Chapter 11 | Classes 1

CS185

Classes and Objects

Introduction to Object-Oriented Programming

Procedural programming vs. Object-Oriented programming

Procedural programming:

- There is a distinct division between code (functions) and data.
- Data is passive, it gets acted upon by functions.
- We generally pass the data between functions.

Object-Oriented programming:

- Code and data are encapsulated together in a single *object*.
- The functions that work on the data are part of the object.
- We don't have to pass data to the functions.

Usually, for a language to be considered object-oriented, it should have these three properties:

1. Encapsulation (data abstraction/hiding)
2. Inheritance (relationships between entities)
3. Polymorphism (runtime decisions)

In C++, these three properties are realized as:

1. Classes and objects
2. Extending classes with an *is-a* relationship
3. Virtual methods and dynamic binding

Procedural Programming

Let's use this structure that represents a student:

```
const int MAXLENGTH = 10;

struct Student
{
        char login[MAXLENGTH];
        int age;
        int year;
        float GPA;
};
```

```
void display_student(const Student &student)
{
  using std::cout;
  using std::endl;

  cout << "login: " << student.login << endl;
  cout << "  age: " << student.age << endl;
  cout << " year: " << student.year << endl;
  cout << "  GPA: " << student.GPA << endl;
}
```

This allows this code:

Code

```
void f1(void)
{
        Student st1;

        st1.age = 20;
        st1.GPA = 3.8;
        strcpy_s(st1.login, "jdoe");
        st1.year = 3;

        display_student(st1);
}
```

Output

```
login: jdoe
  age: 20
 year: 3
  GPA: 3.8
```

As well as this:

Code

```
void f2(void)
{
        Student st2;

        st2.age = -5;
        st2.GPA = 12.9;
        strcpy_s(st2.login, "rumplestiltzkin");
        st2.year = 150;

        display_student(st2);
}
```

Output

```
Debug Assertion Failed!
Expression (L"Buffer is too small" && 0)
```

A second attempt to "protect" the data by using functions to set the data instead of the user *directly* modifying it.

```
void set_login(Student &student, const char* login);
void set_age(Student &student, int age);
void set_year(Student &student, int year);
void set_GPA(Student &student, float GPA);
```

```
void set_login(Student &student, const char* login)
{
	int len = std::strlen(login);
	strncpy_s(student.login, login, MAXLENGTH - 1);
	if (len >= MAXLENGTH)
	{
		student.login[MAXLENGTH - 1] = 0;
	}
}
```

```
void set_age(Student &student, int age)
{
	if ((age < 18) || (age > 100))
	{
		std::cout << "Error in age range!\n";
		student.age = 18;
	}
	else
	{
		student.age = age;
	}
}
```

```
void set_year(Student &student, int year)
{
	if ((year < 1) || (year > 4))
	{
		std::cout << "Error in year range!\n";
		student.year = 1;
	}
	else
	{
		student.year = year;
	}
}
```

```
void set_GPA(Student &student, float GPA)
{
	if ((GPA < 0.0) || (GPA > 4.0))
	{
		std::cout << "Error in GPA range!\n";
		student.GPA = 0.0;
	}
	else
	{
		student.GPA = GPA;
	}
}
```

Code	
	`void f3(void)` `{` ` Student st3;` ` set_age(st3, -5);` ` set_GPA(st3, 12.9);` ` set_login(st3, "rumplestiltzkin");` ` set_year(st3, 150);` ` display_student(st3);` `}`
Output	**Error in age range!** **Error in GPA range!** **Error in year range!** **login: rumplesti** **age: 18** **year: 1** **GPA: 0**

Notes:

- It may not be perfect yet, but at least we can try to prevent memory corruption.
- There is still nothing preventing the user (programmer) from accessing the fields directly.
- We can "hide" (*encapsulate*) the data fields by using the **`private`** access specifier:

```
struct Student
{
private:
	char login[MAXLENGTH];
	int age;
	int year;
	float GPA;
};
```

This code will give errors now:

Illegal Code

```
Student st1;

st1.age = 20;
st1.GPA = 3.8;
strcpy_s(st1.login, "jdoe");
st1.year = 3;
```

Error

Severity	Code	Description
Error	C2248	'Student::year': cannot access private member declared in class 'Student'
Error	C2248	'Student::login': cannot access private member declared in class 'Student'
Error	C2248	'Student::GPA': cannot access private member declared in class 'Student'
Error	C2248	'Student::age': cannot access private member declared in class 'Student'

However, even this code will no longer work:

```
void set_age(Student &student, int age)
{
	if ((age < 18) || (age > 100))
	{
		std::cout << "Error in age range!\n";
		student.age = 18;  // ERROR, age is private
	}
	else
	{
		student.age = age; // ERROR, age is private
	}
}
```

The solution? Put the functions *inside* the Student **`struct`** along with the data. This is *encapsulation*.

In C++, encapsulated functions are generally called *methods*.

Encapsulating Functions and Data

Adding functions to the structure is simple. By declaring them in a **`public`** section, the functions (methods) will be accessible from outside of the structure:

Structure with private data, public methods

```
const int MAXLENGTH = 10;

struct Student
{
private:
	char login[MAXLENGTH];
	int age;
	int year;
	float GPA;

public:
	void set_login(const char* login_);
	void set_age(int age_);
	void set_year(int year_);
	void set_GPA(float GPA_);
};
```

Client access is through the public methods

```
void f1(void)
{
	// Create a Student struct (object)
	Student st1;

	// Set the fields using the public methods
	st1.set_login("jdoe");
	st1.set_age(22);
	st1.set_year(4);
	st1.set_GPA(3.8);

	st1.age = 10; // ERROR, private
	st1.year = 2; // ERROR, private
}
```

