COMP 356
Programming Language Structures
Notes for class 6 (and for Chapter 11 of *Concepts of Programming Languages*)
Data Abstraction and Encapsulation

Acknowledgment: Section 2 of these notes is based closely on an earlier version by Prof. Tim Wahls.

1 Data abstraction in practice, for C and C++

1.1 Header and implementation files in C/C++

In C/C++ projects, it is common to place the *interface* and *implementation* of a given piece of code in two separate files. Roughly speaking, the interface consists of function signatures, while the implementation consists of function bodies. The interface goes in a *header file* (.h), and the implementation goes in an *implementation file* (.c for C; .cpp or .C for C++). Implementation files are also referred to as *source files*.

For an example, see the accompanying files Location.cpp, Location.h, CityList.cpp, CityList.h, CityListMain.cpp, and combined.cpp. The first five files here distribute code according to usual C++conventions, separating the interface and implementation of each class into a separate file and also providing a separate file for the `main()` function. The final file (combined.cpp) combines the code from all the other files, demonstrating that this is possible in C++, even though it is unconventional and undesirable.

Separating the interface and implementation makes it easier and more efficient to build and maintain large C/C++ projects. But we will ignore such issues in this course. Therefore, for simplicity, we will mostly take the unconventional approach of avoiding header files—instead, we will keep all the code for a C/C++ project in a single source file.

1.2 Inheritance in C++

Derived classes and inheritance in C++ are mostly similar to Java. But there are three major differences in the way that Java and C++ handle inheritance: (i) destructors; (ii) virtual functions, and (iii) multiple inheritance. We will cover destructors today (see Section 1.3 below). Virtual functions and multiple inheritance are covered in a future lecture.

For now, let's focus on the similarities in the way C++ and Java handle inheritance. The file `inheritance.cpp` gives a simple example of a base class and a derived class. Note the order in which constructors are called when the code is executed: base class first, then derived class. This is the same as Java, of course.

1.3 Destructors in C++

Roughly speaking, a *destructor* is the opposite of a constructor. A constructor is called when an object is created; a destructor is called when an object is destroyed. In C++, objects on the stack are destroyed when they go out of scope; objects on the heap are destroyed when they are explicitly deleted by the programmer (via the `delete` operator).

The most common use of a destructor is to deallocate all memory that has been allocated via the `new` operator during the object's lifetime. See `destructors.cpp` for an example.

In fact, Java does have an analogous concept, known as a *finalizer*. Finalizers are used to free resources that have been acquired during an object's lifetime, and which cannot be automatically

freed by Java's garbage collector. One of the difficulties in working with finalizers is that you can't predict exactly when they will be invoked. An object's finalizer is invoked when the garbage collector chooses to destroy the object, and there is typically no way of predicting when this will happen.

1.4 Templates in C++

C++ *templates* fulfill a similar function to generics in Java (e.g. `ArrayList<T>`, where `T` is a Java class). C++ permits function templates and class templates, but in this course we study only class templates. See `templates.cpp` for an example.

2 Data abstraction in theory

Abstraction means suppressing unnecessary detail to allow a higher level of understanding.

Kinds of abstraction in programming:

- **process abstraction**
  - replacing a computation with a call to a subprogram that performs the computation
  - allows readers to understand a program without keeping in mind every detail of how each subprogram is implemented
  - "client" code can use a subprogram without knowing how it is implemented
- **data abstraction**
  - hiding the representation of data and the implementation of operations on that data
  - along with proper documentation, allows client code to use a data type without knowing how it is implemented

Abstraction is the key to managing software complexity. Human beings can NOT develop and understand large software systems one line of code at a time.

