CHAPTER 1
INTRODUCTION TO DATA STRUCTURE AND ALGORITHM.

1. What is data structure?

A **data structure** is a *representation of data* and the *operations allowed on that data*.
Data structure allows us to achieve important Object Oriented goals, component reuse. Once a data structure has been implemented, it can be use over and over again in a various applications.

The study of Data Structure concern with two parts:

- Data structure (description of the type of data to be stored/how to organize data).
- Algorithm (description of operations that can be performed on the data/how to manipulate data).

The *examples* of the data structure are an array, linked list, tree, stack, queue and graph.

2. What is abstract data type?

ADT consists of a set of a definition that allows the programmers to use the function while hiding the implementation. With an ADT users are not concerned with how the task is done but rather with what it can do. This generalization of operations with unspecified implementation is known as *abstraction.* We abstract the essence of the process and leave the implementation details hidden.

The concept of data abstraction means:

- We know *what* a data type can do
- *How* it is done is hidden

Consider the concept of a list. We can use an array or a linked list to support a list. If we place our list in ADT, users should not be aware of the structure we use. As long as they can insert and retrieve data, it should make no difference how we store the data.

Let us now formally defined an ADT. An **abstract data type** is a data declaration packaged together with the operations that are meaningful for the data type.

Abstract data type:

- Declaration of data
- Declaration of operation
- Encapsulation of data and operations

ADT has three things associated with it :

typeName *//name of the ADT*
clockType

domain *//set of values belonging to the ADT* each clockType value is a time of day in the form of hours, minutes, seconds.

operations *//set of operations on the data*
Set the time.
Return the time.
Print the time.
Increment the time by one second.
Increment the time by one minute.
Increment the time by one hour.
Test the two times to see whether they are equal.

3. Introduction to C++

1.Main function

Every C++ program must include the following code:

```
int main( )
{
<local data declaration>
<statement>
return 0;
}
```

This is called the *main function.* The program statements that are to be executed are placed between the braces. That section is called the body of the *main ( ) function.*

Here is a simple program that prints a message:

```
#include <iostream>    // defines the std::cout and std::endl objects
int main ( )
{// prints "Welcome to Data Structure and Algorithms Class!"
std::cout << "Welcome to Data Structure and Algorithms Class!"<<std::endl;
}
Welcome to Data Structure and Algorithms Class!
```

The preprocessor directive on the first line tells the C++ compiler to include the definitions from the standard header file iostream that is part of the Standard C++ Library. It defines the cout object and the endl object in the std namespace. The scope resolution operator :: is used to indicate the location of those definitions.

Using the standard std namespace

```
#include<iostream>
using namespace std;
int main( )
{//prints “Welcome to Data Structure and Algorithms Class!”
cout<< “Welcome to Data Structure and Algorithms Class!”<<endl; }
 Welcome to Data Structure and Algorithms Class!
```

The using declaration on the second line adds the name std to the local scope, obviating the std::scope resolution prefix on the cout and endl objects.

In this course we will use a pre-Standard compiler. We take out the using namespace std.

```
#include<iostream.h>
int main( )
{//prints “Welcome to Data Structure and Algorithms Class!!”
cout<< “Welcome to Data Structure and Algorithms Class!”<<endl; }
 Welcome to Data Structure and Algorithms Class!
```

2.Programs comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

// line comment
discards everything from where the pair of slash signs (//) is found up to the end of that same line

/* block comment */
discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line.

3. Simple data type

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2bytes	1 wide character

4. Literals

Character and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). All of them are preceded by a backslash (\). Here you have a list of some of such escape codes:

\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

5.Variables and Declaration

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable :

int a; // declares a variable of type int with the identifier a.
float mynumber; //declares a variable of type float with the identifier mynumber.

ex: int
main() {
int n; *//declaring variables n of type int*
n=44; *//initializing variables and n is assigned value 44*
cout<<"The value of n is"<<n<<'\n'; }
The value of n is 44

6. Standard input (cin) and output (cout) operator

ex: int
main( ) {
int n; cin >> n; cout << "The value of n
is" << n << '\n';
}
The value of n is 44

- >> is called the input operator/extractor .

