Back To Course

Learn

Classroom

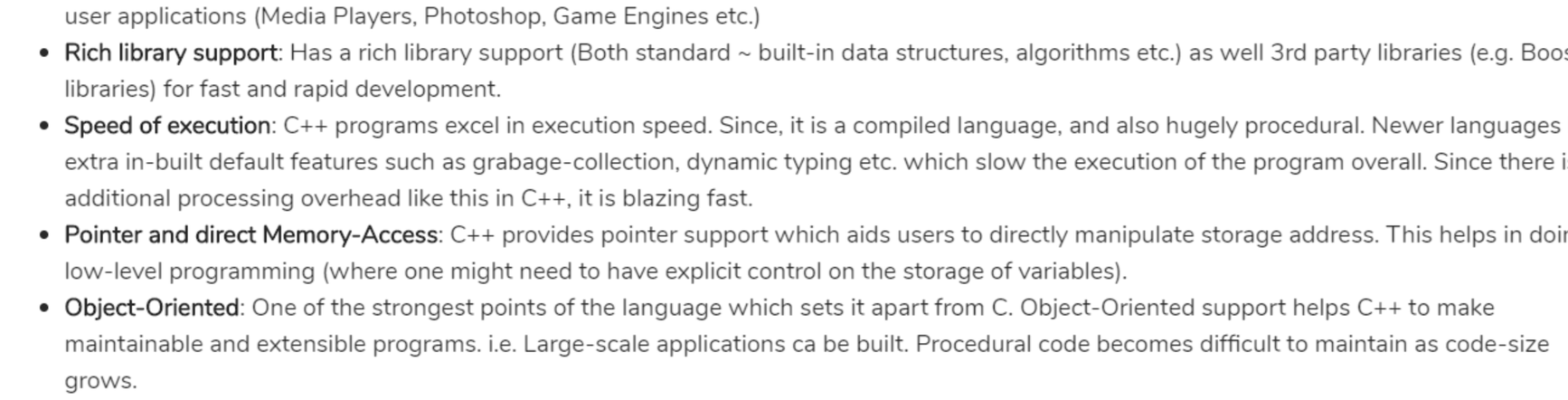
Theory

Quiz

We have combined Classroom and Theory tab and created a new Learn tab for easy access. You can access Classroom and Theory from the left panel.

Introduction to C++

C++ is a general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm. It is an imperative and a **compiled** language.



C++ is a middle-level language rendering it the advantage of programming low-level (drivers, kernels) and even higher-level applications (games, GUI, desktop apps etc.). The basic syntax and code structure of both C and C++ are the same.

Some of the ***features & key-points*** to note about the programming language are as follows:

- **Simple**: It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich libray support and a variety of data-types.
- **Machine Independent but Platform Dependent**: A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.
- **Mid-level language**: It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)
- **Rich library support**: Has a rich library support (Both standard ~ built-in data structures, algorithms etc.) as well 3rd party libraries (e.g. Boost libraries) for fast and rapid development.
- **Speed of execution**: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as grabage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.
- **Pointer and direct Memory-Access**: C++ provides pointer support which aids users to directly manipulate storage address. This helps in doing low-level programming (where one might need to have explicit control on the storage of variables).
- **Object-Oriented**: One of the strongest points of the language which sets it apart from C. Object-Oriented support helps C++ to make maintainable and extensible programs. i.e. Large-scale applications ca be built. Procedural code becomes difficult to maintain as code-size grows.
- **Compiled Language**: C++ is a compiled language, contributing to its speed.

C++ finds varied usage in applications such as:

- Operating Systems & Systems Programming. e.g. *Linux-based OS (Ubuntu etc.)*
- Browsers *(Chrome & Firefox)*
- Graphics & Game engines *(Photoshop, Blender, Unreal-Engine)*
- Database Engines *(MySQL, MongoDB, Redis etc.)*
- Cloud/Distributed Systems

First C++ program - Printing "Hello World"

Learning C++ programming can be simplified into:

- Writing your program in a text-editor and saving it with correct extension(.CPP, .C, .CP)
- Compiling your program using a compiler or online IDE

The "Hello World" program is the first step towards learning any programming language and also one of the simplest programs you will learn. All you have to do is display the message "Hello World" on the screen. Let us now look at the program:

```
1
2  // Simple C++ program to display "Hello World"
3
4  // Header file for input output functions
5  #include<iostream>
6
7  using namespace std;
8
9  // main function -
10 // where the execution of program begins
11 int main()
12 {
13     // prints hello world
14     cout<<"Hello World";
15
16     return 0;
17 }
18
```

Run

Output:

```
Hello World
```

Let us now understand every line of the above program:

1. **// Simple C++ program to display "Hello World"** : This line is a comment line. A comment is used to display additional information about the program. A comment does not contain any programming logic. When a comment is encountered by a compiler, the compiler simply skips that line of code. Any line beginning with '//' without quotes OR in between /*...*/ in C++ is comment.
2. **#include**: In C++, all lines that start with pound (#) sign are called directives and are processed by preprocessor which is a program invoked by the compiler. The **#include** directive tells the compiler to include a file and **#include** . It tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.
3. **int main()**: This line is used to declare a function named "main" which returns data of integer type. A function is a group of statements that are designed to perform a specific task. Execution of every C++ program begins with the main() function, no matter where the function is located in the program. So, every C++ program must have a main() function.
4. **{ and }**: The opening braces '{' indicates the beginning of the main function and the closing braces '}' indicates the ending of the main function. Everything between these two comprises the body of the main function.
5. **cout<<"Hello World";**: This line tells the compiler to display the message "Hello Worlld" on the screen. This line is called a statement in C++. Every statement is meant to perform some task. A semi-colon ';' is used to end a statement. Semi-colon character at the end of the statement is used to indicate that the statement is ending there.
   The *cout* is used to identify the standard character output device which is usually the desktop screen. Everything followed by the character "<<" is displayed to the output device.
6. **return 0;** : This is also a statement. This statement is used to return a value from a function and indicates the finishing of a function. This statement is basically used in functions to return the results of the operations performed by a function.
7. **Indentation**: As you can see the *cout* and the return statement have been indented or moved to the right side. This is done to make the code more readable. In a program as Hello World, it does not hold much relevance seems but as the programs become more complex, it makes the code more readable, less error-prone. Therefore, you must always use indentations and comments to make the code more readable.

Basic Input/Output in C++

C++ comes with libraries that provide us many ways for performing input and output. In C++ input and output is performed in the form of a sequence of bytes or more commonly known as **streams**.

Input Stream: If the direction of flow of bytes is from a device(for example Keyboard) to the main memory then this process is called input.

Output Stream: If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.

Header files available in C++ for Input - Output operation are:

- **iostream**: iostream stands for standard input output stream. This header file contains definitions to objects like cin, cout, cerr etc.
- **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.
- **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

In C++ articles, these two keywords **cout** and **cin** are used very often for taking inputs and printing outputs. These two are the most basic methods of taking input and output in C++. For using cin and cout we must include the header file *iostream* in our program.

In this article, we will mainly discuss the objects defined in the header file *iostream* like cin and cout.

- **Standard output stream (cout)**: Usually the standard output device is the display screen. **cout** is the instance of the ostream class. cout is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator (**<<**).

```
1
2  #include <iostream>
3
4  using namespace std;
5
6  int main( ) {
7     char sample[] = "GeeksforGeeks";
8
9     cout << sample << " - A computer science portal for geeks";
10
11    return 0;
12 }
13
```

Run

Output:

```
GeeksforGeeks - A computer science portal for geeks
```

As you can see in the above program the insertion operator(**<<**) insert the value of the string variable **sample** followed by the string "A computer science portal for geeks" in the standard output stream **cout** which is then displayed on screen.

- **standard input stream (cin)**: Usually the input device is the keyboard. cin is the instance of the class **istream** and is used to read input from the standard input device which is usually the keyboard.
  The extraction operator(**>>**) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keboard.

```
1
2  #include<iostream>
3  using namespace std;
4
5  int main()
6  {
7      int age;
8
9      cout << "Enter your age:";
10     cin >> age;
11     cout << "\nYour age is: "<<age;
12
13     return 0;
14 }
15
```

Output:

```
Enter your age:
Input : 18

Your age is: 18
```

The above program asks the user to input the age. The object cin is connected to the input device. The age entered by the user is extracted from cin using the extraction operator(**>>**) and the extracted data is then stored in the variable **age** present on the right side of the extraction operator.

- **Un-buffered standard error stream (cerr)**: cerr is the standard error stream which is used to output the errors. This is also an instance of the ostream class. As cerr is un-buffered so it is used when we need to display the error message immediately. It does not have any buffer to store the error message and display later.

```cpp

#include <iostream>

using namespace std;

int main( )
{
   cerr << "An error occured";

   return 0;
}

```

Run

Output:

```
An error occured
```

- **buffered standard error stream (clog)**: This is also an instance of the ostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. The error message will be displayed on the screen too.

```cpp

#include <iostream>

using namespace std;

int main( )
{
   clog << "An error occured";

   return 0;
}

```

Run

Output:

```
An error occured
```

Preprocessor Directives in C++

Consider the following basic *Hello World* program.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    cout<<"Hello World!";

    return 0;
}

```

Run

Output:

```
Hello World!
```

The program begins with the highlighted statement (in grey): **#include**, which technically is known as a **preprocessor directive**. There are other such directives as well such as **#define**, **#ifdef**, **#pragma** etc.

So, what is a preprocessor directive?

> *A preprocessor directive is a statement which gets processed by the C++ preprocessor before compilation.*

In layman terms, such statements are evaluated prior to the procedure of generation of executable code.

For basic programming in C++, we only need to understand the **#include** and the **#define** directives.

1. **#include directive**: This type of preprocessor directive tells the compiler to include a file in the source code program and are also known as File Inclusion preprocessor directives. There are two types of files which can be included by the user in the program:
   - **Header File or Standard files**: These files contains definition of pre-defined functions like printf(), scanf() etc. These files must be included for working with these functions. Different function are declared in different header files. For example standard I/O funuctions are in 'iostream' file whereas functions which perform string operations are in 'string' file.

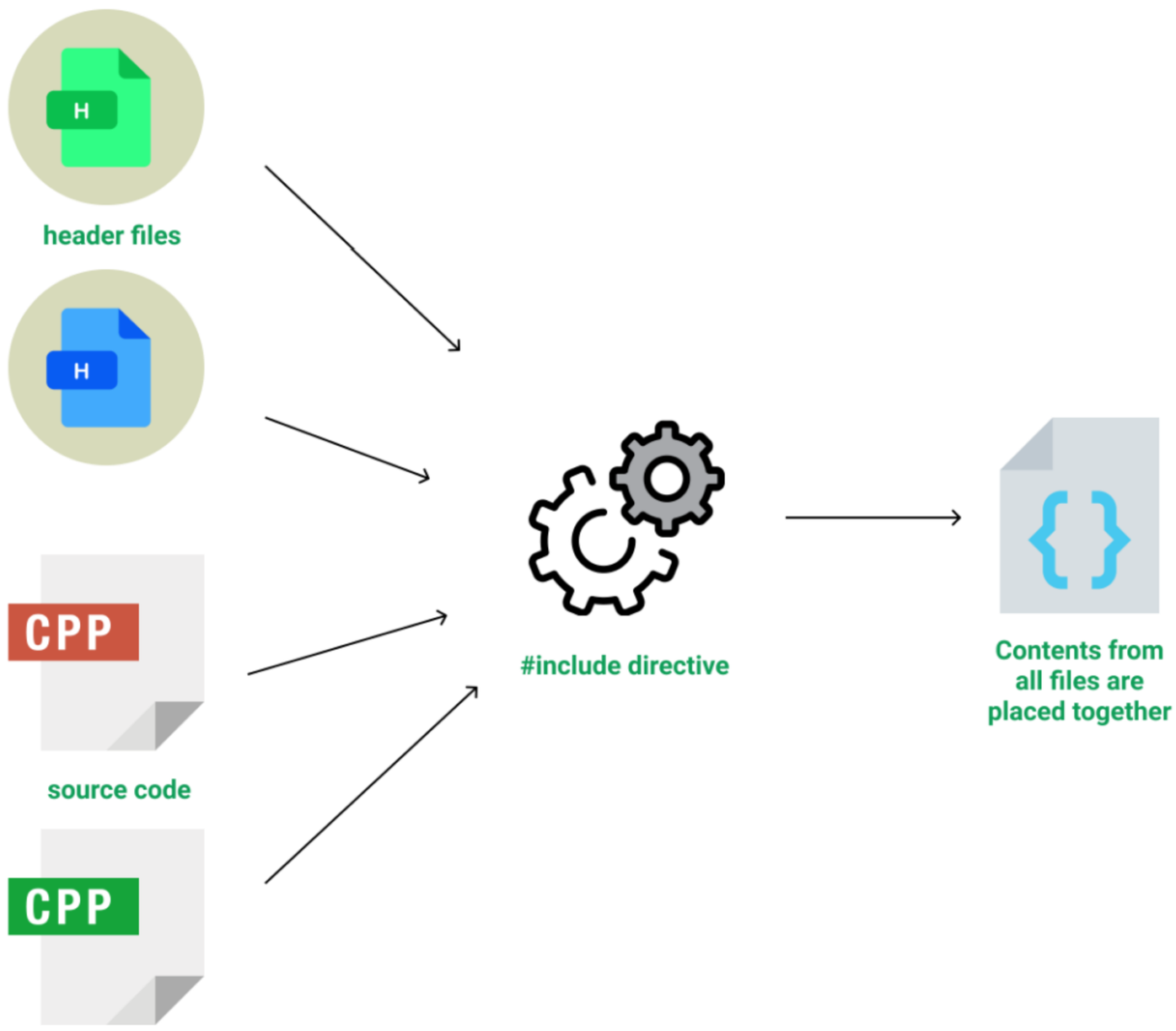
     ```
     #include <file_name>
     ```

     where *file_name* is the name of file to be included. The '<' and '>' brackets tells the compiler to look for the file in standard directory.
   - **User-defined Files**: When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as:

     ```
     #include "filename"
     ```

The #include directive instructs the preprocessor to fetch the contents of the file encapsulated within the quotes/arrow and placing it in the source file before compilation (simple copy-paste in layman terms). Some of the important includes are listed below:

- ***#include < iostream >***: Defines input/output stream objects (cin, cout etc.)
- ***#include < cstdio >***: C-standard input/output functions (printf, scanf, fscanf, fprintf, fgets etc.)
- ***#include < cstdlib >***: General purpose utilities functions (random number generation, dynamic memory allocation, integer arithmetic, conversions) (atoi, rand, srand, calloc, malloc)
- ***#include < bits/stdc++.h >***: Master directive to include all standard header files. (Note that it is not a standard header file and might not be available in compilers other than GCC)
- ***#include "user-defined-file"***: When we use quotes instead of arrows, we instruct to include user-defined files (.h, .cpp). (useful in projects where one needs to keep custom-functions together)



2. **#define directive**: This directive is used to declare **MACRO**s in the program. A MACRO is a piece of code which is designated a name. When the file is processed by the preprocessor, all appearances of the MACRO name gets replaced by its definition.

Example:

```
#include <bits/stdc++.h>
#define PI 3.14159265
using namespace std;
int main()
{
    int r=5;
    cout<<"Perimeter of Circle: "<<(2 * PI * r)<<endl;
    cout<<"Area of Circle: "<<(PI * r * r)<<endl;
    return 0;
}
```

Run

Output:

```
Perimeter of Circle: 31.4159
Area of Circle: 78.5398
```

Each appearance gets replaced by the value of *PI* resulting in (2 * 3.14159265 * r) and (3.14159265 * r * r). Some of the better use-cases of the

#define directive can be found in competitive-programming. Such declarations at the beginning of the program greatly increases coding speed (during competitions & placement tests).

```
#define f(l,r) for (int i = l; i < r; i++)
#define ll long long
#define ull unsigned long long
#define abs(x) (x < 0 ? (-x) : x)
```

Data Types and Variables in C++

Variable: A variable or an identifier in C++ is basically a storage location to hold some data till the completion of program execution. It has some memory allocated to it (the amount of memory allocated depends on the data-type or by the user, in case of user-defined data-types such as struct, union, etc).

We will learn about variables in detail later in this post. Let us first look at what are *Data-Types*?

Data Types: Data Types are used to bind an identifier (variable) to hold only values of certain types. Thereafter, only specific operations and manipulations supported by that type is allowed. A data-type also determines the amount of storage to be allocated to that identifier, (exact values of which is compiler/implementation dependent). Predefined data-types are as follows:

- **Integer**: Keyword used for integer data types is int. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.
- **Character**: Character data type is used for storing characters. Keyword used for the character data type is char. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
- **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either true or false. Keyword used for the boolean data type is bool.
- **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is a float. Float variables typically require 4 bytes of memory space.
- **Double Floating Point**: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for the double floating point data type is double. Double variables typically require 8 bytes of memory space.
- **void**: Void means without any value. void datatype represents a valueless entity. The void data type is used for those function which does not return a value.
- **Wide Character**: Wide character data type is also a character data type but this data type has a size greater than the normal 8-bit datatype. Represented by wchar_t. It is generally 2 or 4 bytes long.

Variables *Contd.*

C++ is a **strongly-typed** language, thus any variable defined with a data-type can't be changed to hold a value of a different type (in Python & Javascript, we can do so).

A variable is defined by specifying a data-type and a name as shown below:

```
int a;
float radius;
char ch;

// declaring multiple variables of same type:
int a, b, c;
float l, b, h;

```

There are some rules to variable declaration as well:

1. A variable name can begin only with an underscore or an alphabet (any case).
2. It can thereafter consist of any number of _, alphabets and numbers.
3. No special characters other than _ is allowed.
4. Depending on C++ implementations, there can be limitations to the length of variable name too. (e.g. Microsoft C++: 2048 characters)

Declaration, Definition & Initialization:

- Variable **declaration** refers to the moment when a variable is first declared/introduced to the program.
- **Definition** is the part where that particular variable is allocated a storage location in memory. Initially, if the user doesn't provide any explicit, it picks up whatever garbage value was present (in case of a local variable). In case of global and static variables are assigned by default 0 and NULL values even if not explicitly stated.
- **Initialization** is the moment when the user explicitly assigns a value to the variable.

```

int a; // Declaration + Definition
int a = 5;  //Declaration + Definition + Initialization

// Global context (declaration + definition + initialization)
int v;
...
int main() { ... }

```

One may argue that in almost cases, definition/memory allocation is bound to happen once a variable is declared. However in case of extern variables and functions, one can separate out the declaration and definition parts. e.g.

```

// Only variable is declared to the executable program,
// The compiler is informed that the memory
// for this identifier will be declared later
// (in this file or another)
extern int a;

// Similar is the case for prototype declarations of
// functions. A prototype declaration doesn't allocate
// space to store the function instructions.
int factorial(int n);

// Only when it is fully defined, storage is assigned
int factorial(int n)
{
    return n * factorial(n - 1);
}

```

Scope of Variables

Scope, in general, is defined as the extent up to which one can work with something. Variable scope is the extent of program code/block within which one has access to a variable. There are mainly 2 types of variables based on their respective scope:

- **Local Variables**: Variables defined inside a function/block are local variables. They can't be accessed outside that particular function/block. Local variables also have a lifetime equal to that of the function/block execution. i.e. They are allocated on **stack** and as soon as control exits the function/block, they are de-allocated. An example showing the scope of a local variable is given below:

```cpp

#include <bits/stdc++.h>
using namespace std;

void func()
{
    int age = 18;
    cout << "Inside Function: " << age <<endl;
}

int main()
{
    func();
    cout << "Outside Function: " << age << endl;

    return 0;
}

```

Run

Output:

```
Compilation Error:
prog.cpp: In function 'int main()':
prog.cpp:14:37: error: 'age' was not declared in this scope
     cout << "Outside Function: " << age << endl;
                                     ^
```

The above program won't compile because of we access variable *age* in line: 14, which is out of the scope of the function where age is declared.

- Global Variables

  As the name suggests, they can be accessed in any part of the program. They are defined at the top of the program after the include directives, outside any function. They are not allocated inside any function stack. Instead, they are allocated on the Initialized/Uninitialized segment of Program Memory. Thus, they have a lifetime equal to that of the whole program. Example:

```cpp

#include <bits/stdc++.h>
using namespace std;

int global_var = 10;

void func()
{
    cout << "Access inside func: " << global_var << endl;
}

int main()
{
    func();
    cout << "Access inside main: " << global_var << endl;

    return 0;
}

```

Run

Output:

```
Access inside func: 10
Access inside main: 10
```

NOTE: There might be a case where a variable of the same name is declared locally. Compiler in such a situation **gives precedence the local variable instead of the global one**. If we however, want to access the global variable specifically, we use scope-resolution as shown in the example below:

```cpp
#include <bits/stdc++.h>
using namespace std;

// Global x
int x = 5;

int main()
{
    // Local x
    int x = 10;

    cout << "Value of global x is " << ::x;
    cout<< "\nValue of local x is " << x;
    return 0;
}

```

Run

Output:

```
Value of global x is 5
Value of local x is 10
```

Data-type Modifiers

Data-type Modifiers

These are special keywords modifying the size of a particular data-type:

- signed
- unsigned
- short
- long

Some of the possible combinations, their memory limit (most compilers) and corresponding ranges are given below:

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
int	4	-2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4,294,967,295
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long int	8	0 to 18,446,744,073,709,551,615
char	1	-128 to 127
unsigned char	1	0 to 255
float	4	3.4E +/- 38 (7 digits)
double	8	1.7E +/- 308 (15 digits)
long double	8	same as **double**
wchar_t	2	0 to 65,535

auto keyword in C++

The **auto** keyword, as discussed earlier is a storage-specifier for variables. However, in C++, since all variables are by default automatic (created at the point of definition and destroyed when the block is exited), there remained no-use for this particular keyword. However, in modern C++, the auto keyword has been given a new meaning: ***Automatic Determination of Data-Type during Assignment***. e.g.

