OBJECT ORIENTED PROGRAMMING (PCC-CS503)

Unit – 1

Procedural programming: An Overview of C

TOKENS: The smallest individual units in program are known as tokens.

C++ has the following tokens.

i. Keywords

ii. Identifiers

iii. Constants

iv. Operators

KEYWORDS: The keywords implement specific C++ language feature. They are explicitly reserved identifiers and can't be used as names for the program variables or other user defined program elements. The table shows C++'s keywords. Keywords shown in boldface are also keywords in ANSI C99. Keywords in italics are C++11 additions.

alignas	*alignof*	asm	**auto**	bool
break	**case**	catch	**char**	*char16_t*
char32_t	class	**const**	const_cast	*constexpr*
continue	*decltype*	**default**	delete	**do**
double	dynamic_cast	**else**	**enum**	explicit
export	**extern**	false	**float**	**for**
friend	**goto**	**if**	**inline**	**int**
long	mutable	namespace	new	*noexcept*
nullptr	operator	private	protected	public
register	reinterpret_cast	**return**	**short**	**signed**
sizeof	**static**	*static_assert*	static_cast	**struct**
switch	template	this	*thread_local*	throw
true	try	**typedef**	typeid	typename
union	**unsigned**	using	virtual	**void**
volatile	wchar_t	**while**		

IDENTIFIERS: Identifiers refers to the name of variable, functions, array, class etc. created by programmer. Each language has its own rule for naming the identifiers. The following rules are common for both C and C++.

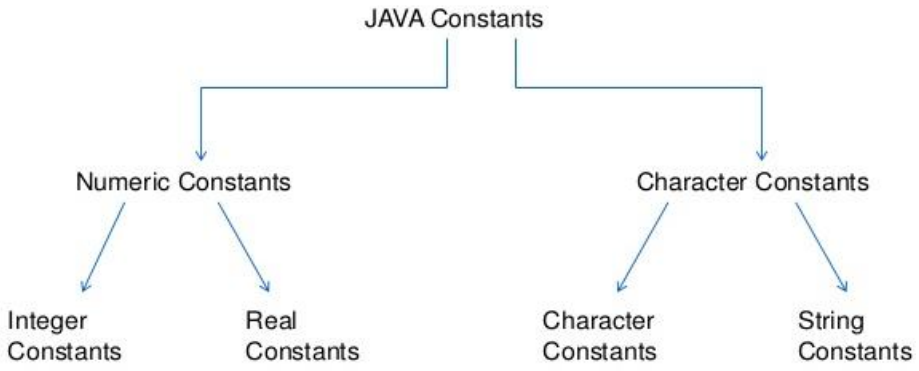
1. Only alphabetic chars, digits and underscore are permitted.

2. The name can't start with a digit.

3. Upper case and lower case letters are distinct.

4. A declared keyword can't be used as a variable name.

CONSTANTS: A **constant** is a fixed value that cannot be changed during the execution of the program.



Integer constants:

- Refers to sequence of digits
- Are of 3 types:
    - Decimal (354, -37, 0, 246543)
    - Octal (037, 0, 0435, 0551)
    - Hexadecimal (0X2, 0X9F, 0X, 0Xbcd)

Real or **floating point** constants:

- Numbers containing fractional part
- Examples:
    - 4.642, -0.034, 374.7860
    - Exponential notation:
    - 5.98e24 (mass of earth),
    - 9.11e-31 (mass of electron)

Character constants:

- Contains a single character enclosed within a pair of single quote marks.
- Examples:
  'A', 'y', '8', '*', ' ' , ';'

String constants:

- Contains a sequence of characters enclosed within double quotes.
- Examples:
  “Hello Java”, “1997”, “WELLDONE !”, “5+3”, “X”

OPERATORS: An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides the following types of operators−

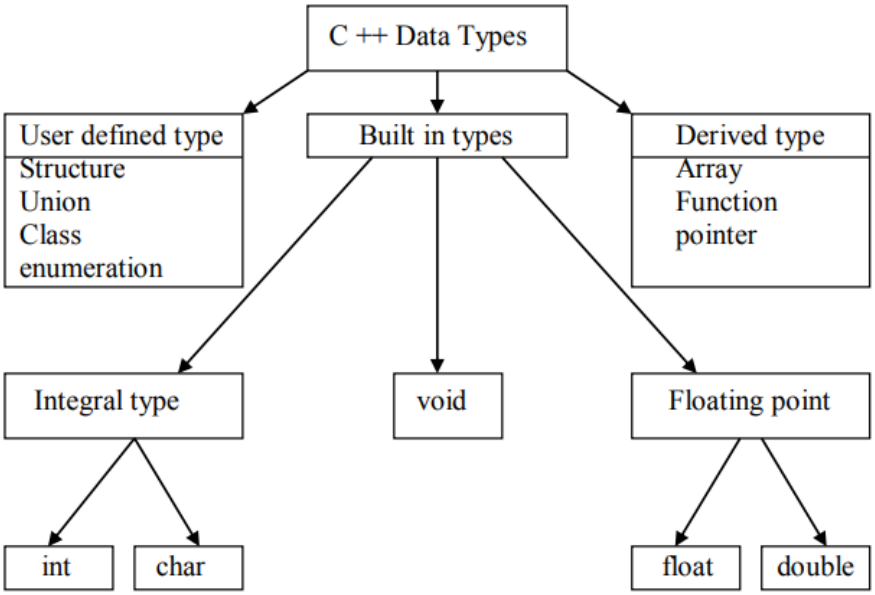
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

