Introduction to Classes

The material in this handout is collected from the following reference:

- Chapter 7 of the text book C++ Primer.
- Chapter 9 of Programming Principles and Practice Using C++.

Abstract data types

We live in a complex world. Throughout the course of each day, we are constantly bombarded with information, facts, and details. To cope with complexity, the human mind engages in *abstraction* - the act of separating the essential qualities of an idea or object from the details of how it works or is composed.

With abstraction, we focus on the *what*, not the *how*. For example, our understanding of automobiles is largely based on abstraction. Most of us know *what* the engine does [it propels the car], but fewer of us know - or want to know - precisely *how* the engine works internally. Abstraction allows us to discuss, think about, and use automobiles without having to know everything about how they work.

In the world of software development, abstraction is an absolute necessity for managing immense, complex software projects. In introductory computer science courses, programs are usually small (perhaps 50 to 100 lines of code) and understandable in their entirety by one person. By comparison, large commercial software products composed of hundreds of thousands - even millions - of lines of code cannot be designed, understood, or tested thoroughly without using abstraction in various forms. To manage complexity, software developers regularly use two important abstraction techniques: functions and data abstraction.

Functions are a form of *control abstraction* that separates the logical properties of an action from its implementation. We engage in control abstraction whenever we write a function that reduces a complicated algorithm to an abstract action performed by a function call. By invoking a function, as in the expression

```
4.6 + std::sqrt(x)
```

we depend only on the function's *specification* - the written description of what it does. We can use the function without having to know its *implementation* [the algorithms and the code that accomplish the result]. In this case, by invoking function `sqrt`, the program is less complex because all the details involved in computing square roots remain hidden.

Abstraction techniques also apply to data. Every data type consists of a set of values along with a collection of allowable operations on those values. *Data abstraction* is the separation of a data type's logical properties from its implementation details. The C++ language provides some built-in types such as `char`, `int`, and `double`. A type is called *built-in* if the compiler knows how to represent objects of the type and which operations can be done on them [such as `+` and `*`]. Data abstraction comes into play when we need a data type that is not built into the programming language. We then define the new type as an *abstract data type*, concentrating only on its logical properties and deferring the details of its implementation.

> ***Abstract data type is a data type whose properties [domain and operations] are specified independently of any particular implementation.***

As with functions, an abstract data type has both a specification [the *what*] and an implementation [the *how*]. The specification of an ADT describes the characteristics of the data values as well as the behavior of each of the operations on those values. The ADT user needs to understand only the specification, not the implementation, to use it.

Here's a very informal specification of an `int_list` ADT that contains `int` values:

TYPE
int_list

DOMAIN
Each *int_list* value is a collection of up to 100 separate *int values*.

OPERATIONS
Create a list
Insert an item into the list
Delete an item from the list
Search for an item in the list
Return the current length of the list
Sort the list into ascending order
Print the list

Notice the complete absence of implementation details. No mention is made of how the data might actually be stored in a program [it could be in an array, or on a file, or in a `struct`] or how the operations might be implemented [for example, the length operation could keep the number of elements in a variable, or it could count the elements each time it is called]. Concealing the implementation details reduces complexity for the user and also shields the user from changes in the implementation.

Why do we build new types? The compiler doesn't know all the types we might like to use in our programs. It couldn't, because there are far too many useful types - no language designer or compiler implementer could know them all. We invent new ones every day. Why? What are types good for? Types are good for directly representing ideas in code. When we write code, the ideal is to represent our ideas directly in our code so that we, our teammates, and the compiler can understand what we wrote. When we want to do integer arithmetic, `int` is a great help; when we want to manipulate text, `string` is a great help; when we want to many `string`s, `vector` is a great help, when we want to program a clone of Battleship, `ocean` and `boat` are a great help. The help comes in two forms:

- *Representation*: A type *knows* how to represent the data needed in an object.
- *Operations*: A type *knows* what operations can be applied to objects.

Many ideas follow this pattern: "something" has data to represent its current value - sometimes called the current *state* - and a set of operations that can be applied. Think of a computer file, a web page, a toaster, an alarm clock, a media player, a coffee cup, a car engine, a cell phone; all can be characterized by some data and all have a more or less fixed set of standard operations that you perform. In each case, the result of the operation depends on the data - the "current state" - of an object.

The specification of an ADT defines data values and operations in an abstract form that can be given as documentation to both the user and the programmer. Ultimately, of course, the ADT must be implemented in program code. To implement an ADT, the programmer must do two things:

1. Choose a concrete *data representation* of the abstract data, using built-in and other user-defined data types that already exist.
2. Implement each of the allowable operations in terms of functions.

To implement the `int_list` ADT, we could choose a concrete data representation consisting of a structure with two items: a 100-element array in which to store the items and an `int` variable that keeps track of the number of items in the list. To implement the `int_list` operations, we must create algorithms based on the chosen data representation and encapsulate them in functions. The specification of the ADT does not confine us to any particular data representation. As long as we satisfy the specification, we are free to choose among the various alternative data representations and their associated algorithms. Our choice may be based on time efficiency [the speed at which the algorithms execute], space efficiency [the economical use of memory space], or simplicity and readability of the algorithms. Over time, you will acquire knowledge and experience that will help you decide which implementation is best for a particular context.

Class

In HLP1 and so far in HLP2, we've treated data values as passive quantities to be acted upon by functions. That is, we've passed values to and from functions as arguments, and our functions performed some operations. We took the same approach even when we defined new data types using keyword `struct`. For example, in the Battleship exercise, an ocean and boats were viewed as passive data by passing them to functions to be processed.

This separation of operations and data does not correspond very well with the notion of an abstract data type. After all, an ADT consists of *both* data values and operations on those values. It is preferable to view an ADT as defining an *active* data structure - one that contains both data and operations within a single, cohesive unit. C++ supports this view through the user-defined type known as a `class`.

A class is composed of built-in types, other user-defined types, and functions. The parts used to define the class are called *members*. A class has zero or more members. For example:

```
class X {
public:
  int m{};                                          // data member
  int mf(int v) { int old {m}; m = v; return old; } // member function
};
```

We've defined a new type called `X` and as with any other type, we can define a variable [or object] of type `X` and access members using the `object.member` notation:

```
X var{10}, obj; // var and obj are variables of type X
                // just as with structs, data members can be initialized
obj.m = var.mf(5); // assign to obj's data member m the value returned
                   // by a call to var's member function mf
std::cout << "var.m: " << var.m << " | obj.m: " << obj.m << "\n";
```

Interface and implementation

The point of using a class is to provide data abstraction to a type that consists of an *interface* and an *implementation*. The interface is the part of the class's declaration that its clients access directly. The implementation is that part of the class's declaration that its clients access only indirectly through the interface. The public interface is identified by access label `public:` and the implementation by access label `private:` [in C/C++, a label consists of an identifier and the `:` character]. You can think of a class declaration like this:

```
class X {
public:
  // public members - the interface accessible to everybody:
  // functions
  // types
  // data [best kept private to ensure adherence to data abstraction or
  // separation of interface and implementation or data hiding]
private:
  // private members - implementation details only accessible
  // to members of this class:
  // functions
  // types
  // data
};
```

What happens if access labels are missing? Class members are private by default - therefore

```
class X {
  int mf(int);
  // ...
};
```

means

```
class X {
private:
  int mf(int);
  // ...
};
```

so that

```
X x; // variable (or object) of type X
// error: mf is private and therefore inaccessible to clients
int y {x.mf(10)}; // but is accessible to other member functions of
```

A client cannot directly refer to a private member. Instead, clients go through a public function that can access that private member. For example:

```
class X {
  int m{};
  int mf(int v) {int old = m; m = v; return old; }
public:
  int f(int k) { return mf(k); }
};

X x;
int y {x.f(11)}; // y is initialized with 11
```

> ***Remember that we use private and public to represent the important distinction between an interface [the client's view of the class] and implementation details [the implementer's view of the class].***

There is a useful simplified notation for a class that has no private implementation details. A `struct` is a `class` where members are public by default:

```
struct X {
  int m;
  // ...
};
```

means

```
class X {
public:
  int m;
  // ...
}
```

`struct`s are primarily used for data structures that are too simple to need hiding of implementation details. For example:

```
struct Point {
  double x, y;
  // member functions ...
};
```

Evolving a class

Let's illustrate the language facilities supporting classes and the basic techniques for using them by showing how and why - we might evolve a simple C-style data structure defined with keyword `struct` into a ADT that is defined with keyword `class` with private implementation details and supporting operations. Throughout this process, we use a structure that represents you - a student.

