Scope resolution: https://www.geeksforgeeks.org/scope-resolution-operator-in-c/

Namespace

When a big program split across files, it is possible that the variables introduced in the global scope may clash resulting in linker time errors.

```
using namespace std;
```

FIle only use fn from std libary

To avoid the clashes of the names in the global space, names are introduced in one or more named space by the programmer.

There is a namespace called **std** to which the names of all standard library functions and other declarations are introduced

For your further study

Namespace using Declarations

A using declaration lets us use a name from a namespace without qualifying the name with a namespace_name::prefix.

Syntax: `using namespace::name;`

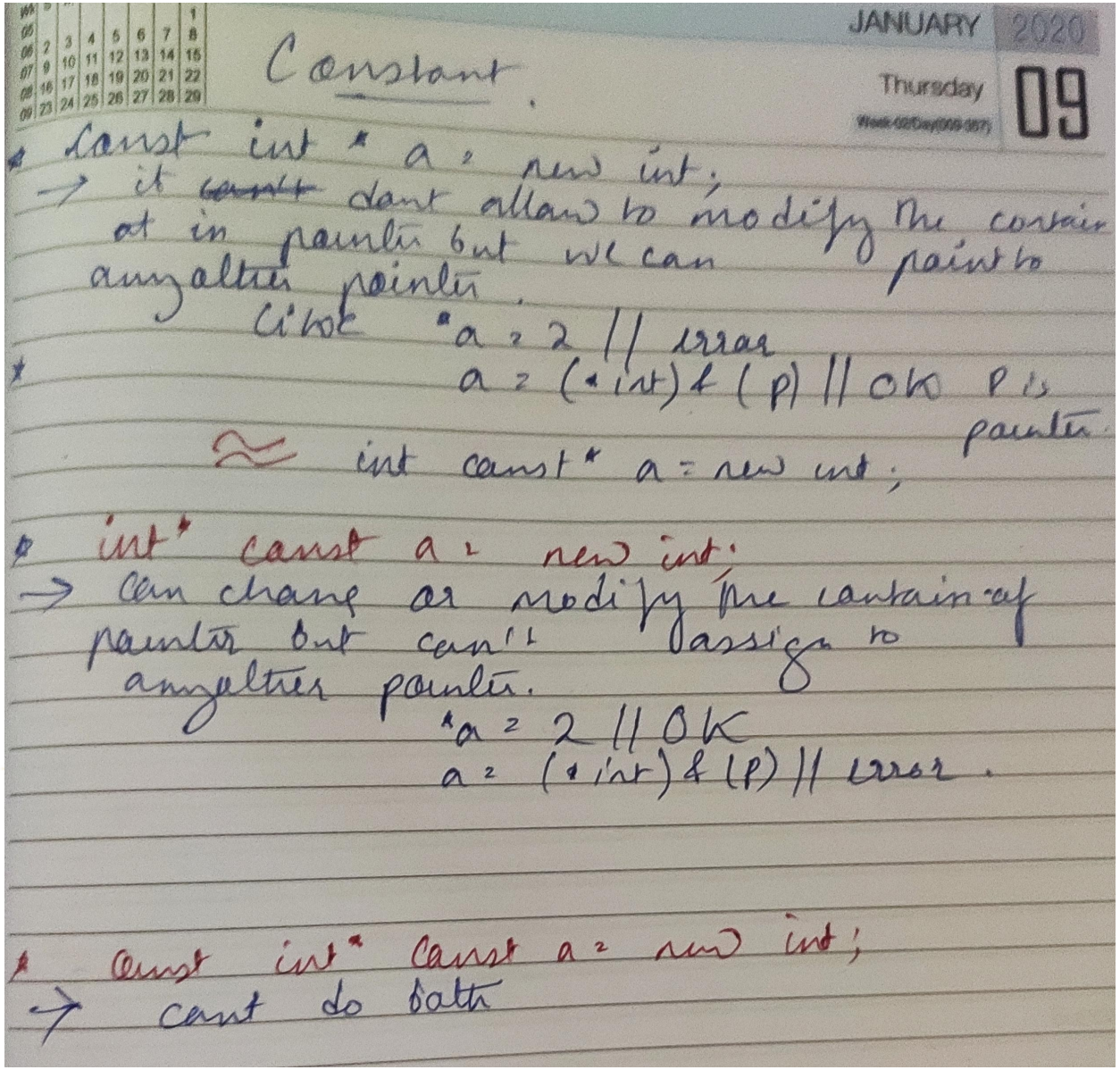
Example

```
#include <iostream>
using std::cout;
int main()
{
        cout << "Enter any number:";
        int n1 = 0;
        std::cin >> n1;
        cout << "The entered number is " << n1;
        return 0;
}
```

Constant

- Constant in C++ have a name, type as well as a value.
- Definition of constant can be placed in .h files.
- Named constants are used in the program for the following reasons
  - Improves readability and therefore decreases the effort for maintenance
  - [redacted]

- Constants could be used -
  - Constants in program body
  - Constant parameter in function
  - Constant members in object / structure
  - Constant object through which a method is invoked


JANUARY 2020
Thursday
09
Constant.
* const int * a = new int;
→ it can't dont allow to modify the contain at in pointer but we can point to anyother pointer.
*a = 2 // error
a = (*int) & (p) // ok p is pointer.
≈ int const * a = new int;
* int * const a = new int;
→ can change or modify the contain of pointer but can't assign to anyother pointer.
*a = 2 // OK
a = (*int) & (p) // error.
* const int * const a = new int;
→ cant do both

Constant

Example

```
1.    int x; int y; int z;
2.    const int a = 10;    // a is a constant integer
3.    // const int b;      // syntax error - no value provided
```

```
1.  int * const p = &x; // p is a const pointer to integer
2.                      // should be initialized with
definition
3.    // p = &y;          // value of p can NOT be changed
4.    *p = z;             // value of *p can be changed
```

```
1.   const int * q = &x; // q is a pointer to an int which is
a const
2.   q = &y;              // value of q can be changed
3.   // *q = z;          // value of *q can NOT be changed
```

when we declare pointer without initialization then it called as Wild Pointer

Pointers and dynamic allocation

- C++ supports dynamic allocation (and deallocation) through two operators : new and delete
- The operator new requires a type as the operand. It allocates as much memory as required for the particular type and returns a pointer to the type specified
- The operator new also works on user defined types
- The operator new and delete can be overloaded (will be discussed later)

Example code :

```
int *p = NULL;
p = new int;
*p = 20;
delete p;
int *q = new int[5];
for(int i = 0; i < 5; i++){
        q[i] = i;
}
delete []q;
```

Remembering Alias

Alias example code

```
int *p, *q;
p = new int;
*p = 20;
q = p;   // q and p refer to the same location,
         // they are now aliases
*q = 30; // even *p would have changed
```

Remembering Garbage

Garbage example code

```
int *p = Null;
p = new int;    // Space for an int is
                // create/reserved
*p = 20;
p = new int;    // AHA ! Space for one more int is
                // created, earlier value and
                // location lost - it's a memory
                // leak and also called garbage
                // location without accesses
```

Remembering Dangling reference

Dangling reference example code

```
int *p, *q;
p = new int;
*p = 22;
q = p;   // q and p refer to the same location,
         // they are now aliases
delete p; // space for int is released
*q = 333; // AHA ! The pointer q points to a non-
          // existing location. This is called
          // dangling reference. This can causes
          // memory corruption
```

This typical issue happens in case of shallow copy

Shallow Copy stores the references of objects to the original memory address.
Deep copy stores copies of the object's value in new address

Reference Variable

Example code :

```
int a = 10; int x = 30;
int &b = a; // b is a reference to a; b is same as a.
            // no new integer location is allocated to b
            // whatever b is used, it automatically refers to a

cout << &a << " " << &b << endl;  // outputs same address
cout << b << endl; // outputs 10

int c = b; // c becomes 10
cout << c << endl; // outputs 10

c = 20;  // b remains unchanged
cout << b << endl; // still outputs 10
```

Reference Variable

Contd...

Example code (continued from previous slide):

```
b = c; // same as a = c
cout << a << endl; // outputs 20
// int &d; // Syntax error
// &b = x; // Syntax error

// int & arr[] = {a, x}; // Array of reference are not
    allowed
```

declared as reference but not initialized

&d = x;

Few points on Pointers & Reference

- Pointers may be undefined or NULL; reference should always be associated with a variable
- Pointers may be made to point to different variable at different time; reference is always associated with the same variable throughout its life
- Pointers should be explicitly dereferenced; References are automatically dereferenced

if it goes out of scope

Call by Reference or Reference Parameters

- 'C' provide only one mechanism of parameter passing – by Value.
  - Arguments are copied to the parameters and changes to the arguments are not reflected in the calling functions
  - If arguments should be changed by the called function, then the pointer (that is also value) to the arguments is passed. Thus simulating parameter passing by reference in C (but its is just a simulation).
  - In the called function, pointer has to be dereferenced to access the arguments. This is costly operation.

