

# OBJECT ORIENTED PROGRAMMING (PCC-CS503)

## Unit – 2

### Some difference between C and C++

#### Single line comments

Comments are portions of the code ignored by the compiler which allow the user to make simple notes in the relevant areas of the source code. Comments come either in block form or as single lines.

- **Single-line comments** (informally, *C++ style*), start with `//` and continue until the end of the line. If the last character in a comment line is a `\` the comment will continue in the next line.
- **Multi-line comments** (informally, *C style*), start with `/*` and end with `*/`.

**NOTE:** Since the 1999 revision, C also allows *C++ style* comments, so the informal names are largely of historical interest that serves to make a distinction of the two methods of commenting.

#### Local variable declaration within function scope

- In case of C, the **local variables must always be defined at the top of a block**.
- When a local variable is declared and not initialised, it will hold some garbage value (assigned by the system).
- A local variable is defined inside a **block** and is only visible from within the block.

```
int main()
{
    int i=4;                //local variable i
    int j=10;               //local variable j

    i++;

    if (j > 0)
    {
        printf("i is %d\n",i);    // i defined in 'main' can be seen
    }

    if (j > 0)
    {
        int i=100;               /* 'i' is defined and so local to
                                   * this block */
        printf("i is %d\n",i);
    }                            /* 'i' (value 100) dies here          */
    printf("i is %d\n",i);        /* 'i' (value 5) is now visible.      */
}
```

**However, if you initialize the local variables anywhere other than the top of the block, C compiler will give an error!**

```
int main()
{
    int i=4;                //local variable i
    i++;

    int j=10;               //local variable j can't be initialized here
    if (j > 0)
    {
        printf("i is %d\n",i);    // i defined in 'main' can be seen
    }

    if (j > 0)
    {
        int i=100;               /* 'i' is defined and so local to
                                   * this block */
        printf("i is %d\n",i);
    }
}
```

```

        }                                /* 'i' (value 100) dies here          */

        printf("i is %d\n",i);           /* 'i' (value 5) is now visible.      */
        return 0;

    }

```

**In case of C++, the local variables can be declared and defined anywhere in the code and not just at the start of the block.** The idea of this feature is to allow the programmer to put variable declarations near to the place you wish to use them.

```

int main()
{
    float pi = 3.142;    // Usual location for variable definitions

    cout << "PI is " << pi << endl;

    int Count = 1;       // C++ allows us to place a definition here but not C.

    while (Count < 10)
    {
        cout << Count << endl;
    }
    return 0;
}

```

## Function declaration

Why function declaration?

Declaring a value without defining it allows you to write code that the compiler can understand without having to put all of the details. This is particularly useful if you are working with multiple source files, and you need to use a function in multiple files. You don't want to put the body of the function in multiple files, but you do need to provide a declaration for it.

Example: `int add(int x, int y);`

This is a function declaration; it does not provide the body of the function, but it does tell the compiler that it can use this function and expect that it will be defined somewhere.

Why function definition?

Defining something means providing all of the necessary information to create that thing in its entirety. Defining a function means providing a function body.

Example:

```

int func();           // function declaration

int main()
{
    int x = func();    //calling function func()
}

int func()
{
    return 2;
}

```

Since the compiler knows the return value of func, and the number of arguments it takes, it can compile the call to func even though it doesn't yet have the definition. In fact, the definition of the method func could go into another file!

NOTE: If in a C program, a function is called before its declaration then the C compiler automatically assumes the declaration of that function in the following way:

```
int function_name();
```

Example:

```
//int func();

int main()
{
    int x = func();
}

void func()          // ERROR!- type mismatch!
{
    return 2;
}
```

And in that case if the return type of that function is different than “int”, compiler would show an error.

**In OOP, since there is bottom-up approach, so declaration is actually not a necessity.**

### Function overloading

Function overloading is a feature of a programming language that allows one to have many functions with same name but with different signatures.

// Different signatures of the same function- add()

```
Int add (int a, int b);
```

```
Void add (int a, int b);
```

```
Int add (int a, int b, int c);
```

```
Int add (int a, double b);
```

This technique is used to enhance the readability of the program because we don't need to use different names for the same action again and again.

There are two ways to overload a function, i.e. –

- Having different number of arguments
- Having different argument types

Function overloading is normally done when we have to perform one single operation with different number or types of arguments.