```
auto d = 5.67;
auto i = (5 + 6);

typedef struct node {
    int x;
    node(int x) { this->x = x; }
};

auto root = new node(5);

```

In all the above examples, we don't need to explicitly state the data-type of the resultant expression on the RHS during the assignment. This is called **Type Inference in C++**.

Good use of auto is to avoid long initializations when creating iterators for containers:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<pair<int,int> > v = {{1,1}, {2,2}, {3,3}};

    //don't have to bother about the type
    //of the container
    for (auto p: v)
        cout << p.first << " " << p.second << endl;

    return 0;
}

```

Run

Output:

```
1 1
2 2
3 3
```

In the above code, we avoid long declarations of variables and simply use *auto* keyword to infer the type of elements held by the container class.

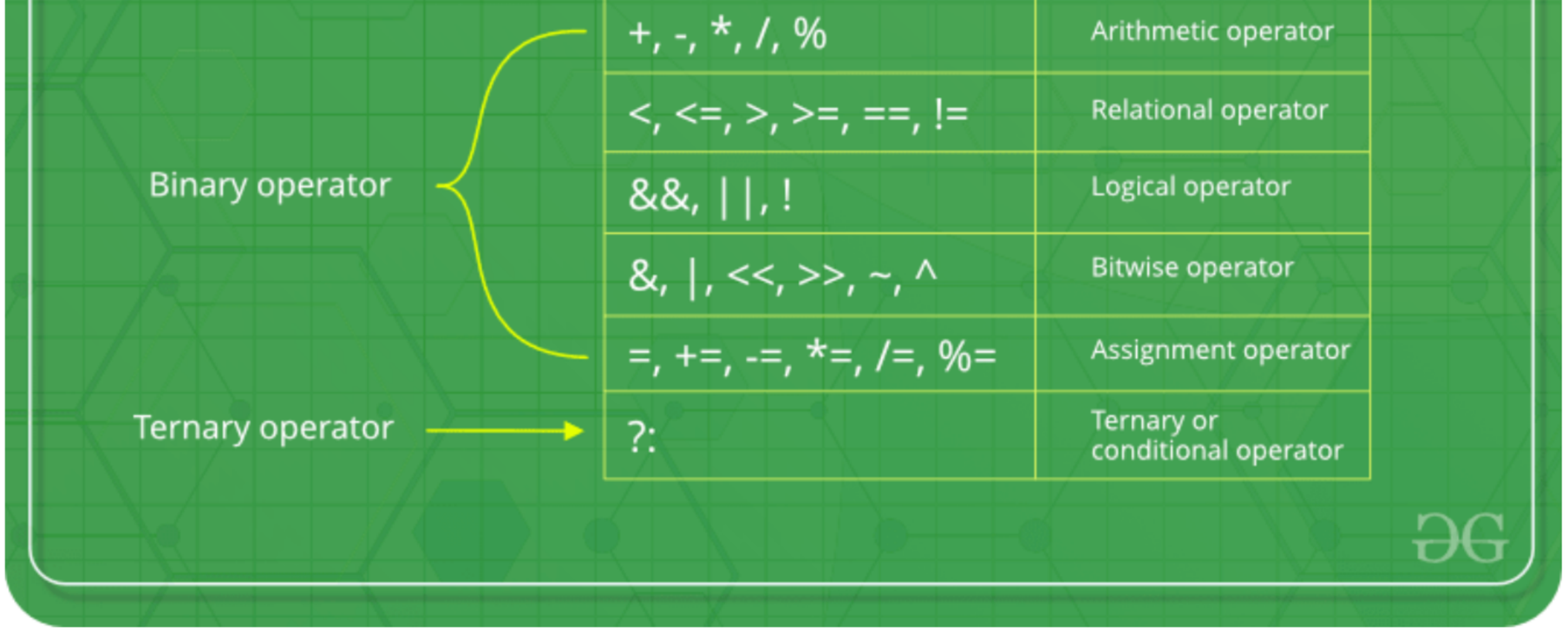
Operators in C++

Operators are the foundation of any programming language. Thus the functionality of C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands.

For example, consider the below statement:

```
c = a + b;
```

Here, '+' is the operator known as the addition operator and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b' and assign the computed value of addition to 'c'.



An operator always requires 1 or more data entities to produce computation upon known as operands. Based on the no. of operands, they are classified as **unary**, **binary** and **ternary** operators:

- *Unary* - a++, !flag
- *Binary* - a + b, a * b
- *Ternary* - (test) ? value1: value2

Based on the functionality they perform on predefined data-types, we can classify them as follows:

Assignment Operators: Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of variable on the left side otherwise the compiler will raise an error.
Different types of assignment operators are shown below:

- a = b
- a += b (a = a + b)
- a -= b (a = a - b)
- a *= b (a = a * b)
- a /= b (a = a / b)

  An example usage of the above operators is given:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int a = 12, b = 6;

    a += b;
    cout << "Add & Assign: "<< a <<"\n";

    a -= b;
    cout << "Subtract & Assign: " << a << "\n";

    a *= b;
    cout << "Multiply & Assign: " << a << "\n";

    a /= b;
    cout << "Divide & Assign: " << a << "\n";

    return 0;
}
```

Run

Output:

```
Add & Assign: 18
Subtract & Assign: 12
Multiply & Assign: 72
Divide & Assign: 12
```

- a %= b (a = a % b)

  % is the modulo operator (calculates remainder when *a* is divided by *b*). As an example:

```
int a = 10, b = 2, c = 4;
a %= b; // a = (10 % 2) = 0
a %= c;  //a = (10 % 4) = 2
```

- a &= b (a = a & b)
- a |= b (a = a | b)
- a ^= b (a = a ^ b)
- a <<= b (a = a << b)
- a >>= b (a = a >> b)

  An example usage of the above bitwise-assignments is given below:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // a: 1100, b: 0110, c: 1001, d: 0010
    int a = 12, b = 6, c = 2, d = 9, e = 1;

    a &= b;
    //   1100
    // & 0110
    //-------
```

```
14      //   0100 (4)
15      cout << "AND & Assign: "<< a <<"\n";
16
17      a |= c;
18      //   0100
19      // | 0010
20      //-------
21      //   0110 (6)
22      cout << "OR & Assign: " << a << "\n";
23
24      a ^= d;
25      //   0110
26      // ^ 1001
27      //-------
28      //   1111 (15)
29      cout << "XOR & Assign: " << a << "\n";
30
```

Run

Output:

```
AND & Assign: 4
OR & Assign: 6
XOR & Assign: 15
Left-shift & Assign: 30
Right-shift & Assign: 15
```

Increment/Decrement Operators:

- Pre-Increment: Increments/decrements the value of the operand first and then returns a reference to the modified value. **++a**, **--a**
- Post-Increment: Increment/decrement is postponed till the statement execution. Afterwards the value of the variable is modified. **a++**, **a--**

```
1
2   // C++ program to illustrate the increment/decrement
3   // operator
4   #include<iostream>
5   using namespace std;
6
7   int main()
8   {
9       int a = 5, b = 5;
10
11      cout<< ((a++) + 1)<<endl;    // prints 6
12      cout<<a<<endl;               // prints 6
13      cout<<((++b) + 1)<<endl;     // prints 7
14
15      return 0;
16  }
17
```

Run

Output:

```
6
6
7
```

In the 1st statement, we have **a** getting its value incremented afterwards the statement is evaluated, resulting in 5 + 1 = 6. Printing afterwards shows it's value changed to 6.
However, in the 3rd statement, **b** gets incremented prior to the evaluation, resulting in 6 + 1 = 7.

Arithmetic Operators: These are the operators used to perform arithmetic/mathematical operations on operands.

- +a (unary plus)
- -a (unary minus => negates the value: 5 becomes -5)
- a + b
- a - b
- a * b
- a / b
- a % b (get reminder when a is divided by b). e.g.

```
(20 % 3) = 2
```

- ~ a (bitwise NOT, reverses all the bits)

```
(~ 20) = -21
20 in binary (8-bit): 00010100
(~ 20): 11101011, which is -21 in 2's complement form
for negative integers.
```

- a & b (bitwise AND)
- a | b (bitwise OR)
- a ^ b (bitwise XOR)
- a << b (bitwise left shift a by b places).e.g.

```
(5 << 2) = 20
5 in binary: 101
left-shift by 2 places: 10100 ~ 20 (in decimal)
```

- a >> b (bitwise right shift a by b places). e.g.

```
(30 >> 3) = 3
30 in binary: 11110
right-shift by 3 places: 11 ~ 3 (in decimal)
```

Comparison Operators: Relational operators are used for comparison of the values of two operands. For example: checking if one operand is equal to the other operand or not, an operand is greater than the other operand or not etc.

- a == b
- a != b
- a < b
- a > b
- a <= b
- a >= b

```cpp

//Program to demonstrate
//comparison operators
//Output produced is 0(false) or 1(true)
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int a = 5, b = 6, c = 6;

    //check equality
    cout << "a == b: " << (a == b) << endl;

    //check inequality
    cout << "a != b: " << (a != b) << endl;

    //check less-than
    cout << "a < b: " << (a < b) << endl;

    //check greater-than
    cout << "a > b: " << (a > b) << endl;

    //check less-than-or-equal-to
    cout << "a <= c: " << (a <= c) << endl;

    //check greater-than-or-equal-to
    cout << "b >= c: " << (b >= c) << endl;

    return 0;
```

Run

Output:

```
a == b: 0
a != b: 1
a < b: 1
a > b: 0
a <= c: 1
b >= c: 1
```

Logical Operators: Logical Operators are used for combining two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a boolean value either true or false.

Below is the list of different *logical operators*:

- !a (negate boolean variable)
- (a && b), also (a and b) => Boolean AND
- (a || b), also (a or b) => Boolean OR

```cpp

//Program to demonstrate
//Logical Operators
#include <bits/stdc++.h>
using namespace std;

int main()
{
    bool a = true, b = false;

    //negate a boolean
    cout << "Negation: " << !a << endl;

    //logical AND
    cout << "AND: " << (a && b) << endl;

    //logical OR
    cout << "OR: " << (a || b) << endl;

    //Some examples using expressions
    int x = 5, y = 6, z = 5;

    //x is not equal to y, but negation yields true
    cout << !(x == y) << endl;

    //x is smaller than y, AND yields false
    //despite x==z being true
    cout << ((x > y) && (x == z)) << endl;

    //x is smaller than y, but x==z is true,
```

Run

Output:

```
Negation: 0
AND: 0
OR: 1
1
0
1
```

Member-Access Operators: Most of the operators discussed requires knowledge of pointers and structures. Hence, the detailed meanings of each of

them will be covered afterwards.

- a[b] (access b^{th} element of array *a*)
- *a (access data value at location pointed at by *a*)
- &a (storage address of variable *a*)

Below given is a program showing the usage of the above 3 operators:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int a[] = {1, 2, 3, 4, 5};
    //access 3rd element
    cout << "4th element of the array (0-indexing): "  << a[3] << endl;

    int b = 6, *p = &b;

    //access value using pointer
    cout << "Access via Pointer: " << *p << endl;
    *p = 5;
    cout << "Value changed via pointer: " << b << endl;

    //Access address of b
    cout << "Address of b: " << &b << endl;

    return 0;
}
```

Run

Output:

```
4th element of the array (0-indexing): 4
Access via Pointer: 6
Value changed via pointer: 5
Address of b: 0x7ffdc51518d4
```

- a->b (access member variable using pointer)
- a.b (access member variable using instance)
- *(a->b) (access data value of member pointer variable using pointer)
- *(a.b) (access data value of member pointer variable using instance)

```
//Following Member-Access Operators can be demonstrated
//using structures only. Please proceed to the main() part
//of the code as structure declaration will be covered later
#include <bits/stdc++.h>
using namespace std;

struct test
{
    int x;
    int *p;
};

int main()
{
    struct test t;
    struct test *ptr_t = &t;

    t.x = 5;

    int b = 10;
    t.p = &b;

    cout << "Direct Access: " << t.x << endl;
    cout << "Pointer Access: " << ptr_t->x <<endl;

    cout << "Direct Access of Pointer: " << *(t.p) << endl;
    cout << "Pointer Access of Pointer: " << *(ptr_t->p) << endl;
}
```

Run

Output:

```
Direct Access: 5
Pointer Access: 5
Direct Access of Pointer: 10
Pointer Access of Pointer: 10
```

- Miscellaneous
  - a(...) (function call)
  - a, b (comma)
  - new (allocates memory on heap)

    ```
    A *aptr = new A("Class A Instance");
    ```

  - delete (de-allocates memory on heap)

    ```
    delete aptr; //frees memory pointed to by aptr
    ```

  - sizeof (used to get size of identifier in bytes)

    ```
    int a;
    cout<< sizeof(a);   //prints 4
    ```

  - a ? b : c (ternary/conditional)

```
int value = (flag) ? value1 : value2
```

- (type) ~ For type-casting one data-type to another

```
char p = (char)(65);
cout << (double)(2);
```

Operator Precedence

Below given is the precedence table of the various operators and their corresponding associativity:

Precedence	Operator	Description	Associativity
1	a++ a-- a( ) a[ ] . ->	Post-increment/decrement Function call Array subscript Member access	Left-to-right
2	++a --a +a -a ! ~ (type) *a &a sizeof new delete	Pre-increment/decrement Unary plus & minus Logical NOT & bitwise NOT Type-casting Dereferencing Address-of sizeof Dynamic Memory Allocation Memory De-allocation	Right-to-left
3	*(a.b) *(a->b)	Pointer to member	Left-to-right
4	a*b a/b a%b	Multiplication, division & remainder	Left-to-right
5	a+b a-b	Addition & subtraction	Left-to-right
6	<< >>	Bitwise left-shift and right-shift	Left-to-right
7	< <= > >=	Relational operators < and ≤ Relational operators > and ≥	Left-to-right
8	== !=	Relational operators = and ≠	Left-to-Right
9	&	Bitwise-AND	Left-to-right
10	^	Bitwise-XOR	Left-to-right
11	|	Bitwise-OR	Left-to-right
12	&&	Logical AND	Left-to-right
13	||	Logical OR	Left-to-right
14	a?b:c = += -= *= /= %= <<= >>= &= ^= |=	Ternary/conditional Assignment Assignment (sum, difference) Assignment (multiply, divide, modulo) Assignment (left-shift, right-shift) Assignment (AND, XOR, OR)	Right-to-left
15	,	Comma	Left-to-right

Sample Problems I (Operators, Data Types, Input/Output)

The following are some basic implementation problems covering the topics discussed until now.

- **Problem 1)** Change the case of the character entered. (**using operators only**).

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    char ch1 = 'p', ch2 = 'P';

    cout << "Character p changed to: " << (char)((ch1 >= 'a' && ch1 <= 'z') ? (ch1 - 'a' + 'A') : (ch1 -
    cout << "Character P changed to: " << (char)((ch2 >= 'a' && ch2 <= 'z') ? (ch2 - 'a' + 'A') : (ch2 -

    return 0;
}

```

Run

Output:

```
Character p changed to: P
Character P changed to: p
```

In the above program, we use ternary-operator to output the character with changed case. If the character *ch* entered lies between 'a' and 'z', it means it is in lower-case, so we output: (ch - 'a' + 'A'), otherwise, we output (ch - 'A' + 'a').

- **Problem 2)** Write a program to convert temperature given in Celsius (user input) to Fahrenheit.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int cel1 = 47;
    int cel2 = 29;

    cout << "47 deg Celsius in Fahrenheit: " << (9.0/5*cel1 + 32) << endl;
    cout << "29 deg Celsius in Fahrenheit: " << (9.0/5*cel2 + 32) << endl;
    return 0;
}
```

Run

Output:

```
47 deg Celsius in Fahrenheit: 116.6
29 deg Celsius in Fahrenheit: 84.2
```

The above program is a good example of implicit type-casting in C++. Conversion of temperatures is a floating point operation. Thus, we need to have one operand as a double. i.e. 9.0 here. Otherwise, (9/5) would yield 1 and not 1.8. Once, we have converted the literal operands to a double, automatically cel1 and cel2 gets type-casted. This behaviour is also termed as **up-casting**.

- **Problem 3)** Write a program to find the area of a triangle. Take the length of sides as user input. (Area printed should be rounded off to two decimal places).

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    int a = 6, b = 8, c = 10;
    cin >> a >> b >> c;

    //We shall use heron's formula to calculate triangle area
    double s = (a + b + c)/2.0;

    double area = sqrt(s*(s-a)*(s-b)*(s-c));

    cout << setprecision(2) << fixed << area << endl;
    return 0;
}

```

Run

Output:

```
24.00
```

Above program uses Heron's formula for calculating the area of a triangle. Similar to the previous example, we here divide by 2.0 to upcast the integral lengths to yield a final double *s* value.

- **Problem 4)** Take user input amount of money and consider an infinite supply of denominations 1, 20, 50 and 100. What is the minimum number of denominations to make the change?

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int amt = 253, q, ans=0;
    //To calculate min. no. of denominations
    //we need to first divide by 100
    //then, by 50
    //then by 20
    //then the rest of the amount can be made using 1

    //add notes of 100 required
    q = amt / 100;
    ans += q;
    amt -= q*100;

    //add notes of 50 required
    q = amt/50;
    ans += q;
    amt -= q*50;

    //add notes of 20 required
    q = amt/20;
    ans += q;
    amt -= q*20;

    //add remaining amount (to be paid using Re. 1)
    ans += amt;
```

Run

Output:

```
Min. notes to get a sum of 253: 6
```

In this problem, we have been given the relaxation of having an unlimited supply of the said denominations. Hence, to make it possible with the minimum number of notes in total, we try to accommodate the sum in the decreasing order of value. i.e. We try out first with 100, then the remaining amount with 50, then 20 and finally the remaining amount can be made up using 1s.

Decision Making in C++ (if , if..else, Nested if, if-else-if)

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.

Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in C++ are:

- if statement
- if..else statements
- nested if statements
- if-else-if ladder

- switch statements

if statement

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

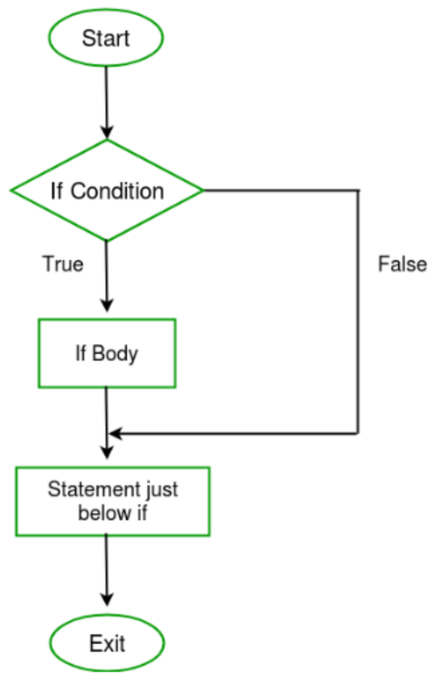
Here, **condition** after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not.

If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

Example:

```
if(condition)
   statement1;
   statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```

Flowchart:




```
1
2  // C++ program to illustrate If statement
3  #include<iostream>
4  using namespace std;
5
6      int main()
7      {
8          int i = 10;
9
10         if (i > 15)
11         {
12            cout<<"10 is less than 15";
13         }
14
15         cout<<"I am Not in if";
16     }
17
```

Run

Output:

```
I am Not in if
```

As the condition present in the if statement is false. So, the block below the if statement is not executed.

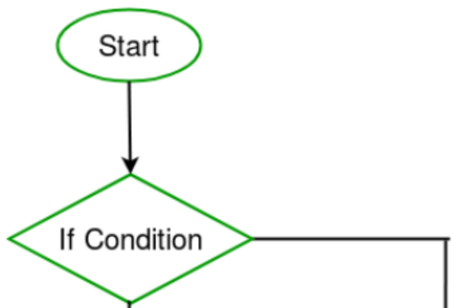
if- else

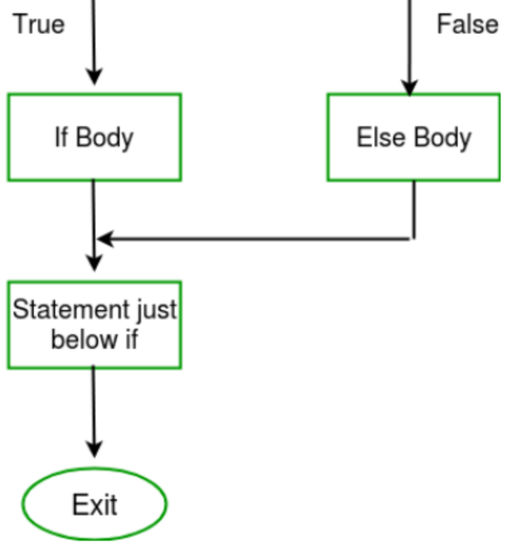
The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

Flowchart:





Example:

```
1
2  // C++ program to illustrate if-else statement
3  #include<iostream>
4  using namespace std;
5
6  int main()
7   {
8        int i = 20;
9
10       if (i < 15)
11           cout<<"i is smaller than 15";
12       else
13           cout<<"i is greater than 15";
14
15     return 0;
16 }
17
18
```

Run

Output:

```
i is greater than 15
```

The block of code following the *else* statement is executed as the condition present in the *if* statement is false.

