# 2.Revision of C++

## Why C++ ?

C++ is the most popular programming language in competitive programming. There are various reasons for it :-

I. It is incredibly fast (or at least fast enough!) .

II. Its rich STL (You will learn about them soon!)

III. Most of the resources focus on C++.

IV. Most competitive programmers use it (Well, this is a cause as well as consequence).

That being said, there are a few areas where Java and Python beats C++. One example of this is the BigInteger class in Java. But on most cases, C++ is good enough.

So, If you are comfortable with C++, here is a quick revision for you. If you are not comfortable with C++, I suggest you to read the text in this chapter. Hopefully, you will be comfortable. But, if you still face problems, follow some tutorial on Internet and then come back.

On the other hand, if you think you are ready to win all wars with some other language, do solve all the problems in this chapter in your favourite language.

## Hello World !

Here is a program to print Hello World on your screen.

```
#include <iostream>
using namespace std;
int main(){
        cout << ”Hello World!\n” ;
        return 0;
}
```

If you had problem understanding why did we use iostream instead of iostream.h or what is “using namespace std”, my only guess is that either you are a C programmer or were using Turbo C as a compiler previously (Well CBSE schools still follow Turbo C for reasons unknown to mankind!). If you are the former, go and grab a tutorial on Internet. It should not take you more than 10 days to shift mentally to C++. If you are the latter, here is a link which may be of your use:- *http://foobar.iiitd.edu.in/turbo.pdf*

So, basically “#include <iostream>” adds a file which suppourts basic standard input and output in C++.
“using namespace std” indicates the compiler to use the namespace “std”. All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name std. So in order to access its functionality we declare with this expression that we will be using these entities.
“;” indicates the end of a statement C++.
main() is the main function in C++ where the execution starts. A program in C++ without the main() function wont’t run. And since this function has a return type “int”, it returns 0 in the end.
“cout” is used for displaying a message or a value.
“<<” is an essential operator (known as insertion operator) to be used after every cout command.
Here the message to be displayed is “Hello World”.
“\n” indicates to print a new line after printing the message.
Every function is a block of code. So it is enclosed within “{“ and “}”.

## Comments

They do nothing except giving the programmer space to insert notes. In C++, you can write a comment in two ways :-

```
I. //my cooment starts here and ends here (line comment)
II. /* my comment starts here
       .....
       and ends here (block comment) */
```

I always appreciate using comments because you never know when you would need a bit of modification in a piece of code to use for another problem.

= = = = = = = = = = = = = = =

**Exercise 2.1**

1. Without seeing the above code, write a program to print your name.
2. I wrote a comment like below, Is it wrong?
   `/ /*Test comment*/ /`
3. Suppose C++ suppourted commenting like this :-
   `{`

```
        My comment starts here
        .......
        My comment ends here
    }
```

What problem would it cause? If we ignore the problem for a moment, what would nested comment mean?

= = = = = = = = = = = = = = =

## Variables

When I studied algebra for the first time, the most weird thing to me was **x.** Off course that is not the case now (hopefully for you as well). So, what is x? Its a variable. What is the dfference between x and Π ? x's value will change over time but the value of Π wont change ever. It is fixed to be 3.14159..(= 22/7 ?). Would you imagine maths if there was no x (or y, z,..) ? Similarily, its difficult to imagine the use of a  programming language which don't support the use of variables. C++ is no exception.

Here is how we declare variables in C++ ,

**Syntax: <data type> <name> ;**

**Example: int x;**

What is data type ? It is the type of value which will be stored in your variable. It may be one of the type – int, float, double, char, short int, long int, long long int, bool.

All these variables are stored in memory. Off course different type of variables take different amount of memory. For example, the size of int data type on most 32-bit systems is 4 bytes. We must remember that 1 byte = 8 bits.

A n-bits data type can store upto $\mathbf{2^n}$ numbers. Thus, the range of a n-bits data type is from $\mathbf{-\,2^{n-1}}$ to $\mathbf{2^{n-1} - 1}$. If the variable is unsigned, the range becomes **0** to $\mathbf{2^n - 1}$.

One other thing that you have to keep in mind is that you can't name the variable the way you want. It has to follow some rules. For instance, you can't name a variable starting with a digit or a special character. Spaces are not allowed. Special characters are not allowed either. Digits may be used only in between or at the end. Underscore (_) and letters can be used freely. Also,the variable name shouldn't be a keyword (reserved words in C++). One example of keyword is **new**. Also keep in mind the fact that variable names in C++ are case-sensitive i.e. auto (another keyword), Auto, aUtO, AUTO are all different.

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

*Box 2.1 : List of all reserved keywords in C++*

So, **int a;** , **unsigned int x = 5;** , **float pi = 3.14 ;** are all valid declarations while **int new;** , **char 5ax;** , **unsigned long long A$3 = 5897456321;** are invalid declarations and shall fetch error form the compiler.

Here is a small program in which we add two numbers ,

```
#include <iostream>
using namespace std;
int main(){
        // Declaring Variables
        int a , b ;
        int c
        // Initialising values to Variables
        a = 5 ;
        b = 3 ;
        c = a + b ;
        cout << c ;     // printing the sum
        return 0;
}
```

One more thing to understand here is that we don’t assign the value to a char type in the same way as in int type i.e. **char X = A;** is wrong. Instead, we assign it this way- **char X = ‘A’;** Please notice here that a character can contain only one letter or a digit or a special character. Oh! Did I say a digit? So how do we assign that - **char X = 6;** or **char X = ‘6’;** The thing is that both are correct. But the values stored in X are different for the two types. The second type stores 6 in X while the first type stores ♠ in X. After teacher says this in the class, the following conversation takes place :-

**Student**: Wait! What?

**Teacher**: What wait ? Go and check this program in your system and see the output.

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

```
int main(){
        char X = ‘A’;
        char Y , Z;
        Y = 6;
        Z = ‘6’;
        cout << X << “ ” << Y << “ ” << Z;     // “ ” for space
        return 0;
}
```

**Student:** No, I get that. But why ?

**Teacher:** Ummm....See every character has a integral code which is known as its ASCII value. 65 just happens to be the ASCII code of A. This code goes from 0 to 255.

**Student:** I guess, There is a problem in this ASCII system. What is the code for 65 ?

**Teacher:** 65 is not a character. You may say 6 is a character or 5 is a character.

**Student:** This seems to be quite confusing. Is there any use of this in future ?

**Teacher:** You will realize its power soon !

Off course, some teachers explain about ASCII values first and then about char. But I prefer to let the students explore things themselves. One last thing. We can also declare a constant the same way. We just need to add the keyword “const” in the begining. Like this – **const double PI = 3.14 ;** . For obvious reason, we can’t change the value of a constant during the execution of the program.

