Brief Overview of Classes and Objects

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 200 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

```
1 | 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 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 it [such as `+` and `*`] without being told by declarations supplied by a programmer in source code. 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 project mates, 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:

```
1  class X {
2  public:
3    int m;                                    // data member
4    int mf(int v) { int old {m}; m = v; return old; } // member function
5  };
```

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:

```
1  X var{10}, obj; // var and obj are variables of type X
2                  // just as with structs, data members can be initialized
3  obj.m = var.mf(5); // assign to obj's data member m the value returned
4                     // by a call to var's member function mf
5  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 `private:` . You can think of a class declaration like this:

```cpp
class X {
public:
   // public members - the interface accessible by all:
   // 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 are only accessible
   // to members of this class only:
   // functions
   // types
   // data
};
```

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

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

means

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

so that

```cpp
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:

```cpp
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:

```
1   struct X {
2     int m;
3     // ...
4   };
```

means

```
1   class X {
2   public:
3     int m;
4     // ...
5   }
```

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

```
1   struct Point {
2     double x, y;
3     // member functions ...
4   };
```

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:

```
1   const int MAXLENGTH = 10;
2
3   struct Student {
4     char login[MAXLENGTH];
5     int age, year;
6     double GPA;
7   };
8
9   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:

```
1   s1.age = 20;
2   s1.GPA = 3.8;
3   s1::strcpy(s1.login, "jdoe");
4   s1.year = 3;
```

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

```
1   s2.age = -5;
2   s2.GPA = 12.9;
3   std::strcpy(s2.login, "rumplestiltzkin");
4   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:

```
1    void set_login(Student& student, char const* log) {
2      int len = std::strlen(log);
3      std::strncpy(student.login, log, MAXLENGTH - 1);
4      if (len >= MAXLENGTH) {
5        student.login[MAXLENGTH - 1] = '\0';
6      }
7    }
8
9    void set_year(Student& student, int y) {
10     if ( (y < 1) || (y > 4) ) {
11       std::cerr << "Error in year range!\n";
12       student.year = 1;
13     } else { student.year = y; }
14   }
15   void set_age(Student &student, int a) {
16     if ( (a < 18) || (a > 100) ) {
17       std::cout << "Error in age range!\n";
18       student.age = 18;
19     } else { student.age = a; }
20   }
21
22   void set_GPA(Student& student, double gpa) {
23     if ( (gpa < 0.0) || (gpa > 4.0) ) {
24       std::cout << "Error in GPA range!\n";
25       student.GPA = 0.0;
26     } else { student.GPA = gpa; }
27   }
28
29   void display(Student const& student) {
30     std::cout << "login: " << student.login << "\n";
31     std::cout << "  age: " << student.age << "\n";
32     std::cout << " year: " << student.year << "\n";
33     std::cout << "  GPA: " << student.GPA << "\n";
34   }
```

Now, this code:

```
1  Student s3;
2  set_age(s3, -5);
3  set_GPA(s3, 12.9);
4  set_login(s3, "rumplestiltzkin");
5  set_year(s3, 150);
6  display(s3);
```

will result in this:

```
1  Error in age range!
2  Error in GPA range!
3  Error in year range!
4  login: rumplesti
5    age: 18
6   year: 1
7    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:

```
1  Student s4;
2  display(s4);
3  set_age(s3, -5);
4  set_GPA(s3, 12.9);
5  set_login(s3, "rumplestiltzkin");
6  set_year(s3, 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 in a `public` section, 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 class definition:

```
1   int const MAXLENGTH {10};
2
3   struct Student {
4     void init(char const *log, int a, int y, double gpa) {
5       set_login(log); set_age(a); set_year(y); set_GPA(gpa);
6     }
7     void set_login(char const* log);
8     void set_age(int a);
9     void set_year(int y);
10    void set_GPA(double gpa);
11    void display();
12
```

```
13    char login[MAXLENGTH];
14    int age, year;
15    double GPA;
16  };
```

The definitions of the member functions will change slightly. 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:

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

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 waas 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

```
1  struct Student {
2    char login[MAXLENGTH];
3    int age, year;
4    // ...
5  };
```

is identical to this structure definition:

```
1  struct Student {
2  public:
3    char login[MAXLENGTH];
4    int age, year;
5    // ...
6  };
```