https://www.tutorialspoint.com/cplusplus/cpp_operators.htm

https://www.w3schools.com/cpp/cpp_operators.asp

Data-types in C++



- Both C and C++ compilers support all the built in types.
- With the exception of 'void', the basic datatypes may have several modifiers preceding them to serve the needs of various situations.
- The modifiers signed, unsigned, long and short may applied to character and integer basic data types.
- However the modifier long may also be applied to double.

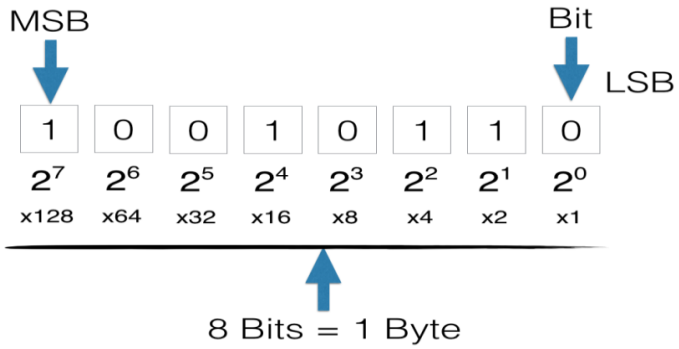
Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types.

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
Int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 18,446,744,073,709,551,615
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

Concept of sign ?



In case of signed numbers, the MSB bit is reserved for the sign (-ve or +ve) and so only the rest of the seven bits can represent the number. On the other hand, all eight bits are used to represent the number in case of unsigned numbers.

Scope and Lifetime of Variables

- The scope and lifetime of variables are defined by the variable's "**Storage-class**"
- A **scope** of a variable is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.
- **Lifetime** of any variable is the time for which the particular variable outlives in memory during running of the program.
- **Storage classes** in C are used to determine the lifetime, visibility (scope), memory location (storage space), and the default initial value of a variable.
- There are four types of storage classes in C
  - Automatic
  - External
  - Static
  - Register

Auto:

- Variables which are defined within a function or a block (block is a section of code which is grouped together. E.g. statements written within curly braces constitute a block of code) by default belong to the auto storage class.
- These variables are also called **local variables** because these are local to the function and are **by default assigned some garbage value**.
- Since these variables are declared inside a function, therefore these can only be accessed inside that function.
- There is no need to put 'auto' while declaring these variables because these are by default auto.

Example:

```c
#include <stdio.h>
int sum(int n1, int n2){
  auto int s;        //declaration of auto(local) variable
  s = n1+n2;
  return s;
}
int main(){
  int i = 2, j = 3, k;
 k = sum(i, j);
  printf("sum is : %d\n", k);
  return 0;

}

O/P:      sum is : 5
```

Extern:

- We write **extern** keyword before a variable to tell the compiler that this variable is declared somewhere else. Basically, by writing extern keyword before any variable tells us that this variable is a global variable declared in some other program file.
- Now let's see what actually happens.
- You must be knowing what a global variable is. A global variable is a variable which is declared outside of all the functions. It can be accessed throughout the program and we can change its value anytime within any function as follows.

```c
#include <stdio.h>
int g;
void print(){
  g = 10;
  printf("g = %d\n", g);
}
int main(){
  g = 7;
```

```
  printf("g = %d\n", g);
  print();
  return 0;
}

O/P:
g = 7
g = 10
```

Here, g is the global variable defined outside of all the functions. In the main function, its value was assigned as 7 and in the print function as 10.
While declaring a global variable, some space in memory gets allocated to it like all other variables. We can assign a value to it in the same program file in which it is declared as we did in the above example. But what if we want to use it or assign it a value in any other program file.

We can do so by using extern keyword as shown below.

firstfile.c

```
int g = 0;
```

In the first program file firstfile.c, we declared a global variable g.

