

Classes and Objects

So far you have explored the structure of a simple program that starts execution at `main()` and enables you to declare local and global variables and constants and branch your execution logic into function modules that can take parameters and return values. All this is very similar to a procedural language like C, which has no object-orientation to it. In other words, you need to learn about managing data and connecting methods to it.

In this lesson, you will learn

- What classes are
- How classes help you consolidate data with methods (akin to functions) that work on them
- About constructors and destructors
- Object-oriented concepts of encapsulation and abstraction
- What *this* pointer is about

Declaring a Class

Declaration of a class involves the usage of keyword *class* followed by the name of the class, followed by a statement block `{...}` that encloses a set of member attributes and methods within curly braces, finally terminated by a semicolon.

A declaration of a class is akin to the declaration of a function. It tells the compiler about the class and its properties. Declaration of a class alone does not make a difference to the execution of a program, as the class needs to be used just the same way as a function needs to be invoked.

A class that models a human looks like the following (ignore syntactic short-comings for the moment):

```
class Human
{
// Data attributes:
string Name;
string DateOfBirth;
string PlaceOfBirth;
string Gender;
// Methods:
void Talk(string TextToTalk);
void IntroduceSelf();
...
};
```

Needless to say, *IntroduceSelf()* uses *Talk()* and some of the data attributes that are grouped within *class Human*. Thus, in keyword class, C++ has provided you with a powerful way to create your own data type that allows you to encapsulate attributes

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

and functions that work using those. All attributes of a class, in this case *Name*, *DateOfBirth*, *PlaceOfBirth*, and *Gender*, and all functions declared within it, namely *Talk()* and *IntroduceSelf()*, are called members of class *Human*.

Encapsulation, which is the ability to logically group data and methods that work using it, is a very important property of Object Oriented Programming.

Instantiating an Object of a Class

Declaring a class alone has no effect on the execution of a program. The real-world avatar of a class at runtime is an object. To use the features of a class, you typically instantiate an object of that class and use that object to access its member methods and attributes.

Instantiating an object of type class *Human* is similar to creating an instance of another type, say *double*

```
double Pi = 3.1415; // a double declared as a local variable (on stack)
```

```
Human Tom; // An object of class Human declared as a local variable
```

Alternatively, you would dynamically allocate for an instance of class *Human* as you would an *int* using *new*:

```
int* pNumber = new int; // an integer allocated dynamically on free store
```

```
delete pNumber; // de-allocating the memory
```

```
Human* pAnotherHuman = new Human(); // dynamically allocated Human
```

```
delete pAnotherHuman; // de-allocating memory allocated for a Human
```

Accessing Members Using the Dot (.) Operator.

An example of a human would be Tom, male, born in 1970 in Alabama. Instance Tom is an object of class *Human*, an avatar of the class that exists in reality that is at runtime:

```
Human Tom; // an instance of Human
```

As the class declaration demonstrates, Tom has attributes such as *DateOfBirth* that can be accessed using the dot operator (.):

```
Tom.DateOfBirth = "1970";
```

This is because attribute *DateOfBirth* belongs to class *Human* being a part of its blueprint as seen in the class declaration. This attribute exists in reality—that is, at runtime—only when an object has been instantiated. The dot operator (.) helps you access attributes of an object.

Ditto for methods such as *IntroduceSelf()*:

```
Tom.IntroduceSelf();
```

Ex1:	
class Human { // Data attributes:	

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

<pre>string Name; string DateOfBirth; string PlaceOfBirth; string Gender; // Methods: void Talk(string TextToTalk); void IntroduceSelf(); ... };</pre>	
--	--

Accessing Members Using the Pointer

If you have a pointer pTom to an instance of class Human, you can either use the pointer operator (->) to access members, as explained in the next section, or use the indirection operator (*) to reference the object following the dot operator.

```
Human* pTom = new Human();
(*pTom).Name = "Adam";
(*pTom).Age = 30;
(*pTom).IntroduceSelf();
delete pTom;
```

Ex1: Accessing Members Using the Pointer Operator (->)

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

class Human
{
    public:
        string Name;
        int Age;

        void IntroduceSelf()
        {
            cout << "I am " + Name << " and am ";
            cout << Age << " years old" << endl;
        }
};

int main()
{
    Human* pTom = new Human();
    (*pTom).Name = "Adam";
    (*pTom).Age = 30;

    (*pTom).IntroduceSelf();
    delete pTom;
    system("pause");
}
```

Output:

I am Adam and am 30 years old

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

If an object has been instantiated on the free store using new or if you have a pointer to an object, then you use the pointer operator (->) to access the member attributes and functions:

```
// Alternatively when you have a pointer:
Human Tom;
Human* pTom = &Tom; // Assign address using reference operator&
pTom->Name = "Adam"; // is equivalent to Tom.DateOfBirth = "1970";
pTom->Age = 30;
pTom->IntroduceSelf(); // is equivalent to Tom.IntroduceSelf();
```

Ex2: Accessing Members Using the Pointer Operator (->)

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

class Human
{
    public:
        string Name;
        int Age;

        void IntroduceSelf()
        {
            cout << "I am " + Name << " and am ";
            cout << Age << " years old" << endl;
        }
};

int main()
{
    // Alternatively when you have a pointer:
    Human Tom;
    Human* pTom = &Tom; // Assign address using reference operator&
    pTom->Name = "Adam"; // is equivalent to Tom.DateOfBirth = "1970";
    pTom->Age = 30;
    pTom->IntroduceSelf(); // is equivalent to Tom.IntroduceSelf();

    system("pause");
}
```

Output:

I am Adam and am 30 years old

Keywords *public* and *private*

Each of us has a lot of information. Some of this information is available to people around us—for example, our names. This information can be called public. However, certain attributes are those that you might not want the world to see or know—for example, your income. This information is private and often kept a secret.

C++ enables you to model class attributes and methods as *public*—implying one with an object of the class can invoke them—or *private*—implying that only artifacts that belong to the class (or its “friends”) can invoke those private members. C++ keywords *public* and *private* help you as the designer of a class decide what parts of a class can be invoked from outside it, for instance, from main() , and what cannot.

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

A compile-worthy form of class Human featuring keywords such as private and public is demonstrated by the following example:

Ex3: A Compile-worthy Class Human

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6: private:
7:     string Name;
8:     int Age;
9:
10: public:
11:     void SetName (string HumansName)
12:     {
13:         Name = HumansName;
14:     }
15:
16:     void SetAge(int HumansAge)
17:     {
18:         Age = HumansAge;
19:     }
20:
21:     void IntroduceSelf()
22:     {
23:         cout << "I am " + Name << " and am ";
24:         cout << Age << " years old" << endl;
25:     }
26: };
27:
28: int main()
29: {
30:     // Constructing an object of class Human with attribute Name as "Adam"
31:     Human FirstMan;
32:     FirstMan.SetName("Adam");
33:     FirstMan.SetAge(30);
34:
35:     // Constructing an object of class Human with attribute Name as "Eve"
36:     Human FirstWoman;
37:     FirstWoman.SetName("Eve");
38:     FirstWoman.SetAge (28);
39:
40:     FirstMan.IntroduceSelf();
41:     FirstWoman.IntroduceSelf();
42: }
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

Output:

```
I am Adam and am 30 years old
I am Eve and am 28 years old
```

What advantages does this ability to mark attributes or methods as *private* present you as the programmer? Consider the declaration of class *Human* ignoring all but the member attribute *Age*:

```
class Human
{
private:
    // Private member data:
    int Age;
    string Name;
public:
    int GetAge()
    {
        return Age;
    }
    void SetAge(int InputAge)
    {
        Age = InputAge;
    }
    // ...Other members and declarations
};
```

When the user of this instance tries to access *Human's* age with
`cout << FirstWoman.Age; // compile error`

then this user would get a compile error akin to "Error: *Human::Age* —cannot access private member declared in class *Human* ." The only permissible way to know the *Age* would be to ask for it via public method *GetAge()* supplied by class *Human* and implemented in a way the programmer of the class thought was an appropriate way to expose the *Age*:

```
cout << FirstWoman.GetAge(); // OK
```

If the programmer of class *Human* so desires, he could use *GetAge()* to show *FirstWoman* as younger than she is! In other words, this means C++ allows the class to control what attributes it wants to expose and how it wants to expose the same. If there were no *GetAge()* public member method implemented by class *Human* , the class would effectively have ensured that the user cannot query *Age* at all. This feature can be useful in situations that are explained later in this lesson.