- 'C++' supports parameter passing by Value as well by Reference.
  - As reference parameters are just a copy of the actual parameter, there is no need of dereferencing, Reference parameters are automatically dereferenced.

Call by Reference

Example

```
void swap1(int x, int y) {
    int temp; temp = x; x = y; y = temp; }
void swap2(int * x, int * y) {
    int temp; temp = *x; *x = *y; *y = temp; }
void swap3(int & x, int & y) {
    int temp; temp = x; x = y; y = temp; }


int main() {
    int a = 10, b = 20;
    swap1(a, b);     // call by value - changes not reflect
    swap2(&a, &b);   // call by value with pointer -
                     // changes reflects
    swap3(a, b);     // call by reference - changes reflects
    return 0; }
```

Call by Reference

Few points

- Less overhead in terms of time & space at runtime compared to parameter passing by value
- Can avoid problems resulting in shallow copy of parameter passing by value – this happens when a structure or an object has a pointer as a member
- Can be passed const reference to avoid changing the argument in the function – this is in case arguments are not required to change in the called function
- It is also possible to return by reference when an lvalue is required in the calling function
- It is efficient w.r.t. passing by pointer for a complex structure as dereferencing is not required

Conclusion : Always preferable to pass by reference

Return by Reference

If you return a reference from a function, you must take the same case as if you are return a pointer from a function

- Whatever the reference is connected to shouldn't go away when the function returns
- Otherwise you will be referring to unknown memory location

```
int * f (int *x) {
    (*x)++;
    return x; // Safe
}
```

```
int & g (int & x) {
    x++;
    return x; // Safe
}
```

```
int main() {
    int a = 20;
    f(&a); // Ugly (but explicit)
    g(a); // Clean (but hidden)
}
```

Function Overloading - Static Polymorphism

It is possible to have more than one function with the same name where the calls are resolved at compile time based on matching the arguments to the parameters

Points on function overloading :

- More than one function with the same name
- Function call resolution is a compile time phenomenon and there is no extra overhead at runtime
- Functions should be declared in the same scope
- Resolution based on the number, type and order of arguments in the function
- Resolution does not depends on the result type

Function Overloading

An example

```
void display(char * name) {
    cout << "Name " << name << endl;
}
void display(char * name, int age){
    cout << "Name " << name << " Age " << age << endl;
}
int main() {
    char name[10] = "Aman";
    int age = 10;
    display(name, age); // Outputs : Name Aman Age 10
    display(name); // Name Aman
    return 0;
}
```

Default Parameter

There are many functions which have a large number of parameters. The user may not be interested in providing all these arguments in the call and he might be happy if the system can choose some default appropriate values for some of these parameters.

Example code

```
void display(const char * = "India", int = 72);
int main() {
    display("Bangladesh", 48); // Calling with two arguments
    display("India"); // Calling with one argument
    display(); // Calling with no argument
    return 0;
}
void display(const char * name, int age) {
    cout << "Country " << name << " Age " << age << endl;
}
```

Default Parameter

Few points

- The parameters should have a reasonable default
- It is a way of expanding the function
- Default value(s) are specified as part of the declaration
- Can be specified for functions whose source codes are not available – function writer may not know anything about these defaults
- Only rightmost parameters can be default – all parameters can also be default
- Leftmost arguments should be specified

Inline function

- Inline functions are used to reduce the overhead of normal function call
- It's a request to the C++ compiler that inline substitution of the function body is to be preferred to the usual function call implementation
- Suggestion could be ignored based on the compilers discretion
- Unlike Macro, it does not leave any side effect due to expansion of code
- The compiler checks for the proper use of the function argument list and return value – macro is incapable of.

Example Code :

```
#define MC_SQUARE(X) X*X
inline int fn_square(int x) { return x*x;  }
int a = b = 3;
int i = MC_SQUARE(++a);     // results in 20
int j = fn_square(++b);     // results in 16
```

Inline function Cont…

- Any function defined within a class body is automatically inline
- For a non-class inline function, the function name is preceded with inline keyword
  - It must include function body with the declaration, otherwise compiler will treat it as a ordinary function.
  - Put inline definition in a header file
- Compiler keeps the inline function type and body in its symbol table. When inline function called correctly, compiler substitute the function body from the function call – thus eliminating the overhead
- The inline function does occupy space, but if the function is small, this can actually take less space than the code generated from a ordinary function call (pushing arguments on the stack and doing the function call)

Template function

There are many instances where the structure of the code (logic) does not change with the input type. In such case it is possible to write generic routines where the type information can be passed as parameter in the calling code.

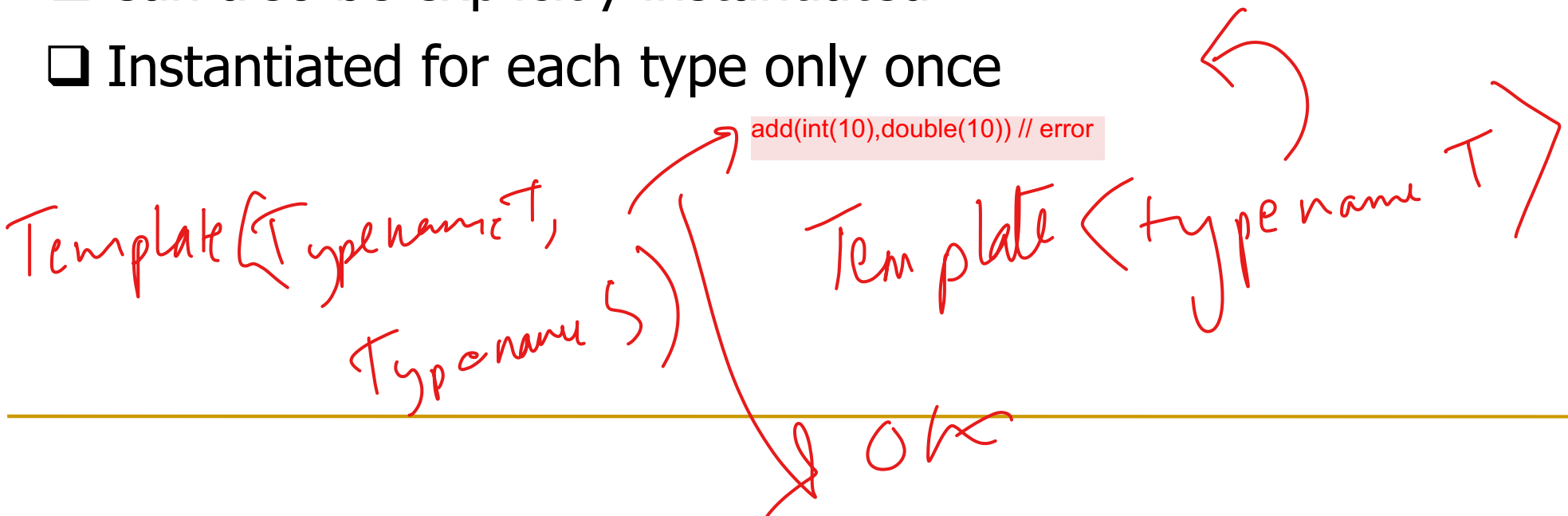
Example code :

```
template <typename T>  // T is a type parameter
void add(T &x, T &y) {
    T temp = x + y;
    cout << temp << endl;
}
int main() {
    int a = 10, b = 20;
    double p = 2.4, q = 4.5;
    add(a, b); // Causes compilation type instantiation of the
               // template function with type T = int.
    add<double>(p, q); // Another instance of the template
                       // function with type T = double.
}
```