The implementation of the methods will change slightly:

```
void Student::set_login(const char* login_)
{
	int len = std::strlen(login);
	strncpy_s(login, login, MAXLENGTH - 1);
	if (len >= MAXLENGTH)
	{
		login[MAXLENGTH - 1] = 0;
	}
}
```

```
void Student::set_age(int age_)
{
	if ((age_ < 18) || (age_ > 100))
	{
		std::cout << "Error in age range!\n";
		age = 18;
	}
	else
	{
		age = age_;
	}
}
```

```
void Student::set_year(int year_)
{
	if ((year_ < 1) || (year_ > 4))
	{
		std::cout << "Error in year range!\n";
		year = 1;
	}
	else
	{
		year = year_;
	}
}
```

```
void Student::set_GPA(float GPA_)
{
	if ((GPA_ < 0.0) || (GPA_ > 4.0))
	{
		std::cout << "Error in GPA range!\n";
		GPA = 0.0;
	}
	else
	{
		GPA = GPA_;
	}
}
```

You'll notice a few things about these implementations:

- We no longer need to pass a reference to the Student to the method. (The methods "know" which object they are accessing.)
- This means we don't have to use the structure dot operator (i.e. `age` instead of `st1.age`)
- We need to indicate that the method belongs to the Student structure by using the scope resolution operator:

Method vs Global function

```
// This method belongs to the Student
// struct
void Student::set_year(int year_)
{
  if ((year_ < 1) || (year_ > 4))
  {
       year = 1;
  }
  else
  {
    year = year_;
  }
}
```

```
// This is a "normal" global function
void set_year(int year)
{
       // body of function
}
```

Incidentally, the default access for a **`struct`** is public. (This is for C compatibility.) These two structures are identical:

Menmbers are public by default

```
struct Student
{
       char login[MAXLENGTH];
       int age;
       int year;
       float GPA;
};
```

OK, but redundant

```
struct Student
{
public:
       char login[MAXLENGTH];
       int age;
       int year;
       float GPA;
};
```

You generally won't see the **`public`** keyword used with structures.

Finally, we need to get back a way to display the values. Our original *display_student* no longer can access the private members, so we have to make it part of the *Student* structure:

Add the *display* method	Modify the implementation
`struct Student` `{` `private:` `  char login[MAXLENGTH];` `  int age;` `  int year;` `  float GPA;` `public:` `  void set_login(const char* login);` `  void set_age(int age);` `  void set_year(int year);` `  void set_GPA(float GPA);` `  void display(void);` `};`	`void Student::display(void)` `{` `  using std::cout;` `  using std::endl;` `  cout << "login: " << login << endl;` `  cout << "  age: " << age << endl;` `  cout << " year: " << year << endl;` `  cout << "  GPA: " << GPA << endl;` `}`

Now, this is how we use it:

```
void f1(void)
{
	// Create a Student object
	Student st1;

	// Using the public methods
	st1.set_login("jdoe");
	st1.set_age(22);
	st1.set_year(4);
	st1.set_GPA(3.8);

	// Tell the object to display itself
	st1.display();
}
```

- Now that the data in the structure is private, the user can't "screw" it up by assigning arbitrary values.
- This is one of the major reasons for "hiding" (encapsulating) the data.
- There are still many problems with the code and we will address those soon.

Classes

In short, a **class** is identical to a **struct** with one (almost) exception: the default accessibility is **private**.

These will work the same:

Default for struct is public	Explicit public
`struct Student` `{` `  char login[MAXLENGTH];` `  int age;` `  int year;` `  float GPA;` `};`	`class Student` `{` `public:` `  char login[MAXLENGTH];` `  int age;` `  int year;` `  float GPA;` `};`

And these will work the same:

Explicit private	Default for class is private
`struct Student` `{` `private:` `  char login[MAXLENGTH];` `  int age;` `  int year;` `  float GPA;` `};`	`class Student` `{` `  char login[MAXLENGTH];` `  int age;` `  int year;` `  float GPA;` `};`

We will generally be using the **`class`** keyword when creating new types that have methods associated with them. We'll use the **`struct`** keyword for POD types. (**P**lain **O**ld **D**ata types).

Initializing Objects: The Constructor

This is the problem we need to solve:

Code	`Student s;  // Uninitialized student` `s.display(); // ???`
Output	**login: ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠9$fÇ** **age: -858993460** **year: -858993460** **GPA: -1.07374e+08**

We never want to have any objects that are in an *undefined* state. Ever.

Recall how we initialize structures:

```
struct Student
{
	// Public by default
	char login[MAXLENGTH];
	int age;
	int year;
	float GPA;
};

void f(void)
{
	// Uninitialized Student
	Student st1;

	// Set values by assignment
	strcpy_s(st1.login, "jdoe");
	st1.age = 20;
	st1.year = 3;
	st1.GPA = 3.08;

	// Set values by initialization
	Student john = { "jdoe", 20, 3, 3.10f };
	Student jane = { "jsmith", 19, 2, 3.95f };
}
```

But with private data, using the initializer list is illegal:

```
class Student
{
	// Private by default
	char login[MAXLENGTH];
	int age;
	int year;
	float GPA;
};

void f(void)
{
	// This is now illegal
	Student john = { "jdoe", 20, 3, 3.10f };
	Student jane = { "jsmith", 19, 2, 3.95f };
}
```

You'll get errors like these:

Severity	Code	Description
Error	C2440	'initializing': cannot convert from 'initializer list' to 'Student'
Error	C2440	'initializing': cannot convert from 'initializer list' to 'Student'

So, we declare another method that will be called to *construct* (initialize) the object: (notice the order of **`public`** and **`private`**, the order is arbitrary)

```
class Student
{
public:
	// Constructor
	Student(const char * login_, int age_, int year_, float GPA_);

	void set_login(const char* login_);
	void set_age(int age_);
	void set_year(int year_);
	void set_GPA(float GPA_);
	void display(void);

private:
	char login[MAXLENGTH];
	int age;
	int year;
	float GPA;
};
```