**C does not support function overloading but C++ does include this feature.**

Example:

```
#include <iostream.h>

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

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

```
double add(double x, double y) {
    return x + y;
}

int main() {
    int myNum1 = add(8,5,9);
    double myNum2 = add(4.3,6.26);
    cout << "Int: " << myNum1 << "\n";
    cout << "Double: " << myNum2;
    return 0;
}
```

**Prog: Overload the function “volume” for a cube, a cylinder and a cone.**

### Stronger type checking

- **Type checking** checks and enforces the rules of the type system to prevent type errors from happening.

```
int a = ghana; // compile time err!
```

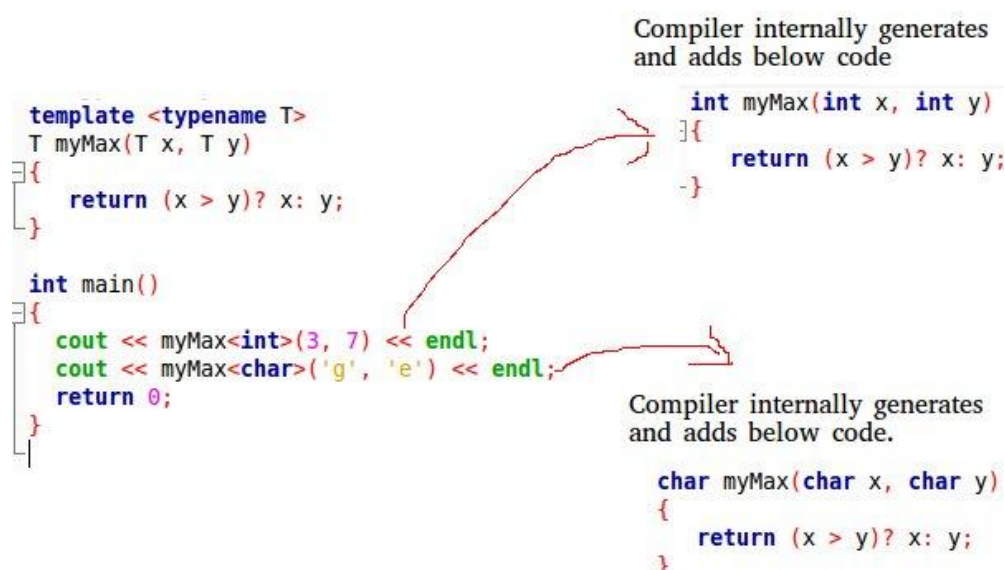
```
int a = “123”; // compile time err!
```

```
string x=123; // compile time err!
```

- A **type error** happens when an expression produces a value outside the set of values it is supposed to have.
- **Strong type checking** prevents all type errors from happening, either at compile time or at run time or partly at compile time and partly at run time.
- A **strongly-typed language** is one that uses strong type checking.
- **Weak type checking** does not prevent type errors from happening.
- A **weakly-typed language** is one that uses weak type checking.

**C++ which is otherwise a strongly typed language can allow flexibility in type-checking using the concept of Templates. C however has no concept of Templates.**

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.



## Reference variable

- C++ introduces a new kind of variable known as **Reference Variable**. It provides an alias (alternative name) for a previously defined variable.
- However, there is no concept of reference variables in C.
- A reference variable must be initialized at the time of declaration. This establishes the correspondences between the reference and the data object which it references to.
- A reference variable once initialized cannot be reinitialized.

Define:

```
[data_type] &[reference_variable]=[regular_variable];

#include <iostream.h>
#include <conio.h>
int main()
{
    clrscr();
    int a=10;    //existing variable a,c
    int c=20;
    int &b=a;    // reference variable b

    cout<< " value of a :"<< a << endl; //10
    cout<< " value of b :"<< b << endl; //10

    b=c;        //ERROR: We are just reassigning the value but not
                //rebinding the reference.

    cout<<"value of variables after reassignment:"<<endl;

    cout<< " value of a :"<< a << endl; //20
    cout<< " value of b :"<< b << endl; //20
    cout<< " value of c :"<< c << endl; //20

    cout<<" address of variables after reassignment:"<<endl;
    cout<< " address of a :"<< &a << endl;    //2004
    cout<< " address of b :"<< &b << endl;    //2004
    cout<< " address of c : "<<&c << endl;    //8002

