CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 3
September 7, 2016

Insertion Sort Example

- Program specification
- Monolithic solution
- Modular solution in C
- Modular solution in C++

Classes

- Header file
- Implementation file
- Main program
- Building `InsertionSortCpp`

Insertion Sort Example

Design process: Insertion Sort

Here's a simple problem similar to what might be taught in a second programming class.

Write a C++ program to sort a file of numbers.

This is hardly a specification. A few questions immediately come to mind:

- What file?
- What kind of numbers?
- What sorting algorithm should be used?
- Where does the output go?

A more refined specification

Here's a more detailed specification. The program should:

1. Prompt the user for the name of a file containing numbers.
2. The numbers are assumed to be floating point, one per line.
3. The numbers should be sorted using insertion sort.
4. The output should be written to standard output.

A first solution

03-InsertionSortMonolith satisfies the requirements.

Characteristics:

- It's monolithic – everything is in `main()`.
- It defines `BT` to be the type of number to be sorted. The definition uses a `typedef` statement.
- It uses dynamic storage to hold the list of numbers to be sorted.
- The macro `LENGTH` gives the maximum size list that it can handle. `#define` defines it to be 20.
- It proceeds in a logical step-by-step fashion through the entire solution process.

What is wrong with this?

This code violates many of the design principles I talked about in the first two lectures:

- Lack of isolation between the parts of the code that interact with the user, manage the dynamic storage, read the file, perform the sort, and print the results.
- It is not modular.
    - Variables used by the different parts are mixed together.
    - The storage management is intertwined with the other activities.
    - I/O and computation are mixed together.
- Reuse of the sorting algorithm is surprisingly difficult because of its entanglement with the other parts of the program.

A modular solution

03-InsertionC is a more modular solution that follows many OO-design principles, *even though it is written in C*.

- `main()` sequences the steps of the solution but delegates the implementation to functions defined in `databack.h`.
- `datapack.h` declares a `stuct DataPack` that brings together the variables needed to adequately represent the data to be processed.

A modular solution (cont.)

- datapack.h also declares three functions that make use of a DataPack:
  - setup() Prompts the user for a file name, creates a DataPack, and initializes it with the data from the file.
  - printData() writes a dataPack to an output stream.
  - sortData sorts the data in a dataPack.
- datapack.c contains the implementations of these three functions.
- It also contains a *private* function readData() that does the actual user interaction for setup(). The static keyword in C restricts visibility of readData() to this one file.

C++ version

03-InsertionSortCpp is a solution written in C++ that uses many C++ features to achieve greater modularity than was possible in C.

It mirrors the file structure of the C version with the three files `main.cpp`, `datapack.hpp`, and `datapack.cpp`.

It achieves better modularity primarily by its use of **classes**. We give a whirlwind tour of classes in C++, which we will be covering in greater detail in the coming lectures.

Classes

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
    ...
    // private function methods
    ...
  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.

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 sort();` 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.

It is automatically called whenever a new class instance is created.

Its 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";
    read();
}
```

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

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

`<<` is output operator.

`read()` 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::read()` is the member function `read()` 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.)

`read()` 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.sort();`. Everything else is just printout.

Manual compiling and linking

One-line version

```
g++ -o isort main.cpp datapack.cpp tools.cpp
```

Separate compilation

```
g++ -c -o datapack.o datapack.cpp
g++ -c -o main.o main.cpp
g++ -c -o tools.o tools.cpp
g++ -o isort main.o datapack.o tools.o
```

Compiling and linking using make

The sample Makefile given in lecture 02 slide 24 is easily adapted for this project.

Compare it with the Makefile on the next slide.

Building InsertionSortCpp

```
#-----------------------------------------------------------
# Macro definitions
CXXFLAGS = -O1 -g -Wall -std=c++14
OBJ = main.o datapack.o tools.o
TARGET = isort
#-----------------------------------------------------------
# Rules
all: $(TARGET)
$(TARGET): $(OBJ)
        $(CXX) -o $@ $(OBJ)
clean:
        rm -f $(OBJ) $(TARGET)
#-----------------------------------------------------------
# Dependencies
datapack.o: datapack.cpp datapack.hpp tools.hpp
main.o: main.cpp datapack.hpp tools.hpp
tools.o: tools.cpp tools.hpp
```