# Chapter 1: Key Concepts of Programming and Software Engineering

## Software Engineering

- Coding without a solution design increases debugging time - known fact!
- A team of programmers for a large software development project requires
  - An overall plan
  - Organization
  - Communication
- **Software engineering** provides techniques to facilitate the development of computer programs. It is the use of technologies and practices from computer science, project management, and other fields in order to specify, design, develop, and maintain software applications. There is no single best way to build software, no unifying theory about how to do this, but software engineers do share certain common ideas about software development.
- *Software engineering is not just for large projects.* There are principles of software engineering that are applicable to small-scale program development as well. They include methods of design, testing, documentation, and development.

## Object-Oriented Problem Solving

- **Object-oriented analysis and design** (OOA & D) is one particular method of problem solving. In OOA & D,
  - A problem solution is a program consisting of a system of interacting classes of **objects**.
  - Each object has characteristics and behaviors related to the solution.
  - A **class** is a set of objects having the same type.
- A solution is a C++ program consisting of modules, each of which contains one or more of
  - A single, stand-alone function
  - A class
  - Several functions or classes working closely together
  - Other blocks of code

## Abstraction and Information Hiding

### Abstraction

- Abstraction separates the purpose of an object from its implementation
- Specifications for each object or module are written before implementation

### Functional Abstraction

**Function abstraction** is the separation of what a program unit does from how it does it. The idea is to write descriptions of what functions do without actually writing the functions, and separate the what from the how. The client software, i.e., the software calling the function, only needs to know the parameters to pass to it, the return value, and what the function does; it should not know how it actually works. In this sense, functions become like black boxes that perform tasks.

### Data abstraction

**Data abstraction** separates what can be done to data from how it is actually done. It focuses on the operations of data, not on the implementation of the operations. For example, in data abstraction, you might specify that a set of numbers provides functions to find the largest or smallest values, or the average value, but never to display all of the data in the set. It might also provide a function that answers yes or no to queries such as, "is this number present in the set?" In data abstraction, data and the operations that act on it form an entity, an object, inseparable from each other, and the implementation of these operations is hidden from the client.

### Abstract data type (ADT)

- An **abstract data type** is a representation of an object. It is a collection of data and a set of operations that act on the data.
- An ADT's operations can be used without knowing their implementations or how data is stored, as long as the interface to the ADT is precisely specified.
- A **data structure** is a data storage container that can be defined by a programmer in a programming language. It may be part of an object or even implement an object. ***It is not an ADT!***

## Information Hiding

- **Information hiding** takes data abstraction one step further. Not only are the implementations of the operations hidden within the module, but the data itself can be hidden. The client software does not know the form of the data inside the black box, so clients cannot tamper with the hidden data.
- There are two views of a module: its **public view**, called its **interface**, and its **private view**, called its **implementation**. The parts of a class (object) that are in its public view are said to be exposed by the class.

## Principles of Object-Oriented Programming (OOP)

- Object-oriented languages enable programmers to build classes of objects
- A class combines
  - Attributes (characteristics) of objects of a single type, typically data, called data members
  - Behaviors (operations), typically operate on the data, called methods or member functions
- The principles of object-oriented programming are
  - Encapsulation
    * Objects combine data and operations

- Information hiding
  - Objects hide inner details
- Inheritance
  - Classes can inherit properties from other classes. Existing classes can be reused
- Polymorphism
  - Objects can determine appropriate operations at execution time

## Object-Oriented Analysis and Design

- **Analysis** is the process of breaking down the problem in order to understand it more clearly. It often uses domain knowledge and theory to cast the problem into a perspective that makes it easier to understand.
- The goal is to express what the solution must do, not how it must do it.
- **Object-oriented analysis** tries to cast the problem into a collection of interacting objects. It expresses an understanding of the problem and the requirements of a solution in terms of objects within the problem domain. It does not express a solution – just its requirements.
- **Object-oriented design**, in contrast, is the act of using the results of an object-oriented analysis to describe a solution as a set of objects and how they interact. The interactions between a pair of objects are called collaborations. One object sends a message to another as a means of asking the latter to perform soem service or task on its behalf.

### Uniform Modeling Language (UML)

- UML is a modeling langauge used to express an object-oriented design. Unlike programming languages and spoken languages, its elements include many pictures, or diagrams, that depict the solution. It is therefore easier for most people to understand than descriptions consistsing entirely of words.
- Object-oriented analysis can be made easier using **use cases**, which in turn lead to the creation of UML **sequence diagrams**.
- A use case consists of one or more **scenarios**, which are plain text descriptions of what the solution to the problem should do in response to actions by its user. Some scenarios describe "good" behaviors (the user enters two numbers, and the system displays their sum), whereas some represent "error" behaviors (the user enters one number, and the system displays an error message.)
- Scenarios represent what the system must do, i.e., its responsibilities, not how it should do it.
- Scenarios are converted to sequence diagrams during object-oriented design. There are specific rules for drawing sequence diagrams, such as that all objects are represented in a single row as rectangles, and that solid lines mean one thing and dashed lines another. This set of notes does not elaborate on this topic.
- The sequence diagrams are annotated with text that eventually gets converted to class methods and data, as well as ordinary variables and parameters.
- The sequence diagrams lead to the design of actual classes, in a language independent way.
- UML **class diagrams** represent the classes derived from the object-oriented design and sequence diagrams. A class diagram is a rectnagle with the name of the class and possibly its methods and data, in a very specific syntax. It does not show how the class is implemented, but only its interface.

- Class diagrams can consist of multiple classes connected by various kinds of lines indicating the relationships among the classes. A relationship might be containment (this class is a part of the other) or associations, which are less constrained and just indicate that one class might require the services of the other. Yet another relationship is called generalization, which means that one class inherits from the other.
- UML is just one approach to object-oriented analysis and design.

## The Software Life Cycle

Software follows a pattern of development known as the **software life cycle**, which is the sequence of events that take place from the moment software is conceived until it is removed from service. There are many ways to view this life cycle. This is one traditional view of the cycle:

1. **User Requirements** document
   (a) Specifies informally the user's understanding of the problem
   (b) Example: I want a program that displays a text file on the screen.
2. **Requirements Analysis** -> problem specification document
   (a) Specifies in unambiguous, complete, and consistent logical sentences, how the program is supposed to respond to inputs and other external events that happen while it is running.
   (b) Example: should the program wrap long lines, or try to justify lines on the screen; should it display tabs as tabs or replace them as sequences of blanks; what if the file does not exist, or if it is not a text file, or what if the user does not have permission to access that file?
3. **Design Analysis** -> program design document
   (a) Specifies how the program is to be constructed. What are the different classes? How do they interact with each other? What are their interrelationships? How is the development effort to be divided up, i.e., which logical elements will be in the same physical files or even within the same classes? What are the dependencies? What is the data? How will error conditions be handled? These are the types of questions answered during design analysis.
   (b) Example: The main program will prompt the user for the name of the input file and check that the user has permissions. The module that reads from the input file stream will have "switches" to control how it behaves. For example, it will have a parameter that tells it whether to convert tabs to spaces and if so, how many, and it will have a parameter that tells it whether to wrap lines, and another on whether to justify lines.
4. **Implementation** -> program source code and documentation
   (a) The program source code is the human readable executable code that gets compiled, linked, and loaded for execution. It is the result of tedious logical design and understanding.
5. **Test Design** -> Test plan document
   (a) Before you test your code, you should plan out the entire set of tests. The plan includes expected outputs, parts of the specification that the test covers, and so on. This document is required by many government agencies if you are building software for controlled applications like medical, pharmaceutical, military, or financial products.
6. **Testing and Validation** -> test cases documentation

   (a) This contains the actual test results, i.e., what really happened and what was done to fix errors if errors were found.
   (b) Example: When the program was given a file whose last character was a tab, and tab translation was turned on, it did not convert the last tab to a sequence of blanks. The error was traced to a loop condition that had the wrong comparison operator.
   (c) If tests fail, return to whichever step must be repeated. (E.g., sometimes it is a design flaw or a specification flaw)

7. **Production** -> released software
   (a) The software is placed into production, so it is now in use by the user, and exposed to potential risks and losses.

8. **Maintenance** -> changes in software
   (a) The users discover bugs or weaknesses such as confusing interfaces or hard to understand forms and these are reported and eventually fixed.

## Software Quality

- The "goodness" of software is its quality. The factors that contribute to the worth or value of software from a user's perspective include:
  - Cost, including cost of acquisition, efficiency, resource consumption
  - Correctness or reliability
  - Ease of use
- From the developer's perspective, the factors are:
  - Cost of development
  - Correctness or reliability
  - Modularity
  - Modifiability
  - Readability and style
  - Quality of documentation
  - Robustness

### Software Correctness and Reliability

- Programs must be correct. They often are not. When they are not, we can measure their relative correctness by their software reliability, which is a statistical assessment of the probability that they will be correct in an arbitrary run. Measuring reliability is a complicated matter, subject to much error.
- In general, software must undergo a rigorous development process, including testing, debugging, and verification when possible. Verification is hard; testing is something anyone can and should do.
- Verification is a mathematical proof that software is correct. In general, it is impossible to prove that software is correct, for the same reason that it is impossible to prove that a program is guaranteed never to enter an infinite loop. It is theoretically impossible. In spite of this, many program fragments and pieces of code can be proved to do what they are intended to do. This is through the use of pre-conditions, post-conditions, and loop invariants. All of these are logical statements involving the relationships of program variables to each other or to fixed constants. The following is a trivial example:

```
int x,y;
y = 0;
x = 2;
// y == 0 && x == 2     is an assertion that is true here
```

- Here is a slightly more interesting example:

```
x = 0;
cin >> n;
if ( n < 0 )
    n = -n;
// n >= 0      an assertion about n
while ( x < n )
    x = x + 1;
// x == n   an assertion about x and n
```

- After a loop it is always true that the loop entry condition is false, so its negation is always true!
- **Loop invariants** are special assertions that are true at specific points in the execution of the program. Specifically, a loop invariant is true
  1. Immediately preceding the evaluation of the loop entry condition before the loop is ever entered, and
  2. Immediately preceding the evaluation of the loop entry condition after each execution of the loop body.
- In other words, just at the point that the loop entry condition is to be evaluated, no matter when that happens, the loop invariant must be true, regardless of how many times the body has been executed. If an assertion is true at this point, it is a loop invariant.
- Some invariants are useful and others are useless. Here are examples:

```
int sum = 0;
int j = 0;
while ( j < n) //  j >= 0    true but not very useful
{
j++;
    sum = sum + j;
}
```

- It is true that j >= 0 but it does not help us in understanding what the loop does. This is better:

```
int sum = 0;
int j = 0;
while ( j < n) //  sum >= j    better but still not very useful
{
    j++;
    sum = sum + j;
}
```

- This is a little more interesting but still not useful enough. It is a little harder to prove that this is true. First, it is true before the loop is ever entered because `sum` and `j` are each 0. Second, if it is true when the loop is entered, it is true the next time the condition is evaluated because `sum` is always increased by the value of `j`, so it has to be at least `j`. This is what we really need:

```
int sum = 0;
int j = 0;

while ( j < n)  //  sum == 0 + 1 + 2 + ... + j
{
    j++;
    sum = sum + j;
}
```

- This one is strong enough to prove that the loop computes the *arithmetic sum* of the numbers from 1 to n. First we prove that the loop invariant is true by using mathematical induction on the number of times the loop body is entered.

  *Proof.* If the loop is never entered, then □

  ```
  sum = j = 0
  ```

  so it is true. Assume that it is true before the loop is entered the kth time. Then `j` has the value k-1 and by assumption,

  ```
  sum = 1 + 2 + ... + (k-1).
  ```

  After incrementing `j`, `j` has the value k. After the assignment to `sum`,

  ```
  sum = 1 + 2 + 3 + ... + k
  ```

  so

  ```
  sum = 1 + 2 + ... + j
  ```

  remains true. By induction on k it follows that it is true for all values of `j`. Since the loop invariant is true when the loop ends and since `j` == n when the loop ends, it follows that `sum` = 1 + 2 + ... + n.

## The Interface as a Contract

- The **pre-condition** of a function is a logical condition that must be satisfied just prior to execution of the function.
- The **post-condition** of a function is a logical condition that must be true when the function terminates.
- Pre- and post-conditions together form a **contract** that states that if the pre-conditions are true, then after the function executes, the post-conditions will be true.
- The **signature** of a function is what some people call the prototype. It includes
  - return value type and qualifiers (e.g., const )
  - function name
  - argument list with types and qualifiers
- The interface to a class or a module is also a contract, if every function has been given pre-and post-conditions in the interface.

**Example**

First attempt at writing a contract for a sorting function:

```
sort(int anArray[], int num);
// Sorts an array.
// Precondition: anArray is an array of num  integers; num > 0.
// Postcondition: The integers in anArray are  sorted.
```

Second refinement at writing a contract for sorting function:

```
void sort(/*inout*/ int anArray[], /*in */ const int num);
// Sorts anArray into ascending order.
// Precondition: anArray is an initialized array of num integers;
// 1 <= num <= MAX_ARRAY, where
// MAX_ARRAY is a global constant that specifies
// the maximum size of anArray.
// Postcondition: anArray[0] <= anArray[1] <= ...
// <= anArray[num-1], num is unchanged.
```

## Modularity

- The modularity of software is measured by its cohesion, coupling, and completeness.
- The **cohesion** of a module is the degree to which the parts of the module are related.
  - A highly cohesive module performs one well-defined task
  - High cohesion implies that it contains only methods and data that are related to its narrow purpose.
- The **coupling** of a set of modules is the degree to which the modules depend on each other.
  - Modules with low coupling are almost independent of one another
  - Low coupled modules are
    - Easier to change: A change to one module won't affect another
    - Easier to understand
  - Coupling cannot be and should not be eliminated entirely
- Classes should be easy to understand, and have as few methods as possible (idea of minimality). but should be **complete** – providing all methods that are necessary.

## Documentation

- Documentation is critical for all programs, because
  - Other people have to read your program and understand it, including your supervisor or teacher or colleagues, or even a client.
  - You may have to revise a program you did a year ago and without good documentation you will not remember what it does or how it works.
  - Government and other agencies will require certain software to be properly documented. Software in various products requiring certifications needs proper documentation.
- What is proper documentation? It must include, at a minimum:

1. a preamble for every file, as described in the Programming Rules document.
2. pre-and post-conditions for all function prototypes, including descriptions of each parameter
3. descriptions of how error conditions are handled,
4. a revision history for every file (list of dates of modifications, who made them, and what they fixed or changed),
5. data dictionary (descriptions of most variables)
6. detailed descriptions of all complex algorithms.

- This is an example of a suitable preamble:

```
/******************************************************************************
Title :        draw_stars.c
Author :       Stewart Weiss
Created on :   April 2, 2010
Description :  Draws stars of any size in a window, by dragging the mouse to
               define the bounding rectangle of the star
Purpose :      Reinforces drawing with the backing-pixmap method, in which the
               application maintains a separate, hidden pixmap that gets drawn
               to the drawable only in the expose-event handler. Introduces the
               rubberbanding technique.
Usage :        draw_stars
               Press the left mouse button and drag to draw a 5-pointed star
Build with :   gcc -o drawing_demo_03 draw_stars.c \
                  `pkg-config --cflags --libs gtk+-2.0`
Modifications :
******************************************************************************/
```

## Readability and Style

- Programs should be **readable**. Factors contributing to readability include:
- The systematic use of white space (blank lines, space characters, tabs) to separate program sections and keep related code close together.
- A systematic identifier naming convention for program entities such as variables, classes, functions, and constants.
- Using lines of asterisks or other punctuation to group related code and separate code sections from each other.
- Indentation to indicate structural elements.

## Other Stylistic Tips

- Use global constants for all program parameters and numeric and string constants, such as

```
int WINWIDTH = 420 // width of our window
int WINHEIGHT = 400 // height of our window
```

- Use `define` macros for constants and functions also:

```
#define EPSILON 1.0e-15
#define ALMOSTZERO(x) (((-EPSILON)<(x))&&((x)<(EPSILON)))
```

- Try to design general solutions, not specific ones, to make functions very general.

## Modifiability

- The modifiability of a program is the ease with which it can be modified. If a modification requires making changes to many parts of a program, it is not easy and can introduce errors when something is overlooked. A good program is one in which a simple modification requires only a change to one or two places in the program.
- Examples of things that make a program more modifiable include
  - Using named constants and macros instead of numeric and string literals.
  - Using `typedef` statements to define frequently used structured types.
  - Creating functions for common tasks.
  - Making modules very cohesive.
  - Keeping coupling low.

## Ease of Use

- In an interactive program, the program should prompt the user for input in a clear manner.
- A program should always echo its input.
- The output should be well labeled and easy to read.

## Robustness

- **Robust** (fail-safe) programs will perform reasonably no matter how anyone uses them. They do not crash. They do not produce nonsense, even when the user misuses them. Examples of things to do to make programs robust:
  - Test if input data, including program command line arguments, is invalid before trying to use it.
  - Prevent the user from entering invalid data when possible.
  - Check that all function parameters meet preconditions before using them.
  - Handle all error conditions that can arise, such as missing files, bad inputs, wrong values of variables, and so on, and exit gracefully when the program cannot continue.

# Introduction to Recursion

## Recursion

Recursion is a powerful tool for solving certain kinds of problems. Recursion breaks a problem into smaller problems that are identical to the original, in such a way that solving the smaller problems provides a solution to the larger one.

It can be used to define mathematical functions, languages and sets, and algorithms or programming language functions. It has been established that these are essentially equivalent in the following sense: the set of all functions that can be computed by algorithms, given some reasonable notion of what an algorithm is, is the same as the set of all functions that can be defined recursively, and each set (or language) for which membership can be determined by an algorithm corresponds to a function that can be defined recursively.

We are interested here mostly in the concepts of recursive algorithms and recursion in programming languages, but we also informally introduce recursive definitions of functions.

## Recursive Algorithms

### Example: The Dictionary Search Problem

Suppose we are given a problem to find a word in a dictionary. This is known as a **dictionary search**. Suppose the word is "yeoman". You could start at the first page and look for it and then try the second page. then the third, and so on, until finally you reach the page that contains the word. This is called a **sequential search**. Of course no one in their right mind would do this because everyone knows that dictionaries are sorted alphabetically, and that therefore there is a faster way that takes advantage of the fact that they are sorted. A **dictionary** by definition has the property that the words are listed in it in alphabetical order, which in computer science means it is a sorted list.

One more efficient solution might be to use **binary search**, which is described by the following *recursive algorithm*:

```
1 if ( the set of pages is just one page )
2     scan the page for the word
3 else {
4     open the dictionary to the page halfway between the first and last pages
5     of the set of pages being searched;
6     compare the word to the word at the top of the page;
7     if ( the word precedes the index word on the page alphabetically )
8         search the lower half of the set of pages using this same algorithm
9     else
10        search the upper half of the set of pages using this same algorithm
  }
```

The recursion in this algorithm occurs in lines 8 and 10, in which the instructions state that we must use this algorithm again, but on a smaller set of pages. The algorithm basically reduces the problem to one in which it compares the word being sought to a single word, and if it is "smaller", it looks for the word in the first half, and if it is larger, it looks in the second half. When it does this "looking again", it repeats this exact logic. This approach to solving a problem by dividing it into smaller identical problems and using the results of conquering the smaller problems to conquer the large one is called **divide-and-conquer**. Divide-and-conquer is a problem-solving **paradigm**, menaing it is a model for solving many different types of problems.