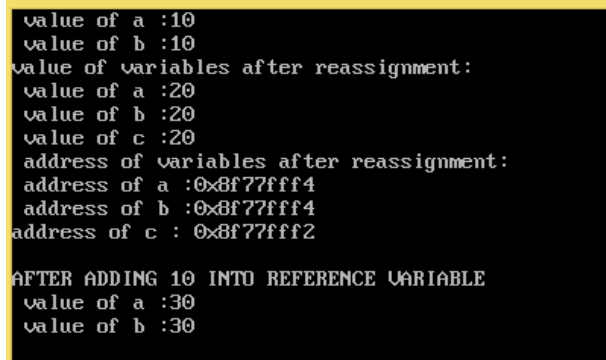
    //b=b+10;        //increment ref variable
    a=a+10;          //increment ref variable

    cout<<"\nAFTER ADDING 10 INTO REFERENCE VARIABLE \n";

    cout<< " value of a :"<< a << endl; //30
    cout<< " value of b :"<< b << endl; //30

    getch();
    return 0;
}
```

The output:



```
value of a :10
value of b :10
value of variables after reassignment:
value of a :20
value of b :20
value of c :20
address of variables after reassignment:
address of a :0x8f77fff4
address of b :0x8f77fff4
address of c : 0x8f77fff2

AFTER ADDING 10 INTO REFERENCE VARIABLE
value of a :30
value of b :30
```

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

```

int main()
{
    int a=10;
    int &b=a;        // reference variable
    int *p=&a;       // pointer variable

    cout<< " value of a :"<< a << endl;    //10
    cout<< " value of b :"<< b << endl;    //10
    cout<< " value of p :"<< p << endl;    //2004
    cout<< " value at *p :"<< *p << endl; //10

    b=b+10;

    cout<<"\nAFTER ADDING 10 INTO REFERENCE VARIABLE \n";

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

    cout<< " address of a :"<< &a << endl; //2004
    cout<< " address of b :"<< &b << endl; //2004
    cout<< " address of p :"<< &p << endl; //8006

    return 0;
}

```

### **Pointer vs Reference variable:**

1. A pointer can be re-assigned while reference cannot, and must be assigned at initialization only.
2. Pointer can be assigned NULL directly, whereas reference cannot.
3. Pointers can iterate over an array, we can use ++ to go to the next item that a pointer is pointing to.
4. A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.
5. A pointer to a class/struct uses '→' (arrow operator) to access it's members whereas a reference uses a '.' (dot operator)
6. A pointer needs to be de-referenced with \* to access the memory location it points to, whereas a reference can be used directly.

### **Parameter passing – value vs. reference**

Parameters are the data values that are passed from the calling function to a called function.

In C/C++, there are two types of parameters:

- **Actual Parameters:** parameters that are specified in calling function e.g. swap(10, 20); // here 10 & 20 are actual parameters
- **Formal Parameters:** parameters that are declared at called function e.g. swap(int x, int y) // x & y are formal arguments

When a function gets executed, the copies of actual parameter values are copied into formal parameters.

In C we can pass parameters in two different ways:

- Call by value, and
- Call by address

**Call by value:** In call by value parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. **The changes made on the formal parameters does not effect the values of actual parameters.** That means, after the execution control comes back to the calling function, the actual parameter values remains same. For example:

### **Swapping numbers using Function Call by Value**

```

#include<conio.h>
#include<iostream.h>

void swap(int x, int y) // called function, x & y are formal arguments
{
    int temp;
    temp = x;
    x = y;
    y = temp;
    //cout<<x<<y;    //40 10
}

void main() //calling function
{
    int a, b; // a & b are actual arguments
    clrscr();
    a = 10;
    b = 40;
    cout << "(a,b) = (" << a << ", " << b << ")\\n";
    swap(a, b);    //function call
    cout << "(a,b) = (" << a << ", " << b << ")\\n";
    getch();
}

```

O/P:

(a,b) = (10, 40)  
(a,b) = (10, 40)

**Call by Address:** In Call by Address parameter passing method, the memory location address of the actual parameters is copied to formal parameters. This address is used to access the memory locations of the actual parameters in called function. In this method of parameter passing, the formal parameters must be **pointer** variables.

That means in call by address parameter passing method, the address of the actual parameters is passed to the called function and is received by the formal parameters (pointers). Whenever we use these formal parameters in called function, they directly access the memory locations of actual parameters. So **the changes made on the formal parameters effects the values of actual parameters**. For example:

### Swapping numbers using Function Call by Address