`struct` and functions

How to represent a student? We use built-in types and an array:

```
struct Student {
  enum {MAXLENGTH = 10};
  char login[MAXLENGTH];
  int age, year;
  double gpa;
};

Student s1, s2;
```

So, now we've `Student`s; what can we do with them? We can do everything - in the sense that we can access the members of `s1` and `s2` and read and write them as we like. The snag is that nothing is really convenient. Just about anything we want to do with a `Student` has to be written in terms of reads and writes of those members. For example:

```
s1.age = 20;
s1.gpa = 3.8;
s1::strcpy(s1.login, "jdoe");
s1.year = 3;
```

This is tedious. Everything that's tedious is error prone! For example, does this make sense?

```
s2.age = -5;
s2.gpa = 12.9;
std::strcpy(s2.login, "rumplestiltzkin");
s2.year = 150;
```

What we do then is to provide some helper functions to do the most common operations. That way, we don't have to repeat the same code over and over again and we won't make, and then find, and fix the same mistakes over and over again. For just about every type, initialization and assignment are among the most common operations:

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

void set_year(Student& student, int y) {
  if ( (y < 1) || (y > 4) ) {
    std::cerr << "Error in year range!\n";
    student.year = 1;
  } else { student.year = y; }
}
void set_age(Student &student, int a) {
  if ( (a < 18) || (a > 100) ) {
    std::cout << "Error in age range!\n";
```

```
    student.age = 18;
  } else { student.age = a; }
}

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

void display(Student const& student) {
  std::cout << "login: " << student.login << "\n";
  std::cout << "  age: " << student.age << "\n";
  std::cout << " year: " << student.year << "\n";
  std::cout << "  GPA: " << student.gpa << "\n";
}
```

Now, this code:

```
Student s3;
set_age(s3, -5);
set_gpa(s3, 12.9);
set_login(s3, "rumplestiltzkin");
set_year(s3, 150);
display(s3);
```

will result in this:

```
Error in age range!
Error in GPA range!
Error in year range!
login: rumplesti
  age: 18
 year: 1
  GPA: 0
```

Member functions and initialization

We provided initialization functions for `Student`s; they're not perfect, but they at least ensure valid values for the implementation and prevent memory corruption. However, such functions are of little use if we fail to use them. For example:

```
Student s4;
display(s4);
set_age(s4, -5);
set_gpa(s4, 12.9);
set_login(s4, "rumplestiltzkin");
set_year(s4, 150);
```

Here, we "forgot" to immediately initialize `s4` and "someone" used it before we got around to calling `set` functions. This separation of operations and data does not correspond very well with the notion of an abstract data type. After all, an ADT consists of *both* data values and operations on those values. After all, an ADT is an *active* data structure - one that contains both data and operations within a single, cohesive unit. The basic tool for that in C++ is *member functions*, that is, functions declared as members of the structure within the structure body. Adding functions to the structure is simple. By declaring them after a `public:` label, the functions will be accessible from outside of the structure. We've added a new function `init`. Member functions can be defined right in the structure definition, as with the new function `init` or outside the structure definition:

```
struct Student {
  void init(char const *log, int a, int y, double g) {
    set_login(log); set_age(a); set_year(y); set_gpa(g);
  }
  void set_login(char const *log);
  void set_age(int a);
  void set_year(int y);
  void set_gpa(double g);
  void display();

  enum {MAXLENGTH = 10};
  char login[MAXLENGTH];
  int age, year;
  double gpa;
};
```

When we define a member outside its structure, we need to say which structure it is a member of. We do that using the `structure_name::member_name` notation:

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

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

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

void Student::set_gpa(double g) {
```

```
  if ( (g < 0.0) || (g > 4.0) ) {
    std::cout << "Error in GPA range!\n";
    gpa = 0.0;
  } else { gpa = g; };
}

void Student::display() {
  std::cout << "login: " << login << "\n";
  std::cout << "  age: " << age << "\n";
  std::cout << " year: " << year << "\n";
  std::cout << "  GPA: " << gpa << "\n";
}
```

A member function, such as `Student::set_login` doesn't need to use the `s.login` notation where `s` is a variable of type `Student`. It can use the plain member name [`login` in this example]. Within a member function, a member name refers to the member of that name in the object for which the member function was called. Thus, in the call `s.set_login("jdoe")`, the `login` in the definition of `Student::set_login` refers to `s.login`.

Note that member function `init` can refer to `set_login`, `set_age`, and other names declared after `init`. A member can refer to a member function or data member of its structure independently of where in the structure that other member is declared.

> ***The rule that a name must be declared before it is used is relaxed within the limited scope of a structure.***

Writing the definition of a member function within the structure definition has three effects:

- The function will be *inline*; that is, the compiler will try to generate code for the function at each point of call rather than using function-call mechanism. This can be a significant advantage for functions that hardly do anything but are used a lot.
- All uses of the structure will have to be recompiled whenever we make a change to the body of an inlined function. If the function body is out of the structure declaration, recompilation of clients' code is needed only when the class declaration is itself changed. Not recompiling when the body is changed can be a huge advantage in large programs.
- The structure definition gets larger. Consequently, it can be harder to find the members among the member function definitions.

> ***Don't put member function bodies in structure definition unless you know that you need the performance boost from inlining tiny functions. Large functions, say five or more lines of code, don't benefit from inlining and make a structure declaration harder to read. We rarely inline a function that consists of more than one or two expressions.***

Incidentally, to keep C++ `struct`s compatible with C `struct`s, they've a default access label `public:`. Therefore, the following structure definition

```
struct Student {
  enum {MAXLENGTH = 10};
  char login[MAXLENGTH];
  int age, year;
  // ...
};
```

is identical to this structure definition:

```
struct Student {
public:
  enum {MAXLENGTH = 10};
  char login[MAXLENGTH];
  int age, year;
  // ...
};
```

Now, this is how we use the member functions of `Student`:

```
void f() {
  Student s; // define but not initialize a Student object
  s.set_login("jdoe"); // using the public member functions
  s.set_age(22);
  s.set_year(4);
  s.set_gpa(3.8);
  s.display(); // tell the object to display itself
}

void g() {
  Student s;
  s.init("jdoe", 22, 4, 3.8); // call set_ functions
  s.display(); // tell the object to display itself
}
```

Keep details private

We still have a problem: What if someone forgets to use member function `set_login`? What if someone decides to change a student's age directly?

```
Student s;
std::strcpy(s.login, "rumplestiltzkin");
s.age = -5;
s.set_year(2);
s.set_gpa(4.0);
```

As long as we've the representation of `Student` accessible to everybody, somebody will - by accident or design - mess it up; that is, someone will do something that produces an invalid value. In this case, we created an ill-defined `Student` variable `s` with an ill-defined name and an invalid age. Such invalid objects are time bombs; it is just a matter of time before someone innocently uses the invalid value and gets a runtime error or - usually worse - produces a bad result.

Such concerns lead us to conclude that the representation of `Student` should be inaccessible to users except through the public member functions that we supply.

> ***Each class or structure member has a specified accessibility provided through labels. C++ provides three of them:***
>
> - `public:` ***declarations are accessible from everywhere;***
> - `private:` ***- declarations are accessible only within the class or structure; and***
> - `protected:` ***- declarations are accessible in the class or structure itself and its derived classes or structures.***

Here is a first cut:

```
struct Student {
public:
  void set_login(char const *log);
  void set_age(int a);
  void set_year(int y);
  void set_gpa(double g);
  void init(char const *log, int a, int y, double g);
  void display();
private:
  enum {MAXLENGTH = 10};
  char login[MAXLENGTH];
  int age, year;
  double gpa;
};
```

Programmers put the public interface first because the interface is what most people are interested in. The compiler doesn't care about the order of member functions and data members; it takes the declarations in any order you care to present them.

Constructors

We've now ensured that it is not possible for clients to directly access data members of a `Student` variable:

```
Student s0 {"jdoe", 21, 3, 3.8}; // error: data members are private
Student s1;
std::strcpy(s1.login, "jdoe"); // error: login is private data member
s1.age = 20;                   // error: age is private data member
s1.year = 3;                   // error: year is private data member
s1.gpa = 3.8;                  // error: gpa is private data member
```

Clients are now correctly forced to use `init` or `set` member functions to assign *valid* values to variables of type `Student`:

```
Student s0; // define a Student object
s0.set_login("jdoe"); // using the public member functions
s0.set_age(22);
s0.set_year(4);
s0.set_gpa(3.8);
s0.display(); // tell the object to display itself

Student s1;
s1.init("jdoe", 22, 4, 3.8); // call set_ functions
s1.display(); // tell the object to display itself
```

However, we're still unable to *force* a `Student` object to be initialized when it is first created. This can lead to someone forgetting to immediately initialize a `Student` object before using it:

```
Student s;   // uninitialized student
s.display(); // forgot to call s.init_student()!!!
```

These sorts of errors leads to a demand for an initialization function that can't be forgotten by clients. The language provides a special member function with the same name as its structure. It is called *constructor* and will be used for initialization ["construction"] of objects of the structure. Here is the second cut with the addition of a constructor and the removal of member function `init` [which is no longer required since we're defining a constructor]:

```
struct Student {
public:
  Student(char const *log, int a, int y, double g); // constructor
  void set_login(char const *log);
  void set_age(int a);
  void set_year(int y);
  void set_gpa(double g);
  //void init(char const *log, int a, int y, double g);
  void display();
private:
  enum {MAXLENGTH = 10};
  char login[MAXLENGTH];
  int age, year;
  double gpa;
};
```