The binary search algorithm will eventually stop because each time it checks to see how many pages are in the set, and if the set contains just one page, it does not do the "recursive part" but instead scans the page, which takes a finite amount of time. It is easier to see this if we write it as a pseudo-code function:

```
void  binary_search( dictionary_type dictionary,  word keyword ) {
    if ( the set of pages in dictionary has size = 1  ) {
        // This is called the base case
        scan the page for keyword;
        if ( keyword is on the page )
            print keyword's definition;
        else
            print "not in dictionary";
    }
    else { // the size > 1
        let middlepage be the page midway between the first and last pages
        of the set of pages being searched; // e.g., middlepage=(first+last)/2
        compare keyword to indexword at the top of middlepage;
        if ( keyword < indexword )
            // recursive invocation of function
            binary_search( lower half of dictionary, keyword );
        else
            // recursive invocation of function
            binary_search( upper half of dictionary including middlepage, keyword );
    }
}
```

**Observations**

1. This function calls itself recursively in two places.
2. When it calls itself recursively, the size of the set of pages passed as an argument is at most one-half the size of the original set.
3. When the size of the set is 1, the function terminates without making a recursive call. This is called the base case of the recursion.
4. Since each call either results in a recursive call on a smaller set or it terminates without making a recursive call, the function must eventually terminate in the base case code.
5. If the keyword is on the page checked in the base case, the algorithm will print its definition, otherwise it will say it is not there. This is not a formal proof that it is correct!

If a recursive function has these two properties:

- that each of its recursive calls diminishes the problem size, and
- that the function takes a finite number of steps when the problem size is less than some fixed constant size,

then it is guaranteed to terminate eventually. Later we will see a more general rule for guaranteeing that a recursive function terminates.

## Example: The Factorial Function

Although the factorial function ($n!$) can be computed by a simple iterative algorithm, it is a good example of a function that can be computed by a simple recursive algorithm. As a reminder, for any positive integer $n$, *factorial(n)*, written mathematically as $n!$ is the product of all positive integers less than or equal to $n$. It is often defined by $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. The "..." is called an **ellipsis**. It is a way of avoiding writing what we really mean because it is impossible to write it since the number of numbers between $(n-2)$ and 2 depends on the value of $n$. The reader and the author both agree that the ellipsis means "and so on" without worrying about exactly what "and so on" really means.

From the definition of $n!$ we have

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

and

$$(n-1)! = (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

By substituting the left-hand side of the second equation into the right-hand side of the first, we get

$$n! = n \cdot (n-1)!$$

This would be a circular definition if we did not create some kind of stopping condition for the application of the definition. In other words, if we needed to find the value of10!, we could expand it to $10 \cdot 9!$ and then $10 \cdot 9 \cdot 8!$ and then$10 \cdot 9 \cdot 8 \cdot 7!$ and so on, but if we do not define what 0! is, it will remain undefined. Hence, this circularity is removed by defining the base case, $0! = 1$.

The definition then becomes:

$$n! \quad = \quad \begin{cases} 1 & if\ n = 0 \\ n \cdot (n-1)! & if\ n > 0 \end{cases}$$

*This is a recursive definition of the factorial function.* It can be used to find the value of $n!$ for any non-negative number $n$, and it leads naturally to a recursive algorithm for computing $n!$, which is written in C below.

```
int factorial( int n)
    /*
    Precondition:  n >= 0
    Postcondition: returns  n!
    */
    {
        if ( 0 == n)
            return 1;
        else
            return n * factorial(n-1);
    }
```

**Observations**

1. This does not result in an infinite sequence of calls because eventually the value passed to the argument of factorial is 0, if it is called with $n >= 0$, because if you "unwind" the recursion, you see each successive call is given an argument 1 less than the preceding call. When the argument is 0, it returns a 1, which is the base case and stops the recursion.

2. This function does not really compute $n!$ because on any computer, the number of bits to hold an `int` is always finite, and for large enough $n$, the value of $n!$ will exceed that largest storable integer. For example; $13! =$ 6,227,020,800 which is larger than the largest `int` storable on a 32-bit computer.

**Food For Thought**

There are elementary functions of the non-negative integers that are so simple that they do not need to be defined recursively. Two examples are

$$\begin{aligned} n(x) &= 0 \\ s(x) &= x+1 \end{aligned}$$

The first has the value zero for all numbers, and the second is the successor of the number. Consider this function, defined recursively:

$$\begin{aligned} a(x,0) &= x \\ a(x,y+1) &= s(a(x,y)) = a(x,y)+1 \end{aligned}$$

What does it compute?

## Tracing Recursion: The Box Method

It is hard to trace how a recursive function works. You have to be systematic about it. There are a few established systems for doing this. One is called the **box method**. It is a visual way to trace a recursive function call.

The box method is a way to organize a trace of a recursive function. Each call is represented by a box containing the value parameters of the function, the local variables of the function, a place to store the return value of the function if it has a return value, placeholders for the return values of the recursive calls (if any), and a place for the return address. The steps are as follows.

**Steps**

1. Label each recursive call in the function (e.g., with labels like 1,2,3,... or A,B,C,...). When a recursive call terminates, control returns to the instruction immediately following one of the labeled points. If it returns a value, that value is used in place of the function call.

2. Create a box template, containing

   (a) a placeholder for the value parameters of the parameter list,

   (b) a placeholder for the local variables,

   (c) a placeholder for each value returned by the recursive calls from the current call,

   (d) a placeholder for the value returned by the function call.

3. Now you start simulating the function's execution. Write the instruction that calls the recursive function with the given arguments. For example, it might be

   cout << factorial(3);

4. Using your box template, create a box for the first call to the function. Draw an arrow from the instruction you wrote in step 3 to this first box.
5. Execute the function by hand, updating values of local variables and reference parameters as needed. For each recursive call that the function makes, create a new box for that call, with an arrow from the old box to the box for the called function. Label the arrow with the label of the function being called.
6. Repeat step 5 for each new box.
7. Each time a function exits, if it has a return value, update the value in the box that called it, i.e., the one on the source side of the arrow, and then cross off the exiting box. Resume execution of the box that called the function at the instruction immediately following the label of the arrow.

We can trace the factorial function with this method to demonstrate the method. The factorial function has a single value parameter, n, and no local variables. It has a return value and a single recursive call. Its box should be

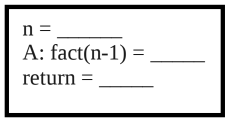


Figure 1: Factorial Box Template

There are three placeholders, one for n, one for the return value of the recursive call, which is labeled A, and one for the return value of the function. The name of the function is abbreviated.

We trace the function for the call when the argument is 3. The figure below illustrates the sequence of boxes. Each row represents a new step in the trace. The first row shows the initial box. The value of n is 3. The other values are unknown. The function is called recursively so the next line shows two boxes. The box in bold is the one being traced. In that box, n=2, since it was called with n-1. That function calls factorial again, so in the next line there are three boxes. Eventually n becomes 0 in the fourth line. It does not make a recursive call. Instead it returns 1 and the box is deleted in the next line and the 1 sent to the fact(n-1) placeholder of the box that called it. This continues until the return values make their way back to the first box, which returns it to the calling program.

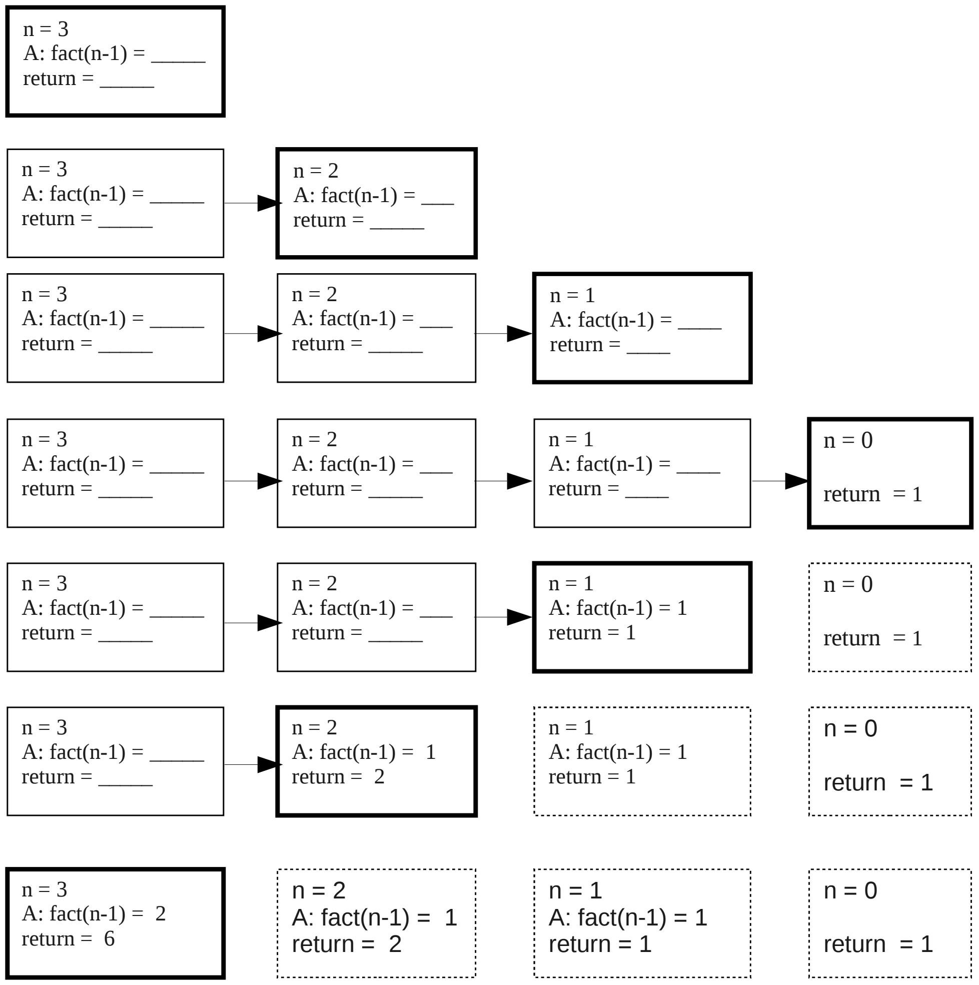


Figure 2: Box Trace of Factorial Function

## Other Examples

### Fibonacci Numbers

Recursion is not usually the most efficient solution, although it is usually the easiest to understand. One example of this is the Fibonacci sequence. The Fibonacci numbers are named after Leonardo de Pisa, who was known as Fibonacci. He did a population study of rabbits in which he simplified how they mated and how their population grew. In short, the idea is that rabbits never die, and they can mate starting at two months old, and that at the start of each month every rabbit pair gives birth to a male and a female (with very short gestation period!)

From these premises, it is not hard to show that the number of rabbit pairs in month 1 is 1, in month 2 is also 1 (since they are too young to mate), and in month 2, 2, since the pair mated and gave birth to a new pair. Let$f(n)$ be the number of rabbits alive in month $n$. Then, in month $n$, where $n > 2$, the number of pairs must be the number of pairs alive in month $n-1$, plus the number of new offspring born at the start of month $n$. All pairs alive in month $n-2$ contribute their pair in month $n$, so there are$f(n-1)+f(n-2)$ rabbit pairs alive in month $n$. The recursive definition of this sequence is thus

$$f(n) = \begin{cases} 1 & if\ n \le 2 \\ f(n-1)+f(n-2) & if\ n > 2 \end{cases}$$

This will generate the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so on. A recursive algorithm to compute the nth Fibonacci number, for $n > 0$, is given below, written as a C/C++ function.

```
int fibonacci (int n)
{
    if ( n <= 2 )
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Although this looks simple, it is very inefficient. If you write out a box trace of this function you will see that it leads to roughly $2^n$ function calls to find the $n^{th}$ Fibonacci number. This is because it computes many values needlessly. There are far better ways to compute these numbers.

## Choosing k Out of n Objects

A well-known and common combinatorial (counting) problem is how many distinct ways there are to pick $k$ objects from a collection of $n$ distinct objects. For example, if I want to pick 10 students in the class of 30, how many different sets of 10 students can I pick? I do not care about the order of their names, just who is in the set. Let $c(n,k)$ represent the number of distinct sets of $k$ objects out of a collection of $n$ objects.

The solution can be difficult to find with a straight-forward attack, but a recursive solution is quite simple. Let me rephrase the problem using the students in the class. Suppose I single out one student, say student X. Then there are two possibilities: either X is in the group I choose or X is not in the group.

How many solutions are there with X in the group? Since X is in the group, I need to pick $k-1$ other students from the remaining $n-1$ students in the class. Therefore, there are $c(n-1,k-1)$ sets that contain student X.

What about those that do not contain X? I need to pick $k$ students out of the remaining $n-1$ students in the class, so there are $c(n-1,k)$ sets.

It follows that

$$c(n,k) = c(n-1,k-1) + c(n-1,k)$$

when $n$ is large enough. Of course there are no ways to form groups of size $k$ if $k > n$, so $c(n,k) = 0$ if $k > n$. If $k = n$, then there is only one possible group, namely the whole class, so $c(n,k) = 1$ if $k = n$. And if $k = 0$, then there is just a single group consisting of no students, so $c(n,k) = 1$ when $k = 0$. In all other cases, the recursive definition applies. Therefore the recursive definition with its base cases, is

$$c(n,k) = \begin{cases} 1 & k = 0 \\ 1 & k = n \\ 0 & k > n \\ c(n-1,k-1)+c(n-1,k) & 0 < k < n \end{cases}$$

Once again it is easy to write a C/C++ function that computes this recursively by applying the definition:

```
int combinations (int n, int k)
{
    if ( k == 0 || k == n )
        return 1;
    else if ( k > n )
        return 0;
    else
        return combinations(n-1, k-1) + combinations(n-1, k);
}
```

The interesting question is how to show that this always terminates. In each recursive call, $n$ is diminished. In some, $k$ is not diminished. Therefore, eventually either $k >= n$ or $k = 0$.

In any case, this is again a very inefficient way to compute c(n,k) and it should not be used.

## Binary Search Revisited

The binary search algorithm was presented in pseudo-code earlier. Now we can work out some of the programmatic details.

- First we consider an arbitrary array, not a dictionary with pages and words on pages. The algorithm is given an array of values.
- Second we remove printing from the algorithm. An algorithm should return its results to its caller, not print them on a device. *In general, functions whose purpose is not to perform I/O should not perform any I/O, as this makes them less portable, cohesive, and reduces their performance.* The return value should be either the index in the array where the item is found or an indication that it is not in the array at all. Since array indices are always non-negative, we can use -1 to indicate that the search failed.
- Third is the issue of how to find the middle of the array. The middle is the index halfway between the top index and the bottom index of the part of the array being searched. Since the part of the array being searched must vary depending on the results of comparisons, the top and bottom of the search range will be parameters of the function.

Putting this together, the algorithm becomes

```
int  binary_search( const ordered_type theArray[],
                    int bottom,
                    int top,
                    ordered_type keyword )
{
    if ( bottom > top ) {
        // the function was called with an empty range
        return -1;
    else {
        // find the middle index
        int middle = ( top + bottom ) /2;
        // compare keyword to value at the middle
        if ( keyword < theArray[middle] )
            // keyword is not in the upper half so try again in lower half
            return binary_search( theArray, bottom, middle-1, keyword );
        else if ( keyword > theArray[middle] )
            // keyword is not in the lower half so try again in upper half
```

```
                return binary_search( theArray, middle+1, top, keyword );
            else
                // keyword >= theArray[middle] && keyword <= theArray[middle],
                // so keyword == theArray[middle] and we found it
                return middle;
        }
    }
```

**Notes.**

1. The array parameter is declared constant in order to prevent the code from accidentally modifying it, since it is passed by reference in C++ and passed as a pointer in C.
2. The type of the array must be a type for which comparison operations are defined; the type `ordered_type` represents an arbitrary ordered type such as int, char, or double.
3. The comparisons are ordered so that first it checks if the keyword is less than the array element, then larger, and if both fail, it must be equal. This is more efficient than checking equality first. Why?

The prototype and its contract, using Doxygen-style comments, are

```
/**
 * @precondition 0 <= bottom && top < ARRAYSIZE && for every i 0 <= i <= top-1
 *               theArray[i] <= theArray[i+1]  && ARRAYSIZE is the size of the array
 * @postcondition none (the array is unchanged )
 * @param  theArray  the array to search
 * @param  bottom    low index of range to search in the array
 * @param  top       high index of range to search in the array
 * @param  keyword   value to look for (the search key)
 * @return if keyword is in the array between bottom and top, its index
 *         otherwise -1
 */
int  binary_search( const ordered_type theArray[],
                    int bottom,
                    int top,
                    ordered_type keyword ) ;
```

**Exercise 1.** Suppose the array contains the integers 5, 10, 16, 17, 19, 30, 32, 33, 34, 67, 68, 69, 81, 83, 87, 91, 92 and suppose the keyword is 10.

1. What is the set of array values against which the keyword will be compared?
2. How many comparison operations will be executed?
3. What if the keyword is 3? What is the sequence of array values against which it will be compared, and how many comparisons are executed?

## Towers of Hanoi

The last problem we will consider is the famous *Towers of Hanoi* problem. You are given three pegs, labeled A, B, and C. Each peg can hold $n$ disks. Initially, $n$ disks of different sizes are arranged on peg A in such a way that above any disk are only smaller disks. The largest is therefore at the bottom of the pile and the smallest at the top. The problem is to devise an algorithm that will move all of the disks to peg B moving one disk at a time subject to the following two rules:

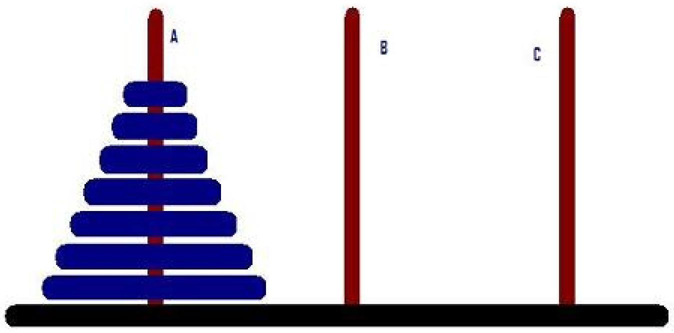


Figure 3: Towers of Hanoi

- Only the top disk can be moved from its stack; and
- A larger disk may never be on top of a smaller disk.

This is a problem with no obvious, simple, non-recursive solution, but it does have a very simple recursive solution.

1. Ignore the bottom disk and solve the problem for $n-1$ disks, moving them to peg C instead of peg B.
2. Move the bottom disk from peg A to peg B.
3. Solve the problem for moving $n-1$ disks from peg C to peg B.

Now all disks will be on peg B in the correct order.

If we write `towers(count, source, destination, spare)` represent the algorithm that moves count many disks from source to destination using the spare as needed, subject to the rules above, then the algorithm we just described can be written as follows:

```
towers(n-1, A, C, B);
towers(1, A, B, C);
towers(n-1, C, B, A);
```

It is astonishingly simple. The base case is when there is a single disk. Putting this together, we have

```
typedef char  peg;
void towers(int count, peg source, peg dest, peg spare)
{
    if ( count == 1 )
        cout << “move the disk from peg” << source << “ to peg “ <<  dest << endl;
    else {
       towers(count - 1, source, spare, dest);
       towers(1, source, dest, spare);
       towers(count - 1, spare, dest, source );
    }
}
```

This is an example of a recursive algorithm with three recursive calls. In each call the problem size is smaller. The problem size is the number of disks. The recursion stops when the problem size is 1.

What is not clear is how many steps this takes. If we were to measure the steps by how many times a disk is moved from one peg to another, then the interesting question is how many moves this makes when given an initial set of N disks on a peg. We will return to this when we discuss recurrence relations a bit later.

**Exercise 2.** Write up this algorithm and run it to verify that it works. Add statements to see how many recursive calls were made. Then, add the pre- and post-conditions to make the contract for the function's prototype.

# Chapter 3: Data Abstraction

## 1 Abstract Data Types

In Chapter 1 it was stated that:

- Data abstraction separates *what* can be done to data from *how* it is actually done.
- An abstract data type (ADT) is a collection of data and a set of operations that act on the data.
- An ADT's operations can be used without knowing their implementations or how the data is stored, as long as the interface to the ADT is precisely specified.
- A data structure is a data storage container that can be defined by a programmer in a programming language. It may be part of an object or even implement an object, but **it is not an ADT*!***

A soft drink vending machine is a good analogy. From a consumer's perspective, a vending machine contains soft drinks that can be selected and dispensed when the appropriate money has been inserted and the appropriate buttons pressed. The consumer sees its interface alone. Its inputs are money and button-presses. The vending machine outputs soft drinks and change. The user does not need to know how a vending machine works, only what it does, and in this sense a vending machine can be viewed as an ADT.

The vending machine company's support staff need to know the vending machines internal workings, its data structure. The person that restocks the machine has to know the internal structure, i.e., where the spring water goes, where the sodas go, and so on. That person is like the programmer who implements the ADT with data structures and other programming constructs.

The ADT is sometimes characterized as a wall with slits in it for data to pass in and out. The wall prevents outside users from accessing the internal implementation, but allows them to request services of the ADT.

ADT operations can be broadly classified into the following types:

- operations that add new data to the ADT's data collection,
- operations that remove data from the ADT's data collection,
- operations that modify the data in the ADT's collection,
- operations to query the ADT about the data in the collection.

## 2 UML Syntax

We will occasionally use the Uniform Modelling Language syntax for describing the interfaces to an ADT. This syntax is relatively simple.

To describe the ADT's **attributes** (which will become data members of a class), use the syntax

*[visiblity] name [:type] [=defaultValue] [{property}]*

where:

- square brackets `[]` indicate optional elements
- *visibility* is one of the symbols `+`, for public accessibility, `-`, for private accessibility, or `#`, for protected accessibility. If this element is not included, the *visibility* defaults to private access.
- *name* is the attribute's name.
- *type* is the data type of the element.
- *defaultValue* is the element's default value. It will not have a default value if this is omitted.
- *property* is one of the values `changeable`, indicating an attribute that can be modified, such as a variable, or `frozen`, indicating a constant value.

**Examples**

```
-count: integer  {changeable}      // a private integer variable
-count: integer                    // same as above - changeable is the default
+MaxItems : integer = 100 {frozen} // a public constant, defaulting to 100
-name: string                      // a private string variable
```

The **operations** of an ADT are described by the following syntax:

*[visiblity] name ([parameter_list]) [:type] [{property}]*

where:

- square brackets `[]` indicate optional elements
- *visibility* is the same as for attributes except that the default is public access.
- *name* is the operation's name.
- *type* is the return value of the operation. It can be `void` or nothing if the operation does not return a value.
- *property* is usually either omitted or is the value `query`, which means that the operation does not modify any attributes of the object on which this operation is called.
- The parameter list is either empty (and the parentheses are not omitted – they are outside of the square brackets on purpose ) or it is a comma-separated list whose elements are of the form

  *[direction] name:type [=defaultValue]*

  where

  – square brackets are again optional.
  – *direction* is one of the words `in`, meaning the parameter is an input to the function, `out`, meaning it is an output, or `inout`, meaning it is both.
  – *type* is the parameter's type and is mandatory.
  – an optional default value can be specified as an argument if there is no actual argument in the call.

**Examples**

```
+insert( in position_to_insert: integer,
         in new_item: list_item_type,
         out success: boolean): void

+fill( inout input_stream: input_file_stream,
       in number_to_read: integer,
       out success: boolean): void
```

Both of the above operations are public and return nothing. Both also have a third parameter for indicating the success of the operation. It is a style of coding to put each parameter on a separate line. This makes them easier to read.

## 3 Specification of an ADT

The first step in any object-oriented solution to a problem is to specify the ADTs that are in it. Each module's specification must be written before the module itself.

We will use the example of a list to demonstrate how to specify an ADT. Lists are ubiquitous containers. People use lists every day in their lives: shopping lists, lists of assignments to finish for school, lists of people to call, lists of movies to watch, and so on. Computer scientists use lists in a multitude of ways, such as:

- lists of blocks allocated to files in a file system
- lists of active processes in a computer system
- lists of memory blocks to be written to disk after a write operation to them
- lists of atm transactions to be performed
- lists of packets waiting to be routed
- polynomials (as lists of terms)

What then is a list, assuming it is not empty? What characterizes a list? In other words, what makes something a list? Obviously it is a *collection of things*, but there are many different types of collections of things. The defining property of a list is that it is **sequential**; it is a collection of items with the property that it has a first item and a last item (which may be the same item) and that each item that is not the first has a unique predecessor and each item that is not the last has a unique successor.

This is a characterization of the data in the list, but not the things that we can do with a list. What do we do with lists? We insert items into them. We remove items from them. We check whether a specific item is in the list. We count how many items are in the list. Sometimes we might want to know what item is in position k in a list (e.g., which horse came in third place in the fourth race today?)

Where can an item be inserted in a list? Only at the end? Only at the beginning or end? Anywhere? Which items can be deleted? Any of them? Only those at the end? Only those at either end? The answers to this question characterize the list's type. We will see soon that stacks and queues are two kinds of lists that constrain where insertions and deletions take place. Can lists be sorted? Is sorting something that a list, as an ADT, should support?

There are two operations that people normally do not perform with lists but computer programs must: creation and destruction. People do not normally say they are creating an empty list, but that is an operation that a list ADT must provide to client software. Similarly, computer programs need to be able to destroy lists because they are no longer needed and they hold resources. People just crumple up the paper with the list on it and dispose of it that way.

## 4 The List ADT

We will define a list ADT. We know that a list contains a sequence of elements as its data, so we will assume that the sequence is $a_1$, $a_2$, $a_3$, ..., $a_N$. Notice that the list index values are 1-based, not 0-based. There is no item at position 0. The first item is at position 1. Based on the above considerations, we will assume that the list operations are:

1. Create an empty list.
2. Destroy a (possibly non-empty) list.
3. Report whether it is empty or not.
4. Report how many elements it contains.
5. Insert an element at a given position in the list.
6. Delete the element at a given position in the list.
7. Retrieve the element at a given position in the list.
8. (Optional) Report whether a particular element is a member of the list. (e.g., is milk in the shopping list?)

The last operation does not have to be a supported operation, as we can repeatedly use the previous operation to determine whether a particular element is in the list, but doing so may not be efficient. We will address this issue later. In fact we do not need an operation to determine whether it is empty if we have one that tells us how many items are in the list, but it is convenient.

Now we make the specified operations more precise by writing them with enough detail so that they are unambiguous and clear. They are written first with a combination of C++ and pseudocode. None of the operations have the list as a parameter because it is assumed that they are called on a list object of some kind. This level of detail will come when they are converted to actual code.

### 4.1 The List Interface (Operation Contract)

This description does not use the UML syntax.

```
create_list();
/* Creates an empty list
   A new empty list is created. This is essentially a C++ constructor.
*/

destroy_list();
/* Destroys the list
   This is a destructor. It deallocates all memory belonging to the list.
*/

bool is_empty() const;
/* Checks if the list is empty
   This returns true if the list is empty and false if it is not.
*/
int  length() const;
/* Determines the length of a list. This returns the length of the list.*/
```

```
void insert( [in] int position_to_insert,
             [in] list_item_type new_item,
             [out] bool& success);
// Inserts new_item into a list
// If  1 <= position_to_insert <= list.length() + 1, then new_item
// is inserted into the list at position position_to_insert and
// success is true afterward. Otherwise success is false.
// The items in positions >= position_to_insert are shifted so that
// their positions in the list are one greater than before the insertion.
// Note: Insertion will not be successful if
// position_to_insert < 1 or > ListLength()+1.

void delete( [in] int position,
             [out] bool& success);
// Deletes an item from a list.
// Precondition: position indicates where the deletion
//               should occur.
// Postcondition: If 1 <= position <= list.length(),
//               the item at position position in the list is
//               deleted, other items are renumbered accordingly,
//               and success is true; otherwise success is false.

void retrieve([in] int position,
              [out] list_item_type& DataItem,
              [out] bool& success) const;
// Retrieves a list item by position number.
// Precondition: position is the number of the item to
//               be retrieved.
// Postcondition: If 1 <= position <= list.length(),
//               DataItem is the value of the desired item and
//               success is true; otherwise success is false.
```

The insert, delete, and retrieve operations require a bitof explanation. Each has a position parameter and a parameter to indicate whether the operation succeeded. The position parameter is 1-based:

```
insert(1,Sam, succeeded)
```

will always succeed because position 1 is at least 1 and never greater than the list length +1. Therefore the effect is to put Sam into the first position, shifting all elements as necessary. In contrast

```
insert(0, Sam, succeeded)
```

fails because $0 < 1$. If the list is empty, then any delete operation will fail because list length $< 1$ and so the condition

```
1 <= position <= listlength()
```

can never be true. If the list has length N, then the operation

```
delete(k, succeeded)
```

will succeed as long as 1 <= k <= N, and the effect will be to remove the item $a_k$, shifting items $a_{k+1}, a_{k+2}, ..., a_N$ to positions $k, k+1, ..., N-1$ respectively. Similarly, the retrieve operation must be given a position parameter whose value lies between 1 and the list length, otherwise it will fail, so if the list is empty, it will fail.

The above is just the specification of the interface. The job of the programmer is to convert this interface to code and to implement it. Those steps come later.

## 4.2 A UML Description of the List Interface (Operation Contract)

```
+create_list(): void
/* Creates an empty list
   A new empty list is created. This is essentially a C++ constructor.
*/
+destroy_list(): void
/* Destroys the list
   This is a destructor. It deallocates all memory belonging to the list.
*/
+is_empty(): boolean {query}
/* Checks if the list is empty
   This returns true if the list is empty and false if it is not.
*/
+length(): integer {query}
/* Determines the length of a list. This returns the length of the list. */

+insert( in position_to_insert: integer,
         in new_item: list_item_type,
         out success: boolean): void
// Inserts new_item into a list
// If  1 <= position_to_insert <= list.length() + 1, then new_item
// is inserted into the list at position position_to_insert and
// success is true afterward. Otherwise success is false.
// The items in positions >= position_to_insert are shifted so that
// their positions in the list are one greater than before the insertion.
// Note: Insertion will not be successful if
// position_to_insert < 1 or > ListLength()+1.

+delete( in position: integer,
        out success: boolean): void
// Deletes an item from a list.
// Precondition: position indicates where the deletion
//               should occur.
// Postcondition: If 1 <= position <= list.length(),
//               the item at position position in the list is
//               deleted, other items are renumbered accordingly,
//               and success is true; otherwise success is false.

+retrieve(in position: integer,
          out DataItem: list_item_type,
          out success: boolean): void {query}
// Retrieves a list item by position number.
// Precondition: position is the number of the item to
//               be retrieved.
// Postcondition: If 1 <= position <= list.length(),
//              DataItem is the value of the desired item and
//              success is true; otherwise success is false.
```

## 5 The Sorted List ADT

Now we turn to a slightly different ADT, also a list, but one that provides a different set of operations. A **sorted list** is a list in which the elements are ordered by their values. It is distinguished from an unsorted list in the following ways:

- Insertions and deletions must preserve the ordering of the elements.
- The insert operation does not insert an item into a specific position; it is inserted without a supplied position; the list decides where it belongs based on its value.
- The delete operation is given the value of an element, called a **key**, and if it finds the element, it deletes it. It does not delete by position.
- It is often endowed with a "find" operation, which returns the position of an element in the list. This operation might be named locate, or search, or even something like get_position, but the traditional (historical) name is find.

Notice that the list does not need a sort operation! As long as the insert and delete operations perform as described above, the list will always be in sorted order whenever an operation is about to be called. (It may not be in sorted order at certain points in the middle of operations.)

The find operation may seem unnecessary at first. Why would you want the position of an item? The answer is that the retrieve operation is still present in a sorted list, and retrieve still expects a position. Therefore, if you want to get the element whose key is, for argument's sake, "samantha", you would first find the position of that key in the list and then retrieve the element atthat position.

The ADT for a sorted list is therefore similar to that of an unsorted list except that insert and delete are replaced by different ones and find is added:

```
bool is_empty() const;
// Same semantics as the unsorted list operation
// Check if list is empty
// This returns true if the list is empty and false if it is not.

int  length() const;
// Same semantics as the unsorted list operation
// Determines the length of a list
// This returns the length of the list.

void insert( [in] list_item_type new_item,
             [out] bool& success);
// Inserts new_item into a list in a position such that
// the items before new_item are not greater than it and the
// items after new_item are not smaller than it.
// success is true if the insertion succeeded, and false otherwise

void delete( [in] list_item_type new_item,
             [out] bool& success);
// If the list contains an element with value new_item,
// then that element is deleted from the list and success is set to
// true.
// If not, no deletion takes place and success is set to false.
// NOTE: if the list contains multiple items with value new_item, this
// operation may either:
// remove the first, or
// remove all of them, or
// allow another parameter to specify which way it should behave

void retrieve([in] int position,
              [out] list_item_type& DataItem,
              [out] bool& success) const;
// Same semantics as the unsorted list operation
// Retrieves a list item by position number.

int find( [in] list_item_type DataItem) const;
// If there is an element with key DataItem in the list,
// this returns the index of the first occurrence of DataItem
// otherwise it returns -1 to indicate it is not in the list
```

## 6 Implementing ADTs

If you are handed a description of an ADT, how do you implement it? The following steps are a good guide.

1. Refine the ADT to the point where the ADT itself is written completely in programming language code. This way you see clearly what the operations must do. The ADT will then be the interface of a C++ class.
2. Make a decision about the internal data structures that will store the data. This decision must be made simultaneously with decisions about major algorithms that manipulate the data. For example, if the data is stored in sorted order, search operations are faster than if it is unsorted. But insertions will take longer in order to maintain the order. Will insertions be very frequent, even more frequent than searches, or will the data be inserted once and never removed, followed by many more searches (like a dictionary and like the symbol table created by a compiler as it compiles a program.) The data structures will be the private data of the C++ class.
3. Write the implementations of all of the functions described in the interface. These implementations should be in a separate implementation file for the class. The section "Separating Class Interface and Implementation" below describes the details of separate interface and implementation files.
4. Make sure that the only operations in the interface are those that clients need. Move all others into the file containing the implementation. In other words, only expose the operations of the ADT, not those used to implement it.

## 7 C++ Review

This is a review of selected topics that are important for the development of classes in C++. This material was supposed to be covered in the prerequisite courses.

### 7.1 Separating Class Interface and Implementation

A class definition should always be placed in a `.h` file (called a header file) and its implementation in a `.cpp` file (called an implementation file.) If you are implementing a class that you would like to distribute to many users, you should distribute the header file and the compiled implementation file, i.e., the object code (either in a `.o` file or compiled into a library file.) The header file should be thoroughly documented. If the implementation needs the interface, which it usually does, put an `#include` directive in the `.cpp` file. The `#ifndef` directive is used to prevent multiple includes of the same file, which would cause compiler errors. `#ifndef X` is evaluated to true by the preprocessor if the symbol `X` is not defined at that point. `X` can be defined by either a `#define` directive, or by a `-DX` in the compiler's command-line options. The convention is to write a header file in the folowing format:

```
#ifndef __HEADERNAME_H #define __HEADERNAME_H

// interface definitions appear here

#endif // __HEADERNAME_H
```

For those wondering why we need this, remember that the `#include` incorporates the named file into the current file at the point of the directive. If we do not enclose the header file in this pair of directives, and two or more included files contain an include directive for the header file, then multiple definitions of the same class (or anything else declared in the header file) will occur and this is a syntax error.

## 7.2 Functions with Default Parameters

Any function may have default arguments. A default argument is the value assigned to a formal parameter in the event that the calling function does not supply a value for that parameter. The syntax is:

`return_type function_name (` $t_1$ $p_1$`,` $t_2$ $p_2$`, ...,` $t_k$ $p_k$ `=` $d_k$`, ...,` $t_n$ $p_n$ `=` $d_n$`);`

If parameter $p_k$ has a default value, then all parameters$p_i$, with i > k must also have default values. (I.e., all parameters to its right in the list of parameters must have default values also.)

If a function is declared prior to its definition, as in a class interface, the defaults should not be repeated again – it is not necessary and will cause an error if they differ in any way.

If default parameters are supplied and the function is called with fewer than n arguments, the arguments will be assigned in left to right order. For example, given the function declaration,

```
void carryOn( int count, string name = "", int max = 100);
```

`carryOn(6)` is a legal call that is equivalent to `carryOn(6, "", 100)`, and `carryOn(6, "flip")` is equivalent to `carryOn(6, "flip", 100)`.

If argument k is not supplied an initial value, then all arguments to the right of k must be omitted as well. Thus,

```
void bad( int x = 0, int y);
```

is a compile-time error because x has a default but the parameter y to its right does not.

This topic comes up here because default arguments are particularly useful in reducing the number of separate constructor declarations within a class.

## 7.3 Member Initializer Lists

Consider the following class definitions.

```
class MySubClass
{
public:
    MySubClass(string s) { m_name = s; }
private:
    string m_name;
};

class X
{
public:
    X(int n, string str ): x(n), quirk(str) { }
private:
    const int x;
    MySubClass quirk; MySubClass& pmc;
};
```

The constructor

```
X(int n, string str ): x(n), quirk(str), pmc(quirk) { }
```

causes the constructors for the `int` class and the `MySubClass` class to be called prior to the body of the constructor, in the order in which the members appear in the definition, i.e., first `int` then `MySubClass`. Member initializer lists are necessary in three circumstances:

- To initialize a constant member, such as `x` above;
- To initialize a member that does not have a default constructor, such as `quirk`, of type `MySubClass`;
- To initialize a reference member, such as `pmc` above. References are explained below.

## 7.4 Declaring and Creating Class Objects

This is a review of the different methods of declaring class objects. The three most common ways to declare an object statically are

```
MyClass obj;                 // invokes default constructor
MyClass obj(params);         // invokes constructor with specified # of parameters
MyClass obj = initial value; /* if constructor is not explicit and
                                initial value can be converted to a MyClass object,
                                this creates a temporary object with the initial
                                value and assigns to obj using the copy constructor.
                             */
```

If `initial_value` is a `MyClass` object already, just the copy constructor is called. A copy constructor is a special constructor whose only argument is a parameter whose type is the same as the class object being constructed. If a program does not supply a user-defined copy constructor, a (shallow) copy constructor is provided by the compiler. The call to a copy constructor can be avoided by using the following declaration format instead:

```
MyClass obj = MyClass(params); /* This is an exception to the above rule.
                                  The ordinary constructor is used to create the object
                                  obj directly. The right-hand side is not a call
                                  to a constructor.
                               */
```

Although the above declaration looks very much like the right-hand side is invoking a constructor, it is not. The C++ standard allows this notation as a form of type conversion. The right-hand side is like a cast of the parameter list into an object that can be given to the ordinary constructor of the object. Thus no copy constructor is invoked.