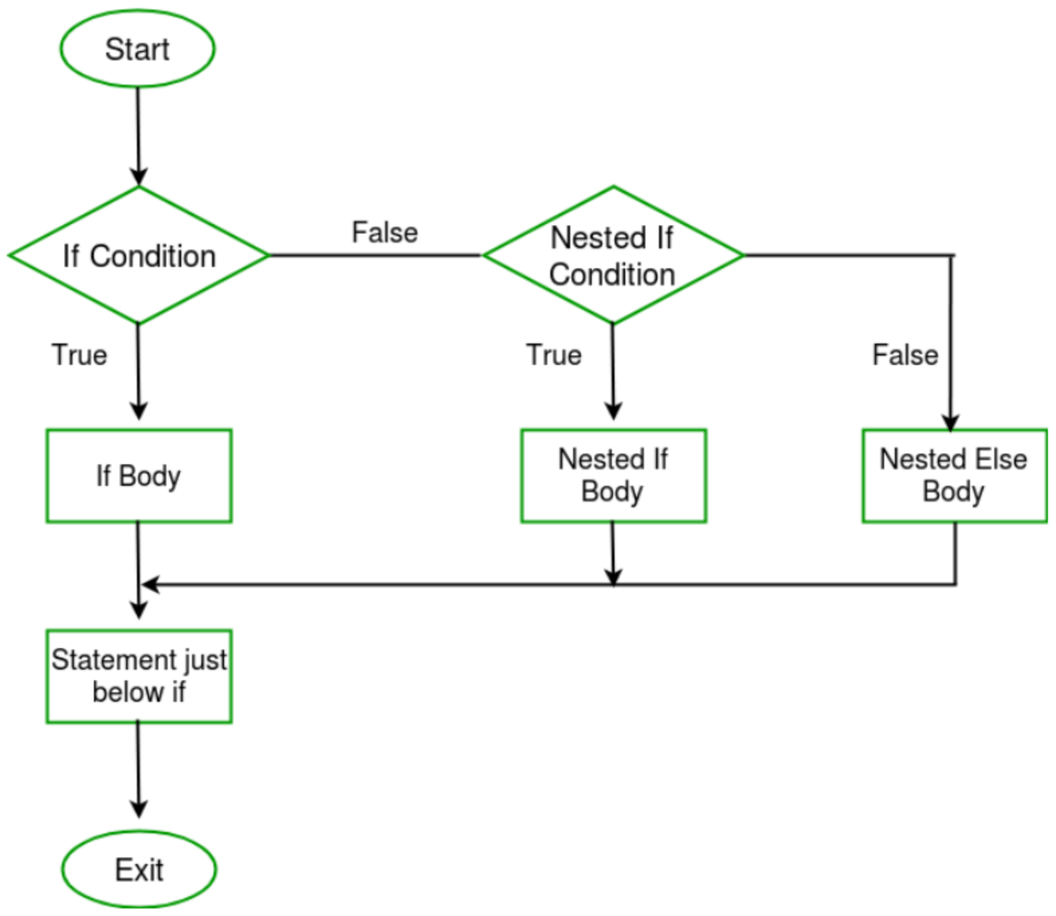
nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, C++ allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

Flowchart:



Example:

```
1
2  // C++ program to illustrate nested-if statement
3  int main()
4      {
5          int i = 10;
6
7          if (i == 10)
8          {
9              // First if statement
10             if (i < 15)
11                 cout<<"i is smaller than 15";
12
13             // Nested - if statement
14             // Will only be executed if statement above
15             // it is true
16             if (i < 12)
```

```
            cout<<"i is smaller than 12 too";
        else
            cout<<"i is greater than 15";
    }

    return 0;
}

```

Run

Output:

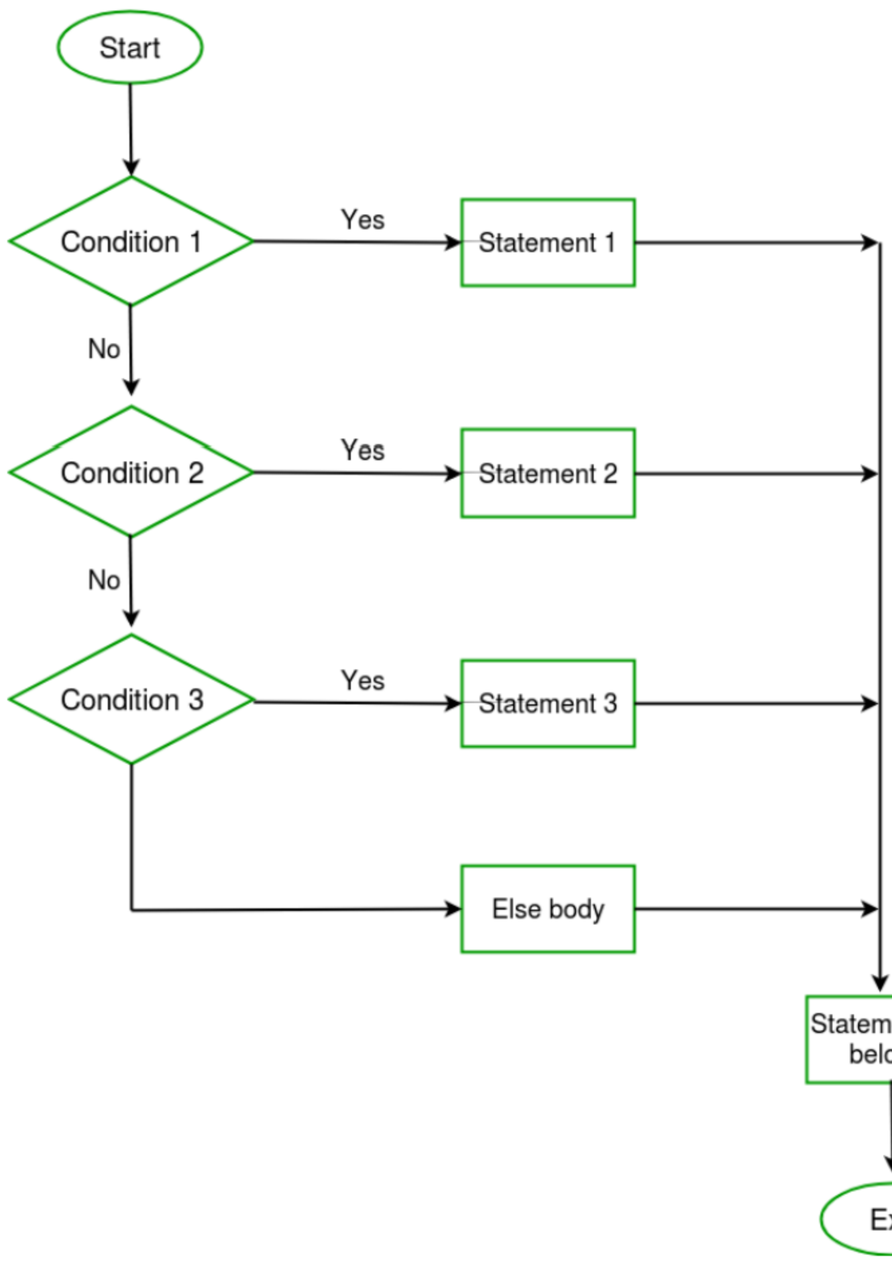
```
i is smaller than 15
i is smaller than 12 too
```

if-else-if ladder

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```



Example:

```
// C++ program to illustrate if-else-if ladder
#include<iostream>
using namespace std;

int main()
{
    int i = 20;

    if (i == 10)
        cout<<"i is 10";
    else if (i == 15)
        cout<<"i is 15";
    else if (i == 20)
        cout<<"i is 20";
    else
        cout<<"i is not present";
}

```

Run

Output:

```
i is 20
```

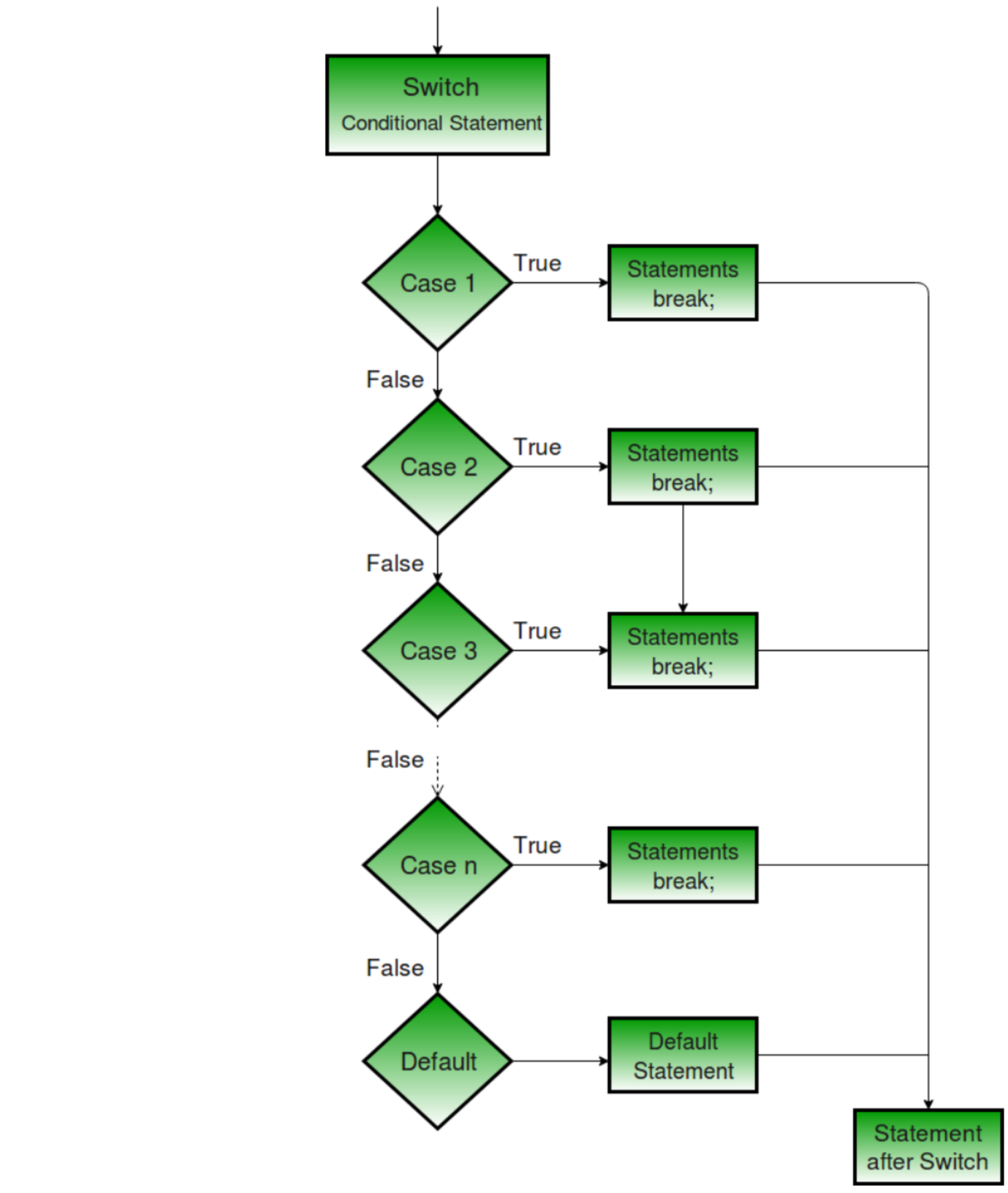
Switch-case

Switch case statements are a substitute for long if statements that compare a variable to several integral values.

- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Switch is a control statement that allows a value to change control of execution.

Syntax:

```
switch (n)
{
    case 1: // code to be executed if n = 1;
        break;
    case 2: // code to be executed if n = 2;
        break;
    default: // code to be executed if n doesn't match any cases
}
```



Example:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
   int x = 2;
   switch (x)
   {
       case 1: cout << "Choice is 1\n";
               break;
       case 2: cout << "Choice is 2\n";
                break;
       case 3: cout << "Choice is 3\n";
               break;
       default: cout << "Choice other than 1, 2 and 3\n";
                break;
   }
   return 0;
}
```

Run

Output:

```
Choice is 2
```

Loops and Jump Statements in C++

Looping Statements

Loops in programming come into use when we need to repeatedly execute a block of statements. For example: Suppose we want to print "Hello World" 10 times. This can be done in two ways as shown below:

Iterative Method

An iterative method to do this is to write the **cout** statement 10 times.

```
// C++ program to illustrate need of loops
#include <iostream>
```

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

int main()
{
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    return 0;
}
```

Run

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Using Loops

In Loop, the statement needs to be written only once and the loop will be executed 10 times as shown below.
In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

- An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.
- **Counter not Reached:** If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeat it.
- **Counter reached:** If the condition has been reached, the next instruction "falls through" to the next sequential instruction or branches outside the loop.

There are mainly two types of loops:

1. **Entry Controlled loops**: In this type of loops the test condition is tested before entering the loop body. **For Loop** and **While Loop** are entry controlled loops.
2. **Exit Controlled Loops**: In this type of loops the test condition is tested or evaluated at the end of loop body. Therefore, the loop body will execute atleast once, irrespective of whether the test condition is true or false. **do - while loop** is exit controlled loop.

for Loop

A for loop is a repetition control structure which allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.

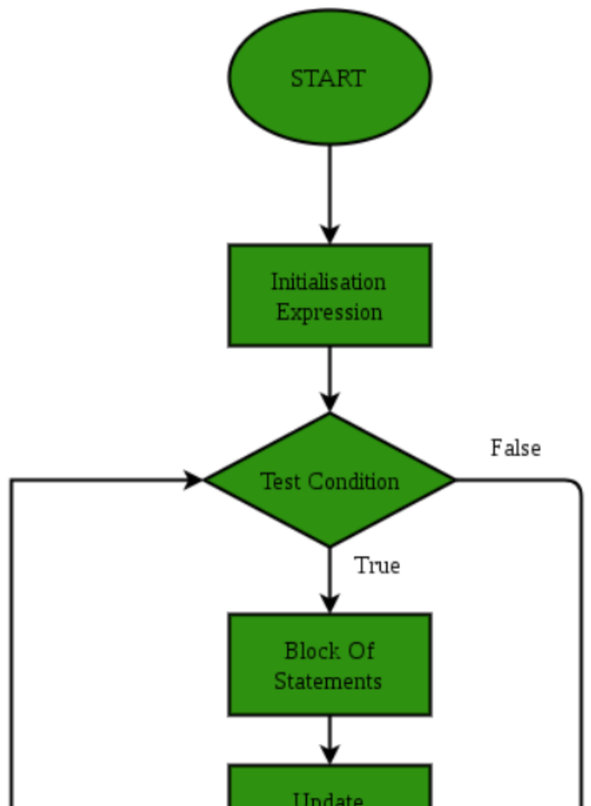
Syntax:

```
for (initialization expr; test expr; update expr)

{

     // body of the loop

     // statements we want to execute

}
```

In for loop, a loop variable is used to control the loop. First, initialize this loop variable to some value, then check whether this variable is less than or greater than counter value. If statement is true, then loop body is executed and loop variable gets updated. Steps are repeated till exit condition comes.

- **Initialization Expression**: In this expression we have to initialize the loop counter to some value. for example: int i=1;
- **Test Expression**: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: i <= 10;
- **Update Expression**: After executing loop body this expression increments/decrements the loop variable by some value. for example: i++;

Equivalent flow diagram for loop :



Example:

```cpp

// C++ program to illustrate for loop
#include <iostream>
using namespace std;

int main()
{
    for (int i = 1; i <= 10; i++)
    {
        cout << "Hello World\n";
    }

    return 0;
}

```

Run

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

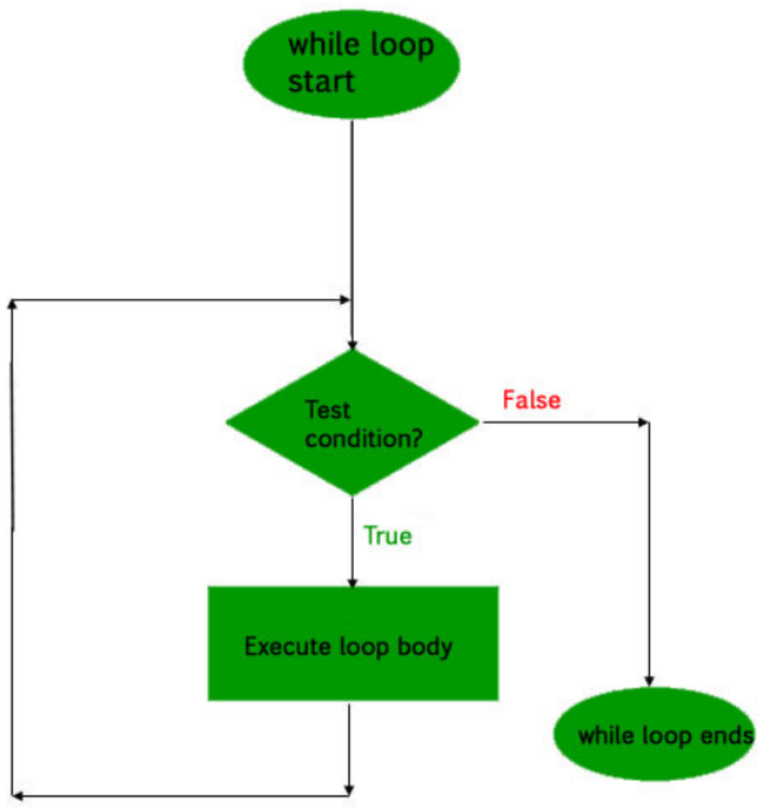
While Loop

While studying **for loop** we have seen that the number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known to us. while loops are used in situations where we do not know the exact number of iterations of loop beforehand. The loop execution is terminated on the basis of the test condition.

Syntax:
We have already stated that a loop is mainly consisted of three statements - initialization expression, test expression, update expression. The syntax of the three loops - For, while and do while mainly differs on the placement of these three statements.

```
initialization expression;
while (test_expression)

{

   // statements



  update_expression;
}
```

Flow Diagram:



Example:

```cpp

// C++ program to illustrate for loop
#include <iostream>
using namespace std;

int main()
{
    // initialization expression
    int i = 1;
```

```cpp

    // test expression
    while (i < 6)
    {
        cout << "Hello World\n";

        // update expression
        i++;
    }

    return 0;
}

```

Run

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

do while loop

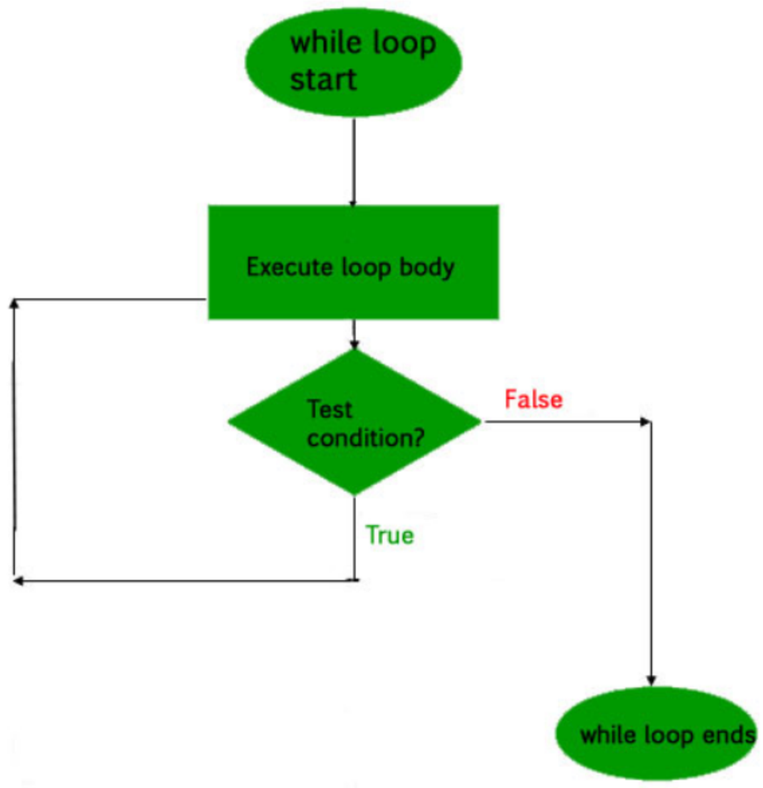
In do while loops also the loop execution is terminated on the basis of test condition. The main difference between do while loop and while loop is in do while loop the condition is tested at the end of loop body, i.e do while loop is exit controlled whereas the other two loops are entry controlled loops.
Note: In do while loop the loop body will execute at least once irrespective of test condition.