The constructor is easily implemented by simply calling member functions that will each set a private data member:

```
Student::Student(char const *log, int a, int y, double g) {
  set_login(log); set_age(a); set_year(y); set_gpa(g);
}
```

It is an error - caught by the compiler - to forget to initialize an object of a structure with a constructor requiring argument(s), and there is a special convenient syntax for doing such initialization:

```
Student s0;                              // error: s0 is not initialized
Student s1("jdoe", 22, 4, 3.8);          // OK - C++98 style
Student s2 = Student("jdoe", 22, 3, 3.6); // Ok - verbose C++98 style
Student s3 {"jdoe", 21, 3, 3.6};         // OK - colloquial style
Student s4 = {"jdoe", 20, 2, 3.4};       // OK - verbose style
Student s5 = Student{"jdoe", 21, 3, 3.4}; // OK - verbose style
```

In C++98, people used parentheses to delimit the initializer list, so you'll see a lot of code like this:

```
Student s1("jdoe", 22, 4, 3.8);
```

We prefer `{}` for initializer lists because that clearly indicates when initialization [construction] is done, and also because that notation is more widely used. The notation can also be used for built-in types. For example:

```
int x {7}; // OK (modern initializer list style)
```

Optionally, we can use a `=` before the `{}` list:

```
Student s2 = {"jdoe", 20, 3, 3.6};
```

Some find this combination of older and new style more readable.

To conclude: The constructor solves the initialization problem with clients never able to define a `Student` variable without also initializing it. This means that every `Student` variable will have a valid state. This is a very important and powerful feature because the alternative is for us to check for validity every time we use an object, or just hope that nobody left an invalid value lying around. Experience shows that "hoping" can lead to programs that occasionally produce erroneous results and occasionally crash. With constructors, we can write code that can be demonstrated to be correct.

Replacing `struct` with `class`

A `class` is similar to a `struct` with one exception: the default accessibility label of members in a `class` is `private:`. Here is the third cut of type `Student` that replaces keyword `struct` with keyword `class`:

```
class Student {
public:
  Student(char const *log, int a, int y, double g); // constructor
  void set_login(char const *log);
  void set_age(int a);
  void set_year(int y);
  void set_gpa(double g);
  void display();
private:
  char login[MAXLENGTH];
  int age, year;
  double gpa;
};
```

When should one use `class` and when should one use `struct` if the difference between them only relates to default accessibility of members? The convention is to use `class` if the notion of a "valid `Student`" is important. We tried to design type `Student` so that values are guaranteed to be valid; that is, we hid the representation of `Student`, provided a constructor that creates only valid objects, and designed all member functions to leave only valid values behind when they return. A rule for what constitutes a valid value is called an *invariant*. If we can't think of a good valid invariant, we're probably dealing with plain data. If so, use a `struct`. This is a good example of a `struct`:

```
struct Point {
  double x, y; // public data members accessible to all
};
```

We can specify constructors to initialize `Point` objects and member functions to facilitate various mathematical operations. Yet a point is such a low-level and commonly used type that it is better if the data is part of the interface. Otherwise, clients will have to access `x` and `y` data members using a clumsy interface.

Categories of member functions

Since data members in a class ~~are usually~~ ***must be*** defined after access label `private:`, clients can manipulate private data only if they're provided an interface in the form of member functions declared after access label `public:`. We can categorize member functions in a class into the following list:

- *Constructor*: A member function that is automatically called to initialize a new instance or variable or object of a class.
- *Destructor*: A member function that is automatically called when an instance or variable or object of a class goes out of scope. We've not implemented a destructor yet.
- *Accessor* or *Observer*: A member function that allows clients to observe or read the [private] state of an object without changing it. Also colloquially called *Getter*.
- *Mutator* or *Transformer*: A member function that allows clients to mutate or change or transform the [private] state of an object. Also colloquially called *Setter*.

To define an ADT, a class definition must specify one or more constructors to ensure a valid state for an object. If constructors have allocated resources from the run-time environment [such as heap memory for dynamically allocated objects], then a [one and only one] destructor must be defined. The purpose of the destructor is to return resources that were allocated to the object during its lifetime. A class may provide one, both, or none of accessor or mutator functions. If only an accessor function is provided for a private data member, the encapsulated data is *read-only*. If only a mutator is provided for a private data member, the data is considered *write-only*. If both accessor and mutator functions are defined, the data is considered *read-write*.

All of the data in class `Student` is write-only; we can change it, but we can't read it. Let's make class `Student` more useful by adding accessors to the definition of class `Student`:

```
1  class Student {
2  public:
3    Student(char const *log, int a, int y, double g); // ctor
4
5    // accessors or observers or gettors
6    int get_age()           { return age;}
7    int get_year()          { return year; }
8    double get_gpa()        { return gpa; }
9    char const* get_login() { return login; }
10
11   // mutators or transformers or settors
12   void set_login(char const *log);
13   void set_age(int a);
14   void set_year(int y);
15   void set_gpa(double g);
16
17   void display();
18
19 private:
20   enum {MAXLENGTH = 10};
21   char login[MAXLENGTH];
22   int age, int year;
23   double gpa;
24 };
```

You control access and modifications to private data by providing or not providing accessors and mutators to individual data members. If you decide a student's login is set in stone during initialization and can never be altered, then remove the `set_login` mutator from the public interface of class `Student`.

Resource management

This is how far type `Student` has evolved:

- Every `Student` object is initialized to a valid state.
- Once initialized to a valid state, the public interface prevents clients from altering the valid state to an invalid state.
- A student's login name is limited to 9 characters; member function `set_login` ensures that names with more than 9 characters are safely truncated. This situation is safe but may not be ideal since students' might wish to have longer login names.

Let's fix the limitation of a student's login name by determining the length at runtime. First, we update the definition of class `Student` so that data member `login` is no longer a static array:

```
class Student {
public:
  // public interface is same as before except set_login ...
private:
  char *login;
  int age, year;
  double gpa;
};
```

This necessitates a change to member function `set_login` that requires dynamically allocating memory from the free store:

```
void Student::set_login(char const *log) {
  login = new char[std::strlen(log) + 1];
  std::strcpy(login, log);
}
```

This is the advantage of data abstraction and ADTs: the ability to change the implementation without clients' having to update their code. Consider the following function:

```
void foo() {
  Student john{"jrumplestiltzkin", 20, 3, 3.10}; // construct Student object
  john.display();                                // display all of the data
}
```

With the earlier version that used static array for data member `login`, `john`'s encapsulated data would be displayed as:

```
login: jrumplest
  age: 20
 year: 3
  GPA: 3.1
```

while the new version would display the following information:

```
login: jrumplestiltzkin
  age: 20
 year: 3
  GPA: 3.1
```

The call to operator `new[]` in member function `set_login` results in allocation of free store memory for every instance of `Student`. To avoid memory leaks, we should augment the public interface of class `Student` with the declaration of a function that clients can call to return the previously allocated memory to the free store:

```
class Student {
public:
  // public interface - as before ...
  void free_login();
private:
  // private implementation - as before ...
};
```

and provide a definition like this:

```
void Student::free_login() {
  delete [] login;
}
```

We hope that clients will do the right thing [of avoiding a memory leak] by making a call to `free_login` before the instance of `Student` goes out of scope:

```
void foo() {
  Student john{"jrumplestiltzkin", 20, 3, 3.10}; // construct Student object
  john.display();                                // display all of the data
  john.free_login();                             // return free memory
}
```

This update to the class interface is wrong on so many levels. Let's list a few of them. First, it possible that the client may entirely forget to call `free_login`:

```
void foo() {
  Student john {"jdoe", 20, 3, 3.10}; // construct a Student object
  john.display();                     // display data
  // Oops, memory leak now!
}
```

Second, the client may call the function at the wrong place and at the wrong time:

```
void foo() {
  Student john {"jdoe", 20, 3, 3.10}; // construct a Student object
  john.free_login();                  // return memory back to free store
  john.display();                     // Oops very bad!!!
  // memory error - reading deallocated memory
}
```

Third, the client may call `free_login` more than once!!!

```
void foo() {
  Student john {"jdoe", 20, 3, 3.10}; // construct a Student object
  john.free_login();                  // return memory back to free store
  john.display();                     // Oops very bad!!!
  // memory error - reading deallocated memory
  john.free_login();                  // Oops!!!
  // possibly returning memory allocated to an other object
}
```

As the situation stands, function `free_login` has made clients responsible for the private memory of the class. Clients *will* forget to call function `free_login`. Clients *will* call function `free_login` and continue to use the object. Clients *will* call the function `free_login` twice or more.

Destructors

We can do better than force clients to call `Student::free_login` to avoid memory leaks or force clients to call `free_login` only once. The basic idea is to have the compiler know about a function that does the opposite of a constructor, just as it knows about the constructor. Inevitably, such a function is called a *destructor*. In the same way that a constructor is implicitly called when an object of a class is created, a destructor is implicitly called when an object goes out of scope. A constructor makes sure that an object is properly created and initialized. Conversely, a destructor makes sure that an object is properly cleaned up before it is destroyed.