The standard input stream cin is connected to the keyboard. It is used to extract input from the cin input stream.

- << is called the output operator/inserter.

The standard output stream cout is connected to the terminal screen. It is used to insert output to the output stream.

7. Expression and assignment

Standards operators are provided to construct expression that performs arithmetic, comparisons, logic, and assignment.

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

= =		Is equal to
<		Is less than
>		Is greater than
<=		Is less than or equal to
>=		Is greater than an equal to
!=		Is not equal to
&&	1	Logical and
1		Logical or
!		Logical not
( )		Parentheses
=		Assignment operator

ex:

```
int main ( ) { int m = 33;    int n = 7;    int q = m/n;
//the quotient of m by n    int r = m%n          //the
remainder of m by n
cout<<m<<"/"<<n<<"="<<q<<'\n';
cout<<m<<"%"<<n<<"="<<r<<'\n';
cout<<q<<"*"<<n<<"+"<<r<<"="<<q*n=r<<'\n';
}
```

33/7=4
33%7=5
4*7+5=33

8. Conditional

i. The if and if…else statement

- The *if* statement is used for conditional execution. Its syntax is:

if(condition)statement;

The statement will be executed only if the condition is true.
The parentheses around the condition are required.
ex: int n;
cin>>n;
if(n>2) cout<<"ok.Thanks."; cout<<"Goodbye.\n";
If user input integer greater than 2,the output:
Ok. Thanks. Goodbye.
Otherwise the output will be:
Goodbye.

- The *if…else* statement is the same as the if statement with an appended else clause:

```
if (condition) statement1;
else statement2;
```

If the condition is true, only statement1 is executed; otherwise only statement2 is. *ex:*

```
if (n%2= =0) cout <<"n is even\n"; else
cout<<"n is odd\n";
```

The value of the expression n%2 is the remainder from dividing n by 2; that will be 0 if n is even, or 1 if n is odd. So this if…else statement will print "n is even" only if n is even, and will print "n is odd" if n is odd.

ii.*switch* statement.

The if statement is a conditional; its action depends upon the value of a condition,which is a Boolean expression. C++ also has a switch statement. Its action depend upon the value of an integer expression. *ex:*

```
int m,n;
cout<<"Enter two integers:";
cin>>m>>n; char op; cout<<"Enter an
operator(+,-,*,/,%):"; cin>>op;
cout<<"\t"<<m<<op<<n<<"=";
switch(op) { case '+' :
cout<< m+n;
        break;
case '-' : cout<<m-n;
        break;
case '*' : cout<< m*n;
break;
case '/' : cout<< m/n;
break;
case '%' : cout<< m%n;
}
}
```

```
Enter two integers : 30  7
Enter an operator (+,-,*,/,%) :%
        30%7 = 2
```

iii. Conditional expression operator.

In addition to the *if* and *switch* statements, C++ also has the conditional expression operator for conditional execution. Its syntax is

(condition? value1:value 2)

Its value is value 1 if the condition is true, and value2 if condition is false.
ex: int m,n:
cout<<"Enter two integers:"; cin>>m>>n;
cout<<"Their maximum is "<<(m>n?m:n);
Enter two integers : 44 33
Their maximum is 44

9.Iteration

i. The *while* and *do…while* statement

- The *while* statement repeats the execution of a statement while its control condition is true. Its syntax is

while(condition)statement;

The system will repeatedly evaluate the condition and execute the statement until the condition is false. *ex:*
int n=1; int
sum=0;
while(n<=100)
{ sum +=
n*n;
++n;
} cout<<sum<<'\n';
338350

The loop iterates 100 times, once for each value of n from 1 to 100. After accumulating 100*100 into sum, it increments n to 101. Then the control condition (n<=100) is false, which stops the loops.

