by "Bjarne stroustrup" in early 1980s

C++ [procedural and OOPS language]

- efficiency
- "Very close to hardware"
- performance
- Extensive standard library

⇒ Ability to combine low-level memory manipulation (concept of pointers & references) with high-level Abstraction (OOPS support)

C++ use Cases

- Software/game development [Unreal Engine]
- Scientific Computing [Eigen Library]
- Embedded/operating system

OOPs [object oriented programming system]

- Class [blueprint to define an object]
- object [real-world entity] (something that has its own identity)

"OOPs 4 pillars/features"

ENCAPSULATION

(data + function packed into single unit)

```
eg. class Animal {
        name;
        void walk(){
        }
    };
```

ABSTRACTION

(hiding complex details) using visibility labels

```
eg. class Aimal{
    private:
        int frequency;
    };
```

INHERITANCE

"Re-usability" class Acquiring properties of another class

```
eg. class parent{
    protected:
        string name;
    };
class child : public parent{
    int age;
};
```

(name treated as private member of child)

POLYMORPHISM

"many forms"

Base class Attribute Labels	child class visibility labels in Inheritance: public	private	protected
public	public	private	protected
private	can't Access directly	Can't Access directly	Can't Access directly
protected	protected	private	protected.

polymorphism [many forms].

Compile time → Func^n overload [same name functions but differ in parameter types and number]

Compile time → operator overload → "giving specific functionality to an operator, Only Applicable on user defined datatypes"

Run-time → virtual functions (func^n overriding)

* function overriding: Same Name function defined in parent and child class;

Exception handling → "handling unwanted events that occur during run-time of program"

```
try{
    // if(error){
        throw "error";
}
Catch (string exception){
    //         → if type is Not
} known use ...
```

eg →

```
class Complex{
    real, imag;
    public:
    complex operator + (complex const & obj){
        complex result;
        result.real = real + obj.real;
        result.imag = imag + obj.imag;
        return res;
    }
}
main() { complex c3 = c1 + c2; }
```

(c2 → passed as argument; c1 → operate on this)

Compile time polymorphism

Base class object can only call the function belonging to its class not CHILD class.

Because this has been a "static linkage" made by compiler at compile time

generic programming.

```
template <typename T>
class demo {};
main(){ demo<int> object;
```

* so that's why we use virtual functions such that a base class pointer can refer to both objects belonging to base class as well as child class

```
class shape{
    public:
    virtual void getarea(){
        cout << "in shape";
    }
};
```

⇒

```
class circle : public shape{
    public:
    void getarea(){
        cout << "in circle";
    }
};
```

⇒

```
main(){
    shape *s1;
    circle c1(10);
    s1 = &c1;
    s1 → getarea(); // in circle
    s1 = new shape (10);
    s1 → getarea(); // in shape
```

* this linkage of call is made at run time that's why it's called run time polymorphism.

* policy based data structure: Just like user defined data structure that are Flexible and used for special purpose / business purpose.

* Bit set library: #include <bitset>, bitset <32> (7); // create a bitmap

0 ....00111 (31st bit ... 0th)

run command → >> g++ filename.cpp -o sample.exe

>> ./sample