type1 &y) {  // and the function
      type1 z;                      // the '&' is used to reference the variable; meaning to
      z = x;                        // modify the variable itself not a copy of it.
      x = y;
      y = z;
}
void main(){
      int a = 1, b = 2;
      float i = 3.14F, d = 4.22F;
      cout << "Before swapping: \n";
      cout << "a = 1,    b = 2\n";
      cout << "i = 3.15, d = 4.22\n\n";
      swapper(a,b);
      swapper(i,d);
      cout << "After swapping: \n";
      cout << "a = " << a << " b = " << b << endl;
      cout << "i = " << i << " d = " << d << endl;
}
```

Output:

```
Before swapping:
a = 1,    b = 2
i = 3.15, d = 4.22

After swapping:
a = 2 b = 1
i = 4.22 d = 3.14
```

D. Exception Handling

Exception handling is a language feature that allows a programmer to have specific functionality executed given a particular error occurs. In the object-oriented world this is known as is *throwing* and *catching* exceptions.

Syntax:

```
try {
      // put code here that you want to trap erros
}
```

```
catch (type argument) {
    // code the execute in the event an exception is thrown.
}
more catch functions can follow...
```

Here is a simple example illustrating the use of exception handling. What it does is “throw” an exception immediately after it is found that a matrix element is negative.

Example B.15:

```
#include <iostream.h>
void main(){

    int int_matrix[5][5]={1,2,3,4,5,2,4,-6,8,10};
    int x,y;

    try{  // try block
        for (x=0;x<5;x++){
            for (y=0;y<5;y++){
                if (int_matrix[x][y] < 0)
                    throw (int_matrix[x][y]);    // exception thrown

                    // any code below this line (in this try block)
                    // will not execute if an exception has been
                    // thrown.
                cout << " Element["<<x<<"]"<<"["<<y<<"] = "<<int_matrix[x][y]<<endl;
            }
        }
    }

    catch(int ex_type){
        cout << "\nException Caught.  The type is: " << ex_type<<endl;
    }
}
```

Output:

```
Element[0][0] = 1
Element[0][1] = 2
Element[0][2] = 3
Element[0][3] = 4
Element[0][4] = 5
Element[1][0] = 2
Element[1][1] = 4

Exception Caught.  The type is: -6
```

E. Files and Streams

We all know what a file is, but what is a *stream?* In C++, a *stream* is a logical interface from which all I/O is operated through. The relationship between a file and a stream is one where a stream is associated and disassociated with a file through the "open" and "close" operations.

The following example creates a file with records where each record contains the following information:

- Coordinate value (x,y).
- Angle made with the x-axis (degrees).
- sine of the first-quadrant angle.

The file is then read back and the info is echoed out. Several concepts are illustrated in this example. Namely, the following:

- How to open a file for writing.
- How to open a file for reading.
- How to overload the extraction operator (<<).
- How to overload the insertion operator (>>).

Example B.16:

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>

#define PI 3.14159265

class point{
        double angle_rad;       // angle in radians
public:
        point(){};
        point(double xval,double yval){x=xval;y=yval;angle=0;sine_val=0;cosine_val=0;};
        double x,y;             // point ordinate values
        double angle;           // the angle formed with the x-axis
        double sine_val;        // the sin of the angle
        double cosine_val;      // the cosine of the angle
        void get_angle();       // get the angle from the point
        void get_cos();         // get the cos of the angle
        void get_sin();         // get the sin of the angle
    };

    void point::get_angle(void){
        angle_rad = atan(y/x);
        angle = atan(y/x)*180/PI;
    }

    void point::get_cos(){
        cosine_val = cos(angle_rad);
    }

    void point::get_sin(){
        sine_val = sin(angle_rad);
    }

    // overload the insertion operator
    ostream &operator<<(ostream &ostr, point pt)
    {
        ostr << pt.x <<" ";
        ostr << pt.y << " ";
        ostr << pt.angle << " ";
        ostr << pt.sine_val << " ";
        ostr << pt.cosine_val<< " ";
        return ostr;
    }
    // overload the extraction operator
    istream &operator>>(istream &istr, point &pt)
    {
        double val;
        istr >> pt.x;
        istr >> pt.y;
        istr >> pt.angle;
        istr >> pt.sine_val;
        istr >> pt.cosine_val;
        return istr;
    }
void main()
{

    point p1(1,1), p2(2,2), p3(3,3);
    point p4, p5, p6;
    p1.get_angle(); p1.get_sin();
    p2.get_angle(); p2.get_sin();
    p3.get_angle(); p3.get_sin();

    ofstream out("example.out");
    if (!out)
    {
        cout << "Error with creating output file 'example.out.'\n";
        exit(0);
    }
```

```
        out << p1;
        out << p2;
        out << p3;
        out.close();

        ifstream in("example.out");
        if (!in)
        {
                cout << "Error with opening file 'example.out.'\n";
                exit(0);
        }

        in >> p4;
        in >> p5;
        in >> p6;
        in.close();

        cout << "Point: (" << p4.x << "," << p4.y << ")" << endl;
        cout << "Angle: " << p4.angle << endl;
        cout << "Sin(angle): " << p4.sine_val << endl;
        cout << "Point: (" << p5.x << "," << p5.y << ")" << endl;
        cout << "Angle: " << p5.angle << endl;
        cout << "Sin(angle): " << p5.sine_val << endl;
        cout << "Point: (" << p6.x << "," << p6.y << ")" << endl;
        cout << "Angle: " << p6.angle << endl;
        cout << "Sin(angle): " << p6.sine_val << endl;

}
```