Template function

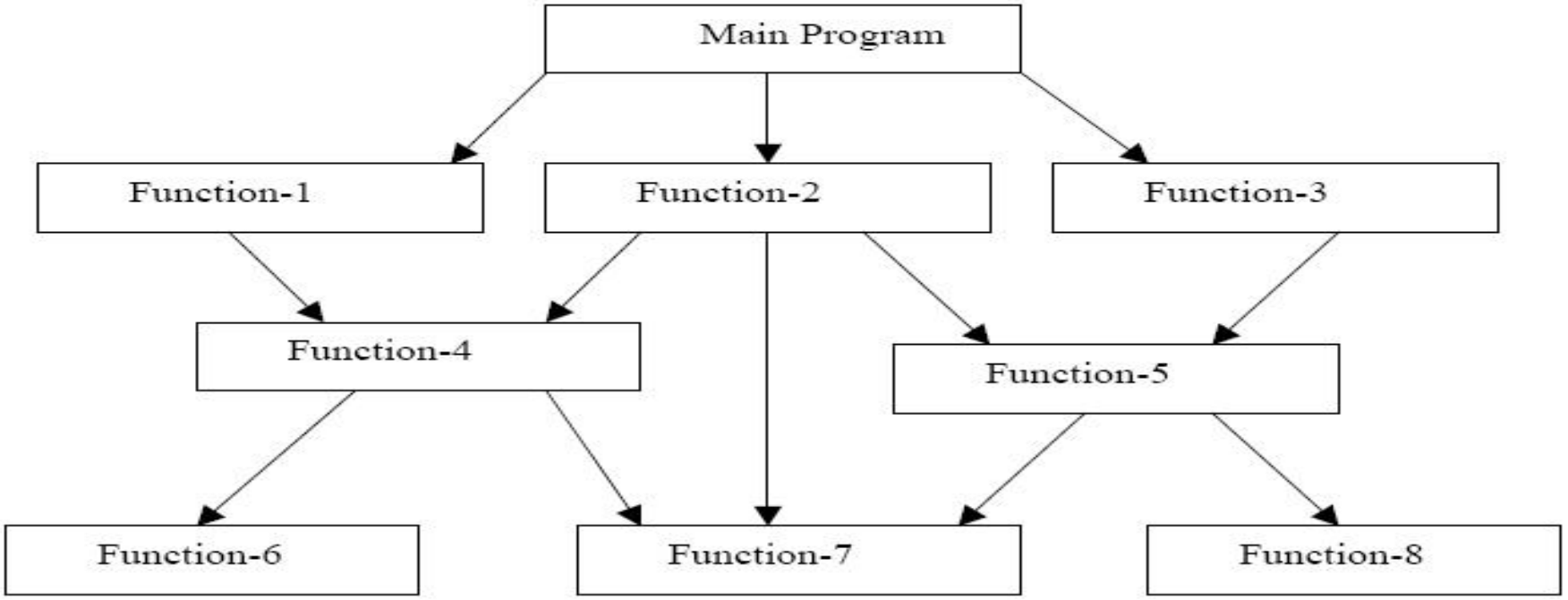
Few points

- Mechanism to generate functions at compile type
- No extra overhead at run time
- Used to making functions generic
- Used when behavior/algorithms of the functions are same
- Gets implicitly instantiated by the call
- Can also be explicitly instantiated
- Instantiated for each type only once



Procedural Programming

- The problem is viewed as the sequence of things to be done such as reading, calculating and printing
- The primary focus is on functions - the technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem



Procedural Programming Contd...

- In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions.
- Global data are more vulnerable to an unintentional change by a function.
- In a large program it is very difficult to identify what data is used by which function.
- In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.
- Can not model real world problems very well - functions are action-oriented and do not really corresponding to the element of the problem

Disadvantage of PPP

Procedural Programming Characteristics

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
  - Data doesn't have a owner – many function can modify data
- Functions transform data from one form to another.
  - Difficulty in maintaining data integrity
  - Difficult to pinpoint bug sources when data is corrupted
- Employs top-down approach in program design.

Object : A more formal definition

An object is an entity that has a well-defined boundary. That is, the purpose of the object should be clear.

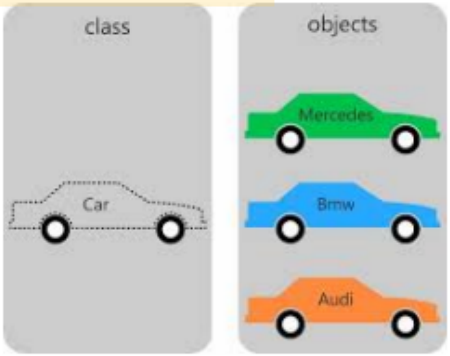
An object has three key components :

- Identity – Distinguishes one object than other
- Attributes – Information describes their current state
- Behaviour – Specific to an object

Object in a computer program are self contained. They are independent to each other.

3) What is a class?

A class is simply a representation of a type of object. It is the blueprint/plan/template that describes the details of an object.



4) What is an Object?

An object is an instance of a class. It has its own state, behavior, and identity.

Abstraction

Abstraction means focusing the essential quality of something, rather than one specific example.

It means automatically discard the unimportant and irrelevant.

Abstraction can be defined as :

"Any model that includes the most important, essential or distinguishing aspects of something while suppressing or ignoring less important, immaterial or diversionary details."

Abstraction

Points to remember

- Abstraction is as an external view of the object.
- Abstraction allows us to manage complexity by concentrating on the essential characteristics of an entity that distinguishes it from all other kind of entities.
- An abstraction is domain and perspective dependent. That is, what is important in one context, may not be in another.
- In the context of OOP, abstraction of an object means its outside view provided by the interface.
  - Note that the interface only tells WHAT the object can do, it does not says HOW it performs its work

how part done in encapsulation

Encapsulation

Encapsulation means hiding implementation detail from external world.

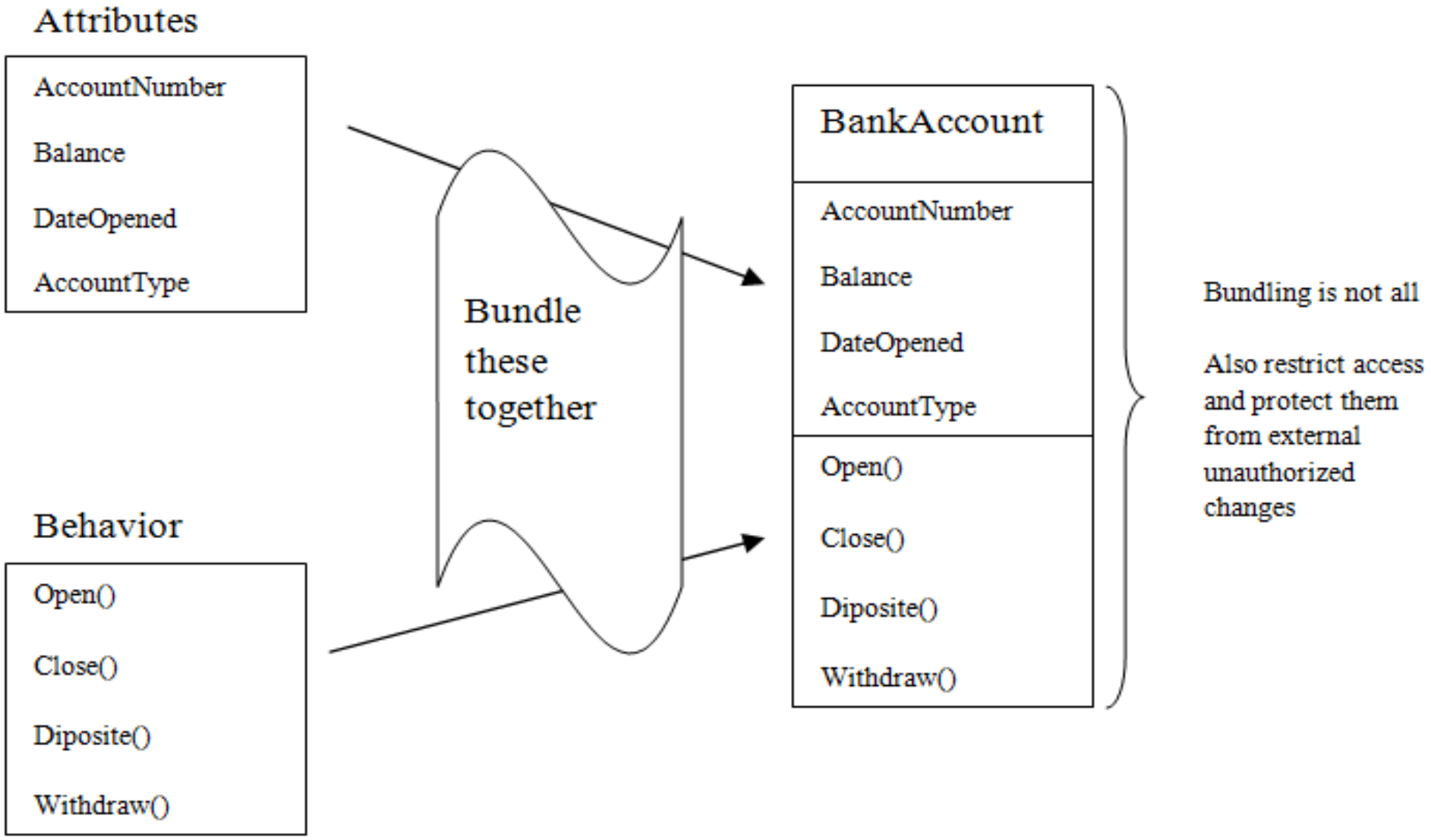
It is the idea of surrounding something, not just the keep the content together, but also to protect the content

Encapsulation can be defined as :

"The physical localization of features (e.g. properties, behaviors) into a single black-box abstraction that hides their implementation (and associated design decisions) behind a public interface"

Encapsulation

Example



Encapsulation

Points to remeber

- It is the approach of putting related things together
- Distinguishes between interface and implementation – exposes interface, hides implementation
- Encapsulation is often referred to as "information hiding", making it possible for the clients to operate without knowing how the implementation fulfills the interface
- It is possible to change the implementation without updating the clients as long as the interface is unchanged

Structure

In “C”; it is possible put related items together, even if they are heterogeneous, in a structure.