Now, we will declare this variable 'g' as extern in a header file firstfile.h and then include it in the second file in which we want to use this variable.

firstfile.h

```
extern int g;
```

Now in the second program file secondfile.c, in order to use the global variable 'g', we need to include the header file in it by writing #include "firstfile.h". Here we assigned a value 4 to the variable 'g' and thus the value of 'g' in this program becomes 4.

secondfile.c

```
#include "firstfile.h"
main(){
 g = 4;
 printf("%d", g);
}
```

Static:

- A variable declared as static once initialized, exists till the end of the program. If a static variable is declared inside a function, it remains into existence till the end of the program and not get destroyed as the function exists (as in auto). If a static variable is declared outside all the functions in a program, it can be used only in the program in which it is declared and is not visible to other program files(as in extern).
- Let's see an example of a static variable.

```
#include <stdio.h>
 static int g = 5;
 void fn(){
       static int i = 0;
       printf("g = %d\t", g--);
       printf("i = %d\n",i++);
 }
int main(){
  while(g >= 2)
  fn();
  return 0;
}

O/P:
g = 5  i = 0
g = 4  i = 1
g = 3  i = 2
g = 2  i = 3
```

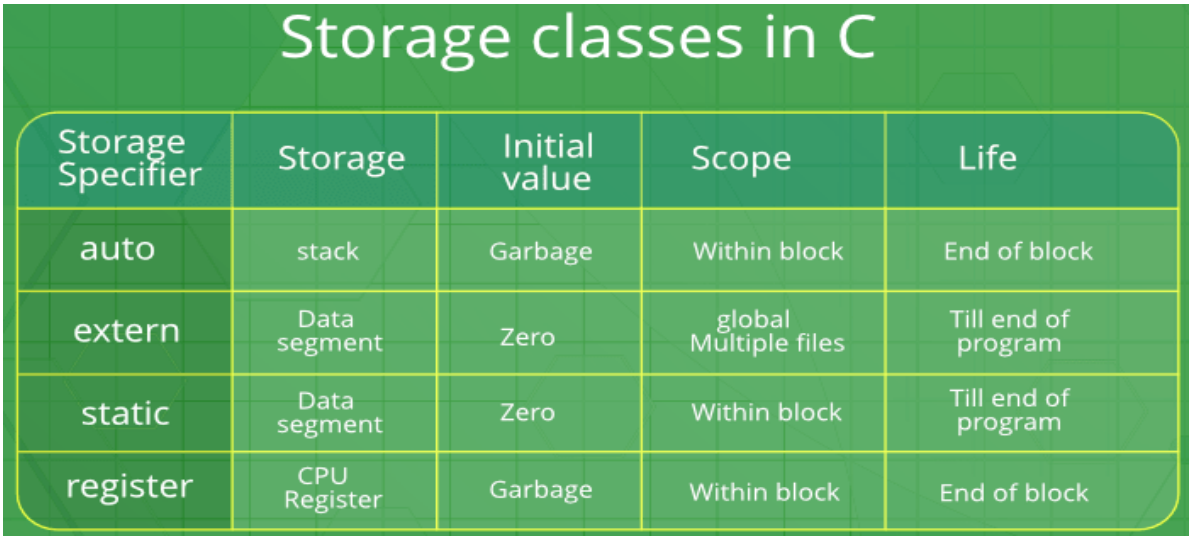
Here, g and i are the static variables in which 'g' is a global variable and 'i' is a local variable. If we had not written static before the declaration of 'i', then everytime the function 'fn()' would have been called, 'i' would have

been declared every time with an initial value of 0 and as the function 'fn()' would exit, it would also have got destroyed.

Register:

- It tells the compiler that the **variable will get stored in a register instead of memory (RAM)**. We can access a register variable faster than a normal variable. Not all the registers defined as register will get stored in a register since it depends on various restrictions of implementation and hardware.
- We cannot access the address of such variables since these do not have a memory location as which becomes clear by the following example.

```c
#include <stdio.h>
int main()
{
    register int n = 20;
    int *ptr;
    ptr = &n;
    printf("address of n : %u", ptr);
    return 0;
}

O/P:
prog.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]
main(){
^
prog.c: In function 'main':
prog.c:5:1: error: address of register variable 'n' requested
ptr = &n;
^
```



Pointers

We will discuss **pointers** here. Playing with pointers in C is really fun but pay a little more attention in this chapter because pointers are considered to be the toughest part of the C.