= = = = = = = = = = = = = = = =

**Exercise 2.2**

1. Prove that a data type having the range from - N to N-1 occupies $\log_2 N + 1$ bits.
2. “A n-bits data type can store upto $\mathbf{2^n}$ numbers”. Can you explain why ? (Hint: Binary number system)
3. Which of the following variable names are valid ?
   ABC, $ABC, 5ABC, AABC, wchar_t, :colon, AB$C, AB5C, wEiRd, new
4. Make a table of all data types in C++, its size and its range.
5. “A n-bits data type can store upto $\mathbf{2^n}$ numbers”. “... a character can contain only one letter or a digit or a special character”. Aren’t these two contradicting thenselves? Why or Why not ? (Hint: Read the conversation of the teacher and the student again)
6. Write a program in C++ which calculates the area and circumference of a circle. Use constants and variables.

= = = = = = = = = = = = = = =

## Strings

Variables that can store non-numerical that are longer than 1 character are called strings.That does not mean that you can’t keep the value of a string as “2” or “abc2”. You can. But the thing is you can’t use any operator (“2”+ “2”= “22”) such as +, -, etc on it. The C++ language library provides support for strings through the standard string class. This is not a fundamental data type, but it behaves in a similar way as fundamental data types do in its most basic usage. Since,its not a fundamental data type, we need to include another header file – string.

Both the below initializations are correct:-

```
string S = “I am Mr. @X” ;
string S (“I am Mr. @X”);
```

## Taking Input

What was the most boring thing in the last problem of Exercise: 2.2. Well, I can’t say much about others but for me it was to assign the value of radius manually before the execution. So, here we are going to look at how to take input in C++. The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int radius;
cin >> radius;
```

*cin* can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced.

## The Problem with cin and entry of string

We can use cin for entry of string as well but the problem is that cin extraction stops reading as soon as it finds a blank space. That means if we try to input “My name is Mr. @X”, the value stored is just “My”. A way to fix it is to use **getline()** function with cin. Use it this way.

```
string A, B;
cin >> A;            //not recommended
getline(cin, B);     //recommended
```

There is another interesting thing about strings - the stringstream class defined in the sstream header file. It allows a string-based object to be treated as a stream. This way we can perform extraction/insertion operations from/to strings. For example, if we want to extract an integer from a string we can write:

```
        string S ("897");
        int X;
        stringstream(S) >> X;
```

= = = = = = = = = = = = = = = =

**Exercise 2.3**

1. What we did just above was extraction of an integer from a string. When do you think it would be more useful ? Do you know any other way of converting one data type into other ?
2. Write a program in C++ which takes your name, age, height, weight, color of eye as input and print the following :-

   "Hey, Do you know about <name> ? He is the new cool guy in our school weighing <weight> kgs and is <height> cms tall."

   "Color of his eye is <color>. I am already looking forward to meeting him today in the evening."

   Replace <X> with the entries. Use data types which are suitable.

= = = = = = = = = = = = = = = =

## Operators

Operators are used for operations.

I. Assignment (=) : You have used it earlier in this chapter. But for the sake of formality, this operator assigns a value to a variable. You know how to use it but for the sake of formality (again!), I show it to you once again.

```
                        A = 5 ;
                        A = B;
                        A = B = C = 3;          //new ? That's why I say we always learn
```

The first two are no brainers. The last statement assigns 5 to all the variables A, B, C.

II. Arithmetic (+, -, *, /, %) : The first four are the same as in mathematics. They perform addition, substraction, multiplication and division respectively. The last one is not for finding percentage. It instead finds the remainder i.e. a%b gives the remainder when a is divided by b (≠ 0). A few comments:- In C++, when we apply % operator between a and b and if | a | < | b |, the answer is simply a. If | a | > | b |, the asnwer is a - ( first integer which is divisible by b while moving from a towards b ). Off course, if | a | = | b| ,the answer is 0. For example,

(i) 5 % 10 = 5 (and not -5)

(ii) -5 % -15 = -5 (and not 10 or -10)

(iii) -2 % -3 = -2 (and not 1 or -1)

(iv) 15 % 7 = 1

(v) 7 % -5 = 2 (and not 3 or -3 or 2)

(vi) -7 % 5 = - 2 (and not 3 or -3 or 2)

(vii) -11 % -3 = -2 (and not 2 or -1 or 1)

(viii) -10 % 10 = 0

A bit about division (/) operator :- It doesn't perform the normal division. If you want a / b, it gives you the integral part of the answer i.e. 3 / 2 gives you 1 and not 1.5. This division is known as integral division. To get 1.5, use 3 / 2.0 or 3.0 / 2 or 3.0 / 2.0.

III. Compound Assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=) : It assigns the value to a variable after modifying the same variable by performing any operation. For example, a += 5 is same as a = a + 5. This makes our code a bit short. You would often find good competitive programers (or any programmer for that matter) using compound operators. You should be able to conceive yourself "why?".

IV. Increment , Decrement (++, --) : They increase or decrease by one the value stored in the variable. There are two ways to use it- Prefix(++a) and Postfix(a++). Prefix increments/decrements and then proceeds while Postfix first executes the statement and then increments/decrements.

V. Relational and Equality (==, !=, >=, <=, >, <) : Suppose, we want to compare two values,then we use relational operator. The result of which is a boolean value- true or false. Do not confuse == with =. The former is a relational operator and the latter an assignment operator. So what? = assigns a value to a variable while == checks if two values are equal. In Pascal, we used := instead of = and so it was a bit clearer but anyways remember this. Also != stands for 'not equal to'. The evaluation of a statement to true or false is very basic and is taught in secondary/higher classes. Don't woory if you never heard of such a chapter(truth table). Its pretty much common sense. Solve the problems in the exercise and you are done.

VI. Logical (!, &&, || ) : If you ever studied the chapter truth tables, you can just skip by knowing that ! stands for NOT(~), && stands for AND and || stands for OR. Otherwise,bear with me.

!(TRUE) = FALSE

!(FALSE) = TRUE

TRUE && TRUE = TRUE

FALSE && TRUE = FALSE

FALSE && FALSE = FALSE

TRUE || TRUE = TRUE

TRUE || FALSE = TRUE

FALSE || FALSE = FALSE

Common sense. Right ? No ? Understand it.

You might not be able to figure out why is it even important. You will get to understand that once you study about if else statement. Or wait! Here is something for you.

VII. Conditional (?) : The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false.

**Syntax: condition ? Statement1 : statement2 ;**

**Example: (a==5) ? cout<<"YES" : cout << "NO";**

Here is a program which takes two integers as input and prints the minimum among them.