We can easily implement this method by simply calling the other methods:

```
Student::Student(const char * login_, int age_, int year_, float GPA_)
{
	set_login(login_);
	set_age(age_);
	set_year(year_);
	set_GPA(GPA_);
}

// The client can initialize now
void f(void)
{
	// Set values by constructor
	Student john("jdoe", 20, 3, 3.10f);
	Student jane("jsmith", 19, 2, 3.95f);
}
```

Notes:

- The constructor solves the initialization problem.
- The initialization seems almost automatic.
- It also prevents any Student classes from ever being uninitialized. This is a very important and powerful feature.
- C++11 allows brace-initialization:

```
void f()
{
    // Set values by constructor
  Student john {"jdoe", 20, 3, 3.10f};
  Student jane {"jsmith", 19, 2, 3.95f};
}
```

Accessors and Mutators (Gettors and Settors)

Since the data in a class is usually **`private`**, the only way to gain access to it is by providing **`public`** methods that explicitly allow it.

- A method that allows you to *read* a private value is called an *accessor method*. (Also called a *gettor method*)
- A method that allows you to *write* (change) a private value is called a *mutator method*. (Also called a *settor method*)
- A class may provide one, both, or none of these method types.
    - If only an accessor is provided for a private data member, the data is considered *read-only*
    - If only a mutator is provided for a private data member, the data is considered *write-only*
    - If both methods are provided for a private data member, the data is considered *read-write*

All of the data in the Student class is write-only, since we can change it, but we can't read it.

Adding accessors

```
class Student
{
public:
  // Constructor
  Student(const char * login_, int
age_,
          int year_, float GPA_);

  // Accessors (gettors)
  int get_age(void);
  int get_year(void);
  float get_GPA(void);
  const char *get_login(void);

  // Mutators (settors)
  void set_login(const char* login_);
  void set_age(int age_);
  void set_year(int year_);
  void set_GPA(float GPA_);
  void display(void);

private:
  char login[MAXLENGTH];
  int age;
  int year;
  float GPA;
};
```

Implementations

```
int Student::get_age(void)
{
       return age;
}

int Student::get_year(void)
{
       return year;
}

float Student::get_GPA(void)
{
       return GPA;
}

const char *Student::get_login(void)
{
       return login;
}
```

Providing (or not providing) accessors and mutators is how you control access and modifications to the private data. What if you didn't want to allow the client to change the login?

Resource Management

The Student class so far:

- All Student objects are initialized.
- Can't corrupt private data. (Public methods validate)
- Login name is limited to 10 chars and can be truncated. (Safe, but not ideal)

We need to change the login so its length is determined *at run-time* (read: dynamically).

Adding accessors

```
class Student
{
public:
  // Public interface ...

private:
  char *login;
  int age;
  int year;
  float GPA;
};
```

Implementations

```
void Student::set_login(const char* login_)
{
	int len = std::strlen(login_);
	login = new char[len + 1];
	strcpy_s(login, len + 1, login_);
}
```

The client doesn't even know there has been a change:

```
void foo(void)
{
	// Construct a Student object
	Student john("jrumplestiltzkin", 20, 3, 3.10f);

	// This will display all of the data
	john.display();
}
```

If you are going to allow the user to call `set_login`, then you'll need to modify the function a little bit more :

Modified method

```
void Student::set_login(const char* login_)
{
	// In case we already had a login
	delete[] login;

	// Now create a new one
	int len = std::strlen(login_);
	login = new char[len + 1];
	strcpy_s(login, len + 1, login_);
}
```

You will also need to add this line as the first line in the constructor, to insure that it has been initialized.

```
login = 0;
```

That is the only change required (sort of). What is the problem?

Adding interface method	Implementations
class Student { public: // Public stuff ... void free_login(void); private: // Private stuff ... };	void Student::free_login(void) { delete[] login; }

Now, the client will do this:

```
void foo(void)
{
	// Construct a Student object
	Student john("jdoe", 20, 3, 3.10f);

	// This will display all of the data
	john.display();

	// Release the memory for login
	john.free_login();
}
```

But this is wrong on so many levels... Here are two of them:

```
void foo(void)
{
  // Construct a Student object
  Student john("jdoe", 20, 3, 3.10f);

  // This will display all of the data
  john.display();

  // Oops, memory leak now!
}
```

```
void foo(void)
{
  // Construct a Student object
  Student john("jdoe", 20, 3, 3.10f);

  // Release the memory for login
  john.free_login();

  // Oops, very bad now!
  john.display();
}
```

The Bad News:

- The client is responsible for the class' private memory.
- The client *will* forget to call the method.
- The client *will* call it and continue to use the object
- The client *will* call it twice (or more).
- The client needs to understand about the internal data structures in use.
- Probably other reasons...

In order to make sure that the memory is deleted, we need something like a constructor in reverse. Let's call it a *destructor*.

Destroying Objects: The Destructor

We'd like some code that will be called when the client is done with the object. The code is another method called a *destructor* and is similar to the constructor.

Adding destructor	Implementations
`class Student` `{` `public:` `  // Constructor` `  Student(const char * login_, int age_,` `  int year_, float GPA_);` `  // Destructor` `  ~Student(void);` `  // Other public members` `private:` `  // private members` `};`	`Student::~Student(void)` `{` `  // Free the memory that was allocated` `  delete[] login;` `}`

Now this code is fine:

```
void foo(void)
{
	// Construct a Student object
	Student john("jdoe", 20, 3, 3.10f);

	// This will display all of the data
	john.display();

} // Destructor is called here.
```

The destructor will be called automatically when the object *goes out of scope*. (The meaning of scope here is the same meaning we've been using since the beginning.)