```

#include<conio.h>
#include<iostream.h>

void swap(int *x, int *y) { //called func
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

void main() { // calling func
    int a, b;
    clrscr();
    a = 10;
    b = 40;
    cout << "(a,b) = (" << a << ", " << b << ")\\n";
    swap(&a, &b);
    cout << "(a,b) = (" << a << ", " << b << ")\\n";
    getch();
}

```

O/P:

(a,b) = (10, 40)  
(a,b) = (40, 10)

Point of difference: In C we can pass parameters in two different ways:

- Call by value, or
- call by address,

In C++, apart from the above two ways, we can get another technique, called “Call by reference”. Let us see the effect of these, and how they work.

**Call by Reference:** This technique only works in C++ and not in C. In this case, we have to put & before variable name at the function definition. The technique or working however remains same as that in “call by address” in case of C.

### Swapping numbers using Function Call by Reference

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

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

void main() {
    int a, b;
    clrscr();
    a = 10;
    b = 40;
    cout << "(a,b) = (" << a << ", " << b << ") \n";
    swap(a, b);
    cout << "(a,b) = (" << a << ", " << b << ") \n";
    getch();
}
```

O/P:

(a,b) = (10, 40)  
(a,b) = (40, 10)

WAP that accepts marks of ten students and calculates the sum and average of the class by passing these values to a function/s.

### **Passing pointer by value or reference**

- Both C and C++ allow pointers to be passed as arguments to a function.
- This is can be possibly done in two ways in case of C and C++:
  - Passing pointer by value
  - Passing pointer by reference: This can be again done in two ways:
    - (using the concept of “pointer to pointer”) – Allowed both in C/C++
    - (using the concept of “Reference to pointer”) – Allowed only in C++

### **Passing pointer by value:**

- A pointer can be passed as an argument to a function.
- Passing in a pointer by copy/value means that you “clone” the pointer of interest but you will not be able to change what the original pointer is pointing to (certainly you can change values of what it is pointing to, but not the pointer itself).
- However, if we try to be modify the value, then the changes made to the pointer does not reflects back outside that function.
- This is because only a copy of the pointer is passed to the function. It can be said that “pass by pointer” is passing a pointer by value.



- In most cases, this does not present a problem. But the problem comes when you modify the pointer inside the function.
- Instead of modifying the variable, you are only modifying a copy of the pointer and the original pointer remains unmodified.

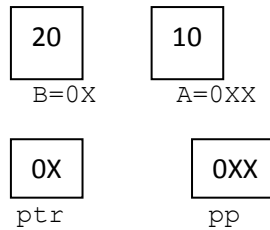
Passing pointer by value:

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

//int a = 10;

// function to change pointer value
void change(int *pp)
{
    cout << "Inside change ptr :" << *pp << endl; // 20
}

void main()
{
    int b = 20;
    clrscr();
    int *ptr = &b;
    cout << "Passing Pointer to function:" << endl;
    cout << "Before :" << *ptr << endl; // 20
    //cout << "only ptr :" << ptr << endl;
    change(ptr);
    cout << "After :" << *ptr << endl; // 20
    getch();
}
```



O/P:

The screenshot shows the output of the program running in DOSBox 0.74. The terminal window has a yellow title bar that says "DOSBox 0.74, Cpu speed: max 100%". The output text is as follows:

```
Passing Pointer to function:
Before :20
Inside change ptr :20
After :20
```

When we change the pointer value:

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

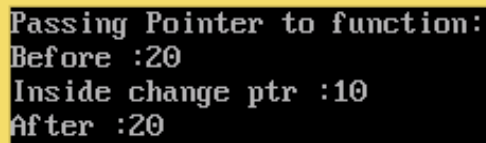
int a = 10;

// function to change pointer value
void change(int *pp)
{
    pp = &a;
    cout << "Inside change ptr :" << *pp << endl; // 10
}

void main()
{
    int b = 20;
    clrscr();
    int *ptr = &b;
    cout << "Passing Pointer to function:" << endl;
    cout << "Before :" << *ptr << endl; // 20
    //cout << "only ptr :" << ptr << endl;
    change(ptr);
    cout << "After :" << *ptr << endl; // 20
}
```

```
    getch();  
}
```

O/P:



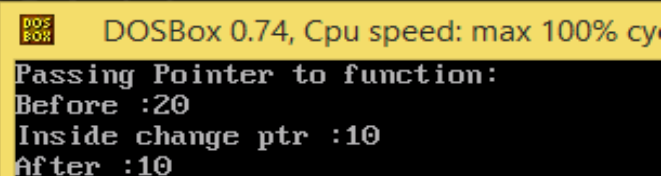
```
Passing Pointer to function:  
Before :20  
Inside change ptr :10  
After :20
```

### Passing “pointer to a pointer” as a parameter to function:

- The above problem can be resolved by passing the address of the pointer to the function instead of a copy of the actual function.
- If we pass by pointer reference (i.e. physically passing in the pointer itself), then you may change what the pointer is pointing to.
- For this, the function parameter should accept a “pointer to pointer” as shown in the below program:

```
#include <iostream.h>  
#include<conio.h>  
  
int a = 10;  
  
// function to change pointer value  
void change(int **pp)  
{  
    *pp = &a;  
    cout << "Inside change ptr :" << **pp << endl; // 10  
}  
  
void main()  
{  
    int b = 20;  
    clrscr();  
    int *ptr = &b;  
    cout << "Passing Pointer to function:" << endl;  
    cout << "Before :" << *ptr << endl; // 20  
    change(&ptr);  
    cout << "After :" << *ptr << endl; // 10  
    getch();  
}
```

O/P:



```
DOSBox 0.74, Cpu speed: max 100% cy  
Passing Pointer to function:  
Before :20  
Inside change ptr :10  
After :10
```

### How to call a function with “Reference to pointer” parameter?

There is NO POINTER TO REFERENCE!!! WHY???

```

#include <iostream.h>
#include<conio.h>

int a = 10;

// function to change pointer value
void change(int *&pp)
{
    pp = &a;
    cout << "Inside change ptr :" << *pp << endl; //10
}

void main()
{
    int b = 20;
    clrscr();
    int *ptr = &b;
    cout << "Passing Pointer to function:" << endl;
    cout << "Before :" << *ptr << endl; // 20
    change(ptr);
    cout << "After :" << *ptr << endl; // 10
    getch();
}

```

O/P:

```

DOS
BOX
DOSBox 0.74, Cpu speed: max 100% cy
Passing Pointer to function:
Before :20
Inside change ptr :10
After :10

```

## Operator new and delete

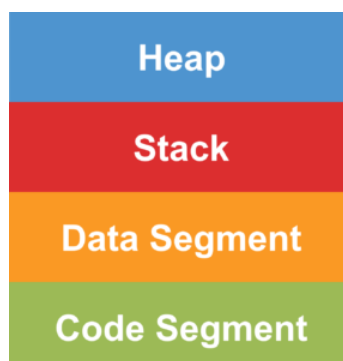
- Often we need to use Dynamic Memory Allocation in allocating the memory to the variables at the runtime.
- C language uses 'malloc', 'calloc' and 'realloc' functions to allocate memory dynamically. To de-allocate the memory allocated dynamically with these functions, it uses 'free' function call. C++ language also supports these functions from the C language to allocate/de-allocate memory.
- Apart from these functions, C++ introduces two new operators which are more efficient to manage the dynamic memory. These are 'new' operator for allocating memory and 'delete' operator for de-allocating memory.

```

int a=10; //static MA
int a[1000]; //static

```

NOTE: The stack is used for static memory allocation and Heap for dynamic memory allocation; both are stored in the computer's RAM.



## Allocation of Memory using *new* Keyword

- In C++ the *new* operator is used to allocate memory at runtime and the memory is allocated in bytes.
- The *new* operator denotes a request for dynamic memory allocation on the Heap.
- If sufficient memory is available then the *new* operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax:

```
datatype *pointer_name = new datatype;
```

```
//We can declare a variable while dynamical allocation in the following two ways.
```

Method 1:

```
int *ptr;  
ptr = new int;  
*ptr = 20;
```

Method 2:

```
int *ptr = new int (20);
```

```
int *ptr = new int {15};
```

```
// new operator is also used to allocate a block(an array) of memory of type data-type.
```

```
int *ptr = new int[20];
```

```
// The above statement dynamically allocates memory for 20 integers continuously of type int and returns a pointer to the first element of the sequence to 'ptr' pointer.
```

**Note:** If the heap does not have enough memory to allocate, the new request indicates failure by throwing an exception `std::bad_alloc`. Therefore, it is a good practice to check for the pointer variable produced by new before using it in the program.

## Deallocation of memory using *delete* Keyword:

Once heap memory is allocated to a variable or class object using the *new* keyword, we can deallocate that memory space using the *delete* keyword.