Now, let's again have a look at the following declaration:

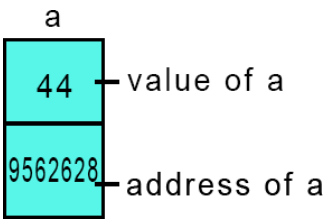
int a= 44;

As we have already seen, it is like

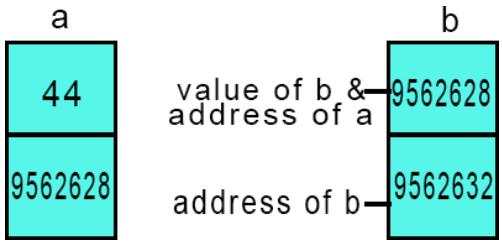
44
a

Now let's go to the undiscussed part. As we all know that when we declare 'a', it is given a memory location and the value of 'a' is stored in that memory location. In the world of programming, 'a' will also have an address. So, this address is the address of that memory location in which the value of 'a' is stored.

Address of 'a' is an integer which is something like 9562628. It will vary for every computer as per memory given to 'a' at that time.

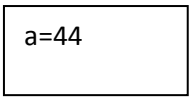


Now coming to pointer, a **pointer** points to some variable, that is, it stores the address of a variable. E.g.- if 'a' has an address 9562628, then the pointer to 'a' will store a value 9562628 in it. So, if 'b' is a pointer to 'a' and the value of 'a' is 10 and its address is 9562628, then 'b' will have a value 9562628 and its address will be something different.



Address in C is represented as **&a**, read as **address of a**. Remember that all the time when we were taking value of variable using scanf, we were taking an input from user and storing it at the address of that variable.

How to Use Pointers?



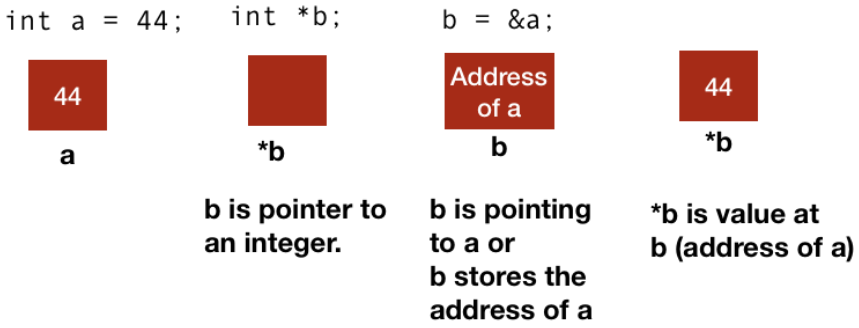

```
int a = 44;
int *b; /* declaration of pointer b */
b = &a;
```

int *b → This statement should mean that '*b' is an integer but then what is significance of '*' before 'b'? It means that b points to some integer ('b' is a pointer to some integer).

Or we can say that 'b' will store the address of some integer.

b = &a → As said, 'b' will store the address of some integer because it is a pointer to an integer. In this declaration, it is storing the address of 'a'. Since, 'b' is a pointer and '&a' represents address, so, by the declaration b = &a; means that 'b' will now store the address of 'a'.

int *b means that *b is an integer and *b is an integer which has the value of the variable which 'b' is pointing to. Here *b is 44. Here, 'b' is pointing to 'a', therefore 'b' will store the address of 'a' and '*b' will be the integer to which 'b' is pointing i.e., 'a'.

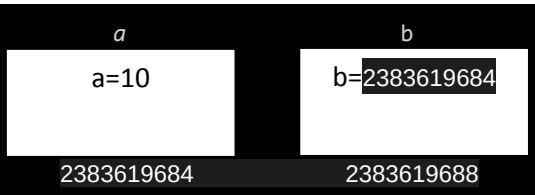


So in short,
int a → 'a' is an integer.
int *b → 'b' is an pointer to an integer.
b = &a → 'b' is now pointing to 'a'.
'b' has the address of 'a' and '*b' has the value of 'a'.

Example of pointer.

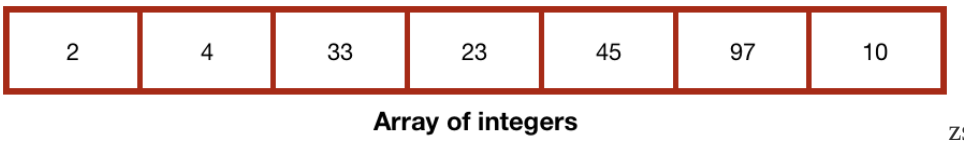
```c
#include <stdio.h>
int main()
```

```c
{
    int a = 10;
    int *b;
    b = &a;
    printf("b = %u\n", b);
    printf("*b = %d\n", *b);
    printf("&b = %u\n", &b);
    printf("*&b = %u\n", *&b);
    return 0;
}

O/P:
b = 2383619684        // address of a
*b = 10               //value of a
&b = 2383619688       // address of b
*&b = 2383619684      //address of a
```



Arrays

it is a collection of similar type of data which can be either of int, float, double, char (String), etc. All the data types must be same. For example, we can't have an array in which some of the data are integer and some are float.