The compiler is smart about calling the destructor for local objects:

```
void foo(void)
{
	// Construct a Student object
	Student john("jdoe", 20, 3, 3.10f);
	if (john.get_age() > 10)
	{
		Student jane("jsmith", 19, 2, 3.95f);
		if (jane.get_age() > 2)
		{
			return; // Destructor's for jane and john called
		}
	}
}
```

This makes the destructor an extremely powerful concept.

Creating Objects

Let's modify the constructor and destructor to print a message each time they are called:

Modified Constructor

```
Student::Student(const char * login_, int age_, int year_, float GPA_)
{
	login = 0;
	set_login(login_);
	set_age(age_);
	set_year(year_);
	set_GPA(GPA_);
	std::cout << "Student constructor for " << login << std::endl;
}
```

Modified Destructor

```
Student::~Student(void)
{
	std::cout << "Student destructor for " << login << std::endl;

	// Free the memory that was allocated
	delete[] login;
}
```

Example:

Code

```
void foo(void)
{
	std::cout << "***** Begin *****\n";
	Student john("jdoe", 20, 3, 3.10f);
	Student jane("jsmith", 19, 2, 3.95f);
	Student jim("jbob", 22, 4, 2.76f);

	// Modify john
	john.set_age(21);
	john.set_GPA(3.25f);

	// Modify jane
	jane.set_age(24);
	jane.set_GPA(4.0f);

	// Modify jim
	jim.set_age(23);
	jim.set_GPA(2.98f);

	// Display all
	john.display();
	jane.display();
	jim.display();
	std::cout << "***** End *****\n";
}
```

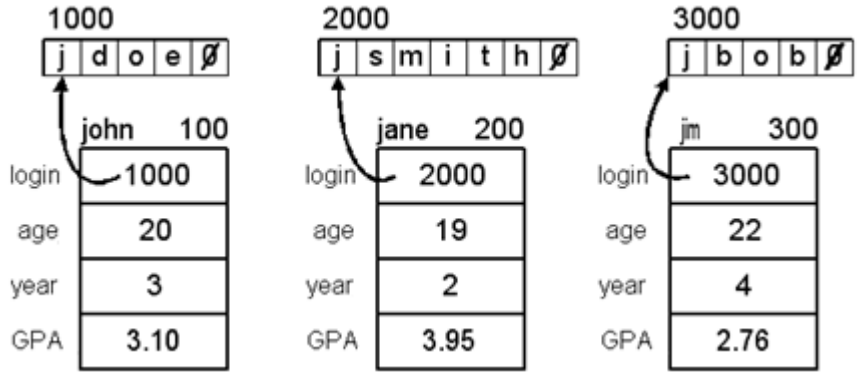
Output

```
***** Begin *****
Student constructor for jdoe
Student constructor for jsmith
Student constructor for jbob
login: jdoe
  age: 21
 year: 3
  GPA: 3.25
login: jsmith
  age: 24
 year: 2
  GPA: 4
login: jbob
  age: 23
 year: 4
  GPA: 2.98
***** End *****
Student destructor for jbob
Student destructor for jsmith
Student destructor for jdoe
```

These three lines:

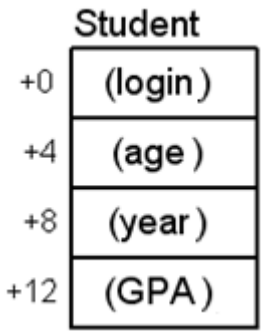
```
Student john("jdoe", 20, 3, 3.10f);
Student jane("jsmith", 19, 2, 3.95f);
Student jim("jbob", 22, 4, 2.76f);
```

will look something like this in memory: (the addresses are arbitrary as usual)



Notes:

- Each object is a separate entity in memory, unrelated to the others.
- Each object has the same exact *structure* (layout) in memory, with possibly different values.
- The names of the fields are for the programmer's convenience (the compiler discards them during compilation).
- A field is accessed via its *offset* from the address of the object.
- Like arrays, the address of the first member is the same as the address of the struct/class.
- For example, the compiler finds the `login` member at offset 0, the `age` member at offset 4, the `year` member at offset 8, and the `GPA` member at offset 12:



- Given the diagrams above, this means that the `age` field for `jane` is at address 204 and the `GPA` field for `jim` is at address 312.
- In fact, for any `Student` object at address *XYZ*, the `age` member will be at address *XYZ + 4*. Always.

Notice that the methods are **not part of the object**. This may seem surprising at first. So how does the *display* method know which data to show?

```
john.display();
jane.display();
jim.display();

void Student::display(void)
{
	using std::cout;
	using std::endl;

	cout << "login: " << login << endl;
	cout << "  age: " << age << endl;
	cout << " year: " << year << endl;
	cout << "  GPA: " << GPA << endl;
}
```

The `this` Pointer

All methods of a class/struct are passed a hidden parameter. This parameter is the address of the invoking object. In other words, the address of the object that you are calling a method on:

```
john.display(); // john is the invoking object
jane.display(); // jane is the invoking object
jim.display();  // jim is the invoking object
```

Really, the *display* method is more like this (with the items in **blue** hidden):

```
void Student::display(Student *this)
{
	using std::cout;
	using std::endl;

	// Members are offset from "this"
	cout << "login: " << this->login << endl;
	cout << "  age: " << this->age << endl;
	cout << " year: " << this->year << endl;
	cout << "  GPA: " << this->GPA << endl;
}
```

The example above would look like this after compiling:

```
john.display(100); // Address of john object passed to display
jane.display(200); // Address of jane object passed to display
jim.display(300);  // Address of jim object passed to display
```

So, in a nutshell, this (no pun intended) is how the magic works. The programmer has access to the **`this`** pointer inside of the methods. (**`this`** is a keyword.)