- using a *do…while* loop

In some cases, it is preferable to execute a loop's statement once before its control condition is evaluated. That can be done with a *do…while* loop. Its syntax is

do statement while (condition);

It works the same way as a while loop except that the condition is evaluated after the statement is executed instead of before.

ii.using *for* loop

for (initialization; condition; increase) statement;

Repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

ex: int main( ) { const
int N=100; double
sum=0.0; for (int x=1;
x<=N; x++) sum +=
1.0/x;
cout << "The sum of the first "<<N<<"reciprocals is"<<sum; }
The sum of the first 100 reciprocals is 5.18738

10.Functions.

A function is a subprogram that can be called (invoked) from another function and can return a value to it. Every C++ program is required to begin with the main ( ) function. Relegating separate tasks to separate functions is a fundamental programming technique that leads to simpler and more efficient programs.

void printRow(const int,const int); // prototype

int main ( )

```
{ const int N=10; for
int(i=0; i<N; i++)
printRow (i,N);
}
void printRow(const int row, const int N)   //implementation
{
for (int j=0; j<2*N; j++) if (j<N-
row l l j>N+row) cout<<" "; else
cout<<"*";

cout<<"\n";
}
```

This program has relegated the task of printing one row to separate printRow ( ) function. This is a void function because it does not return anything to main ( ). It has two parameters : row and N. They are passed values from the arguments i and N.

The function is declared by the one-line prototype above main ( ), and it is defined by its complete implementation below main ( ). Note that the prototype omits the parameter names (optional) and ends with a semicolon (required).

4.Arrays

An array is a sequence of contiguous storage locations all of which can be accessed by the single array name followed by an integer subscript called the index of the array. If a is the name of the array, then its elements are accessed using a[0], a[1], a[2], a[3], etc. The number of elements in the array is called its dimension. In C++, the array index is always begins with 0.

ex; an array of strings.

```
int main ( )
{string a[4];  //an array of 4 strings
a[0] = "Microsoft";
a[1] = "Oracle"; a[2]
= "Inprise"; a[3] =
"IBM"; for (int i = 0;
i<4; i++)
        cout<< "a["<<i<<"] = "<<a[i] << "\n";
}
```

a[0] = Microsoft

a[1] = Oracle
a[2] = Inprise
a[3] = IBM

Arrays are usually processed with *for* loops.

Arrays can be initialized when they are declared using an *initializer list.*

ex: Using an Initializer List to Initialize an Array
```
int
main( )
{ string a[ ] = {“Exxon”, “Shell”, “Texaco”, “BP”};
for (int i=0; i<4; i++)                    cout<<
“a[“<<i<<”] = “<<a[i]<<”\n”;
} a[0] =
Exxon a[1] =
Shell a[2] =
Texaco a[3] =
BP
```
Notice that the array dimension can be omitted when an initializer list is used. The compiler will set the dimension equal to the number of elements in the list.

i.Dynamic Arrays

An Array is declared the previous example is called a *static array* because it is allocated at compile time. The dimension must be a constant integer.

The new operator can be used to create a dynamic array whose dimension may be a variable integer. Dynamic arrays are allocated at run time.

ex : Using a dynamic array
```
int main ( )
{ int n;
cout<< “How many children do you have?”; cin>>n;
string* child = new string[n];
cout<<”Please give me the name of your “ << n
      << “children:\n”;
for (int i=0; i<n; i++) {
cout<<”\t”<<i+1<<”:”;
cin>>child[i];
} cout<<”They
are”<<child[0]; for (int i=1;
i<n; i++) { cout<<”,
“<<child[i]; }   cout<<”\n”;
}
```

```
How many children do you have? 4
Please give me the name of your 4 children:
1 : Sara
2 : John
3 : Andrew
4 : Michael
They are Sara, John, Andrew, Michael
```

Notice that the name of a dynamic array is a pointer to its element type.