Array of integers

zs

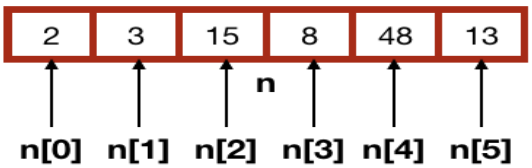
Why Array?

Suppose we need to store marks of 50 students in a class and calculate the average marks. So, declaring 50 separate variables will do the job but no programmer would like to do so. And there comes **array** in action.

How to declare an array?

`int n[ ] = {2, 3, 15, 8, 48, 13};//` 'int n[6]' will allocate space to 6 integers

element	2	3	15	8	48	13
index	0	1	2	3	4	5



Example:

```c
#include <stdio.h>
int main()
{
    int  marks[3];
    float  average;

    printf("Enter marks of first student\n");
    scanf(" %d" , &marks[0]);

    printf("Enter marks of second student\n");
    scanf(" %d" , &marks[1]);

    printf("Enter marks of third student\n");
    scanf(" %d" , &marks[2]);

    average = (marks[0] + marks[1] + marks[2]) / 3.0;
    printf ("Average marks : %f\n" , average);
```

```
    return 0;
}

O/P:
Enter marks of first student
23
Enter marks of second student
25
Enter marks of third student
30
Average marks : 26.000000
```

Same code in C++:

```
#include <iostream>
int main(){

        using namespace std;

        int n[10]; /* declaring n as an array of 10 integers */
        int i,j;

        /* initializing elements of array n */
         for ( i = 0; i<10; i++ )
         {
                 cout << "Enter value of n[" << i << "]"<< endl;
                cin >> n[i];
        }

        /* printing the values of elements of array */
        for (j = 0; j < 10; j++ )
        {
               cout << "n[" << j << "] = " << n[j] << endl;
        }

        return 0;
}
```

Pointer to an Array:

Consider the following code:

```
int age[50];
int *p;
p = age;
```

If **p** is a pointer to the array **age**, then it means that p (or age) points to age[0] (i.e. first element of the array age[])

Now, since **p** points to the first element of the array **age**, ***p** is the value of the first element of the array.

So, *p is age[0], *(p+1) is age[1], *(p+2) is age[2].

Similarly, *age is age[0] (value at age), *(age+1) is age[1] (value at age+1), *(age+2) is age[2] (value at age+2) and so on.

That's all in pointer to arrays.

Now let's see some examples.

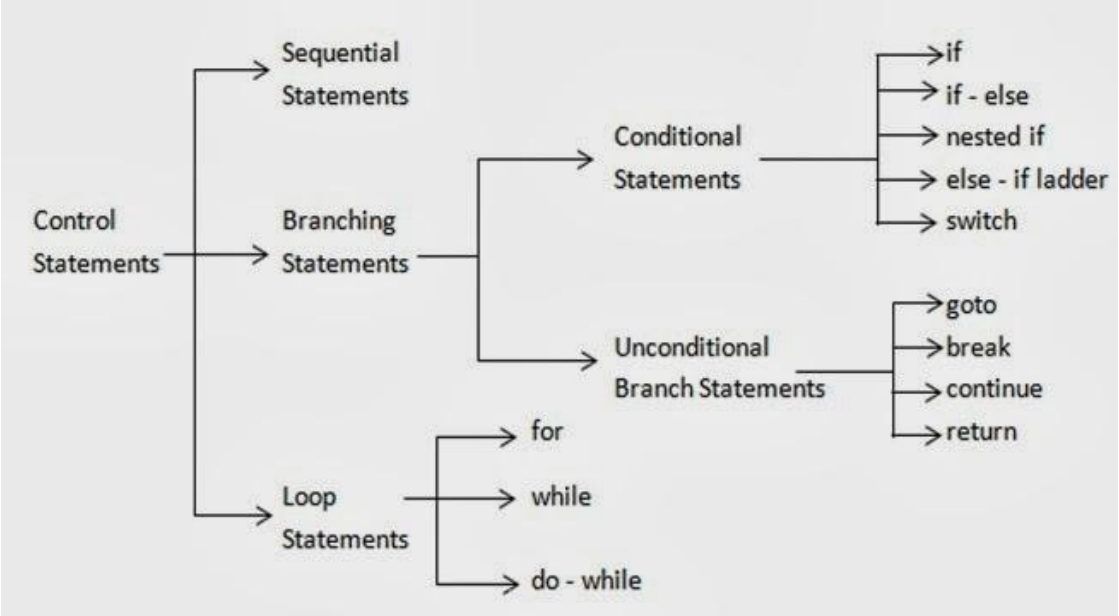
```
#include <iostream>
int main(){

        float n[5] = { 20.4, 30.0, 5.8, 67, 15.2 }; /* declaring n as an array of 5 floats */
        float *p;                   /* p as a pointer to float */
        int i;
```

```
        p = n;            /* p now points to array n */
        /* printing the values of elements of array */
        for (i = 0; i < 5; i++ )
        {
        std::cout << "*(p + " << i << ") = " << *(p + i) << std::endl;/* *(p+i) means value at
(p+0),(p+1)...*/
        }

        return 0;
}

O/P:

*(p + 0) = 20.4
*(p + 1) = 30
*(p + 2) = 5.8
*(p + 3) = 67
*(p + 4) = 15.2
```