Syntax:

```
initialization expression;
do

{

   // statements



   update_expression;
} while (test_expression);
```

Note: Notice the semi - colon(";") in the end of loop.

Flow Diagram:



Example:

```cpp

// C++ program to illustrate do-while loop
#include <iostream>
using namespace std;

int main()
{
    int i = 2; // Initialization expression

    do
    {
        // loop body
        cout << "Hello World\n";

        // update expression
        i++;

    }  while (i < 1);   // test expression

    return 0;
}

```

Run

Output:

```
Hello World
```

In the above program the test condition (i<1) evaluates to false. But still as the loop is exit - controlled the loop body will execute once.

What about an Infinite Loop?

An infinite loop (sometimes called an endless loop) is a piece of coding that lacks a functional exit so that it repeats indefinitely. An infinite loop occurs when a condition always evaluates to true. Usually, this is an error.

```cpp

// C++ program to demonstrate infinite loops
// using for and while
// Uncomment the  sections to see the output

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

    // This is an infinite for loop as the condition
    // expression is blank
    for ( ; ; )
    {
        cout << "This loop will run forever.\n";
    }

    // This is an infinite for loop as the condition
    // given in while loop will keep repeating infinitely
    /*
    while (i != 0)
    {
        i-- ;
        cout << "This loop will run forever.\n";
    }
    */

    // This is an infinite for loop as the condition
    // given in while loop is "true"
```

Run

Output:

```
This loop will run forever.
This loop will run forever.
.................
```

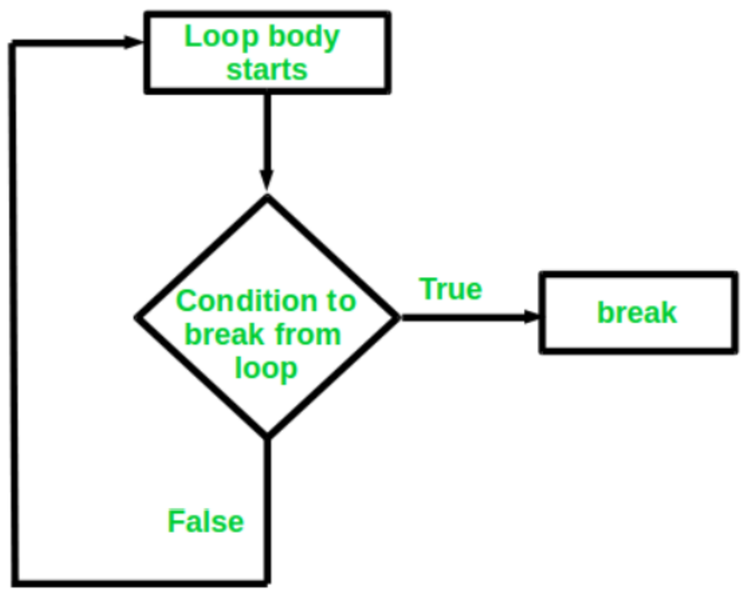
Important Points:

- Use for loop when number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known.
- Use while loops where exact number of iterations is not known but the loop termination condition is known.
- Use do while loop if the code needs to be executed at least once like in Menu driven programs

Jump Statements

Jump statements help programmers to jump directly to a point in program breaking the normal flow of execution. They provide change in program execution flow. There are 3 jump constructs in C/C++:

- **break**: Break statements are used to terminate a loop. Thus the flow jumps directly to the 1st statement after the loop upon encountering a break statement.



As an example of linear search in an array, we want to quit looking further once we found the element we desire.

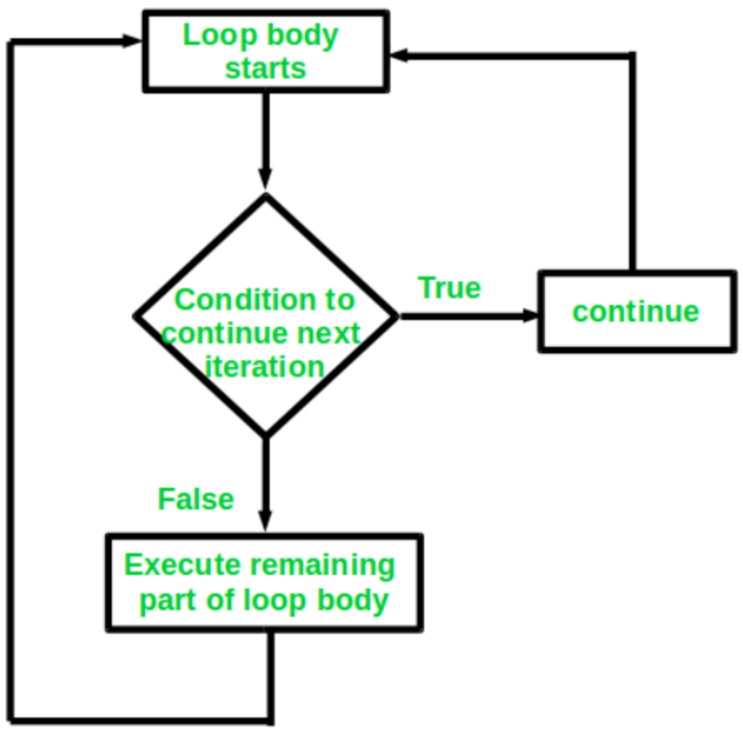
```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int key = 5;
    for (int i = 0; i < 10; i++) {
        if (arr[i] == key) {
            cout << "5 found in array";
            break;
        }
    }
    return 0;
}
```

Run

Output:

```
5 found in array
```

- **continue**: Continue statements force the execution of the next iteration of the loop disregarding the statements following it.i.e. when a continue is encountered, all the statements following are skipped and control returns to the next iteration (condition check).



As an example:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    for (int i = 1; i <= 10; i++) {
        if (i == 6) //If i equals 6, continue to next
                    //iteration without printing
            continue;
        cout << i << " ";
    }
    return 0;
}

```

Run

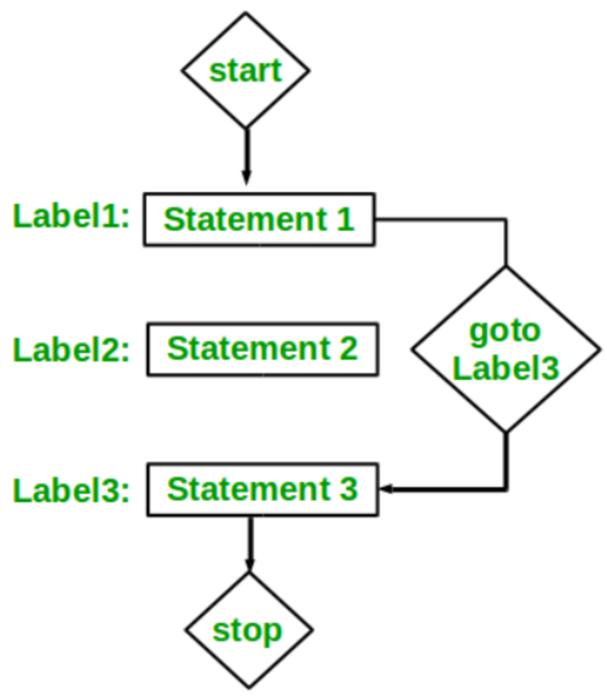
Output:

```
1 2 3 4 5 7 8 9 10
```

- **goto**: It is a statement which enables us to jump anywhere in a program. Unlike break and continue, which interrupts the flow of a loop. i.e. they have scope to jump within a loop (terminate or force next iteration), goto has the capability to reach anywhere in the program. *goto* works with the concept of labels. A label is a *named block of code*. It will be clear from the example given below. First, the syntax looks as:

```
Syntax1      |   Syntax2
--------------------------
goto label;  |    label:
.            |    .
.            |    .
.            |    .
label:       |    goto label;
```

As can be seen from the above syntax, the label can exist anywhere in the program (prior or after). Upon encountering the *goto label;* statement, flow jumps to the label unconditionally.




```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n = 1;
label:
    cout << n << " ";
    n++;
    if (n <= 5)
```

```
    goto label;
  return 0;
}
```

Run

Output:

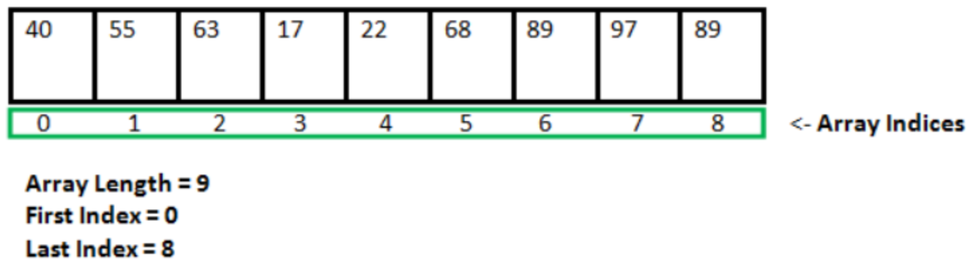
```
1 2 3 4 5
```

In the above program, each time goto is encountered, execution jumps back to *label*.
goto statements are deemed too powerful and hence are discouraged because of the following reasons:

- Makes program logic very complex (harder to debug & modify).
- Usage of goto can be easily substituted with the more safe *continue* & *break* statements.

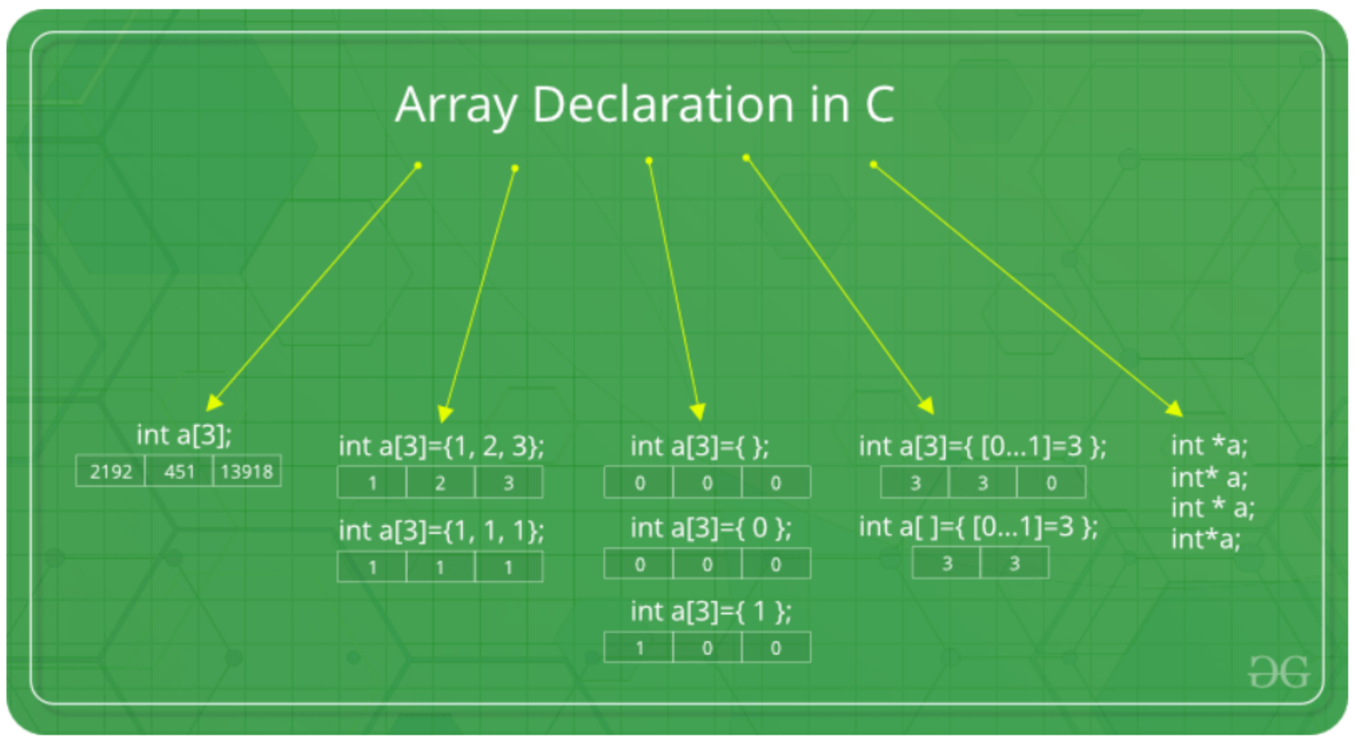
Arrays in C++

An array is collection of items stored at continuous memory locations. In Arrays, all elements stored must be of same data type. In other words, we can also define an array as a set of elements of same data type stored linearly at contigous memory locations.



Why do we need arrays? We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

Array declaration in C/C++:



We can declare an array by specifying its type and size or by initializing it or by both.

1. Array declaration by specifying size

```
int arr[10];  //array declaration with specific size

int n;
cin >> n;
int arr[n];  //declare array after user-input

```

2. Array declaration by initializing elements

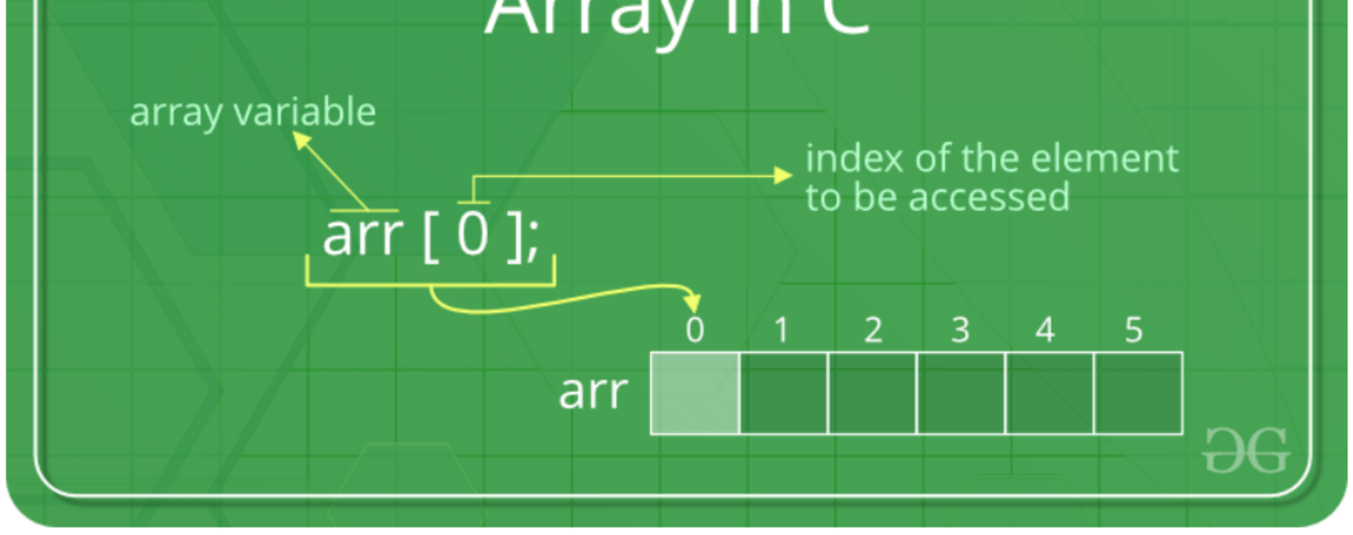
```
// Array declaration by initializing elements
int arr[] = { 10, 20, 30, 40 }

// Compiler creates an array of size 4.
// above is same as  "int arr[4] = {10, 20, 30, 40}"

```

3. Array declaration by specifying size and initializing elements

```

// Array declaration by specifying size and initializing
// elements
int arr[6] = { 10, 20, 30, 40 }

// Compiler creates an array of size 6, initializes first
// 4 elements as specified by user and rest two elements as 0.
// above is same as  "int arr[] = {10, 20, 30, 40, 0, 0}"

```

Facts about Array in C/C++:

- **Accessing Array Elements:** Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.



The following are a few examples.

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  int main()
6  {
7      int arr[5];
8      arr[0] = 5;
9      arr[2] = -10;
10     arr[3 / 2] = 2; // this is same as arr[1] = 2
11     arr[3] = arr[0];
12
13     cout << arr[0] << arr[1] << arr[2] << arr[3];
14
15     return 0;
16 }
17
```

Run

Output:

```
5 2 -10 5
```

- **No Index Out of bound Checking:** There is no index out of bound checking in C/C++, for example the following program compiles fine but may produce unexpected output when run.

```
1
2  // This C++ program compiles fine
3  // as index out of bound
4  // is not checked in C++.
5  #include <bits/stdc++.h>
6  using namespace std;
7
8  int main()
9  {
10     int arr[2];
11     cout << arr[3] << " " << arr[-2] << " ";
12     return 0;
13 }
14
```

Run

Output:

```
2008101287 4195777
```

- Contiguous Allocation in Arrays:

Below program demonstrates the fact that array elements are allocated in a contiguous fashion.

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4  int main()
5  {
6      // an array of 10 integers.  If arr[0] is stored at
7      // address x, then arr[1] is stored at x + sizeof(int)
8      // arr[2] is stored at x + sizeof(int) + sizeof(int)
9      // and so on.
10     int arr[5], i;
11
12     cout <<"Size of int in this compiler is "<<sizeof(int)<<endl;
13
14     for (i = 0; i < 5; i++)
15         // The use of '&' before a variable name, yields
16         // address of variable.
17         cout <<"Address arr[" << i << "] is "<< &arr[i] <<endl;
18
19     return 0;
20 }
21
```

Run

Output:

```
Size of integer in this compiler is 4
Address arr[0] is 0x7ffd636b4260
Address arr[1] is 0x7ffd636b4264
Address arr[2] is 0x7ffd636b4268
Address arr[3] is 0x7ffd636b426c
Address arr[4] is 0x7ffd636b4270
```

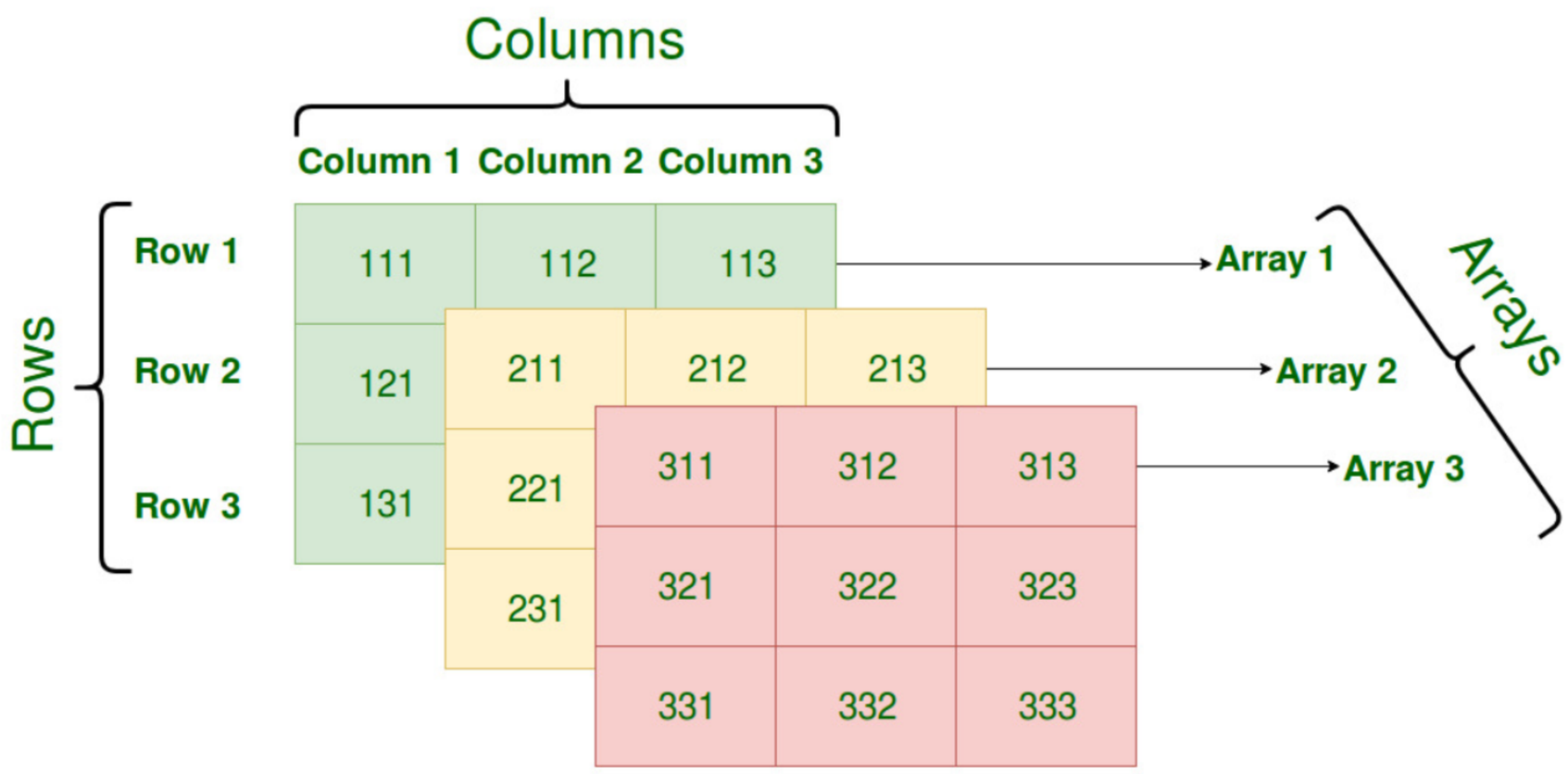
Till now, we have studied about Single-dimensional arrays only. We can create multi-dimensional arrays as well.

Multi-dimensional Arrays

```
//creates (size1 * size2) type 2-D matrix
int arr[size1][size2]

//creates (size1 * size2 * size3) type 2-D matrix
int arr[size1][size2][size3]
...
We can continue with as many dimensions as we like:
// (size1 * size2 * ... *size_n) matrix
int arr[size1][size2][size3]....[size_n]
```

A 2-d array and a 3-d array can be visualized as:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]



Initialization by directly assigning elements:

```
int arr[2][3] = { {0, 1, 2}, {3, 4, 5} };
int arr[2][3][4] =
 {
   { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} },
   { {12, 13, 14, 15}, {16, 17, 18, 19}, {20, 21, 22, 23} }
 };
```

To access a particular element, we do as:

```
arr[i][j];
arr[i][j][k];
```

Strings in C++

There are mainly 2 ways of handling strings in C++. One is to use the native **C-style** strings or character arrays and the other is to use the **string template class** (available as part of C++ Standard Template Library).

C-style (character arrays and literals)

C-style strings are defined using character arrays. There is an extra character called the **null (\0)** character appended to the character array to mark it's end.

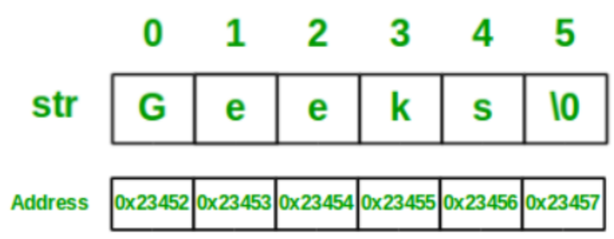
Syntax:

```
char str_name[size];
```

Declaration & Initialization of string

```
char str[size];
char str[] = "Geeks";
char str[5] = "Geeks";
char str[] = {'G','e','e','k','s'};
```

Below is the memory representation of a string "Geeks".



Input & Output C-style strings

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    // Declared a string, str of max size 10
    char str[10];

    // Taking input from user
    cin>>str;

    // Printing the string
    cout<<str<<endl;

    return 0;
}

```

```
Input: Hello
Output: Hello
```

There are some important C-style in-built string functions (standard provided) given below which are generally used for string manipulation:

Note: These functions are available in the header file ***cstring***. Therefore, one must include the line **#include< cstring >** inorder to use these funcitons.

