Member Functions and Privacy

- Design member functions using logic constructs
- Control accessibility to the data members of a class
- Introduce the concept of an object's empty state

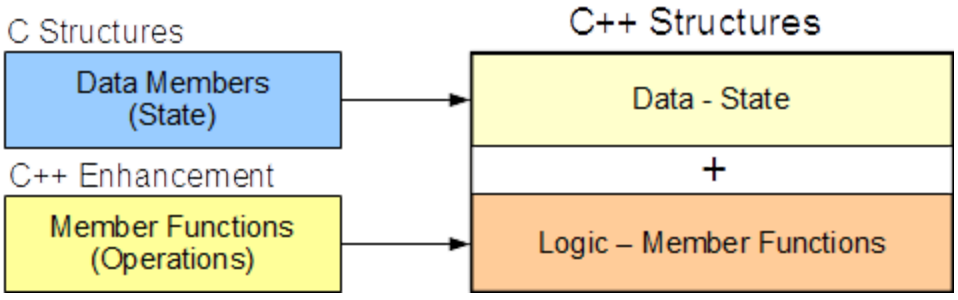
> "Because different structures can have member functions with the same name, we must specify the structure name when defining a member function" **Stroustrup, 1997.**

The primary concept of object-oriented programming is class encapsulation. Encapsulation incorporates within a class the structure of data that its objects store and the logic that operates on that data. In other words, encapsulation creates a clean interface between the class and its clients while hiding the implementation details from its clients. The C++ language describes this logic in the form of functions that are members of the class. The data members of a class hold the information about the state of its objects, while the member functions define the operations that query, modify and manage that state.

This chapter describes the C++ syntax for declaring member functions in a class definition, for defining the member functions in the implementation file and for limiting accessibility to an object's data.

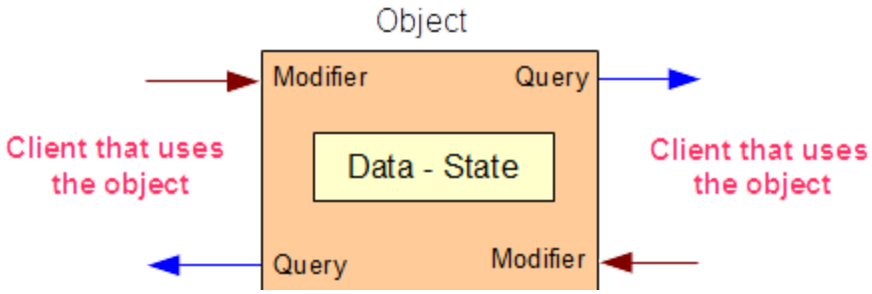
Member Functions

The member functions of a class provide the communication links between client code and objects of the class. Client code calls the member functions to access an object's data and possibly to change that data.



We classify member functions into three mutually exclusive categories:

- **Queries:** Also called accessor methods - report the state of the object
- **Modifiers:** Also called mutator methods - change the state of the object
- **Special:** Also called manager methods - create, assign and destroy an object



Every member function has direct access to the members of its class. Each member function receives information from the client code through its parameters and passes information to the client code through its return value and possibly its parameters.

Adding a Member Function

Consider a `Student` type with the following definition

```
const int NG = 20;

struct Student {
    int no;          // student number
    float grade[NG]; // grades
    int ng;          // number of grades filled
};
```

Function Declaration

To declare a member function to a class, we insert its prototype into the class definition.

For example, to add `display()` as a member to our `Student` type, we write:

```
struct Student {
    int no;
    float grade[NG];
    int ng;

    void display() const; // member function
};
```

The `const` qualifier identifies the member function as a query. A query does not change the state of its object. That is, this query cannot change the value of `no` or any `grade`.

As a member function, `display()` has direct access to the data members (`no` and `grade`). There is no need to pass their values as arguments to the function.