Both of these lines are the same within *Student::display*:

```
// Normal code
cout << "login: " << login << endl;

// Explicit use of this. (Generally only seen in beginner's code.)
cout << "login: " << this->login << endl;
```

`const` Member Functions

Example:

```
void foo(void)
{
	// Create a constant object
	const Student john("jdoe", 20, 3, 3.10f);

	john.set_age(25); // Error, as expected.
	john.set_year(3); // Error, as expected.

	john.get_age();   // Error, not expected.
	john.display();   // Error, not expected.
}
```

In the "old days", we would make our parameters **const** if we were not going to modify them:

```
void display_student(const Student &student)
{
	using std::cout;
	using std::endl;
	cout << "login: " << student.login << endl;
	cout << "  age: " << student.age << endl;
	cout << " year: " << student.year << endl;
	cout << "  GPA: " << student.GPA << endl;
}


void foo(void)
{
	// Create constant object
	const Student john = { "jdoe", 20, 3, 3.10f };

	// This works just fine with const object
	display_student(john);
}
```

Q: How do we accomplish the same thing with member functions (methods) when we don't pass the data as a parameter?

A: We mark the method as **const**.

You must tag both the declaration (in the class definition) and implementation with the ***const*** keyword:

```
struct Student
{
public:
	// Constructor
	Student(const char * login_, int age_, int year_, float GPA_);

	// Destructor
	~Student(void);

	// Mutators (settors) are rarely const
	void set_login(const char* login_);
	void set_age(int age_);
	void set_year(int year_);
	void set_GPA(float GPA_);

	// Accessor (gettors) are usually const
	int get_age(void) const;
	int get_year(void) const;
	float get_GPA(void) const;
	const char *get_login(void) const;

	// Nothing will be modified
	void display(void) const;

private:
	char *login;
	int age;
	int year;
	float GPA;
};

void Student::display(void) const
{
	using std::cout;
	using std::endl;

	cout << "login: " << login << endl;
	cout << "  age: " << age << endl;
	cout << " year: " << year << endl;
	cout << "  GPA: " << GPA << endl;
}

int Student::get_age(void) const
{
	return age;
}

int Student::get_year(void) const
{
	return year;
}
```

```
float Student::get_GPA(void) const
{
	return GPA;
}

const char *Student::get_login(void) const
{
	return login;
}
```

Now, this works as expected:

```
void foo(void)
{
	// Create a constant object
	const Student john("jdoe", 20, 3, 3.10f);

	john.set_age(25); // Error, as expected.
	john.set_year(3); // Error, as expected.

	john.get_age();   // Ok, as expected.
	john.display();   // Ok, as expected.
}
```

***Note:** If a method does not modify the private data members, it should be marked as `const`. This will save you lots of time and headaches in the future. Unfortunately, the compiler won't remind you to do this until you try and use the method on a constant object.*

Separating the Interface from the Implementation

Typically, each class will reside in its own file. In fact, it will generally be in two files:

- The header file - contains the class defininition, usually in a `.h` file ***(Student.h in our case)***.
- The implementation file - contains all of the methods (in a `.cpp`) that are declared in the class file ***(Student.cpp in our case)***.

Default Constructors and Destructors

Recall one of the reasons for a constructor: "To insure that an object's data is not left undefined."

Recall one of the reasons for a destructor: "To insure that any resources (e.g. memory) created are released."

First, constructors:

Code

```
struct Point
{
	double x;
	double y;
};


int main(void)
{
	// Create a Point object
	// x/y are uninitialized
	Point pt1;

	// Display random values for x/y
	std::cout << pt1.x << "," << pt1.y << std::endl;

	system("pause");
	return 0;
}
```

Output (warning not treated as errors)

```
1.89121e-307,1.89121e-307
```

Output (warning treated as errors)

Severity	Code	Description
Error	C4700	uninitialized local variable 'pt1' used

Of course, the client could have initialized the data, but you can't count on that. The solution is to create a *default constructor*. A default constructor is simply a constructor that can be called without any arguments.

Add default constructor	Implementation
`struct Point` `{` `  // Default constructor` `  Point(void);` `  double x;` `  double y;` `};`	`Point::Point(void)` `{` `  // Initialize` `  x = 0;` `  y = 0;` `}`

Now, this object will be defined:

Code	`// Create a Point object` `// x/y are defined now` `Point pt1;` `// Display values for x/y` `std::cout << pt1.x << "," << pt1.y << std::endl;`
Output	`0,0`

It is not uncommon to provide other constructors in addition to a default constructor. (The example below uses the **class** keyword instead of **struct** just to demonstrate the technique works for both.)

Multiple constructors	Implementations
`class Point` `{` `public:` `  // Default constructor` `  Point(void);` `  // Constructor` `  Point(double x, double y);` `  // For convenience` `  void display(void) const;` `private:` `  double x;` `  double y;` `};`	`Point::Point(void)` `{` `  // Initialize to 0` `  x = 0.0;` `  y = 0.0;` `}` `Point::Point(double x, double y)` `{` `  // Initialize with params` `  x = x;` `  y = y;` `}`

```
void Point::display(void) const
{
  // Display random values for x/y
  std::cout << x << "," << y << std::endl;
}
```

Now the user can construct "default" objects or specify the values:

```
// Both are accepted
Point pt1;
Point pt2(3.5, 7);

pt1.display();  // 0,0
pt2.display();  // 3.5,7
```

Note that we can combine the default constructor into a non-default constructor by using default arguments:

```
class Point
{
public:
  // Default constructor
  Point(double x_ = 0.0, double y_ = 0.0);

  // Other stuff ...

};

Point::Point(double x, double y)
{
       // Initialize with params
       x = x_;
       y = y_;
}
```

Also, realize that this is now ambiguous:

```
// Two default constructors (illegal)
Point(void);
Point(double x_ = 0.0, double y_ = 0.0);

Point pt1; // Which one?
```

Notes:

- The only way to construct an object from a class/struct is with a constructor (either default on non-default).
- If you don't provide a constructor, the **compiler will provide a default constructor for you**.