Output:

```
Point: (1,1)
Angle: 45
Sin(angle): 0.707107
Point: (2,2)
Angle: 45
Sin(angle): 0.707107
Point: (3,3)
Angle: 45
Sin(angle): 0.707107
```

B.5 A Final Note

The C++ programming language is powerful. What comes with this power is the required complexity to tap into it. For a more comprehensive view of C++ with annotations and commentaries, [9] is the clear choice. Since becoming a solid C++ programmer requires practice and guidance, I recommend [10]–[12] for gaining good insight in solid design. As your skill increases, at some time you may come across obscure, subtle issues with C++. In these instances, you will find [13] a valued addition to your library. Furthermore, for a more advanced discussion on templates,

[14] is a good place to start. Finally, any confusion with object-orientation in general should be cleared with the help of [15, 16].

References

[1] Plauger, P.J., *The Standard C Library.* Englewood Cliffs, NJ: Prentice-Hall, 1992.

[2] Plauger, P.J., *The Draft Standard C++ Library.* Upper Saddle River, NJ: Prentice-Hall, 1995.

[3] Schildt, H., *C++ from the Ground Up.* Berkeley, CA: Osborne McGraw-Hill, 1994, p. 41.

[4] Stroustrup, B., *The C++ Programming Language.* 3rd ed., Reading, Massachusetts: Addison-Wesley Publishing Company, 1997, pp. 119–121.

[5] Pappas, C.H., Murray, W.H., *Borland C++ Handbook.* Berkeley, CA: Osborne McGraw-Hill, 1991, pp. 160–171.

[6] Finkel, R.A., *Advanced Programming Language Design.* Menlo Park, CA: Addison-Wesley Publishing Company, 1996, p. 267.

[7] Cezzar, Ruknet, *A Guide to Programming Languages: Overview Comparison,* Norwood, MA: Artech House, Inc., 1995, p. 263.

[8] Lafore, R., *Object-Oriented Programming in Microsoft C++.* CA: Waite Group Press, 1993, pp. 5–6.

[9] Ellis, M.A., Stroustrup, B., *The Annotated C++ Reference Manual.* Reading, MA: Addison-Wesley Publishing Company, 1990.

[10] Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* MA: Addison-Wesley Publishing Company, Inc., 1992.

[11] Meyers, *More Effective C++: 35 new Ways to Improve Your Programs and Designs.* Massachusetts: Addison-Wesley Publishing Company, Inc., 1996.

[12] Eckel, B., “C++ Programming Style Guides,” Unix Review, March 1995, pp. 43–54.

[13] Eckel, B., *C++ Inside & Out.* Berkeley, CA: Osborne McGraw-Hill, 1993.

[14] Glass, G., Schuchert, Brett, *The STL <Primer>.* New Jersey: Prentice-Hall, 1996.

[15] Khoshafian, S., Abnous, R., *Object Orientation: Concepts, Languages, Databases, User Interfaces*. New York: John Wiley & Sons, Inc., 1990, pp. 6–10.

[16] Meyer, Bertrand, *Object-Oriented Software Construction,* 2nd ed., Upper Saddle River, NJ: Prentice-Hall, 1997.