Function Definition

We define `display()` in the implementation file as follows:

```
void Student::display() const {

    cout << no << ": \n";
    for (int i = 0; i < ng; i++)
        cout << grade[i] << endl;

}
```

The definition consists of four elements:

- **The `Student::` prefix** on the function name identifies it as a member of our `Student` type
- **The empty parameter list:** This function does not receive any values from the client code or return any values through the parameter list to the client code
- **The `const` qualifier** identifies this function as a query - this function cannot change any of the values of the object's data members
- **The data members:** The function accesses no and grade are defined outside the function's scope but within the class' scope, which encompasses the function's scope

Calling a Member Function

Client code calls a member function in the same way that an instance of a `struct` refers to one of its data members. The call consists of the object's identifier, followed by the `.` operator and then followed by the member function's identifier.

For example, if `harry` is a `Student` object, we display its data by calling `display()` on `harry`:

```
Student harry = {975, 78.9f, 69.4f};

harry.display();  // <== client call to the member function

cout << endl;
```

The object part of the function call (the part before the member selection operator) identifies the data that the function accesses.

Scope of a Member Function

The scope of a member function lies within the scope of its class. That is, a member function can access any other member within its class' scope. For example, a member function can access another member function directly:

```
struct Student {
    int no;
    float grade[NG];
    int ng;

    void display() const;
    void displayNo() const;
};

void Student::displayNo() const {
    cout << no << ": \n";
}

void Student::display() const {
    displayNo(); // calls the member function defined above
    for (int i = 0; i < ng; i++)
        cout << grade[i] << endl;
}
```

Accessing Global Functions

A member function can also access a function outside its class' scope. Consider the following global function definition:

```
void displayNo() {
    cout << "Number...\n";
}
```

Note that this definition does not include any scope resolution identifier. This global function shares the same identifier with one of the member functions, but does not introduce any conflict, since the client code calls each function using different syntax.

```
displayNo();       // calls the global display function
harry.displayNo(); // calls the member function on harry
```

To access the global function from within the member function we apply the scope resolution operator:

```
void Student::display() const {
    ::displayNo(); // calls the global function
    displayNo();   // calls the member function
    for (int i = 0; i < ng; i++)
        cout << grade[i] << endl;
}
```

Privacy

Data privacy is central to encapsulation. Data members defined using the `struct` keyword are exposed to client code. Any client code can change the value of a data member. To limit accessibility to any member, the C++ language lets us hide that member within the class by identifying it as private.

Well-designed object-oriented solutions expose to client code only those members that are the class's communication links. In a good design, the client code should not require direct access to any data that describes an object's state or any member function that performs internally directed operations.

Accessibility Labels

To prohibit external access to any member (data or function), we insert the label `private` into the definition of our class:

```
private:
```

`private` identifies all subsequent members listed in the class definition as inaccessible.

To enable external access, we insert the label `public`:

```
public:
```

`public` identifies all subsequent members listed in the class definition as accessible.

For example, in order to:

- Hide the data members of each `Student` object
- Expose the member function(s) of the `Student` type

We insert the accessibility keywords as follows

```
struct Student {
private:
    int no;
    float grade[NG];
    int ng;
public:
    void display() const;
};
```

Note that the keyword struct identifies a class that is public by default.

`class` Keyword

The keyword `class` identifies a class that is private by default.

We use the keyword `class` to simplify the definition of a `Student` type:

```
class Student {
    int no;
    float grade[NG];
    int ng;
public:
    void display() const;
};
```

The `class` keyword is the common keyword in object-oriented programming, much more common than the `struct` keyword. (The C language does not support privacy and a derived type in C can only be a `struct`).

Any attempt by the client code to access a private member generates a complier error:

```
void foo(const Student& harry) {
    cout << harry.no;  // ERROR - this member is private!
}
```

The function `foo()` can only access the data stored in `harry` indirectly through public member function `display()`.

```
void foo(const Student& harry) {
    harry.display(); // OK
}
```

Modifying Private Data

If the data members of a class are private, client code cannot initialize their values directly. We use a separate member function for this specific task.