Structures thus encapsulate related data, but it does not provide a mechanism by which we can also specify how the data can be acted upon

```
struct Person {
    char name[20];
    int age;
}
void Read(Person &x){
    cout << "enter name and age
:";
    cin >> z.name >> z.age;
}
```

```
void Write(const Person & y) const
{
    cout << "Name : "<< y.name << "
age : " << y.age << endl;
}
int main() {
    Person p;
    Read(p);
    p.age = -10; // it is possible
    Write(p)
    return 0;
}
```

Structure

Contd...

In "C++"; it is possible put not only data, but also operation with in a structure. It is new concept in C++

But in structure, by default all members are **Public** and therefore can be access by client program.

```
struct Person {
    char name[20];
    int age;
    void Read();
    void Write() const;
}
void Person::Read(){
    cout << "enter name and
age :";
    cin >> name >> age;
}
```

```
void Person::Write() const {
    cout << "Name : "<< name << " age
: " << age << endl;
}
int main() {
    struct Person p;
    p.Read();
    p.age = -10; // Still it is
possible
    p.Write()
    return 0;
}
```

Class

The extended feature of structure in C++, we would prefer to call as Class where we can have the **Access Specifier**.

Note that in Class, by default all members are **Private**

```
class Person {
    private : // Even if not specified
        char name[20];
        int age;
    public :
        void Read();
        void Write() const;
}
void Person::Read(){...}
void Person::Write() const {...}
```

```
int main() {
    Person p;  // p is an object of the class Person
    p.Read(); // p implicitly passed by reference

    p.age = -10; // It is NOT possible

    p.Write() // p is passed as a constant object
    return 0;
}
```

Class

Few points to note

- Encapsulates data and functions
- Same as structure, but all the members are private by default
- An object is an instance of the Class
- Size of the object depends only on the data members of the Class ad their layout and does not depends on the member functions
- Invoking member functions of the class is resolved at compile time and therefore no extra overhead at run time

Access Specifier

Private : default specifier for the class, can be access only by member function of the class or friend function (*This point will be revisited while discussing friend function*).

Public : can be access from the client function also

Protected : can be access from member function of the same class or any of it's publicly derived class – can not be access by the client function (*This point will be revisited while discussing inheritance*).

Static Members

A Class may have static data members and static member function

- A static data member belongs to the class and not to each object – any information related to the class can be maintained in the static data member
- All the objects of the same class will share the same static data member
- A static member is declared in the class definition like any other non-static data member, but it has to be defined in one of the implementation files explicitly
- A static member function can only access the static data members
- A non-static member function can also access the static data members of the class

Static Members

Example

```
class Person {
    private :
        char * name;
        int age;
        static int personCount;
    public :
        Person() { personCount++ } // Constructor accesses the static
data member
        ...
        ~Person() { personCount-- } // Destructor accesses the static
data member
        static void DisplayNumOfPerson();
};

int  Person::personCount = 0; // definition of the static data member
```

Static Members

Example contd...

```
// definition of the static member function
void Person::DisplayNumOfPerson() {
    cout << "Number of Person : " << personCount << endl;
}
int main() {
    Person::DisplayNumOfPerson(); // displays 0
    Person a;
    a.DisplayNumOfPerson(); // displays 1
    {
        Person b;
        b.DisplayNumOfPerson(); // displays 2
        a.DisplayNumOfPerson(); // displays 2
    }
    Person::DisplayNumOfPerson(); // displays 1
}
```

'This' pointer

- The keyword this identifies a special type of pointer.
- If an Object of a Class is created that has a non-static member function -
  - When the non-static member function is called – the keyword ***this*** in the function body stores the address of the Object (acts as a stack variable).
  - When a non-static member function is called, the ***this*** pointer is passed as an extra argument (hidden).
- Its not possible to declare the this pointer or make assignments to it
- A static member function does not have a this pointer

'This' pointer

Example

```
class Person {
    private :
        char name[20];  int age;
    public :
        void Read();
        void Write() const;
}
void Person::Read(Person * this){
    cout << "enter name and age :";
    cin >> this->name >> this->age;
}
void Person::Write(const Person * this) const {
    cout << "Name : "<< (*this).name << endl;
    cout << " age : " << (*this).age << endl;
}
```

Default member functions of a Class

- What are the member-functions a class has by default ?
- By default, if not implemented by the user, the compiler add some member functions to the class. Those are below four -
  - Default Constructor A constructor which takes no arguments is called Default Constructor
  - Destructor
  - Copy constructor When a object is initialized with already existed object is called copy constructor
  - Assignment operator When a object is assigned with already existed object is called Assignment constructor

Note : This is true with some exceptions

Constructor

An object of the class can be initialize by special function called the constructor.

- Is a member function which is used to initialize the object
- Has the same name as the class
- Is invoked when an object is created
  - Is NOT invoked when a pointer of an Class is defined
  - Is invoked when an object is created dynamically
  - Is invoked as many times as the member of the elements in an array of objects
- Has no return type
- Can have parameter and default parameter
- Can be overloaded
- A constructor which takes no arguments is called Default Constructor
- A constructor can be private (will be discussed later)

Private Constructor

```
class Base {
    int a;
    static int object_cnt;
    Base() {
        a = 3030;
    }
    public:
    void display() {
        cout<<"val: "<<a<<"\n";
    }
    static Base* getInstance() {
        if(object_cnt >= 1) {
            cout<<"Its Singleton Class\n";
            return NULL;
        }
        ++object_cnt;
        Base* ptr = new Base;

        return ptr;
    }
};

int Base::object_cnt = 0;
```

```
int main()
{
    Base *basePointer = Base::getInstance();
    if(basePointer)
        basePointer->display();

    basePointer = Base::getInstance();
    if(basePointer)
        basePointer->display();


    return 0;
}
```

Private constructor with Singleton class

Destructor

Any clean-up operation of an object can be done by another special function called the destructor.

- Is a member function which is used to release resources
- Has the same as the class, preceded by ~
- Is invoked when an object is removed
  - Not invoked when a pointer of a Class goes out of scope
  - Invoked when a dynamically allocated object is deleted
  - Invoked as many time as the number of elements in an array when array goes out of scope
- Has no return type
- Can have no parameters
- Can not be overloaded

we can differentiate 2 fn of same name with its return type or parameter
but here u cant put parameter nor return type. So no overloading

Constructor and Destructor

Example

```
class Person {
    private :
        char name[20];
        int age;
    public :
        Person(); // default
        Person(char *, int=20);
        ~Person();
        void Read();
        void Write() const;
}
Person::Person(){
    name[0] = '\0'; age = 0;
}
```

```
Person::Person(char *s, int n){
    strcpy(name, s); age = n;
}
Person::~Person(){
    cout << "destructor called" <<
endl;
}
int main() {
    Person p;
    Person q("Raman", 53);
    Person * ptr = Null;
    ptr = new Person("Sundar");
    delete ptr;
}
```

Initialization List

- Initialization lists are used to initialize the data members of the class
- They can be used only in the constructor and no other member function
- Initialization list is executed before the body of the constructor is executed

Example code :

```
Person::Person() : age(0) {
    name[0] = '\0';
}
Person::Person(char *s, int n) : age(n) {
    strcpy(name, s);
}
```

Initialization List

Few points

- Used in constructor to initialize data members of the class
- Invoked in the order of declaration in the class and not in the order of the occurrence in the initialization list
- Efficient compared to making assignment
- Necessary to initialize constant or reference data member of the class

Some more points will be discussed while discussing inheritance

Dynamic Memory allocation using Constructors and Destructors

```
class Person {
    private :
        char * name; // a pointer
        int age;
    public :
        Person(); //default
        Person(char *, int);
        ~Person();
        void Read();
        void Write() const;
}
Person::Person(): name(NULL),
age(0) { }
```

```
Person::Person(char *s, int
n):age(n){
    name = new char[strlen(s)+1];
    strcpy(name, s);
}

Person::~Person(){
    delete []name;
}

int main() {
    Person p;
    Person q("Raman", 53);
}
```

Copy Constructor

There are instances in the code where an object is initialize with an existing object.

In such case, a special constructor called the copy constructor gets invoked.

```
int main() {
    Person p("raman", 53);
    p.write();
    Person q(p);
    // q is a new object instantiated by an existing object p,
  equivalent to "Person q = p;"
    q.write();
}
```

A default copy constructor provided by the compiler that does a member wise copy (shallow copy). This will not work if the object has a pointer.

