

```
1
2 #include<iostream>
3 using namespace std;
4 // class : ( blue print of object )
5
6 class className{
7
8     // attributes :
9
10    // functions :
11
12 };
13 // accessibility
14 /*
15  ✨ access specifiers(modifiers)
16     public :   visible for all program
17     private :  visible into the class
18     protected : visible into the parent class and derived class (child)
19 */
20
21 /*
22
23     data members : any variable declared inside the class
24     members methods(functions) : any functions or procedure
25     declared inside the class
26     class members = data + methods
27
28 */
29
30 // properties : get/set
31 /*
32     one of the most useful functions that allow us access (read | update )
33     the private members into a class
34
35     ✨get : read
36     ✨set : update
37 */
38 // example :
39 class clsPerson
40 {
41
42 protected:
43     int v1 = 5;
44     int f1()
45     {
46         return 1;
47     }
48
49 private:
50     string cFirstName;
51     string cLastName;
52
53 public:
```

```

54     string getFullName()
55     {
56         return cFirstName + " " + cLastName;
57     }
58
59     // getters :
60     string getFirstName() { return cFirstName; }
61     string getLastName() { return cLastName; }
62
63     // setters :
64     void setFirstName(string firstName) { cFirstName = firstName; }
65     void setLastName(string lastName) { cLastName = lastName; }
66 };
67
68 // properties get and set through '=' (just for microsoft environment )
69 class className{
70     __declspec(property(get=getFunction,put=setFunction)) datatype varNameToShowUser;
71 };
72 // ✨Encapsulation
73 /*
74 In normal terms Encapsulation is defined as wrapping up of data and
75 information under a single unit.
76 In Object Oriented Programming, Encapsulation is defined as binding
77 together the data and the functions that manipulates them
78 */
79
80 // ✨Abstraction
81 /*
82 In simple terms, abstraction “displays” only the relevant attributes
83 of objects and “hides” the unnecessary details.
84 */
85
86 // ✨ constructor :
87 class className{
88     className(){
89         // code
90     }
91 };
92
93 // A constructor is a special type of member function that is called
94 //automatically when an object is created
95 /* types
96     empty : no parameters
97     parametrized with parameters
98     copy : used to initialize the members of a newly created object
99           by copying the members of an already existing object.
100 */
101
102 // ✨ destructor :
103 class className{
104     ~className(){
105         // code
106     }
107 };
108
109 /*

```

```

110     Destructor is an instance member function which is invoked automatically
111     whenever an object is going
112     to be destroyed. Meaning, a destructor is the last function that is going
113     to be called before an object is destroyed.
114 */
115 //example :
116 class cRectangle
117 {
118 private:
119     string firstName;
120     string lastName;
121
122 public:
123     // empty constructor :
124     cRectangle()
125     {
126         firstName = "";
127         lastName = "";
128     }
129
130     // parametrized constructor :
131     cRectangle(string firstName, string lastName)
132     {
133         this->firstName = firstName;
134         this->lastName = lastName;
135     }
136
137     // copy constructor :
138     cRectangle(cRectangle &copy)
139     {
140         firstName = copy.firstName;
141         lastName = copy.lastName;
142     }
143
144     string getFirstName()
145     {
146         return firstName;
147     }
148     string getLastName()
149     {
150         return lastName;
151     }
152
153     // destructor :
154     ~cRectangle()
155     {
156
157         cout << "good night : " << firstName << endl;
158     }
159 };
160
161
162 // ✨static members :
163 class className{
164     static varName;
165

```

```

166     };
167 // initialize a static variable :
168     type className varName =value;
169     /*
170     Static Member is a variable that is shared for all objects, any object modifies it
171     it get modified for all other objects.
172     */
173
174 //example :
175 class cA
176 {
177 private:
178     int var;
179     static int counter;
180
181 public:
182     cA()
183     {
184         var = 0;
185         counter++;
186     }
187
188     void print()
189     {
190         cout << "\n var = " << var << "\n";
191         cout << "counter = " << counter << "\n";
192     }
193 };
194
195 int cA::counter = 0;
196
197
198 // ✨ static functions :
199 class className{
200     static functionName(){
201         //code
202     }
203
204 };
205 /*
206 ✓ Static function is a function that is shared for all objects
207 ✓ Static Functions can be called at class level without a need to have an object.
208 ✓ No, Static methods can only access static members , because static methods can be called
  at
209     class level without objects, and non static members you cannot access them without having
210     object first.
211 */
212
213 // access to a static function :
214 int main(){
215     className::functionName();
216 }
217
218 // example :
219 class cA
220 {

