CHAPTER 1

Introduction to Data Structures and C++

LESSON 1.1

Introduction to Data Structures

As we develop more powerful computers, our history so far has always been to use that additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater. Worse yet, as tasks become more complex, they become less like our everyday experience.

In all these cases, the large amount of information that is to be processed in some sense represents an abstraction of a part of reality. The information that is available to the computer consists of a selected set of data about the actual problem, namely that set that is considered relevant to the problem at hand, that set from which it is believed that the desired results can be derived. The data represent an abstraction of reality in the sense that certain properties and characteristics of the real objects are ignored because they are peripheral and irrelevant to the particular problem. An abstraction is thereby also a simplification of facts.

In solving a problem with or without a computer it is necessary to choose an abstraction of reality, i.e., to define a set of data that is to represent the real situation. This choice must be guided by the problem to be solved. Then follows a choice of representation of this information. This choice is guided by the tool that is to solve the problem, i.e., by the facilities offered by the computer. In most cases these two steps are not entirely separable.

When selecting a data structure to solve a problem, you should follow these steps.

1. Analyze your problem to determine the basic operations that must be supported. Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to selecting a data structure operationalizes a data centered view of the design process. The first concern is for the data and the operations to be performed on them, the next concern is the representation for those data, and the final concern is the implementation of that representation.

Today's computer scientists must be trained to have a thorough understanding of the principles behind efficient program design, because their ordinary life experiences often do not apply when designing computer programs.

Language in Data Structures

In this context, the significance of programming languages becomes apparent. A programming language represents an abstract computer capable of interpreting the terms used in this language, which may embody a certain level of abstraction from the objects used by the actual machine. Thus, the programmer who uses such a higher-level language will be freed (and barred) from questions of number representation, if the number is an elementary object in the realm of this language.

The importance of using a language that offers a convenient set of basic abstractions common to most problems of data processing lies mainly in the area of reliability of the resulting programs. It is easier to design a program based on reasoning with familiar notions of numbers, sets, sequences, and repetitions than on bits, storage units, and jumps. Of course, an actual computer represents all data, whether numbers, sets, or sequences, as a large mass of bits. But this is irrelevant to the programmer as long as he or she does not have to worry about the details of representation of the chosen abstractions, and as long as he or she can rest assured that the corresponding representation chosen by the computer (or compiler) is reasonable for the stated purposes.

The closer the abstractions are to a given computer, the easier it is to make a representation choice for the engineer or implementor of the language, and the higher is the probability that a single choice will be suitable for all (or almost all) conceivable applications. This fact sets definite limits on the degree of abstraction from a given real computer. For example, it would not make sense to include geometric objects as basic data items in a general-purpose language, since their proper representation will, because of its inherent complexity, be largely dependent on the operations to be applied to these objects. The nature and frequency of these operations will, however, not be known to the designer of a general-purpose language and its compiler, and any choice the designer makes may be inappropriate for some potential applications.

Costs and Benefits

Each data structure has associated costs and benefits. In practice, it is hardly ever true that one data structure is better than another for use in all situations. If one data structure or algorithm is superior to another in all respects, the inferior one will usually have long been forgotten.

A data structure requires a certain amount of space for each data item it stores, a certain amount of time to perform a single basic operation, and a certain amount of programming effort. Each problem has constraints on available space and time. Each solution to a problem makes use of the basic operations in some relative proportion, and the data structure selection process must account for this. Only after a careful analysis of your problem's characteristics can you determine the best data structure for the task.

Example 1.1

A company is developing a database system containing information about cities and towns in the United States. There are many thousands of cities and towns, and the database program should allow users to find information about a particular place by name (another example of an exact-match query). Users should also be able to find all places that match a particular value or range of values for attributes such as location or population size. This is known as a range query.