```
#include <iostream>
using namespace std;
int main(){
        int a, b, mini;
        cin >> a >> b;
        mini = (a<b) ? a : b;
        cout << mini;
        return 0;
}
```

So, how does this work? Actually we have declared the variable mini to store the minimum among a and b. If a < b, that means the condition is true and hence the first statement is executed i.e. the value of mini becomes a. In case a > b, that means the condition is false and hence the second statement is executed i.e. the value of mini becomes b. If a==b, that again means the condition is false and hence the second statement is executed i.e.the value of mini becomes b. Please notice that there is nothing wrong in it. If two nos. are equal, the minimum among them is either of them.

VIII. Explicit type casting : Remember the stringstream class. One use of it was to convert string to int. But how do we go about converting a float to a int? First convert it into string and then convert to int. That is too much! Here is a way – typecasting operator. What does it do? It converts one datum to another. How? Let’s see

```
int i, j ;
float PI = 3.14 ;
i = int (PI) ;      // i = 3 What did you think? Int storing a float? Huh!
J = (int) PI ;      //just another way
```

There are few other operators as well – the comma operator, the bitwise operator. Bitwise operator will be discussed in detail in some other module.

= = = = = = = = = = = = = = = =

**Exercise 2.4**

1. Write a program in C++ which takes three numbers as input and prints the minimum among them. (Hint: How about *nested* conditions ?)
2. Write a program in C++ which takes two integers as input and prints the greater among them. The only problem is that you can’t use the ‘<’ or ‘>’ operator . Also given that the numbers lie in the range [-1000,1000] (Don’t focus on this too much) and one of them is -ve while the other is +ve. (Hint: A bit of  maths? Simple though.)
3. Write a program in C++ which takes two integers as input and prints the greater among them. The only problem is that you can’t use the ‘<’ or ‘>’ operator. Also given that the numbers lie in the range [-1000,1000] (Don’t focus on this too much) and either both of them are -ve or both are +ve.. (Hint: A bit of maths? Simple though.)
4. Write the boolean value corresponding to the statements:-

   (i) $5 > 5$

   (ii) $!(5 > 5)$

   (iii) $\sin \Pi == 0 \;\&\&\; \cos \Pi == 1$

   (iv) $\sin \Pi == 0 \;||\; \cos \Pi == 1$

   (v) $(x^2 + 1)_{min} = 1 \;\&\&\; (u^2 + v^2 + w^2 >= uv + vw + wu \;||\; (-3 < 5 \;\&\&\; (x + 1/x)_{min} = 2) )$

   (vi) $– 8 \% 7 != 1 \;\&\&\; !( |x| + |y| = -2$ has no solution)
5. What is the difference between a++, c = a+1 and a = a + 1 ?

= = = = = = = = = = = = = = = =

## <u>if, else if, else</u>

Remember the conditional operator? if, else if, else works in a similar fashion. *if* keyword would execute a given block of statements if and only if the condition is true. *else* keyword word would execute a given block of statements if the conditions turn out to be false. Additionaly, *else if* can be used in between to check for other conditons.

```
Syntax: if (condition1) {
                statement(s)
                ....
        }
        else if (condition2) {
                statements(s)
                .....
        }
        else {
                statements(s)
                ......
        }
```

Few things to note:-

(i) You can use *if* even without using *else if* and *else*. But to use *else if* or *else*, you must first use *if*.

(ii) If the condition for *if* is true, then statements in the if block would be executed and then the program execution would directly move on the line after *else* i.e. the remaining condition wont be checked. If the condition for *if* is false, the program execution moves on to next condition until it finds a condition which evaluates to true. If all the conditions are false, statements in the block *else* will be executed.

(iii) You can write complex conditions using logical operators(!, && and ||) . (Do you realize the importance now?). They evaluate to TRUE or FALSE and hence are quite handy.

(iv) If there is only one statement, there is no compulsion of using parantheses.

Here is a simple program illustrating this. It takes an integer as input and checks if it is positive, negative or zero.

```
#include <iostream>
using namespace std;
int main(){
        int x;
```

```
                    cin >> x;
                    if (x>0)
                              cout<<x<< " is positive";
                    else if (x<0)
                              cout<<x<< " is negative";
                    else
                              cout<<x<< " is neither positive nor negative";
                    return 0;
          }
```

(v) The best thing is that you can use nested *if*s i.e if inside if.

Here is a program which illustrates this. It takes a year as input and checks if it is a leap year or not.

First of all, you need to know what is a leap year? If your teacher taught you that any year which is divisible by 4 is a leap year, then you have been fooled all your life. For example, 2100 is divisble by 4 but is not a leap year. So what is a leap year? A non-century year (not divisible by 100) which is divisible by 4 is a leap year and the one which is not divisible by 4 is not a leap year. For example, 2010 is not a century year and is not divisible by 4. So it is not a leap year whereas, 2016 which is again not a century year but is divisible by 4 and hence it is leap year. What about century years? It is definitely going to be divisible by 4 (Why?). So, we again check if the year is divisible by 400. If it is, it is a leap year otherwise it is not. For example, 2000 and 2100 are both century years. The former is divisible by 400 while the latter is not. So, 2000 is a leap year while 2100 is not.

Here is the program which is just the translation of the above paragraph into C++ : -

```
          #include <iostream>
          using namespace std;
          int main(){
                    int year;
                    cin >> year;
                    if ( year % 100 != 0 ) {
                              if ( year % 4 == 0)
                                        cout << "Leap year";
                              else
                                        cout << "Not a leap year";
                    }
                    else if ( year % 400 == 0)
                            cout << “Leap year”;
                    else
                            cout << “Not a leap year”;
                    return 0;
            }
```

Pretty complex? Do you know why is it so complex? To know that, you first need to know what is a leap year in terms of no. of days taken to complete one revolution of earth arond the sun. Go and Google it (Why do you expect a computer science book to explain you mathematics and then even physics!)

= = = = = = = = = = = = = = = =

**Exercise 2.5**

1. The program for checking if a year is leap year or not seems to be quite lengthy. Shorten it a bit with the help of conditional operator.
2. Solve Q2 and Q3 of Exercise 2.4.
3. Suppose that a planet in some other galaxy completes a revolution around its star in 365 days 3 hours and 58 minutes. The Aliens there got to know that you just googled out the real meaning of leap year. They kidnap you and tell you to write a program to check if the current year is leap year or not for them. (Hint: Didn’t you google?)
4. Let a polynomial p(x) be $ax^2 + bx + c$. Write a program to check if for given values of a. b and c, the roots are imaginary or not. If they are not, check if the roots are equal or not. If the roots are imaginary, print “imaginary roots”. If the roots are equal, print “real and equal roots”. If they are not equal, print “real roots”.
5. Write a program which accepts the percentage of marks of a student and assigns the grades as follows:-

   A : 75 <= Input <=100

   B : 50 <= Input < 75

   C : 33 <= Input < 50

   D : 0 <= Input < 33

   Solve with and without using logical operators (even if it feels unnecessary).

= = = = = = = = = = = = = = = =

## <u>while</u>

What would you do if I say you to print your name twice on the screen. Simple stuff! Write cout << “Mr @X\n”; cout << “Mr. @X\n”. But what if I tell you to do it 100 times. You would say – copy and paste 100 times. Well then, a million times and finally you would accept defeat (No?). So here I am going to tell you about one of the most powerful techniques in the computing world – loops. Without understanding loops, don’t ever dream to become a competitive programmer (or any programmer for that matter).