Copy Constructor

Example

In that case programmer is required to provide an implementation of copy constructor if the object has some resource (for example : pointer data member having dynamically allocated memory) – provide deep copy instead of shallow copy.

```
class Person {
    private :
        char * name;
        int age;
    public :
        Person(); //default
        Person(char *, int);
        Person(const Person &); // object is passed by
  reference
        ~Person();
        void Read();
        void Write() const;
}
```

Copy Constructor

Example Contd...

```
Person::Person(const Person & rhs): age(rhs.age){
    name = new char[strlen(rhs.name)+1]),
    strcpy(name, rhs.name);
}

int main() {
    Person p("raman", 53);
    p.write();

    Person q(p); // q is a new object instantiated by an existing
object p using copy constructor
    q.write();

    Person r = p; // r is a new object instantiated by an existing
object p using copy constructor
    r.write();

}
```

Assignment Operator

There are instances in the code where an object is initialize with an existing object using assignment operator.

```
int main() {
    Person p("raman", 53);
    p.write();
    Person q; // q is a new object instantiated
    q = p; // q object updated by an existing object p using
  assignment operator
    q.write();
}
```

A default implementation of assignment operator is provided by the compiler that does a member wise copy (shallow copy). This will not work if the object has a pointer.

Assignment Operator

Example

In that case programmer is required to provide an implementation of assignment operator if the object has some resource (for example : pointer data member having dynamically allocated memory) – provide deep copy instead of shallow copy.

```
class Person {
    private :
        char * name;
        int age;
    public :
        Person(); //default
        Person(char *, int);
        Person(const Person &);
        Person & operator=(const Person &);
        ~Person();
        void Read();
        void Write() const;
};
```

Assignment Operator

Example Contd...

```
Person & Person::operator=(const Person &rhs) {
    name = new char[strlen(rhs.name)+1];
    strcpy(name, rhs.name);
    age = rhs.age;
    return *this;
}
int main() {
    Person p("raman", 53);
    p.write();

    Person q; // q is a new object instantiated
    q = p; // q object updated by an existing object p using
assignment operator
    q.write();

    p = p; // Also valid operation, but has side effect
    Person r("ravi", 33);
    r = p; // Also valid operation, but has side effect
}
```

Assignment Operator

Enhanced code

```
int main() {
    Person p("raman", 53);
    p = p; // This is valid operation, but may cause memory leak

    Person r("ravi", 33);
    r = p; // This is valid operation, but may cause memory leak
}

Person & Person::operator=(const Person &rhs) {
    if(this == &rhs) // self checking
        return *this;
    delete []name; // remove whatever existed before

    name = new char[strlen(rhs.name)+1];
    strcpy(name, rhs.name);
    age = rhs.age;
    return *this;
}
```

Automatic type conversion

- In C and C++, if the compiler sees an expression or function call using a type that isn't quite the one it needs, it can often perform an automatic type conversion from the type it has to the type it wants.
- In C++, this same effect for user-defined types can be achieved by defining automatic type conversion functions.

These functions come in two flavours:

1. a particular type of constructor
2. an overloaded **operator**.

Constructor conversion

If you define a constructor that takes as its single argument an object (or reference) of another type, that constructor allows the compiler to perform an automatic type conversion.

```
class One {
  public:
    One() {...}
};
class Two {
  public:
    Two(const One&) {...}
};
void f(Two) {...}
int main() {
  One one;
  f(one); // Wants a Two, has a One
}
```

Preventing constructor conversion: Using keyword explicit

```
class Two {
  public:
    explicit Two(const One&) {...}
};

int main() {
  One one;
  //! f(one); // No auto conversion allowed
  f(Two(one)); // OK -- user performs conversion
}
```

Operator conversion

Create a member function that takes the current type and converts it to the desired type using the **operator** keyword followed by the type you want to convert to.

This form of operator overloading is unique because you don't appear to specify a return type – the return type is the *name* of the operator you're overloading

```
class Three {
  int elem;
  public:
    Three(int i = 0, int j = 10) : elem(i) {}
};
class Four {
  int num;
  public:
    Four(int x) : num(x) {}
    operator Three() const {
        return Three(num); }
};
```

```
void g(Three) {...}
int main() {
  Four four(1);
  g(four);
  g(1); // Calls
Three(1,10)
}
```

Friend Function

There are instance where a function might be required to access the data members of a class even though they are private and this function can not be made member of a class.

In such case, we declare that this function is a **friend** of the class. A friend function can access the private members of the class to which it is a friend.

Note that a friend function can be member function of one class also. As well it can be friend to entire class also.

Friend Function

Example code

Example code :

```
class Person {
    private :
    char * name; int age;
    public :
    ...
    friend void doctor(Person &);
    // friend declaration
generally  placed in public
section
};
```

```
void doctor(Person & p) {
    cout << "Administrating age
reduction tonic" << endl;
    p.age = p.age - 10;
    // accessing private member
}

int main() {
    Person p("suparman", 50);
    p.write();
    doctor(p);
    p.write();
    return 0;
}
```

Friend Class

There are instances where a class might require to use another class – the former class may want to access private members of the latter class. The former class is made friend to the latter class.

Example code : Please complete by your own

```
class Node {
    private :
        int info;
        Node* link;
        Node(int) //
private constructor
        friend class Queue;
};
```

```
class Queue {
    public :
        Queue();
        ~Queue();
        void add(int); // Add at rear
        void remove(); // Remove from
front
        bool isEmpty();

    private :
        Node * f, *r;
        // front and rear pointers
};
```

Operator Overloading

Two ways to overload an operator. Using -

1. global overloaded operators
2. member overloaded operators

- One of the most convenient reasons to use global overloaded operators instead of member operators is that in the global versions, automatic type conversion may be applied to either operand
- Whereas with member objects, the left-hand operand must already be the proper type.

Reflexivity of operators

```
class Number {
  int num;
  public:
    Number(int i = 0) : num(i) {}
    const Number
      operator+(const Number&) const;
    friend const Number
      operator-(const Number&, const Number&);
};
const Number
Number::operator+(const Number& n) const {
    return Number(num + n.num);
}
const Number
operator-(const Number& n1, const Number& n2){
   return Number(n1.num - n2.num);
}
```

```
int main() {
  Number a(47), b(11);
  a + b; // OK
  a + 10; // 2nd arg
converted to Number
  //! 10 + a; // Wrong!
1st arg not of type
Number
  a - b; // OK
  a - 10; // 2nd arg
converted to Number
  10 - a; // 1st arg
converted to Number
}
```

Operator Overloading

There are several other aspects of operator overloading.
Those are for your detail study !!

When an operator is overloaded with multiple jobs, it is known as operator overloading.
It is a way to implement compile time polymorphism.

Rules to Remember
Any symbol can be used as function name

- if it is a valid operator in C language
- If it is preceded by operator keyword

You can not overload sizeof and ?,: operator

```
  void showData()
  { cout<<"\na="<<a<<"  b="<<b;}
  Complex add(Complex c)
  {
   Complex temp;
   temp.a=a+c.a;
   temp.b=b+c.b;
   return(temp);
  }
};
void main()
{
 clrscr();
 Complex c1,c2,c3;
 c1.setData(3,4);
 c2.setData(5,6);
 c3=c1.add(c2);
 c3.showData();
 getch();
}
```

```
  { a=x; b=y;}
  void showData()
  { cout<<"\na="<<a<<"  b="<<b;}
  Complex operator +(Complex c)
  {
   Complex temp;
   temp.a=a+c.a;
   temp.b=b+c.b;
   return(temp);
  }
};
void main()
{
 clrscr();
 Complex c1,c2,c3;
 c1.setData(3,4);
 c2.setData(5,6);
 c3=c1+c2;
 c3.showData();
 getch();
}
```

Class Template

There are many data structures which have certain properties independent of the components they contain. For example :

- Queue : It has property of FIFO, doesn't not matter what type of component we put in the queue
- Stack : It has property of LIFO, doesn't not matter what type of component we put in the stack

- In c++ we can write such generic data structure using template class concept.
- In this approach, the class as well as the member functions are generated at compile time based on the definition of the object in the client code.