```

```

221 private:
222     static int counter;
223
224 public:
225     cA()
226     {
227         counter++;
228     }
229     static int getCounter()
230     {
231         return counter;
232     }
233 };
234
235
236 // ✨ Inheritance :
237 /*
238     Inheritance: Inheritance is one in which a new class is created that
239     inherits the properties
240     of the already exist class. It supports the concept of code
241     reusability and reduces the length
242     of the code in object-oriented programming.
243 */
244 // base class / super class / parent class
245 class baseClass{
246
247 };
248 // sub class / derived class / child class
249 class derivedClass : modifiers className{
250
251 };
252
253 // access to function from the base class :
254 class baseClass {
255
256 public :
257     void functionExample(){
258         // code
259     }
260 };
261 class derivedClass : public baseClass {
262     void functionExample(){
263         baseClass::functionExample();
264         // added code
265     }
266 };
267
268 //example
269 class cPerson
270 {
271
272     int id;
273     string firstName;
274     string lastName;
275     string email;
276     string phone;

```

```

277
278 public:
279     // empty constructor :
280     cPerson()
281     {
282         id = 0;
283         firstName = "";
284         lastName = "";
285         email = "";
286         phone = "";
287     }
288     // parametrized constructor :
289     cPerson(int id, string firstName, string lastName, string email, string phone)
290     {
291
292         this->id = id;
293         this->firstName = firstName;
294         this->lastName = lastName;
295         this->email = email;
296         this->phone = phone;
297     }
298
299     // print function :
300     void print(bool isBaseClass = true)
301     {
302
303         cout << "\n_____ \n";
304         cout << "the id      : " << id << "\n";
305         cout << "the firstName : " << firstName << "\n";
306         cout << "the lastName  : " << lastName << "\n";
307         cout << "the email    : " << email << "\n";
308         cout << "the phone    : " << phone << endl;
309         if (isBaseClass)
310             cout << "_____ \n";
311     }
312
313     void sendEmail(string subject, string body)
314     {
315         cout << "\nThe following message sent successfully to email:" << email << "\n";
316         cout << "subject : " << subject << "\n";
317         cout << "body : " << body << "\n";
318     }
319
320     void sendSms(string sms)
321     {
322         cout << "\nThe following SMS sent successfully to phone:" << phone << "\n";
323         cout << sms << "\n";
324     }
325
326     int getId() { return id; }
327     string getFirstName() { return firstName; }
328     string getLastName() { return lastName; }
329     string getEmail() { return email; }
330     string getPhone() { return phone; }
331
332     // setters :

```

```

333     void setFirstName(string firstName) { this->firstName = firstName; }
334     void setLastName(string lastName) { this->lastName = lastName; }
335     void setEmail(string email) { this->email = email; }
336     void setPhone(string phone) { this->phone = phone; }
337 };
338
339 class cEmployee : public cPerson
340 {
341     string title;
342     string department;
343     float salary;
344
345 public:
346     void print(bool isBaseClass = true)
347     {
348         cPerson::print(false);
349         cout << "the title      : " << title << "\n";
350         cout << "the department : " << department << "\n";
351         cout << "the salary    : " << salary << "\n";
352         if (isBaseClass)
353             cout << " _____\n";
354     }
355
356     // setters :
357     void setTitle(string title) { this->title = title; }
358     void setDepartment(string department) { this->department = department; }
359     void setSalary(float salary) { this->salary = salary; }
360
361     // getters :
362     string getTitle() { return title; }
363     string getDepartment() { return department; }
364     float getSalary() { return salary; }
365
366     cEmployee(int id, string firstName, string lastName, string email, string phone, string
title, string department, float salary)
367         : cPerson(id, firstName, lastName, email, phone)
368     {
369
370         this->title = title;
371         this->department = department;
372         this->salary = salary;
373     }
374 };
375
376 // multi level inheritance : class1 inherited from class2 and class2 inherited from class1
...
377
378 // -- inheritance visibility modes
379
380 public : public keep public , protected keep protected
381 private : -- public && protected will be private (you can access them within the base and
the derivedClass )
382 protected : -- public && protected will be protected (you can access them within the base
and derivedClass and all nextLevelDerivedClass)
383
384 // type of inheritance :
385