Syntax:

```
delete pointer_variable;
```

```
// Here, pointer_variable is the pointer that points to the data object created by new.
```

```
delete[] pointer_variable;
```

```
//To free the dynamically allocated array memory pointed by pointer-variable we use the following form of delete:
```

**Example:**

```
delete ptr;  
delete[] ptr;
```

**Note:** The object's extent or the object's lifetime is the time for which the object remains in the memory during the program execution. Heap Memory allocation is slower than a stack, since, in heap, there is no particular order in which you can allocate memory whereas in the stack it follows LIFO.

Example:

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

```

void main()
{
    // declare an int pointer
    int* pointInt;

    // declare a float pointer
    float* pointFloat;
    clrscr();

    // dynamically allocate memory
    pointInt = new int;
    pointFloat = new float;

    // assigning value to the memory
    *pointInt = 45;
    *pointFloat = 45.45f;

    // int *ptr = new int (10); //

    cout << *pointInt << endl;
    cout << *pointFloat << endl;

    // deallocate the memory
    delete pointInt, pointFloat;

    getch();
}

```

## Dynamically Allocating Arrays

The major use of the concept of dynamic memory allocation is for allocating memory to an array when we have to declare it by specifying its size but are not sure about it.

Let's see, an example to understand its usage.

```

#include <iostream.h>
#include<conio.h>

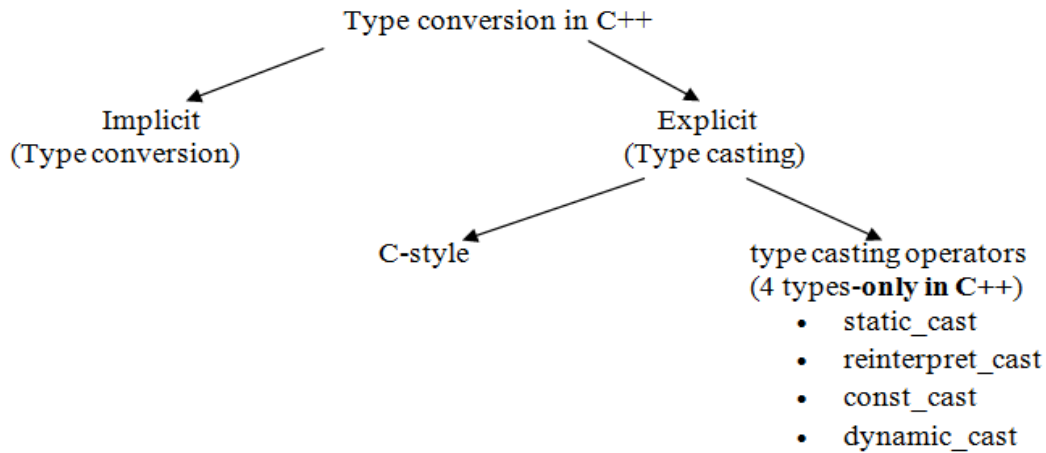
void main()
{
    int len, sum = 0;
    clrscr();
    cout << "Enter the no. of students in the class" << endl;
    cin >> len;

    int *marks = new int[len]; //Dynamic memory allocation
    cout << "Enter the marks of each student" << endl;
    for( int i = 0; i < len; i++ )
    {
        cin >> *(marks+i); //
    }
    for( int j = 0; j < len; j++ )
    {
        sum += *(marks+j);
    }
    cout << "sum is " << sum << endl;
    getch();
}

```

In this example first we ask the user for the number of students in a class and we store its value in the len variable. Then we declare an array of integer and allocate it space in memory dynamically equal to the value stored in the len variable using this statement `int *marks = new int[length]`; thus it is allocated a space equal to 'length \* (size of 1 integer)'.

## Type conversion and Type casting in C++



**Type conversion (implicit or automatic)** is a mechanism in which a data type is automatically (there is no need for a casting operator) converted into another data type by a compiler at the compiler time. One more important thing is that it can only be applied to compatible data types.

The type compatibility is being able to use two types together without modification and being able to substitute one for the other without modification.