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

```
1  void f() {
2    Student s; // define but not initialize a Student object
3    s.set_login("jdoe"); // using the public member functions
4    s.set_age(22);
5    s.set_year(4);
6    s.set_GPA(3.8);
7    s.display(); // tell the object to display itself
8  }
9
10 void g() {
11   Student s;
12   s.init("jdoe", 22, 4, 3.8); // call set_ functions
13   s.display(); // tell the object to display itself
14 }
```

Keep details private

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

```
1   Student s;
2   std::strcpy(s.login, "rumplestiltzkin");
3   s.age = -5;
4   s.set_year(2);
5   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. C++ provides three of them:*
>
> - `public`*: accessible from everywhere;*
> - `private`*: accessible only within the class or structure; and*
> - `protected`*: accessible in the class or structure itself and its derived classes or structures.*

Here is a first cut:

```
1    struct Student {
2    public:
3      void set_login(char const* log);
4      void set_age(int a);
5      void set_year(int y);
6      void set_GPA(double gpa);
7      void init(char const *log, int a, int y, double gpa);
8      void display();
9    private:
10     char login[MAXLENGTH];
11     int age, year;
12     double GPA;
13   };
```

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:

```
1   Student s0 {"jdoe", 21, 3, 3.8}; // error: data members are private
2   Student s1;
3   std::strcpy(s1.login, "jdoe"); // error: login is private data member
4   s1.age = 20;                    // error: age is private data member
5   s1.year = 3;                    // error: year is private data member
6   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`:

```
1    Student s0; // define a Student object
2    s0.set_login("jdoe"); // using the public member functions
3    s0.set_age(22);
4    s0.set_year(4);
5    s0.set_GPA(3.8);
6    s0.display(); // tell the object to display itself
7
8    Student s1;
9    s1.init("jdoe", 22, 4, 3.8); // call set_ functions
10   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 leads to someone forgetting to immediately initialize a `Student` object before using it:

```
1    Student s;   // uninitialized student
2    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]:

```
1    struct Student {
2    public:
3      Student(char const *log, int a, int y, double gpa); // constructor
4      void set_login(char const* log);
5      void set_age(int a);
6      void set_year(int y);
7      void set_GPA(double gpa);
8      //void init(char const *log, int a, int y, double gpa);
9      void display();
10   private:
11     char login[MAXLENGTH];
12     int age, year;
13     double GPA;
14   };
```

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

```
1  Student::Student(char const* log, int a, int y, double gpa) {
2    set_login(log); set_age(a); set_year(y); set_GPA(gpa);
3  }
```

It is an error - caught by the compiler - to forget to initialize an object of a structure that has a constructor that requires an argument, and there is a special convenient syntax for doing such initialization:

```
1  Student s0; // error: s0 is not initialized
2  Student s1("jdoe", 22, 4, 3.8); // OK - C++98 style
3  Student s2 = Student("jdoe", 22, 3, 3.6); // Ok - verbose C++98 style
4  Student s3 {"jdoe", 21, 3, 3.6}; // OK - colloquial style
5  Student s4 = {"jdoe", 20, 2, 3.4}; // OK - verbose style
6  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:

```
1  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:

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

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

```
1  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. Important and powerful 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 of members in a `class` is `private`. Here is the third cut of type `Student` that replaces keyword `struct` with keyword `class`:

```
1   class Student {
2   public:
3       Student(char const *log, int a, int y, double gpa); // constructor
4       void set_login(char const* log);
5       void set_age(int a);
6       void set_year(int y);
7       void set_GPA(double gpa);
8       void display();
9   private:
10      char login[MAXLENGTH];
11      int age, year;
12      double GPA;
13  };
```

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`:

```
1   struct Point {
2       double x, y;
3   };
```

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 the data in a class is usually `private`, the only way to gain access to the data is by providing `public` member functions that explicitly allow it. We can categorize the 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 but their purpose is to return resources that were allocated to the object during its lifetime.
- *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 one or more constructors are defined, the class definition must define one and only one destructor. 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`:

```cpp
int const MAXLENGTH {10};
class Student {
public:
  Student(char const* log, int a, int y, double gpa); // ctor

  // accessors or observers or gettors
  int get_age()        { return age;}
  int get_year()       { return year; }
  double get_GPA()     { return GPA; }
  char const* get_login() { return login; }

  // mutators or transforms or settors
  void set_login(char const* log);
  void set_age(int a);
  void set_year(int y);
  void set_GPA(double gpa);

  void display();