Similarly, *Human::Age* cannot be assigned directly either:

```
FirstWoman.Age = 22; // compile error
```

The only permissible way to set the age is via method *SetAge()*:

```
Eve.SetAge(22); // OK
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

This has many advantages. The current implementation of `SetAge()` does nothing but directly set the member variable `Human::Age`. However, you can use `SetAge()` to verify the Age being set is non-zero and not negative and thus validate external input:

```
class Human
{
private:
    int Age;
public:
    void SetAge(int InputAge)
    {
        if (InputAge > 0)
            Age = InputAge;
    }
};
```

Thus, C++ enables the designer of the class to control how data attributes of the class are accessed and manipulated.

Abstraction of Data via Keyword *private*

While allowing you to design a class as a container that encapsulates data and methods that operate on that data, C++ empowers you to decide what information is unreachable to the outside world (that is, unavailable outside the class) via keyword *private*. At the same time, you have the possibility to allow controlled access to even information declared private via methods that you have declared as *public*. Thus your implementation of a class can abstract what you think the world outside it—other classes and functions such as `main()`—don't need to know.

Going back to the example related to `Human::Age` being a private member, you know that even in reality many people don't like to reveal their true ages. If class `Human` was required to tell an age two years younger than the current age, it could do so easily via a public function `GetAge()` that uses the `Human::Age` parameter, reduces it by two, and supplies the result as demonstrated by the following.

Ex4: A Model of Class Human Where the True Age Is Abstracted from the User and a Younger Age Is Reported
--

<pre>0: #include <iostream> 1: using namespace std; 2: 3: class Human 4: { 5: private: 6: // Private member data: 7: int Age; 8: 9: public: 10: void SetAge(int InputAge) 11: { 12: Age = InputAge;</pre>
--

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
13:  }
14:
15:  // Human lies about his / her Age (if over 30)
16:  int GetAge()
17:  {
18:      if (Age > 30)
19:          return (Age - 2);
20:      else
21:          return Age;
22:  }
23: };
24: int main()
25: {
26:     Human FirstMan;
27:     FirstMan.SetAge(35);
28:     Human FirstWoman;
29:     FirstWoman.SetAge(22);
30:     cout << "Age of FirstMan " << FirstMan.GetAge() << endl;
31:     cout << "Age of FirstWoman " << FirstWoman.GetAge() << endl;
32:     return 0;
33: }
```

Output:

```
Age of FirstMan 33
Age of FirstWoman 22
```

Constructors

A constructor is a special function (or method) that is invoked when an object is created. Just like functions, constructors can also be overloaded.

Declaring and Implementing a Constructor

A constructor is a special function that has the name of the class and no return value. So, class Human has a constructor that is declared like this:

```
class Human
{
public:
    Human(); // declaration of a constructor
};
```

This constructor can either be implemented inline in the class or can be implemented externally outside the class declaration. An implementation or definition inside the class looks like this:

```
class Human
{
public:
    Human()
    {
        // constructor code here
    }
};
```


OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
}  
};
```

A variant enabling you to define the constructor outside the class' declaration looks like this:

```
class Human  
{  
public:  
    Human(); // constructor declaration  
};  
// constructor definition (implementation)  
Human::Human()  
{  
    // constructor code here  
}
```

NOTE. :: is called the scope resolution operator. For example, Human::Age is referring to variable Age declared within the scope of class Human. ::Age, on the other hand, refers to another variable Age in a global scope.

When and How to Use Constructors

A constructor is always invoked when an object is created. This makes a constructor a perfect place for you to initialize class member variables such as integers, pointers, and so on to known initial values. Take a look at example 4 again. Note that if you had forgotten to SetAge(), the integer variable Human::Age would contain an unknown junk value as that variable has not been initialized (try it by commenting out Lines 28 and 30).

Example below uses constructors to implement a better version of class Human, where variable Age has been initialized.

Ex5: Using Constructors to Initialize Class Member Variables

```
0: #include <iostream>  
1: #include <string>  
2: using namespace std;  
3:  
4: class Human  
5: {  
6: private:  
7:     // Private member data:  
8:     string Name;  
9:     int Age;  
10:  
11: public:  
12:     // constructor  
13:     Human()  
14:     {  
15:         Age = 0; // initialized to ensure no junk value
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
16:     cout << "Constructed an instance of class Human" << endl;
17: }
18:
19: void SetName (string HumansName)
20: {
21:     Name = HumansName;
22: }
23:
24: void SetAge(int HumansAge)
25: {
26:     Age = HumansAge;
27: }
28:
29: void IntroduceSelf()
30: {
31:     cout << "I am " + Name << " and am ";
32:     cout << Age << " years old" << endl;
33: }
34: };
35:
36: int main()
37: {
38:     Human FirstMan;
39:     FirstMan.SetName("Adam");
40:     FirstMan.SetAge(30);
41:
42:     Human FirstWoman;
43:     FirstWoman.SetName("Eve");
44:     FirstWoman.SetAge (28);
45:
46:     FirstMan.IntroduceSelf();
47:     FirstWoman.IntroduceSelf();
48: }
```

Output:

Constructed an instance of class Human
Constructed an instance of class Human
I am Adam and am 30 years old
I am Eve and am 28 years old

NOTE: A constructor that can be invoked without argument is called the default constructor. Programming a default constructor is optional. If you don't program any constructor as seen in the example 3, the compiler creates one for you (that constructs member attributes but does not initialize Plain Old Data types such as *int* to any value).

Overloading Constructors

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

As constructors can be overloaded just like functions, we can create a constructor that requires Human to be created with a name as a parameter, for instance:

```
class Human
{
public:
    Human()
    {
        // default constructor code here
    }
    Human(string HumansName)
    {
        // overloaded constructor code here
    }
};
```

The application of overloaded constructors is demonstrated by the example 6 in creating an object of class Human with a name supplied at the time of construction.

Ex6: A Class Human with Multiple Constructors

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6: private:
7:     // Private member data:
8:     string Name;
9:     int Age;
10:
11: public:
12:     // constructor
13:     Human()
14:     {
15:         Age = 0; // initialized to ensure no junk value
16:         cout << "Default constructor creates an instance of Human" << endl;
17:     }
18:
19:     // overloaded constructor that takes Name
20:     Human(string HumansName)
21:     {
22:         Name = HumansName;
23:         Age = 0; // initialized to ensure no junk value
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
24:     cout << "Overloaded constructor creates " << Name << endl;
25: }
26:
27: // overloaded constructor that takes Name and Age
28: Human(string HumansName, int HumansAge)
29: {
30:     Name = HumansName;
31:     Age = HumansAge;
32:     cout << "Overloaded constructor creates ";
33:     cout << Name << " of " << Age << " years" << endl;
34: }
35:
36: void SetName (string HumansName)
37: {
38:     Name = HumansName;
39: }
40:
41: void SetAge(int HumansAge)
42: {
43:     Age = HumansAge;
44: }
45:
46: void IntroduceSelf()
47: {
48:     cout << "I am " + Name << " and am ";
49:     cout << Age << " years old" << endl;
50: }
51: };
52:
53: int main()
54: {
55:     Human FirstMan; // use default constructor
56:     FirstMan.SetName("Adam");
57:     FirstMan.SetAge(30);
58:
59:     Human FirstWoman ("Eve"); // use overloaded constructor
60:     FirstWoman.SetAge (28);
61:
62:     Human FirstChild ("Rose", 1);
63:
64:     FirstMan.IntroduceSelf();
65:     FirstWoman.IntroduceSelf();
66:     FirstChild.IntroduceSelf();
67: }
```

Output:

Default constructor creates an instance of Human

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
Overloaded constructor creates Eve
Overloaded constructor creates Rose of 1 years
I am Adam and am 30 years old
I am Eve and am 28 years old
I am Rose and am 1 years old
```

Class without a Default Constructor

In the example 7, see how class Human without the default constructor enforces the creator to supply a Name and Age as a prerequisite to creating an object.

Ex7: A Class with Overloaded Constructor(s) and No Default Constructor

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6: private:
7:     // Private member data:
8:     string Name;
9:     int Age;
10:
11: public:
12:
13:     // overloaded constructor (no default constructor)
14:     Human(string HumansName, int HumansAge)
15:     {
16:         Name = HumansName;
17:         Age = HumansAge;
18:         cout << "Overloaded constructor creates " << Name;
19:         cout << " of age " << Age << endl;
20:     }
21:
22:     void IntroduceSelf()
23:     {
24:         cout << "I am " + Name << " and am ";
25:         cout << Age << " years old" << endl;
26:     }
27: };
28:
29: int main()
30: {
31:     // Uncomment next line to try creating using a no default constructor
32:     // Human FirstMan;
33:
34:     Human FirstMan("Adam", 30);
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
35: Human FirstWoman("Eve", 28);  
36:  
37: FirstMan.IntroduceSelf();  
38: FirstWoman.IntroduceSelf();  
39: }
```

Output:
Overloaded constructor creates Adam of age 30
Overloaded constructor creates Eve of age 28
I am Adam and am 30 years old
I am Eve and am 28 years old

Destructor

Destructors, like constructors, are special functions, too. Unlike constructors, destructors are automatically invoked when an object is destroyed.

Declaring and Implementing a Destructor

The destructor, like the constructor, looks like a function that takes the name of the class, yet has a tilde (~) preceding it. So, class Human can have a destructor that is declared like this:

```
class Human  
{  
    ~Human(); // declaration of a destructor  
};
```

This destructor can either be implemented inline in the class or externally outside the class declaration. An implementation or definition inside the class looks like this:

```
class Human  
{  
public:  
    ~Human()  
    {  
        // destructor code here  
    }  
};
```

A variant enabling you to define the destructor outside the class' declaration looks like this:

```
class Human  
{  
public:  
    ~Human(); // destructor declaration
```

```
};
// destructor definition (implementation)
Human::~Human()
{
    // destructor code here
}
```

As you can see, the declaration of the destructor differs from that of the constructor slightly in that this contains a tilde (~). The role of the destructor is, however, diametrically opposite to that of the constructor.