We augment the definition of type `Student` with a destructor:

```
class Student {
public:
  // public interface - as before ...
  ~Student(); // destructor
private:
  // private implementation - as before ...
};
```

which is defined like this:

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

With the destructor defined by the class designer, the following code is fine:

```
void foo() {
  Student john {"jdoe", 20, 3, 3.10}; // construct Student object
  john.display();                     // display data
} // destructor is called here when john goes out of scope
```

The destructor will be called *automagically* when the object *goes out of scope*:

```
void foo() {
  Student john {"jdoe", 20, 3, 3.10};
  if (john.get_age() > 10) {
    Student jane {"jsmith", 19, 2, 3.95};
    // other code here ...
  } // jane's destructor called
  // other code here ...
} // john's destructor called
```

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

```
void f7() {
  Student john {"jdoe", 20, 3, 3.10};
  if (john.get_age() > 10) {
    Student jane {"jsmith", 19, 2, 3.95};
    if (jane.get_age() > 2) {
      return; // Destructor's for both jane and john called
    }
  }
}
```

> ***Destructors are conceptually simple but are the foundation for many of the most effective C++ programming techniques. The basic idea is simple:***
>
> - ***Whatever resources a class object needs to function, it acquires in a constructor.***
> - ***During the object's lifetime it may release resources and acquire new ones.***
> - ***At the end of the object's lifetime, the destructor releases all resources still owned by the object.***

Problem with mutator `Student::set_login`

While analyzing the many ways things can go wrong when clients are responsible for calling `free_login` to release the memory allocated by `set_login`, we realize that there's nothing to prevent clients' from changing students' login names after initializing the login name via the constructor. In the following example, the student requests a simpler login name

```
void foo() {
  Student john {"jrumplestiltzkin", 20, 3, 3.10}; // ctor
  john.display();
  john.set_login("jdoe");                         // set simpler login name
  john.display();
} // john::~Student() automagically called here
```

Studying the definition of member function `set_login`

```
// original definition of Student::set_login
void Student::set_login(char const *log) {
  login = new char[std::strlen(log) + 1];
  std::strcpy(login, log);
}
```

the following client code

```
void foo() {
  Student john {"jrumplestiltzkin", 20, 3, 3.10}; // ctor
  john.display();
  john.set_login("jdoe");                         // Oops!!! memory leak
  john.display();
} // john::~Student() automagically called here
```

will cause a memory leak because the call to member function `set_login` will orphan the free store memory pointed to by data member `login`. To fix this memory leak, member function `set_login` is modified:

```
// new definition of Student::set_login
void Student::set_login(char const *log) {
  delete [] login;
  login = new char[std::strlen(log) + 1];
  std::strcpy(login, log);
}
```

Member initialization list

The astute reader would've observed that modified member function `set_login` will itself cause undefined program behavior in the constructor. In the following code fragment, consider the definitions of the constructor, `Student::set_login`, and a `Student` object's definition and initialization:

```
Student::Student(char const *log, int a, int y, double g) {
  set_login(log);
  set_age(a);
  set_year(y);
  set_gpa(g);
}

void Student::set_login(char const *log) {
  delete [] login;
  login = new char[std::strlen(log) + 1];
  std::strcpy(login, log);
}

Student john {"jrumplestiltzkin", 20, 3, 3.10};
```

and the knowledge that construction of objects via a constructor proceeds in two phases:

1. Initialization of data members.
2. Execution of the constructor's body that was called.

This means that when member function `set_login` is called in line 2 of the constructor's body, data member `login` is already default-initialized to an indeterminate value. This means that line 2 of `Student::set_login` will call operator `delete[]` on an indeterminate value causing undefined program behavior.

We use the fact that data members are first initialized and then the constructor's body is executed and the knowledge that operator `delete` and operator `delete[]` have no effect on `nullptr` pointer operands to rewrite the constructor like this:

```
1 Student::Student(char const *log, int a, int y, double g)
2 : login{nullptr}, age{a}, year{y}, gpa{g} {
3   set_login(log);
4 }
```

The `: login(nullptr) ...` notation on line 2 is called *member initialization list* and ensures data members are initialized before the constructor's body is executed. There are two reasons for you to always use the member initialization list:

1. The first reason is efficiency. Suppose you've a data member of type `string`. If you don't use a member initialization list, then the data member is first default-initialized via the `string`s default constructor and then assigned via the overloaded copy-assignment operator. With a member initialization list, you replace these two steps with a copy-construction of the `string` data member.
2. The second reason is that with certain data members, the member initialization is necessary and required. In particular, `const` and reference members may *only* be initialized, never assigned.

The final thing to remember about the member initialization list is that the compiler doesn't care how you list the members in an initialization list. Instead, the compiler will always initialize data members in the order in which they're declared. Therefore, it is in your best interest to list the data members in the initialization list in the order in which they're declared in the class.

Memory layout of objects

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

```
1  Student::Student(char const *log, int a, int y, double g)
2  : login{nullptr}, age{a}, year{y}, gpa{g} {
3    set_login(log);
4    std::cout << "_PRETTY_FUNCTION__\n";
5  }
6
7  Student::~Student() {
8    std::cout << "_PRETTY_FUNCTION__\n";
9    delete [] login;
10 }
```

Consider the following code:

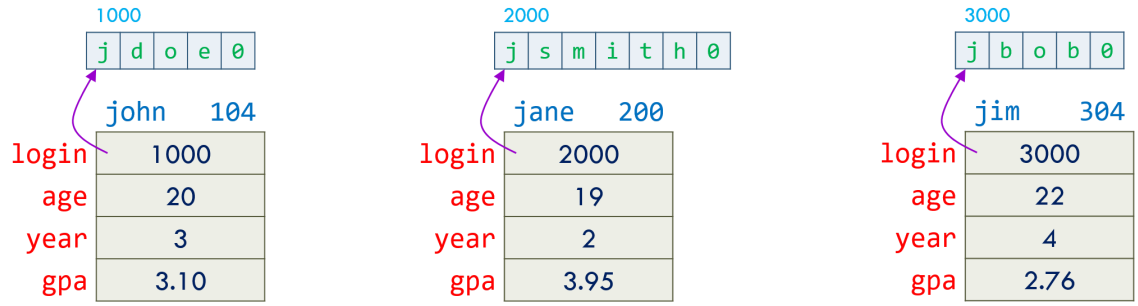
```
1 void foo() {
2   std::cout << "***** Begin *****\n";
3   Student john {"jdoe", 20, 3, 3.10};
4   Student jane {"jsmith", 19, 2, 3.95};
```

```
5     Student jim  {"jbob", 22, 4, 2.76};
6
7     john.set_age(21);   // Modify john
8     john.set_gpa(3.25);
9
10    jane.set_age(24);   // Modify jane
11    jane.set_gpa(4.0);
12
13    jim.set_age(23);    // Modify jim
14    jim.set_gpa(2.98);
15
16    john.display();     // Display all
17    jane.display();
18    jim.display();
19    std::cout << "***** End *****\n";
20 }
```

Lines $3 - 5$

```
1 Student john {"jdoe", 20, 3, 3.10};
2 Student jane {"jsmith", 19, 2, 3.95};
3 Student jim  {"jbob", 22, 4, 2.76};
```

will look something like this in memory [as usual, the addresses are arbitrary]:



The important points that can be understood from the picture include:

- Each object is a separate entity in memory, unrelated to the others.
- Each object has the same exact *structure* [layout] in memory.
- 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` or `class`.
- For example, the compiler finds the `login` member at offset 0, the `age` member at offset 8, the `year` member at offset 12, and the `gpa` member at offset 16. Recall that pointers and values of type `double` require 8 bytes of storage. Also, recall that machines require values to be stored at addresses that are multiples of their byte sizes.

	Student
+0	(login)
+8	(age)
+12	(year)
+16	(gpa)

- Given the diagrams above, this means that the `age` member for `jane` is at address 208 and the `gpa` member for `jim` is at address 316.
- In fact, for any `Student` object at address XYZ, the `age` member will be at address $XYZ + 8$. Always.