```
Syntax: while (condition) {
            statement(s)
            .....
        }
```

*while* statement executes a given block of code till the condition evaluates to TRUE. Can you now think of how would you display your name a million times on the screen ?

Here we do a few examples. The first one simply prints numbers from 1 to 100.

```
#include <iostream>
using namespace std;
int main(){
        int i = 1;
        while ( i <= 100 ){
                cout << i << ”\n”;
                i++;
        }
        return 0;
}
```

This is how it works :-

(i) *i* is initialized to 1.

(ii) Is 1 <= 100 – YES, i is printed.

(iii) *i* is increased by 1 i.e. i = 2 now.

(iv) Is 2 <= 100 – YES, i is printed.

(v) *i* is increased by 1 i.e i = 3 now.

...

...

(what?) Is 100 <= 100 – YES, *i* is printed.

(?) *i* is increased by 1 i.e. i = 101 now.

(?) Is 101 <= 100 – NO, get out of the loop.

Few comments :-

(i) There is no need of parantheses if the statement is one line.

(ii) *i++* is very important. You see if you don't increase i by 1, the condition will always evaluate to TRUE and your screen will be filled with 1. But to more worry, the execution of program will never stop. This is known as *infinite loop*.

(iii) In place of *i++*, you could have written, *i += 2* and that would print all the odd nos. between 1 and 100. What I want to say is that Its not *i++* which is important. The important thing is *updation*.

(iv) You must also *initialize i* before the loop starts.

(v) You can have complex conditions using logical opertaors.

Here is a program which takes an integer as input and prints its multiplication table :-

```
#include <iostream>
using namespace std;
int main() {
        int i , n ;
        cin >> n ;
        i = 1 ;
        while (i <= 10) {
                cout << n << " X " << i << " = " << n*i << endl ;
                i ++ ;
        }
        return 0;
}
```

Make sure you type this in your computer and see the output. Also understand each and every thing. If you don't get any thing, look at the previous program and its explanation.

## do while

*do while* is exactly the same as *while* except for the fact that it checks for the condition after execution guaranteeing at least one execution even if the condition never evaluates to TRUE.

```
Syntax: do {
                statement(s)
```

**.....**

**} while (condition) ;**

Here is a program which prints “Go”  till you press 0.

```
#include <iosttream>
using namespace std;
int main() {
        int i ;
        do {
                cout << “Go\n” ;
                cin >> i ;
        } while ( i != 0) ;
        return 0 ;
}
```

Please notice here that it will print Go once even if you press 0 in the start. Also notice the semicolon at the end of condition. Don’t forget to add it.

## <u>for</u>

It is also same as *while* except for the fact that it *initializes*, *checks and udates* at the same place.

```
Syntax: for ( initialize ; condition ; update ) {
                statement(s)
                .....
        }
```

Here is a program which prints all the numbers from 1 to 1000.

```
#include <iostream>
using namespace std;
int main(){
        for (int i = 1; i <= 1000; i++){
                cout << i <<”\n”;
        }
        return 0;
}
```

Few comments :-

(i) The order is :- initialization -> checking -> execution -> updation -> checking -> execution -> updation and so on.
(ii) You can initialize more than 1 variables by using comma (,) operator.
(iii) You can have complex conditions with the help of logical operators.
(iv) You can use *for* like *while*. Just leave the places where you are inittializing and updating (Don't omit the semicolon though) , initialize somewhere above and update somewher inside the block.

## break and continue

Using *break* we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end.
The *continue* statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached.
Now we are going to write a program which cheks if a number is prime or not. What is a prime number? The number whose factors are 1 and itself only is known as prime number. So what do we do? We iterate from 2 to n-1 and check if any of the number divides n, then the number is not prime.

```
#include <iostream>
using namespace std;
int main(){
        int n , i ;
        bool flag = 0 ;         //new data type?
        cin >> n ;
        for ( i = 2; i < n; i++ ){
                if (n%i == 0){
                        flag = 1;
                        break;
                }
        }
        if( flag == 1)
                cout << “Not prime”;
        else
                cout << “Prime”;
        return 0;
}
```

The *flag* variable becomes 1 if for any i, i divides n and the loop breaks. In the end we check the value of *flag* and print accordingly.

## A nice problem

With all the things you learnt about loops try to solve the following problem :-

*Given a number n, print the sum of its digits.*

If somebody among you laughed after seing this problem titled as A nice problem, there is a great chance that he/she has already solved that problem and I guarantee that the first time they encountered this problem, they would have termed it nice as well.

So how do we go about solving this problem ? Think .... Think! Solve it if number of digit is 1. Oh that was quite easy. What if number of digits is 2? Let N = ab where a and b are digits and N is not the product of a and b. Notice that b = N % 10 and a = N / 10 (integral division). Solve if N = abc. Again, c = N % 10. Now reduce N to N / 10. Again b = N % 10. Reduce N to N / 10 again. Now, a = N / 10. Now, we gotta stop. Tht’s it! We solved the problem. Continuously, find the last digit and reduce N to N / 10. When to stop ? N = 0.

```
#include <iostream>
using namespace std;
int main(){
        int N , sum = 0 ;
        cin >> N ;
        while (N != 0){
                sum += N % 10 ;
                N /= 10 ;
        }
        cout << sum ;
        return 0;
}
```

There are many variations to this problem. Try reversing the number.

= = = = = = = = = = = = = = =

**Exercise 2.6**

1. Write a program to print all the numbers between 1 and 1000 which are divisible by 3.
2. Write a program which takes a positive integer as its input and find the number of factors it has. Can you use this to check if a numer is prime number ?

3. Prove that even if we iterated from 2 to √n instead of 2 to n-1, our prime no. program would have been correct. (Hint : Factors come in pair )
4. Using sqrt () function from cmath.h file, modify the prime number program according to the previous question.
5*. Print all the prime numbers between 1 and 100.
6. A palindromic number is a number which reads same from right and left. Write a program which checks if a number is palindrome or not.
7. A fibonacci series is defined as t (n) = t (n-1) + t (n-2) where t (0) = 0 and t (1) = 1. The first few terms of fibonacci series is 0,1,1,2,3,5,8,13,......Write a program which prints the first n fibonacci numbers.

= = = = = = = = = = = = = = =

## Nested Loops

Loops inside loops is called nested loops. Did you solve the starred problem of previous exercise? It uses the concept of nested looping. First think about how would you check if a number is prime or not? You did it in the previous section. Now think about doing it for all numbers from 1 to 100. Iterate from i = 2 to i = 100 and for each i check if it is prime or not. Here is the program :-

```
#include <iostream>
using namespace std;
int main(){
        int i , j ;
        bool flag ;
        for (i=2 ; i <= 100 ; i++){
                flag = 0;                       //we have to reinitialize it always
                for (j=2 ; j < i ; j++){
                        if(i%j == 0){
                                flag = 1;       //i is composite
                                break;          //break of the j-loop
                        }
                }
                if (!flag)
                        cout << i ;
        }
        return 0;
    }
```

Nested loops are of great use. To illustrate this, let's try to print the following pattern :-

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

This is interesting. How do we print this? We would need two loops for sure. Why? One for controlling till where to go and other for controlling what to print. We iterate from i = 1 to 5. For all i, we iterate from j = 1 to i and print j. Once, we finish with the j-loop, we go to a new line. Let's code it out.