```

```

386 // single : class inherit one class
387 // multi-level : class inherit a class and the class inherited by another one ..
388 // hierarchal : one class inherited by multiple classes
389
390 // -----special type -----
391 // multiple : one class inherit multiple classes (not recommended supported by cpp )
392
393 // -----special type -----
394 // hybrid : one class inherit multiple classes that also inherit a class (not recommended
    supported by cpp )
395
396
397 // up casting vs down casting
398
399 // up casting : convert from a derived class to base class (using pointers )
400
401 // down casting : convert from a base class to a derived class (you can't convert it )
402
403 // example :
404 class cPerson
405 {
406 public:
407     string name = "ayoub";
408 };
409
410 class cEmployee : public cPerson
411 {
412 public:
413     string title = "nice";
414 };
415 int main()
416 {
417
418     cEmployee e1;
419     // up casting :
420     cPerson *p1 = &e1;
421
422     cout << p1->name << endl;
423
424     cPerson p2;
425     // down casting :
426     cEmployee *e2 = &p2;
427     cout << e2->name << endl;
428
429     return 0;
430 }
431
432 In C++, the virtual keyword is used to declare a member function in a base class
433 that can be overridden by a function with the same signature in a derived class.
434 This concept is a fundamental aspect of polymorphism in object-oriented programming.
435 Here are some key points about the base usage of virtual:
436
437 Polymorphism:
438 Virtual functions enable polymorphism, allowing different objects to be treated
439 as instances of a common base class.
440 Polymorphism allows you to write code that can work with objects of

```


different derived classes through a common interface.

Function Overriding:

When a **function** is declared as **virtual** in a base **class**, it can be overridden in derived classes.

Function overriding allows derived classes to provide their own implementation of the **virtual function**.

Late Binding (Dynamic Binding):

The decision of which **function** to call is made at runtime rather than compile time. This is achieved through the use of a **virtual function** table (vtable) or similar mechanism, which maintains a mapping of **virtual functions** to their actual implementations in derived classes.

Base Class Pointers and Derived Class Objects:

Virtual functions are particularly useful when dealing with base **class** pointers pointing to objects of derived classes.

When a **virtual function** is called through a base **class** pointer, the appropriate version of the **function** in the derived **class** is invoked.

```
// example :
class cPerson
{
public:
    string name = "ayoub";

    virtual void print()
    {
        cout << "HI, i'm person \n";
    }
};

class cEmployee : public cPerson
{
public:
    void print() override
    {
        cout << "HI, i'm an employee \n";
    }
};

class cStudent : public cPerson
{
public:
    void print() override
    {
        cout << "HI, i'm a student \n";
    }
};

int main()
{
    cEmployee e1;
    cStudent s1;

    cPerson *p1 = &e1;
    cPerson *p2 = &s1;
```

```

496     p1->print();
497     p2->print();
498
499     return 0;
500 }
501
502
503 /*
504 static/Early binding
505
506     vs
507
508 dynamic/late binding
509
510 */
511
512 // static/Early binding
513 Static Binding: The binding which can be resolved at compile time by the
514 compiler is known as static or early binding.
515
516
517 // ✨ polymorphism :
518 /*
519 Polymorphism is one of the important features/principles/concepts of OOP,
520 word Ploy means "Many" and word
521 Morphism means "Form" so it means "Many Forms", the ability to take more than one form.
522 */
523 // examples :
524 /*
525 1- function overloading
526 2- function overwriting
527 3- operator overloading
528 4- virtual functions
529 */
530 In C++, an abstract class is a class that cannot be instantiated on its
531 own and is meant to serve as
532 a base class for other classes. It may contain abstract methods, which are declared
533 but not defined
534 in the abstract class. The derived classes must provide concrete implementations for these
535 abstract
536 methods. Abstract classes are used to define an interface or a common set of features that
537 derived classes must implement.
538
539 Here are the key features and concepts related to abstract classes in C++:
540
541 1. **Abstract Class Declaration:**
542     - An abstract class is declared using the `class` keyword, along with the `virtual`
543     keyword for abstract methods.
544     - It may contain both concrete (implemented) and abstract (unimplemented) methods.
545     - Abstract methods are declared with the `virtual` keyword and are followed by `= 0`
546     to indicate that they have no implementation in the abstract class.
547
548 ```cpp
549 class AbstractClass {
550 public:
551     // Concrete method
552     void concreteMethod() {

