CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 4
January 28, 2016

Classes

C++ version

We look at the C++ version in some detail.

This will be a whirlwind tour of **classes** in C++, which we will be covering in greater detail in the coming lectures.

Header file format

A `class` definition goes into a header file.

The file starts with **include guards**.

```
#ifndef DATAPACK_H
#define DATAPACK_H
// rest of header
#endif
```

or the more efficient but non-standard replacement:

```
#pragma once // rest of header
```

Class declaration

Form of a simple class declaration.

```
class DataPack {
  private: // ----------------------------------------------------
    // data member declarations, like struct in C
    ...

  public: // -----------------------------------------------------
    // constructor and destructor for the class
    DataPack() {...}
    ~DataPack() {...}
    }
    // ----------------------------------------------------------
    // public function methods
    ...
};
```

class DataPack

```
class DataPack {
...
};
```

defines a new class named `DataPack`.

By convention, class names are capitalized.

Note the *required* semicolon following the closing brace.

If omitted, here's the error comment:

../datapack.hpp:11: error: new types may not be defined in a return type
../datapack.hpp:11: note: (perhaps a semicolon is missing after the definition of 'DataPack')
../datapack.cpp:12: error: two or more data types in declaration of 'readData'

Class elements

- A class contains declarations for *data members* and *function members* (or *methods*).
- `int n;` declares a data member of type `int`.
- `int getN(){ return n; }` is a complete member function definition.
- `void sortData();` declares a member function that must be defined elsewhere.
- By convention, member names begin with lower case letters and are written in camelCase.

Inline functions

- Methods defined inside a class are *inline* (e.g., `getN()`).
- Inline functions are recompiled for every call.
- Inline avoids function call overhead but results in larger code size.
- `inline` keyword makes following function definition inline.
- Inline functions must be defined in the header (.hpp) file. Why?

Visibility

- The visibility of declared names can be controlled.
- `public:` declares that following names are visible outside of the class.
- `private:` restricts name visibility to this class.
- Public names define the interface to the class.
- Private names are for internal use, like local names in functions.

Constructor

A *constructor* is a special kind of method.

Automatically called whenever a new class instance is allocated.

Job is to initialize the raw data storage of the instance to become a valid representation of an initial data object.

In `dataPack` example, `store` must point to storage of `max` bytes, `n` of which are currently in use.

Constructor

```
DataPack(){
    n = 0;
    max = LENGTH;
    store = new BT[max]; cout << "Store allocated.\n";
    readData();
}
```

`new` does the job of `malloc()` in C.

`cout` is name of standard output stream (like `stdout` in C).

`<<` is output operator.

`readData()` is private function to read data set from user.

Design question: Why is this a good idea?

Destructor

A *destructor* is a special kind of method.

Automatically called whenever a class instance about to be deallocated.

Job is to perform any final processing of the data object and to return any previously-allocated storage to the system.

In `dataPack` example, the storage block pointed to by `store` must be deallocated.

Destructor

```
~DataPack(){
    delete[] store;
    cout << "Store deallocated.\n";
}
```

Name of the destructor is class name prefixed with `~`.

`delete` does the job of `free()` in C.

Empty square brackets `[]` are for deleting an array.

dataPack.cpp

Ordinary (non-inline) functions are defined in a separate *implementation file*.

Function name must be prefixed with class name followed by `::` to identify which class's member function is being defined.

Example: `DataPack::readData()` is the member function `readData()` declared in class `DataPack`.

File I/O

C++ file I/O is described in Chapter 3 of textbook. Please read it.

`ifstream infile( filename );` creates and opens an input stream `infile`.

The Boolean expression `!infile` is true if the file failed to open.

This works because of a built-in coercion from type `ifstream` to type `bool`. (More later on coercions.)

`readData()` has access to the private parts of class `dataPack` and is responsible for maintaining their consistency.

main.cpp

As usual, the header file is included in each file that needs it:
`#include "datapack.hpp"`

`banner();` should be the first line of every program you write for this course. It helps debugging and identifies your output. (Remember to modify `tools.hpp` with your name as explained in Chapter 1 of textbook.)

Similarly, `bye();` should be the last line of your program before the return statement (if any).

The real work is done by the statements `DataPack theData;` and `theData.sortData();`. Everything else is just printout.

C++ I/O

Streams

C++ I/O is done through **streams**.

Four standard streams are predefined:

- `cin` is the standard input stream.
- `cout` is the standard output stream.
- `cerr` is the standard output stream for errors.
- `clog` is the standard output stream for logging.

Data is read from or written to a stream using the input and output operators:

`>>` (for input). Example: `cin >> x >> y;`
`<<` (for output). Example: `cout << "x=" << x;`

Opening and closing streams

You can use streams to read and write files.

Some ways of opening a stream.

- `ifstream fin ( "myfile.in" );` opens stream `fin` for reading. This implicitly invokes the constructor `ifstream( "myfile.in" )`.
- `ifstream fin;` creates an input stream not associated with a file. `fin.open( "myfile.in" );` attaches it to a file.

Can also specify open modes.

To test if `fin` failed to open correctly, write `if (!fin) {...}`.

To close, use `fin.close();`.

Reading data

Simple forms. Assume `fin` is an open input stream.

- `fin >> x >> y >> z;` reads three fields from `fin` into `x`, `y`, and `z`.
- The kind of input conversion depends on the types of the variables.
- No need for format or `&`.
- Standard input is called `cin`.
- Can read a line into a buffer with `fin.get(buf, buflen);`. This function stops before the newline is read. To continue, one must move past the newline with a simple `fin.get(ch);` or `fin.ignore();`.

Writing data

Simple forms. Assume `fout` is an open output stream.

- `fout << x << y << z;` writes `x`, `y`, and `z` into `fout`.
- The kind of output conversion depends on the types of the variables or expressions..
- Standard output is called `cout`. Other predefined output streams are `cerr` and `clog`. They are usually initialized to standard output but can be redirected.
- Warning: The eclipse debug window does not obey the proper synchronization rules when displaying `cout` and `cerr`. Rather, the output lines are interleaved arbitrarily. In particular, a line written to `cerr` **after** a line written to `cout` can appear **before** in the output listing. This won't happen with a Linux terminal window.

Manipulators

Manipulators are objects that can be arguments of `>>` or `<<` but do not necessarily produce data.

Example: `cout << hex << x << y << dec << z << endl;`

- Prints `x` and `y` in hex and `z` in decimal.
- After printing `z`, a newline is printed and the output stream is flushed.

Manipulators are used in place of C formats to control input and output formatting and conversions.

End of file and error handling

I/O functions set status flags after each I/O operation.

- `bad` means there was a read or write error on the file I/O.
- `fail` means the data was not appropriate to the field, e.g., trying to read a non-numeric character into a numeric variable.
- `eof` means that the end of file has been reached.
- `good` means that the above three bits are all off.

The whole state can be read with one call to `rdstate()`.

Status functions

Functions are also provided for testing useful combinations of status bits.

- `good()` returns true if the `good` bit is set.
- `bad()` returns true if the `bad` bit is set.

  This is *not* the same as `!good()`.
- `fail()` returns true if the `bad` bit or the `fail` bit is set.
- `eof()` returns true if the `eof` bit is set.

As in C, correct end of file and error checking require paying close attention to detail of exactly when these state bits are turned on. To continue after a bit has been set, must call `clear()` to clear it.