When and How to Use Destructors

Destructors are always invoked when an object of a class goes out of scope or is deleted via delete and is destroyed. This property makes destructors the ideal place to reset variables and release dynamically allocated memory and other resources.

This lesson has consistently recommended the usage of std::string over a C-style char buffer where you have to manage memory allocation and the likes yourself. std::string and such other utilities are nothing but classes themselves that make the best of constructors and destructors. Analyze a sample class MyString as shown in the example 8 that allocates memory for a string in the constructor and releases it in the destructor.

EX8: A Simple Class That Encapsulates a C-style Buffer to Ensure Deallocation via the Destructor

```
0: #include <iostream>
1: using namespace std;
2:
3: class MyString
4: {
5: private:
6:     char* Buffer;
7:
8: public:
9:     // Constructor
10:    MyString(const char* InitialInput)
11:    {
12:        if(InitialInput != NULL)
13:        {
14:            Buffer = new char [strlen(InitialInput) + 1];
15:            strcpy(Buffer, InitialInput);
16:        }
17:        else
18:            Buffer = NULL;
19:    }
20:    // Destructor: clears the buffer allocated in constructor
21:    ~MyString()
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
22:  {
23:      cout << "Invoking destructor, clearing up" << endl;
24:      if (Buffer != NULL)
25:          delete [] Buffer;
26:  }
27:
28:  int GetLength()
29:  {
30:      return strlen(Buffer);
31:  }
32:
33:  const char* GetString()
34:  {
35:      return Buffer;
36:  }
37: }; // end of class MyString
38:
39: int main()
40: {
41:     MyString SayHello("Hello from String Class");
42:     cout << "String buffer in MyString is " << SayHello.GetLength();
43:     cout << " characters long" << endl;
44:
45:     cout << "Buffer contains: ";
46:     cout << "Buffer contains: " << SayHello.GetString() << endl;
47: }
```

Output:

String buffer in MyString is 23 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up

Note: Destructors cannot be overloaded. A class can have only one destructor. If you forget to implement a destructor, the compiler creates and invokes a dummy destructor, that is, an empty one (that does no cleanup of dynamically allocated memory).

this Pointer

An important concept in C++, this is a reserved keyword applicable within the scope of a class that contains the address of the object. In other words, the value of *this* is &object. Within a class member method, when you invoke another member method, the compiler sends this pointer as an implicit, invisible parameter in the function call:

```
class Human
{
private:
```


OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
// ... private member declarations
void Talk (string Statement)
{
    cout << Statement;
}
public:
    void IntroduceSelf()
    {
        Talk("Bla bla");
    }
};
```

What you see here is the method *IntroduceSelf()* using private member *Talk()* to print a statement on the screen. In reality, the compiler embeds the *this* pointer in calling *Talk*, that is invoked as *Talk(this, "Bla bla")*.

From a programming perspective, *this* does not have too many applications, except those where it is usually optional. For instance, the code to access *Age* within *SetAge()*, as shown in the example 3, can have a variant:

```
void SetAge(int HumansAge)
{
    this->Age = HumansAge; // same as Age = HumansAge
}
```

Declaring a *friend* of a *class*

A class does not permit external access to its data members and methods that are declared *private*. This rule is waived for those classes and functions that are disclosed as *friend* classes or functions, using keyword *friend* as seen in the example 9.

Ex9: Using the friend Keyword to Allow an External Function DisplayAge() Access to Private Data Members
--

<pre>0: #include <iostream> 1: #include <string> 2: using namespace std; 3: 4: class Human 5: { 6: private: 7: string Name; 8: int Age; 9: 10: friend void DisplayAge(const Human& Person); 11: 12: public: 13: Human(string InputName, int InputAge)</pre>

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
14: {  
15:     Name = InputName;  
16:     Age = InputAge;  
17: }  
18: };  
19: void DisplayAge(const Human& Person)  
20: {  
21:     cout << Person.Age << endl;  
22: }  
23: int main()  
24: {  
25:     Human FirstMan("Adam", 25);  
26:     cout << "Accessing private member Age via friend: ";  
27:     DisplayAge(FirstMan);  
28:     return 0;  
29: }
```

Output:

Accessing private member Age via friend: 25