Since **p** is pointing to the first element of array, so, ***p** or ***(p+0)** represents the value at **p[0]** or the **value at the first element of p**. Similarly, ***(p+1)** represents value at **p[1]**. So ***(p+3)** and ***(p+4)** represent the values at **p[3]** and **p[4]** respectively. So, accordingly, we will get the output.

Control-Flow

The order in which statements are executed known as control flow and the statements that are used to control the flow of execution of the program are called as control statements. Based on the order in which the statements are executed, the various control statements are classified as follows:



If-Else

If-Else statements allow the program to execute different blocks of code depending on conditionals. All If statements have the following form:

```
if ( condition ) {
    //body
}
```

An If statement executes the code in the body section if the condition evaluates to True, otherwise it skips the body. When the condition evaluates to False, the program will either test another condition with a following Else-if, run the code inside an Else statement, or continue as normal if neither an Else-if nor Else block exist.

An Else statement acts as a default case for If statements. In the case that an If is directly followed by an Else and the condition is false, the code in the Else is executed. Else statements do not have a condition themselves.

```
if ( condition ) {
    //body
} else {
    // else body
}
```

An If statement can followed by any number of Else-if blocks. Each Else-if has its own conditional statement and body of code. If an Else-if conditional is False, then its body of code is skipped and the program will check the next Else-if in order that they appear. An If followed by a long list of Else-IF's is usually referred to as an If-Else ladder.

```
if ( condition ) {
    //body
} else if ( 2nd condition ) {
    // else if body
} else {
    // else body
}
```

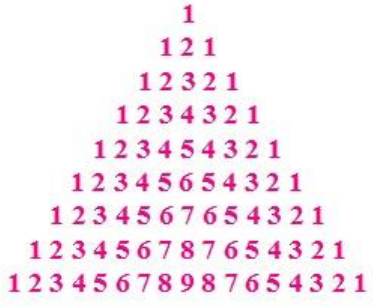
WAP: to find the largest of three numbers and print the largest number.
WAP: which is able to match the password with the corresponding username. If the password matches the username, it should be able to print "correct password", other print "incorrect password"?

For Loop

A for loop allows for a block of code to be executed until a conditional becomes false. For loops are usually used when a block of code needs to executed a fixed number of times. Each loop consists of 3 parts, an initialization step, the conditional, and an iteration step. The initialization is run before entering the loop, the condition is checked at the beginning of each run through the loop (including the first run), and the iteration step executes at the end of each pass through the loop, but before the condition is rechecked. It is usual practice to have the iteration step move the loop on step closer to making the condition false, thus ending the loop, but this does not need to be the case.

```
for( initialization ; conditional ; iteration ) {
    // loop body
}
```

WAP: to print the following pattern:



WAP: to find the factorial of a number using for loop.

While Loop

A while loop is a simple loop that will run the same code over and over as long as a given conditional is true. The condition is checked at the beginning of each run through the loop (including the first one). If the conditional is false for the beginning, the while loop will be skipped all together.

```
while ( conditional ) {
    // loop body
}
```

Do-while Loop

A do-while loop acts just like a while loop, except the condition is checked at the end of each pass through the loop body. This means a do-while loop will execute at least once.

```
do {
    // loop body
} while ( condition );
```

WAP: to check whether a number is palindrome or not.
WAP: to print the sum of digits of a number entered by the user.
WAP: to find the sum of positive numbers entered by the user using a do-while loop. If the user enters a negative number, the loop ends. The negative number entered is not added to the sum.

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

int main() {
    int number = 0;
    int sum = 0;

    do {
        sum += number;

        // take input from the user
        cout << "Enter a number: ";
        cin >> number;
    }
    while (number >= 0);

    // display the sum
    cout << "\nThe sum is " << sum << endl;

    return 0;
}
```

Functions

- A function is block of code which is used to perform a particular task.
- A function **declaration** tells the compiler about a function's name, return type, and parameters. The actual body of the function can be defined separately.

  Syntax:
  ```
  return_type function_name( parameter list );
  e.g. int max(int num1, int num2);
  ```

- A function **definition** provides the actual body of the function.

  Syntax:

```
return_type function_name( parameter list )
{
   body of the function
}
```

Return Type − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
Function Body − The function body contains a collection of statements that define what the function does.

- To use a function, we need to call it. Once we call a function, it performs its operations and after that, the control again passes to the main program. To call a function, we need to specify the function name along with its parameters followed by ;

Syntax:

```
function_name ( parameters );
```

Example of a function:

```
#include <iostream.h>

float average( int x, int y )        //formal arguments
{
        float avg; /* declaring local variable */
        avg = ( x + y )/2.0;
        return avg; /* returning the average value */
}

int main(){
        using namespace std;
        int num1, num2;              //actual arguments
        float c;
        cout << "Enter first number" << endl;
        cin >> num1;
        cout << "Enter second number" << endl;
        cin >> num2;
        c = average( num1, num2 ); /* calling the function average and storing its value in c*/
        cout << "Average is " << c << endl;
        return 0;
}
```

Parameter Passing to functions

The parameters passed to function are called ***actual parameters***. For example, in the above program num1 and num2 are actual parameters.

The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters.

There are two most popular ways to pass parameters.

Pass by Value: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

Pass by Reference: Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.
This can be further done using:

1. Call by pointers : `float average( int *x, int *y )`
2. Call by address : `float average( int &x, int &y )`

Question: What is the difference between following two codes?

Ex:

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

void add(int a, int b)
{
cout << a+b;
}
```

Ans: When you call a function that called function (which has a return type of 'void') doesn't return any value to the called function. In this example the above function adds the values and returns them as an integer to the calling function, whereas the later just adds and prints them.

Program Structure in C++

```
// my first program in C++            //comments     /*bklfdl*/
#include<iostream.h>                  //Pre-processor directives
#include<conio.h>
using namespace std;                                 //blank statement
void main()                           //main function
{                                     //opening brace
  cout << "Hello World!";             //standard character output, insertion operator
  getch();                            //to hold the output screen
}                                     //closing brace
```

Practical session 1: Branching and Looping Constructs

Practical session 2: Call by value and call by reference, recursion, user-defined and inbuilt functions.

Recursion:

1. WAP to display the n terms in Fibonacci series using a recursive call.
2. Define a function to calculate power of a number raised to other i.e. a^b using recursion where the numbers 'a' and 'b' are to be entered by the user
3. WAP to generate the random numbers in any specific range specified by the user. Ensure that for each run the numbers generated even in the same range are random and not same.

Enter the range for random nos.:100

200

Enter the no. of random nos. to be generated: 20

The random nos. are: 115, 165, 132

Namespace

A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.
Defining a Namespace

Hello1.cpp
Display() → "helo 1"
X=20

Hello2.cpp
Display() → "helo 2"
X=30

Hello3.cpp
Display(); //ambiguity: which Display()?
x; // ambiguity: which X

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows −

```
namespace namespace_name {
   // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows-

```
name::code; // code could be variable or function
```

Example 1: how namespace scope the entities including variable and functions –

```
// Creating namespaces
#include <iostream>
using namespace std;
namespace ns1
{
      int value() { return 5; }
}
namespace ns2
{
      const double x = 100;
      double value() { return 2*x; }
}

int main()
{
      // Access value function within ns1
      cout << ns1::value() << '\n';

      // Access value function within ns2
      //cout << ns2::value() << '\n';

      // Access variable x directly
      //cout << ns2::x << '\n';

      return 0;
}
```

Another example:

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

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
```

```
    }
}

// second name space
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}

using namespace first_space;
int main () {
   // This calls function from first name space.
   func();

   return 0;
}
```

C – Library functions

- Library functions in C language are inbuilt functions which are grouped together and placed in a common place called library.
- Each library function in C performs specific operation.
- We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.
- These library functions are created by the persons who designed and created C compilers.
- All C standard library functions are declared in their respective header files (library) which are saved as file_name.h.
- Actually, function declaration, definition for macros are given in all header files.
- When we include header files in our C program using "#include<filename.h>" command, all C code of the header files are included in C program. Then, this C program is compiled by compiler and executed.

LIST OF INBUILT C FUNCTIONS IN STDLIB.H FILE:

Function	Description
malloc()	This function is used to allocate space in memory during the execution of the program.
calloc()	This function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn't
realloc()	This function modifies the allocated memory size by malloc () and calloc () functions to new size
free()	This function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.
abs()	This function returns the absolute value of an integer. The absolute value of a number is always positive. Only integer values are supported in C.
div()	This function performs division operation
abort()	It terminates the C program
exit()	This function terminates the program and does not return any value
system()	This function is used to execute commands outside the C program.
atoi()	Converts string to int
atol()	Converts string to long
atof()	Converts string to float

strtod()	Converts string to double
strtol()	Converts string to long
getenv()	This function gets the current value of the environment variable
setenv()	This function sets the value for environment variable
putenv()	This function modifies the value for environment variable
perror()	This function displays most recent error that happened during library function call.
rand()	This function returns the random integer numbers
delay()	This function Suspends the execution of the program for particular time

LIST OF INBUILT C FUNCTIONS IN MATH.H FILE:

- "math.h" header file supports all the mathematical related functions in C language. All the arithmetic functions used in C language are given below.
- Click on each function name below for detail description and example programs.