A reasonable database system must answer queries quickly enough to satisfy the patience of a typical user. For an exact-match query, a few seconds is satisfactory. If the database is meant to support range queries that can return many cities that match the query specification, the entire operation may be allowed to take longer, perhaps on the order of a minute. To meet this requirement, it will be necessary to support operations that process range queries efficiently by processing all cities in the range as a batch, rather than as a series of operations on individual cities.

The hash table suggested in the previous example is inappropriate for implementing our city database, because it cannot perform efficient range queries. The B+-tree which to be discussed later in this module supports large databases, insertion and deletion of data records, and range queries.

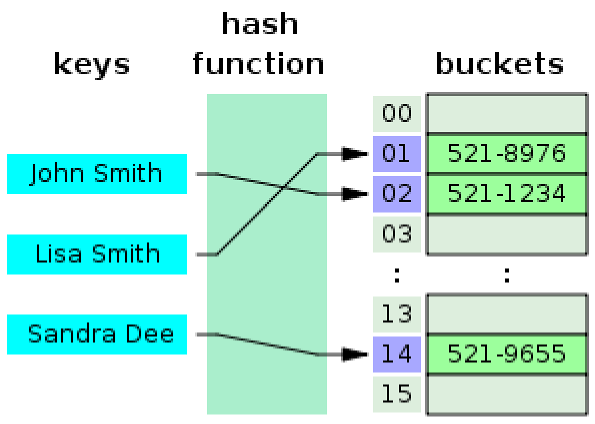

Figure 1.1.1
Hash Table

What are Data Structures?

Data structures are software artifacts that allow data to be stored, organized and accessed. They are more high-level than computer memory (hardware) and lower-level than databases and spreadsheets (which associate meta-data and meaning to the stored data). Ultimately data structures have two core functions: put stuff in, and take stuff out.

Why?

1. software is complex
    a. more than any other man made system
    b. even more so in today's highly interconnected world

2. software is fragile
    a. smallest logical error can cause entire systems to crash
3. neither you, nor your software, will work in a vacuum
4. the world is unpredictable
5. Clients are unpredictable!

Software must be correct, efficient, easy to maintain, and reusable.

LESSON 1.2

Data Structures Data Types

In computer science, an abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.

A type is a collection of values. For example, the Boolean type consists of the values true and false. The integers also form a type. An integer is a simple type because its values contain no subparts. Such a record is an example of an aggregate type or composite type. A data item is a piece of information or a record whose value is drawn from a type. A data item is said to be a member of a type.

> An **abstract stack data structure** could be defined by three operations: push, that inserts some data item onto the structure, pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and peek, that allows data on top of the structure to be examined without removal. When analyzing the efficiency of algorithms that use stacks, one may also specify that all operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs.

The term abstract data type can also be regarded as a generalised approach of a number of algebraic structures, such as lattices, groups, and rings. This can be treated as part of subject area of artificial intelligence. The notion of abstract data types is related to the concept of data abstraction, important in object-oriented programming and design by contract methodologies for software development.

Abstract Data Types Defined

An abstract data type (ADT) is the realization of a data type as a software component. The interface of the ADT is defined in terms of a type and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as encapsulation.

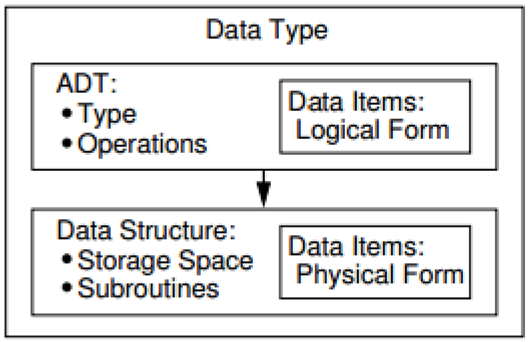

Figure 1.2.1
The relationship between data items, abstract data types, and data structures. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

A data structure is the implementation for an ADT. In an object-oriented language such as C++, an ADT and its implementation together make up a class.

