Construction and Destruction

- Describe some basic features of a class
- Introduce constructors and destructors as essential to encapsulation
- Overload constructors to enhance client communication

> "A class is a cohesive package that ... describes the rules by which objects behave; these objects are referred to as instances of that class." **Wikipedia, 2008.**

Object-oriented languages encapsulate the state and logic of a type using a class. A class describes the structure of the data that its objects hold and the rules under which its member functions access and change that data. The implementation of a well-encapsulated class has all details hidden within itself. Client code communicates with the objects of the class solely through its public member functions.

This chapter describes some basic features of classes and introduces the special member functions that initialize and tidy up objects of a class. This chapter covers the order of memory allocation and deallocation during object construction and destruction as well as overloading of the special function that initializes objects.

Class Features

Instance of a Class

Each object or instance of a class occupies its own region of memory. The data for the object is stored in that region of memory.

A definition of an object takes the form

```
Type Identifier;
```

`Type` is the name of the class. `Identifier` is the name of the object.

Consider the following class definition

```
const int NG = 20;

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

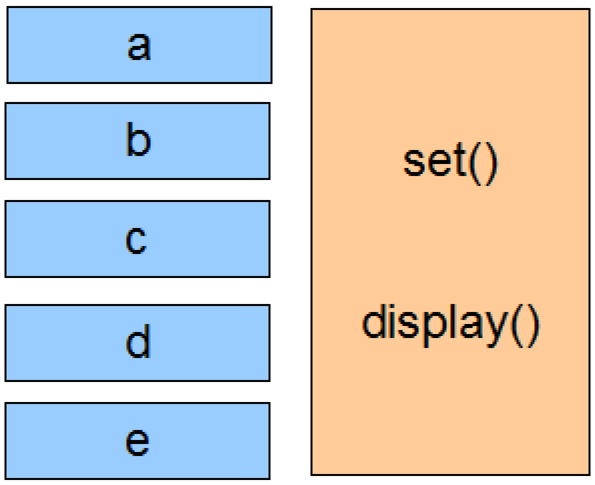
To create an object of our `Student` class named `harry`, we write:

```
Student harry;
```

To create five objects of our `Student` class, we write:

```
Student a, b, c, d, e;
```

The compiler allocates five regions in static memory, each of which holds the data for one of the five objects. Each region stores the values of three data members - `no`, the array `grade` and `ng`. The compiler stores the member function instructions separately and only once for all objects of the class.



Instance Variables

We call the data members in the class definition the object's instance variables. Instance variables may be of:

- Fundamental type (`int`, `double`, `char`, etc.)

- Compound type
    - Class type (`struct` or `class`)
    - Pointer type (to instances of data types - fundamental or compound)
    - Reference type (to instances of data types - fundamental or compound)

Logic

The member function instructions apply to all objects of the class and there is no need to allocate separate logic memory for each object. At run-time each call to a member function on an object accesses the same code, but different instance variables - those of the object on which the client code has called the member function.

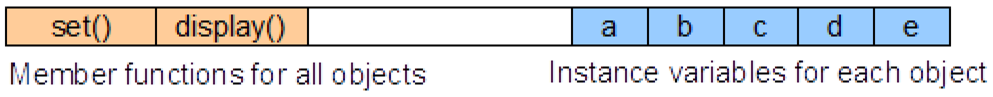
Consider the following client code. This code calls the same member function (`display()`) on five different `Student` objects and displays five different sets of information in the same format:

```
Student a, b, c, d, e;

// different data for each object - same logic

a.display();  // displays the data stored in a
cout << endl;
b.display();  // displays the data stored in b
cout << endl;
c.display();  // displays the data stored in c
cout << endl;
d.display();  // displays the data stored in d
cout << endl;
e.display();  // displays the data stored in e
cout << endl;
```

The memory allocated for member function code is shown on the left. The memory allocated for the instance variables is shown on the right:


set()
display()
a
b
c
d
e
Member functions for all objects
Instance variables for each object

Class Privacy

C++ compilers apply privacy at the class level. Any member function can access any private member of its class, including any data member of any instance of its class. In other words, privacy is not implemented at the *individual object level*.

In the following example, we refer to private data members of a `Student` object within a member function called on a different `Student` object:

```cpp
const int NG = 20;

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

// ...

void Student::copyFrom(const Student& src) {
    no = src.no; // copy data from one object to another
    ng = src.ng; // copy data from one object to another
    for (int i = 0; i < NG; i++)
        grade[i] = src.grade[i]; // copy from one object to another
}

