CSC224: Data Structures: Tutorial1

Introducing Data Structures and Algorithms

1. Overview of Data Structures
2. Arrays
3. Ordered Arrays
4. The Bubble Sort
5. The Insertion Sort

As you start this course, you might have some questions:

- What are data structures ?
- What good will it do me to know about them?
- Why can't I use simple program features like arrays and for loops to handle my data?
- When does it make sense to apply what I learn here?

We'll also introduce some terms you'll need to know and generally set the stage for the more detailed material to follow. Finally, for those of you who have not yet been exposed to object-oriented programming (OOP), we'll briefly explain just enough about it to get you started.

Some Uses for Data Structures and Algorithms

The subjects of this tutorial are data structures and algorithms.

A *data structure* is an arrangement of data in a computer's memory (or sometimes on a disk). Data structures include linked lists, stacks, binary trees, and hash tables, among others.

Algorithms manipulate the data in these structures in various ways, such as inserting a new data item, searching for a particular item, or sorting the items. You can think of an algorithm as a recipe: a list of detailed instructions for carrying out an activity. What sorts of problems can you solve with knowledge of these topics? As a rough approximation, we might divide the situations in which they're useful into three categories:

- Real-world data storage
- Programmer's tools
- Modeling

These are not hard-and-fast categories, but they might help give you a feeling for the usefulness of this tutorial's subject matter.

Real-World Data Storage

Many of the structures and techniques you'll learn are concerned with how to handle real-world data storage. By real-world data, we mean data that describes physical entities external to the computer. Some examples are a personnel record that describes an actual human being, an inventory record that describes an existing car part or grocery item, and a financial transaction record that describes, say, an actual check written to pay the grocery bill.

A non-computer example of real-world data storage is a stack of index cards. These cards can be used for a variety of purposes. If each card holds a person's name, address, and phone number, the result is an address book. If each card holds the name, location, and value of a household possession, the result is a home inventory.

Overview of Data Structures

Another way to look at data structures is to focus on their strengths and weaknesses. This section provides an overview, in the form of a table, of the major data storage structures discussed in this tutorial. This is a bird's-eye view of a landscape that we'll be covering later at ground level, so don't be alarmed if it looks a bit mysterious.

Table 1.1 shows the advantages and disadvantages of the various data structures described in this book.

TABLE 1.1 CHARACTERISTICS OF DATA STRUCTURES

Data Structure	*Advantages*	*Disadvantages*
Array	*Quick insertion, very fast access if index known*	*Slow search, very slow deletion, fixed size*
Ordered array	*Quicker search than unsorted array*	*Slow insertion and deletion, fixed size*
Stack	*Provides last-in, first out access*	*Slow access to other items*
Queue	*Provides first-in, first-out access*	*Slow access to other items*
Linked list	*Quick insertion, quick deletion*	*Slow search*
Binary tree	*Quick search, insertion, deletion(if tree remains balanced)*	*Deletion algorithm is complex*
Hash table	*Very fast access if key known, Fast insertion*	*Slow deletion, access slow if key not known, inefficient memory usage*
Heap	*Fast insertion, deletion*	*Slow access to other items*
2-3-4 Tree	*Quick search, deletion, insertion. Trees always balanced. Similar trees*	*Complex*

Overview of Algorithms

An algorithm can be thought of as the detailed instructions for carrying out some operation. In a computer program these instructions take the form of program statements. Many of the algorithms we'll discuss apply directly to specific data structures. For most data structures, you must know how to do the following:

- Insert a new data item.
- Search for a specified item
- Delete a specified item.

You might also need to know how to *traverse* through all the items in a data structure, visiting each one in turn so as to display it or perform some other action on it.

Another important algorithm category is *sorting*. There are many ways to sort data

The concept of *recursion* is important in designing certain algorithms. Recursion involves a function calling itself.

Some Initial Definitions

Before we move on to a more detailed look at data structures and algorithms in the chapters to come, let's look at a few terms that will be used throughout this book.

Datafile

We'll use the term *datafile* to refer to a collection of similar data items. As an example, if you create an address book using the Cardfile program, the collection of cards you've created constitutes a datafile. The word *file* should not be confused with the files stored on a computer's hard disk. A datafile refers to data in the real world, which might or might not be associated with a computer.

Record

Records are the units into which a datafile is divided. They provide a format for storing information. In the Cardfile program, each card represents a record. A record includes all the information about some entity, in a situation in which there are many such entities. A record might correspond to a person in a personnel file, a car part in an auto supply inventory, or a recipe in a cookbook file.

Field

A record is usually divided into several *fields*. A field holds a particular kind of data. In the Cardfile program there are really only two fields: the index line (above the double line) and the rest of the data (below the line); both fields hold text. Generally, each field holds a particular kind of data. In Figure 1.1, we show the index line field as holding a person's name. More sophisticated database programs use records with more fields than Cardfile has. Figure 1.2 shows such a record, where each line represents a distinct field.