Like -

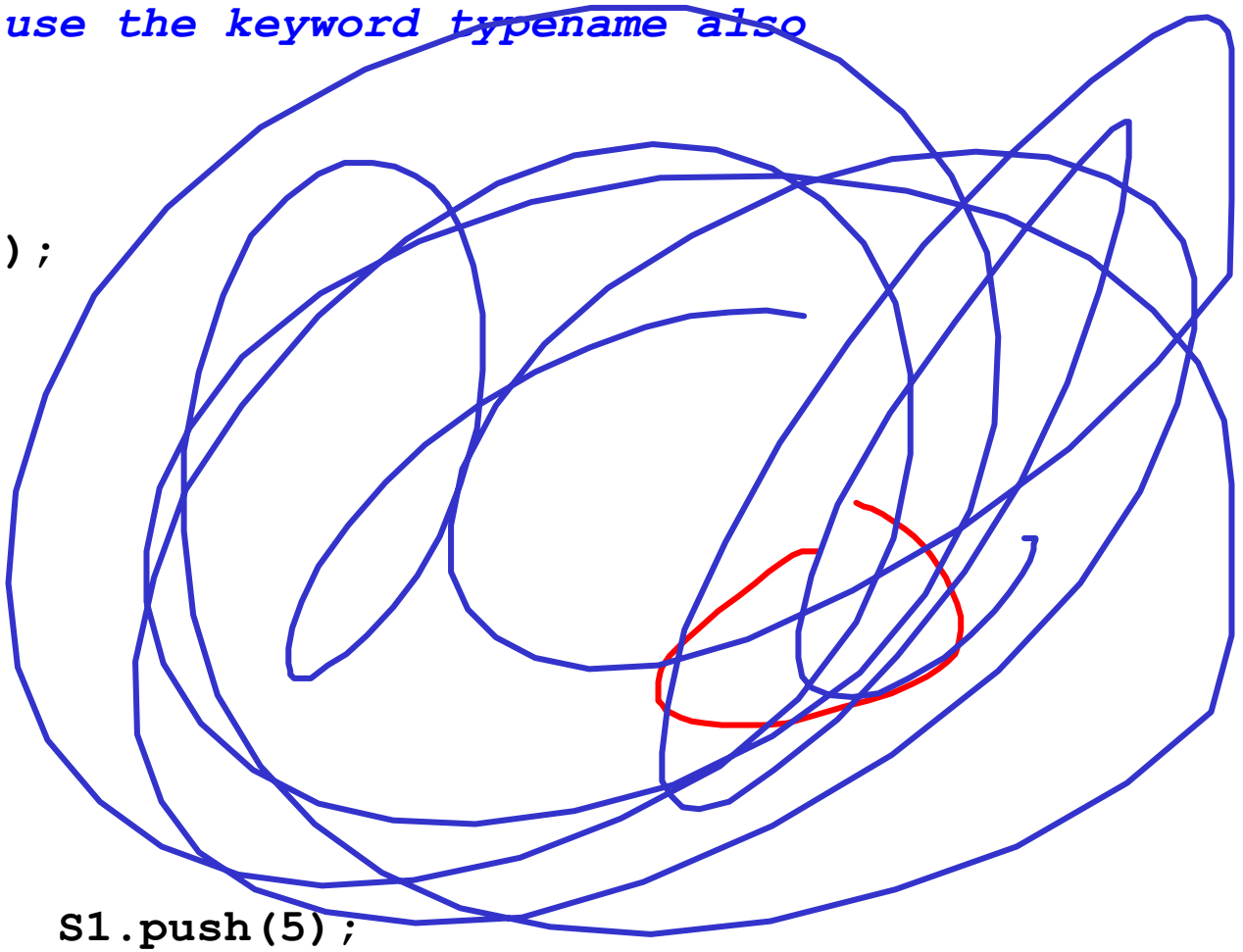
```
MyStack<int> s1(10); // stack of 10 integer
MyStack<double> s2(5); // stack of 10 integer
```

To carry out this process, the code has to be exposed to the client.

Class template

Example

```
template <class T> // can use the keyword typename also
class MyStack {
    public :
        MyStack(int m = 10);
        void Push(T x);
        T Pop();
        ...
    Private :
        T * elementList;
        ...
};
int main() {
    MyStack<int> S1(10);      S1.push(5);
    MyStack<double> S1(5);    S2.push(9.33);
}
```



Inheritance

Inheritance is used to set-up a class hierarchy. It is a method of reuse, but this is not it's main purpose.

When do we use inheritance ?

A derived class should pass the litmus test of "**is a**"

- 2 wheeler "is an" automobile. OK
- 4 wheeler "is an" automobile. OK
- Car "is a" 4 wheeler. OK
- Steering wheel "is a" Car. Not OK
- Event "is a" Date. Not OK

The last two are example of **Composition** or "**has a**" hierarchy, hence using Inheritance mechanism here is a misuse

- Car "has a" steering wheel. OK
- Event "has a" date. OK

Inheritance

```
class Person {
    private :
        char * name;
        int age;
    public :
        Person(); //default
        Person(char *, int);
        Person(const Person &);
        Person & operator=
                (const Person &);
        ~Person();
        void Read();
        void Write() const;
}
```

Example

```
class Employee : public Person {
    private :
        int empId;
        Int salary
    public :
        Employee(...);
        Employee(const Employee &);
        Employee & operator=
                (const Employee &);
        ~Employee();
        void ReadEmp();
        void WriteEmp() const;
}
```

Inheritance

Example

Person class is called the base class and Employee class is called the derived class

Employee class inherits all the methods of base class Person except the Constructor, Destructors, Copy constructor and Assignment Operator

Any of the inherited functions can be directly invoked using an object of class Employee

Protected Access Specifier

Protected : can be access from member function of the same class or any of it's publicly derived class – can not be access by the client function.

Private members would always be hard-and-fast **private**, but in real projects there are times when you want to make something hidden from the world at large and yet allow access for members of derived classes. The **protected** keyword is used for this reason.

"This is **private** as far as the class user is concerned, but available to anyone who inherits from this class."

Protected example

The best approach is to leave the data members **private** – you should always preserve your right to change the underlying implementation. You can then allow controlled access to inheritors of your class through **protected** member functions:

```
class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int calcValue(int m) const {
    return m*i;
  }
};
```

```
class Derived : public Base {
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};

int main() {
  Derived d;
  d.change(10);
}
```

Constructor & Destructor

- When an object of the derived class is created, the control is transferred to the constructor of derived class
- The base class sub-objects are initialized in the initialization list by invoking the constructor of the base class
- If the base class constructor is not explicitly invoked in the initialization list, then the default constructor of the base class is invoked
- After the base class sub-object are initialized by executing the constructors of the corresponding base classes, the data members of the derived class are initialized
- Destructor are executed in the reverse order of constructor. The derived class destructors will be executed before that of base class destructor

Constructor & Destructor

Example

```
class CBase {
    public :
        CBase() { cout << "In base class
   constructor" << endl; }
        ~CBase() { cout << "In base class
   destructor" << endl; }
};


class CDerived : public CBase {
    public :
        CDerived() { cout << "In derived class
   constructor" << endl; }
        ~CDerived() { cout << "In derived class
   destructor" << endl; }
};
```

```
int main() {
    CDerived d;
}
```

Output :

In base class constructor

In derived class constructor

In derived class destructor

In base class destructor

Constructor & Destructor

Example II

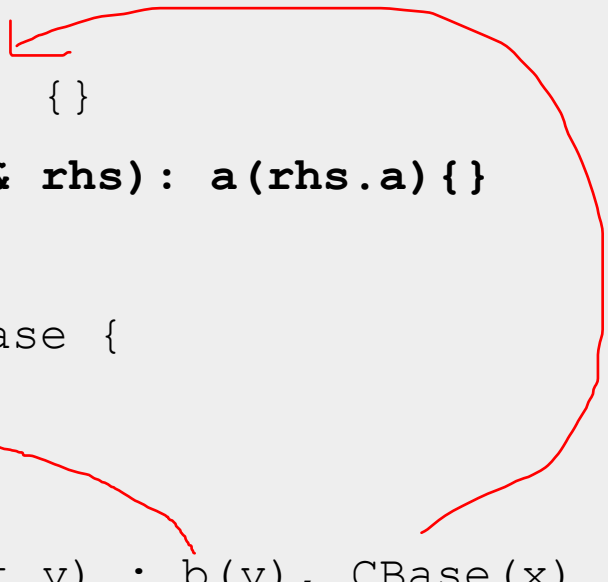
```
class CBase {
        int a;
    public :    ...
        CBase(int z) : a(z) { }
        void Display() { cout << "a : " << a <<
   endl; }
};
class CDerived : public CBase {
        int b;
    public :    ...
        CDerived(int x, int y) : b(y), CBase(x) { }
        void Display() {
            CBase::Display();
            cout << "b : " << b << endl;
        }
};
```

```
int main() {
    CDerived d(1,2);
    d.Display();
}
```

Copy Constructor

Example

```
class CBase {
        int a;
    public :    ...
        CBase(int z) : a(z) {}
        CBase(const CBase & rhs): a(rhs.a){}
};
class CDerived : public CBase {
        int b;
    public :    ...
        CDerived(int x, int y) : b(y), CBase(x) {}
        CDerived(const CDerived & rhs):b(rhs.b),
   Cbase(rhs)      {}
        // note how the base class constructor is
   invoked - derived class object to a base class
   reference
};
```



```
int main() {
    CDerived d1(1,2);
    d1.Display();
    CDerived d2(d1);
    d2.Display();
}
```

Initialization List

Few points

- Necessary to invoke the constructor of the base class
- Base class constructors are invoked in the order of derivation and not in the order of occurrence in the initialization list
- If a base class constructor is not specified, default base class constructor will be invoked
- Base class constructor are invoked before the members of the class are initialized