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

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

// nowhere does this code reference john, jane, or jim
void Student::display() {
  // These members are just offsets. But offsets from what?
  std::cout << "login: " << login << "\n";
  std::cout << "  age: " << age << "\n";
  std::cout << " year: " << year << "\n";
  std::cout << "  GPA: " << gpa << "\n";
}
```

The `this` pointer

All methods of a class 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 member function on:

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

is being supplied as a parameter to the function. In reality, member function `Student::display` is transformed from:

```
void Student::display() {
  std::cout << "login: " << login << "\n";
  std::cout << "  age: " << age   << "\n";
  std::cout << " year: " << year  << "\n";
  std::cout << "  GPA: " << gpa   << "\n";
}
```

to a function that takes a pointer to the `Student` object that invoked the function:

```
void Student::display(Student *this) {
  std::cout << "login: " << this->login << "\n";
  std::cout << "  age: " << this->age   << "\n";
  std::cout << " year: " << this->year  << "\n";
  std::cout << "  GPA: " << this->gpa   << "\n";
}
```

Note that the name of the parameter that points to the invoking object is always `this` which is a C++ keyword.

In turn, the calls to member function `display` by variables `john`, `jane`, and `jim` are transformed to:

```
Student::display(&john);
Student::display(&jane);
Student::display(&jim);
```

So, in a nutshell, this [no pun intended] is how the magic works. The programmer has access to the `this` pointer inside of the functions. Both of these lines are the same within `Student::display`:

```
// normal code
std::cout << "login: " << login << "\n";

// explicit use of this - generally only seen in beginner's code
std::cout << "login: " << this->login << "\n";
```

`const` member functions

Some variables are meant to be changed - that's why we call them "variables" - but some are not; that is, we've "variables" representing immutable values. These, we typically call *constant* or just `const`s. Consider:

```
void boo() {
  const Student john {"jdoe", 20, 3, 3.10};
  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.
}
```

Rightfully, the compiler will flag lines 3 and 4 as errors because the calls to mutator or transformer functions will attempt to change an immutable [or read-only or constant] object. However, lines 5 and 6 consist of calls to accessor or observer functions that do not change the state of object `john`. As the definition of `Student` stands so far, `john.get_age()` is an error because the compiler doesn't know that `get_age` doesn't change its `Student`. We never told it, so the compiler assumes that `get_age` may modify its `Student` and reports an error.

We can deal with this problem by classifying operations on a class as modifying and nonmodifying. That's a pretty fundamental distinction that helps us understand a class, but it also has a very practical importance: operations that don't modify the object can be invoked for `const` objects. For example:

```
class Student {
public:
  Student(char const* log, int a, int y, double g); // ctor
  ~Student(); // dtor

  void set_login(const char* login); // mutators can't be const
  void set_age(int age);             // ditto
  void set_year(int year);           // ditto
  void set_gpa(double g);            // ditto

  char const* get_login() const; // accessors are nonmodifying
  int get_age() const;           // ditto
  int get_year() const;          // ditto
  double get_gpa() const;        // ditto

  void display() const; // ditto with display()
private:
  char *login;
  int age, year;
  double gpa;
};

void foo() {
  Student const john {"jdoe", 20, 3, 3.10}; // read-only object

  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.
}
```

We use `const` right after the parameter list in a member function declaration to indicate that the member function can be called for a `const` object. We must also tag the definition of the function as `const`. Once we've declared a member function `const`, the compiler holds us to our promise not to modify the object.

> ***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 read-only object.***

Copying

Consider again our incomplete `Student`:

```
class Student {
public:
  Student(char const *log, int a, int y, double g); // ctor
  ~Student();                                       // dtor
  // ...
private:
  char *login;
  int age, year;
  double gpa;
};
```

Let's try to copy one of these students:

```
void foo() {
  Student s {"jdoe", 21, 4, 3.11}; // define a Student s
  Student s2 = s;                  // what happens here?
  // ...
}
```

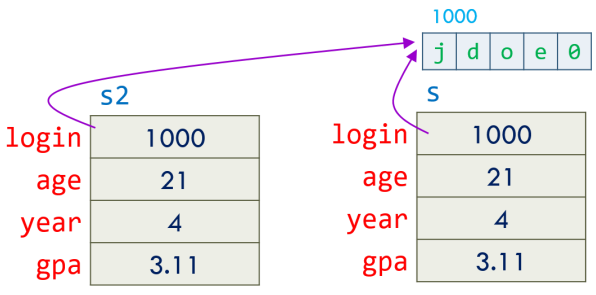
Ideally, `s2` becomes a copy [or replica] of `s`; that is, `std::strcmp(s.get_login(), s2.get_login())==0`, `s.get_age()==s2.get_age()`, `s.get_year()==s2.get_year()`, and `s.get_gpa()==s2.get_gpa()`. Furthermore, all memory is returned to the free store upon exit from function `foo`. That's what we except, but that's not what happens for our still-far-too-simple `Student`. Before we improve our `Student`, let's figure out what our current version actually does. Exactly what does it do wrong? How? And why? Once we know that, we can probably fix the problems. More importantly, you've a chance to recognize and avoid similar problems when you see them in other contexts.

Objects instantiated from classes and structures are first-class objects [similar to objects of built-in types]. Thus, the default meaning of copying for a class and structure is *shallow copy*; that is, "copy all the data members." That's what we learned in HLP1 and it makes perfect sense. For example, we copy a `Point` object by copying its coordinates:

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

Point p{1.1, 2.2}, p2 = p;
```

But for a pointer member, just copying the members causes problems. In particular, for the `Student`s in our example, it means that after the copy, we've `s.login==s2.login` so that our `Student`s look like this:



That is `s2` doesn't have a copy of `s`'s login character string; it shares `s`'s character string. Therefore, the result of the following code would be disastrous:

```
void foo() {
  Student s {"jdoe", 21, 4, 3.11}; // define a Student s
  Student s2{s};
  s.set_login("john");
  s2.set_login("bill");
  std::cout << s.get_login() << " | " << s2.get_login();
  // ...
}
```

Had there been no "hidden" connection between `s` and `s2`, we would have gotten the output $\text{john} \mid \text{bill}$. However, what we get now is undefined behavior. Since `login` points to the same memory location in both `s` and `s2`, that memory will be freed by the line `s.set_login("john")` using

```
delete [] login;
```

The subsequent lines in function `set_login`:

```
  login = new char[std::strlen(log) + 1];
  std::strcpy(login, log);
```

would make `s.login` proper [by re-allocating free store memory to store argument string `"john"` and copying the argument string into that memory]. However, `s2.login` is now dangling [since it will be pointing to the previously deleted memory]. As a consequence, the line `s2.set_login("bill")` would again free the memory that was previously deleted causing undefined behavior.

Even if we remove the lines causing this undefined behavior

```
void foo() {
  Student s {"jdoe", 21, 4, 3.11}; // define a Student s
  Student s2{s};
  std::cout << s.get_login() << " | " << s2.get_login();
  // ...
}
```

what happens when we return from `foo` is an unmitigated disaster. Then, the destructors for `s` and `s2` are implicitly called; `s`'s destructor frees the storage used for the login name using:

```
 delete [] login;
```

and so does `s2`'s destructor. Since `login` points to the same memory location in both `s` and `s2`, that memory will be freed twice with disastrous results.

Copy constructors

So, what do we do? We'll do the obvious: provide a copy operation that performs *deep copy* by copying the login name and make sure that this copy operation gets called when we initialize one `Student` with another.

Initialization of objects of a class is done by a constructor. So, we need a constructor that copies. Unsurprisingly, such as constructor is called a *copy constructor*. It is defined to take as its argument a reference to the object from which to copy. So, for class `Student` we need

```
1  Student(Student const&);
```

This constructor will be called when we initialize one `Student` with another. We pass by reference because we don't want to copy the argument of the constructor that defines copying and pay the attendant costs. We pass by `const` reference because we don't want to modify our argument. So, we refine `Student` like this:

```
1  class Student {
2  public:
3    Student(Student const&); // copy ctor: define copy
4    // same as before ...
5  private:
6    char *login;
7    int age, year;
8    double gpa;
9  };
```

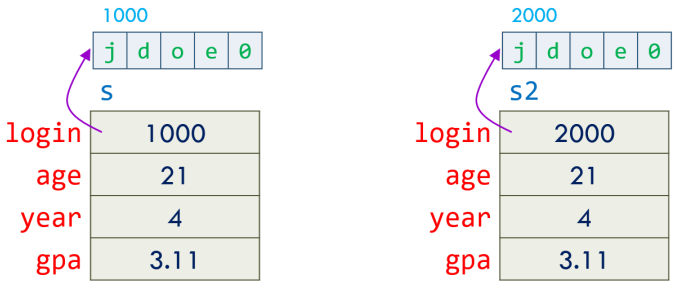
The copy constructor allocates memory for the character string [initializing `login`] before copying the individual characters of the login name from the argument `Student`:

```
1  Student::Student(Student const& rhs)
2      : login{new char [std::strlen(rhs.login)+1]}, age{rhs.age},
3        year{rhs.year}, gpa{rhs.gpa} {
4    std::strcpy(login, rhs.login);
5  }
```

Given this copy constructor, consider again our example:

```
1  Student s {"s", 20, 3, 3.10}; // define a Student s
2  Student s2{s};                // what happens here?
```

The definition on line 2 will initialize `s2` by a call of `Student`'s copy constructor with `s` as its argument. We now get:



Given that, the destructor can do the right thing. Each set of character strings is correctly freed. Obviously, the two `Student`s are now independent so that we can change the login name in `s` without affecting `s2` and vice versa. For example:

```
Student s {"s", 20, 3, 3.10}; // define a Student s
Student s2{s};                // what happens here?
s.set_login("john");
s2.set_login("bill");
std::cout << s.get_login() << " | " << s2.get_login();
```

The above code fragment will output $\text{john} \mid \text{bill}$.