```
#include <iostream>
using namespace std;
int main(){
        int i , j ;
        for ( i = 1 ; i <= 5 ; i++ ){
                for ( j = 1 ; j <= i ; j++){
                        cout << j << " " ;
                }
                cout << "\n";
        }
        return 0 ;
}
```

Please note that we are not limited to use just one loop inside the other.

= = = = = = = = = = = = = = = =

**Exercise 2.7**

1. Print the following pattern :-

```
5
5 4
5 4 3
5 4 3 2
5 4 3 2 1
```

2. Print the following pattern :-

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

3. Print the following pattern (Hint: Look '*around*' for help but dont use Internet):-

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

4. Print the followng pattern (Hint: Remember the conversation?):-

```
A
B C
D E F
G H I J
K L M N O
```

5. Factorial of a number N, N! is defined as 1 x 2 x 3 x ...... x N. We define a special number as a number whose sum of digits of factorial is 1.5 times the number. For example, 4! = 1 x 2 x 3 x 4 = 24 and 2 + 4 = 6 = 1.5 x 4. Therefore, 4 is a special number. Print all the the special numbers between 1 and 19.

6. Print the following pattern :-

```
        *
      *   *
    *   *   *
  *   *   *   *
*   *   *   *   *
```

= = = = = = = = = = = = = = = =

## Functions

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(). Suppose that in a program, we are doing the same thing many times , say finding the minimum of two numbers. Then it would be nice if we could write a min() function somewhere else and call it again and again from the main() function. Thanks to C++ we can do this!

**Syntax: return-data-type name ( parameter1, parameter 2,....){**

```
                    statement(s)
                    .......
        }
```

What is *return data type*? Well, consder the min() function which would find the minimum of two integers, what would be its answer? An integer. That's what *return data type* is. Please note that we can use *void* to return nothing i.e. if we just wished to print the minimum from the min() function itslef, we could have used *void*. What is *name*? It is the name which you would like to assign to your function. In min() case, the name is min. Please note that you can name a function using the same rule as for naming a variable. What are *parameters*? These are the variables on which the function works. Here, the two integers of which we are finding the minimum are parameters. Also note that if there is no *void* return type, then you have to return something from the function of the specified return data type.

Here is a program which shows different ways of using the function min().

```
#include <iostream>
using namespace std;
int min(int a,int b){
        int c = (a>b) ? b : a;
        return c;
}
void print_min(int a, int b){
        int c = (a>b) ? b : a;
        cout << "Minimum is: " << c;
}
int main(){
        int a , b;
        cin >> a >> b ;
        cout << "The minimum is: " << min(a,b); //this is the way of calling a
                                                  function
        print_min(a,b);
        return 0;
}
```

Please note that you can’t print *c* directly form main() because the scope of a variable is limited to its parent block. Also, note that we can use a function even without a parameter i.e. if you wished you could have done all the entry and printing inside the main() function and call it just in the begining of main().

Here is a program: -

```
#include <iostream>
using namespace std;
void min(){
        int a , b , c ;
        cin >> a >> b;
        c = (a > b) ? b : a ;
        cout << “The minimum is: ” << c;
}
int main(){
        min();
        return 0;
}
```

## Scope of a variable

A scope is a region of the program and in general there are three places, where variables can be declared:

1. Inside a function or a block which is called local variable.
2. In the definition of function parameters which is called formal parameters.
3. Outside of all functions which is called global variable.

A local variable can be accessed inside the same function or the same block. A foramal parameter can also be used in the same function while a global variable can be used anywhere.

Remember the golden rule : ***“The scope of a variable is its parent block”***

Consider that we have a variable which is declared inside a block and also declared globally. We use “::” at the begining of the variable name to access the global variable then.

## Call by value and Call by reference

Try this program and note the output :-

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

```
void change(int a){
        cout << "Initially in the function: "  << a;
        a++;
        cout << "Finally in the function: " << a;
}
int main(){
        int x = 5;
        cout << "Initially in the main: " << x;
        change(x);
        cout << "Finally in the main: " << x;
        return 0;
}
```

What do you notice? *a++* in the change() function doesn't effect the real value of x.

Now try this :-

```
#include <iostream>
using namespace std;
void change(int &a){                    //only difference
        cout << "Initially in the function: "  << a;
        a++;
        cout << "Finally in the function: " << a;
}
int main(){
        int x = 5;
        cout << "Initially in the main: " << x;
        change(x);
        cout << "Finally in the main: " << x;
        return 0;
}
```

Now, the value of x gets changed. Previously, we were passing the argument by value. A duplicate of it was created and used inside the change() function and hence the real value didn't get affected. But now we pssed the variable itself and hence the change gets reflected.

= = = = = = = = = = = = = = = =

**Exercise 2.8**

1. Write a program for a simple calculator which performs addition, substraction, multiplication and division. Use functions.
2. A number is called a perfect number if sum of all its divisor is twice the number. For example, sum of divisors of 6 = 1 + 2 + 3 + 6 = 2 * 6 = 12. Write a function which takes a number as a parameter and returns 1 if it is perfect and 0 if it is not.

3*. Find the $n^{th}$ Fibonacci number using functions.

= = = = = = = = = = = = = = =

## **Recursion**

When a function calls itself, it is called recursion. Let’s write a program to calcualte the factorial of a number using recursion. We know that n! = n x (n-1)! We would be exploiting this recursive definition of factorial. Look at the program first and then we will discuss some important things about recursion.