In a C++ program, records are usually represented by objects of an appropriate class. (In C, records would probably be represented by structures.) Individual data members within an object represent fields within a record. We'll return to this later in this hour.

Key

To search for a record within a data file you must designate one of the record's fields as a *key*. You'll search for the record with a specific key. For example, in the Cardfile program you might search in the index-line field for the key Brown. When you find the record with that key, you'll be able to access all its fields, not just the key. We might say that the key *unlocks* the entire record.

Objects in a Nutshell

The idea of objects arose in the programming community as a solution to the problems we just discussed with procedural languages. In this section, we'll discuss objects, classes, and several other topics.

Objects

Here's the amazing breakthrough that is the key to OOP: An object contains both functions and variables. A Thermostat object, for example, would contain not only furnace_on() and furnace_off() functions, but also

currentTemp and desiredTemp variables. This new entity, the object, solves several problems simultaneously. Not only does a programming object correspond more accurately to objects in the real world, it also solves the problem engendered by global data in the procedural model. The furnace_on() and furnace_off() functions can access currentTemp and desiredTemp. However, these
variables are hidden from functions that are not part of thermostat, so they are less likely to be accidentally changed by a rogue function.

Classes

You might think that the idea of an object would be enough for one programming revolution, but there's more. Early on, it was realized that you might want to make several objects of the same type. Maybe you're writing a furnace control program for an entire apartment house, for example, and you need several dozen Thermostat objects in your Program. It seems a shame to go to the trouble of specifying each one separately. Thus the idea of classes was born. A *class* is a specification—a blueprint—for one or more objects

```
THE Thermostat CLASS
class Thermostat
{
        private:
        float currentTemp();
        float desiredTemp();
        public:
        void furnace_on()
{
// function body goes here
}
void furnace_off()
{
// function body goes here
}
}; // end class Thermostat
```

The C++ keyword class introduces the class specification, followed by the name you want to give the class; here it's Thermostat. Enclosed in curly brackets are the data members and member functions (variables and functions) that make up the class. We've left out the body of the member functions; normally there would be many lines of program code for each one. C programmers will recognize this syntax as similar to that of a structure.

Object Creation

Specifying a class doesn't create any objects of that class. (In the same way specifying a structure in C doesn't create any variables.) To actually create objects in C++ you must define them as you do other variables. Here's how we might create two objects of class
Thermostat:

Thermostat therm1, therm2;
Incidentally, creating an object is also called *instantiating* it, and an object is often referred to as an *instance* of a class.

Accessing Object Member Functions

After you've specified a class and created some objects of that class, other parts of your program must interact with these objects. How do they do that? Typically, other parts of the program interact with an object's member functions, not with its data members. For example, to tell the therm2 object to turn on the furnace, we would say
therm2.furnace_on();

The dot operator is simply a period (.). It associates an object with one of its member functions (or occasionally with one of its data members). At this point we've covered (rather briefly) several of the most important features of OOP. To summarize:

● Objects contain both member functions and data members (variables).
● A class is a specification for any number of objects.
● To create an object, you must define it as you would an ordinary variable.
● To invoke a member (usually a function) for a particular object, you use the dot operator.

These concepts are deep and far-reaching. It's almost impossible to assimilate them the first time you see them, so don't worry if you feel a bit confused. As you see more classes and what they do, the mist should start to clear.

BANK.CPP

```
//bank.cpp
//demonstrates basic OOP syntax
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class BankAccount
{
private:
double balance; //account balance
public:
//-----------------------------------------------------------
BankAccount(double openingBalance) //constructor
{
balance = openingBalance;
}
//-----------------------------------------------------------
void deposit(double amount) //makes deposit
{
balance = balance + amount;
}
```

```
//-------------------------------------------------------------
void withdraw(double amount) //makes withdrawal
{
balance = balance - amount;
}
//-------------------------------------------------------------
void display() //displays balance
{
cout << "Balance=" << balance << endl;
}
}; //end class BankAccount
////////////////////////////////////////////////////////////////
int main()
{
        Bank Account    ba1(100.00); //create account
        cout << "Before transactions, ";
        ba1.display (); //display balance
        ba1.deposit(74.35); //make deposit
        ba1.withdraw (20.00); //make withdrawal
        cout << "After transactions, ";
        ba1.display (); //display balance
        return 0;
} //end main()
```

Summary

In this hour, you've learned the following:

- A data structure is the organization of data in a computer's memory (or in a disk file).
- The correct choice of data structure allows major improvements in program efficiency.
- Examples of data structures are arrays, stacks, and linked lists.
- An algorithm is a procedure for carrying out a particular task.
- Data structures can be used to build data files.
- A data file is a collection of many similar records.
- Examples of data files are address books, recipe books, and inventory records.
- Data structures can also be used as programmer's tools: they help execute an algorithm.
- A record often represents a real-world object, like an employee or a car part.
- A record is divided into fields. Each field stores one characteristic of the object described by the record.