Each operation associated with the ADT is implemented by a member function or method. The variables that define the space required by a data item are referred to as data members. An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

The term “data structure” often refers to data stored in a computer’s main memory. The related term file structure often refers to the organization of data on peripheral storage, such as a disk drive or CD.

Example:

The mathematical concept of an integer, along with operations that manipulate integers, form a data type. The C++ int variable type is a physical representation of the abstract integer. The int variable type, along with the operations that act on an int variable, form an ADT.

Unfortunately, the int implementation is not completely true to the abstract integer, as there are limitations on the range of values an int variable can store. If these limitations prove unacceptable, then some other representation for the ADT “integer” must be devised, and a new implementation must be used for the associated operations.

An ADT for a list of integers might specify the following operations:

- Insert a new integer at a particular position in the list.
- Return true if the list is empty.
- Reinitialize the list.
- Return the number of integers currently in the list.
- Delete the integer at a particular position in the list.

From this description, the input and output of each operation should be clear, but the implementation for lists has not been specified.

A data type is a type together with a collection of operations to manipulate the type. For example, an integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type. A distinction should be made between the logical concept of a data type and its physical implementation in a computer program. For example, there are two traditional implementations for the list data type: the linked list and the array-based list. The list data type can therefore be implemented using a linked list or an array.

Even the term “array” is ambiguous in that it can refer either to a data type or an implementation. “Array” is commonly used in computer programming to mean a contiguous block of memory locations, where each memory location stores one fixed-length data item. By this meaning, an array is a physical data structure. However, array can also mean a logical data type composed of a (typically homogeneous) collection of data items, with each data item identified by an index number. It is possible to implement arrays in many different ways.

An **abstract data type** is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Imperative

In the "imperative" view, which is closer to the philosophy of imperative programming languages, an abstract data structure is conceived as an entity that is mutable — meaning that it may be in different states at different times.

Some operations may change the state of the ADT; therefore, the order in which operations are evaluated is important, and the same operation on the same entities may have different effects if executed at different times —just like the instructions of a computer, or the commands and procedures of an imperative language. To underscore this view, it is customary to say that the operations are executed or applied, rather than evaluated. The imperative style is often used when describing abstract algorithms. This is described by Donald E. Knuth and can be referenced from here The Art of Computer Programming.

Example: abstract stack (imperative)

As another example, an imperative definition of an abstract stack could specify that the state of a stack S can be modified only by the operations

- push(S,x), where x is some value of unspecified nature; and
- pop(S), that yields a value as a result; with the constraint that
- For any value x and any abstract variable V, the sequence of operations { push(S,x); V ← pop(S) } is equivalent to { V ← x };

Since the assignment { V ← x }, by definition, cannot change the state of S, this condition implies that { V ←

pop(S) } restores S to the state it had before the { push(S,x) }. From this condition and from the properties of

abstract variables, it follows, for example, that the sequence

{ push(S,x); push(S,y); U ← pop(S); push(S,z); V ← pop(S); W ← pop(S); }

where x,y, and z are any values, and U, V, W are pairwise distinct variables, is equivalent to

{ U ← y; V ← z; W ← x }

Here it is implicitly assumed that operations on a stack instance do not modify the state of any other ADT instance,

including other stacks; that is,

- For any values x,y, and any distinct stacks S and T, the sequence { push(S,x); push(T,y) } is equivalent to {push(T,y); push(S,x) }.

A stack ADT definition usually includes also a Boolean-valued function empty(S) and a create() operation that returns a stack instance, with axioms equivalent to

- create() ≠ S for any stack S (a newly created stack is distinct from all previous stacks)
- empty(create()) (a newly created stack is empty)

- not empty(push(S,x)) (pushing something into a stack makes it non-empty)

Functional

Another way to define an ADT, closer to the spirit of functional programming, is to consider each state of the structure as a separate entity. In this view, any operation that modifies the ADT is modeled as a mathematical function that takes the old state as an argument, and returns the new state as part of the result. Unlike the "imperative" operations, these functions have no side effects. Therefore, the order in which they are evaluated is immaterial, and the same operation applied to the same arguments (including the same input states) will always return the same results (and output states).