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;
    }
}

int main() {
    Student harry, backup;
    float grade[] = {78.9f, 67.5f, 45.5f, 64.35f};
    harry.set(975, grade, 4);
    backup.copyFrom(harry);
    backup.display();
}
```

Output:

```
975:
 78.90
 67.50
 45.50
 64.35
```

The `copyFrom(const Student& src)` member function copies the values of the private data members of `harry` to the private data members of `backup`.

Constructor

Complete encapsulation requires a mechanism for initializing data members at creation-time. Without initialization at creation-time, an object's data members contain undefined values until client code calls a modifier that sets that data. Before any modifier call, client code can inadvertently 'break' the object by calling a member function that assumes valid data. For instance, client code could call `display()` before ever calling `set()`.

The following code generates the spurious output below:

```
// Calling an Object with Uninitialized Data
// uninitialized.cpp

#include <iostream>
using namespace std;
const int NG = 20;

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

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

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;
    }
}

int main() {
    Student harry;
    harry.display();

    float grade[] = {78.9f, 67.5f, 45.55f};
    harry.set(975, grade, 3);
    harry.display();
}
```

```
12052848

975:
 78.90
 67.50
 45.55
```

Initially `harry`'s student number, grades and their number are undefined. If the value stored in `ng` is negative, the first call to `display()` outputs an unrecognizable student number and no grades. After the call to `set()`, the data values are defined and the subsequent call to `display()` produces recognizable results.

To avoid undefined behavior or broken objects, we need to initialize each object to an empty state at creation-time.

Definition

The special member function that any object invokes at creation-time is called its class' constructor. We use the default constructor to execute any preliminary logic and set the object to an empty state.

The default constructor takes its name from the class itself. The prototype for this no-argument constructor takes the form

```
Type();
```

`Type` is the name of the class. Its declaration does not include a return type.

Example

To define a default constructor for our Student class, we declare its prototype explicitly in the class definition:

```
const int NG = 20;

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

We define the constructor in the implementation file:

```
Student::Student() {
    no = 0;
    ng = 0;
}
```

Default Behavior

If we don't declare a constructor in the class definition, the compiler inserts a default no-argument constructor with an empty body:

```
Student::Student() {
}
```

Note that this default constructor leaves the instance variables uninitialized.

Understanding Order

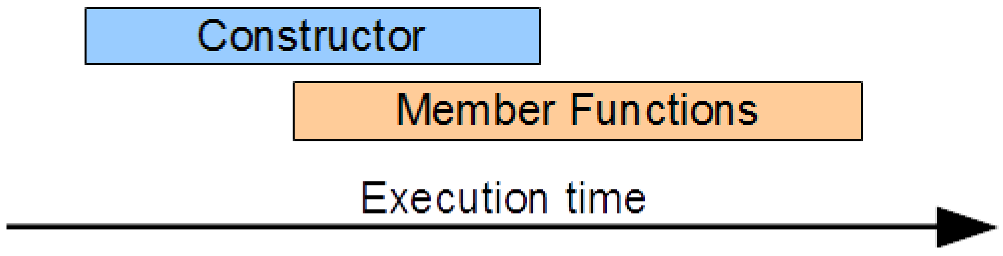
Construction

The compiler assembles an object in the following order:

1. Allocates memory for each instance variable in the order listed in the class definition
2. Executes the logic, if any, within the constructor's definition

Member Function Calls

Since the constructor starts executing at instantiation, no normal member function is called before the constructor. Every normal member function is called after instantiation.



Multiple Objects

The compiler creates multiple objects defined in a single declaration in the order specified by the declaration.

For example, the following code generates the output below:

```
// Constructors
// constructors.cpp

#include <iostream>
#include <cstring>
using namespace std;
const int NG = 20;

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

// initializes the data members
//
Student::Student() {
    cout << "In constructor" << endl;
    no = 0;
    ng = 0;
}

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

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;
    }
}

int main () {
    Student harry, josee;
    float gh[] = {89.4f, 67.8f, 45.5f};
    float gj[] = {83.4f, 77.8f, 55.5f};
    harry.set(1234, gh, 3);
    josee.set(1235, gj, 3);
    harry.display();
    josee.display();
}
```

```
In constructor
In constructor
1234:
 89.40
 67.80
 45.50
1235:
 83.40
 77.80
 55.50
```

The compiler assembles `harry` and calls its constructor first and assembles `josee` and calls its constructor afterwards.