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

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 class `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 but are unable to change their 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:

```
1  class Student {
2  public:
3    // public interface is same as before except set_login ...
4  private:
5    char *login;
6    int age, year;
7    double GPA;
8  };
```

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

```
1  void Student::set_login(char const* log) {
2    login = new char[std::strlen(log) + 1];
3    std::strcpy(login, log);
4  }
```

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:

```
1  void foo() {
2    Student john{"jrumplestiltzkin", 20, 3, 3.10}; // construct Student object
3    john.display(); // display all of the data
4  }
```

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

```
1  login: jrumplest
2    age: 20
3   year: 3
4    GPA: 3.1
```

while the new version would display the following information:

```
1  login: jrumplestiltzkin
2    age: 20
3   year: 3
4    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:

```
1  class Student {
2  public:
3    // public interface - as before ...
4    void free_login();
5  private:
6    // private implementation - as before ...
7  };
```

and provide a definition like this:

```
1   void Student::free_login() {
2      delete [] login;
3   }
```

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:

```
1   void foo() {
2      Student john{"jrumplestiltzkin", 20, 3, 3.10}; // construct Student object
3      john.display(); // display all of the data
4      john.free_login(); // return free memory
5   }
```

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`:

```
1   void foo() {
2      Student john {"jdoe", 20, 3, 3.10}; // construct a Student object
3      john.display(); // display data
4      // Oops, memory leak now!
5   }
```

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

```
1   void foo() {
2      Student john {"jdoe", 20, 3, 3.10}; // construct a Student object
3      john.free_login(); // return memory back to free store
4      john.display(); // Oops very bad!!!
5      // memory error - reading deallocated memory
6   }
```

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

```
1   void foo() {
2      Student john {"jdoe", 20, 3, 3.10}; // construct a Student object
3      john.free_login(); // return memory back to free store
4      john.display(); // Oops very bad!!!
5      // memory error - reading deallocated memory
6      john.free_login(); // Oops!!!
7      // possibly returning memory allocated to an other object
8   }
```

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 class `Student` with a declaration of destructor:

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

The destructor is defined like this:

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

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

```cpp
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*:

```cpp
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:

```
1  void f7() {
2    Student john {"jdoe", 20, 3, 3.10};
3    if (john.get_age() > 10) {
4      Student jane {"jsmith", 19, 2, 3.95};
5      if (jane.get_age() > 2) {
6        return; // Destructor's for both jane and john called
7      }
8    }
9  }
```

> *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 member function `free_login` to release the memory allocated by member function `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

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

Studying the definition of member function `set_login`

```
1  // original definition of Student::set_login
2  void Student::set_login(char const* log) {
3    login = new char[std::strlen(log) + 1];
4    std::strcpy(login, log);
5  }
```

the following client code

```
1  void foo() {
2    Student john {"jrumplestiltzkin", 20, 3, 3.10}; // ctor
3    john.display();
4    john.set_login("jdoe"); // Oops!!! memory leak
5    john.display();
6  } // 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:

```
1   // new definition of Student::set_login
2   void Student::set_login(char const* log) {
3     delete [] login;
4     login = new char[std::strlen(log) + 1];
5     std::strcpy(login, log);
6   }
```

Member initialization list

The astute reader would've observed that modified member function `set_login` will itself cause undefined program behavior in the constructor. Consider the constructor's definition:

```
1   Student::Student(char const* log, int a, int y, double gpa) {
2     set_login(log);
3     set_age(a);
4     set_year(y);
5     set_GPA(gpa);
6   }
```

the definition of `Student::set_login`:

```
1   void Student::set_login(char const* log) {
2     delete [] login;
3     login = new char[std::strlen(log) + 1];
4     std::strcpy(login, log);
5   }
```

a `Student` object's definition and initialization:

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

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

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

All of this means that by the time 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 gpa)
2   : login(nullptr), age(a), year(y), GPA(gpa) {
3     set_login(log);
4     set_age(a);
5     set_year(y);
6     set_GPA(gpa);
7   }
```

The `: login(nullptr) ...` notation 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:

```cpp
Student::Student(char const* log, int a, int y, double gpa)
: login(nullptr), age(a), year(y), GPA(gpa) {
  set_login(log);
  set_age(a);
  set_year(y);
  set_GPA(gpa);
  std::cout << "Student constructor for " << login << "\n";
}