The syntax for declaring a dynamic array is

`xxxx* a = new xxxx[n];` where xxxx is the element type, and n is the dimension of the array, which maybe any integer expression.

ii.Passing an Array to A Function

An array parameter can be declared like this:

```
void sort(double a[ ], int size);
```

But in C++, an array name is actually a constant pointer to its first element. So an array parameter can also be declared like this:

```
void sort(double* a, int size);
```

The two forms are equivalent. Either way, the function is called by passing the array name as the argument, like this:

```
sort(a, 100);
```

ex: Finding the Maximum Element of an Array.

```
int main( )
{ int a[] = {44,77,33,66,55,88,22};
cout<<"max(a,7) = "<<max(a,7)<<"\n";
}
int max(int* a, int n)
{ int m=a[0]; for (int
i=1; i<n; i++)
        if (a[i]>m) m=a[i];
return m; }
max(a,7) = 88
```

iii.Multidimensional arrays

Multidimensional arrays are declared the same way as one dimensional arrays:

```
double m [5] [4];    // a two-dimensional array
int x[4] [2] [4] [3] [2];   //a five-dimensional array
```

They can also be initialized the same way:

int c[2] [3] = { {22,66,88} , {55,77,44}}

Notice that a two dimensional array can be regarded as an array of arrays.
When processing a multidimensional array, it is usually helpful to use a *typedef* to define the array type, especially if it has to be passed to a function.

ex: Processing a two-dimensional array

```
#include<iostream.h>
const int ROWS=2; const
int COLS=3;
typedef int Array[ROWS][COLS];   //define the type Array void
swapCols(Array,int,int);
void print(const Array);

int main ( )
{ Array a= { {11,33,55}, {22,44,66}};
print(a); swapCols(a,1,2); print(a);
}

void swapCols (Array a, int c1, int c2)
{ for (int i=o; i<ROWS; i++)
{ int temp = a[i][c1];
a[i][c1] = a[i][c2];
            a[i][c2] = temp;
          }
}

void  print(const  Array  a)  {
for (int i=0; i<ROWS; i++)
          { for (int j=0;  j<COLS;  j++)
          cout<<a[i][j]<<" ";
cout<<"\n";
          }
          cout<<"\n";
```

```
}
11  33  55
22  44  66

11  55  33
22  66  44
```

The *typedef* defines Array to be the type for arrays of ints with ROWS rows and COLS columns. Those two constant are defined to be 2 and 3. The call *swapCols(a,1,2)* function interchanges column 1 with column 2(which are the second and third columns since numbering begins with 0).

5.Structures

A structure is a group of data elements grouped together under one name. These data elements, known as *members*, can have different types and different lengths.

```
struct   structure_name
{
member_type1          member_name1;
member_type2          member_name2;
member_type3          member_name3;
} object_names;
```

where structure_name is a name for the structure type, object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a type and a valid identifier as its name.

The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as structure_name is created and can be used in the rest of the program as if it was any other type. For example:

struct product { // structure type called product with two members: weight and price
int weight; float price;
} ;
product apple;
product banana, melon; //declare three objects of *product* type

Or we can declare like this :

struct product {
int weight; float
price;
} apple, banana, melon;
Once we have declared our three objects of a determined structure type (apple, banana and melon) we can operate directly with their members. To do that we use a dot (.) inserted between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

apple.weight apple.price
banana.weight
banana.price
melon.weight
melon.price

```
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
struct movies_t {
string title;
int year;
} mine, yours;
void printmovie (movies_t movie);
int main ()
{
string mystr;
mine.title = "2001 A Space Odyssey";
mine.year = 1968;
cout << "Enter title: ";
getline (cin,yours.title);
cout << "Enter year: ";
getline (cin,mystr);
stringstream(mystr) >> yours.year;
cout << "My favorite movie is:\n ";
printmovie (mine);
cout << "And yours is:\n ";
printmovie (yours);
return 0;
}
void printmovie (movies_t movie)
{
cout << movie.title;
cout << " (" << movie.year << ")\n";
}
```

```
Enter title: Alien
Enter year: 1979
My favorite movie is:
2001 A Space Odyssey (1968)
And yours is:
Alien (1979)
```

The example shows how we can use the members of an object as regular variables. For example, the member yours.year is a valid variable of type int, and mine.title is a valid variable of type string.

