**Access Modifiers in C++**

# Hiding the Implementation

## Setting limits

In a C struct(UDDT), as with most things in C, there are no rules. Client programmers can do anything they want with that struct(UDDT), and there’s no way to force any particular behaviors. Even though you would really prefer that the client programmer not directly manipulate some of the members of your struct(UDDT), in C there’s no way to prevent it. Everything’s naked to the world.

Two reasons for controlling access to members:

1. To keep the client programmer’s hands off tools they shouldn’t touch, tools that are necessary for the internal machinations of the data type, but not part of the interface the client programmer needs to solve their particular problems.
2. To allow the library designer to change the internal workings of the structure without worrying about how it will affect the client programmer.

# C++ access control

C++ introduces three new keywords to set the boundaries in a structure: public, private, and protected, used to implement Data Hiding feature of OOP

Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members.

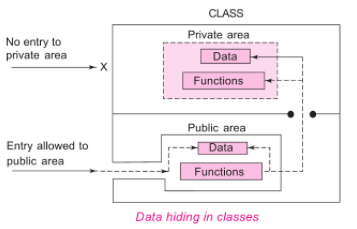
That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

Access modifiers are decided at compile time

3 types of access modifiers available in C++:

1. Public
2. Private
3. Protected

default access modifier for the members will be Private



## Public:

* All the class members declared under public will be available to everyone.
* The data members and member functions declared public can be accessed by other classes too.
* The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

#include <iostream>

using namespace std;

class Circle {

public:

double radius;

double compute\_area() {

return 3.14\*radius\*radius;

}

};

int main() {

Circle obj;

//radius = 10; // error: 'radius' was not declared in this scope

obj.radius = 5.5;

cout << "Radius is : " << obj.radius << "\n";

cout << "Area is : " << obj.compute\_area();

return 0;

}

Output:

Radius is : 5.5

Area is : 94.985

## Private:

* The class members declared as private can be accessed only by the functions inside the class.
* They are not allowed to be accessed directly by any object or function outside the class.
* Only the member functions or the friend functions are allowed to access the private data members of a class.

#include <iostream>

using namespace std;

class Circle {

private:

double radius;

public:

double compute\_area(double r) { // member function can access private

radius = r;

double area = 3.14\*radius\*radius;

cout << "Radius is : " << radius << endl;

cout << "Area is : " << area;

}

};

int main() {

Circle obj;

//radius = 10; // error: 'radius' was not declared in this scope

//obj.radius = 15; // error: 'double Circle::radius' is private

// error: within this context

obj.compute\_area(1.5);

return 0;

}

Output:

Radius is : 1.5

Area is : 7.065

## Protected:

* Similar to that of private access modifiers,
* The difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

#include <bits/stdc++.h>

using namespace std;

class Parent {

protected:

int id\_protected;

};

class Child : public Parent {

public:

void setId(int id) {

id\_protected = id;

}

void displayId() {

cout << "id\_protected is : " << id\_protected << endl;

}

};

int main() {

Child obj1;

//id\_protected = 35; // error: 'id\_protected' was not declared in this scope

//obj1.id\_protected = 23; // error: 'int Parent::id\_protected' is protected

// error: within this context

obj1.setId(81);

obj1.displayId();

return 0;

}

Output:

id\_protected is : 81

# Friends

To explicitly grant access to a function that isn’t a member of the current structure. This is accomplished by declaring that function a friendinside the structure declaration.

It’s important that the friend declaration occurs inside the structure declaration because you (and the compiler) must be able to read the structure declaration and see every rule about the size and behavior of that data type.

You can declare a global function as a friend, and you can also declare a member function of another structure, or even an entire structure, as a friend.

#include <iostream>

using namespace std;

struct X;

struct Y { void f(X\*); };

struct X { // Definition

private:

int i;

public:

void initialize();

friend void g(X\*, int); // Global friend

// friend can be used this way to simultaneously declare the function and give it friend status.

friend void Y::f(X\*); // Struct member friend

friend struct Z; // Entire struct is a friend

friend void h();

};

void X::initialize() { i = 0; }

void g(X\* x, int i) { x->i = i; }

void Y::f(X\* x) { x->i = 47; }

struct Z {

private:

int j;

public:

void initialize();

void g(X\* x);

};

void Z::initialize() { j = 99; }

void Z::g(X\* x) { x->i += j; }

void h() { X x; x.i = 100; }// Direct data manipulation

int main() {

X x;

Z z;

z.g(&x);

return 0;

}

## Nested friends

To give access to private members first, declare (without defining) the nested structure, then declare it as a friend, and finally define the structure. The structure definition must be separate from the friend declaration, otherwise it would be seen by the compiler as a non-member.

struct Holder {

private:

int a[sz];

public:

void initialize();

struct Pointer;

friend Pointer;

struct Pointer {

private:

Holder\* h;

int\* p;

public:

void initialize(Holder\* h);

// Move around in the array:

void next();

void previous();

void top();

void end();

// Access values:

int read();

void set(int i);

};

};

## Is it pure?

C++ is a hybrid object-oriented language, not a pure one, and friend was added to get around practical problems that crop up. It’s fine to point out that this makes the language less “pure,” because C++ is designed to be pragmatic, not to aspire to an abstract ideal.

# Object layout

In general, it’s not a good idea to depend on anything that’s implementation-specific when you’re writing a program. When you must have implementation-specific dependencies, encapsulate them inside a structure so that any porting changes are focused in one place.

# The class

Access control is often referred to as implementation hiding. Access control is for two important reasons:

1. You can build your internal mechanisms into the structure without worrying that client programmers will think that these mechanisms are part of the interface they should be using.
2. To separate the interface from the implementation.

# Handle classes

Access control in C++ allows you to separate interface from implementation, but the implementation hiding is only partial. The compiler must still see the declarations for all parts of an object in order to create and manipulate it properly.

Including the private implementation has two effects:

The implementation is visible even if you can’t easily access it, and it can cause needless recompilation.

## Hiding the implementation

Hiding the implementation Some projects cannot afford to have their implementation visible to the client programmer. It may show strategic information in a library header file that the company doesn’t want available to competitors.

In all these situations, it’s valuable to have the actual structure compiled inside an implementation file rather than exposed in a header file.

## Reducing recompilation

**fragile base-class problem**

Any time you make a change to a class, whether it’s to the public interface or to the private member declarations, you’ll force a recompilation of anything that includes that header file. This is often referred to as the fragile base-class problem.

The technique to solve this is sometimes called handle classes or the “Cheshire cat”.

Everything about the implementation disappears except for a single pointer, the “smile.” The pointer refers to a structure whose definition is in the implementation file along with all the member function definitions.

Thus, as long as the interface is unchanged, the header file is untouched. The implementation can change at will, and only the implementation file needs to be recompiled and relinked with the project.

# END OF FILE