Assignment Operator

Example

Do on your own

```
int main() {
    CDerived d1(1,2);
    d1.Display();
    CDerived d2(d1);
    d2.Display();
}
```

CDrived d2;
d2 = d1;
//in cdeive class we have make default constructor

Overloading member functions from base and derived classes

```
class CBase
{   ...
    public :    ...
        void fn() { cout << "In base
class" << endl; }

};

class CDerived : public CBase
{   ...
    public :    ...
        void fn() { cout << "In derived
class" << endl; }

};
```

```
int main() {
    CBase oB, *pB = NULL;
    CDerived oD, *pD = NULL;

    oB.fn();

    oD.fn();

    oD.CBase::fn();


    pD = &oB;
    pD->fn();


    pB = &oD;
    pB->fn();

}
```

Overloading member functions from base and derived classes

```
class CBase
{    ...
    public :     ...
        void fn() { cout << "In
base class" << endl; }

};

class CDerived : public CBase
{    ...
    public :     ...
        void fn() { cout << "In
derived class" << endl; }

};
```

Note : Base class pointer or reference can point or refer to a derived classs object but not vice versa

```
int main() {
    CBase oB, *pB = NULL;
    CDerived oD, *pD = NULL;

    oB.fn(); // calls fn of Base
class

    oD.fn(); // calls fn of Derived
class
    oD.CBase::fn(); // calls fn of
Base class

    // pD = &oB; // will not compile
    // pD->fn();


    pB = &oD;
    pB->fn(); // calls fn of the
base class
}
```

Understanding

Base class pointer or reference can point or refer to a derived class object

If I tell you I have a dog, you can safely assume that I have a pet.
If I tell you I have a pet, you don't know if that animal is a dog, it could be a cat or maybe even a giraffe. Without knowing some extra information you can't safely assume I have a dog.

similarly a derived object is a base class object (as it's a sub class), so it can be pointed to by a base class pointer. However, a base class object is not a derived class object so it can't be assigned to a derived class pointer.

Derived class pointer can not automatically be made to point to the base class object

- Generally derived class may have additional members compared to the base class. If the derived class pointer were allowed to point to the base class object, and if it tries to access the additional members using this pointer, the compiler will not be able to give an error.
  - If this is allowed, this may cause **dangling reference** at run time.

Function Overriding or Dynamic Polymorphism

Even though base class pointer can point to an object of the derived class, the base class function is invoked as the pointer by type is a pointer to the base class.

- This happens as function calls are generally resolved at compile time

Many a times, it is desirable to postpone the resolution of the call until run times and the function should be invoked based on the object to which the pointer points to rather than the type of the pointer.

- This is achieved by declaring the function to be **virtual**

This concept, where the function calls are resolved at run time based on the object to which they point, is called **Dynamic Polymorphism**.

- In C++, it is implemented using **virtual function**

Function Overriding

Example

```
class CBase
{    ...
     public :     ...
          virtual void fn() {
          cout << "In base class" <<
endl; }

};
```

all fn named: fn() will be virtual fn

```
class CDerived : public CBase
{    ...
     public :     ...
          void fn() {
          cout << "In derived class"
<< endl; }

};
```

```
int main() {
     CBase oB, *pB = NULL;
     CDerived oD, *pD =
NULL;

     oB.fn();

     oD.fn();


     pB = &oD;
     pB->fn();
}
```

Function Overriding

Example

```
class CBase
{    ...
     public :     ...
          virtual void fn() {
          cout << "In base class"
<< endl;
     }

};

class CDerived : public CBase
{    ...
     public :     ...
          void fn() {
          cout << "In derived
class" << endl;
     }
};
```

```
int main() {
     CBase oB, *pB = NULL;
     CDerived oD, *pD = NULL;

     oB.fn(); // calls fn of Base
class

     oD.fn(); // calls fn of
Derived class

     pB = &oD;
     pB->fn(); // calls fn of the
derived class
}
```

Start From Here!

Function Overriding

Few points

Few points :

- Should be member function of the class
- Only in inheritance
- Function should be virtual
- Signature of the functions should be exactly same
- Can extends the functionality of the base class function
- *Extra overhead at run time of a per-class table and a per-object pointer*
- *Extra overhead at run time of a de-referencing of pointer*

Last 2 points are discussed in the next section

Virtual Table / VTBL

Virtual Table is a lookup table of function pointers used to dynamically bind the virtual functions to objects at runtime.

Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it's own virtual table as a secret data member.

It is not intended to be used directly by the program, and as such there is no standardized way to access it. This table is set up by the compiler at compile time.

A virtual table contains one entry as a function pointer for each virtual function that can be called by objects of the class.

Virtual Pointer / VPTR

This vtable pointer or _vptr, is a hidden pointer added by the Compiler to the base class. And this pointer is pointing to the virtual table of that particular class.

This _vptr is inherited to all the derived classes.

Each object of a class with virtual functions transparently stores this _vptr.

Call to a virtual function by an object is resolved by following this hidden _vptr.

Understanding VTBL and VPTR

```
class CBase
{   ...
    public :    ...
        virtual void fn1() { cout <<
"Fn1 in base class" << endl; }

        void fn2() { cout << "Fn2 in
base class" << endl; }
};

class CDerived : public CBase
{   ...
    public :    ...
        void fn1() { cout << "Fn1 in
derived class" << endl; }

        void fn2() { cout << "Fn2 in
derived class" << endl; }
};
```

```
int main() {
    CBase oB, *pB = NULL;
    CDerived oD;

    pB = &oD;
    pB->fn1(); // calls fn1 of
the derived class

    pB->fn2(); // calls fn2 of
the base class - as fn2 in not a
virtual function in base class

}
```

How C++ implements late binding

To accomplish this, the typical compiler creates a single table (called the VTABLE) for each class that contains **virtual** functions. The compiler places the addresses of the virtual functions for that particular class in the VTABLE. In each class with virtual functions, it secretly places a pointer, called the *vpointer* (abbreviated as VPTR), which points to the VTABLE for that object. When you make a virtual function call through a base-class pointer (that is, when you make a polymorphic call), the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling the correct function and causing late binding to take place.

All of this – setting up the VTABLE for each class, initializing the VPTR, inserting the code for the virtual function call – happens automatically, so you don't have to worry about it. With virtual functions, the proper function gets called for an object, even if the compiler cannot know the specific type of the object.

Pure Virtual Functions and Abstract Base Class

If the base class provides only the interface of the methods, without implementation, then the **virtual functions** of the base class are made **pure virtual** – no definition is provided for the functions and the pointer to the function is grounded.

Such a Class is called an **Abstract Class**.

- Abstract class can not be instantiated
- A derived class of a Abstract class becomes a **concrete** class only if all the pure virtual functions of the base class are overridden by this class or any of its ancestors

A Class with only pure virtual function provides interface only

- Each of the derived classes will inherit the signature of the methods and override them in their classes

Virtual functions supporting dynamic dispatch provides flexibility to the programmer, the client code becomes more robust and does not break down when new classes are derived.

Pure Virtual Functions and Abstract Base Class

```
class Shape {
    public :
        virtual void Read() = 0;
        virtual double Area() = 0;
};

class Rectangle : public Shape {
        double length, breadth;
    public :
        void Read(){
            cin >> length >>
breadth;
        }
        double Area() {
            return length*breadth;
        }
};
```

Example

```
class Triangle : public Shape {
        double base, height;
    public :
        void Read(){
            cin >> base >> height;
        }
        double Area() {
            return 0.5*base*height;
        }
};
int main() {
    Double totalArea = 0;
    Shape *p[2];
    p[0] = new Rectangle();
    p[1] = new Triangle();
    for(int i = 0; i < 2; i++)
p[i]->Read();

    for(int i = 0; i < 2; i++)
        totalArea += p[i]->Area();
}
```

Virtual Destructor

Whenever the constructor of the derived class is non-trivial, the base class destructor should be **virtual** to force the execution of the derived class destructor before that of the base class

Virtual Destructors

Example

```
class CBase {
    public :
        CBase() {
            Cout << "Constructor of
the base class" << endl;
        }

        ~CBase() {
            Cout << "Destructor of
the base class" << endl;
        } // Incorrect way
};


int main() {
    CBase * ptr = new CDerived();
    delete ptr;
}
```

```
class CDerived : public CBase {
    int * p;
    public :
        CDerived () {
            cout << "Constructor of
the derived class" << endl;
            p = new int;
        }
        ~CDerived() {
            cout << "Destructor of
the derived class" << endl;
            delete p;
        }
}
```

Outputs :

```
Constructor of the base class
Constructor of the derived class
Destructor of the base class
```

Virtual Destructors

Example contd

```
class CBase {
    public :
        CBase() {
            Cout << "Constructor of
the base class" << endl;
        }

        virtual ~CBase() {
            Cout << "Destructor of
the base class" << endl;
        }
};


int main() {
    CBase * ptr = new CDerived();
    Delete ptr;
}
```