Safe Empty State

Initializing an object's instance variables in a constructor ensures that the object has a well-defined state from the *time of its creation*. In the above example, we say that `harry` and `josee` are in *safe empty states* until the `set()` member function changes those states. If client code calls member functions on objects in safe empty states, the objects do not break and behave as expected.

For example, the following client code produced the `no data available` message listed below:

```
// Safe Empty State
// safeEmpty.cpp

#include <iostream>
using namespace std;

int main ( ) {
    Student harry, josee;

    harry.display();
    josee.display();
    float gh[] = {89.4f, 67.8f, 45.5f};
    float gj[] = {83.4f, 77.8f, 55.5f};
    harry.set(1234, gh, 3);
    josee.set(1235, gj, 3);
    harry.display();
    josee.display();
}
```

```
In constructor
In constructor
no data available
no data available
1234:
 89.40
 67.80
 45.50
1235:
 83.40
 77.80
 55.50
```

The safe empty state is identical for all objects of the same class.

Destructor

Complete encapsulation also requires a mechanism for tidying up at the end of an object's lifetime. An object with dynamically allocated memory needs to deallocate that memory before going out of scope. An object that has written data to a file needs to flush the file's buffer and close the file before going out of scope.

Definition

The special member function that every object invokes before going out of scope is called its class' destructor. We code all of the terminal logic in this special member function.

The destructor takes its name from the class itself, prefixing it with the tilde symbol (~). The prototype for a destructor takes the form

```
~Type();
```

`Type` is the name of the class. Destructors have no parameters or return values.

An object's destructor:

- is called automatically.
- cannot be overloaded.
- should not be called explicitly.

Example

To define the destructor for our `Student` class, we declare its prototype in the class definition:

```
const int NG = 20;

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

We define the member function in the implementation file:

```
Student::~Student() {
    // insert our terminal code here
}
```

Default Behaviour

If we don't declare a destructor in the class definition, the compiler inserts a destructor with an empty body:

```
Student::~Student() {
}
```

Understanding Order

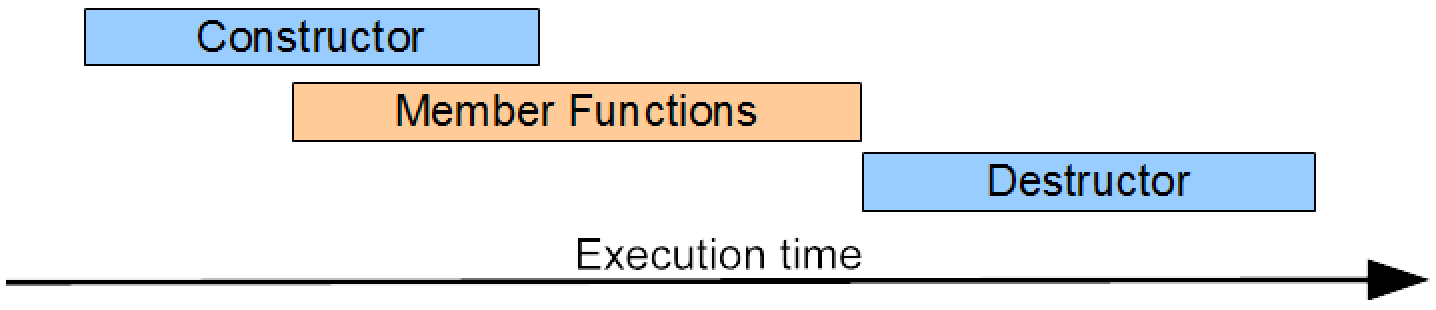
Destruction

Object destruction proceeds in the following order:

1. Execute the logic of the object's destructor
2. Deallocate memory for each instance variable in opposite order to that listed in the class definition

Member Function Calls

An object's destructor starts executing only after every normal member function has completed its execution.



Client code cannot call any member function on an object after the object has called its destructor and gone out of scope.

Multiple Objects

The compiler destroys sets of objects in opposite order to that of their creation.

For example, the following code generates the output below:

```
// Constructors and Destructors
// destructors.cpp

#include <iostream>
#include <cstring>
using namespace std;
const int NG = 20;

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

Student::Student() {
    cout << "In constructor" << endl;
    no = 0;
    ng = 0;
}

// executed before object goes out of scope
//
Student::~Student() {
cout << "In destructor for " << no
        << endl;
}