Type promotion hierarchy: bool -> char -> short int -> int -> Uint -> long -> UL -> long long -> float -> double -> long double

Ex.

```
int a=10;
char b='A';
float f=6.5;
int c= a+b;           //c=int+char= 10 + 65 = 75
//char c=a+b;        //c=int+char= 10+65 = 75 //k
//float k=a+f;        //k=int+float=10.0+6.5=16.5
//return (a+b+f);     //int+char+float= 10.0+65.0+6.5
```

Ex.:

```
int a=10;
float f=5.67
double b=5.73, z= 19.8;
z=z/a + b*60+a;
cout<<z;    //??
```

### Problem with implicit type conversion?

It can result in data loss. Ex.:

```
int a=3;
int b=2;
float k=a/b;    // k=1; DATA-LOSS: k should actually be 1.5
```

So, we need a type-casting that will enable the programmer to convert data-types before assignment.

**Type casting (explicit or by the programmer)** is a mechanism in which a data type is converted into another data type by the programmer using the casting operator during the program design.

C-style type-casting: The **type-cast operator** is used to explicitly cast the value of an *expression* from one *data-type* to another *data-type*. In order to use the *type-cast* operator, let us see its syntax –

(data-type-name) expression;

Example:

```
int a=3;
int b=2;
float k=(float)a/b;    // k=1.5; NO DATA-LOSS
```

There are other casting operators supported by C++, they are listed below and these casting operators will be used while working with classes and objects.

**static\_cast:** It's the most simplest conversion.

**syntax:**            static\_cast<type> (expr)

```
void main()
{
int i;
float f=3.142;
i=f;           //warning: data loss may occur as float is converted to int
              //insted we can use static cast to make code more readable
//i=static_cast<int>(f);
}
```

**reinterpret\_cast:** It's the strongest conversion. Can even convert pointer to an ordinary variable and vice-versa.

**syntax:**            reinterpret\_cast<type> (expr)

```
void main()
{
int i=2000;
int *p;
p=i;           //error: can't be done directly
              //insted we can use reinterpret cast to do this hard conversion
//p=reinterpret_cast<int*>(i);
//cout<<p<<endl;
//will print the hexa value of 2000 i.e. value assigned to i int
}
```

**const\_cast:** The const\_cast operator is used to explicitly override const in a cast.

**Syntax:**            const\_cast<type> (expr)

```
void main()
{
const int i=20;
int *p;
p=&i;          //error: invalid conversion from 'const int*' to 'int*'
              //insted we can use const cast to make code more readable
//p=const_cast<int*>(&i);
//cout<<p<<endl;
//will print the hexa value of 20 i.e. value assigned to i int
}
```

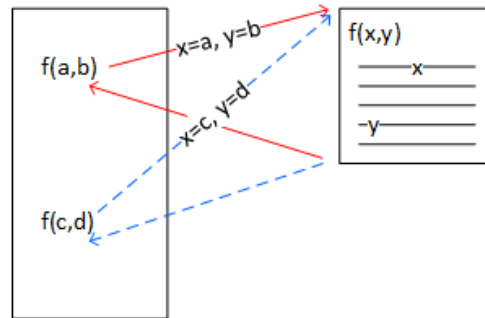
**dynamic\_cast:** The dynamic\_cast can perform down-casting. In c++ by default up-casting is supported i.e. a if a base class pointer holds the address of a derived class object, but if a derived class pointer holds the address of a base class object it is down-casting. But for applying down-casting in C++ using dynamic\_cast, there is a pre-requisite, we need a polymorphic class i.e. the class must contain at least one virtual function.

**Syntax:**            dynamic\_cast<type> (expr)

## Inline Functions in contrast to macro

Before understanding Inline or macro functions, let's check out working of any normal function.