- Here's what we get from the compiler (sort of) if we don't provide any constructors for the Point class:

```
Point::Point(void)
{
}
```

- Unfortunately, this compiler-provided default constructor doesn't do much.
- The compiler will provide a default constructor **ONLY** if you don't provide any constructors at all.
- Put another way, if you provide **ANY** constructors (default or otherwise) the compiler **WILL NOT** provide a default for you. Adding a default constructor to the `Student` class:

```
struct Student
{
public:
	// Default constructor
	Student(void);

	// Constructor
	Student(const char * login_, int age_,
		int year_, double GPA_);

	// Other public stuff ...

private:
	char *login;
	int age;
	int year;
	double GPA;
};

// Default constructor
Student::Student(void)
{
	login = 0;
	set_login("Noname");
	set_age(18);
	set_year(1);
	set_GPA(0.0);
}

// Constructor
Student::Student(const char * login_, int age_, int year_, double GPA_)
{
	login = 0;
	set_login(login);
	set_age(age);
	set_year(year);
	set_GPA(GPA);
}
```

Of course, it's up to you (the class implementer) to decide if something has a "sane" default or must be provided by the user.

Recall one of the reasons for a destructor: "To insure that any resources (memory) created are released."

Just like constructors, the compiler will provide a destructor for us if we fail to do so. Here's what the compiler will generate (sort of) if we don't provide a destructor for the `Point` class or the `Student` class:

```
Point::~Point()
{
}

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

- Like the compiler-generated constructor, these destructors don't do anything useful.
- For the `Point` class, this is sufficient since there is nothing to "clean up".
- The `Student` class is different because it dynamically allocates memory for the `login_` member:

```
void Student::set_login(const char* login_)
{
	delete[] login;

	int len = std::strlen(login_);
	login = new char[len + 1]; // Dynamic memory allocation
	std::strcpy(login, login_);
}
```

We need a better destructor, which we created:

```
Student::~Student(void)
{
	// Since our class allocated the memory,
	// our class must release the memory.
	delete[] login;
}
```

Example using constructors, destructors, static allocation, and dynamic allocation:

Code

```
class Point
{
public:
	// Default constructor
	Point(double x_ = 0.0, double y_ = 0.0);

	// Destructor
	~Point(void);

	// For convenience
	void display(void) const;

private:
	double x;
	double y;
};

Point::~Point(void)
{
	std::cout << "Point destructor: " << x << "," << y <<
std::endl;
}

Point::Point(double x_, double y_)
{
	// Initialize with params
	x = x_;
	y = y_;
	std::cout << "Point constructor: " << x << "," << y <<
std::endl;
}

void foo(void)
{
	// Static allocation
	Point pt1;	// 0,0
	Point pt2(4);	// 4,0
	Point pt3(4, 5); // 4,5

                                        // Similar using dynamic
allocation
	Point *pt4 = new Point;
	Point *pt5 = new Point(8);
	Point *pt6 = new Point(7, 9);

	// Must delete manually (calls destructor)
	delete pt4;
	delete pt5;
	delete pt6;

}  // Destructors for pt1,pt2,pt3 called here
```

Output

```
Point constructor: 0,0
Point constructor: 4,0
Point constructor: 4,5
Point constructor: 0,0
Point constructor: 8,0
Point constructor: 7,9
Point destructor: 0,0
Point destructor: 8,0
Point destructor: 7,9
Point destructor: 4,5
Point destructor: 4,0
Point destructor: 0,0
```

Copy Constructor

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

Code

```
struct Point
{
public:
	// Constructor
	Point(float x_, float y_);

	// Copy Constructor
	Point(const Point &p_);

	void display(void);

	float x, y;
};

Point::Point(float x_, float y_)
{
	std::cout << "Constructor Called" << std::endl;
	x = x_;
	y = y_;
}

Point::Point(const Point &p_)
{
	std::cout << "Copy Constructor Called" << std::endl;
	x = p_.x;
	y = p_.y;
}

void Point::display(void)
{
	std::cout << "x = " << x << std::endl;
	std::cout << "y = " << y << std::endl;
}
```

```
int main(void)
{
	Point p1(10.0f, 20.0f);
	p1.display();

	std::cout << std::endl;

	Point p2(p1);
	p2.display();

	return 0;
}
```

Output

```
Constructor Called
x = 10
y = 20

Copy Constructor Called
x = 10
y = 20
```

If a **copy constructor** is not defined in a class, the compiler itself defines a default one. The compiler provided copy constructor will just respectively assign the properties values of the newly created instance to the properties values of the passed reference.

Code

```
struct Point
{
public:
	// Constructor
	Point(float x_, float y_);

	void display(void);

	float x, y;
};

// Implementation of the Point class methods ....

int main(void)
{
	Point p1(10.0f, 20.0f);
	p1.display();

	std::cout << std::endl;

	Point p2(p1);
	p2.display();

	return 0;
}
```

Output	

```
Constructor Called
x = 10
y = 20

x = 10
y = 20
```

The **copy constructor** is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

Code

```
struct Point
{
public:
	// Constructor
	Point(float x_, float y_);

	// Copy Constructor
	Point(const Point &p_);

	void display(void);

	float x, y;
};

Point::Point(float x_, float y_)
{
	std::cout << "Constructor Called" << std::endl;
	x = x_;
	y = y_;
}

Point::Point(const Point &p_)
{
	std::cout << "Copy Constructor Called" << std::endl;
	x = p_.x;
	y = p_.y;
}

void Point::display(void)
{
	std::cout << "x = " << x << std::endl;
	std::cout << "y = " << y << std::endl;
}


void FunctionThatTakesAPoint(Point p_)
{
	p_.display();
}
```

```
Point FunctionThatReturnsAPoint(void)
{
	Point p(10.0f, 15.0f);

	std::cout << "Returning a point" << std::endl;
	return p;
}

int main(void)
{
	Point p1(10.0f, 20.0f);
	p1.display();
	std::cout << std::endl;