- **strlen**: Finds the length of the string. This is especially useful when looping over each character in the string. As an example, in the below program we iterate over each character and convert them to uppercase.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    char str[]="hello";
    for (int i=0;i<strlen(str);i++)
        cout<<(char)(str[i]-'a'+'A');
    return 0;
}

```

Run

Output:

```
HELLO
```

- **strcat**: Appends a copy of the source string to the end of destination string. Thus it takes two arguments, first is the destination, then source.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    char src[] = "source";
    char dest[] = "destination";

    cout<<strcat(dest,src)<<endl;
    return 0;
}

```

Run

Output:

```
destinationsource
```

- **strcmp**: Performs lexicographical comparison between 2 strings. The function returns an integer.
  - **= 0** (if both strings are equal)
  - **> 0** (if 1st string appears after 2nd string in lexicographical order)
  - **< 0** (if 1st string appears before 2nd string in lexicographical order

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    char str1[]="abc";
    char str2[]="abcd";
    char str3[]="abb";

    cout<<strcmp(str1,str2)<<endl;
    cout<<strcmp(str1,str3)<<endl;
    cout<<strcmp(str2,str3)<<endl;

    return 0;
}

```

Run

Output:

```
-100   <0 indicating str1 appears before str2
1      >0 indicating str1 appears after str3
1      >0 indicating str2 appears after str3
```

- **strcpy**: Copies source string to the destination string. Here also the 1st parameter is the destination string and the 2nd is the source string.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    char dest[100];
    char src[]="Hello World";
    strcpy(dest, src);
    cout<<"Source: "<<src<<endl;
        cout<<"Destination: "<<dest<<endl;

    return 0;
}

```

Run

Output:

```
Source: Hello World
Destination: Hello World
```

- **strstr**: Used for string searching. Finds and returns pointer to the first occurence of one string in another. Here also the 2nd string is searched in the1st string.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    char s1[] = "GeeksforGeeks";
    char s2[] = "for";
    char *p;

    p = strstr(s1, s2);

    if (p)
        cout << "First occurrence of string '" << s2
        << "'\n in '" << s1 << "' is '" << p <<"'"<<endl;
    else
        cout << "String not found\n";

    return 0;
}

```

Run

Output:

```
First occurrence of string 'for'
 in 'GeeksforGeeks' is 'forGeeks'
```

String Class in C++ STL

C++ STL (Standard Template Class) provides a more powerful string class implementation. string template implementation is better than native C-style character arrays in the following aspects.

C-style character array	C++ string class
Array of characters terminated by null (\0)	Class object instance storing stream of characters
Allocated Statically (can't be modified in run-time)	Allocated Dynamically (modifiable at run-time)
Vulnerable to array decay	Class-based implementation evades such vulnerabilities
Limited inbuilt functions for manipulation	Rich inbuilt function support for manipulation

Syntax: Declaring strings using STL class is simpler than declaring character arrays.

```
string str_name;
```

Example:

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    // Declaring String
    string str;

    // Taking input from user
    cin>>str;

    // Printing the String
    cout<<str<<endl;

    cout<<str.length()<<endl;

    return 0;
}

```

```
Input: Hello
Output: Hello
        5
```

We can also declare and initialize a string at same time:

```
// Both of the below methods works fine:
```

```
Method 1:
string str = "This is a sample string"; // This is correct

Method 2:
string str;
str = "This is a sample string";  // This is also correct
```

Below are some of the most frequently used functions available in the std::string class which can be used for string manipulations. We need to understand an ***iterator*** first (in brief) to understand each of these string-class methods.

An **iterator** is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them. Example usage of an iterator to print out the string is given below.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    string s = "GeeksforGeeks";

    //declaration of iterator
    string::iterator it;

    //Loop over the string using iterator
    for (it = s.begin(); it != s.end(); it++)
        cout << *it;
    cout <<endl;

    return 0;
}

```

Run

Output:

```
GeeksforGeeks
```

Some of the most-used methods of the string class with example usage is given below:

- **begin**: Returns an iterator to the first character of the string.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    string s = "GeeksforGeeks";
    cout << "Print 1st character: " << *s.begin() << endl;

    return 0;
}

```

Run

Output:

```
Print 1st character: G
```

- **end**: Returns an iterator to the character following the last character of the string. This character acts as a placeholder and accessing it results in undefined behaviour.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    string s = "GeeksforGeeks";

    //s.end() access is invalid, so we
    //do as s.end() - 1, to get the iterator
    //pointing at the last character
    cout << *(s.end() - 1) << endl;

    return 0;
}

```

Run

Output:

```
s
```

- **size**: Returns the no. of characters in the string. Same as using length().
- **clear**: Removes all the characters from the string, thus making it an empty string ("")
- **empty**: Returns a boolean indicating whether string is empty.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    string s = "GeeksforGeeks";

    cout << "Length of String: " << s.size() << endl;
```

```
    //Clear out the given string
    s.clear();

    //Check if empty
    if (s.empty())
        cout << "String is empty\n";

    return 0;
}
```

Run

Output:

```
Length of String: 13
String is empty
```

- **insert**: Insert another string at a certain position. insert takes 2 arguments (position, string-to-insert) as shown in the example below.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    string s1 = "Hello Everyone";
    string s2 = "and Good Day to ";

    //Insert s2 at index 6 of s1
    s1.insert(6, s2);

    cout << s1 << endl;

    return 0;
}
```

Run

Output:

```
Hello and Good Day to Everyone
```

- **push_back**: Append a character to end of string
- **pop_back**: Remove a character from the end.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    string str = "abc";

    //Append character 'd' to end
    str.push_back('d');

    cout << str << endl;

    //Remove last character
    str.pop_back();

    cout << str << endl;

    return 0;
}
```

Run

Output:

```
abcd
abc
```

- **find**: Find occurence of one string in another. If string is found, the position of occurence is returned, else string::npos is returned (indicating not-found).

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    string s1 = "GeeksforGeeks";
    string s2 = "for";
    string s3 = "abc";

    //Returns 5 as 'for' is found at
    //index 5 in s1
    cout << s1.find(s2) << endl;

    //s3 is not present in s1
    //so it return npos
    if (s1.find(s3) == string::npos)
        cout << "String '" << s3 << "' not found\n";

    return 0;
}
```

Run

Output:

```
5
String 'abc' not found
```

- **substr**: Generate substring from string. It takes two arguments (position, no. of characters). Generates a substring from [position:position+num_characters-1]. However, the 2nd argument is optional, meaning if not provided substring is generated till end-of string.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    string s = "GeeksforGeeks";

    //1st argument is: position
    //2nd argument is: no. of characters
    cout << s.substr(5, 3) << endl;

    //If 2nd argument is not provided, substring
    // is generated from position to end to string
    cout << s.substr(5) << endl;

    return 0;
}

```

Run

Output:

```
for
forGeeks
```

Sample Problems II (Decision, Loops, Arrays and Strings)

The following are some basic implementation problems covering the topics discussed until now.

- **Problem 1)** Count the number of digits of a number.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    unsigned long long int x = 255657434267;

    int num_digits = 0;

    do {
        num_digits++;
        x /= 10;
    }
    while (x);

    cout << num_digits << endl;

    return 0;
}

```

Run

Output:

```
12
```

We have used the do-while loop for the particular problem because we should still increment the count when the user inputs 0. If we had used while loop, then the loop would break immediately. (*while(0) ~ while(false)*). Inside the loop statement, we increment count by 1 and successively divide by 10. By the time *x* becomes 0, *num_digits* equals the number of digits.

- **Problem 2)** Write a program simulating a calculator (supporting +,-,*,/). It should continuously ask for 2 numbers and the particular operation to perform (as a character), and produce the output accordingly. The program should quit if the user enters the character 'Q'.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    //opr ~ operation to perform
    //op1 ~ operand 1
    //op2 ~ operand 2
    //OUTPUT: (op1 opr op2)
    char opr;
    int op1, op2;

    while (true) {

        cin >> opr;

        if (opr == 'Q')
            break;
        else {
            cin >> op1 >> op2;
            switch(opr) {
                case '+':
                    cout << op1 + op2 << endl;
```

```
            break;
        case '-':
            cout << op1 - op2 << endl;
            break;
        case '*':
            cout << op1 * op2 << endl;
```

Input

```
+ 4 5
* 2 3
Q
- 5 4
```

Output

```
9
6
```

Explanation: The above code combines the concepts of while loop and switch-case. *while(true)* allows us to run the program indefinitely, asking for continuous user input. Once, the user enters an operator, we fall-back to the if-part and further ask the operands, then we perform the calculation and print the result, otherwise, if the user enters the character 'Q', then the infinite-loop breaks, quitting the calculator program. As the sample input/output shows, we perform *4+5* and *2*3*, but when we hit *Q*, we quit and skip the calculation of *5-4*.

- **Problem 3)** Take as input n numbers and find the average.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    //the number of array elements
    int n = 6;
    int arr[] = {2,3,1,5,6,8};

    int sum = 0;

    for (int i=0; i<n; i++)
        sum += arr[i];

    cout << 1.0*sum/n;

    return 0;
}

```

Run

Output:

```
4.16667
```

In the above program, we take as input *n* numbers and store them in an array to find the average. Note however that average can be a floating-point value. Thus while printing the output we need to type-cast it to a double, which we do as *1.0*sum/n*. This implicitly converts to double after multiplying by 1.0.

- **Problem 4)** Check if a string entered is a palindrome or not. (A palindrome is a string which is the same as the original when reversed)

```cpp

#include <bits/stdc++.h>
using namespace std;

void check_palindrome(string s)
{
    int len = s.length();
    for (int i=0; i<len; i++)
        if (s[i] != s[len-i-1]) {
            cout << "string " + s + " is not a palindrome\n";
            return;
        }

    cout << "string " + s + " is a palindrome\n";
}

int main()
{
    string s1 = "kayak", s2 = "kappa";

    check_palindrome(s1);
    check_palindrome(s2);

    return 0;
}

```

Run

Output:

```
string kayak is a palindrome
string kappa is not a palindrome
```

For palindrome checking, we need to check whether 1st character and last character match, then 2nd character and 2nd to last character and so on. Generalising, the i^{th} character should match the (len-i-1)th character (0-indexing), where *len* is the length of the string.

- **Problem 5)** Convert decimal number input by the user to binary.

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    long long x = 29;

```

```
    string bin;

    while (x) {
        bin += (char)((x & 1) + '0');
        x >>= 1;
    }

    reverse(bin.begin(), bin.end());

    cout << bin << endl;

    return 0;
}
```

Run

Output:

```
11101
```

The above program is a bit complicated hence needs explanation. Since the Binary form of a number is represented by 0s and 1s, hence for a large decimal number (having 10-15 digits), it might not be even accommodable by *unsigned long long int*. Hence, we use *string* to store the answer. The expression *(x & 1)* finds out whether the remainder when it is divisible by 2. If yes, the remainder is 0, otherwise 1. We append this value after adding ASCII '0' to the binary string and typecasting to char. At each step, we further divide x by 2 by right-shift-1. Finally, we reverse the binary string to get the desired result.

Functions in C++

A function is a set of statements that take inputs, do some specific computation and produce output.

The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can call the function. The syntax for a function is as follows:

```
< return-type > < function-name > (< set-of-arguments >) {
    //block of statements
}
```

As an example:

```
#include <bits/stdc++.h>
using namespace std;

int max(int x, int y)
{
    if (x > y)
      return x;
    else
      return y;
}

int main()
{
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);

    cout << "Maximum of a &amp;amp; b is: " << m);
    return 0;
}
```

Run

Output:

```
Maximum of a & b is: 20
```

Note: Regarding the return-type, every function needs to return a value matching the type if return-type is declared anything other than *void*. In case of void return type, no return statement is required.

Prototype Declaration & Definition

Sometimes, we just want to declare the existence of a function and provide the implementation afterwards. We can do so by using the prototype-declaration scheme, the syntax of which is given below:

```
< return-type > < function-name > (< set-of-arguments >);
```

As an example:

```
int max(int a, int b);
int min(int, int);   // we don't even have to name
                     // our argument variables
```

Once declared, we can define (provide implementation) anywhere in the program. (or even in another file, if the function is declared *extern*).

Formal & Actual Parameters

The parameters passed to function are called **actual** parameters. For example, in the *first program* 10 and 20 are actual parameters. The parameters received by function are called **formal** parameters. For example, in the above program x and y are formal parameters.

Pass by value & Pass by Reference

There are two types of passing arguments to functions. The arguments declared within the function (formal parameter variables) have function block scope only. Hence, any manipulation done on them has no effect on the actual variables with which the function was called. This will be clear with the following **Pass by value** example:

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int a = 5, b = 6;
    swap(a, b);
    cout << "a: " << a << ", b: " << b <<endl;
    return 0;
}

```

Run

Output:

```
a: 5, b: 6
```

As we can see, the values inside of main don't get swapped. This is because they were passed by value. Hence, any change done inside the function isn't reflected in the main() part. This is because in pass-by-value, the value and not the actual memory location gets copied to the formal parameters in the function.

Pass by Reference on the other hand manipulates with the memory location of actual parameters, thus reflecting the changes made inside the function to the actual variables. There are again 2 ways to pass-by-reference in C++, using:

- *Native C pointers* - We shall cover this when we discuss pointers.
- *C++ References* - Reference is the C++ way of passing arguments by reference. The syntax to use is as follows.

```
int swap(int &a, int &b) { ... }
//add & before the parameters (call-by-reference)
...
int a = 5, b = 6;
swap(a, b);  //calling the function remains the same
```

The same above program using call-by-reference yields output as:

```cpp

#include <bits/stdc++.h>
using namespace std;

void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int a = 5, b = 6;
    swap(a, b);
    cout << "a: " << a << ", b: " << b <<endl;
    return 0;
}

```

Run

Output:

```
a: 6, b: 5
```

So, the elements get swapped in real.

C++ Advanced Function Topics

Inline Functions

Inline functions are an improvement provided by C++ to reduce the overhead caused in executing a function call. A function call statement is a jump statement that instructs the program counter to switch to a different address for execution. In a **function call**, a lot of work that the OS needs to be done such as **storing the current address of execution (to return back to), along with other registers. Also, allocation and de-allocation (upon exit) of variables local to the functions are done**. This can greatly hamper performance if a function is called repeatedly in a program. e.g. frequently called utility functions such as min(), max(), or in case of recursive functions such as factorial().

Inline functions reduce this overhead by **replacing the function definition at the place of the function call**. i.e. inline-functions are expanded at the call location itself during compilation. This removes the function call overhead at the cost of a larger compilation code. As an example:

```cpp

#include <bits/stdc++.h>
using namespace std;

inline int max_int(int x, int y) { return (x > y) ? x : y; }

int main()
{
    cout << max_int(5, 6) << endl;
    cout << max_int(8, 7) << endl;

    return 0;
}

```

Run

Output:

```
6
8
```

When the program compiles, the definition is expanded as:

```
cout << (5 > 6) ? 5 : 6 << endl;
cout << (8 > 7) ? 8 : 7 << endl;
```

If there are multiple calls to *max_int* function, usage of inline can greatly improve performance. However, there is an added downside of increased compilation code size, because of inline-expansion. Thus, it is advised to **make only those functions inline** which are **short & frequently used**.

Default Arguments

A **Default Argument** is a default value for a function argument. In case the user forgets to provide the parameter, the default value for that variable would be used. As an example:

```cpp

#include <bits/stdc++.h>
using namespace std;

int add(int x, int y=0)
{
    return x+y;
}

int main()
{
    cout << add(5, 6) << endl;
    cout << add(5) << endl;
    return 0;
}

```

Run

Output:

```
11
5
```

There is an important rule in declaring default parameters: No non-default (mandatory) argument can follow a default argument. The direct consequence of this rule is that all default arguments must follow non-default ones:

```
int add(int y=10, int x);   //not allowed
int add(int x, int y=10);   //allowed
```

as a call to a function like *add(, 3) is not allowed* in C++. Rather *add(3)* is valid. So, all default params should follow the non-default ones.

Function Overloading

Function overloading allows us to create functions with different names, as long as they have different parameters. As an example:

```cpp

#include <bits/stdc++.h>
using namespace std;

int add(int x, int y) { return x + y; }
string add(string x, string y) { return x + y; }

int main()
{
    //adds the 2 numbers
    cout << add(5, 6) << endl;

    //concatenates the 2 strings
    cout << add("Hello ", "World") << endl;
    return 0;
}

```

Run

Output:

```
11
Hello World
```

There are some important rules for overloading functions:

- **Return-Type is not a factor for uniqueness**: Return-Type plays no role in resolving ambiguity of overloaded functions. Thus, two functions declared exactly the same with only the return-type different will generate a compilation error.

```cpp

int add(int, int);
double add(int, int);
//generates Compilation Error

```

- **Argument Names are not a factor for uniqueness**: Similar to above, declaring argument names differently with their types same also doesn't resolve ambiguity.