Note that instead of saying

```
Student s2{s};
```

you could've equally well have said

```
Student s2 = s;
```

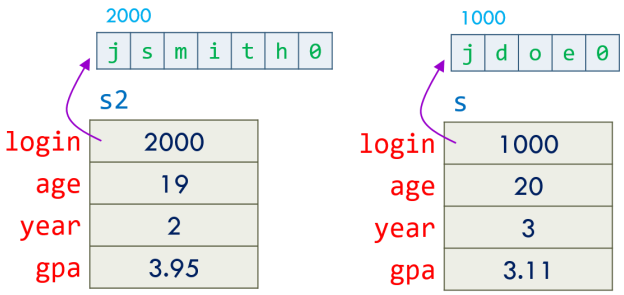
When `s` [the initializer] and `s2` [the variable being initialized] are of the same type and that type has copying conventionally defined, these two notations means exactly the same thing and you can use whichever notation you like better.

Copy assignments

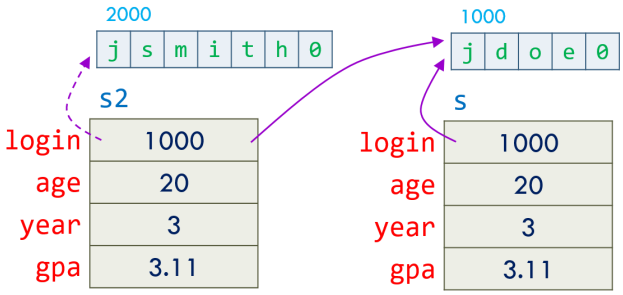
We handled copy construction [initialization], but we can also copy `Student`s by assignment. As with copy initialization, the default meaning of copy assignment is member-wise copy [or shallow copy], so with `Student` as defined so far, assignment will cause the same undefined behavior and double deletions [exactly as shown for copy constructors] plus a memory leak. For example:

```
void boo() {
  Student s {"jdoe", 20, 3, 3.11};    // define a Student
  Student s2 {"jsmith", 19, 2, 3.95}; // define another Student
  s2 = s;                             // assignment: what happens here?
  // ...
}
```

Before the assignment, the two `Student`s look like this:



We would like `s2` to be a copy of `s`, but since we said nothing about the meaning of assignment of our `Student`, the default assignment is used; that is, the assignment is a member-wise copy so that `s2`'s data members will become identical to `s`'s data members. We can illustrate that like this:



When we leave function `boo`, we've the same disaster as we had when leaving function `foo` before we added the copy constructor: the elements pointed to by both `s` and `s2` are freed twice [using `delete[]`]. In addition, we've leaked the memory initially allocated for `s2`'s login name. We "forgot" to free that block of memory. The remedy for this copy assignment is fundamentally the same as for the copy initialization. We define an assignment that copies properly:

```
class Student {
public:
  Student& operator=(Student const&); // copy assignment
  // same as before ...
private:
  char *login;
  int age, year;
  double gpa;
};

Student& Student::operator=(Student const& rhs) {
  char *p = new char [std::strlen(rhs.login)+1]; // allocate new space
  std::strcpy(p, rhs.login);                     // copy login name
  delete [] login;                               // deallocate old space
  login = p;                                     // now we can reset login
  age = rhs.age;
  year = rhs.year;
  gpa = rhs.gpa;
  return *this;                                  // return a self-reference
}
```

Assignment is a bit more complicated than construction because we must deal with the old memory that `login` is pointing to. Our basic strategy is to make a copy of the login name from the source `Student`:

```
  char *p = new char [std::strlen(rhs.login)+1]; // allocate new space
  std::strcpy(p, rhs.login);                     // copy login name
```

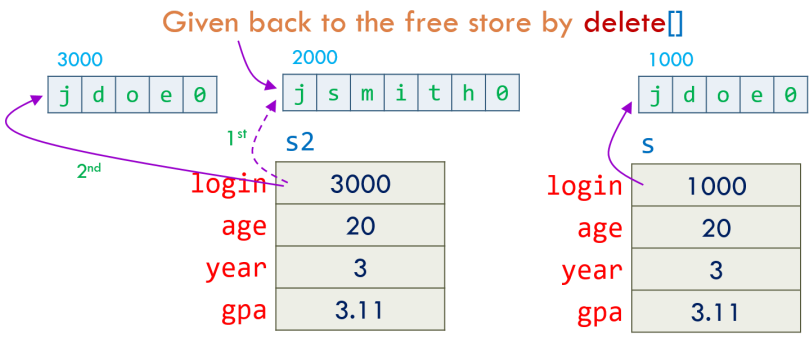
Then we free the old memory from the target `Student`:

```
  delete [] login;                               // deallocate old space
```

Finally, we let `login` point to the new login name and copy the values of the other members:

```
    login = p;          // now we can reset login
    age = rhs.age;
    year = rhs.year;
    gpa = rhs.gpa;
```

We can represent the result graphically like this:



We now have a `Student` that doesn't leak memory and doesn't free [`delete[]`] any memory twice.

When implementing the assignment, you might consider simplifying the code by freeing the memory for the old login name before creating the copy, but it usually is a very good idea not to throw away information before you know that you can replace it. Also, if you did that, strange things would happen if you assigned a `Student` to itself:

```
Student s {"jdoe", 20, 3, 3.11}; // define a Student
s = s;                           // self-assignment
```

Carefully check that our copy assignment function handles that case correctly although not with optimal efficiency.

Copy terminology

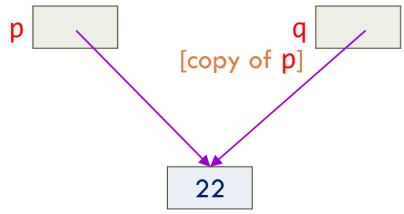
Copying is an issue in most programs and in most programming languages. The basic issue is whether you copy a pointer [or reference] or copy the information pointed to [referred to]:

- *Shallow copy* copies only a pointer so that the two pointers now refer to the same objects. That's what pointers and references do.
- *Deep copy* copies what a pointer points to so that the two pointers now refer to distinct objects. That's what `vector`s and `string`s, etc. and now `Student`s do. We define copy constructors and copy assignments when we want deep copy for objects of our classes.

Here is an example of shallow copy:

```
int *p {new int {11}};
int *q {p}; // copy the pointer p
*p = 22;    // change the value of the int pointed to by p and q
```

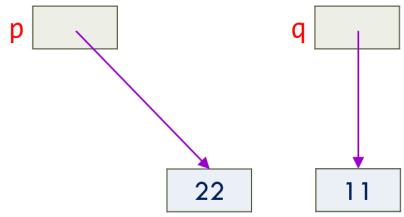
We can illustrate it like this:



In contrast, we can do a deep copy:

```
int *p {new int {11}};
int *q {new int{*p}}; // allocate a new int, then copy valued pointed to by p
*p = 22;              // change the value of the int pointed to by p
```

We can illustrate that like this:



Using this terminology, we can say that the problem with our original `Student` was that it did a shallow copy, rather, than copying the characters pointed to by its `login` pointer. Our improved `Student` does a deep copy by allocating new space for the login name and copying the characters. Types that provide shallow copy [like pointers and references] are said to have *pointer semantics* or *reference semantics* [they copy addresses]. Types that provide deep copy [like `string` and `vector`] are said to have *value semantics* [they copy the values pointed to]. From a user perspective, types with value semantics behave as if no pointers were involved - just values that can be copied. Another way of thinking of types with value semantics is that they "work just like `int`s" as far as copying is concerned.

Default constructors

Uninitialized variables can be a serious source of errors. To counter that problem, we've the notion of a constructor to guarantee that every object of a class is initialized. For example, we declared the constructor `Student(char const*,int,int, double);` to ensure that every `Student` is properly initialized. In the case of `Student`, that means the client must supply four arguments of the right types. For example:

```
Student s0;                         // error: no initializers
Student s1{};                       // error: empty initializer
Student s2{"jdoe"};                 // error: too few arguments
Student s3{"jdoe", 20, 3, 3.11, 4}; // error: too many arguments
Student s4{20, "jdoe", 3, 3.11};    // error: wrong argument type
Student s5{"jdoe", 20, 3, 3.11};    // ok: using four-argument ctor
Student s6{s5};                     // ok: use the copy ctor
```

Many classes have a good notion of a default value; that is, there is an obvious answer to the question "What value should it have if I didn't give it an initializer?" For example:

```
std::string s;              // default value: the empty string ""
std::vector<std::string> v; // default value: the empty vector [no elements]
```

This looks reasonable. It even works the way the comments indicate. That is achieved by giving `string` and `vector` *default constructors* that implicitly provide the desired initialization.

For a type `T`, `T{}` is the notation for the default value, as defined by the default constructor, so we could write

```
std::string s = std::string{}; // default value: the empty string ""
std::vector<std::string> v = std::vector<std::string>{}; // default value:
                                                         // the empty vector
                                                         // with no values
```

However, the equivalent and colloquial is preferred:

```
std::string s;              // default value: the empty string ""
std::vector<std::string> v; // default value: the empty vector [no elements]
```

For built-in types, such as `int` and `double`, the default constructor notation means `{}`, so `int{}` is a complicated way of saying `0`, and `double{}` is a long-winded way of saying `0.0`.