```
#include <iostream>
using namespace std;
int fact(n){
        if(n==0)
                return 1;
        else
                return n*fact(n-1);
}
int main(){
        cout << fact(5);
        return 0;
}
```

Here is the recursion tree for the program : -

```
  /\
   |    120
fact (5)
   |    24
  \/
fact (4)
   |    6
  \/
```

$$\begin{tikzcd}
\text{fact (3)} \arrow[d, "\mathbf{2}"] \\
\text{fact (2)} \arrow[d, "\mathbf{1}"] \\
\text{fact (1)} \arrow[d, "\mathbf{1}"] \\
\text{fact (0)}
\end{tikzcd}$$

`

Here, the number written on the side of the arrow represents the number which has been return by the next call (here fact(n-1)).

Notice how cleverly we terminated the recursion whaen n came down to 0. If we didn’t do so, it would have led into infinite recursive calls. That brings us up to what are the necessary things for recursion :-

1. Recursive definition (Here fact(n) = n * fact(n-1) )
2. Base case/s (Here fact(0) = 1)

Let us use these facts to solve the 3rd question of Exercise 2.8 i.e. to find the nth fibonacci number. We just need two things: - Recursive definition and base case/s.

$$\text{Fib}(n) = \begin{cases} 0 \; ; n = 0 \\ 1 \; ; n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) \; ; n > 1 \end{cases}$$

That’s it. Here is the code:-

```cpp
#include <iostream>
using namespace std;
int fib(n){
        if ( n <= 1)
                return n;
        else
                return fib (n-1) + fib (n-2);
}
int main(){
        cout << fib (3);
```

```
                    return 0;
            }
```

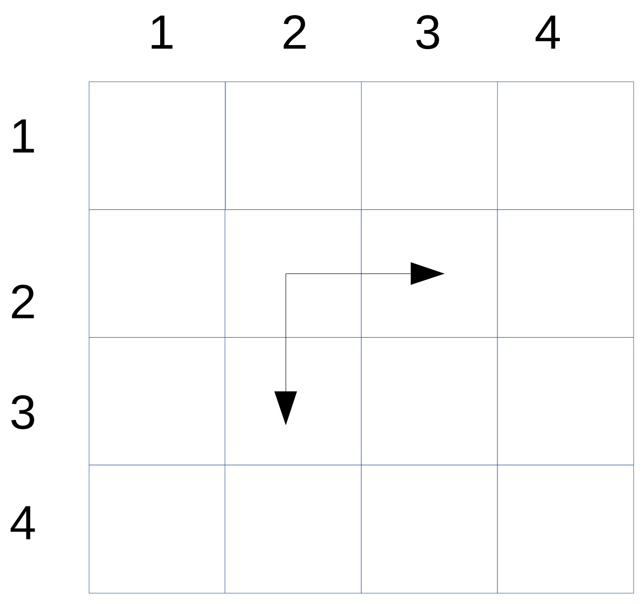
Here is the recursion tree:-

```
                         /\
                          | 2
                       fib (3)
                         / \
                      1 /     \ 1
                      \ /       \ /
                   fib (2)    fib (1)
                      / \
                   1 /     \ 0
                   \ /       \ /
                fib (1)    fib (0)
```

Recursion seems like magic to many people because you don't have to do much. But it is important to understand how it actually works.

Now, we solve yet another interesting problem.

*Given a n x n grid, find the number of ways to reach (n,n) from (1,1) if you can move one step down or one step to the right.*



Notice that the number of ways of reaching (4,4) here is the no. of ways of reaching (3,4) plus the number of ways of reaching (4,3). Generalising this, the number of ways of reaching (i,j) is the number of ways of reaching (i-1,j) plus the number of ways of reaching (i,j-1). What are the base cases then? When we are at (1,1) that means we somehow got a path and i.e. the number of ways to reach (1,1) from (1,1) is 1. If we cross the boundary, that means we can never reach there from (1,1) and hence the answer is zero.

$$\text{numpath}(i,j) = \begin{cases} 0 \;;\; i < 1 \text{ or } j < 1 \\ 1;\; i = 1 \text{ and } j = 1 \\ \text{numpath}(i-1,j) + \text{numpath}(i,j-1) \;;\; \text{otherwise} \end{cases}$$

Code this and see the magic!

= = = = = = = = = = = = = = = =

**Exercise 2.9**

1. Use recursion to find the sum of all numbers from 1 to n.

2*. Draw the recursion tree of fib(5). What do you see? How many times fib(3) is called? How many times fib(2) is called? Can you suggest a method to get over this problem?

3. Find the sum of digits of a number using recursion.

4. Catalan numbers satisfy the following formula :-

   catalan (n) = ∑ [catalan(i) * catalan (n-i-1)] where i ranges from 0 to n-1.

   Find the $n^{th}$ catalan number. (Hint: You can use loops inside recursion)

5. Given coordinates of a source point (x1,y1), determine if it is possible to reach the destination point (x2,y2) from any point (x,y). There are only two types of valid movements:

   (1) (x,x+y)

   (2) (x+y,y)

   Note: All co-ordinates are positive.

= = = = = = = = = = = = = = = =

## Arrays

Consider a situation when you have to get the names of 10 people. How do you do it? Simple stuff! Declare 10 string variables and take input. Okay! How about 1000 people. You would surely not want to waste whole of your day declaring 1000 variables with unique name and then writing 1000 cin statements. Wait! What if you could have used loops to get input of 1000 vaiables? The problem is that - You can’t ! Because loops run for integer values (or real numbers in general) and variable names are not integers. And here comes Array to your rescue. Arrays are continuous block of memory which stores elements of same date type and can be referenced using an index.

This means that, for example, we can store 5 values of type *int* in an array without having to declare 5 different variables, each one with a different identifier. For example, an array to contain 5 integer values of type *int* called X could be represented like this:

**X [____][____][____][____][____]**

**0 1 2 3 4**

How to declare it? Here is the syntax :-

**Syntax: data-type name [no. of elements];**

**Example: int X [5] ;**

Please note the no. of elements has to be a constant i.e. the amount of memory to be reserved has to be declared before the execution of the program starts.

We can access the $i^{th}$ element of the array by using X[i].

Here is a simple program which takes input of 5 integers and prints it :-

```
#include <iostream>
using namespace std;
int main(){
    int A[5] ;
    int i ;
    for (i=0 ; i < 5 ; i++){
        cin >> A[i];
    }
    for (i=0; i < 5 ;i++){
        cout << A[i] << “\n” ;
     }
    return 0;
}
```

Also, you can intialize an array using blocks. This way:-

```
int A[] = { 1, 2, 3120, 428, 501 };
```

Important thing to notice here is that indexing starts from 0 and not 1.

## new and delete Keyword

In the previous section, I told you that we can’t declare an array of variable size in the same way as normal array i.e. this code snippet is completely wrong:-

```
        int n;
        cin >> n;
        char X [n] ;
```

There is a way. We can use the *new* keyword to declare an array of variable size. Note that variable size doesn’t mean that the size of array is goping to change during the program. It means that you can declare an array of your size.

It is difficult to explain it without knowing about pointers and since we are not going to cover pointers in this module, it would be better to google about pointers. Here, we are going to just put forward the code.