```cpp

int add(int a, int b);
double add(int c, int d);
//generates Compilation Error

```

Function Pointers

Functions are an executable block of code which is placed at a certain address when a program is loaded into memory. e.g. If we execute the code below:

```cpp

#include <bits/stdc++.h>
using namespace std;

int add(int x, int y) { return x + y; }

int main()
{
    cout << add << endl;
    return 0;
}
```

```
12
```

Output:

```
0x002717f0
```

On author's machine, the above address gets printed. It may vary depending upon implementation. However, the crux of the matter to understand is that upon printing the function name only without paranthesis. i.e. *add*, instead of *add()*, we get the address to the function. Function Pointers is a facility provided in C/C++ to declare pointers to functions. The syntax for declaration is as follows:

```
< return-type > (* < function-name >)(< argument-list >);
```

Example:

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  int add(int x, int y) { return x + y; }
6  string add(string x, string y) { return x + y; }
7
8  int main()
9  {
10     //Assign integer add() function to funtion pointer
11     int (*fun_ptr_int)(int, int) = add;
12
13     //Assign string add() function to function pointer
14     string (*fun_ptr_str)(string, string) = add;
15
16     //Calling using function pointer
17     cout << fun_ptr_int(5, 6) << endl;
18
19     cout << fun_ptr_str("Hello ", "World") << endl;
20
21     return 0;
22 }
23
```

Run

Output:

```
11
Hello World
```

NOTE: The return-type and argument signature of the function must exactly match the function pointer declaration. e.g. Following are some valid and invalid assignments:

```
1
2  int fun1();
3  double fun2();
4  int fun3(int);
5
6  int (*fun_ptr)() = fun1; //valid
7  fun_ptr = fun2;  //invalid (return-type mismatch)
8  fun_ptr = fun3;  //invalid (argument list mismatch)
9
```

A best case example of using function pointers is when we need to pass function as an argument to another function. Consider a sorting function which we want to sort numbers both ascending and descending order, given 2 seperate functions for comparing numbers. We can do so by using function pointers as:

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  //3rd argument is a function pointer
6  void sort(int a[], int n, bool (*compare)(int, int))
7  {
8      int i,j;
9      for (i=0; i<n-1; i++)
10         for (j=0; j<n-i-1; j++)
11             if (!compare(a[j], a[j+1]))
12                 swap(a[j], a[j+1]);
13 }
14
15 bool asc(int x, int y) { return x < y; }
16 bool desc(int x, int y) { return x > y; }
17
18 int main()
19 {
20     int a[] = {0, 2, 5, 6, 9, 1, 3, 4, 8, 7};
21
22     //pass ascending function
23     sort(a, 10, asc);
24
25     for (int i=0; i<10; i++)
26         cout << a[i] << " ";
27     cout << endl;
28
29     //pass descending function
30     sort(a, 10, desc);
```

Run

Output:

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

Variable Arguments

One might have seen functions such as printf, scanf which are capable of printing/scanning any number of input arguments provided. This is achieved by using variable arguments. The advantage of having a variable argument feature in C++ is to have functions which can operate on any number of arguments and produce the result. e.g. Suppose we want a function that can take the average of any number of integers passed:

```cpp
#include <bits/stdc++.h>
using namespace std;

double average(int num, ...) {
    va_list args;
    double sum = 0;

    va_start(args, num);
    for (int i = 0; i < num; i++)
        sum += va_arg(args, double);
    va_end(args);

    return sum / num;
}

int main()
{
    cout << average (3, 1, 2, 3) <<endl;
    cout << average (5, 1, 2, 3, 4, 5) <<endl;

    return 0;
}
```

Run

Output:

```
6.95312e-310
4.16905e-310
```

Some of the important syntax and keywords used in the code:

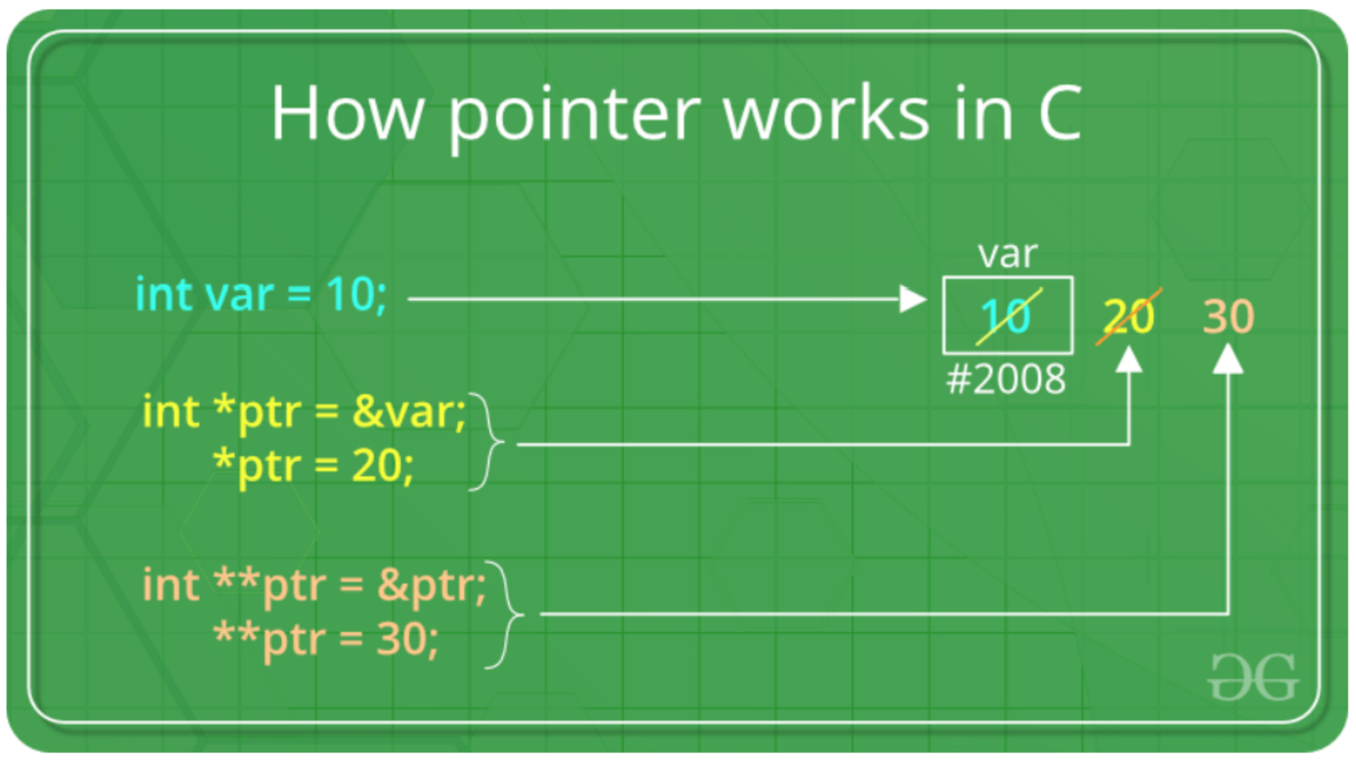
- **Usage of ... in function argument**: This spread operator is the indication of declaring the function as accepting variable arguments.
- **va_list**: Stores the list of variable arguments recieved.
- **va_arg**: Retrieves the next value in the va_list with the type passed as the parameter.
- **va_end**: Clean up the argument list.

Pointers & References in C++

Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. It's general declaration in C/C++ has the format:

Syntax:

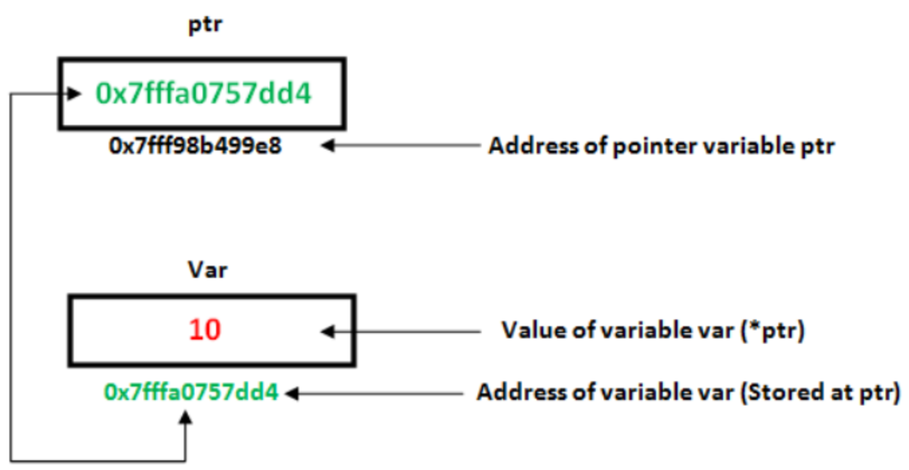
```
datatype *var_name;

int *ptr;   //ptr can point to an address which holds int data
```



How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type to a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of data type to which it points.



```cpp
// C++ program to illustrate Pointers in C++

#include <bits/stdc++.h>
using namespace std;
void geeks()
{
    int var = 20;

    // declare pointer variable
    int* ptr;

    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at var = " << var << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";
}

// Driver program
int main()
{
    geeks();
}
```

Run

Output:

```
Value at ptr = 0x7ffdcf42ad9c
Value at var = 20
Value at *ptr = 20
```

References in C++

C++ References is a new language construct which was introduced to reduce the dependency over pointers for doing indirect memory access. Usage of pointers requires caution as the programmer needs to manually deallocate them to avoid memory leaks. A reference is an alternative name for a memory location. References are useful in following situations -

1. Modify the passed parameters in a function

```cpp
#include <bits/stdc++.h>
using namespace std;

void swap (int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}

int main()
{
    int a = 2, b = 3;
    swap(a, b);
    cout << a << " " << b;
    return 0;
}
```

Run

Output

```
3 2
```

2. Avoiding copy of large structures

```cpp
struct Student {
   string name;
   string address;
   int rollNo;
}

void print(const Student &s)
{
    cout << s.name << "  " << s.address << "  " << s.rollNo;
}
```

3. In For-Each Loops to modify all objects

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> vect{ 10, 20, 30, 40 };

    for (int &x : vect)
        x = x + 5;

    for (int x : vect)
       cout << x << " ";

    return 0;
}
```

Run

References vs Pointers

1. A pointer can be declared as VOID but a reference can't. It must refer to some existing variable/memory location. Also, references can't be assigned NULL unlike pointers.
2. References are less powerful than pointers
   - Once a Reference is created, it can't be made to refer another variable. With pointers we can do so.
   - A reference must be initialized. Pointers can be declared without initialization.

Types of Function Call (References v/s Pointers)

There are 3 ways to pass C++ arguments to a function:

- call-by-value
- call-by-reference with pointer argument
- call-by-reference with reference argument

```cpp

// C++ program to illustrate call-by-methods in C++

#include <bits/stdc++.h>
using namespace std;

// Pass-by-Value
int square1(int n)
{
    // Address of n in square1() is not the same as n1 in main()
    cout << "address of n1 in square1(): " << &n << "\n";

    // clone modified inside the function
    n *= n;
    return n;
}

// Pass-by-Reference with Pointer Arguments
void square2(int* n)
{
    // Address of n in square2() is the same as n2 in main()
    cout << "address of n2 in square2(): " << n << "\n";

    // Explicit de-referencing to get the value pointed-to
    *n *= *n;
}

// Pass-by-Reference with Reference Arguments
void square3(int& n)
{
```

Run

Output:

```
address of n1 in main(): 0x7ffec6732e4c
address of n1 in square1(): 0x7ffec6732e2c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7ffec6732e50
address of n2 in square2(): 0x7ffec6732e50
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7ffec6732e54
address of n3 in square3(): 0x7ffec6732e54
Square of n3: 64
Change reflected in n3: 64
```

In C++, by default arguments are passed by value and the changes made in the called function will not reflect in the passed variable. The changes are made into a clone made by the called function.
If wish to modify the original copy directly (especially in passing huge object or array) and/or avoid the overhead of cloning, we use pass-by-reference. Pass-by-Reference with Reference Arguments does not require any clumsy syntax for referencing and dereferencing.

Array Name as Pointers

An array name contains the address of first element of the array which acts like constant pointer. It means, the address stored in array name can't be changed.
For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

```cpp

// C++ program to illustrate Array Name as Pointers in C++
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
    // Declare an array
    int val[3] = { 5, 10, 20 };

    // declare pointer variable
    int* ptr;

    // Assign the address of val[0] to ptr
    // We can use ptr=&val[0];(both are same)
    ptr = val;
    cout << "Elements of the array are: ";
    cout << ptr[0] << " " << ptr[1] << " " << ptr[2];
}

// Driver program
int main()
{
    geeks();
}
```

Run

Output:

```
Elements of the array are: 5 10 20
```

If pointer ptr is sent to a function as an argument, the array val can be accessed in a similar fashion.

Pointers and String literals

String literals are arrays containing null-terminated character sequences. String literals are arrays of type character plus terminating null-character, with each of the elements being of type const char (as characters of string can't be modified).

```
const char * ptr = "geek";
```

This declares an array with the literal representation for "geek", and then a pointer to its first element is assigned to ptr. If we imagine that "geek" is stored at the memory locations that start at address 1800, we can represent the previous declaration as:

'g'	'e'	'e'	'k'	'\0'
1800	1801	1802	1803	1804

As pointers and arrays behave in the same way in expressions, ptr can be used to access the characters of string literal. For example:

```
char x = *(ptr+3);

char y = ptr[3];
```

Here, both x and y contain k stored at 1803 (1800+3).

Pointers to pointers

In C++, we can create a pointer to a pointer that in turn may point to data or other pointer. The syntax simply requires the unary operator (*) for each level of indirection while declaring the pointer.

```
char a;

char *b;

char ** c;

a = 'g';

b = &a;

c = &b;
```

Here b points to a char that stores 'g' and c points to the pointer b.

Void Pointers

This is a special type of pointer available in C++ which represents absence of type. void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).
This means that void pointers have great flexibility as it can point to any data type. There is payoff for this flexibility. These pointers cannot be directly dereferenced. They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.

```
// C++ program to illustrate Void Pointer in C++
#include <bits/stdc++.h>
using namespace std;

void increase(void* data, int ptrsize)
{
    if (ptrsize == sizeof(char)) {
        char* ptrchar;

        // Typecast data to a char pointer
        ptrchar = (char*)data;

        // Increase the char stored at *ptrchar by 1
        (*ptrchar)++;
        cout << "*data points to a char"
             << "\n";
    }
    else if (ptrsize == sizeof(int)) {
        int* ptrint;

        // Typecast data to a int pointer
        ptrint = (int*)data;

        // Increase the int stored at *ptrchar by 1
        (*ptrint)++;
        cout << "*data points to an int"
             << "\n";
    }
}
```

Run

Output:

```
*data points to a char
The new value of c is: y
*data points to an int
The new value of i is: 11
```

Invalid pointers

A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers. Uninitialized pointers are also invalid pointers.

```
int *ptr1;

int arr[10];

int *ptr2 = arr+20;
```

Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer.
(Note: invalid pointers do not necessarily raise compile errors)

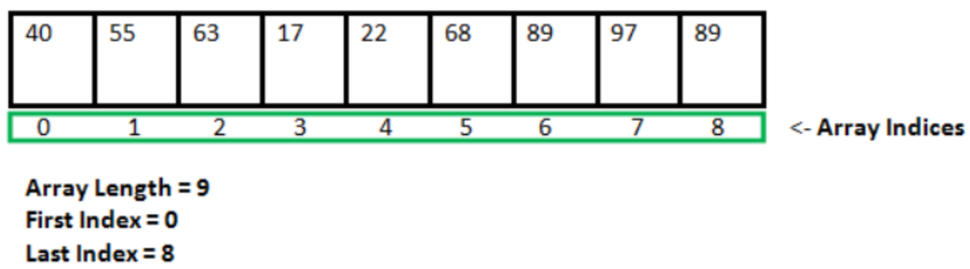
NULL Pointers

Null pointer is a pointer which point nowhere and not just an invalid address.
Following are 2 methods to assign a pointer as NULL;

```
int *ptr1 = 0;

int *ptr2 = NULL;
```

Dynamic Memory Allocation in C++

Since C++ is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.



As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example, if there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.

Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred as **Dynamic Memory Allocation**. Dynamically allocated variables are given storage in **Heap** space instead of normal **Stack** space. Understanding heap and stack space require knowledge of memory-layout of programs, but we will still try to address that at the end of the post. However, for now, we stick to learning the syntax and usage of the functions for dynamic memory allocation.

Therefore, **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C++ provides some functions to achieve these tasks. There are 4 library functions provided by C++ defined under header file to facilitate dynamic memory allocation. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

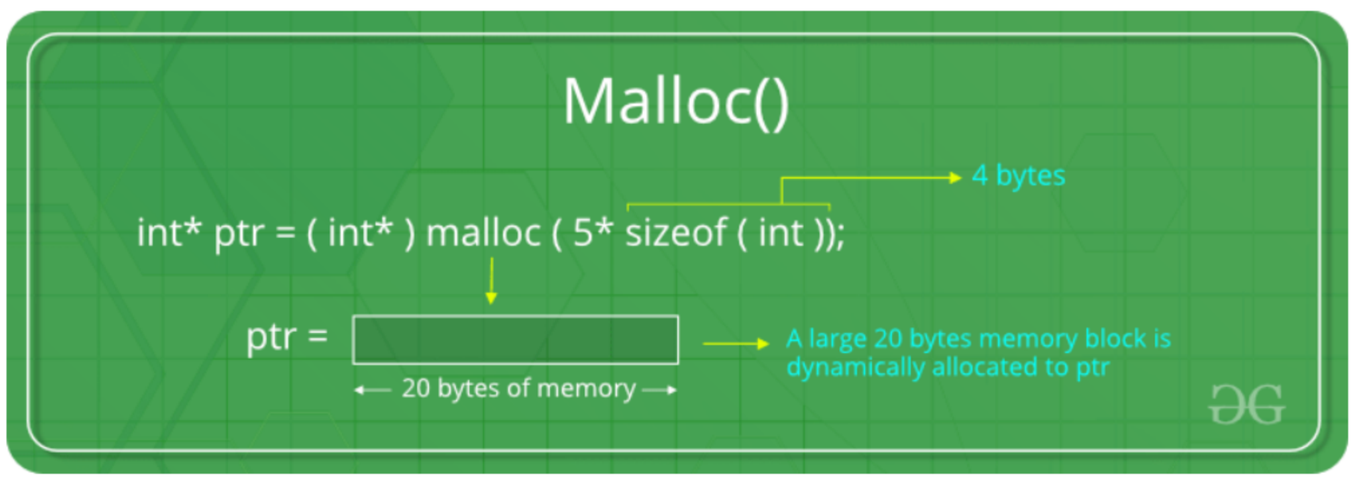
Let's see each of them in detail.

malloc()

"malloc" or **"memory allocation"** method is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

Syntax:

```
ptr = (cast-type*) malloc(byte-size)

For Example:
ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes,
this statement will allocate 400 bytes of memory.
And, the pointer ptr holds the address of the
first byte in the allocated memory.
```



If space is insufficient, allocation fails and returns a NULL pointer.

Example:

```cpp

#include <bits/stdc++.h>  //includes automatically cstdlib
using namespace std;

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr;
    int n, i, sum = 0;

    // Get the number of elements for the array
    n = 5;
    cout << "Enter number of elements: " << n <<endl;

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        cout << "Memory not allocated" << endl;
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        cout << "Memory successfully allocated using malloc" << endl;

```

Run

Output:

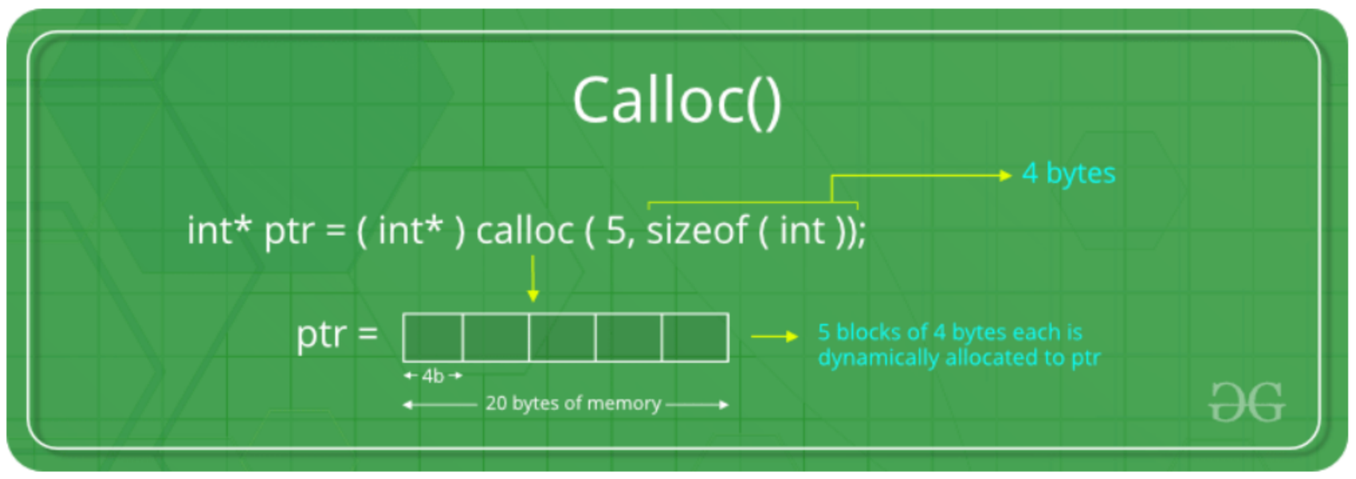
```
Enter number of elements: 5
Memory successfully allocated using malloc
The elements of the array are: 1, 2, 3, 4, 5,
```

calloc()

"calloc" or **"contiguous allocation"** method is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.

Syntax:

```
ptr = (cast-type*)calloc(n, element-size);

For Example:
ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for
25 elements each with the size of float.
```



If space is insufficient, allocation fails and returns a NULL pointer.