```
MyClass obj4(); // syntax error
```

If a member function does not modify any of the class's data, it should be qualified with the `const` qualifier.

### 7.5 Function Return Values

A function should generally return by value (as opposed to by-reference), as in

```
double sum(const vector<double> & a);
string reverse(const string & w);
```

`sum` returns a double and `reverse` returns a string. Returning a value requires constructing a temporary object in a part of memory that is not destroyed when the function terminates. If an object is large, like a class type or a large array, it may be better to return a reference or a constant reference to it, as in

```
const vector<string> & findmax(const vector<string> & lines);
```

Suppose that `findmax()` searches through the string vector `lines` for the largest string and returns a reference to it. But this can be error-prone – if the returned reference is to an object whose lifetime ends when the function terminates, the result is a runtime error. In particular, if you write

```
int& foo ( )
{
    int temp = 1;
    return temp;
}
```

then your function is returning a reference to `temp`. Because temp is a local variable (technically an automatic variable) of `foo`(), it is destroyed when the function terminates. When the calling function tries to dereference the returned value, a bad run-time error will occur.

Usually you return a reference when you are implementing a member function of a class, and the reference is to a private data member of the class or to the object itself.

## 8 A C++ Interface for the List ADT

Below is part of the interface for the List ADT described above. It is missing the actual data structures to be used and all private members, which will be supplied only after we learn about lists and linked lists in particular.

```
typedef actual_type_to_use list_item_type;

class List
{
public:
   List();   // default constructor
   ~List();  // destructor

   bool is_empty() const;
   // Determines whether a list is empty.
   // Precondition: None.
   // Postcondition: Returns true if the list is empty,
   //                otherwise returns false.

   int length() const;
   // Determines the length of a list.
   // Precondition: None.
   // Postcondition: Returns the number of items that are currently in the list.

   void insert(int new_position, list_item_type new_item,
               bool& Success);
   // Inserts an item into a list.
   // Precondition: new_position indicates where the
   //               insertion should occur. new_item is the item to be inserted.
   // Postcondition: If If 1 <= position <= this->length()+1, new_item is
   //                at position new_position in the list, other items are
   //                renumbered accordingly, and Success is true;
   //                otherwise Success is false.

   void delete(int position, bool& Success);
   // Deletes an item from a list.
   // Precondition: position indicates where the deletion should occur.
   // Postcondition: If 1 <= position <= this->length(),
   //                the item at position position in the list is
   //                deleted, other items are renumbered accordingly,
   //                and Success is true; otherwise Success is false.

   void retrieve(int position, list_item_type & DataItem,
                 bool& Success) const;
   // Retrieves a list item by position number.
   // Precondition: position is the number of the item to be retrieved.
   // Postcondition: If 1 <= position <= this->length(),
   //                DataItem is the value of the desired item and
   //                Success is true; otherwise Success is false.

private:
   // to be determined later

};  // end class
```

When we cover inheritance, you will see that we can make this interface an abstract base class without any private data, and make all of the different solutions to the List ADT subclasses with actual private data.

# Implementing Lists

## 1 Overview

This chapter explores implementations of the list abstract data type, and introduces the concept of dynamically-allocated data structures and, in particular, linked lists. We begin with an implementation of the list ADT that uses an array to store the data. For convenience, the C++ List class interface from Chapter 3 is displayed below.

Listing 1: List Class Interface

```
typedef actual_type_to_use list_item_type;

class List
{
public:
   List();   // default constructor
   ~List();  // destructor

   bool is_empty() const;
   // Determines whether a list is empty.
   // Precondition: None.
   // Postcondition: Returns true if the list is empty,
   //                otherwise returns false.

   int length() const;
   // Determines the length of a list.
   // Precondition: None.
   // Postcondition: Returns the number of items that are currently in the list.

   void insert(int new_position, list_item_type new_item,
                    bool& success);
   // Inserts an item into a list.
   // Precondition: new_position indicates where the
   //               insertion should occur. new_item is the item to be inserted.
   // Postcondition: If If 1 <= position <= this->length()+1, new_item is
   //                at position new_position in the list, other items are
   //                renumbered accordingly, and success is true;
   //                otherwise success is false.

   void delete(int position, bool& success);
   // Deletes an item from a list.
   // Precondition: position indicates where the deletion should occur.
   // Postcondition: If 1 <= position <= this->length(),
   //                the item at position position in the list is
   //                deleted, other items are renumbered accordingly,
   //                and success is true; otherwise success is false.

   void retrieve(int position, list_item_type & DataItem,
                 bool& success) const;
```

```
    // Retrieves a list item by position number.
    // Precondition: position is the number of the item to be retrieved.
    // Postcondition: If 1 <= position <= this->length(),
    //                DataItem is the value of the desired item and
    //                success is true; otherwise success is false.
private:
    // this is what we are about to supply
};
```

## 2 Array-Based Implementation of a List

One way to implement a list is to store successive elements in an array or, in C++, a vector. Let us work with an array for simplicity and language-independence. To determine whether it is feasible to use an array, or whether it is a good idea, we have to think about how each list operation could be implemented if the data were in an array. An array is declared to be a fixed size, known at the time the list is created. Suppose the size is `N`:

```
list_item_type items[N];
```

To keep track of how many items are in the list, we need a variable, which we call `size`:

```
int size;
```

(With vectors, the vector object can keep track of this for us.)

The `List` class interface would be modified by the specification of the two private data members as follows:

```
const int MAX_LIST_SIZE = N;
class List {
public:
    // the methods from Listing 1
private:
    list_item_type items[MAX_LIST_SIZE];
    int size;
};
```

With this definition, creating a list simply means setting `size` to 0. Destroying a list resets `size` to 0. Testing a list for emptiness checks whether `size` is 0 and getting the length of the list is simply returning the value of `size`. These are easy and straightforward to implement.

```
//Constructor
List::List()
{
    size = 0;
}

// Destructor
List::~List()
{
    size = 0;
}
```

```
// Test for emptiness
bool List::is_empty() const
{
    return (0 == size);
}
```

All other list operations must enforce the assertion that the list items are stored in consecutive array entries at all times. Hence, in a non-empty list, the first item is in `items[0]` and the $k^{th}$ item is in `items[k-1]`. The `retrieve()` member function could be implemented by the following code:

```
void List::retrieve( int position,
                     list_item_type & item,
                     bool & success ) const
{
    success = ( 1 <= position  && position <= size  );
    if ( success  ) {
        item = items[position - 1];
    }
}
```

Obviously the `retrieve()` method takes the same number of instructions regardless of the position of the item, and this number is a fixed constant that is independent of the size of the list. This is good. When an algorithm takes a constant number of steps independent of the size of the data structure or input parameters, we say it ***runs in constant-time***.

Having to keep the items in consecutive entries implies that to perform an insertion, the items in the insertion point and after it have to be shifted towards the end, and to perform a deletion, say at position `k`, all items from `items[k]` to `items[size-1]` have to be shifted down by one cell. *This is not so good.* Remember that list positions start at 1, not 0, so the item at position `k` is in cell `items[k-1]`, not `items[k]`, and deleting the $k^{th}$ item means that the array entry `items[k-1]` must be removed.

The insertion has to be done carefully as well. Suppose `k` is the index of the array at which the new item must be inserted. Since items must be shifted towards the high end, `items[size-1]` has to be moved to `items[size]`, then `items[size-2]` moved to `items[size-1]`, and so on until `items[k]` is moved to `items[k+1]`. (You cannot start by moving `items[k]` to `items[k+1]`, then `items[k+1]` to `items[k+2]` and so on. Why not?) Then the item can be put in `items[k]`. Of course if `size` is `N` before the insertion, we cannot insert anything new, because we would overflow the allocated storage and cause a run-time exception. (If we use a vector, then we could "grow" it when this happens.)

The insertion function is therefore

```
void List::insert( int position_to_insert,
                   list_item_type new_item,
                   bool & success )
{
    if ( N == size )
        success = false;   // array is full -- cannot insert
    else if ( position_to_insert < 1 || position_to_insert > size+1 )
        success = false;   // bad value of position_to_insert
    else {
        // make room for new item by shifting all items at
        // k >= position_to_start toward the end of the list
        for (int k = size-1; k >= position_to_insert-1; k--)
             items[k+1] = Items[k];
```

```
            // insert new item
            items[position_to_insert-1] = new_item;
            size++;
            success = true;
        }  // end if
    }
```

It should be clear that the insertion takes the most time when we insert a new first element, since the shifting must be done roughly N times. It takes the least time when we insert a new last element. But in this worst case, when we insert in the front of the list, the number of instructions executed is proportional to the size of the list. We therefore say it is a ***linear-time operation*** or that its ***running time is linear in the size of the list*** because the running time is a linear function of the size of the data structure. (A linear function $f$ of one variable $x$ is of the form $f(x)=ax+b$.)

The deletion operation also requires shifting items in the array, roughly the reverse of insertion. We start at the point of deletion and move towards the end, i.e., we move `items[position]` to `items[position-1]`, then `items[position+1]` to `items[position]`, and so on until we get to `items[size-1]`, which we move to `items[size-2]`. Since there is no `items[size]`, we cannot move it into `items[size-1]`. Hence, the loop that shifts items must stop when the index of the entry being filled is `size-2`:

```
    void List::delete( int position,
                       bool& success)
    {
        if ( 0 == size )
            success = false;  // array is empty
        else if ( position < 1 || position > size )
            success = false;  // bad position
        else {
            // shift all items at positions position down by 1
            // toward the end of the list. In this loop
            // k is the index of the target of the move, not the
            // source
            for (int k = position-1; k <= size-2; k++)
                 items[k] = items[k+1];
            size--;
            success = true;
        }  // end if
    }
```

In this algorithm the number of loop iterations is the length of the list minus the position at which the deltion occurs. In the worst this is when the first element in the list is deleted, because in this case the number of loop iterations is the length of the list (minus one). So we see that the deletion operation also has a worst-case running time that is linear in the size of the list.

## 2.1 Weaknesses of the Array Implementation of a List

Arrays are fixed size, so if the list needs to grow larger than the size of the array, either the implementation will fail, in the sense that it cannot be used in this case, or it has to provide a means to allocate a new, larger array when that happens. Even if a C++ vector is used, this behavior occurs, because the C++ vector class automatically resizes the vector when it becomes too small. (In fact it doubles the vector capacity.) Alternatively, the array can be made so large that it will always be big enough, but that means it will have many unused entries most of the time and be wasteful of storage. So this is one major weakness.

On average, an insertion requires moving half of the list each time, and a deletion, on average, also requires moving half of the list each time. As noted above, the amount of time to do an insertion or a deletion in the worst case (or in the average case) is proportional to the length of the list. As the size of the data set grows, the time to insert and delete increases. This is an even more serious weakness than the capacity problem. Thus, in summary, the problems with arrays as a data structure for implementing a list are that

- they are fixed size – they cannot grow if size exceeded, or if they are auto-resized, it is very time-consuming;
- if they are sized large enough, there is wasted space, and
- inserting and deleting cause significant movement of data and take too much time.

The alternative is to create an implementation

- in which insertions and deletions in the middle are efficient,
- which can grow as needed, and
- which uses no more space than is actually required.

To do this, we need to be able to allocate memory as needed, and release that memory when no longer needed. For this we need pointers[1].

## 3 Review of Pointers

A pointer variable is a variable that stores the address of a memory cell. The memory cell might be the start of any object or function, whether a simple variable, constant, object of a class, and so on.

### 3.1 Usage

Declaring pointer variables:

```
int * intp;         // declares a pointer to an integer memory cell
char *pc;           // pointer to char
int* a[10];         // a is an array of pointers to int
int (*b)[10];       // b is a pointer to an array of 10 ints
int* f(int x);      // f is a function returning a pointer to an int
int (*pf)(int x);   // pf is a pointer to a function returning an int
int* (*pg)(int x);  // pg is a pointer to a function returning a pointer to an int
```

The `*` is a **pointer declarator** operator. In a declaration, it turns the name to its right into a pointer to the type to the left. It is a unary, **right associative** operator. It does not matter whether you leave space to the right or left of it, as demonstrated above.

```
int * p, q;        // p is a pointer not q
int *p, *q;        // p and q are both int pointers
```

A `typedef` is useful for declaring types that are pointers to things:

[1]At least in languages such as C and C++ we do.

```
typedef    int*  intPtrType;
intPtrType intp, intq;       // p and q are both pointer to int
typedef    bool (*MessageFunc) (char* mssge);
// This defines a MessageFunc to be a pointer to a function with a
// null-terminated string as an argument, returning a bool.
// It can be used as follows:
void handle_error( MessageFunc f, char* error_mssge);
// Here, f is a function with a char* argument returning a bool.
```

The preceding declarations do not give values to the pointers. None of the pointers declared above have any valid addresses stored in their memory cells. To define a pointer, i.e., to give it a value, you give it an address. The & operator is the **address-of** operator. It "returns" the address of its operand.

```
int x = 12;
int* p = &x;    // p is a pointer to the int x.
*p = 6;         // this changes x to store 6 and not 12
(*p)++;         // * has lower precedence than the post-increment ++
// so this increments x
int a[100] = {0};
int *q = &a[0]; // q is the address of a[0]
q++;            // q is the address sizeof(int) bytes after a[0], which is a[1]
*q = 2;         // now a[1] contains a 2!
```

## 3.2 Dynamic Memory Allocation

Pointers are the means to perform dynamic memory allocation. The `new` operator allocates memory:

```
int* p;       // p is a pointer to int
p = new int;  // creates an unnamed int cell in memory; p points to it
```

Arrays can be allocated dynamically:

```
int *q;             // q is a pointer to int
q = new int[100];   // 100 ints are allocated and q points to the first of them
for (int i = 0; i < 100; i++)
    q[i] = i*i;
```

Now `q` can be treated like the name of an array, but it is different, because unlike an array name, `q` is not a `const` pointer. It can be assigned to a different memory location. Note that `*q` is the first element of the array, i.e., `q[0]`, and `*(q+1)` is the same as `q[1]`. More generally,

`*(q+k)` is the same as `q[k]`.

The `delete` operator frees memory allocated with `new`. To delete a single cell, use

```
delete p; // delete the cell pointed to by p
```

If `q` is a dynamically allocated array, you need to use the `delete[]` operator, as in

```
delete[] q;
```

## 3.3 Potential Errors: Insufficient Memory, Leaks and Dangling Pointers

There are several things that you can easily do wrong when using pointers, and they are the most dangerous part of C and C++. (This is why Java does not have pointers.) The first common mistake is failing to check the result of using the `new` operator. Most people do not think about the fact that `new` may fail. It can, if your program has run out of memory, so you must always check that it did not. If `new` fails, the value it returns is `NULL`. Good programmers always check this, as in the following code snippet:

```
p = new int;
if ( NULL == p ) {
    // bail out
    cerr << “Out of memory\n”;
    exit(1);
}
*p = ...;
```

Now consider the following lines of code:

```
p = new int;
if ( NULL == p ) {
    // bail out
    cerr << “Out of memory\n”;
    exit(1);
}
*p = 5;
q = p;
delete q;
cout << *p; // error - the memory is gone - dangling pointer
```

Memory is allocated and a pointer to it is given to `p`. The location is filled with the number 5. Then `q` gets a copy of the address stored in `p` and `delete` is called on `q`. That means that `p` no longer points to an allocated memory cell, so the next line causes an error known as a **dangling pointer** error.

One must use caution when multiple pointers point to same cells.

A different problem is a **memory leak**. Memory leaks occur when allocated memory is not freed by the program. For example, the following function allocates an array:

```
void someTask( int n)
{
    int *array = new int[n];
    // do stuff with the array here
}
```

Notice that the function did not delete the array. If this function is called repeatedly by the program, it will accumulate memory and never release it. The program will start to use more memory than might be available for it, and it will start to get slower and slower as the operating system tries to make room for it in memory. If things get really out of hand, it can slow down the entire machine.

# 4 Linked Lists

**Linked lists** are lists in which each data item is contained in a **node** that also has a **link** to the next node in the list, except for the last, which does not link to anything. A **link** is simply a reference to a node. Links are implemented as pointers in a language such as C++ or C, but they may be represented by other means as well. Conceptually, a link is something that can be dereferenced to get the contents of the node to which it points.

## 4.1 Singly-linked Linked List

In a ***singly-linked*** list, each node has a link to the next node in the list, except the last node, which does not. We will use a pointer-based solution to this problem. A pointer-based linked list is a linked list in which the links are implemented as pointers to nodes. A general description of a node is

```
typedef int list_item_type;  // for simplicity assume node data is type int
struct node {
    list_item_type data;  // the node's data element
    node*          next;  // the pointer to the next node
};

typedef node* link;       // link is short for "pointer to node"
```

This can be written more succinctly as follows:

```
typedef struct node * link;  // in C++ it can be written typedef node * link;
struct node {
    list_item_type data;
    link           next;
};
```

Notice in the above that a `link` is defined as another name for a pointer to a `node` structure even though a `node` structure has not yet been defined at that point in the code. This is not incorrect – it will compile, because the only thing the compiler needs to "know" to compile it is how much storage the `link` needs and what type it is. A pointer is always a fixed number of bytes, and the compiler sees that a `link` is only allowed to point to things of type `node` structure, so these requirements are met.

The actual data in a node will vary from one list to another, depending on how the `itemType` is defined. The key point is how the pointers are used. Each pointer points to the next item in the list, unless it is in the last node, in which case it is a null pointer. A linked list is a sequence of these nodes. The list itself is accessed via a special link called the **head**. The head of a list is a pointer to the first node in the list. The **tail** of a list is the last node. Figure 1 depicts this.

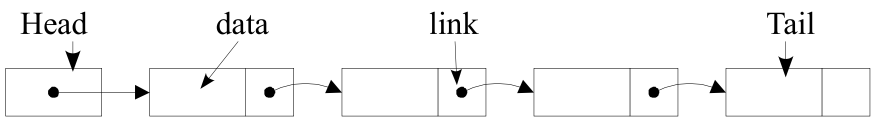


Figure 1: Linked list.

It is convenient to create a special variable to store the head of the list, with the declaration

```
link head;
```

which creates a pointer named `head` that can be used to point to a list. Of course at this point `head` has only garbage in it; it has no address and there is no list. By itself this declaration does not create a list!

It is also convenient to keep track of the length of the list with an integer variable that stores its size. Therefore, a linked list's private data must include a `head` pointer and *should* include a `size` variable:

```
class List
{
public:
    // the member functions defined in Listing 1
private:
    link  head;         // pointer to the first node in the list
    unsigned int size; // number of nodes of the list
};
```

We can now run through how to implement the various operations (methods) defined in a `List` ADT using this pointer-based representation, assuming that `itemType` is a `typedef` for `int` in the above definitions.

### 4.1.1 Creating an Empty List

To create an empty list is trivial:

```
head = NULL; // head points to nothing
size = 0;    // list has zero items
```

An empty list is a list with no nodes, so setting `head` to `NULL` makes `head` a pointer to an empty list, and setting size to 0 says it has no nodes. This implies that the default constructor is simply

```
List::List()
{
    head = NULL;
    size = 0;
}
```

### 4.1.2 Creating a Single List Node

To create a node that is not yet part of a list and put data into it, we need the `new` operator to create the node and then we assign the data to the `data` member and we set the `next` member to `NULL`:

```
link p;
p = new node;              // p points to a new node without data.
if ( NULL == p ) exit(1); // ALWAYS handle this possible error
p->data = 6;               // p->data contains 6
p->next = NULL;            // p->next points nowhere
```

This arbitrarily puts the value 6 into the `data` member of the node. Right now this node is not linked to anything.

### 4.1.3 Writing the Contents of a List to a Stream

Most container classes, such as lists, should have a method that can write their contents onto an output stream, such as a display device or a file. This method was not mentioned when we introduced lists in Chapter 3, but now that we are implementing them, it is worthwhile to develop such a method because it is certainly useful, and it is instructive to implement it. Because it needs access to the list private data, it should be either a friend function or a member of the list class. The approach we use here is to create a private method that can be called by a public method in the class interface. The private function prototype is

```
// precondition:  ostream out is open and list is initialized
// postcondition: ostream out has the contents of list appended to it,
//                one data item per line
void write( ostream & out, link list);
```

This private method can print the list contents starting at any node to which it is given a pointer. The public method would be implemented as

```
void List::write( ostream & out )
{
    write( out, head);
}
```

where `write()` would be a private method that actually writes the list data onto the stream named `out`.

The private `write` function logic is the following:

1. Set a new pointer named `current` to point to whatever `list` points to.
2. Repeatedly do the following:
   (a) if `current` is not a `NULL` pointer, write the data in the node that `current` points to and then advance `current` to point to the next node;
   (b) if `current` is a `NULL` pointer, then exit the loop.

Writing the data means writing it to the output stream followed by a newline. This can be written as follows:

```
void write( ostream & out, link list)
{
    link current = list;

    while ( NULL != current ) {
        out << current->data << "\n";
        current = current->next;
    }
}
```

The instruction

```
current = current->next;
```

is guaranteed to succeed because `current` is not `NULL`, so it points to a node, and the `next` pointer in that node exists. This function is general enough that it can write to the terminal or to a file, so we have a way to save a list to a file.

### 4.1.4 Getting the Size of a Linked List

It is trivial to implement the `length()` function:

```
int List::length() const
{
    return size;
}
```

### 4.1.5 Testing for Emptiness

This is also trivial, and there are two ways to do it. One way is simply

```
bool List::is_empty() const
{
    return size > 0 ? false : true;
}
```

### 4.1.6 Inserting a Node at a Specified Position in a Linked List

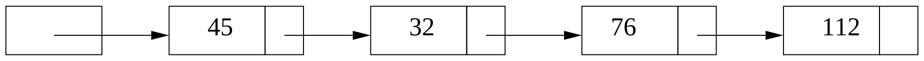
The insert method is a bit more challenging than the preceding ones. Recall that its prototype is:

```
void insert( int position_to_insert,
             list_item_type new_item,
             bool & success );
```

in which 1 <= `position_to_insert` <= `size`+1.

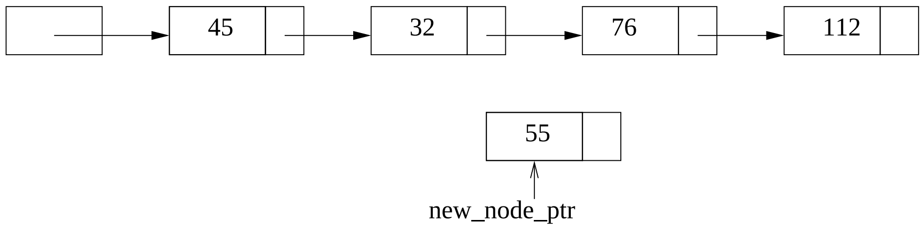
How do we insert into a linked list? What steps must we follow? First consider the case in which the new data is inserted between two existing list nodes. Inserting in the very front of the list or after the last node will be handled afterward.

Assume the list looks like this to start:

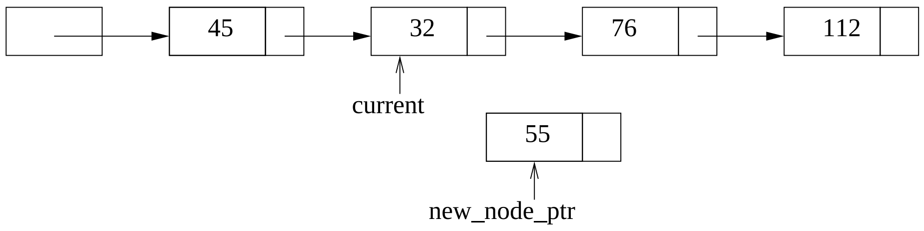


and that we want to insert a node containing the value 55 at position 3. This means that the node must be inserted after 32 and before 76. The steps we must follow are

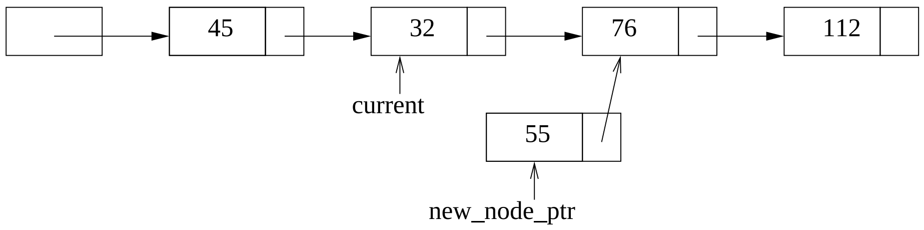
1. Create a new node and fill it with the value to be inserted. Suppose `new_node_ptr` is a pointer to this new node.



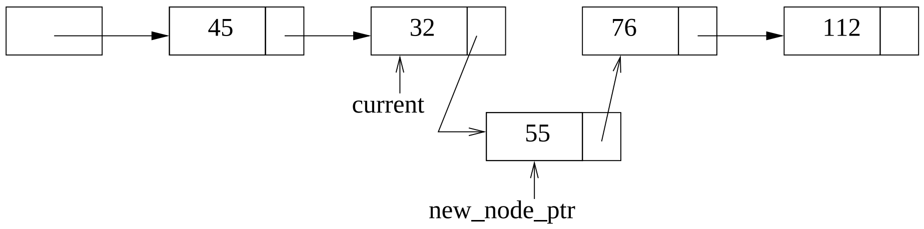
2. Position a pointer variable, which we will name `current`, so that it points to the node containing 32. We will figure out how to do this afterward.



3. Make `new_node_ptr`'s `link` member point to what `current->link` points to. In other words, copy `current->link` into `new_node_ptr->link`.



4. Make `current->link` point to what `new_node_ptr` points to. In other words, copy `new_node_ptr` into `current->link`.



You can see that the new node is now in position 3 in the list, and that the nodes after it were shifted by one position towards the end of the list.

The question that remains is how to position a pointer at a specific position in the list. Because this is an operation that will need to be done for insertion, deletion, and retrieval, we make it a private helper function in our implementation. It is just a matter of advancing a pointer along the list and counting while we do it, like stepping along stones in a pond until we get to the $k^{th}$stone. The following function does this:

```
// precondition: 1 <= pos <= list length
// postcondition: none
// returns a pointer to the node at position pos
link List::get_ptr_to(int pos) const
{
    // sanity check to prevent a crash:
    if ( ( pos < 1 ) || ( pos > size ) )
        return NULL;
    else  { // pos is within the required range
        // count from the beginning of the list
        link current = head;

        for (int i = 1; i < pos; i++ )
             // loop invariant: current points to ith node
          current = current->next;
        // i == pos and therefore current points to pos'th node
        return current;
   }
}
```

Notice that this function will position a pointer at any node in the list, including the last. Also notice that it starts not by pointing to the head node, but by pointing to what `head` points to, namely the first node.

Now what happens if the list is empty? Then `size == 0` and `head == NULL`, so this will return `NULL` because it fails the sanity check.

So tentatively, we can put the preceding steps into a first attempt at an implementation of the insert method:

```
 1  void List::insert(int new_position, list_item_type new_item,
 2                    bool& success)
 3  {
 4      link prev_ptr, new_node_ptr;
 5
 6      int new_size = size + 1;
 7      success = (new_position >= 1) && (new_position <= new_size) ;
 8      if (success) {
 9          // create a new node and place new_item into it
10          new_node_ptr = new node;
11          success = ( NULL != new_node_ptr );
12          if ( success ) {
13              size = new_size;
14              new_node_ptr->data = new_item;
15              prev_ptr = get_ptr_to ( new_position-1 );
16
17              // insert new node after node to which Prev points
18              new_node_ptr->next = prev_ptr->next;
19              prev_ptr->next = new_node_ptr;
20          }
21      }
22  } // end insert
```

This will work, except in one situation: what if we need to insert a node at position 1? The problem is in line 15. When `new_position` == 1, we call `get_ptr_to(new_position-1)` but `get_ptr_to()` does not accept an argument of 0; it makes no sense to have a pointer to the 0th node. Unfortunately, we have to "break out" a special case to handle insertion in the first position, given how we have defined a linked list[2].

In case `new_position` is 1, instead of lines 15, 18, and 19, we need

```
new_node_ptr->next = head;
head = new_node_ptr;
```

Putting this together, we get the correct version of the insertion method:

Listing 2: Linked list insert() method

```
void List::insert(int new_position, list_item_type new_item,
                  bool& success)
{
    link prev_ptr, new_node_ptr;

    int new_size = size + 1;
    success = (new_position >= 1) && (new_position <= new_size) ;
    if (success) {
        // create a new node and place new_item into it
        new_node_ptr = new node;
        success = ( NULL != new_node_ptr );
        if ( success ) {
            size = new_size;
            new_node_ptr->data = new_item;
            if ( 1 == new_position ) {
```

[2]There are alternative implementations that use a "dummy" first node. A dummy node is never used to hold data. It exists just to remove this special case from the code, but it also makes the other operations different. This is a design trade-off that you can explore on your own.

```
                new_node_ptr->next = head;
                head = new_node_ptr;
            }
            else {
                prev_ptr = get_ptr_to ( new_position-1 );
                new_node_ptr->next = prev_ptr->next;
                prev_ptr->next = new_node_ptr;
            }
        }
    }
} // end insert
```

### 4.1.7 Retrieving the Item in a Given Position

We handle this method next because it is much easier than the `delete()` function and the destructor. To retrieve the item at a given position we check that the position is valid, and if so, we use the `get_ptr_to()` function to position a pointer at the node. Then we just extract the value from that node.

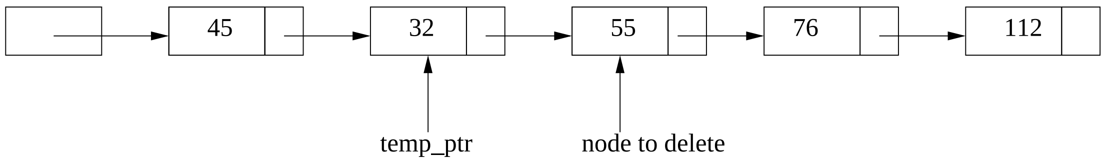
```
void retrieve(int position, list_item_type & DataItem,
              bool& success) const;
{
    link current;
    success = (position >= 1) &&  (position <= size);
    if (success)  {
        // get pointer to node, then data in node
        current = get_ptr_to(position);
        DataItem = current->data;
    }
}
```

### 4.1.8 Deleting an Item from a Linked List

The function that we now need to implement has the prototype

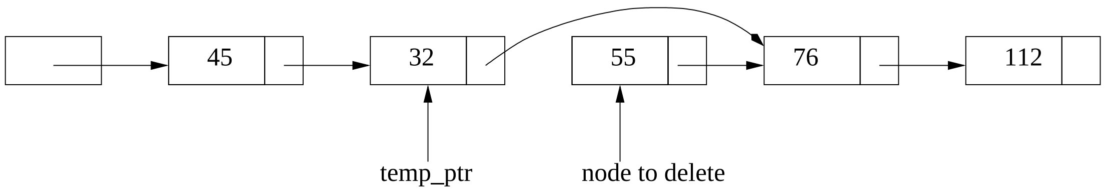
```
void delete(int position, bool& success);
```

Thus, we are given a position and we need to delete the node at that position, assuming it is a valid position. Unlike the insertion method, this requires that we get a pointer to the node ***before*** the node to be deleted. To understand this, consider the following figure.

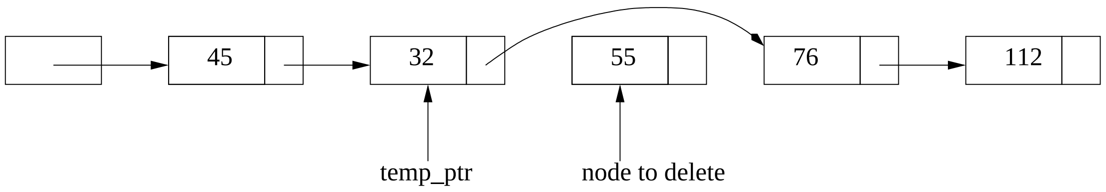


To delete the node containing the item 55, we need to make the `next` link in the previous node, pointed to in the diagram by `temp_ptr`, point to the node ***after*** the node to delete! This means that we need to get a pointer to the node ***before*** the node to be deleted. This is not a problem because we already have the function `get_ptr_to()`, but again we have a special case at the front of the list, which we discuss after the general case. The steps from this point forward are:

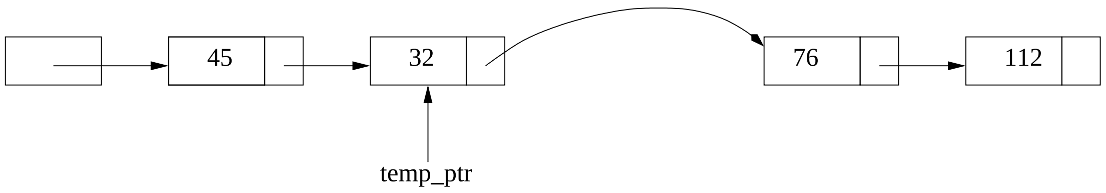
1. Unlink `temp_ptr->next` and relink it so that it points to the node after the node to delete. This is a single assignment statement: `temp_ptr->next = node_to_delete->next`.



2. Unlink the `next` link of the node being deleted, using `node_to_delete->next = NULL`.



3. Release the storage allocated to the node, using the `delete` operator: `delete node_to_delete`.



The preceding steps will not work if the node to delete is the first node. In this case we need to execute the following instructions instead of the ones just described:

```
temp_ptr = head;
head     = head->next;
delete temp_ptr;
```

This is assuming that the list is not empty of course. If the list is not empty, then the head pointer does point to a node, so there will not be an error trying to dereference that pointer to get to the node's `next` member. We put all of this together in the following function.

```
void List::delete(int position,  bool& success)
{
    link node_to_delete, temp_ptr;

    // the following will set success to false if the list is empty and position > 0
    success = (position >= 1) &&  (position <= size) ;

    if (success)  {
        // if we make it here, position >= 1 and we are definitely going to delete
        // something, so we decrement list size.
        --size;
        if (1 == position)  {  // delete the first node from the list
            node_to_delete = head;   // save pointer to node
            head           = head->next;
            }
            else  {
                // Because position must be greater than 1 here, we can safely call
                // get_ptr_to() now.
                temp_ptr = get_ptr_to(position-1);

                // delete the node after the node to which temp_ptr points
                node_to_delete = temp_ptr->next;  // save pointer to node
                temp_ptr->next = node_to_delete->next;
            }
            // node_to_delete points to node to be deleted
            node_to_delete->next = NULL;  // always safe to do this
            delete node_to_delete;
            node_to_delete  = NULL;       // and this
        }
    }
```

Notice in the above function that we purposely set the `next` link in the node we are about to delete to `NULL`. Even though we are returning it to the system to be reused, we break the link from this node to our list, just in case somehow this storage gets reused without changing its bits. This is just safe programming. Similarly, we set `node_to_delete` to `NULL`, even though it is just on the program's run-time stack. Perhaps it is overly cautious.

### 4.1.9 Implementing a Destructor

For the first time we really do need a non-default destructor in a class. If the list is not empty, then it contains one or more nodes that have been allocated from the heap using the `new` operator, and these must be returned to the operating system so that the memory can be re-used if needed. A default constructor will free up the `head` pointer and the `size` member, but it cannot travel down a list deleting the nodes in it! It is our job to do this.

One approach is to use some kind of loop that traverses the list, deleting one node after the other, but if we do it the wrong way, we will detach the list before we delete its nodes. Another way to do this that uses a bit more overhead and is therefore a bit slower is the following:

```
    List::~List()
    {
        bool success;
        while ( size > 0 )
            delete(1, success);
    }
```

This destructor uses the public `delete()` member function to keep deleting the first node in the list as long as size is greater than zero. Obviously it will delete all of the nodes. It is a bit slower because it repeatedly makes a function call for each node in the list. Function calls have setup times. We can use a faster implementation by replacing the function call by inline code that does the same thing.

```
    List::~List()
    {
        link node_to_delete;
        while ( size > 0 ) {
            node_to_delete       = head;       // save pointer to first node
            head                 = head->next; // detach first node from list
            node_to_delete->next = NULL;         // clear its next link
            delete node_to_delete;               // release its storage
            size--;                              // decrement size
    }
```

The loop body basically does what the `delete()` function does.

#### 4.1.10 Anything Else?

We implemented all of the public methods of the `List` ADT, plus a `write()` function as well. We should add a copy constructor, if we want to be able to create one list from an existing list. We might want a `read()` function, which would read from an input stream and create a list from it. These are good programming exercises.

## 4.2 Variations on the Linked List

There are several variations on linked lists. We already mentioned using a dummy head, or first, node. A dummy node has no data in it. It exists just to eliminate the need for special cases in operations such as insertions and deletions. The test for emptiness changes when a dummy head node is used. Later we shall explore doubly-linked lists, in which each node has a pointer to its predecessor and successor. One can also keep a pointer to the tail of a list, to make appending a new item at the end more efficient, otherwise one has to travel the entire list each time to do this, which is inefficient.

## 4.3 Sorted Linked List

A sorted list is really a different type of object, as we discussed in the third chapter. If the linked list is kept in sorted order, then inserting, deleting, and retrieving operations must be different than if it is unsorted. The other operations are the same and they are not discussed here.

### 4.3.1 Inserting into a Sorted List

The insertion algorithm must advance through the list, comparing each node's item with the item to insert. While the new item is greater than the list item, the pointer is advanced. If the new item is equal to the current list node item, searching stops and the algorithm reports that nothing was inserted. If the new item is less than the list node's item, it must inserted before that list node, because it was greater than all nodes before it in the list and smaller than the current one.

The problem with this idea is that, in the course of discovering where the node must be inserted, the pointer is advanced too far. We only know where to insert the node after we passed the spot. The pointer points to the first node whose data is larger than the new item, but in order to insert the new item before that node we need a pointer to the node preceding it. One solution is to advance two pointers through the loop, one trailing the other. This complicates the algorithm because we have to be careful how to handle the empty list and the case when the insertion must happen at the beginning. The following code solves these problems.

We use a private function, `insert_at()`, to simplify the solution.