```
        #include <iostream>
        using namespace std;
        int main(){
                int *A;  //This is how we declare a pointer
                int n,i;
                cin >> n;
                A = new int [n];        //You can replace int with char, float,etc.
                //Take input
                for(i=0;i<n;i++){
                        cin >> A[i];
                }
                //Print them
                for(i=0;i<n;i++){
                        cout << A[i] << “ ”;
                }
                delete A[n];            //To free the memory
                return 0;
        }
```

Don’t get disheartened if you don’t understand this mess (really?). In most of the problems on Online Judges, we have the constraints for ‘n’ known beforehand and so we can declare the array of maximum size and use the amount which we need. Also, once we know about *vectors*, there would be no need of these types of arrays.

The primary use of arrays is to store data of a given kind.

## Searching and Sorting

Now, we would be writing a function which searches for an element in the array. The parameters would be the array itself, its size and the value to be searched and we return the index number of the first place where the value is found. If the given value is not present in the array, we return -1. Ho do we solve this? Its simple! Iterate from zero to size-1 and check if the value at the given index matches with the value to be searched. Here is the code:-

```
#include <iostream>
using namespace std;
int linear_search(int X[], int size, int key){
        for(int i=0 ; i<size ; i++){
                if(X[i] == key)
                        return i;
        }
        return -1;
}
int main(){
        int *A;
        int i,n,x,idx;
        cin >> n;                   //size of the array
        A = new int [n];
        for(i=0 ; i<n ; i++){
                cin>>A[i];
        }
        cout << "What to search? ";
        cin >> x;
        idx = linear_search ( A , n , x );   // don't send like this – F(A[],y)
        if(idx==-1){
                cout<< "Not found!";
        }
        else{
                cout << "Found at: " << idx;
        }
        delete A[n];
        return 0;
        }
```

If you want to get away from pointer mess, I suggest you to declare the array of size 1000 (Assuming you don’t need more) and then use the amount necessary. Hopefully, you learnt how to pass an array as parameter. One thing to note here is that the changes dont to the array elements in the function is reflecteed inide the *main()* function. So no call by value and call by reference here.

Its time for yet another interesting problem :-

*We have an array A[] of size N. We would like to print the array in ascending order. For example, if A [] = { 1, 5, 3, -8, 101, 4 }, we would like to make it A[] = { -8, 1, 3, 4, 5, 101}*

Think for a minute how can we do this. Think more! Why do you want to jump into solution straightaway? Okay, No clue? Here is a hint – At what index will the minimum element be in the array? What about second smallest? What about largest?

The smallest element will be at index 0, 2nd lowest at index 1 and so on...So what are we going to do? Simple! We are going to find kth smallest element and put it at index k-1 for all k<=N.

Suppose at some point of time you have to find the kth smallest element, how do you do this? This means we have already found out and put k-1 elements at its position from index 0 to k-2. So, we find the minimum of all elements from index k-1 to N-1 and swap it with the element currently at index k-1. It is clear that we will need two loops – one for k and the other for finding minimum in the range k–1 to N-1. If you found it tough to understand, read the paragraph carefully again and then look at the code below.

```
#include <iostream>
using namespace std;
void selection_sort(int A[], int size){
        int k , j , tmp , min_idx;
        for(k=0;k<size;k++){
                min_idx = k;   //Initialize that the minimum element is at k
                for(j=k+1;j<N;j++){
                        if(A[min_idx] > A[j]){
                                min_idx = j; //Found new minimum
                        }
                }
                //Swap the element at k with that at min_idx
                tmp = A[k];
                A[k] = A[min_idx];
                A[min_idx] = tmp;
        }
}
int main(){
        int A[1000] ;   //no pointer stuff
        int i , N ;
        cin >> N;
        for(i=0;i<N;i++){
                cin>>A[i];
        }
        selection_sort(A,N);
        cout << "Array in ascending order : " ;
        for(i=0;i<N;i++){
                cout << A[i] << " ";
        }
        return 0;
}
```

Make sure you understand each and everything before moving ahead. Try to justify why didn't we iterate form 0 to size-1 in the second loop, why didn't we explicitly keep the value of the k$^{th}$ smallest element but instead kept its index and why did we initialize min_idx to k in the starting of the second loop. What errors (if) each of them would have caused?

= = = = = = = = = = = = = = = =

**Exercise 2.10**

1. Write a function which takes an array and its size as parameter and returns the largest of all elements in the array.
2. Write a function which takes an array and its size as parameter and return the index at which the largest element is present.
3. Prefix Sum at index i is defined as the sum of all elements form index 0 to that index. Given an array of size N, print the prefix sum at all i for 0<=i<N. (Hint: Did you use two loops? - Oh! pre[0] =A [0] and pre[i] = pre[i-1]+A[i])
4. Given an integer array A[] of size N, calculate its mean, median and mode. (Hint: Does sorting help?)

5. Suppose the memory address of A[5] is 928. Find the memory address of A[27] if A[] is an integer array.

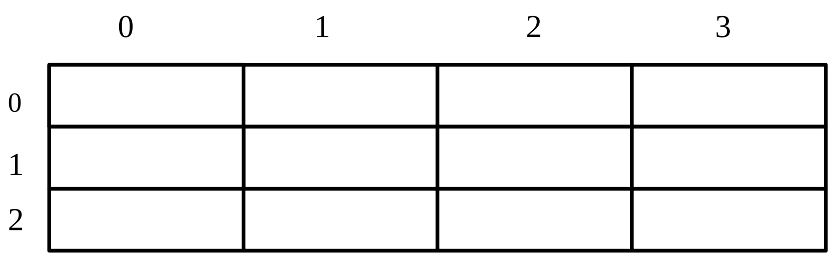
6. Without looking at the last program, sort and print an array in descending order.

= = = = = = = = = = = = = = = =

## Multi-dimensional Arrays

Multidimensional arrays can be understood as 'tables' or 'array of arrays'. When do you need it? The simplest example can be of say storing the marks of 5 subjects of 100 students. Offcourse, you can either create 5 arrays of size 100 - each for a subject or 100 arrays of size 5 - each for a student. That doesn't sound pretty or does it? For these purposes, we use what we call a multidimensional array. Here, we can simply declare a bidimensional array (table) of size 100 x 5.

Here is what bidimensional arrays look like :-



This can be extended to more number of dimensions if needed.

**Syntax: data_type name [row_size][column_size];**

**Example: int X [100][5];**

We can access the elements in a similar way as in single-dimensional array. But one thing to notice here is that there is no single index using which we can refer to an element. We need two of them in 2-dimensional arrays. Again the indexing of row and column starts from 0 and not 1. Here is an example program in which we simple take entry and output the elements. We need tow loops. One for controlling the row index and other for column index.

```
#include <iostream>
using namespace std;
int main(){
        int A[100][100];  //Assuming maximum size to be 100 x 100
        int m , n , i , j;
        cout << "Enter no. of rows: ";
        cin >> m;
        cout << “Enter no. of columns: ”;
        cin >> n;
        // Take entry
        for(i=0 ; i<m ; i++){
                for(j=0 ; j<n ; j++){
                        cin >> A[i][j];
                }
        }
        //Output them
        for(i=0; i < m ; i++){
                for(j=0 ; j < n ; j++){
                        cout << A[i][j] << “ ”;
                }
                cout << “\n”;
        }
        return 0;
}
```

Similarily, we can do it for 3-dimensional arrays using 3 instead of 2 loops. However, for most of the problems in this book, we wont need more than 2-dimensional arrays. To demonstrate how to pass a multidimensional array as a parameter, we solve another problem. We have a 2-d array and we need to find the sum of the elements of the two diagonals. First thing to note is that we cannot do it for a rectangular table. Here is the code.