Example:

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr;
    int n, i, sum = 0;

    // Get the number of elements for the array
    n = 5;
    cout << "Enter number of elements: " << n << endl;

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
```

```
    exit(0);
}
else {

    // Memory has been successfully allocated
    cout << "Memory successfully allocated using calloc << endl;

```

Run

Output:

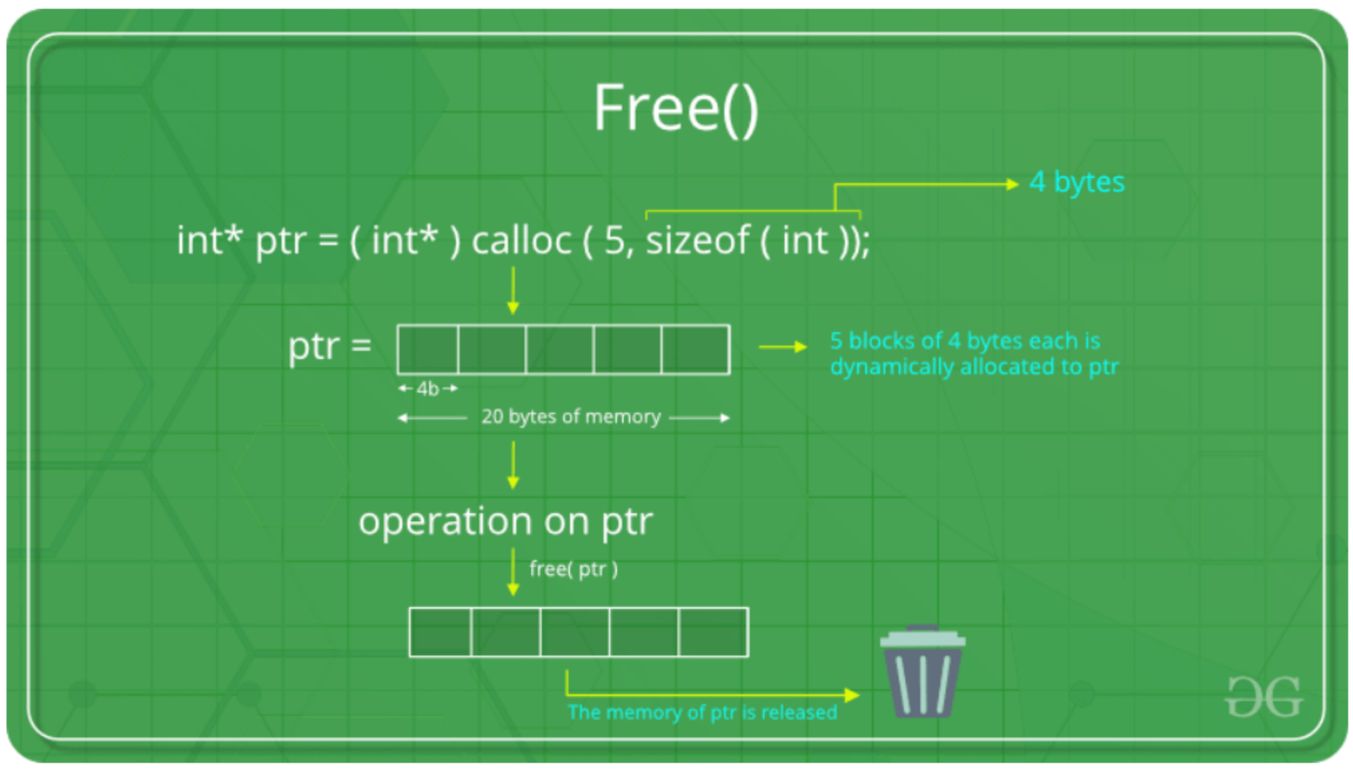
```
Enter number of elements: 5
Memory successfully allocated using calloc
The elements of the array are: 1, 2, 3, 4, 5,
```

free()

"free" method is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```



Example:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i, sum = 0;

    // Get the number of elements for the array
    n = 5;
    cout << "Enter number of elements: " << n << endl;

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        cout << "Memory not allocated" << endl;
        exit(0);
    }
    else {

```

Run

Output:

```
Enter number of elements: 5
Memory successfully allocated using malloc
Malloc Memory successfully freed.

Memory successfully allocated using calloc
Calloc Memory successfully freed.
```

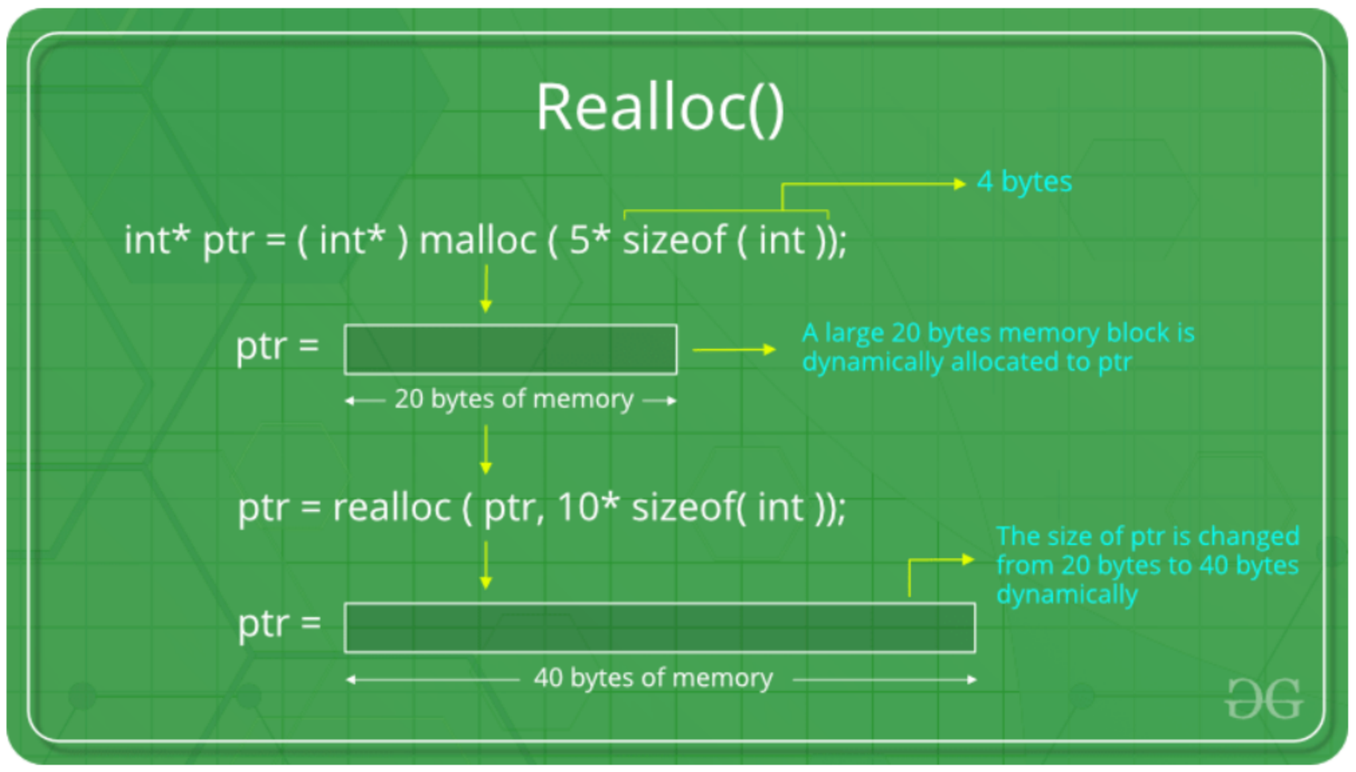
realloc()

"realloc" or **"re-allocation"** method is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the

memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.

Syntax:

```
ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.
```



If the space is insufficient, allocation fails and returns a NULL pointer.

Example:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr;
    int n, i, sum = 0;

    // Get the number of elements for the array
    n = 5;
    cout << "Enter number of elements: " << n << endl;

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        cout << "Memory not allocated" << endl;
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        cout << "Memory successfully allocated using calloc" << endl;

```

Run

Output:

```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

new & delete (C++ operators)

C++ provides two extra keywords called *new* and *delete* for dynamic memory. Syntax for these operators is given below:

```
= new
```

As an example:

```cpp
int *p = new int;
int *p = new int(10);  //initialize value 10 altogether
int *p = new int[10];  //declare an array of integers on heap

```

delete is used to de-allocate the memory pointed by the pointer variable:

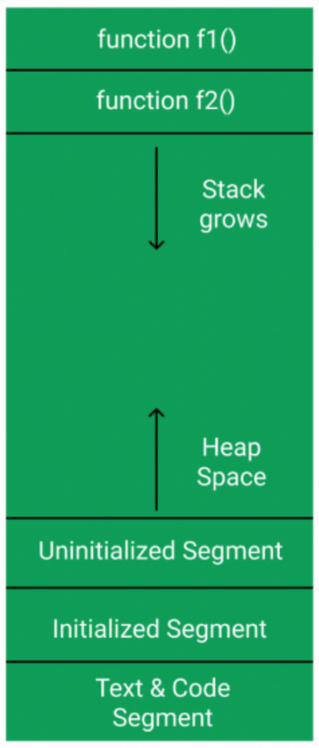
```
delete
```

As an example:

```cpp
delete p;   //de-allocate pointer
delete[] p; //de-allocate pointer to an array

```

Heap Space vs Stack Space

Heap Space v/s Stack Space



Above figure is a rough representation of the memory layout of a C/C++ program. We can see that **stack** space grows from top towards bottom. Rest of the available space belongs to the **Heap**. We see stack space being marked as *f1()*, *f2()*. This signifies the allocation of memory during the execution of the respective functions. Once the functions exit (return), the allocated stack space also gets removed. Thus, all the data (variables allocated inside the function) gets lost.

On the contrary, variables allocated on Heap doesn't belong to any function and they thus have global scope. Thus, they exist as long as the program keeps running. Hence, a variable allocated on the heap is unaffected by the exit of the function where they have been declared.

NOTE: All dynamically allocated variables are placed in the heap space and thus they have program-lifetime. Some key differences between Stack and Heap are given below:

1. In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
2. Memory shortage problem is likely to happen in Stack compared to Heap
3. Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.

Sample Problems III (Functions and Pointers)

The following are some basic implementation problems covering the topics discussed until now.

- **Problem 1)** Write a program to swap two numbers.

```cpp
#include <bits/stdc++.h>
using namespace std;

void swap(int &x, int &y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

int main()
{
    int a = 5, b = 6;

    swap(a, b);

    cout << "Value of a: " << a << endl;
    cout << "Value of b: " << b << endl;

    return 0;
}
```

Run

Output:

```
Value of a: 6
Value of b: 5
```

In the above program, we use C++ references and functions for swapping. When we pass a C++ reference to the function, it becomes *call-by-reference*, hence references of the actual variables get swapped thus reflecting the change is *main()* also.

- **Problem 2)** Print a 2-D matrix, given the pointer to the 1st element and the matrix dimensions.

```cpp
#include <bits/stdc++.h>
using namespace std;

void print_matrix(int *p, int n, int m)
{
    int i,j;

    for (i=0;i<n;i++) {
        for (j=0;j<m;j++)
            cout << *(p + i*m + j) << " ";
        cout << endl;
    }
}

int main()
{
    int a[][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
```

```
    print_matrix((int*)a, 3, 4);

    return 0;
}

```

Run

Output:

```
1 2 3 4
5 6 7 8
9 10 11 12
```

In the above example, we print a 2-D matrix passed as a pointer to the function *print_matrix*. Since *p* is the pointer to the 1st element of the matrix, we can access the j^{th} element of the i^{th} row by shifting *p* by *i*m* times (i.e. each row contains *m* elements, so to reach row *i*, we need to move *i*m* times), then, to get to the j^{th} element, we add *j* to *p*.

Void & NULL/nullptr pointers in C++

VOID Pointers

A void pointer is a pointer that doesn't have any type and thus can point to any data-type by explicit casting. Since the data-type this pointer refers to isn't defined upon declaration, we can't dereference a void pointer.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int x = 5;
    float f = 6.5;
    string s = "Hello World";

    void *p;

    p = &x;  //valid
    p = &f;  //valid
    p = &s;  //valid

    cout << *p << endl;
    cout << *(static_cast<string*>(p)) << endl;
}

```

Run

Generates the following error:

```
prog.cpp: In function 'int main()':
prog.cpp:17:14: error: 'void*' is not a pointer-to-object type
     cout << *p << endl;
             ^
```

All the assignment statements are valid because a void pointer can be made to point to any data-type. However, the above code doesn't compile because of the line:
*cout << *p << endl*. The reason being, we tried to de-reference a void pointer without casting. The subsequent line prints the string: "Hello World" successfully.

NULL Pointers

A pointer declared a value always contains garbage value, thus pointing to a random location in memory. To signify a pointer as not pointing to any meaningful variable, we use the **NULL** Macro. NULL is a Macro defined in standard library, and it's value is generally implementation specific, but most compilers set it to value: 0. Thus, assigning NULL to a pointer, and putting it in an if-statement generates boolean-false (because of 0):

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int *p = NULL;

    cout << p << endl;

    if (p)
        cout << "Pointer is not NULL\n";
    else
        cout << "Pointer is NULL\n";
}

```

Run

Output:

```
0
Pointer is NULL
```

Since NULL is a MACRO defined to have 0 value, it poses a problem in ambiguity issues of overloaded functions:

```
#include <bits/stdc++.h>
using namespace std;

void fun(int x) { cout << "Integer Call: " << x << endl; }
void fun(int *x) { cout << "Pointer Call: " << x << endl; }

int main()
{
    int *p = NULL;
```

```
11
12      fun(0);     //Integer Call: 0
13      fun(p);     //Pointer Call: 0
14      fun(NULL);    //ambiguity issue
15      return 0;
16  }
17
```

Run

Generates the following error:

```
prog.cpp: In function 'int main()':
prog.cpp:13:10: error: call of overloaded 'fun(NULL)' is ambiguous
  fun(NULL); //ambiguity issue
          ^
prog.cpp:4:6: note: candidate: void fun(int)
 void fun(int x) { cout << "Integer Call: " << x << endl; }
      ^
prog.cpp:5:6: note: candidate: void fun(int*)
 void fun(int *x) { cout << "Pointer Call: " << x << endl; }
      ^
```

Since NULL is 0, it may mean int as well as int*, causing ambiguity. To fix this, C++ introduced **nullptr** pointer-literal keyword. In modern C++ usage, thus we set a pointer to **nullptr instead of NULL**, when we declare a pointer pointing to nothing. Same above code with the call: *fun(nullptr)* will not generated any errors as ambiguity is removed.

User-defined Data Types in C++

In this post, we introduce you to user-defined data-types namely, **structure**, **union** & **enumeration** data types. We have classes too in C++, however it requires prior knowledge of object-oriented principles to comprehend them.

Structures Structures are user-defined constructs that can be built by encapsulating predefined data-types (int, char, float, double, strings, arrays etc. and even structures) together. The syntax to declare a structure is as follows:

```
struct  {
    //data members
}
```

Usage example of structure:

```
1
2   #include <bits/stdc++.h>
3   using namespace std;
4
5   struct Employee {
6       string name; // name of person
7       int age; // age of person
8       double salary; // in lakhs
9       bool remote; // whether remote or not
10  };
11
12  int main()
13  {
14      // Initialization-cum-Declaration
15      Employee e = { "David", 24, 2.5, true };
16
17      cout << "Name: " << e.name << endl;
18      cout << "Age: " << e.age << endl;
19      cout << "Salary: " << e.salary << endl;
20      cout << "Is remote?: " << e.remote << endl; // Member-access
21
22      Employee* eptr = &e;
23
24      // Access using structure pointer
25      cout << "Pointer Access: " << (eptr->name) << endl;
26
27      Employee office[10]; // array of employees
28
29      return 0;
30  }
```

Run

Output:

```
Name: David
Age: 24
Salary: 2.5
Is remote?: 1
Pointer Access: David
```

NOTE: One might guess that the size of a structure variable equals the summation of size of its data members. However, it is far from true, the reason being the compiler adds some padding to deal with alignment issues. e.g.

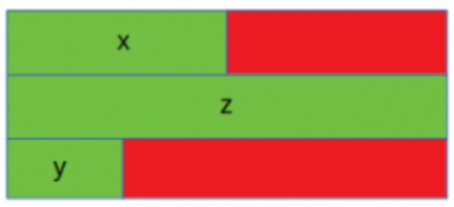
```
1
2   #include <bits/stdc++.h>
3   using namespace std;
4
5   struct A {
6       int x; // size of int is 4 bytes
7       double z; // size of double is 8 bytes
8       short int y; // size of short int is 2 bytes
9   };
10
11  int main()
12  {
13
14      cout << sizeof(struct A) << endl;
15
16      return 0;
17  }
18
```

Run

Output:

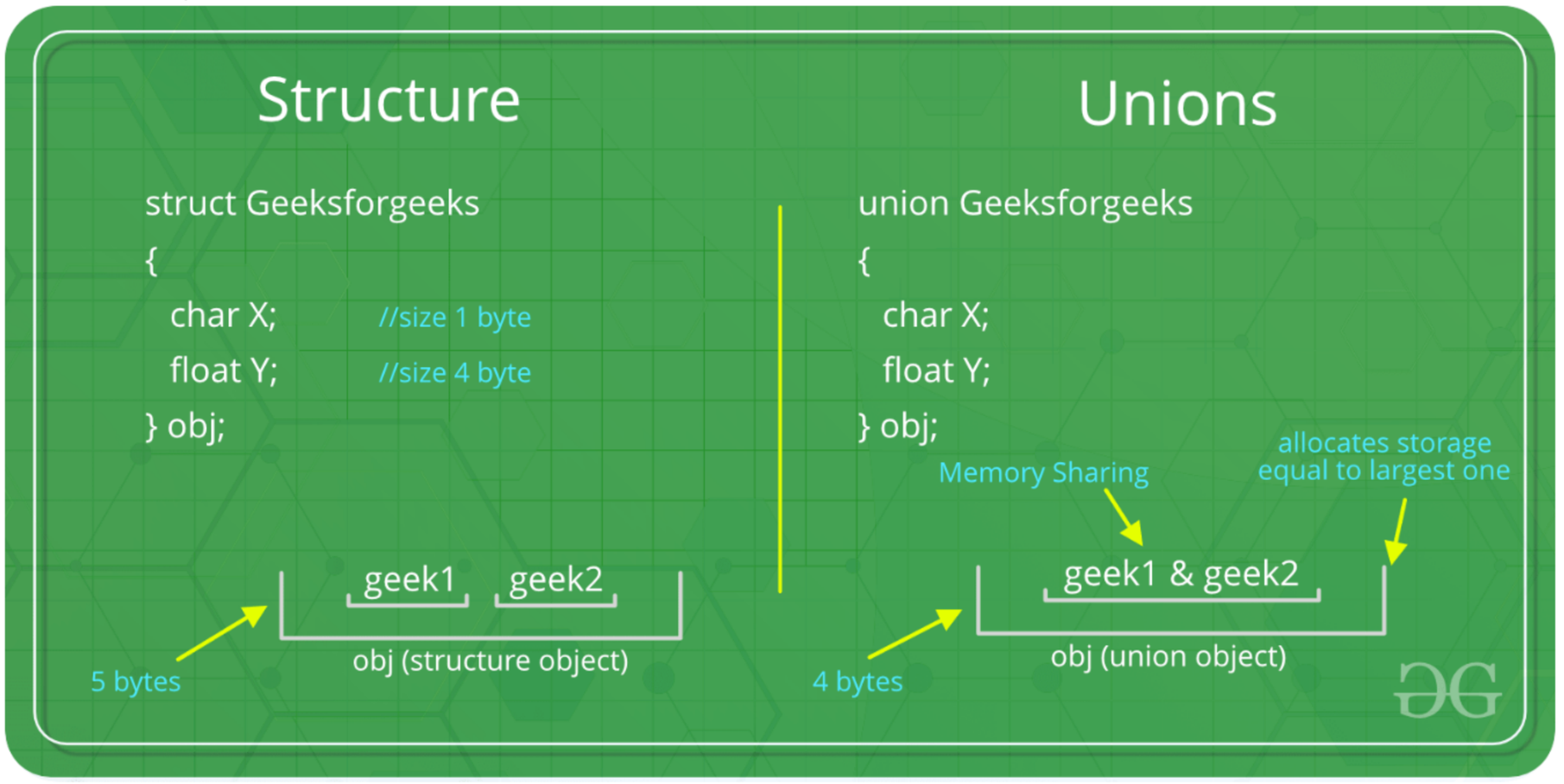
```
24
```

The reason being the extra padding added shown below:



There are more to structures in C++ (as compared to C). Structures in C++ are very much similar to classes. (we can define constructor, methods, access modifiers etc. and perform all object-oriented operations on them). We shall discuss object-oriented features of C++ structures once we discuss classes.