```
    bool insert_at( link & prev, list_item_type new_item )
    /**
     * @pre prev != NULL
     * @post prev points to a new node containing new_item and the
     *       next link of the new node points to what prev previously
     *       pointed to.
     */
    {
        link temp_ptr = new node;
        if ( NULL == temp_ptr )
            return false;
        temp_ptr->data = new_item;
        temp_ptr->next = prev;
        prev           = temp_ptr;
        return true;
    }
```

Listing 3: Sorted list insertion method

```
bool SortedList::insert( list_item_type new_item)
{
    link current, previous;
    bool success;

    // If list is empty, or if it is not and the new_item is smaller
    // than the first item in the list, insert it at the front
    if ( NULL == head || head->data > new_item ) {
        success = insert_at( head, new_item);
        if (success)
            size++;
        return success;
    }

    // list is not empty and first item is not larger than new_item
    if ( head->data == new_item )
        // quit since it is in the list already
        return false;
    }

    // head->data < new_item, so we start at 2nd node if it exists
    previous = head;
    current  = head->next;
    while ( current != NULL  ) {
        if ( current->data < new_item ) {
            previous = current;
            current = current->next;
        }
        else if ( current->data > new_item ) {
            // previous points to an item that was smaller and current*
            // is bigger so new_item should be inserted after previous
            if ( success = insert_at(prev->next, new_item) )
                size++;
            return success;
        }
        else  // found a duplicate, so do not insert
            return false;
    } //end while

    // if we reach here, the item is bigger than everything in list
    // and previous points to last node
    if ( success = insert_at(previous->next, new_item) )
```

```
        size++;
    return success;
}
```

### 4.3.2 Searching a Sorted List

The problem of finding an item in a list is not called retrieving; it is called **searching**, or **finding**, because the algorithm is searching for a particular value, not a particular position. Usually we search for keys in order to return the data associated with them. For example, a node might contain records that represent driving records, and we search for a driver's license number in order to retrieve the entire record. If a node contains nothing but a key, then there is nothing to return other than true or false – it was found or it was not found. To generalize the algorithm, we will assume that the item is a complex structure.

The following search procedure is designed to find all occurrences of the item within the list, as opposed to just the first. Because it might find multiple items, it uses an output stream as the means for returning them to the calling function. If it were to return a single item, it could just make the item its return value. Another way to return multiple items is to construct a second list whose members are all matching nodes. That is a bit more complex, so we handle it this way here.

```
void  SortedList::find (ostream & output, list_item_type  item) const
{
    NodePtr          current;
    list_item_type   temp;

    // If list is empty, return
    if ( NULL == head )
        return;

    // list is not empty
    current  = head;
    while ( current != NULL ) {
        temp = current->item;
        if ( temp  < item ) {
            current = current->next;
        }
        else if ( temp > item ) {
            // item is not in remainder of list
            return;
        }
        else  { // found the item
            output << current->item;
            current = current->next;
        }
    }
}
```

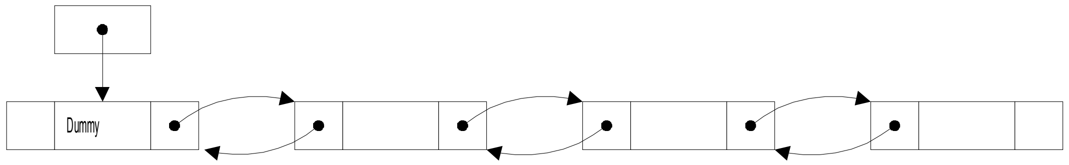
### 4.3.3 Deleting from a Sorted List

Deleting from a sorted list is similar to finding, except that items are removed as they are found. The following procedure deletes all occurrences of matching items, and returns the number of items deleted. As with deleting from an unsorted list, handling the first node is a special case that must be checked within the code. The deletion algorithm uses two pointers, one "chasing" the other. The `prev` pointer is always one node behind the `current` pointer, except when it starts, when `prev` is `NULL`. This is how the code can detect if the node to be deleted is the first node, because `prev` is still `NULL`.

```
int SortedList::delete( list_item_type item )
{
    link current, prev;
    int  count = 0;
    // If the list is empty, there is nothing to do
    if ( NULL != head ) {
        current = head;
        prev = NULL;
        while ( current != NULL ) {
            if ( current->item < item ) {
                // item is bigger than current item so advance pointers
                prev = current;
                current = current->next;
            }
            else if ( ( current->item > item ) {
                // item is smaller than current item, so item cannot be in
                // remainder of the list and we can stop searching
                break;
            }
            else { // current->item == item
                if ( NULL == prev ) {
                    // first node: handle separately
                    head          = current->next; // detach from list
                    current->next = NULL;          // clear next link
                    delete current;                // free node
                    current = head;
                    count++;
                }
                else { // not first node, so prev can be dereferenced
                    current = current->next;    // advance current
                    prev->next->next = NULL;    // clear next link in node
                    delete prev->next;          // free node
                    prev->next = current;       // detach node from list
                    count++;
                }
            }
        }  // end while
    } // end if head != NULL
    size -= count;
    return count;
}
```

## 4.4 Doubly-Linked Lists

In a **doubly-linked list**, each node contains a pointer to the node preceding it in the list as well a a pointer to the node following it. Because of this, there is no need to have a pointer to the previous node in the list to do insertions and deletions. On the other hand, it also makes insertions and deletions a bit more work to do. Doubly-linked lists make it possible to traverse a list in reverse order. Many lists are implemented as doubly-linked lists with dummy nodes. The use of dummy nodes eliminates special cases for operations at the front of the list. The following diagram shows what a doubly-linked list with a dummy node looks like.



Rather than going through all of the operations and how they are implemented, we can just look at a few of the things that make them different. For example, to delete the node to which a link named `cur` points, it is sufficient to do the following:

```
(cur->prev)->next = cur->next; // make the previous node point to the one after cur
(cur->next)->prev = cur->prev; // make the one after cur point to the one before it
cur->next = NULL;              // clear its links and delete it
cur->prev = NULL;
delete cur;
```

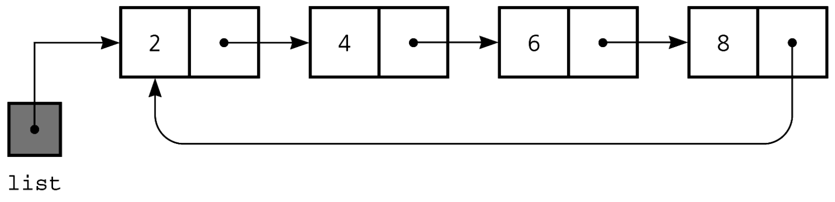
To insert a new node pointed to by `newPtr` before the node pointed to by `cur`

```
newPtr->next = cur;
newPtr->prev = cur->prev;
cur->prev    = newPtr;
(newPtr->prev)->next = newPtr;
```

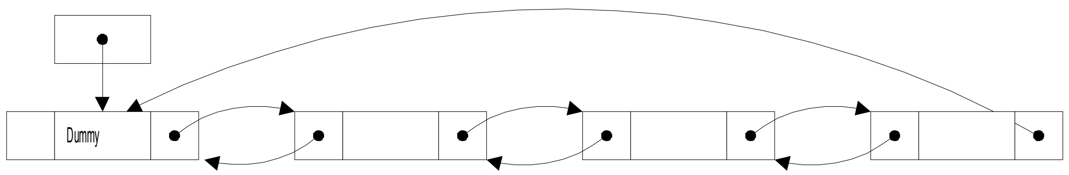
The operations that do not modify a list do not need to be changed. The constructor needs to set up the dummy node.

## 4.5 Circular Linked Lists and Circular Doubly Linked Lists

Ordinary singly-linked lists have the disadvantage that one must always start from the beginning of the list to do any type of traversal. In a **circular list**, the tail node points back to the first node. This eliminates need for special cases for first and last node insertions and deletions, and it also makes it possible to start searches where one left off. The diagram below illustrates a circular, singly-linked list.



In a **circular doubly-linked list**, the tail points to the dummy node, not the first node. Circular doubly-linked lists are also very useful for many applications. A circular doubly-linked list is illustrated below.



## A C++ Exceptions

This section is an excerpt of my exception handling notes in C++ in the Resources page of the website, which has a more thorough explanation of it. An ***exception*** is an event that is invoked when something "goes wrong" during the execution of a program. Exceptions can be ***raised*** by hardware or by software. Dividing by zero, running out of memory, and attempting to access protected parts of memory, are examples of events typically caught by the hardware and then ***handled*** by the operating system. Usually the operating system will terminate the program that caused it. Software can also raise exceptions if the language run-time environment supports it, which C++ does.

In C++, instead of saying that the software *raised* an exception, we say that software ***throws*** an exception. The three operators that make exception handling work are: `throw`, `try`, and `catch`, which are keywords in C++. The idea is that you create a *try-block*, which is the `try` keyword followed by a statement block, which is then followed by *exception handlers*. If within that block, an exception is thrown, then control can pass to one of the exception handlers that follow the block, if they have been set up properly. An exception handler is a function that is executed when an exception is thrown, and is introduced by the keyword `catch`. The basic form is

```
try
{
     a statement sequence containing function calls to functions that
     might throw exceptions, or to functions that might  call functions
     that might call functions that might, and so on ... throw exceptions
}
catch ( exception_declaration_1 )
{
     statements to handle exceptions of type exception_declaration_1
}
catch (exception_declaration_2 )
{
     statements to handle exceptions of type exception_declaration_2
}
catch (exception_declaration_3 )
{
     statements to handle exceptions of type exception_declaration_3
}
...
```

```
catch (exception_declaration_N )
{
     statements to handle exceptions of type exception_declaration_N
}
```

If an exception is thrown by some function that is called directly or indirectly from within the try-block, execution immediately jumps to the first exception handler whose exception declaration matches the raised exception. If there are no handlers that can catch the exception, the try-block behaves like an ordinary block, which means it will probably terminate.

An example demonstrates the basic idea.

```
struct Ball
{
    int b;
    Ball(int x = 0) : b(x) { } // initializes member variable b
};

void test ( )
{
    /*
        ...
    */
    throw Ball(6);
    // This is a throw-statement. It is throwing an anonymous object
    // of Ball structure, initialized by a call to the constructor
}

int main ( )
{
    /* ... stuff here */
    try
    {
        test();
    }
    catch (the_Ball c)
    {
        cerr << " test() threw ball " << c.b << "\n";
    }
...
}
```

A `throw` statement is, as shown above, of the form

```
throw expression;
```

The thrown expression is any expression that could appear on the right hand side of an assignment statement. The most likely candidate is a call to a constructor, such as I showed above. The safest way to handle exceptions is to define unique types for the distinct exceptions. One way to guarantee that the types are unique is to enclose them in a namespace. This is also a good place for an error count variable.

```
// The Error namespace defined below declares three types: Towel, Game, and Ball,
// an int variable that counts errors, and a function named update() .
namespace Error {
    struct Towel {  };  // empty struct, default generated constructor
    struct Game { };    // empty struct
    struct Ball {
        int b;
    Ball(int x = 0) : b(x) { }
    };

    int count = 0;      // counts errors thrown so fa
    void update()  { count++; }  // increments error count
}
```

A program can use these error types as follows. There are three functions, `Pitcher()`, `Boxer()`, and `PoorSport()`. Each might throw an exception of some type if their input meets some specific set of conditions. For simplicity, the exception is thrown if the single integer parameter is a multiple of a fixed number. Notice that to throw an exception in the Error namespace, the function must use the syntax `Error::` followed by the type being thrown.

```
void Pitcher ( int n )
{
    /* ... */
    if ( n % 2 == 0 )
    throw Error::Ball(6);
}
void Boxer ( int n )
{
/* ... */
    if ( n % 3 == 0 )
    throw Error::the_Towel();
}
void PoorSport ( int n )
{
/* ... */
    if ( n % 11 == 0 )
    throw Error::the_Game();
}

// The main program calls Pitcher(), Boxer(), and PoorSport(). Because these might throw
// exceptions and main () does not want to be terminated because of an unhandled
// exception, if places the calls in a try-block that is followed by exception handlers.
int  main ( )
{
    /* ... stuff here */
    int n;
    cin >> n;
    try {
        Pitcher( n );
        Boxer( n );
        PoorSport( n );
    }
    catch (Error::Ball c)
    {
        Error::update();
```

```
            cerr <<Error::count << ": threw ball " << c.b << "\n";
        }
        catch (Error::Towel )
        {
            Error::update();
            cerr <<Error::count << ": threw in the towel\n";
        }
        catch (Error::Game )
        {
            Error::update();
            cerr << Error::count << ": threw the game\n";
        }
        /* ... */
        return 0;
    }
```

There is a special notation that you can use for the parameter of an exception handler when you want it to catch all exceptions.

```
catch (...)
```

means catch every exception. It acts like the default case of a switch statement when placed at the end of the list of handlers.

This is just a superficial overview of the exception handling mechanism. To give you a feeling for why there is much more to learn, consider some of the following complexities.

- Since any type is a valid exception type, suppose it is publicly derived class of a base type. Do the handlers catch it if they catch the base type? What if it is the other way around?
- Can a handler throw an exception? If so, can it throw an exception of the same type as it handles? If so, what happens? Does it call itself? (Fortunately, although the answer is "yes" to the first question, it is "no" to the third.)
- What if the exception is thrown as a reference, but the handler declares it as a `const` parameter?
- Are there standard exceptions, and if so, what are they? Which library functions can raise them? The answer to this is that there is a class hierarchy of exceptions, with, not surprisingly, a root named `exception`. All exceptions derive from `exception`. There are `logic_error` exceptions, `bad_cast`, `bad_alloc` (thrown by `new()` ), `underflow_error`, and `out_of_range` (thrown by `at( )`) exceptions, to name a few.

Lastly, you may see function declarations that look like

```
void f ( int a) throw ( err1, err2);
```

which specifies that `f()` is only allowed to throw exceptions of types `err1` and `err2`, and any exceptions that may be derived from these types. Why would you want to do this? Because it is a way of telling the reader of your interface that the function promises to throw only those exceptions, and so the caller does not have to handle any exceptions except those. So

```
void f( int a) throw ( );
```

means that `f()` throws no exceptions. If `f()` does throw a different exception, the system will turn it into an `std::unexpected()` call, which results in a terminate call.

## B The Standard Exception Hierarchy

The set of standard exceptions forms a hierarchy in C++. The language defines a base class named `exception` from which other exceptions are derived. Some of these are defined in the `<stdexcept>` header file; others are defined in `<typeinfo>`, `<new>`, `<ios>`, and `<exception>`. The standard exceptions are listed below. The indentation indicates sub-classing. To the right of the exception name is the header file that contains its definition.

| **Exception Type** | | | **Header File** |
|---|---|---|---|
| exception | | | <exception> |
| | bad_alloc | | <new> |
| | bad_cast | | <typeinfo> |
| | bad_exception | | <exception> |
| | bad_typeid | | <typeinfo> |
| | ios_base::failure | | <ios> |
| | logic_error | | <stdexcept> |
| | | length_error | <stdexcept> |
| | | domain_error | <stdexcept> |
| | | out_of_range | <stdexcept> |
| | | invalid_argument | <stdexcept> |
| | runtime_error | | <stdexcept> |
| | | overflow_error | <stdexcept> |
| | | range_error | <stdexcept> |
| | | underflow_error | <stdexcept> |

# Recursion and Problem Solving

## 1 Problem Solving and Backtracking

***Backtracking*** is an algorithm paradigm that can be used for finding one or more solutions to certain types of computational problems. It works by incrementally building partial solutions, checking if a partial solution can be extended to a complete solution, and discarding a partial solution if it determines that it cannot be so extended. It is best understood through examples. We begin with a practical, non-computational, example.

Imagine that you are in a forest without a map. The forest consists of thick brush and trees with many paths through them. You are at a point that we will call point **A**. You are trying to get to point **B**. There are many forks along the path you are on, sometimes it forks into over a dozen paths. Without a map that lets you know which paths to take, you need a methodical way to get to point **B**. It may even be possible that there is no path from **A** to **B**. The only thing you are guaranteed is that none of these paths are cyclical, so you will never return to a place you already visited by going straight on any path from that point.

*Backtracking* is a method that can organize your search for **B**. Suppose you use the following rules:

1. Walk straight until you come to a fork in the road (the end of the current partial solution) or you reach **B**. If you reach **B**, you are done (a complete solution.)
2. At every fork in the road, always choose the rightmost path that you have not yet tried (a systematic way to extend a partial solution.) If, when you come to a fork, you have tried all of the paths possible at that fork already, then turn around and walk back to the closest fork you just left (this is the backtrack part), mark the branch on which you traveled back as "tried already" (a discarded partial solution), and then repeat this step.
3. If walking back to the nearest fork (backtracking) brings you back to point **A** because all branches of the first fork failed to reach **B**, and there are no other branches to try at **A**, then there is no path to **B**.

This algorithm will allow you to reach **B** if there is a path to it. The aspects of it that make it an example of backtracking are that it methodically tries all solutions by tracing backwards when a search leads to failure and choosing the next possible search. This is an example of a problem that is considered to be solved if you find a single solution. Usually if you are lost in a forest you just want to find the way out, rather than finding all possible ways out. If we wanted to find all possible paths from **A** to **B**, then in step 1, we would not terminate if we reached **B**. In addition, in step 2, we would not consider paths as being discarded unless they failed to reach **B**.

For backtracking to be a useful technique, the problem to be solved must admit to two conditions:

1. There has to be a concept of a partial candidate solution.
2. There must be an efficient (fast) test to determine whether a partial solution can be completed to a valid solution.

### 1.1 Backtracking and Recursion

Recursion is a useful tool for writing backtracking algorithms because it implicitly traces back to the last guess for you. When a recursive function returns to the point immediately after it was called, it has traced backwards. By putting the recursive calls into either a loop or a set of cascading if-statements, you can organize the function to systematically check all possible guesses.

**Example 1: The Sum of Four Squares**

LaGrange's 4-squares theorem states that any natural number can be written as the sum of four squares (including 0 if need be.) Suppose we wanted an algorithm that, given any natural number, could find four squares whose sum equals the number. Backtracking can do this. The idea is to pick a square less than the number, subtract it, and then find three numbers that add up to the difference. If we succeed, then we found the four numbers. If we do not, then we back up to the first number we picked and try a different first number. We can proceed by picking the smallest first number first, or by picking the largest first number first. Either way we can proceed methodically. The algorithm below picks the smallest number first.