Using a default constructor is not just a matter of looks. Imagine that we could have an uninitialized `string` or `vector`:

```
std::string s;
for (int i{}; i<s.size(); ++i) // oops: loop an undefined number of times
  s[i] = std::toupper(s[i]);   // oops: read and write a random memory
location

std::vector<std::string> v;
v.push_back("error");          // oops: write to random address
```

If the values of `s` and `v` were genuinely undefined, `s` and `v` would have no notion of how many elements they contained or where those elements were supposed to be stored. The results would be use of random addresses - and that can lead to the worst kind of errors. Basically, without a constructor, we cannot establish an *invariant* [a rule for what constitutes a valid value] - we cannot ensure that the values in those variables are valid. To ensure validity of variables, we must insist that such variables are initialized. We could insist on an initializer and then write

```
std::string s = "";
std::vector<std::string> v{};
```

But that's not particularly pretty. For `string`, `""` is rather obvious for "empty string." For `vector`, `{}` means a vector with no elements. However, for many types, it is not easy to find a reasonable notation for a default value. For many types, it is better to define a constructor that gives meaning to the creation of an object without an explicit initializer. Such a constructor takes no arguments and is called a *default constructor*.

There isn't an obvious default value for students. That's why we haven't defined a default constructor for `Student` so far, but let's provide one [just to show we can]:

```
class Student {
public:
  Student(); // default ctor
  // same as before ...
private:
  char *login;
  int age, year;
  double gpa;
};
```

We have to pick a default login name, age, year of study, and GPA for the default student. There are no reasonable choices [which is why we did not design a default constructor in the first place]. Supposing that a default students has empty string, 19, 1, and 4.0 for the login name, age, year, and GPA, respectively, we can then write the default constructor for `Student` like this:

```
Student::Student() : login{new char[1]}, age{19}, year{1}, gpa{4.0} {
  *login = '\0';
}
```

Constructors initialize objects, so default constructors initialize objects without any information from the place where the object is being created. Since `Student` has a default constructor, we can default initialize `Student` objects:

```
Student s0; // ok:
            // login name is ""
            // age is 19
            // study year is 1
            // GPA is 4.0
```

Initializing objects without any information sometimes makes perfect sense. Objects that acts like numbers, for example, may reasonably be initialized to zero. Objects that act like pointers may reasonably be initialized to `nullptr`. Not all objects fall into this category. For many objects such as `Student`s, there is no reasonable way to perform a complete initialization in the absence of outside information. For example, every legal resident in Singapore must have a NRIC number, so it would rarely make sense to model Singapore residents without insisting that a NRIC number be provided. An object representing an entry in an address book makes no sense unless the name of the thing being entered is provided. For this reason, you can't create a contact in your phone unless there is a specific name [and a phone number] for that contact. In many schools, all students must be tagged with a student ID number, and creating an object to model a student in such schools is nonsensical unless the appropriate ID number is provided.

> ***For these reasons, our `Student` class should not have a default constructor!!!***

On the other hand, it is reasonable for a type `Point`

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

to have a default constructor. 0.0 is an obvious default value for both the x and y coordinates of a point in two-dimensional Cartesian coordinate system. That is, when a point is created from nothing it would be located at the origin $(0, 0)$ of the coordinate system. For example:

```
struct Point {
  double x, y;
  Point();                      // default ctor
  Point(double cx, double cy); // ctor to locate point at specific location
};

Point::Point() : x{}, y{} {}
Point::Point(double cx, double cy) : x{cx}, y{cy} {}
```

The best way to specify the default value for a class data member is as an *in-class initializer* - an initializer that is specified as part of the data member declaration. The default constructor then doesn't have to initialize the data members through a member initializer list:

```
struct Point {
  double x{}, y{]};            // in-class initializer
  Point();                      // default ctor
  Point(double cx, double cy); // ctor to locate point at specific location
};

Point::Point() {} // nothing to do: members have in-class initializers
Point::Point(double cx, double cy) : x{cx}, y{cy} {}
```

We can now define `Point` objects like this:

```
Point p0;           // ok: p0(0, 0)
Point p1{};         // ok: p1(0, 0)
Point p2{1.1};      // error: too few [or too many] arguments
Point p3{1.1, 2.2}; // ok: p3(1.1, 2.2)
Point p4{p3};       // ok: use the default shallow copy
```

Since a default constructor can be called without any arguments, it means that such a constructor either has no parameters or has a default value for every parameter. This means we can combine the two `Point` constructors into a single constructor:

```
struct Point {
  double x{}, y{};
  Point(double cx=0.0, double cy=0.0); // default ctor
};

Point::Point(double cx, double cy) : x{cx}, y{cy} {}
```

We can now define `Point` objects like this:

```
Point p0;           // ok: p0(0, 0)
Point p1{};         // ok: p1(0, 0)
Point p2{1.1};      // ok: p2(1.1, 0)
Point p3{1.1, 2.2}; // ok: p3(1.1, 2.2)
Point p4{p3};       // ok: use the default shallow copy
```

Role of default constructor

In a perfect world, classes in which objects could reasonably be created from nothing would contain default constructors and classes in which information was required for object construction would not. There are additional concerns that class designers must take into account. In particular, if a class lacks a default constructor, there are restrictions on how you can use that class.

Consider a `Point` class without a default constructor:

```
struct Point {
  double x, y;
  Point(double cx, double cy); // ctor to locate point at specific location
};

Point::Point(double cx, double cy) : x{cx}, y{cy} {}
```

Because `Point` lacks a default constructor, its use is problematic in four contexts:

1. The first is the creation of static arrays. It is possible to explicitly specify constructor arguments for objects in a *small array*. But, in general, there is no way to explicitly specify constructors for objects in larger arrays, so it is not usually possible to create arrays of `Point`:

   ```
   std::array<Point, 3> p1
       {Point{1.1,2.2}, Point{2.1,3.2}, Point{3.1,4.2}}; // ok
   Point p2[10];              // error: no way to call Point ctor
   std::array<Point, 30> p3; // error: no way to call Point ctor
   ```

2. We also cannot create dynamic array for the same reasons we cannot create static arrays:

   ```
   Point *pp1 = new Point [3]
               {Point{1.1, 2.2}, Point{2.1, 3.2}, Point{3.1, 4.2}}; // ok
   Point *pp2 = new Point [30]; // error: no way to call Point ctor
   ```

3. The third problem with classes lacking default constructors is that they can be used in a limited way with containers from the standard library such as `std::vector`, `std::forward_list`, `std::list`, and so on. That's because it's a common requirement for these containers that the type of objects that will be stored in the container must have a default constructor. Therefore, it is not possible to have a vector of 10 `Point`s:

   ```
   std::vector<Point> vp(10); // error: no way to call Point ctor
   ```

   Luckily, we can reserve a certain number of `Point`s and insert them one by one:

```
std::vector<Point> vp;
vp.reserve(100);
vp.push_back(Point{1.1, 2.2});
vp.push_back(Point{1.2, 2.3});
vp.push_back(Point{1.3, 2.4});

for (Point const& rp : vp) {
  std::cout << '(' << rp.x << ", " << rp.y << ')' << '\n';
}
```

4. What may be less obvious is the impact on classes that have data members that do not have a default constructor:

```
class NoDefault {
public:
  NoDefault(std::string const&);
  // additional members follow, but no other ctors
};
struct A {
  NoDefault my_mem;
};

A a; // error: no way to call NoDefault ctor
```

Because of the restrictions imposed on classes lacking default constructors, some people believe all classes should have them, even if a default constructor doesn't have enough information to full initialize objects of that class. Adherents to this philosophy would ensure `Student` would have a default constructor [which was discussed earlier] allowing `Student` objects to be created like this:

```
Student s0; // ok:
            // login name is ""
            // age is 19
            // study year is 1
            // GPA is 4.0
```

Such a transformation almost always complicates the design and implementation of other member functions of the class, because there is no longer any guarantee that the data members of a `Student` object would have been meaningfully initialized. Assuming it makes no sense to have a `Student` without a login name and other attributes, most member functions must check to see if these attributes are present. If not, they'll have to figure out how to stumble on anyway. Often it's not clear how to do that, and many implementations choose a solution that offers nothing but expediency: they throw an exception or they call a function that terminates the program. When that happens, it's difficult to argue that the overall quality of the software has been improved by including a default constructor in a class where none was warranted.