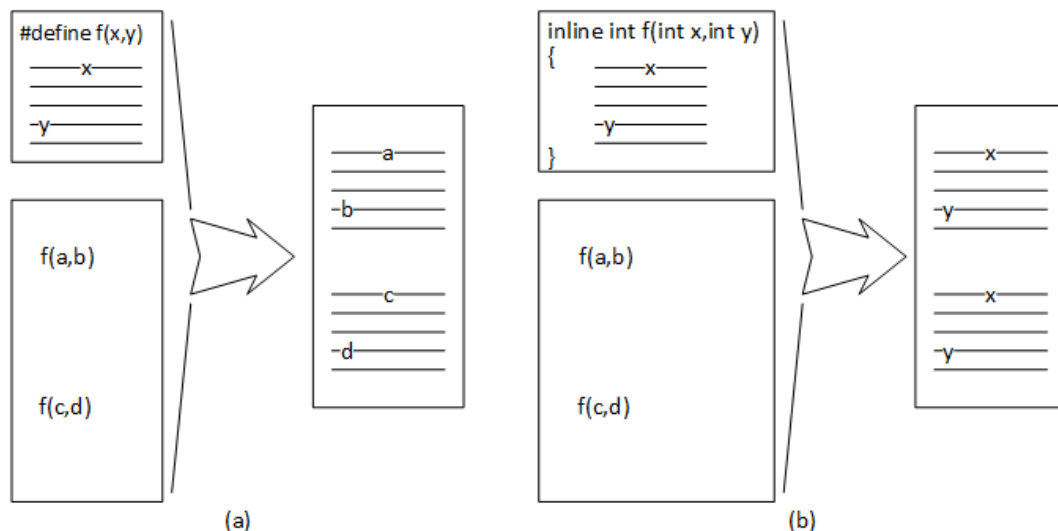
**Normal/regular function:** In case of a normal/regular function, only one copy of the function's instructions exists in the program and those instructions are executed each time the function is called.



Normal function's working

**Macro vs. inline function:** The *behaviour* of a parameterized macro (a) and of an inline function (b) is very similar but there are some important differences.

- The **pre-processor** *expands* macros by replacing the macro (e.g., "f(x, y)") with the statements in the macro body. The character or characters represented by a and b (including operators) replace x and y in the expansion. There is a copy of the macro statements for every occurrence of the macro in the source code.
- Inline functions are also expanded and replace the function call, but it is the **compiler** and not the pre-processor that performs the expansion. Like a macro, there is one copy of the function body for every function call in the source code. However, like a "normal" function and unlike a macro, the arguments are evaluated and the resulting *value* is passed to the function parameters before the expansion takes place.



(a) a parameterized macro and (b) an inline function

### Example of an Inline function:

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

// Inline function
inline int Maximum(int a, int b)
{
    return (a > b) ? a : b;
}

// Main function for the program
int main()
{
    cout << "Max (100, 1000):" << Maximum(100, 1000) << endl;
```



```

        cout << "Max (20, 0): " << Maximum(20, 0) << endl;

        return 0;
}

```

O/P:

```

Max (100, 1000): 1000
Max (20, 0): 20

```

### Example of Macro:

```

#include <iostream>
using namespace std;

// macro with parameter
#define MAXIMUM(a, b) (a > b) ? a : b

// Main function for the program
int main()
{
    cout << "Max (100, 1000):";
    int k = MAXIMUM(100, 1000);
    cout << k << endl;

    cout << "Max (20, 0):";
    int k1 = MAXIMUM(20, 0);
    cout << k1;

    return 0;
}

```

O/P:

```

Max (100, 1000):1000
Max (20, 0):20

```

1. Diff b/w macros and Inline-Functions?
2. Whether In-line is a cmd or a request to a compiler?

### Default arguments

- A **default argument** is a default value provided for a function parameter.
- If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used.
- If the user does supply an argument for the parameter, the user-supplied argument is used.
- **Default arguments is only used in C++ and not in C.** (In C, it is compulsory to pass all arguments during function call)

Example:

```

#include <iostream>
using namespace std;
int sum(int a, int b=10, int c=20);

int main()
{
    // In this case a=1 and b and c = default arguments
    cout<<sum(1)<<endl;

    // In this case a=1 and b=2 and c = default argument
    cout<<sum(1, 2)<<endl;

    // In this case no default arguments
    cout<<sum(1, 2, 3)<<endl;
    return 0;
}

```

```

}

int sum(int a, int b, int c)
{
    int z;
    z = a+b+c;
    return z;
}

```

O/P:

31  
23  
6

**Default value assignment rule:** If a default value is assigned to an argument, the subsequent arguments must have default values assigned to them; else a compilation error occurs. Following the invalid declarations:

```

int sum(int a=10, int b, int c=30);
int sum(int a, int b=20, int c);
int sum(int a=10, int b=20, int c);

```

1. Write a macro to calculate simple interest from principal, rate of interest and time.  
Simple interest = (principal\*rate of interest\*time)/100.
2. WAP that can demonstrate the functionality of a calculator and it must include the following operations:  
addition, subtraction, division, multiplication and getting two input values from the user.

You have to perform implementation of all these operations using three cases:

Case 1: using normal functions

Case 2: using Inline functions

Case 3: using macros

Comment which will be the most suitable way of implementation in this particular problem and why?