```
int tryout_solution( int number, int num_squares )
{
    int i;

    if ( number == 0 ) {
        // If number == 0 it can be written as the sum of however many 0's we
        // need so we have succeeded.
        return true;
    }

    // assert: number > 0
    if (num_squares == 0 )
        // If we reach here, since number > 0 and we have used up our quota of
        // four squares, we could not find 4 squares whose sum is number
        return false;

    // try to find a number i such that i*i is less than number
    // if we do, then subtract i*i from the number and recursively
    // do the same thing for number-i*i with one less square than before.
    // if one particular i fails, we try the next i. This is the backtracking
    // part.
    for ( i = 1; i*i <= number; i++ ) {
        int square = i*i;
        if ( tryout_solution(number - square, num_squares -1 ) ) {
            printf ( " %d" , square );
            if ( num_squares < 4 )
                printf( " + " );
            return true;
        }
    }
    return false;
}
```

The above function would be called with an initial value of 4 for the second parameter, `num_squares`, as in:

```
if ( tryout_solution( value, 4 ) )
    printf ( " = %d\n", value );
```

The function attempts to find `num_squares` squares that sum to `number`. If `number` is zero, it is trivial to satisfy so it returns `true`. Since each recursive call diminishes `num_squares`, it is possible that it is zero. If `number` is not zero but `num_squares` is zero, it means that it has run out of chances – it has used four squares but their sum is not the original number, so it returns `false`. Otherwise there is still hope – for each square less than `number`, it calls `tryout_solution(number-square, num_squares-1)`, hoping that one of those squares will result in `tryout_solution()` returning `true`. If it does, it means that it found the remaining squares and that `square` is one of the squares that add up to `number`. It prints the value of `square` (with a '+' after if need be), and returns to the calling function. If the main program is named `find_lagrange_squares`, the output could look like

```
$ find_lagrange_squares  2000
1764 +  196 +  36 +  4 = 2000
```

**Example 2: The Eight Queens Problem**

The eight queens problem is based on chess. A chess board is an eight by eight grid of squares. A queen is a piece that can attack another piece if and only if that piece lies in the same row or column as the queen, or along either of the two diagonals through the queen's square. The Eight Queens Problem asks for a set of eight squares on which to place eight queens in such a way that none can attack any other.

There is a natural backtracking solution to this problem. Since there must be exactly one queen in each column of the board, and exactly one queen in each row, every queen must be in its own unique row and column. We arbitrarily use the columns of the board to organize the search. Assume the columns are numbered 1 to 8. Try to think recursively now. Imagine that you have placed queens on the board already and that the queens that have been placed so far cannot attack each other. In other words, so far the queens on the board are a potential solution. Initially this is true because no queens are on the board. You have been placing the queens on the board by putting them in successive columns. Now suppose further that you have a means of checking, for any given potential position in which you want to place the current queen, whether doing so would put it under attack by one of the queens already on the board. The task at the moment is to place the queen in the current column, so you try each row position in that column to see if it leads to a solution. If there is a row that is safe in this column, then you place the queen and recursively advance to the next column, trying to place the next queen there, unless of course you just placed the 8th queen, in which case you found a solution. However, if there is no row in the current column that is safe for the queen, then you have to backtrack – you have to go back to the queen you placed in the preceding column and try a different row for it, repeatedly if necessary, until it is safe, applying this same recursive strategy there. If you backtrack to the very first queen and cannot find a row for it, then there is no solution to this problem.

The following pseudocode function implements this strategy and is a good example of a backtracking algorithm.

```
bool placeQueens( int current column, int current row)
{
    // Base case.  Try to place Queen in a column after end of board
    // if we reach here it means we placed all queens on board
    if ( Current Column >= 8)  {
        successfully placed all queens so exit with success
     }
     bool isQueenPlaced = false;
     while (Queen is not placed in current Column  &&
            Current Row < 8)
     {
         // If the queen can be attacked in this position, then
         // try moving it to the next row in the current column.
         if (isUnderAttack(current row, current column))
             current row = current row + 1;
         else {
            // Queen is not under attack so this position is good.
            // Advance to next column and try starting in row = 0
            isQueenPlaced = placeQueens(current column+1, 0);

            // If it wasn't possible to put the new Queen in the next
            // column, backtrack by deleting the new Queen and
            // removing the last Queen placed and moving it down one row.
```

```
                if (!isQueenPlaced)
                    current row ++;
            } // end if
        } // end while
        return isQueenPlaced;
    } // end placeQueens
```

The details are left to the reader.

## 2 Recursion in the Definition of Languages

Another use of recursion is to define infinite sets of various types. There are always at least two rules. The first is analogous to a base case in an induction proof, and can be called the *basis clause* or simply the *basis*. The second is the *inductive clause*. For example, the set of *natural numbers*, denoted $\mathbb{N}$, can be defined by the following recursive definition:

1. $0 \in \mathbb{N}$.
2. $n \in \mathbb{N} \Longrightarrow n + 1 \in \mathbb{N}$

Repeated application of Rule 2 generates the set of all natural numbers.

Implicit in any definition of a set is that the set contains nothing but what the definition places into it. This does not have to be stated explicitly. The fact that 1.5 is not a natural number is because it is not placed into the set by either of Rules 1 or 2. Some authors make this rule explicit and call it the *extremal clause*.

A more interesting set of numbers is defined by this recursive definition:

1. $0 \in \mathbb{A}$
2. $n \in \mathbb{A} \Longrightarrow 2n + 1 \in \mathbb{A}$

If you apply Rule 2 repeatedly, you will see that this set consists of the numbers 0, 1, 3, 7, 15, 31, and so on, which is the set $\{2^n - 1 | n \in \mathbb{N}\}$.

These two examples show that recursive definitions generate the numbers that are in the set by repeated application of the rules.

### 2.1 Grammars

Sets of words are called *languages*. Every language has an underlying *alphabet*, which is the set of symbols used to form the words. Just as sets of numbers can be generated by recursive definitions, so can sets of words. A recursive definition that defines a language is called a *grammar*. There are various conventions for the notation used in grammars. We will not follow the conventions exactly. As long as the notation is well-defined, it does not matter what it looks like.

### 2.2 Syntax of Grammars.

A *rule* in a grammar is of the form

```
variable = replacement_string
```

which means that the variable on the left can be replaced by the replacement string on the right. The replacement string is a sequence of variables and/or symbols of the underlying alphabet. When a variable appears in the replacement string, it is enclosed in angle brackets (<>) to distinguish it from the non-variables in the replacement. Grammars usually have a designated start variable, which is the one that appears on the left-hand side of the rule and is the first rule to be applied. Some books use a special letter such as S to denote this, but here it is enough to use a symbol that is self-explanatory. When a variable can be replaced by more than one replacement string, we use a vertical bar as a symbol meaning "or". An example of a grammar over the alphabet consisting of the letters `a`, `b`, and `c` is

```
1. PAL      = a <PAL> a | b <PAL> b  | c <PAL> c
2. PAL      = a | b | c
3. PAL      =
```

Rule 2 specifies that `PAL` can be replaced by any of the letters `a`, `b`, and `c`. Rule 3 specifies that `PAL` can be replaced by an empty string (also called the **null string**.) What words are in this language? The null string is in this language, as are "a", "b", and "c". By applying Rule 1 and then Rule 3, "aa", "bb", and "cc" are in it too. For example, to get "aa' we write

```
PAL => a<PAL>a => aa
```

By using the first two rules like this we get nine words: "aaa", "aba", "aca", "bab", "bbb", "bcb", "cac", "cbc", and "ccc." What do these words have in common? They are all the same when read forward or backward. A **palindrome** is a word that is the same read forwards and backwards, such as "radar" or "madam." This language is the language of all palindromes over the alphabet consisting of the letters `a`, `b`, and `c`. This is not a proof of this claim, although the claim is true. Proving that a grammar generates a particular set is beyond the scope of these notes; toprove this you need to show that every word that is generated by it is a palindrome and that every palindrome has a "derivation" from the start symbol `PAL` of this grammar.

**Exercise 1.** Write a recursive function which, when given a C string $s$, returns true or false depending on whether it is a palindrome.

## 2.3 Genetics

The word palindrome in the context of genetics has a slightly definition than this. A **DNA string**, also called a **DNA strand**, is a finite sequence consisting of the four letters `A`, `C`, `G`, and `T` in any order[1]. The four letters stand for the four **nucleotides**: **adenine**, **cytosine**, **guanine**, and **thymine**. Nucleotides, which are the molecular units from which DNA and RNA are composed, are also called *bases*. Each nucleotide has a **complement** among the four: `A` and `T` are complements, and `C` and `G` are complements. Complements are chemically-related in that when they are close to each other, they form **hydrogen bonds** between them. For example, the complement of `TGGC` is `ACCG`, and the complement of `TCGA` is `AGCT`. Notice that this last string has the property that its complement is the same as the string when read backward.

A sequence of nucleotides is **palindromic** if the complement read right to left is the same as the string read from left to right. For example, the DNA string `TGCAACGCGTTGCA` is palindromic because the complement is `ACGTTGCGCAACGT`, which when read backwards is the original string.

**Exercise 2.** Write a recursive function that, when given a DNA string $s$, returns true or false depending on whether $s$ is palindromic. Note that this is different from the preceding exercise.

[1]Some sources use lowercase while others use uppercase. It does not matter.

## 2.4 Infix Expressions

An important class of languages relevant to the programmer are the languages of *algebraic expressions*. Intuitively, an algebraic expression is an expression made up of constants and/or variables upon which the operations of addition, subtraction, multiplication, and division are applied. Parentheses are used to change the order of evaluation in the standard form of these expressions, which are known as *infix expressions*, because the operator is always in between its operands.

A grammar that generates the set of all infix expressions whose operands are single digit numbers is:

```
IE       = <IE> <operator> <IE>
IE       = ( <IE> )
IE       = <token>
operator = + | - | * | /
token    = 0 | 1 | 2 | 3 | ... | 9
```

Examples of words in this language (with spaces added for ease of reading):

```
1 + 8 / 3 - ( 4 - 5 ) * 6
5 + 5 + 4 * 3 * (3 - (2 - (9 + 2) / 2) )
```

A grammar that generates the set of all such expressions whose operands are valid C++ identifiers and/or numeric literals requires more rules; this is a simplification.

## 2.5 Prefix Expressions

Unparenthesized infix expressions are ambiguous in the sense that, unless a precedence is established for the order in which the operators should be applied, an expression could have more than one value. For example,

```
6 + 4 * 3
```

can be interpreted as $6 + (4 * 3) = 18$ or as $(6 + 4) * 3 = 30$ depending upon whether the addition or multiplication takes place first. Operators are given precedence to disambiguate these expressions, and parentheses are used to change the order of evaluation. However, there are unambiguous ways to write algebraic expressions.

A *prefix expression* is one in which the operator precedes its two operands, as in

1. `+ab`
2. `*+abc`
3. `+/ab*-cde`

The first expression is the same as the infix `a+b`. The second applies `*` to the operand `+ab` and the operand `c`, which means that it is equivalent to `(a+b)*c`. The third applies `+` to the operand `/ab` and the operand `*-cde`, which is in turn is `*` applied to `(-cd)` and `e`, which means that it is equivalent to `(a/b)+(c-d)*e`. The general rule is that the operator is applied to the two operands that immediately follow it. The operands may themselves be prefix expressions containing operators, so this procedure is applied recursively.

Prefix expressions are unambiguous under the rules by which they are evaluated. The language of prefix expressions whose operands are single lowercase letters is defined by the following grammar:

```
prefix     = <identifier>
           | <operator> <prefix> <prefix>
operator   = + | - | * | /
identifier = a | b | c | ... | x | y | z
```

Notice that there are no parentheses in these expressions. This leads to recursive algorithms for recognizing and for evaluating prefix expressions. The grammar tells us that a prefix expression is either a single identifier, or it is an operator followed by two prefix expressions. The recognition algorithm should look at the next character, and if

- it is an identifier, it is a prefix, and
- if it is an operator, it has to be followed by two prefix expressions.

The problem is finding where the first prefix ends and the second begins. The key observation is that if you add any characters to the end of a valid prefix expression, you break it – it is no longer a prefix expression. This implies that if we scan across a string and we find the end of a prefix expression, this must be the only end. For example, if we scan `+d*bc`, starting at the `+` character, then the character immediately after `+`, i.e. the `d`, must be the start of the first operand, which is a prefix expression. Since `d` is a prefix expression all by itself, we know it ends there; no other characters can be part of the first operand of `+`. Similarly, when we scan further and see the `*` we begin to look for two more prefix expressions. If we find that the first ends at the `b`, we know that the next character (the `c`) is the start of the second prefix expression.

Another example:

```
+/ab-cd
```

If this is a prefix it is of the form $+E_1E_2$ where $E_1$ and $E_2$ are both prefix expressions. Since $E_1$ begins with /, it is of the form $/E_3E_4$ where $E_3 = a$ and $E_4 = b$. Also, $E_2$ is of the form $-E_5E_6$ where $E_5$ =c and $E_6 = d$. The key is therefore to write a function that finds the end of a prefix expression.

### 2.5.1 Recognizing Prefix Expressions

An algorithm to find the end of a prefix expression:

```
int end_of_prefix(const string & prefix_str, const int first)
{
    int last = prefix_str.length() - 1; // index of last character in string
    if (first < 0 || first > last )     // first is out of range
        return -1;

    char ch = prefix_str[first];        // get character at position first
    if ( is_identifier(ch) )            // if an identifier
        return first;                   // return first since it is also the
                                        // end of its own prefix
    else if ( is_operator(ch))   {
        // recursive call to find end of prefix that starts at the character
        // immediately after first
        int first_end = end_of_prefix(prefix_str, first + 1);

        // check if the call was able to find an end
        // Return of -1 means it failed
        if (first_end > -1)
```

```
                // It succeeded, so return the end of the prefix after it, which
                // starts at first_end+1
                return end_of_prefix(prefix_str, first_end + 1);
            else return -1;
        }
        else
            return -1;
    }
```

Given the preceding algorithm, it is trivial to check whether a string is a prefix expression:

```
    bool is_prefix(string str)
    {
        last_char = end_of_prefix(str, 0);
        return ( last_char >= 0 && last_char == str.length() -1);
    }
```

### 2.5.2 Evaluating Prefix Expressions

A recursive algorithm that evaluates prefix expressions:

```
    // This algorithm modifies its argument in the process of evaluating it.
    float evaluate_prefix(string & prefix_str)
    {
        char ch = prefix_str[first];        // get character at position first

        // Delete first character from prefix_str;
        prefix_str = prefix_str.substr(first+1);
        if ( is_identifier(ch) )
            return value of the identifier;

        // if the character is an operator, then
        else if ( is_operator(ch))   {
            op = ch;
            operand1 = evaluate_prefix(prefix_str);
            operand2 = evaluate_prefix(prefix_str);
            return operand1 op operand2 ;
        }
```

## 2.6 Postfix Expressions

Another form of algebraic expression that is also unambiguous in the sense described above is called a ***postfix expression***. In postfix, the operator follows immediately after its operands. The table below shows the prefix expressions from above with their infix and postfix equivalences.

| **Prefix** | **Infix** | **Postfix** |
|---|---|---|
| `+ab` | `a+b` | `ab+` |
| `*+abc` | `(a+b)*c` | `ab+c*` |
| `+/ab*-cde` | `(a/b)+((c-d)*e)` | `ab/ cd-e*+` |

Postfix expressions are used by many calculators. They are also used when a compiler generates assembly code from higher-level code. A *postfix expression* over the alphabet of single lowercase letter operands and the standard operators is defined by the following grammar:

```
postfix     = <identifier>
            |  <postfix> <postfix> <operator>
operator   = + | - | * | \
identifier = a | b | c | ... | x | y | z
```

Here are a few more examples of postfix expressions and their equivalent infix expressions:

| **Postfix** | **Equivalent Infix** |
|---|---|
| `a b + c *` | `(a + b) * c` |
| `a b c d e - - - -` | `a - (b - (c - (d - e)))` |
| `a b * c d * e f * + -` | `(a * b)- ((c * d) + (e * f))` |

Like the prefix grammar, this leads to recursive algorithms for recognizing and for evaluating postfix expressions. When we cover stacks, we will see non-recursive algorithms developed using stacks that evaluate postfix expressions and convert infix to postfix.

# Stacks

## 1 Introduction

Stacks are probably the single most important data structure of computer science. They are used across a broad range of applications and have been around for more than fifty years, having been invented by Friedrich Bauer in 1957.

A ***stack*** is a list in which insertions and deletions are allowed only at the front of the list. The front in this case is called the ***top***, insertions are called ***push*** operations, and deletions are called ***pop*** operations. Because the item at the top of the stack is always the one most recently inserted into the list, and is also the first one to be removed by a pop operation, the stack is called a ***last-in-first-out*** list, or a ***LIFO*** list for short, since the last item in is the first item out.

The restriction of insertions and deletions to the front of the list is not the only difference between stacks and general lists. Stacks do not support searching or access to anywhere else in the list. In other words, the only element that is accessible in a stack is the top element.

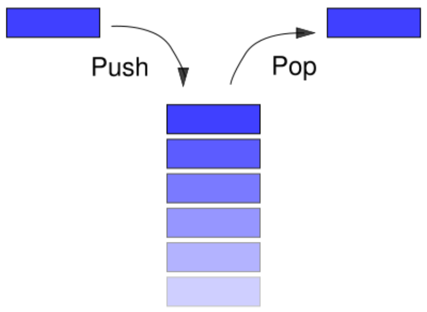


Figure 1: A stack

## 2 The Stack ADT

The fundamental methods that can be performed on a stack are

**create()** – Create an empty stack.

**destroy()** – Destroy a stack.

**empty()** – Return true if the stack is empty, otherwise return false.

**push(new_item)** - Add a new item to the top of the stack.

**item pop()** – Remove the top element from the stack and return it to the caller.

**item top()** – Return the top element of the stack without removing it.

The reason that the word "stack" is appropriate for this abstract data types is that it is exactly like a stack of trays in a cafeteria, the kind that are spring-loaded. New trays (presumably washed) are put on top of the stack. The next tray to be removed is the last put on. This is why the operations are called pop and push, because it conjures up the image of the spring-loaded stack of trays. We literally push against the spring, which wants to pop one off.

## 3 Refining the Stack ADT

The operations, written more formally using a slight modification of UML, are:

```
+empty():boolean  {query}
// returns true if the stack is empty, false if not
+push(in new_item:StackItemType): void  throw stack_exception
// pushes new_item on top of the stack; throws a stack_exception if
// it fails
+pop(): void  throw stack_exception
// removes the top element from the stack; if it fails
// it throws an exception (if the stack is empty for example)
+pop(out stack_top:StackItemType): void  throw stack_exception
// same as pop() above but stores the item removed into stack_top,
// throwing an exception if it fails
+top( out stack_top:StackItemType): void {query} throw stack_exception
// stores into stack_top the item on top of the stack,
// throwing an exception if it fails.
```

**Notes.**

1. Notice that there are two versions of `pop()`, one that has no parameters and no return value, and one that has an *out* parameter and no return value. In theory, when the stack is popped, we think of it as a deletion operation, discarding whatever was on the top of the stack. In practice, we may sometimes want to inspect the item that was deleted, and so the second version of `pop()` could be used to retrieve that top element and look at it after we have removed it from the stack.
2. The `top()` method has an out parameter. It could instead return the top element as its return value, but it is considered better coding style to pass it out through the parameter list, as otherwise it could be used in an expression with side effects.
3. The `pop()` and `top()` methods declare that they may throw an exception of type `stack_exception`, and that this is the only type of exception that they may throw. If a program tries to pop an empty stack, or to retrieve the top element of an empty stack, this is considered to be an error condition. Now that we have added C++ exception handling to our toolbox (in the preceding chapter), we can use it as a means of notifying the client software that an error has occurred.
4. The `push()`, also may throw an exception. No computer has infinite memory, and it is possible that when the stack tries to allocate more memory to store this new item, it fails. The exception in this case would indicate that the stack is "full."

## 4 Applications of Stacks

Stacks are useful for any application requiring LIFO storage. There are many, many of these.

- parsing context-free languages
- evaluating arithmetic expressions
- function call management
- traversing trees and graphs (such as depth first traversals)
- recursion removal

## Example: A Language Consisting of Balanced Brackets

A stack can be used to recognize strings from the language of ***balanced brackets***. Brackets are either parentheses, curly braces, square brackets, angle brackets, or any other pair of characters designated to be left and right matches of each other, as well as characters that are not brackets of any kind. Brackets are balanced when they are used in matching pairs. To be precise, the following grammar defines a language of balanced brackets whose alphabet is the four different types of brackets and the letter `a`. The word "`NULLSTRING`" means the string with no characters in it.