	Point p2(p1);
	p2.display();
	std::cout << std::endl;

	std::cout << "Sending a point class by value" << std::endl;
	FunctionThatTakesAPoint(p1);
	std::cout << std::endl;

	FunctionThatReturnsAPoint();

	return 0;
}
```

Output

```
Constructor Called
x = 10
y = 20

Copy Constructor Called
x = 10
y = 20

Sending a point class by value
Copy Constructor Called
x = 10
y = 20

Constructor Called
Returning a point
Copy Constructor Called
```

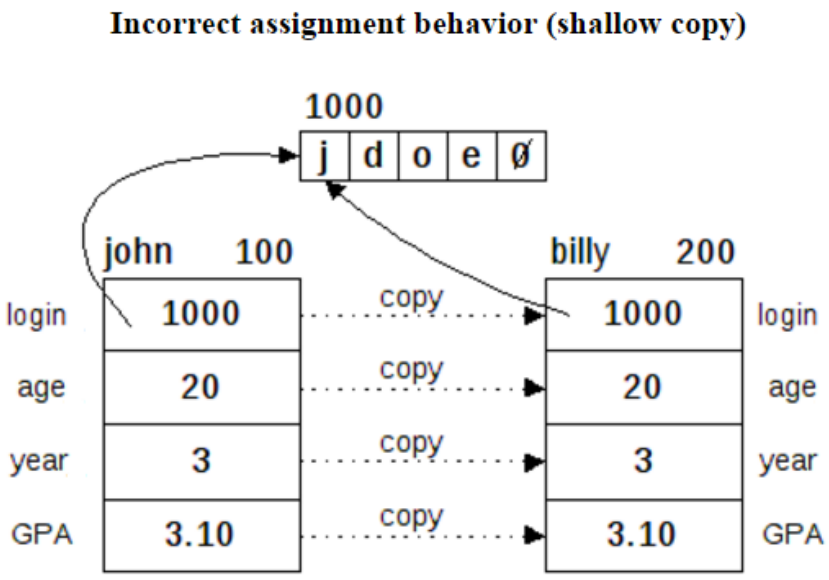
If the class has **pointer variables** and has some **dynamic memory allocations**, then it is a **MUST** to have a copy constructor.

Code with Problem!

```
struct Student
{
public:
	// Constructor
	Student(const char * login_, int age_,
		int year_, double GPA_);

	void display(void);

private:
	char *login; //dynamically allocated
	int age;
	int year;
	double GPA;
};

Student::Student(const char * login_, int age_,
	int year_, double GPA_)
{
	age = age_;
	year = year_;
	GPA = GPA_;

	int length = strlen(login_) + 1; //add 1 for the null
character
	login = new char[length];
	strcpy_s(login, length, login_);
}

void Student::display(void)
{
	std::cout << "Login: " << login << std::endl;
	std::cout << "Age: " << age << std::endl;
	std::cout << "Year: " << year << std::endl;
	std::cout << "GPA: " << GPA << std::endl;
}

int main(void)
{
	Student john("jdoe", 20, 3, 3.10);
	john.display();

	std::cout << std::endl;

	Student billy(john);
	billy.display();


	return 0;
}
```

Tricky Output	Login: John Age: 20 Year: 3 GPA: 3.1 Login: John Age: 20 Year: 3 GPA: 3.1

Looking at the output, everything seems to be fine but actually we have a big problem:

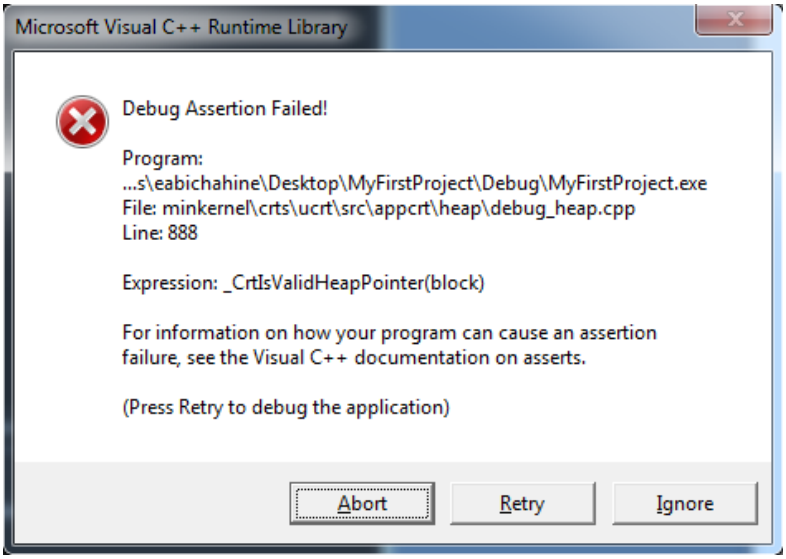


Since the default copy constructor blindly assigns billy's properties to john's properties, both login's are pointing to the same memory. That means, if one of the instances changes the login data or deletes it, that data is changed or deleted for both instances!

If you add a correct destructor to the Student class, the above code will lead to a crash.

Destructor Code	``` Student::~Student(void) { delete[] login; } ```

When john goes out of scope, its destructor will be called and will free the memory allocated for its login property. Since that memory is also shared with billy's login property, when billy goes out of scope its destructor will also be called and will attempt to free the same memory which will lead to the following crash:

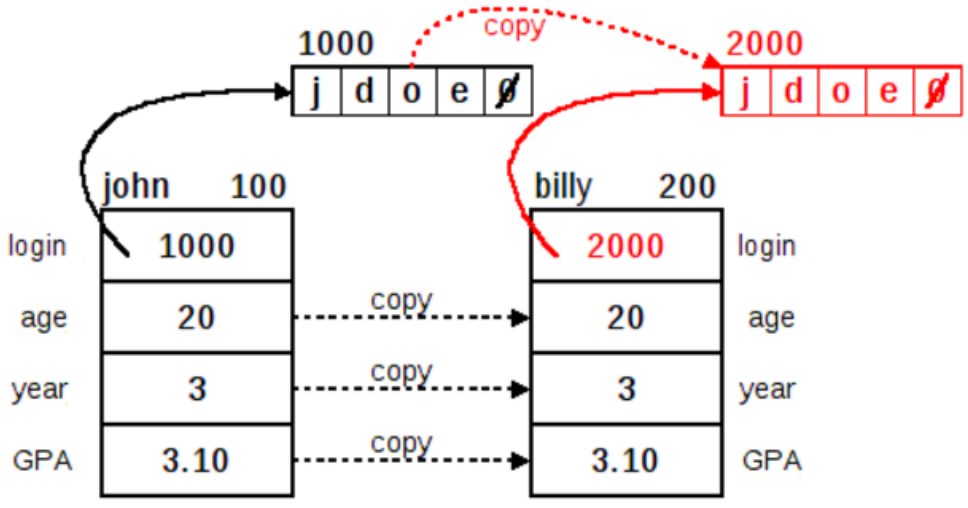


The solution is to implement a correct copy constructor for our student class:

Code