For example, to store data in `Student` objects, let us introduce a `public` modifier named `set()`:

```
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
public:
    void set(int, const float*, int);
    void display() const;
};
```

`set()` receives a student number, the address of an unmodifiable array of grades and the number of grades in that array from the client code and stores this information in the data members of the `Student` object:

```
void Student::set(int sn, const float* g, int ng_) {
    ng = ng_ < NG ? ng_ : NG;
    no = sn; // store the Student number as received
    // store the grades as received within the available space
    for (int i = 0; i < ng; i++)
        grade[i] = g[i];
}
```

Communications Links

The `set()` and `display()` member functions are the only communication links to client code. Clients can call `set()` or `display()` on any `Student` object, but no client code can access the data stored within any `Student` object directly.

For example, the compiler traps the following privacy breach:

```
Student harry;
float g[] = {78.9f, 69.4f};

harry.set(975, g, 2);
harry.display();

cout << harry.no; // ERROR .no IS PRIVATE!
```

Empty State

Hiding all data members from client code gives us control over which data to accept, which data to reject and which data to expose. We can validate information incoming from client code before storing it in an object. If the data is invalid, we can reject it and store default values that identify the object's state as an empty state.

Upgrading `set()`

Let us upgrade our `set()` member function to validate incoming data only if:

- The student number is positive-valued
- The grades are between 0 and 100 inclusive
- If any incoming data fails to meet one of these conditions, let us ignore all incoming data and store a value that places the object in an *empty state*. For instance, let us use a student number of 0 to identify an empty state:

```
void Student::set(int sn, const float* g, int ng_) {
    int n = ng_ < NG ? ng_ : NG;
    bool valid = true; // assume valid input, check for invalid values

    if (sn < 1)
        valid = false;
    else
        for (int i = 0; i < n && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;
```

```
    if (valid) {
        // accept the client's data
        no = sn;
        ng = n;
        for (int i = 0; i < n; i++)
            grade[i] = g[i];
    }
    else {
        no = 0; // ignore the client's data, set an empty state
    }
}
```

This validation logic ensures that either the data stored in any `Student` object is valid data or the object is in an empty state.

Design Tip

Select one data member to hold the special value that identifies an empty state. Then, to determine if an object is in an empty state, we only need to interrogate that data member.

Upgrading `display()`

To match this upgrade, we ensure that our `display()` member function executes gracefully if our object is in an empty state:

```
void Student::display() const {
    if (no != 0) {
        cout << no << ":\n";
        for (int i = 0; i < ng; i++)
            cout << grade[i] << endl;
    } else {
        cout << "no data available";
    }
}
```

Looking Forward

Although this upgrade validates data incoming from client code, our class definition still leaves the data in a `Student` object uninitialized before the first call from client code to `set()`. To address this deficiency, we will introduce a special member function in the chapter entitled Construction and Destruction.

Input and Output Examples

The `iostream` type that represents the standard input and output objects, like `cin` and `cout`, provides member functions for controlling the conversion of characters from the input stream into data types stored in system memory and the conversion of data types stored in system memory into characters sent to the output stream.

`cin`

The `cin` object is an instance of the `istream` type. This object extracts a sequence of characters from the standard input stream, converts that sequence into a specified type and stores that type in system memory.

The general expression for extracting characters from the standard input stream takes the form

```
cin >> identifier
```

where `>>` is the extraction operator and `identifier` is the name of the destination variable.