```

```

551         // Implementation
552     }
553
554     // Abstract method
555     virtual void abstractMethod() = 0;
556 };
557 ```
558
559 2. **Cannot be Instantiated:**
560 - Objects of an abstract class cannot be created directly. It is meant to be used
561   as a blueprint for other classes.
562
563 ```cpp
564 // Cannot do this - results in a compilation error
565 // AbstractClass obj;
566 ```
567
568 3. **Derived Classes Implementation:**
569 - Any class that inherits from an abstract class must provide concrete implementations
570   for all the pure virtual (abstract) methods declared in the abstract class.
571
572 ```cpp
573 class DerivedClass : public AbstractClass {
574 public:
575     // Concrete implementation for the abstract method
576     void abstractMethod() override {
577         // Implementation
578     }
579 };
580 ```
581
582 4. **Abstract Class as Interface:**
583 - Abstract classes are often used to define interfaces,
584   where the derived classes provide
585   specific implementations for the methods declared in the interface.
586
587 ```cpp
588 class Interface {
589 public:
590     virtual void method1() = 0;
591     virtual void method2() = 0;
592 };
593
594 class ConcreteClass : public Interface {
595 public:
596     void method1() override {
597         // Implementation for method1
598     }
599
600     void method2() override {
601         // Implementation for method2
602     }
603 };
604 ```
605
606 5. **Destructor in Abstract Class:**

```

607 - An abstract **class** can have a **virtual** destructor, **and** it's a good practice **to** provide
608 a **virtual** destructor **to** ensure proper cleanup **when** objects **of** derived classes are deleted.

```
609  
610 ```cpp  
611 class AbstractClass {  
612 public:  
613     virtual ~AbstractClass() {}  
614 };  
615 ```
```

616
617 Abstract classes provide a way **to** achieve abstraction **and** polymorphism **in** C++ by
618 defining a common
619 interface that derived classes must adhere **to**. They are an essential part **of**
620 **object**-oriented programming
621 **and** are widely used **in** designing **class** hierarchies.

622
623 *// INFO :*
624 An abstract **class** **in** C++ has at least one pure **virtual** **function**
625 by definition. In other words, a **function** that has no definition.

626
627 The C++ interfaces are implemented using abstract classes **and** these abstract classes
628 should not be confused **with** data abstraction which is a concept **of** keeping implementation
629 details separate from associated data.

630
631
632 *// friend class :*
633 A friend **class** can access both **private** **and** protected members **of** the **class**
634 **in** which it has been declared **as** friend.

```
635  
636 class className{  
637  
638 //  
639 friend className2;  
640  
641 };  
642 class className2{  
643  
644 };  
645
```

646 *// friend function :*
647 A friend **function** **in** C++ is a **function** that is declared outside a **class** but is capable **of**
648 accessing the **private** **and** protected members **of** the **class**. There could be situations
649 **in** programming wherein we want two classes **to** share their members. These members may be
650 data members, **class** functions **or** **function** templates. In such cases, we make the desired
651 **function**, a friend **to** both these classes which will allow accessing **private** **and**
652 protected data **of** members **of** the **class**.

```
653  
654 class className{  
655  
656 friend datatype functionName(arg);  
657  
658 };  
659 datatype functionName(arg){  
660  
661     //code  
662 }
```

```

663
664
665 // using struct with classes :
666 class className{
667
668     struct structName{
669         att1;
670         att2;
671     };
672 public :
673     structName ob1;
674
675 };
676
677 // example :
678 #include <iostream>
679 #include "./input.h"
680
681 using namespace std;
682
683 class cPerson
684 {
685
686 private:
687     struct stAddress
688     {
689         string city;
690         string street;
691     };
692
693     string fullName;
694     stAddress add;
695
696 public:
697     friend istream &operator>>(istream &inp, cPerson &person);
698
699     void printINfo()
700     {
701         cout << "\n----- Person info ----- \n";
702         cout << "the full name : " << fullName << "\n";
703         cout << "the city : " << add.city << "\n";
704         cout << "the street : " << add.street << endl;
705         cout << "\n----- ===== \n";
706     }
707 };
708
709 // output stream operator :
710 istream &operator>>(istream &inp, cPerson &person)
711 {
712
713     person.fullName = input::readString("enter the full name : ");
714     person.add.city = input::readString("enter the city : ");
715     person.add.street = input::readString("enter the street : ");
716
717     return inp;
718 }