The objects mine and yours can also be treated as valid variables of type movies_t, for example we have passed them to the function printmovie as we would have done with regular variables. Therefore, one of the most important advantages of data structures is that we can either refer to their members individually or to the entire structure as a block with only one identifier.

6.Class

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
  } object_names;
```

All is very similar to the declaration on structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords:

- private members of a class are accessible only from within other members of the same class or from their *friends*.
- protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. For example:

```
class CRectangle {
    int x, y;   public:
 void set_values (int,int);
int area (void);
} rect;
```

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.
After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect .

rect.set_values (3,4);
myarea = rect.area();

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:

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

class CRectangle {
   int x, y;   public:
   void set_values (int,int);
   int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b)
{
   x = a;   y = b;
}

int main ()
{
   CRectangle rect;   rect.set_values (3,4);
   cout << "area: " << rect.area();
   return 0;
}
```

```
area: 12
```

The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class declaration itself.

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly

included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class and to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them.

```
// example: one class, two objects
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;   public:
    void set_values (int,int);
    int area () {return (x*y);}
};
 void CRectangle::set_values (int a, int b)
{
     x = a;   y = b;
}
 int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() <<
endl;   return 0;
}
```

```
rect area: 12
rectb area: 30
```

In this concrete case, the class (type of the objects) to which we are talking about is CRectangle, of which there are two instances or objects: rect and rectb. Each one of them has its own member variables and member functions.

Notice that the call to rect.area() does not give the same result as the call to rectb.area(). This is because each object of class CRectangle has its own variables x and y, as they, in some way, have also their own function members set_value() and area() that each uses its object's own variables to operate.

That is the basic concept of *object-oriented programming*: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

i.Constructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function area() before having called function set_values()? Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void.

We are going to implement CRectangle including a constructor:

```
// example: class constructor #inc
using namespace std;

class CRectangle
{
   int width, height;   public:
   CRectangle (int,int);
   int area () {return (width*heigh
};
CRectangle::CRectangle (int a, in
{
```

```
rect area: 12
rectb area: 30
```

```
   width = a;   height = b;
}

int main ()
{
 CRectangle rect (3,4);
 CRectangle rectb (5,6);
 cout << "rect area: " << rect.area
 cout  <<  "rectb  area:  "  <<  rect
return 0;
}
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set_values(), and have included instead a constructor that performs a similar action: it initializes the values of x and y with the parameters that are passed to it.

CRectangle rect (3,4);
CRectangle rectb (5,6);

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

ii.Destructor

The *destructor* fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
// example on constructors and destructors
#include <iostream>
using namespace std;
class CRectangle {
int *width, *height;

public:
CRectangle (int,int);
~CRectangle ();
```

```
rect area: 12
rectb area: 30
```

```
int area () {return (*width * *height);}
};
CRectangle::CRectangle (int a, int b) {
width = new int;
height = new int;
*width = a;
*height = b;
}
CRectangle::~CRectangle () {
delete width;
delete height;
}
int main () {
CRectangle rect (3,4), rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
```

iii.Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

```
// overloading class constructors
#include <iostream>
using namespace std;

class CRectangle
{
   int width, height;
public:
   CRectangle ();
   CRectangle (int,int);
   int area (void) {return (width*height);}
};
```

```
rect area: 12
rectb
```

```
CRectangle::CRectangle ()
{
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int b)
{
  width = a;   height = b;
}

int main ()
{
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
return 0;
}
```