Function	Description
floor ( )	This function returns the nearest integer which is less than or equal to the argument passed to this function.
round ( )	This function returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from ".1 to .5", it returns integer value less than the argument. If decimal value is from ".6 to .9", it returns the integer value greater than the argument.
ceil ( )	This function returns nearest integer value which is greater than or equal to the argument passed to this function.
sin ( )	This function is used to calculate sine value.
cos ( )	This function is used to calculate cosine.
cosh ( )	This function is used to calculate hyperbolic cosine.
exp ( )	This function is used to calculate the exponential "e" to the x^{th} power.
tan ( )	This function is used to calculate tangent.
tanh ( )	This function is used to calculate hyperbolic tangent.
sinh ( )	This function is used to calculate hyperbolic sine.
log ( )	This function is used to calculates natural logarithm.
log10 ( )	This function is used to calculates base 10 logarithm.
sqrt ( )	This function is used to find square root of the argument passed to this function.
pow ( )	This is used to find the power of the given number.
trunc.(.)	This function truncates the decimal value from floating point value and returns integer value.

LIST OF INBUILT C FUNCTIONS IN STRING.H FILE:

String functions	Description
strcat ( )	Concatenates str2 at the end of str1
strncat ( )	Appends a portion of string to another
strcpy ( )	Copies str2 into str1
strncpy ( )	Copies given number of characters of one string to another
strlen ( )	Gives the length of str1
strcmp ( )	Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns >0 if str1 > str2
strcmpi ( )	Same as strcmp() function. But, this function negotiates case. "A" and "a" are treated as same.
strchr ( )	Returns pointer to first occurrence of char in str1
strrchr ( )	last occurrence of given character in a string is found
strstr ( )	Returns pointer to first occurrence of str2 in str1
strrstr ( )	Returns pointer to last occurrence of str2 in str1
strdup ( )	Duplicates the string
strlwr ( )	Converts string to lowercase
strupr ( )	Converts string to uppercase
strrev ( )	Reverses the given string
strset ( )	Sets all character in a string to given character
strnset ( )	It sets the portion of characters in a string to given character
strtok ( )	Tokenizing given string using delimiter

string functions	Description
memset()	It is used to initialize a specified number of bytes to null or any other value in the buffer
memcpy()	It is used to copy a specified number of bytes from one memory to another
memmove()	It is used to copy a specified number of bytes from one memory to another or to overlap on same memory.
memcmp()	It is used to compare specified number of characters from two buffers
memicmp()	It is used to compare specified number of characters from two buffers regardless of the case of the characters
memchr()	It is used to locate the first occurrence of the character in the specified string

Error Handling in C

- C language does not provide any direct support for error handling.
- However a few methods and variables defined in **error.h** header file can be used to point out error using the return statement in a function.
- In C language, a function returns -1 or NULL value in case of any error and a global variable **errno** is set with the error code. So the return value can be used to check error while programming.

Errno : Whenever a function call is made in C language, a variable named errno is associated with it. It is a global variable, which can be used to identify which type of error was encountered while function execution, based on its value. Below we have the list of Error numbers and what does they mean.

errno value	Error
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Argument list too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

C language uses the following functions to represent error messages associated with **errno**:

- perror(): returns the string passed to it along with the textual represention of the current errno value.
- strerror() is defined in **string.h** library. This method returns a pointer to the string representation of the current errno value.

Example:

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main ()
{
    FILE *fp;
```

```
    /*
        If a file, which does not exists, is opened,
        we will get an error
    */
    fp = fopen("IWillReturnError.txt", "r");

    printf("Value of errno: %d\n ", errno);
    printf("The error message is : %s\n", strerror(errno));
    perror("Message from perror");
    return 0;
}
```

O/P:
Value of errno: 2
The error message is: No such file or directory
Message from perror: No such file or directory

Command line arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example (argu1.c) which checks if there is any argument supplied from the command line and take action accordingly –

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

O/P 1:

C:\TURBOC\BIN> argu1 neha

argv[0]= argu1
argv[1]=neha

The argument supplied is neha
O/P 2:

C:\TURBOC\BIN> argu1 neha heena

argv[0]= argu1
argv[1]=neha
argv[2]=heena

Too many arguments supplied.

O/P 3:

C:\TURBOC\BIN> argu1
argv[0]= argu1

One argument expected