```
#include <iostream>
using namespace std;
void diag_sum(int A[][100], int row, int col){
        if(row!=col){
                cout << “Diagonal sum not possible”;
        }
        else{
                //First diagonal – upper left to lower right
                int i = 0 ;
                int sum1 = 0 ;
                for ( i = 0 ; i < row ; i++){
                        sum1 += A[i][i] ;
                    }
                    //Second diagonal – upper right to lower left
                    int i = 0 ;
                    int sum2 = 0 ;
                    for(i=0 ; i<row ; i++){
                        sum2 += A[i][col-i-1] ;
                    }
                    cout << sum1 << “ ” << sum2 << endl;
                }
        }
        int main(){
                int A[100][100] ;
                int m , n , i , j ;
                cin >> m >> n;
                for(i=0 ; i < m ; i++){
                    for(j=0 ; j < n ; j++){
                        cin >> A[i][j];
                    }
                }
                diag_sum(A,m,n);      // Not like this – F(A[][],m,n)
                return 0;
         }
```

Notice how we pass the array. We must give the column size of the array while passing. In general, we have to give the size of all the dimensions except the first one. Please note that we may even give the size of the first dimension as well.

= = = = = = = = = = = = = = =

**Exercise 2.11**

1. Can you take input in a 2-d array using just one loop? How many loops would you need for taking input in a 3-d array? (Hint: Remember the Euclid’s Division Lemma?)
2. Given some points in a cartesian 3-d coordinate system. Sort them according to x-cordinate. Break the tie by y-coordinate and finally by z-coordinate.

3. You have two matrices of order M x N. Perform matrix addition on them and store the result in the third matrix.
4. There is an integer matrix A of order 25 x 101. Suppose the memory address of A[12][92] is 1234. Find the memory address of A[7][61].

= = = = = = = = = = = = = = =

## Structure

Let's get back to one of our former examples. We had to store the marks of 5 subjects of 100 students. Suppose, in addition, we had to store the name of each student as well. Now, you know that arrays store like data i.e. we can't keep a string data along with integer data. Structures are useful for this purpose. Here is how we declare a structure.

**Syntax: *struct* name {**

**data_type_of_1st_data name;**

**data_type_of_2nd_data name;**

. . . .

. . . .

**} ;**

**Example: *struct* student{**

***int* marks1, marks2, marks3, marks4, marks5;**

**string name;**

**};**

After this we need to declare an object of type student. Like this :-

**Syntax: name_of_the_struct name_of_the_object;**

**Example: student S;**

Then we can access any data member like this using the '.' operator :-

**Syntax: name_of_the_object . name_of_the_data_member**

**Example: S.name;**

Few comments :-

1. The object can be an array.

2. The data member can also be an array.

Combining all the above points, we are going to write a program of our need.

```
#include <iostream>
using namespace std;
struct student {
```

```
            int marks [5] ;           //the 2nd point
            string name ;
    };
    int main(){
            student S[100];           //the 1st point
            int i , j ;
            // take input for each student
            for(i=0 ; i < 100 ; i++){
                    // take input for marks of ith student
                    for(j=0 ; j < 5 ; j++){
                            cin >> S[i].marks[j];
                    }
                    // take input for name of ith student
                    getline(cin,S[i].name);
            }
            // Output for each student
            for(i=0 ; i < 100 ; i++){
                    cout << "Student " << i+1 << " : " ;
                    // print marks of ith student
                    cout << S[i].name<< " ";
                    for(j=0 ; j < 5 ; j++){
                            cout << S[i].marks[j] << " ";
                    }
                    cout << "\n";
            }
            return 0;
    }
```

Final comments :-

(i) For a function which needs a *struct* object, you need to take structure objects like normal variables in functions. This way :-

```
    return-type name(struct-name object-name, other parameters){
            ....
    }
```

(ii) You can pass the objects like a normal variable or array i.e. use just the name while calling a function.

(iii) Call by value and call by reference is valid for *struct* objects.

(iv) You can initialize a *struct* object like a normal single-dimensional array.

## typedef and #define

Now we will tell you something which will help you in shortening the code.

Using the command typedef it is possible to give a shorter name to a datatype like :-

```
typedef unsigned long long ull ;
```

After this, the code

```
unsigned long long a = 50986556421345;
unsigned long long b = 1235564554566;
cout << a + b;
```

can be written as

```
ull a = 50986556421345;
ull b = 1235564554566;
cout << a + b;
```

Another way to shorten your code is by using macros. A macro means that certain strings in the code will be changed before the compilation. In C++, macros are defined using #define keyword.

For example, we can define the following macro:-

```
#define PI 3.14159
```

and use it like this:-

```
cout << “Area: ” << PI * r * r ;
```

instead of :-

```
cout << “Area: ” << 3.14159 * r * r;
```

The difference becomes more evident once we read about STL.

A macro can also have parameters which makes it possible to shorten loops and other structures. For example, we can define the following macro:

```
#define REP(i,a,b)  for (int i = a; i <= b; i++)
```

After this, the code

```
for (int i = 1; i <= n; i++) {
        search (i) ;
```

```
                }
```

can be shortened as follows:

```
                REP(i,1,n) {
                        search(i);
                }
```

Sometimes macros cause bugs that may be difficult to detect. For example, consider the following macro that calculates the square of a number:

```
                #define SQ(a) a*a
```

This macro does not always work as expected. For example, the code

```
                cout << SQ(3+3) << "\n";
```

corresponds to the code

```
                cout << 3+3*3+3 << "\n";   // 15
```

A better version of the macro is as follows:

```
                #define SQ(a) (a)*(a)
```

Now the code

```
                cout << SQ(3+3) << "\n";
```

corresponds to the code

```
                cout << (3+3)*(3+3) << "\n";   // 36
```

= = = = = = = = = = = = = = =

**Exercise 2.12**

1. You are fortunate enough to be in a school which has a non-fictional and a functional library. Your librarian (See, I told you, you are lucky) has told you to make a record of all the books information – Name, Author's Name, Publication, Price. You make a *struct* Book and an array of object of type Book. Now, he tells you to sort the data according to price in descending order. As you are very lucky to be in such a school, you will have to do it.

= = = = = = = = = = = = = = =

## What to do next ?

1. Learn about C++ STL from web.
2. Make an account on Codechef (or HackerEarth) and practice some problems(the most submitted ones) from beginner's section.
3. Learn about time complexities.

4. Learn about Data Structures and Algorithms.
5. Start participating in Long and Short contests on various sites.

For points 3 and 4, www.commonlounge.com might be helpful. Make an account there and join the ‘Competitive Programming’ community. Then, subscribe and learn from the playlist there. http://www.geeksforgeeks.org/ and tutorials from HackerEarth might also be helpful. Points 4 and 5 should be done simultaneously.

## Don’t Give Up!