Siddhartha Rao

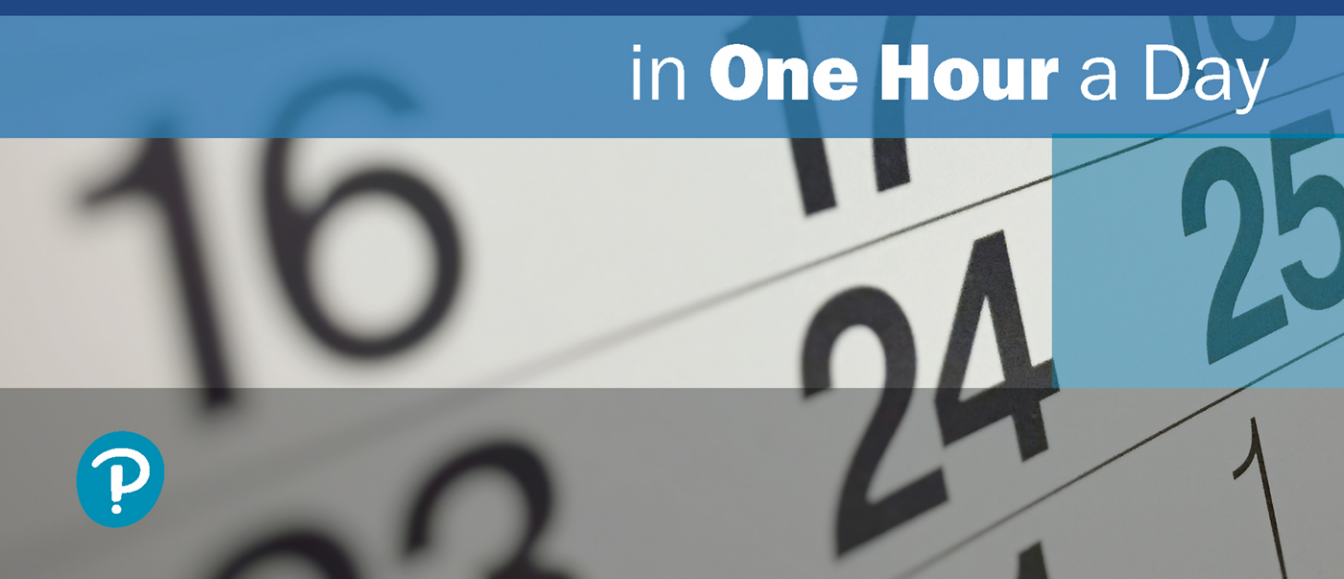
NINTH EDITION

New C++20 C++23 Coverage

Sams **Teach Yourself**

# C++

in **One Hour** a Day



# Sams **Teach Yourself C++** in **One Hour** a Day

PART I: **The Basics**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| Getting Started | The Anatomy of a C++ Program | Using Variables, Declaring Constants | Managing Arrays and Strings | Working with Expressions, Statements, and Operators | Controlling Program Flow | Organizing Code with Functions |

PART II: **Fundamentals of Object-Oriented C++ Programming**

| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|
| Pointers and Explained References | Classes and Objects | Implementing Inheritance | Polymorphism | Operator Types and Operator Overloading | Casting Operators | An Introduction to Macros and Templates |

PART III: **Learning the Standard Template Library (STL)**

PART IV:

| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|
| An Introduction to the The Standard Template Library | The String Class | STL Dynamic Array Classes | STL `list` and `forward_list` | STL `set` and `multiset` | STL `map` and `multimap` | Understanding Function Objects |

**Lambda Expressions and STL Algorithms**

PART V: **Advanced C++ Concepts**

| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
|---|---|---|---|---|---|---|
| Lambda Expressions | STL Algorithms | Adaptive Containers: Stack and Queue | Working with Bit Flags Using the STL | Understanding Smart Pointers | Using Streams for Input and Output | Exception Handling |

| 29 | 30 | 31 | | | | |
|---|---|---|---|---|---|---|
| C++20 Concepts, Ranges, Views, and Adaptors | C++ Threads | C++20 Modules and C++23 | | | | |

```
Human* firstWoman = new Human();
firstWoman->dateOfBirth = "1993";
firstWoman->IntroduceSelf();
delete firstWoman;
```

Listing 9.1 shows a compile-worthy form of the class `Human` featuring a new keyword, `public`.