```
Student::Student(const Student &s_)
{
	age = s_.age;
	year = s_.year;
	GPA = s_.GPA;

	int length = strlen(s_.login) + 1;
	login = new char[length];
	strcpy_s(login, length, s_.login);
}
```

Correct assignment behavior (deep copy)



Conversion Constructor

A constructor declared without the *function-specifier* ***explicit*** that can be called with a single parameter specifies a conversion from the type of its first parameter to the type of its class. Such a constructor is called a conversion constructor (or a converting constructor).

Example

```
#include <iostream>

class Foo
{
public:
	Foo(int value_);

	int value;
};

Foo::Foo(int value_)
{
	std::cout << "Conversion constructor called" << std::endl;
	value = value_;
}

int main(void)
{
	Foo f = 5;
	std::cout << "The value in f is: " << f.value << std::endl;

	return 0;
}
```

Output

```
Conversion constructor called
The value in f is: 5
```

Beware though, with great power comes great responsibility! The compiler is implicitly calling the conversion constructor without informing you, sometimes this will lead to messy code.

If you don't want the compiler to implicitly call the conversion constructor, you should specify that by adding the keyword ***explicit*** next to your constructor.

Example

```
#include <iostream>

class Foo
{
public:
	explicit Foo(int value_);

	int value;
};
```

```
Foo::Foo(int value_)
{
	std::cout << "Conversion constructor called" << std::endl;
	value = value_;
}

int main(void)
{
	Foo f = 5;
	std::cout << "The value in f is :" << f.value << std::endl;

	return 0;
}
```

Output

Severity	Description
Error	'initializing': cannot convert from 'int' to 'Foo'

Arrays of Objects

Just like any other type, you can create arrays of objects.

- You can initialize the array with an initializer list.
- If the size of the array is larger than the number of initializers, the compiler will initialize the remaining elements by calling the default constructor.
- This means that unless you initialize each element individually, the class must have a default constructor.
- If a class does not have a default constructor, you cannot create an array without initializers.

Built-in types:

```
// Compiler initializers elements that you don't
int a[3] = { 1, 2, 3 }; // 1, 2, 3
int b[3] = { 1, 2 };    // 1, 2, 0
int c[3] = { 0 };        // 0, 0, 0
double d[3] = { 1.0 };   // 1.0, 0.0, 0.0
```

User-defined types:

Code	`// Requires default constructor` `Student st1[2];` `for (int i = 0; i < 2; i++)` `{` `  st1[i].display();` `}`
Output	**login: Noname** **age: 18** **year: 1** **GPA: 0** **login: Noname** **age: 18** **year: 1** **GPA: 0**

PS: If no default constructor is provided in the Student class the above code will lead to a compiler error! The array is calling the default constructor of every element its creating.

Code

```
// Requires default constructor
Student st2[3] = { Student("jdoe", 20, 3, 3.10F),
                   Student("jsmith", 19, 2, 3.95F)
                 };

for (int i = 0; i < 3; i++)
{
       st2[i].display();
}
```

Output

```
login: jdoe
  age: 20
 year: 3
  GPA: 3.1
login: jsmith
  age: 19
 year: 2
  GPA: 3.95
login: Noname
  age: 18
 year: 1
  GPA: 0
```

Code

```
// No default constructor required
Student st3[2] = { Student("jdoe", 20, 3, 3.10F),
                   Student("jsmith", 19, 2, 3.95F)
                 };

for (int i = 0; i < 2; i++)
{
       st3[i].display();
}
```

Output

```
login: jdoe
  age: 20
 year: 3
  GPA: 3.1
login: jsmith
  age: 19
 year: 2
  GPA: 3.95
```

If the `Student` class does not have a default constructor, then the first two examples above would cause a compiler error.

Note: Arrays of objects and conversion constructors can get tricky.

Code

```
#include <iostream>

class Foo
{
public:
	Foo(int value_);

	int value;
};

Foo::Foo(int value_)
{
	std::cout << "Conversion constructor called" << std::endl;
	value = value_;
}

int main(void)
{
	Foo A[3] = { 1, 2, 3 };

	for (int i = 0; i < 3; ++i)
	{
		std::cout << A[i].value << " ";
	}

	std::cout << std::endl;


	return 0;
}
```

Output

```
Conversion constructor called
Conversion constructor called
Conversion constructor called
1 2 3
```