For example,

```
int i;
char c;
double x;
char s[8];
cout << "Enter an integer,\n"
        "a character,\n"
        "a floating-point number and\n"
        "a string : " << flush;
cin >> i;
cin >> c;
cin >> x;
cin >> s;  // possible overflow
cout << "Entered " << i << ' '
    << c << ' ' << x << ' ' << s << endl;
```

Outputs:

```
Enter an integer,
a character,
a floating-point and
a string : 6  -  9.75  Harry
```

```
Entered 6 - 9.75 Harry
```

The `cin` object skips leading whitespace with numeric, string and character types (in the same way that `scanf("%d"...)`, `scanf("%lf"...)`, `scanf("%s"...)` and `scanf(" %c"...)` skip whitespace in C).

```
// Leading Whitespace
// leading.cpp

#include <iostream>
using namespace std;

int main() {
    char str[11];

    cout << "Enter a string : " << endl;
    cin >> str;
    cout << "|" << str << "|" << endl;
}
```

Outputs (Note: `_` denotes space):

```
Enter a string :
_abc
|abc|
```

`cin` treats whitespace in the input stream as a delimiter for numeric and string data types. For C-style null-terminated string types, `cin` adds the null byte after the last non-whitespace character stored in memory:

```
// Trailing Whitespace
// trailing.cpp

#include <iostream>
using namespace std;

int main() {
    char str[11];

    cout << "Enter a string : " << endl;
    cin >> str;
    cout << "|" << str << "|" << endl;
}
```

Outputs (Note: `_` denotes space):

```
Enter a string :
_abc_
|abc|
```

The `istream` type supports the following member functions:

- `ignore(...)`: Ignores/discards character(s) from the input buffer
- `get(...)`: Extracts a character or a string from the input buffer
- `getline(...)`: Extracts a line of characters from the input buffer

For detailed descriptions of `get()` and `getline()`, see the chapter entitled Input and Output Refinements.

`ignore()`

The `ignore()` member function extracts characters from the input buffer and discards them. `ignore()` does not skip leading whitespace. Two versions of `ignore()` are available:

```
cin.ignore();
cin.ignore(2000, '\n');
```

The no-argument version discards a single character. The two-argument version removes and discards up to the specified number of characters or up to the specified delimiting character, whichever occurs first and discards the delimiting character. The default delimiter is end-of-file (not end-of-line).

`cout`

The `cout` object is an instance of the `ostream` type. An `ostream` object copies data from system memory into an output stream; in copying, it converts the data in system memory into a sequence of characters.

The general expression for inserting data into the standard output stream takes the form

```
cout << identifier
```

where `<<` is the insertion operator and `identifier` is the name of the variable or object that holds the data.

For example,

```
int i = 6;
char c = ' ';
double x = 9.75;
char s[] = "Harry";
cout << i;
cout << c;
cout << x;
cout << c;
cout << s;
cout << endl;
cout << "Data has been written";
```

Outputs:

```
6 9.75 Harry
Data has been written
```

`endl` inserts a newline character into the stream and flushes the stream's buffer.

We may combine these expressions into a single statement that specifies multiple insertions:

```
int i = 6;
char c = ' ';
double x = 9.75;
char s[] = "Harry";

cout << i << c << x << c << s << endl;

cout << "Data has been written";
```

```
6 9.75 Harry
Data has been written
```

We call such repeated use of the insertion operator *cascading*.

The `ostream` type supports the following public member functions for formatting conversions:

- `width(int)`: Sets the field width to the integer received
- `fill(char)`: Sets the padding character to the character received
- `setf(...)`: Sets a formatting flag to the flag received

- `unsetf(...)`: Unsets a formatting flag for the flag received
- `precision(int)`: Sets the decimal precision to the integer received

`width()`

The `width(int)` member function specifies the minimum width of the **next** output field:

```
// Field Width
// width.cpp

#include <iostream>
using namespace std;

int main() {
    int attendance = 27;
    cout << "1234567890" << endl;
    cout.width(10);
    cout << attendance << endl;
    cout << attendance << endl;
}
```

```
1234567890
        27
27
```

`width(int)` applies only to the next field. Note how the field width for the first display of `attendance` is 10, while the field width for the second display of `attendance` is just the minimum number of characters needed to display the value (2).