**Input ▼**

**LISTING 9.1** A Compile-Worthy Class `Human`

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: public:
 7:     string name;
 8:     int age;
 9:
10:     void IntroduceSelf()
11:     {
12:        cout << "I am " + name << " and am ";
13:        cout << age << " years old" << endl;
14:     }
15: };
16:
17: int main()
18: {
19:     // An object of class Human with attribute name as "Adam"
20:     Human firstMan;
21:     firstMan.name = "Adam";
22:     firstMan.age = 30;
23:
24:     // An object of class Human with attribute name as "Eve"
25:     Human firstWoman;
26:     firstWoman.name = "Eve";
27:     firstWoman.age = 28;
28:
29:     firstMan.IntroduceSelf();
30:     firstWoman.IntroduceSelf();
31: }
```

### Output ▼

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

### Analysis ▼

Lines 4 through 15 demonstrate the basic C++ class `Human`. Note the structure of the class `Human` and how this class has been used in `main()`.

This class contains two member variables: one of type `string` called `name` at Line 7 and another of type `int` called `age` at Line 8. It also contains the function (also called a method) `IntroduceSelf()` in Lines 10 through 14. Lines 20 and 25 in `main()` instantiate two objects of the class `Human`, named `firstMan` and `firstWoman`, respectively. The lines following this instantiation of objects set the member variables of the objects `firstMan` and `firstWoman` by using the dot operator. Note that Lines 29 and 30 invoke the same function `IntroduceSelf()` on the two objects to create two distinct lines in the output. In a way, this program demonstrates how the objects `firstMan` and `firstWoman` are unique and individually distinct real-world representatives of an abstract type defined by the class `Human`.

Did you notice the keyword `public` in Line 6? It's time you learned features that help you protect attributes your class should keep hidden from those using it.

## The Keywords `public` and `private`

Information can be classified into at least two categories: data that you don't mind the *public* knowing and data that is *private*. Gender is an example of information that most people may not mind sharing. However, income may be a private matter.

C++ enables you to model class attributes and methods as public or private. Public class members can be used by anyone in possession of an object of the class. Private class members can be used only within the class (or its "friends"). The C++ keywords `public` and `private` help you as the designer of a class decide what parts of the class can be invoked from outside it—for instance, from `main()`—and which cannot.

What advantages does this ability to mark attributes or methods as `private` present you as the programmer? Consider the declaration of the class `Human` and the member attribute `age` in particular:

```
class Human
{
```

```
private:
   // Private member data:
   int age;
   string name;

public:
   int GetAge()
   {
      return age;
   }

   void SetAge(int humansAge)
   {
      age = humansAge;
   }

// ...Other members and declarations
};
```

Assume an instance of a `Human` called `eve`:

```
Human eve;
```

When the user of this instance tries to access member `age`:

```
cout << eve.age; // compile error
```

this user gets 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 the public method `GetAge()` supplied by the class `Human` and implemented in a way the programmer of the class thought was an appropriate way to share `age`:

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

`GetAge()` gives the programmer of the class `Human` the opportunity to know when `age` is being queried and share it in a way that suits. In other words, C++ allows the class to control what attributes it wants to expose and how it wants to expose them. If there were no `GetAge()` public member method implemented by the class `Human`, the class would effectively ensure that the user cannot query `age` at all. This feature can be useful in situations that are explained later in this lesson.

Note that `Human::age` cannot be assigned directly either:

```
eve.age = 22; // compile error
```

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

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

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 that the `age` being set is nonzero and not negative and thus validate external input:

```
class Human
{
private:
   int age;

public:
   void SetAge(int humansAge)
   {
      if (humansAge > 0)
          age = humansAge;
   }
};
```

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

## Abstraction of Data via the Keyword `private`

C++ empowers you to decide what information remains unreachable to the outside world (that is, unavailable outside the class) via the keyword `private`. At the same time, it enables you to control access to even information declared `private` via methods that you have declared as `public`. Your implementation of a class can therefore abstract member information that classes and functions outside this class don't need to have access to.

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 age. If the class `Human` were 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 Listing 9.2.

### Input ▼

**LISTING 9.2** A Model of the Class `Human` Where the True `age` Is Abstracted from the User, and a Younger `age` Is Reported

```
 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;
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:
25: int main()
26: {
27:     Human firstMan;
28:     firstMan.SetAge(35);
29:
30:     Human firstWoman;
31:     firstWoman.SetAge(22);
32:
33:     cout << "Age of firstMan " << firstMan.GetAge() << endl;
34:     cout << "Age of firstWoman " << firstWoman.GetAge() << endl;
35:
36:     return 0;
37: }
```



## Output ▼

```
Age of firstMan 33
Age of firstWoman 22
```

## Analysis ▼

Note the public method `Human::GetAge()` in Line 16. As the actual age contained in the private integer `Human::age` is not directly accessible, the only resort external users of this class have in querying an object of the class `Human` for attribute `age` is via the method `GetAge()`. Thus, the actual age held in `Human::age` is abstracted from the

outside world. Indeed, our `Human` lies about its age, and `GetAge()` returns a reduced value for all humans who are older than 30, as shown in Lines 18 through 21.

Abstraction is an important concept in object-oriented languages. It empowers programmers to decide what attributes of a class need to remain known only to the class and its members, with nobody outside it (with the exception of those declared as its "friends") having access to it.

# Constructors

A *constructor* is a special function (or method) invoked during the instantiation of a class to construct an object. Just like functions, constructors can also be overloaded.

## Declaring and Implementing a Constructor

A constructor is a special function that takes the name of a class and returns no value. So, the class `Human` would have a constructor that is declared like this:

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

This constructor can be implemented either inline within the class or externally outside the class declaration. An implementation (also called definition) inside the class looks like this:

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

A variant that enables you to define the constructor outside the class's declaration looks like this:

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