Unions Unions are also similar to structures, however instead of having seperate memory locations for each data member, there is a single storage location shared by all the members:




```
#include <bits/stdc++.h>
using namespace std;

union test {
    int x, y;
};

int main()
{
    test t;
    t.x = 2;
    cout << t.x << " " << t.y << endl;
    t.y = 3;
    cout << t.x << " " << t.y << endl;

    return 0;
}

```

Run

Output:

```
2 2
3 3
```

We see both x and y get updated with the value 2.
NOTE: Size of a union variable equals the size of the largest data member.
One might question the utility of such a construct because once we update another data member, the information possessed by the other one gets lost. Unions are useful in cases where one needs to conserve data. e.g. Suppose we want to implement a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data. If we declare this as:

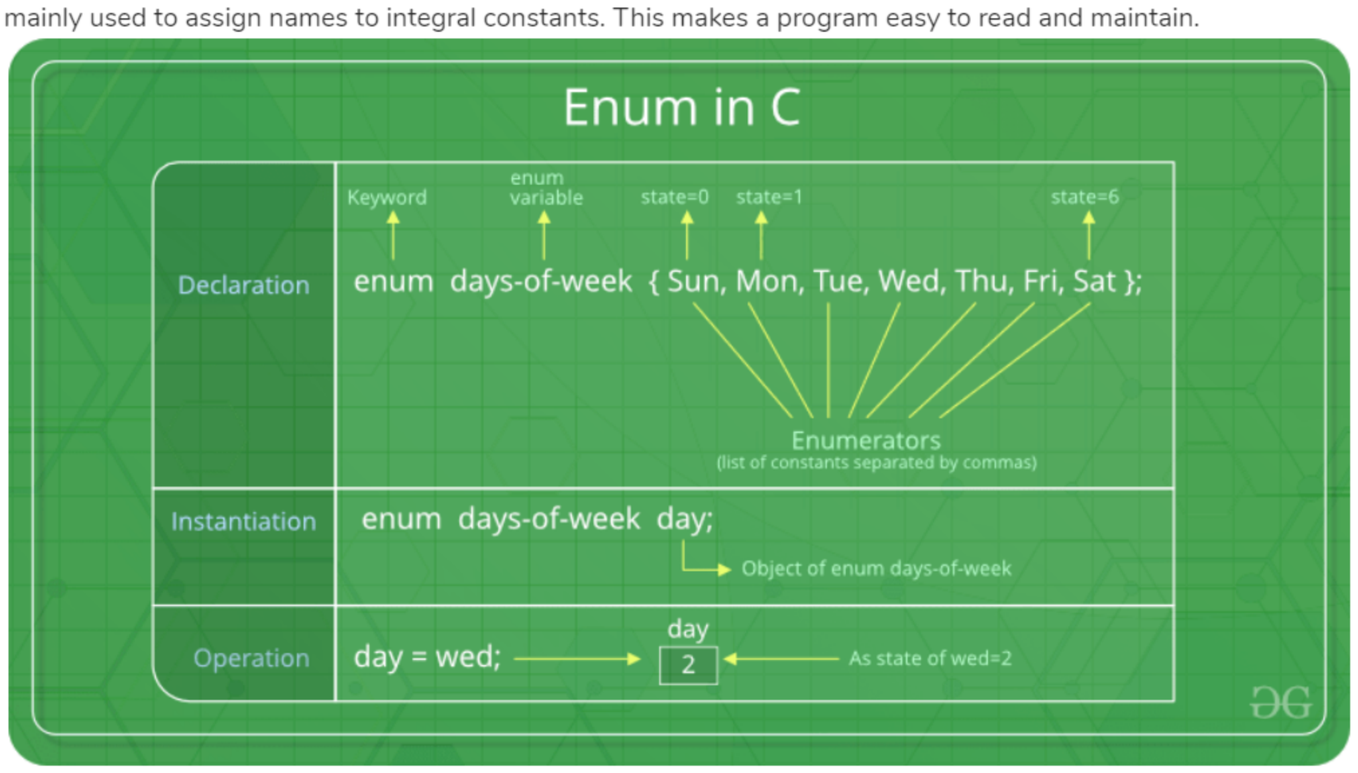
```

struct NODE {
    struct NODE* left;
    struct NODE* right;
    double data;
};

```

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as following, then we can save space.

```

struct NODE {
    bool is_leaf;
    union {
        struct
        {
            struct NODE* left;
```

```
        struct NODE* right;
    } internal;
    double data;
} info;
};

```

Enums Enums are mainly used to assign names to integral constants. This makes a program easy to read and maintain.



Enums are used in C++ as:

```
#include <bits/stdc++.h>
using namespace std;

enum week { Mon,
            Tue,
            Wed,
            Thur,
            Fri,
            Sat,
            Sun };

int main()
{
    week day;
    day = Wed;
    cout << day << endl;
    return 0;
}

```

Run

Output:

```
2
```

The arguments inside the enum are assigned integers starting from 0. However, the programmer has the flexibility to assign integral values of choice to the enum constants:

```

enum state { working = 1, idle = 0 };

// even duplicate values are allowed
enum duplicate { a = 1, b = 2, c = 2, d = 1 };

enum day { sunday = 1, monday, tuesday = 5, wednesday, thursday = 10, friday, saturday };

```

In the last case, we observe that all the constants inside the enum have not been assigned explicitly. In such a case, it gets assigned integral value + 1 than its previous one. In the case of the enum *day*:

```
sunday: 1,
monday: 2,
tuesday: 5,
wednesday: 6,
thursday: 10,
friday: 11,
saturday: 12
```

Namespaces in C++

Variable and Function names in C++ need to be unique in order to avoid ambiguity (Imagine having 2 variables with the same name, upon accessing compiler has no way of knowing which variable to look for). This raises compiler error. Imagine a large project being developed that has thousands of header files and modules. Name-collisions are bound to occur in such cases. As program-size increases, the number of identifiers increases linearly increasing the probability of name-collisions exponentially.
C++ introduces the concept of namespaces to deal with name collisions. A **namespace** is a block of code in which all identifiers are unique. By declaring unique namespaces, we can have the same name for identifiers and use them all together. The syntax for declaring a namespace looks as:

```
namespace < namespace-name > {
   //declare variables
   //declare classes, structures, enums, unions
   //declare functions
}
```

Example Suppose we have 2 header files A.h and B.h containing the same-named function *compute()*:

```

//A.h

int compute(int x, int y) { return x + y; }

```

```cpp

//B.h

int compute(int x, int y) { return x * y; }

```

We include both of them into our main.cpp file:

```cpp

#include <bits/stdc++.h>
#include "A.h"
#include "B.h"
using namespace std;

int main()
{
    cout << compute(5,6) << endl;

    return 0;
}

```

main.cpp upon compilation will throw error because of same-named compute() function declaration in different files.

To solve this using namespace, we do as:

```cpp

//A.h
namespace Mul {
    int compute(int x, int y) { return x * y; }
};

//B.h
namespace Add {
    int compute(int x, int y) { return x + y; }
};

//main.cpp
#include <bits/stdc++.h>
#include "A.h"
#include "B.h"
using namespace std;

int main()
{
    //To resolve scope we use :: (scope-resolution operator)
    cout << Mul::compute(5,6) << endl;
    cout << Add::compute(5,6) << endl;

    return 0;
}

```

```
Output:
30
11
```

As we can see, by using namespaces and referring to the appropriate function (using **:: Scope Resolution Operator**, we can avoid name-collisions.
Nested namespace declarations are also allowed:

```cpp

namespace Compute {
    namespace String {
        string add(string x, string y) { return x + y; }
    }

    namespace Integer {
        int add(int x, int y) { return x + y; }
    }
}

//Usage:
Compute::String::add("Hello ", "World");
Compute::Integer::add(5, 6);

```

Global Namespace Global variables and normal functions come under the global namespace. Thus, whenever we include two files with the same functions, we end up having same-named functions in the global namespace.

***std* Namespace & *using* directive** std namespace is a standard namespace built-in C++, which defines basic functions and template libraries (such as cin, cout, STL containers ~ vector, string, etc.). This was created to avoid name-collisions in case we want to declare identifiers having same-names. However, it might seem tedious to use repeatedly *std::* each time we use these built-in functions:

```cpp

#include <bits/stdc++.h>

int main()
{
    std::string str;
    std::cin >> str;
    std::cout << str << std::endl;

    return 0;
}

```

In the above code, if we don't use *std::*, the compiler will look for declarations of string, cin, cout & endl in the same file and report and error. Thus, we need to prefix *std::* each time we use these default keywords/functions. However, this is tedious. **using** directive eases out this task, as by ***using namespace std;***, we by-default ask our compiler to look for the keywords (or identifiers) in the std namespace. We thereafter don't need to prefix std:: each time we use the functions. The compiler first checks in the current file (for declarations which it doesn't find), then looks in the std namespace.

Sample Problems IV (User-defined Types and DMA)

The following are some basic implementation problems covering user-defined data-types and dynamic memory allocation.

- **Problem 1)** Write a program to store student records in a structure. It should contain the following information:
  - Student ID
  - Name
  - Year
  - CGPA

Solution - We shall create a student structure with the mentioned fields. Student ID, Name and Year are strings wherease CGPA will take floating-point values. To keep a set of such records, we create an array of structures as given below:

```cpp

#include <bits/stdc++.h>
using namespace std;

typedef struct student {
    string id, name;
    int year;
    double cgpa;
};

int main()
{

    student team[] = {
        { "GFG1", "Robert", 3, 8.67 },
        { "GFG2", "Chris", 2, 9.13 },
        { "GFG3", "Mark", 4, 8.43 },
    };

    for (int i = 0; i < 3; i++)
        cout << "ID: " << team[i].id << ", Name: " << team[i].name << ", Year: " << team[i].year << ", C

    return 0;
}
```

Run

Output:

```
ID: GFG1, Name: Robert, Year: 3, CGPA: 8.67
ID: GFG2, Name: Chris, Year: 2, CGPA: 9.13
ID: GFG3, Name: Mark, Year: 4, CGPA: 8.43
```

- **Problem 2)** Consider a library management system containing a large number of books, where one needs to store ISBN no, author, title and availability information of each book. Thereafter, implement functions supporting the following requirements:
    1. Show book information.
    2. Add new book.
    3. Display all books of a particular author.
    4. Display total number of books.
    5. Issue a book.

  Solution - We shall create an interactive program (like we did with the calculator) with options redirecting to a switch-case statement implementing each of the above functions. Cases 1, 2, 3..., 6 implement each of the requirements mentioned above. Similar to the previous example, we create a structure to store the book information, and for the library we create an array of such structures. We implement each of the functionalities as follows:

  1. *Book Information* - We iterate over the array and print details of each book in the library.
  2. *Add new Book* - We ask for user inputs for *title, author and ISBN*. Increment the *total* (for tracking the total number of books) and then insert the book into the array.
  3. *Display Books of Author* - Iterate over the library array and then display the books matching the appropriate author name.
  4. *Display Total no. of books* - Print the variable *total*.
  5. *Issue a book* - For issuing a book, ask for user input about the ISBN of the book to be issued and the member to whom it should be issued. Iterate over the array to locate the particular book. Thereafter, we need to check for the book's availability. If it is issued, we decline the request, otherwise we feed in the name of the member.

```cpp

#include <bits/stdc++.h>
using namespace std;
#define SIZE 1e5 + 6;

typedef struct book {
    string title, author, isbn;
    bool issued;
    string member_issued;
};

int total;
book library[SIZE];

int main()
{
    int option;
    while (1) {
        cin >> option;
        if (!option)
            break;

        switch (option) {
        case 1:
            for (int i = 0; i < total; i++)
                cout << "Name: " << library[i].title << ", Author: " << library[i].author << ", ISBN: "
            break;
        case 2:
            cout << "Enter Name, Author & ISBN\n";
            cin >> library[total].title >> library[total].author >> library[total].isbn;
```

In the above program, we define a structure to store information related to books. The structure contains the information regarding the book's title, author, ISBN as well as status information such as whether it is issued and if yes, then to whom. We create an array of books representing the library. We initialize the size of the array as a large value (to avoid overflow). We then prepare an interactive style program using a while loop and switch-cases to support each functionality of the library.

- **Problem 3)** Implement a list of numbers which doesn't require being provided any initial size, and can extend to as much as we want. There are 2 ways to implement this:
    - Make a dynamically-allocated array. When it gets full, extend it by using realloc.

```cpp

#include <bits/stdc++.h>
```

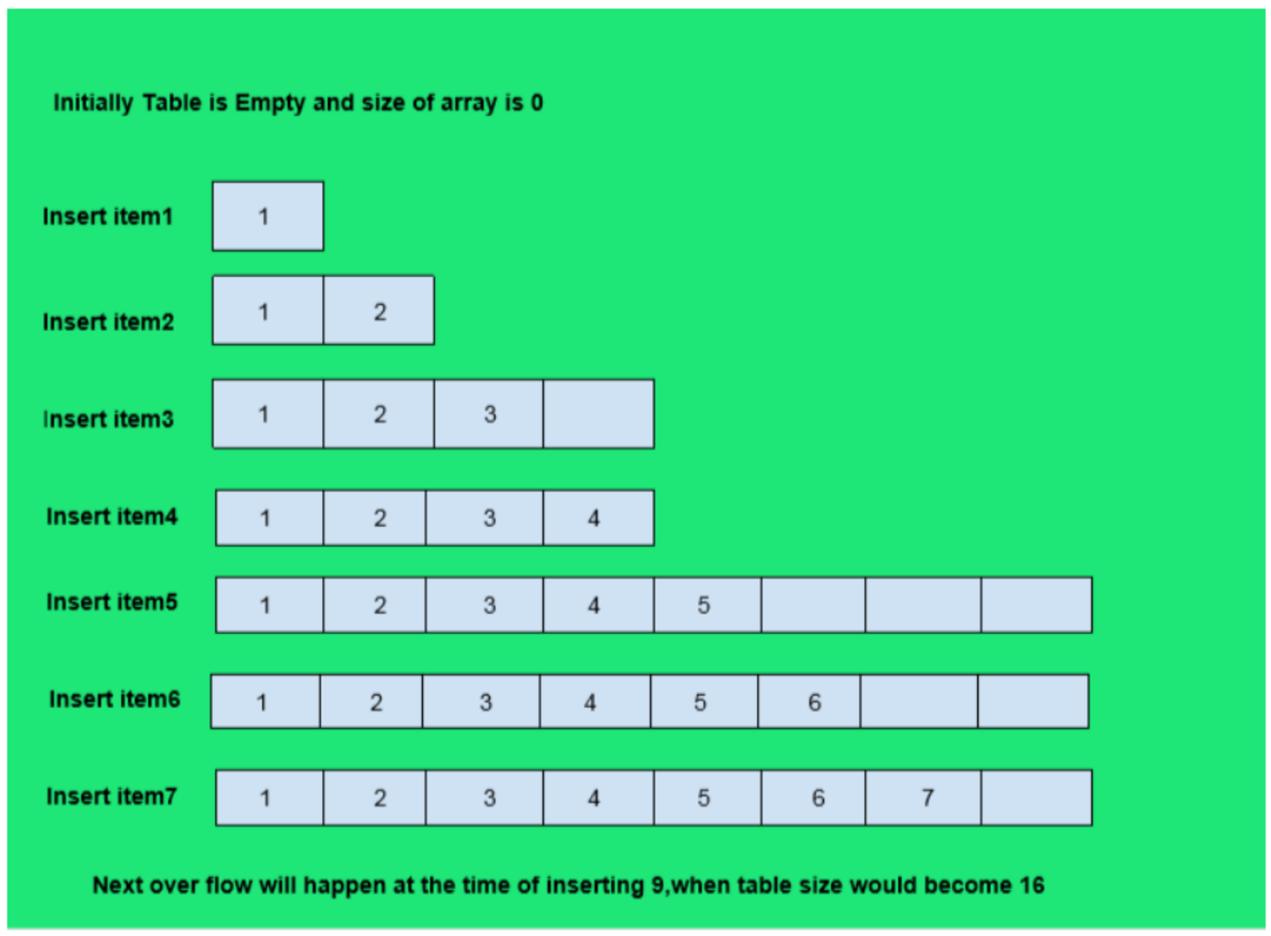
```cpp
using namespace std;

int *start, *last;
int size;

void insert(int x)
{
    if (!start) {
        // Initially no element present
        // hence malloc()
        start = (int*)malloc(sizeof(int));
        size = 1;
        last = start;
        *last = x;
        return;
    }
    else if (last - start + 1 == size) {
        // Once capacity is full,
        // we realloc to double the size
        size <<= 1;
        start = (int*)realloc(start, size * sizeof(int));
    }

    // Insert the element
    *(++last) = x;
}

int main()
```

Run

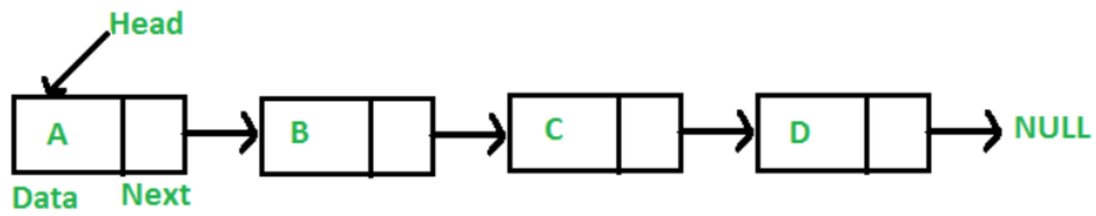
Output:

```
1 2 3 4 5 6
```

Above program implements a dynamically growing array depending upon how many numbers we keep inserting. We keep 3 pieces of information to facilitate the storage - *start*, *last* pointers and *size* variable. *size* indicates the total size of the array currently. Whenever we insert an element, we compare *(last - start + 1)* with *size*. This indicates whether the array is full/currently has space. If it is equal to the size, then we realloc and increase the size by twice of the earlier present size. Hence, size of the array grows in powers of 2: *1 -> 2 -> 4 -> 8* and so on. We then insert the element using **(++last) = x;*. This is the exactly how the vector template class under C++ STL is implemented under the hood.



- Make a Linked-List
  A linked-list is a dynamic data structure which can be used to accomodate indefinite amount of numbers. Each of the node in an LL stores the *key/data* and the *pointer to the next node*. We keep 2 pointers here ~ *start* and *last*. Whenever, we insert a new element, we append a new node to the end of the list and move the last pointer to that node. We then initialize the key with the value to be inserted. Below diagram shows figuratively the whole process.




```cpp

#include <bits/stdc++.h>
using namespace std;

typedef struct node {
    int key;
    struct node* next;
};

node *start, *last;

void insert(int x)
{
    if (!start) {
        start = (node*)malloc(sizeof(node));
        start->key = x;
        start->next = NULL;
        last = start;
    }
    else {
        last->next = (node*)malloc(sizeof(node));
        last = last->next;
```

```
        last->key = x;
        last->next = NULL;
    }
}

int main()
{
    insert(1);
```

Run

Output:

```
1 2 3 4 5 6
```

Type Casting in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as 'automatic type conversion'.
   - Done by the compiler on its own, without any external trigger from the user.
   - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
   - All the data types of the variables are upgraded to the data type of the variable with largest data type.

```
bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long -> float -> double -> long double
```

   - It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

```
// An example of implicit conversion

#include <iostream>
using namespace std;

int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    float z = x + 1.0;

    cout << "x = " << x << endl
         << "y = " << y << endl
         << "z = " << z << endl;

    return 0;
}
```

Run

Output:

```
x = 107
y = a
z = 108
```

2. **Explicit Type Conversion**: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

   In C++, it can be done by two ways:
   - **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

     Syntax:

     ```
     (type) expression
     ```

     where *type* indicates the data type to which the final result is converted.

     Example:

```
// C++ program to demonstrate
// explicit type casting

#include <iostream>
```

```cpp
using namespace std;

int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    cout << "Sum = " << sum;

    return 0;
}

```

Run

Output:

```
Sum = 2
```

- **Conversion using Cast operator:** A Cast operator is an **unary operator** which forces one data type to be converted into another data type. C++ supports four types of casting:

1. *Static Cast* - This is used for ordinary/normal type conversion. It does implicit type conversions (e.g. float to int, void* to int*).

   Example:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    float f = 3.5;

    //how we do in C
    int a = f;

    //how to do in C++
    int b = static_cast<int>(f);
    cout << b;
}

```

Run

Output:

```
3
```

2. *Dynamic Cast* - Used for handling polymorphism. One can assign derived class object to parent/base class reference using this casting. e.g. We have a pointer to Base Class, but we want to access members defined only in Derived Class.

```cpp
#include <bits/stdc++.h>
using namespace std;

class Base {
    public:
        virtual ~Base() {}
};

class Derived: public Base {
    public:
        void printChild() {
            cout << "I'm in Child\n";
        }
};

int main()
{
    Base *b = new Derived();

    Derived *d = dynamic_cast<Derived*>(b);
    d->printChild();

    return 0;
}

```

Run

Output:

```
I'm in Child
```

3. *Const Cast* - This is used to add/remove constness of a variable.

```cpp
#include <bits/stdc++.h>
using namespace std;

class student
{
private:
    int roll;
public:
    // constructor
    student(int r):roll(r) {}

    // A const function that changes roll
    // with the help of const_cast
    void fun() const
```

```
    {
        ( const_cast <student*> (this) )->roll = 5;
    }

    int getRoll()  { return roll; }
};

int main(void)
{
    student s(3);
    cout << "Old roll number: " << s.getRoll() << endl;

    s.fun();

    cout << "New roll number: " << s.getRoll() << endl;
```

Run

Output:

```
Old roll number: 3
New roll number: 5
```

4. *Reinterpret Cast* - It is the most dangerous cast, which should be used sparingly. It turns one type directly into another without a safety check. (e.g. one can store pointer value to an int).

```
#include <bits/stdc++.h>
using namespace std;

typedef struct Test
{
    int x, y;
} Test;

int main(void)
{
    Test *ptr_T = new Test();

    // gives compilation error
    // (Invalid casting from Test* to int*)
    // comment out this line and the code works
    int *ptr_i = static_cast<int*>(ptr_T);

    // Convert Test* to int* using reinterpret_cast
    // works like a charm!, but it is dangerous to use
    int *ptr_i = reinterpret_cast<int*>(ptr_T);

    return 0;
}
```

Run

Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.