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

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;
    }
}

int main () {
    Student harry, josee;
    float gh[] = {89.4f, 67.8f, 45.5f};
    float gj[] = {83.4f, 77.8f, 55.5f};
    harry.set(1234, gh, 3);
    josee.set(1235, gj, 3);
    harry.display();
    josee.display();
}
```

```
In constructor
In constructor
1234:
 89.40
 67.80
 45.50
1235:
 83.40
 77.80
 55.50
In destructor for 1235
In destructor for 1234
```

The compiler destroys `josee` first followed by `harry`.

Construction and Destruction of Arrays

The order of construction and destruction of elements of an array of objects follows the order described above.

The compiler creates the elements of an array one at a time sequentially starting from the first element and ending with the last. Each object calls the default constructor at creation-time. When the array goes out of scope, the last element calls its destructor first and the first element calls its destructor last.

For example, the following code generates the output below:

```
// Constructors, Destructors and Arrays
// ctorsDtorsArrays.cpp

#include <iostream>
#include <cstring>
using namespace std;
const int NG = 20;

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

Student::Student() {
    cout << "In constructor" << endl;
    no = 0;
    ng = 0;
}

Student::~Student() {
cout << "In destructor for " << no
        << endl;
}

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
```

```
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

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;
    }
}

int main () {
    Student a[3];
    float g0[] = {89.4f, 67.8f, 45.5f};
    float g1[] = {83.4f, 77.8f, 55.5f};
    float g2[] = {77.8f, 83.4f, 55.5f};
    a[0].set(1234, g0, 3);
    a[1].set(1235, g1, 3);
    a[2].set(1236, g2, 3);
    for (int i = 0; i < 3; i++)
        a[i].display();
}
```

```
In constructor
In constructor
In constructor
1234:
 89.40
 67.80
 45.50
1235:
 83.40
 77.80
 55.50
1236:
 77.80
 83.40
 55.50
In destructor for 1236
In destructor for 1235
In destructor for 1234
```

The destructor for element `a[2]` executes before the destructor for a`[1]`, which executes before the destructor for `a[0]`. The order of destruction is based on order of construction and not on order of usage.

Overloading Constructors

Overloading a class' constructor adds communication options for client code. Client code can select the most appropriate set of arguments at creation time.

For example, to let client code initialize a `Student` object with a student number and a set of grades, let us define a three-argument constructor similar to our `set()` function:

```
// Overloaded Constructor
// overload.cpp

#include <iostream>
using namespace std;
const int NG = 20;

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

```
};

Student::Student() {
    cout << "In constructor" << endl;
    no = 0;
    ng = 0;
}

Student::Student(int sn, const float* g, int ng_) {
    cout << "In 3-arg constructor" << endl;
    set(sn, g, ng_);
}

Student::~Student() {
cout << "In destructor for " << no
        << endl;
}

void Student::set(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        no = 0;
        ng = 0;
    }
}

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;
    }
}

int main () {
    float gh[] = {89.4f, 67.8f, 45.5f};
    float gj[] = {83.4f, 77.8f, 55.5f};
    Student harry(1234, gh, 3), josee(1235, gj, 3);
    harry.display();
    josee.display();
}
```

```
In 3-arg constructor
In 3-arg constructor
1234:
 89.40
 67.80
 45.50
1235:
 83.40
 77.80
 55.50
In destructor for 1235
In destructor for 1234
```

This new constructor includes the validation logic by calling `set()`. The compiler calls only one constructor at creation-time. In this example, the compiler does not call the default constructor.

No-argument constructor is not always implemented

If the class definition includes the prototype for a constructor with some parameters but does not include the prototype for a no-argument default constructor, the compiler **DOES NOT** insert an empty-body, no-argument default constructor. The compiler only inserts an empty-body, no-argument default constructor if the class definition does not declare **ANY** constructor.

If we define a constructor with some parameters, we typically also define a no-argument default constructor. This is important in the creation of arrays of objects. The creation of each element in the array requires a no-argument default constructor.

Summary

- We refer to the data members of an object as its instance variables
- Privacy operates at the class level, not at the object level
- The constructor is a special member function that an object invokes at creation time
- The name of the constructor is the name of the class
- The destructor is a special member function that an object invokes at destruction time
- The name of the destructor is the name of the class prefixed by a `~`
- The constructor and destructor do not have return types
- The compiler inserts an empty body constructor/destructor into any class definition that does not declare a constructor/destructor
- The compiler does not insert an empty-body, no-argument constructor into a class definition that declares any form of constructor