Essential class operations

We've now reached the point where we can discuss how to decide which constructors a class should have, whether it should have a destructor, and whether you need to provide copy operations. There are five essential operations to consider:

- Constructors from one or more arguments

- Default constructor
- Copy constructor [copy object of same type]
- Copy assignment [copy object of same type]
- Destructor

Usually we need one or more constructors that take arguments needed to initialize an object. For example:

```
std::string str{"world.jpg"}; // initialize str to cstring "world.jpg"
Image ii{Point{200,300}, "world.jpg"}; // initialize a Point with the
                                       // coordinates (200, 300)
                                       // then display the contents of
                                       // file world.jpg at that Point
Student s {"jdoe", 20, 3, 3.10}; // define a Student s with login name jdoe
                                 // age of 20 years, enrolled in year 3
                                 // with GPA 3.10
```

The `std::string`'s constructor uses a character string as an initial value, whereas `Image`'s constructor uses the string as the name of a file to open. Usually we use a constructor to establish an invariant [a rule for what constitutes a valid value]. If we can't define a good invariant for a class that its constructors can establish, we probably have a poorly designed class or a plain data structure.

Constructors that take arguments are as varied as the class they serve. The remaining operations have more regular patterns.

How do we know if a class needs a default constructor? As discussed in an earlier section, we need a default constructor if we want to be able to make objects of the class without specifying an initializer. The most common example is when we want to put objects of a class into a `std::vector`. The following works only because we've default values for `double`, `std::string`, and `std::vector<int>`:

```
std::vector<double> vd(10); // vector of 10 doubles: each initialized to 0.0
std::vector<std::string> vs(10); // vector of 10 strings:
                                 // each initialized to ""
std::vector<std::vector<int>> vii(10); // vector of 10 vectors:
                                       // each initialized to vector<int>{}
```

So, having a default constructor is often useful. The question then becomes: "When does it make sense to have a default constructor?" An answer is: "When we can establish the invariant for the class with a meaningful and obvious default value." For value types such as `int` the obvious value is `0` and for `double` the obvious value is `0.0`. For `std::string`, the empty string `""` is the obvious choice. For `std::vector`, the empty `vector` serves well. Recall that we could not establish meaningful default values for `Student` and therefore decide to exclude the default constructor from its interface.

A class needs a destructor if it acquires resources. A resource is something you "get from somewhere" and that you must give back once you've finished using it. The obvious example is memory that you get from the free store using `new` or `new[]` and have to give back to the free store using `delete` or `delete[]`. Our `Student` acquires memory during execution of its constructors to hold the login name, so it has to give that memory back: therefore, it needs a

destructor. Other resources that you might encounter as your programs increase in ambition and sophistication are files, threads, locks, and network sockets.

Another sign that a class needs a destructor is simply that it has members that are pointers or references. If a class has a pointer or a reference member, it often needs a destructor and copy operations.

A class that needs a destructor almost always also needs a copy constructor and a copy assignment. The reason is simply that if an object has acquired a resource [and has pointer member pointing to it], the default meaning of copy [which is shallow copy] is almost certainly wrong. Again, `std::vector` is the classic example.

Separating interface from implementation

So, let's put it all together and see what that `Student` class might look like when we combine all of the above ideas. Typically, each class will reside in its own file. In fact, it will generally be in two files:

- A header file contains the class definition, usually in a `.h` or `.hpp` *interface* file.

```
class Student {
public:
  Student(char const* log, int a, int y, double g); // ctor
  Student(Student const&); // copy ctor: define copy
  ~Student(); // dtor

  Student& operator=(Student const&); // copy assignment

  void set_login(const char* login); // mutators can't be const
  void set_age(int age);             // ditto
  void set_year(int year);           // ditto
  void set_gpa(double g);            // ditto

  char const* get_login() const; // accessors are nonmodifying
  int get_age() const;           // ditto
  int get_year() const;          // ditto
  double get_gpa() const;        // ditto

  void display() const; // ditto with display()
private:
  char *login;
  int age, year;
  double gpa;
};
```

- The implementation file - contains all of the methods [in a `.cpp`] that are declared in the *implementation* file:

```
#include <iostream>
#include <cstring>
#include "student.hpp"

Student::Student(char const *log, int a, int y, double g)
: login{nullptr}, age{a}, year{y}, gpa{g} {
```

```
  set_login(log);
}
Student::Student(Student const& rhs)
    : login{new char [std::strlen(rhs.login)+1]}, age{rhs.age},
      year{rhs.year}, gpa{rhs.gpa} {
  std::strcpy(login, rhs.login);
}
Student::~Student() {
  delete [] login;
}
Student& Student::operator=(Student const& rhs) {
  char *p = new char [std::strlen(rhs.login)+1]; // allocate new space
  std::strcpy(p, rhs.login);                     // copy login name
  delete [] login;                               // deallocate old space
  login = p;                              // now we can reset login
  age = rhs.age;
  year = rhs.year;
  gpa = rhs.gpa;
  return *this;                           // return a self-reference
}
void Student::set_login(char const *log) {
  delete [] login;
  login = new char[std::strlen(log) + 1];
  std::strcpy(login, log);
}
void Student::set_year(int y) {
  if ( (y < 1) || (y > 4) ) {
    std::cerr << "Error in year range!\n";
    year = 1;
  } else { year = y; }
}
void Student::set_age(int a) {
  if ( (a < 18) || (a > 100) ) {
    std::cout << "Error in age range!\n";
    age = 18;
  } else { age = a; }
}
void Student::set_gpa(double g) {
  if ( (g < 0.0) || (g > 4.0) ) {
    std::cout << "Error in GPA range!\n";
    gpa = 0.0;
  } else { gpa = g; }
}
void Student::display() const {
  std::cout << "login: " << login << "\n";
  std::cout << "  age: " << age << "\n";
  std::cout << " year: " << year << "\n";
  std::cout << "  GPA: " << gpa << "\n";
}

int Student::get_age()  const          { return age;}
int Student::get_year() const          { return year; }
double Student::get_gpa() const        { return gpa; }
char const* Student::get_login() const { return login; }
```

Thus, our simple project is split into three files:

1. The interface file `student.hpp`
2. The implementation file `student.cpp`
3. The client code `main.cpp`

We could then build an executable `student.exe` like this:

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror main.cpp student.cpp
-o student.exe
```

Final thoughts

We designed and implemented `Student` as a mechanism to learn about classes and class operations. Because of the choice of the `login` data member in `Student`

```
class Student {
public:
  // public interface ...
private:
  char *login;   // raw pointer exhibits pointer semantics
  int age, year; // exhibit value semantics
  double gpa;    // exhibits value semantics
};
```

For `Student` objects to exhibit value semantics [recall that a type with value semantics is that it works just like an `int`, as far as copying is concerned], we had to implement a copy constructor, copy assignment, and a destructor. Life would be a lot simpler if a abstract data type is used to store and manipulate login names of students. Replacing the raw pointer `char*` type for data member `login` with standard `std::string` enables `login` to have value semantics similar to other data members. This means that we no longer have to provide a copy constructor, copy assignment, and a destructor. As a final simplification, we use function overloading to simplify the names provided to accessors and mutators. For example:

```
class Student {
public:
  Student(std::string const&, int, int, double);

  std::string const& Login() const;             // accessor
  std::string&       Login();                   // modifier
  void               Login(std::string const&); // modifier

  int const& Age() const;    // accessor
  int&       Age();          // modifier
  void       Age(int);       // modifier

  int const& Year() const;   // accessor
  int&       Year();         // modifier
  void       Year(int);      // modifier

  double const& Gpa() const; // accessor
  double&       Gpa();       // modifier
  void          Gpa(double); // modifier
```

```
20 private:
21   std::string login;
22   int age, year;
23   double gpa;
24 };
25 std::ostream& operator<<(std::ostream&, Student const&);
```

The definitions of the interface is left to the reader as an exercise.

Review

1. A class represents a concept. Instead of groups of related variables, design a class for the underlying concept.
2. Every class has a public interface: a collection of member functions through which class objects can be manipulated.
3. A constructor is used to initialize objects when they are created. A constructor with no parameters is called a *default constructor*.
4. A mutator member function changes the state of the object on which it operates.
5. An accessor member function does not modify the object. Accessors must be tagged with `const`.
6. Every class has a private implementation: data fields that store the state of an object.
7. Encapsulation is the act of hiding implementation details.
8. Encapsulation protects the integrity of object data.
9. By keeping the implementation private, we protect it from being accidentally corrupted.
10. Encapsulation enables changes in the implementation without affecting users of a class.
11. Use keyword `const` when defining accessor member functions.
12. The object on which a member function is applied is the implicit parameter. Every member function has an implicit parameter specified by keyword `this`.
13. Explicit parameters of a member function are listed in the function definition.
14. The purpose of a constructor is to initialize an object's data fields.
15. A default constructor has no parameters.
16. It is particularly important to initialize all numeric fields in a constructor because they are not automatically initialized.
17. A function name is overloaded if there are different versions of the function, distinguished by their parameter types.
18. Private data fields can only be accessed by member functions of the same class.
19. What is the default meaning of copying for class objects?
20. When is the default meaning of copying of class objects appropriate? When is it inappropriate?
21. What is a copy constructor?
22. What is a copy assignment?
23. What is the difference between copy assignment and copy initialization?
24. What is shallow copy? What is deep copy?
25. What is a destructor?
26. What are the five "essential operations" for a class?
27. The code of complex programs is distributed over multiple files.
28. Header files contain the definitions of classes and declarations of shared constants, functions, and variables.
29. Source files contain implementations of member functions.