`fill()`

The `fill(char)` member function defines the padding character. The output object inserts this character into the stream wherever text occupies less space than the specified field width. The default fill character is `' '` (space). To pad a field with `'*'`'s, we add:

```
// Padding
// fill.cpp

#include <iostream>
using namespace std;

int main() {
```

```
    int attendance = 27;
    cout << "1234567890" << endl;
    cout.fill('*');
    cout.width(10);
    cout << attendance << endl;
}
```

```
1234567890
********27
```

The padding character remains unchanged, until we reset it.

`setf()`, `unsetf()`

The `setf()` and `unsetf()` member functions control formatting and alignment. Their control flags include:

Control Flag	Result
`ios::fixed`	ddd.ddd
`ios::scientific`	d.dddddd Edd
`ios::left`	align left
`ios::right`	align right

The scope resolution (`ios::`) on these flags identifies them as part of the `ios` class.

`setf()`, `unsetf()` - Formatting

The default format in C++ is *general format*, which outputs data in the simplest, most succinct way possible (1.34, 1.345E10, 1.345E-20). To output a fixed number of decimal places, we select *fixed format*. To specify fixed format, we pass the `ios::fixed` flag to `setf()`:

```
// Fixed Format
// fixed.cpp

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

```
int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.width(10);
    cout.setf(ios::fixed);
    cout << pi << endl;
}
```

```
1234567890
  3.141593
```

Format settings persist until we change them. To unset fixed format, we pass the `ios::fixed` flag to the `unsetf()` member function:

```
// Unset Fixed Format
// unsetf.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.width(10);
    cout.setf(ios::fixed);
    cout << pi << endl;
    cout.unsetf(ios::fixed);
    cout << pi << endl;
}
```

```
1234567890
  3.141593
3.14159
```

To specify scientific format, we pass the `ios::scientific` flag to the setf() member function:

```
// Scientific Format
// scientific.cpp

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

```
int main() {
    double pi = 3.141592653;
    cout << "12345678901234" << endl;
    cout.width(14);
    cout.setf(ios::scientific);
    cout << pi << endl;
}
```

```
12345678901234
  3.141593e+00
```

To turn off scientific format, we pass the `ios::scientific` flag to the `unsetf()` member function.

`setf()`, `unsetf()` - Alignment

The default alignment is right-justified.

To specify left-justification, we pass the `ios::left` flag to the s`etf()` member function:

```
// Left Justified
// left.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.width(10);
    cout.fill('?');
    cout.setf(ios::left);
    cout << pi << endl;
}
```

```
1234567890
3.14159???
```

To turn off left-justification, we pass the `ios::left` flag to the `unsetf()` member function:

```
cout.unsetf(ios::left);
```

`precision()`

The `precision()` member function sets the precision of subsequent floating-point fields. The default precision is `6` units. General, fixed, and scientific formats implement precision differently. General format counts the number of significant digits. Scientific and fixed formats count the number of digits following the decimal point.

For a precision of `2` under general format, we write

```
// Precision
// precison.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.setf(ios::fixed);
    cout.width(10);
    cout.precision(2);
    cout << pi << endl;
}
```

```
1234567890
      3.14
```

The precision setting applies to the output of all subsequent floating-point values until we change it.

Student Example

The code snippet produces the output shown below:

```
void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
```

```
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}
```

```
975:
  78.90
  69.40
```

Summary

- Object-oriented classes may contain both data members and member functions
- The keyword `private` identifies subsequent members as inaccessible to any client
- The keyword `public` identifies subsequent members as accessible to any client
- Data members hold the information about an object's state
- Member functions describe the logic that an object performs on its data members
- A query reports the state of an object without changing its state
- A modifier changes the state of an object
- An empty state is the set of data values that identifies the absence of valid data in an object
- A field width setting only holds for the next field
- All settings other than a field width setting persist until changed
- Precision has different meanings under general, scientific, and fixed formats