In the functional view, in particular, there is no way (or need) to define an "abstract variable" with the semantics of imperative variables (namely, with fetch and store operations). Instead of storing values into variables, one passes them as arguments to functions.

Example: abstract stack (functional)

For example, a complete functional-style definition of a stack ADT could use the three operations:

- push: takes a stack state and an arbitrary value, returns a stack state;
- top: takes a stack state, returns a value;
- pop: takes a stack state, returns a stack state;

with the following axioms:

- top(push(s,x)) = x (pushing an item onto a stack leaves it at the top)
- pop(push(s,x)) = s (pop undoes the effect of push)

In a functional-style definition there is no need for a create operation. Indeed, there is no notion of "stack instance". The stack states can be thought of as being potential states of a single stack structure, and two stack states that contain the same values in the same order are considered to be identical states. This view actually mirrors the behavior of some concrete implementations, such as linked lists with hash cons.

Instead of create(), a functional definition of a stack ADT may assume the existence of a special stack state, the empty stack, designated by a special symbol like Λ or "()"; or define a bottom()

operation that takes no argument and returns this special stack state. Note that the axioms imply that:

- push(Λ,x) ≠ Λ

In a functional definition of a stack one does not need an empty predicate: instead, one can test whether a stack is empty by testing whether it is equal to Λ.

Note that these axioms do not define the effect of top(s) or pop(s), unless s is a stack state returned by a push.

Since push leaves the stack non-empty, those two operations are undefined (hence invalid) when s = Λ. On the other hand, the axioms (and the lack of side effects) imply that push(s,x) = push(t,y) if and only if x = y and s = t.

As in some other branches of mathematics, it is customary to assume also that the stack states are only those whose existence can be proved from the axioms in a finite number of steps. In the stack ADT example above, this rule means that every stack is a finite sequence of values that becomes the empty stack (Λ) after a finite number of pops.

By themselves, the axioms above do not exclude the existence of infinite stacks (that can be popped forever, each time yielding a different state) or circular stacks (that return to the same state after a finite number of pops). In particular, they do not exclude states as such that pop(s) = s or push(s,x) = s for some x. However, since one cannot obtain such stack states with the given operations, they are assumed "not to exist".

Advantages of Abstract Data Typing

- Encapsulation

  Abstraction provides a promise that any implementation of the ADT has certain properties and abilities; knowing these is all that is required to make use of an ADT object. The user does not need any technical knowledge of how the implementation works to use the ADT. In this way, the implementation may be complex but will be encapsulated in a simple interface when it is actually used.

- Localization of change

  Code that uses an ADT object will not need to be edited if the implementation of the ADT is changed. Since any changes to the implementation must still comply with the interface, and since code using an ADT may only refer to properties and abilities specified in the interface, changes may be made to the implementation without requiring any changes in code where the ADT is used.

- Flexibility

  Different implementations of an ADT, having all the same properties and abilities, are equivalent and may be used somewhat interchangeably in code that uses the ADT. This gives a great deal of flexibility when using ADT objects in different situations. For example, different implementations of an ADT may be more efficient in different situations; it is possible to use each in the situation where they are preferable, thus increasing overall efficiency.

Some common ADTs, which have proved useful in a great variety of applications, are

- Container
- Deque
- List
- Map
- Multimap
- Multiset
- Priority queue
- Queue
- Set
- Stack
- String
- Tree

Each of these ADTs may be defined in many ways and variants, not necessarily equivalent. For example, a stack ADT may or may not have a count operation that tells how many items have been pushed and not yet popped. This choice makes a difference not only for its clients but also for the implementation.