2.1 Abstract Data Types

An **abstract data type** (ADT) is a data type that satisfies the following conditions:

- the declarations of the type and the operations on objects of the type are contained in the same syntactic unit (**encapsulation**). (The implementation of the operations may be in a different syntactic unit.)
- the representation of the type is hidden from client code
- client code does not need to know how operations on objects of the type are implemented
- client code can declare variables of the ADT and create instances of the ADT

Advantages of abstract data types:

- logical organization of programs
- separate compilation

- ability to change the representation without breaking client code
- increased reliability - clients can not make arbitrary changes to objects of the abstract data type, so data **invariants** can be maintained

2.2 Language Approaches to Supporting ADTs

In most modern programming languages, ADTs are provided via *classes*.

Key features of classes:

- encapsulation of data and operations
- hiding of data and operations from client code (private)
- define a single type (each class defines one ADT)
- implicit calling of constructors and destructors

2.2.1 Classes in C++

Terminology (compared and contrasted with Java):

- **data members**: fields/instance variables
- **member functions**: methods
- **members**: data members and member functions
- **destructor**: an operation that is implicitly called when the lifetime of an object ends. Often used to deallocate any heap-dynamic data members.

Other notes:

- access modifiers (`public`, `protected`, `private`) refer to clauses (sections of classes), not individual members
- instances of a C++ class can be stack-dynamic or heap-dynamic. Heap-dynamic instances are referred to by a pointer.
- member functions can be defined directly within the class (inline), or in another implementation file. Common style:
  - define the class within a **header** (.h) file. For most member functions, only the prototype (signature) will appear in the header file.
  - implement methods in an implementation (.C or .cpp) file, using the **scope resolution** operator (`::`) to indicate the class they belong to.

2.3 Encapsulation Mechanisms

Idea: large programs need more structure than just ADTs - there should be some way to organize logically related ADTs into "units".

2.3.1 Packages in Java

In Java, a package is a collection of classes. Each source file for such a class begins with:

`package` *packageName*`;`

where *packageName* is the name of the package.

Any member of a Java class (instance variable, method) declared without an access modifier (such as `public` or `private`) has package access - it is visible to all other classes in the same package. This is often useful for closely related classes - if class `LinkedList` and class `ListNode` are both defined in package `MyList`, then class `LinkedList` can use the instance variables of `ListNode` directly (if they are declared with package access), but code outside of the package can not.

Java packages are also useful for avoiding name clashes. Classes declared in a package are referred to using the package name from outside of the package, i.e.:.

```
MyList.LinkedList L = new MyList.LinkedList();
```

Another package can define a class called `LinkedList`, and the two classes can be distinguished. If there are no name clashes, then a package can be imported and names defined in the package can be used without the package name qualifier, i.e.:

```
import MyList.LinkedList;
...

  LinkedList L = new LinkedList();
```

Notes:

- a package must be defined in a directory with the same name as the package
- packages can be nested:
  - the child package is placed in a subdirectory of the parent package directory
  - a child package is specified as parent package name . child package name, i.e. `AllLists.MyList.LinkedL` if `MyList` is a child package of `AllLists`
  - child packages have no special access to parent packages (and vice versa) - package nesting is only an organizational mechanism
- the **classpath** specifies which directories java and javac search for packages and classes

2.3.2 Namespaces in C++

Namespaces in C++ provide only name management (no access control). For example:

```
namespace MyList {
  class ListNode {
    // define class here
  };

  class LinkedList {
    // define class here
  };
}
```

defines classes `ListNode` and `LinkedList` in namespace `MyList`. Code outside of the namespace can refer to names defined inside the namespace using scope resolution, i.e.:

```
MyList::LinkedList *l = new MyList::LinkedList;
```

so a large program can have multiple classes called `LinkedList`.

Notes:

- one namespace can be split across multiple files just by using another `namespace` block (with the same name)
- code within a namespace does not need to qualify the names of other classes, functions etc. defined in the same namespace
- a namespace can be imported with keyword `using`, i.e. `using namespace MyList;`

To get the effect of package access in C++, programmers can use `friend` functions and classes. For example:

```
class ListNode {
  friend class LinkedList;
  ...
};
```

Now, class `LinkedList` can access the private members of class `ListNode`, but no other class can.