```

```

719 int main()
720 {
721     cPerson p1;
722
723     cout << "the person info :\n";
724
725     cin >> p1;
726     p1.printINfo();
727     return 0;
728 }
729
730 // nested classes :
731     Nested or Inner Classes : A class can also contain another class definition
732     inside itself, which is called "Inner Class" in C++.
733
734 // enclosing /containing class :
735 class className{
736
737     // inner /nested class :
738     // code
739     class className{
740         // code
741     };
742     // code
743 };
744 // example :
745 #include <iostream>
746 using namespace std;
747
748 class person
749 {
750
751 protected:
752     class c
753     {
754
755         string name;
756         string lastName;
757
758     public:
759         void print()
760         {
761             cout << "the name : " << name << "\n";
762             cout << "the last name : " << lastName << "\n";
763         }
764     };
765
766     public:
767         c a;
768         string km;
769 };
770
771 class e : public person
772 {
773
774     c e2;

```

```

775 };
776
777 int main()
778 {
779
780     e p1;
781     p1.a.print();
782
783     return 0;
784 }
785
786 // separate class in library :
787 Separating Code and Classes in Libraries will make our life easier
788 and we can control our code and organize it better.
789
790 We must use "#pragma once" in each header file to prevent the compiler
791 from loading the library more than one time and have repeated code included.
792
793 /*
794 1- create new file header file with extension .h
795 2- include the included it in your main file :
796 3- add #pragma once to the header file to included one time
797 */
798
799 // example cEmployee.h
800 #pragma once
801 #include<iostream>
802 using namespace std ;
803 #include "cPerson.h"
804 class cEmployee : public cPerson
805 {
806     string title;
807     string department;
808     float salary;
809
810 public:
811     void print(bool isBaseClass = true)
812     {
813         cPerson::print(false);
814         cout << "the title      : " << title << "\n";
815         cout << "the department : " << department << "\n";
816         cout << "the salary    : " << salary << "\n";
817         if (isBaseClass)
818             cout << " _____\n";
819     }
820
821     // setters :
822     void setTitle(string title) { this->title = title; }
823     void setDepartment(string department) { this->department = department; }
824     void setSalary(float salary) { this->salary = salary; }
825
826     // getters :
827     string getTitle() { return title; }
828     string getDepartment() { return department; }
829     float getSalary() { return salary; }
830

```

```

831     cEmployee(int id, string firstName, string lastName, string email, string phone, string
title, string department, float salary)
832         : cPerson(id, firstName, lastName, email, phone)
833     {
834
835         this->title = title;
836         this->department = department;
837         this->salary = salary;
838     }
839 };
840
841
842 // ✨ objects with vectors :
843 int main(){
844
845     vector <clsA> v1;
846     short NumberOfobjects=5;
847
848     // inserting object at the end of vector
849     for (int i = 0; i < NumberOfObjects; i++)
850         v1.push_back(clsA(i));
851
852     // ✨ printing object content
853     for (int i = 0; i < NumberOfObjects; i++)
854         v1[i].Print();
855
856     return 0;
857 }
858 //Objects and Dynamic Array
859 int main(){
860
861     short NumberOfobjects = 5;
862
863     // ✨ allocating dynamic array
864     // of Size NumberOfObjects using new keyword
865
866     clsA * arrA = new clsA[NumberOfobjects];
867
868     // calling constructor
869     // for each index of array
870     for (int i = 0; i < NumberOfObjects; i++)
871         arrA[i] = clsA(i);
872
873     // printing contents of array
874     for (int i = 0; i < NumberOfObjects; i++)
875         arrA[i]. Print();
876
877     return 0;
878 }
879
880 // ✨ Objects with Parameterized Constructor and Array
881 int main(){
882
883     // Initializing 3 array Objects with function calls of
884     // parameterized constructor as elements of that array
885     clsA obj[] = { clsA(10), clsA(20), clsA(30) };

```



```
886 |
887 | // using print method for each of three elements.
888 | for (int i = 0; i < 3; i++)
889 |   obj[i]. Print();
890 |
891 | return 0;
892 | }
```