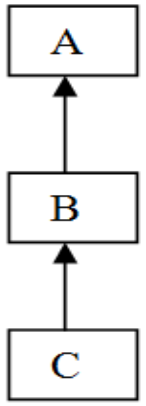
```
class CDerived : public CBase {
    int * p;
    public :
        CDerived () {
            cout << "Constructor of
the derived class" << endl;
            p = new int;
        }
        ~CDerived() {
            cout << "Destructor of
the derived class" << endl;
            delete p;
        }
}
```

Outputs :

Constructor of the base class
Constructor of the derived class
Destructor of the derived class
Destructor of the base class

Multi-level Inheritance

Example




```
class A {
    public :
        void fn() { cout << "In A"
<< endl; }
}

class B : public A {
    public :
        void fn() { cout << "In B"
<< endl; }
}

class C : public B {
    public :
        void fn() { cout << "In C"
<< endl; }

}
```

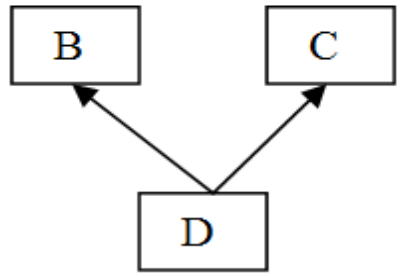
```
int main() {
    C oC;

    oC.fn(); // fn of class C

    oC.B::fn(); // fn of class B

    // oC.A::fn(); // ERROR
}
```

Class A is not direct base of Class C, hence function of same name of class A can not be called.

But other functions can be called.

Multiple Inheritance

Example




```
class B {
    public :
        void fn() { cout << "In
B" << endl; }
}

class C {
    public :
        void fn() { cout << "In
C" << endl; }
}

class D : public B, public C {
    public :
}
```

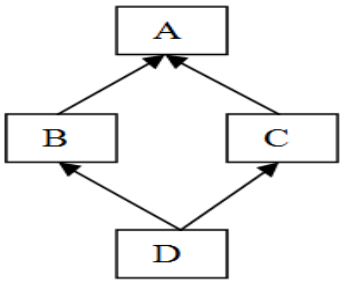
```
int main() {
    D oD;

    oD.fn(); // ERROR

    oD.B::fn(); // fn of class B

    oD.C::fn(); // fn of class C
}
```

When a class inherits from more than one class, then it is called multiple inheritance.

It can lead to ambiguity as the base classes may have the members with the same name.

In that case reference to function can be resolved upon the class name

Hybrid Inheritance



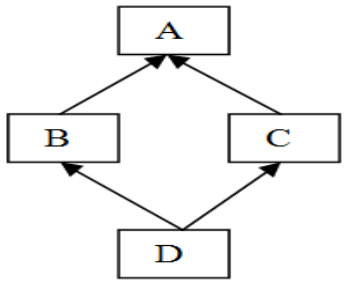

```
class A {
  public :
    void display() {
      cout << "Hello from A" << endl;
    }
};

class B : public A {
};

class C : public A {
};
```

```
class D : public B, public C{
};

int main()  {
    D object;

    //object.display();

    //object.A::display();
}
```

Compilation Error

```
test.cpp:20:12: error: request for member 'display' is ambiguous
     object.display();
            ^
test.cpp:7:10: note: candidates are: void A::display()
     void display (){
          ^
test.cpp:7:10: note:                 void A::display()
test.cpp:20:12: error: 'A' is an ambiguous base of 'D'
        object.A::display();
```

Hybrid Inheritance




```
class A {
        int a;
    public :
        A(int w) : a(w) {}
}

class B : public A {
        int b;
    public :
        B(int w, int x) : A(w),
b(x) {}
}

class C : public A {
        int c;
    public :
        C(int w, int y) : A(w),
c(y) {}
}
```

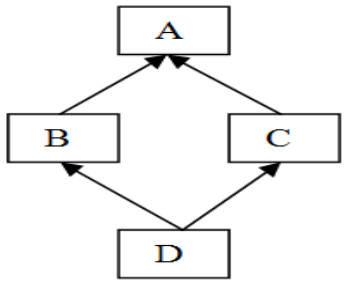
```
class D : public B, public C{
        int d;
    public :
        D(int w, int x, int y, int
z) : B(w, x), C(w, y), d(z) {}
}
```

Note : An object of classe D will have following structure

Base class subobject of class B Also has base class sub object of class A	a b
Base class subobject of class C Also has base class sub object of class A	a c
Members of clas D	d

Observe that the base class subobject of class A appears twice in object of class D

HI & Virtual Base Class




```
class A {
        int a;
    public :
        A(int w) : a(w) {}
};

class B : virtual public A {
        int b;
    public :
        B(int w, int x) : A(w),
b(x) {}
};

class C : public virtual A {
        int c;
    public :
        C(int w, int y) : A(w),
c(y) {}
};
```

```
class D : public B, public C{
        int d;
    public :
        D(int w, int x, int y, int
z) : B(w, x), C(w, y), A(w), d(z) {}
};
```

Note : An object of classe D will have following structure

Base class subobject of class A	a
Base class subobject of class B With a referenct to base class sub object of class A	&a b
Base class subobject of class C With a referenct to base class sub object of class A	&a c
Members of clas D	d

Observe that here Class D will have to explicitely invoke the constructor of class A. But class B and class C will **not** invoke the constructor of class A

What is the difference between C, C++ and Java?

Last updated on Nov 26,2019 *41.7K Views*


Aayushi Johari
A technophile who likes writing about different technologies and spreading knowledge.

Software development has seen transition like any domain out there. This has also resulted in the evolution of programming languages. C, C++, and **Java** are three languages that have defined programming paradigms with time and yet hold great value in the market. In this article, I will be comparing the differences between C, C++ and Java so you can choose one or more for a probable career or a **certification**.

Differences between C, C++ and Java

Metrics	C	C++	Java
Programming Paradigm	Procedural language	Object-Oriented Programming (OOP)	Pure Object Oriented Oriented
Origin	Based on assembly language	Based on C language	Based on C and C++
Developer	Dennis Ritchie in 1972	Bjarne Stroustrup in 1979	James Gosling in 1991
Translator	Compiler only	Compiler only	Interpreted language (Compiler + interpreter)
Platform Dependency	Platform Dependent	Platform Dependent	Platform Independent
Code execution	Direct	Direct	Executed by JVM (Java Virtual Machine)
Approach	Top-down approach	Bottom-up approach	Bottom-up approach
File generation	.exe files	.exe files	.class files
Pre-processor directives	Support header files (#include, #define)	Supported (#header, #define)	Use Packages (import)
keywords	Support 32 keywords	Supports 63 keywords	50 defined keywords
Datatypes (union, structure)	Supported	Supported	Not supported

Inheritance	No inheritance	Supported	Supported except Multiple inheritance
Overloading	No overloading	Support Function overloading (Polymorphism)	Operator overloading is not supported
Pointers	Supported	Supported	Not supported
Allocation	Use malloc, calloc	Use new, delete	Garbage collector
Exception Handling	Not supported	Supported	Supported
Templates	Not supported	Supported	Not supported
Destructors	No constructor neither destructor	Supported	Not supported
Multithreading/ Interfaces	Not supported	Not supported	Supported
Database connectivity	Not supported	Not supported	Supported
Storage Classes	Supported (auto, extern)	Supported (auto, extern)	Not supported

That's all with the differences between C, C++, and Java. I hope you are clear with the basic concepts of these wonderful programming languages and helped you in adding value to your knowledge.

Next up, let' us take a look at some sample program to display the differences between C, C++ and Java.

Sample Program in C, C++ and Java

Hello Word Program in C

```
#include<stdio.h> //header file for standard input output

main() //main method
{
clrscr(); //clears screen
printf("hello world"); //print statement
getch(); //get the character
}
```

Explanation: In the above code, you use header file <stdio.h> for standard input output to implement commands like printf and

Storage Classes in C++

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

- auto
- register
- static
- extern
- mutable

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
   int mount;
   auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
   register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and

implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

Live Demo

```
#include <iostream>

// Function declaration
void func(void);

static int count = 10; /* Global variable */

main() {
   while(count--) {
      func();
   }

   return 0;
}

// Function definition
void func( void ) {
   static int i = 5; // local static variable
   i++;
   std::cout << "i is " << i ;
   std::cout << " and count is " << count << std::endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
```

```
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.cpp

```
#include <iostream>
int count ;
extern void write_extern();

main() {
   count = 5;
   write_extern();
}
```

Second File: support.cpp

```
#include <iostream>

extern int count;

void write_extern(void) {
   std::cout << "Count is " << count << std::endl;
}
```

Here, *extern* keyword is being used to declare count in another file. Now compile these two files as follows −

```
$g++ main.cpp support.cpp -o write
```

This will produce **write** executable program, try to execute **write** and check the result as follows −

```
$./write
5
```

The mutable Storage Class

The **mutable** specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function.