Student::~Student() {
  std::cout << "Student destructor for " << login << "\n";
  delete [] login;
}
```

Consider the following code:

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

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

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

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

  john.display();      // Display all
  jane.display();
```

```
18    jim.display();
19    std::cout << "***** End *****\n";
20  }
```
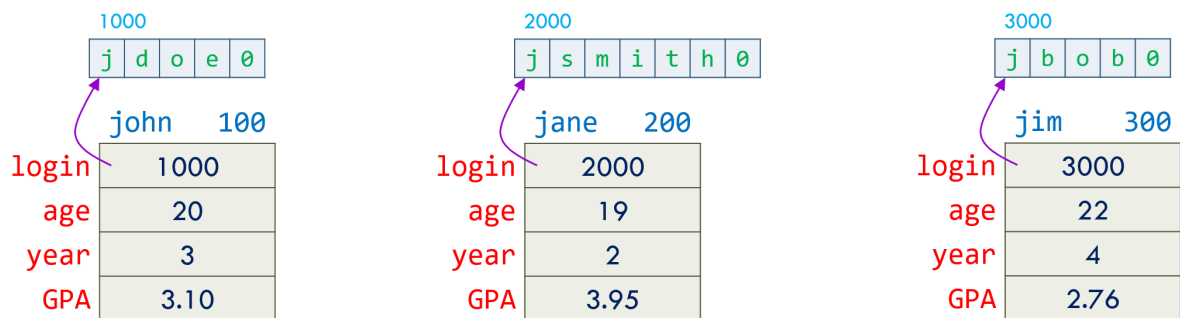
Lines 3-5

```
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.

- 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?

```
1   john.display();
2   jane.display();
3   jim.display();
4
5   // nowhere does this code reference john, jane, or jim
6   void Student::display() {
7     // These members are just offsets. But offsets from what?
8     std::cout << "login: " << login << "\n";
9     std::cout << "  age: " << age << "\n";
10    std::cout << " year: " << year << "\n";
11    std::cout << "  GPA: " << GPA << "\n";
12  }
```

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:

```
1   john.display(); // john is the invoking object
2   jane.display(); // jane is the invoking object
3   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:

```
1   void Student::display() {
2     std::cout << "login: " << login << "\n";
3     std::cout << "  age: " << age   << "\n";
4     std::cout << " year: " << year  << "\n";
5     std::cout << "  GPA: " << GPA   << "\n";
6   }
```

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

```
1   void Student::display(Student *this) {
2     std::cout << "login: " << this->login << "\n";
3     std::cout << "  age: " << this->age   << "\n";
4     std::cout << " year: " << this->year  << "\n";
5     std::cout << "  GPA: " << this->GPA   << "\n";
6   }
```

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:

```
1   Student::display(&john);
2   Student::display(&jane);
3   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`:

```
1  // normal code
2  std::cout << "login: " << login << "\n";
3
4  // explicit use of this - generally only seen in beginner's code
5  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:

```
1  void boo() {
2     const Student john {"jdoe", 20, 3, 3.10};
3     john.set_age(25); // Error, as expected.
4     john.set_year(3); // Error, as expected.
5     john.get_age();   // Error, not expected.
6     john.display();   // Error, not expected.
7  }
```

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:

```
1  struct Student {
2  public:
3    Student(char const* log, int a, int y, double gpa); // ctor
4    ~Student(void); // dtor
5
6    // Mutators (settors) are rarely const
7    void set_login(const char* login); // Mutators can't be const
8    void set_age(int age);
9    void set_year(int year);
10   void set_GPA(double GPA);
11
12   int get_age() const; // accessors are nonmodifying
13   int get_year() const;
14   double get_GPA() const;
15   char const* get_login() const;
16
17   void display() const; // // Nothing will be modified
18  private:
```

```
19      char *login;
20      int age, year;
21      double GPA;
22  };
23
24  void foo() {
25      Student const john {"jdoe", 20, 3, 3.10}; // read-only object
26
27      john.set_age(25); // Error, as expected.
28      john.set_year(3); // Error, as expected.
29
30      john.get_age();   // Ok, as expected.
31      john.display();   // Ok, as expected.
32  }
```

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.*

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

- The header file - contains the class definition, usually in a `.h` *interface* file.
- The implementation file - contains all of the methods [in a `.cpp`] that are declared in the *implementation* file.

Here's our simple project split into three files:

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

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

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

Summary

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. The code of complex programs is distributed over multiple files.
20. Header files contain the definitions of classes and declarations of shared constants, functions, and variables.
21. Source files contain implementations of member functions.