```
word       = identifier
           | NULLSTRING
           | word word
           | (word) | <word> | [word] | {word}
identifier = a
```

The grammar generates the null string of course, as well as the word with the single letter 'a'. You can check that the following two-character words are also in the language that this grammar generates:

```
() {} [] <>
```

because there are derivations such as

```
word => (word) => (NULLSTRING) => ()
```

and similar ones for the other brackets. In general, this grammar generates words such as

```
aa(aa)[a(aa<aaa>a)aaaaaa]aa{aa{aaa}{aa{aaa}}aa}
```

Intuitively, these are words that never have overlapping brackets such as `[ ( ] )` or unmatched brackets such as `(aaa`.

The simplest bracket language has just a single type of bracket. Suppose we allow all lowercase letters into the alphabet but limit the brackets to curly braces. An example of a word with balanced braces is:

```
abc{defg{ijk}{l{mn}}op}qr
```

An example of a word with unbalanced braces is:

```
abc{def}}{ghij{kl}m
```

The following is an algorithm to recognize words in this language. **Recognizing** means identifying which words are in the language and which are not. A **recognition algorithm** returns either true or false, depending on whether the word is in the language.

```
initialize a new empty stack s whose entries hold a single character.
balanced = true;
while ( the end of the string has not been reached ) {
    copy the next character of the string into ch;
    if ( ch == "{" )
        s.push(ch);
    else if ( ch == "}" ) {
```

```
                if s.empty()
                    return !balanced;
                else
                    s.pop();
            }
    }
    if s.empty()
       return balanced;
    else
       return !balanced;
```

Of course this is a simple problem that can be solved with just a counter; increment the counter on reading a "{" and decrement on reading a "}". If the counter is ever negative, it is not balanced. If it is not zero when the end of the string is reached, it is not balanced. Otherwise it is balanced.

**Exercise 1.** Write and implement a recognition algorithm for the above language that just uses a counter rather than a stack.

The stack is needed when there are different types of matching parentheses. For example, suppose the string can have square brackets, curly braces, and regular parentheses, as in:

```
(abc)[ {d (ef) g{ijk}{l{mn}}op} ] qr
```

Then a counter solution will not work but a stack will. The idea is that when a right bracket is found, the stack is popped only if the bracket on top of the stack matches it. If it does not match, the string is not balanced. If the stack is empty, it is not balanced. When the entire string has been read, if the stack is not empty, it is not balanced. Assume that the function `matches(ch1, ch2)` returns true if `ch1` and `ch2` are matching brackets of any kind.

```
initialize a new empty stack s whose entries hold a single character.
balanced = true;
while ( the end of the string has not been reached ) {
    copy the next character of the string into ch;
    if ( ch == '[' || ch == '(' || ch == '{' )
        s.push(ch);
    else
        switch ( ch ) {
        case ']' :
        case ')' :
        case '}' :
           if s.empty()
               return !balanced;
           else {
               s.top( bracket );
               s.pop();
               if ( !matches(bracket, ch) )
                   return !balanced;
           }
           break;
        default: // a non-bracket -- ignore
    }
}
if s.empty()
   return balanced;
else
   return !balanced;
```

## 5 Stack Implementations

We did not need to know how a stack was implemented to use the stack to solve this problem. This is the power of data abstraction. But before we look at other applications, we will consider how one can implement a stack in C++. To do this, we need to make the interface precise enough so that we can implement the operations. In addition, because we will start to use exceptions to handle error conditions, we will create a class of stack exceptions that can be thrown as needed. If you are not familiar with exceptions, refer to the notes on exception handling.

The following exception class will be used by all stack implementations.

```
#include <stdexcept>
#include <string>
using namespace std;
class stack_exception: public logic_error
{
public:
    stack_exception(const string & message="")
         : logic_error(message.c_str())    {}
};
```

The `logic_error exception` is defined in `<stdexcept>`. Its constructor takes a `string`, which can be printed using the `what()` method of the class. Our `stack_exception` class is derived from this `logical_error` class and inherits the `what()` method. The constructor for our `stack_exception` can be given a string, which will be passed to the `logical_error` constructor as its initializing string.

The next logical step could be to derive separate exception types such as `stack_overflow`, or `stack_underflow`, as illustrated below.

```
class stack_overflow: public stack_exception
{
public:
    stack_overflow(const string & message=""): stack_exception(message) {}
};

class stack_underflow: public stack_exception
{
public:
    stack_underflow(const string & message=""): stack_exception(message) {}
};
```

but for simplicity we will not do this. The following is the specific interface we will implement:

```
typedef data_item StackItemType;
class Stack
{
public:
    // constructors and destructor:
    Stack();               // default constructor
    Stack(const Stack &);  // copy constructor
    ~Stack();              // destructor
```

```
        // stack operations:
        bool empty() const;
        // Determines whether a stack is empty.
        // Precondition: None.
        // Postcondition: Returns true if the stack is empty;
        // otherwise returns false.

        void push(const StackItemType& new_item) throw(stack_exception);
        // Adds an item to the top of a stack.
        // Precondition: new_item is the item to be added.
        // Postcondition: If the insertion is successful, new_item
        // is on the top of the stack.
        // Exception: Throws stack_exception if the item cannot
        // be placed on the stack.


        void pop() throw(stack_exception);
        // Removes the top of a stack.
        // Precondition: None.
        // Postcondition: If the stack is not empty, the item
        // that was added most recently is removed. However, if
        // the stack is empty, deletion is impossible.
        // Exception: Throws stack_exception if the stack is empty.


        void pop(StackItemType& top_item) throw(stack_exception);
        // Retrieves and removes the top of a stack.
        // Precondition: None.
        // Postcondition: If the stack is not empty, top_item
        // contains the item that was added most recently and the
        // item is removed. However, if the stack is empty,
        // deletion is impossible and top_item is unchanged.
        // Exception: Throws stack_exception if the stack is empty.


        void top(StackItemType& top_item) const
              throw(stack_exception);
        // Retrieves the top of a stack.
        // Precondition: None.
        // Postcondition: If the stack is not empty, top_item
        // contains the item that was added most recently.
        // However, if the stack is empty, the operation fails
        // and top_item is unchanged. The stack is unchanged.
        // Exception: Throws stack_exception if the stack is empty.
    private:
       ...
    };
```

There are several different ways to implement a stack, each having advantages and disadvantages.

Since a stack is just a special type of list, a stack can be implemented with a list. In this case, you can make a list a private member of the class and implement the stack operations by calling the list methods on the private list. In other words, we could do something like

```
    class Stack
```

```
{
public:
    // public methods here
private:
    List items;
};
```

We could implement all of the member functions by using the `List` member functions of the hidden `List` named items.

It is more efficient to implement a stack directly, avoiding the unnecessary function calls that result from "wrapping" a stack class around a list class. Direct methods include using an array, or using a linked representation, such as a singly-linked list.

A stack implemented as an array is efficient because the insertions and deletions do not require shifting any data in the array. The only problem is that the array size might be exceeded. This can be handled by a resizing operation when it happens.

A stack implemented as a linked list overcomes the problem of using a fixed size data structure. It is also relatively fast, since the push and pop do not require many pointer manipulations. The storage is greater because the links take up extra bytes for every stack item.

## 5.1 Array Implementation

In the array implementation, two private data members are needed by the class: the array of items and an integer variable named `top.`

```
class Stack
{
public:
    // same interface as above
private:
    StackItemType  items[MAX_STACK];   // array of MAX_STACK many items
    int            top_index;          // indicates which index is the top
};
```

**Remarks.** The implementation is very straightforward:

- The variable `top_index` stores the array index of the current top of the stack. We cannot name it `top` because there is a method named `top`. Because 0 is the first array index, we cannot use 0 to mean that the stack is empty, because if `top_index == 0`, it means there is an item in `items[0]`. Therefore, -1 is the value of `top_index` that denotes the empty stack. This implies that the constructor must initialize `top_index` to -1, and the `empty()` function must check if it is -1 as well.
- The `push()` operation has to check if the maximum array size will be exceeded if it adds a new item. If so, it has to throw an exception to indicate this. If not, it increments `top` and copies the new item into `items[top_index]`.
- The `pop()` operation must ensure that the stack is not empty before it decrements `top_index`. There are two versions of `pop()`, the only difference being that one provides the top item before decrementing `top`.
- The `top()` operation returns the top item, throwing a stack exception in the event that the stack is empty.

```
Stack::Stack(): top_index(-1) {}


bool Stack::empty() const
{
    return top_index < 0;
}

void Stack::push(StackItemType new_item) throw(stack_exception)
{
    // if stack has no more room for another item
    if (top_index >= MAX_STACK-1)
        throw stack_overflow("stack_exception: stack full on push");
    else  {
        ++top_index;
        items[top_index] = new_item;
    }
}


void Stack::pop() throw(stack_exception)
{
    if ( empty()  )
        throw stack_exception("stack_exception: stack empty on pop");
    else
       --top_index;
}


void Stack::pop(StackItemType& top_item) throw(stack_exception)
{
    if ( empty()  )
        throw stack_exception("stack_exception: stack empty on pop");
    else {  // stack is not empty;
        top_item = items[top_index];
       --top_index;
    }
}


void Stack::top(StackItemType& top_item) const throw(stack_exception)
{
    if (empty())
        throw stack_exception("stack_exception: stack empty on top");
    else         // stack is not empty; retrieve top
        top_item = items[top_index];
}
```

## 5.2 Linked Implementation

A linked list implementation of a stack does not pre-allocate storage for the stack; it allocates nodes as needed. Each node will have a data item and a pointer to the next node. Because the linked list implementation hides the fact that it is using linked nodes, the node structure is declared within the private part of the class.

The only private data member of the Stack class is a pointer to the node at the top of the stack, which is the node at the front of the list. If the list is empty, the pointer is NULL. The private part of the Stack class would therefore be:

```
private:
   struct StackNode
   {
      StackItemType item;
      StackNode    *next;
   };
   StackNode *top_ptr;     // pointer to first node in the stack
};
```

**Remarks.**

- The constructor simply sets top_ptr to NULL, and the test for emptiness is whether or not top_ptr is NULL.
- The copy constructor has to traverse the linked list of the stack passed to it. It copies each node from the passed stack to the stack being constructed. It allocates a node, fills it, and attaches it to the preceding node
- The destructor repeatedly calls pop() to empty the list.
- The push() operation inserts a node at the front of the list.
- The two pop() operations remove the node at the front of the list, throwing an exception if the stack is empty. One provides this item in the parameter.
- Similarly, top() simply returns the item at the front of the list, throwing an exception if the stack is empty.
- In theory it is possible that the operating system will fail to provide additional memory to the object when it calls new() in both the copy constructor and the push() method, and this failure should be detected in a serious application.

```
// constructor
Stack::Stack() : top_ptr(NULL) { }

// copy constructor
Stack::Stack(const Stack& aStack) throw (stack_overflow)
{
    if (aStack.top_ptr == NULL)
        top_ptr = NULL;         // empty list
    else {
        // copy first node
        top_ptr          = new StackNode;
        if ( NULL == top_ptr )
            throw stack_overflow("copy constructor could not allocate node");
        top_ptr ->item = aStack.top_ptr->item;

        // copy rest of list
        StackNode *newPtr = top_ptr ;    // new list pointer
        for (StackNode *origPtr = aStack.top_ptr ->next; origPtr != NULL;
            origPtr = origPtr->next) {
            newPtr->next = new StackNode;
            if ( NULL == newPtr )
                throw stack_overflow("copy constructor could not allocate node");
            newPtr         = newPtr->next;
            newPtr->item = origPtr->item;
            }
            newPtr->next = NULL;
        }
    }

    // destructor, which empties the list by calling pop() repeatedly
    Stack::~Stack()
    {
        // pop until stack is empty
        while (!empty())
            pop();
    }

    bool Stack::empty() const
    {
        return top_ptr == NULL;
    }

    void Stack::push(StackItemType newItem)  throw(stack_exception)
    {
        // create a new node
        StackNode *newPtr = new StackNode;
        if ( NULL == newPtr )
             throw stack_overflow("push could not allocate node");
        // set data portion  of new node
        newPtr->item = newItem;

        // insert the new node
        newPtr->next = top_ptr;
        top_ptr = newPtr;
    }

    void Stack::pop() throw(stack_exception)
    {
        if (empty())
            throw stack_exception("stack_exception: stack empty on pop");
        else { // stack is not empty;
            StackNode *temp = top_ptr;
            top_ptr = top_ptr->next;
            temp->next = NULL;
            delete temp;
        }
    }

    void Stack::pop(StackItemType& stackTop) throw(stack_exception)
    {
         if (empty())
             throw stack_exception("stack_exception: stack empty on pop");
         else {  // stack is not empty; retrieve and delete top
             stackTop = top_ptr->item;
             StackNode *temp = top_ptr;
             top_ptr = top_ptr->next;

             // return deleted node to system
             temp->next = NULL;  // safeguard
             delete temp;
        }
    }

    void Stack::top(StackItemType& stackTop) const throw(stack_exception)
    {
        if (empty())
            throw stack_exception("stack_exception: stack empty on top");
        else        // stack is not empty; retrieve top
            stackTop = top_ptr->item;
    }
```

**Performance Considerations** A careful look at this implementation shows that the `push()`, `pop()`, `top()`, and `empty()` operations take a constant amount of time, independent of the size of the stack. In contrast, the destructor executes a number of instructions that is proportional to the number of items in the stack, and the copy constructor's running time is proportional to the size of the stack being copied.

## 5.3 The Standard Template Library Class stack[1]

### 5.3.1 About the Standard Template Library

The *Standard Template Library* (*STL*) contains, among other things, a collection of C++ class templates for the most commonly used abstract data types. The STL is part of all standard implementations of C++. A C++ class template is not a class, but a *template for a class.* A simple class template interface has the general form

```
template <typename T>
class Container
{
public:
    Container();
    Container( T initial_data);
    void set( T new_data);
    T get() const;
private:
    T mydata;
};
```

The T after the keyword `typename` in angle brackets is a placeholder. It means that all occurrences of the T in the interface will be replaced by the actual type that instantiates the template when it is used to define an object. In the code above, it is used in four different places.

The implementations of the class template member functions have the form

```
template <typename T>
void Container<T>::set ( T initial_data )
{
    mydata = new_data;
}

template <typename T>
T Container<T>::get() const
{
    return mydata);
}
```

### 5.3.2 The stack Template Class

To use the `stack` template class, you would write

```
#include <stack>
```

[1]The Standard Template Library was originally implemented as an independent library, but was eventually incorporated into the C++ standard.

at the top of the file using the `stack` template. The `stack` template class has the following member functions[2]:

```
bool empty() const;
size_type size() const;
T & top();
void pop();
void push( const T & x);
```

To declare an instance of a `stack` template in a program you would write `stack`<*actual type name*> *variable_name*, as in:

```
stack<int>    int_stack;    // stack contains ints
stack<string> string_stack; // stack contains strings
stack<char>   char_stack;   // stack contains chars.
```

Therefore, if you want to use the STL `stack` class instead of writing your own, you would declare a stack of the needed type, and make ordinary calls upon the member functions of the class, as in

```
stack<int> mystack;
int number;
while ( cin >> number ) {
    mystack.push(number);
}
```

The syntax for the function calls is the same whether it is a class template instantiation or not. Once it is instantiated, it looks like an ordinary class.

## 6 More Applications

### 6.1 Postfix Expressions

Remember that a postfix expression is either a simple operand, or two postfix expressions followed by a binary operator. An operand can be an object or a function call. The object can be constant or variable. Using letters of the alphabet as operands, an example of a postfix expression is

```
ab+cd-/e*f+
```

which is equivalent to the infix expression

```
((a+b)/(c-d))*e +f
```

#### 6.1.1 Evaluating Postfix Expressions

Postfix is often used in command line calculators. Given a *well-formed*[3] postfix expression, how can we use a stack to evaluate it without recursion? To start, let us assume that the only operators are left-associative, binary operators such as `+`, `-`, `*`, and `/`. The idea is that the stack will store the operands as we read the string from left to right. Each time we find an operator, we will apply it to the two topmost operands, and push the result back onto the stack. The algorithm follows. The word "token" below means any symbol found in the string, excluding whitespace characters (if they are present.)

[2]In C++98 this is the complete set of member functions. Two others were added in 2011
[3]A well-formed string is one that can be generated by the grammar, i.e. is in the correct form.

```
create an empty stack that holds the type of values being computed
for each token tok in the string {
    if tok is an operand
        stack.push(tok);
    else if tok is an operator {
         let op denote the operation that tok provides
         stack.pop(right_operand);
         stack.pop(left_operand);
         result = left_operand op right_operand;
         stack.push(result);
    }
}
```

We will turn the stack on its side with the top facing *to the right* to demonstrate the algorithm. We will let the input string consists of single-digit operands and use blanks to separate the tokens for clarity. Suppose the following is the input:

```
8 6 + 9 2 - / 5 * 7 +
```

Each row of the following table shows the result of reading a single token from the input string and applying the body of the for-loop to it. Since the string has 11 tokens, it will take 11 iterations of the loop to compute the value of the expression.

| **Stack (right end is top)** | **Input Remaining** | **Action to Take** |
|---:|---|---|
| (*empty*) | `8 6 + 9 2 - / 5 * 7 +` | push 8 on stack |
| 8 | `6 + 9 2 - / 5 * 7 +` | push 6 on stack |
| 8 6 | `+ 9 2 - / 5 * 7 +` | 8+6 = 14; push on stack |
| 14 | `9 2 - / 5 * 7 +` | push 9 on stack |
| 14 9 | `2 - / 5 * 7 +` | push 2 on stack |
| 14 9 2 | `- / 5 * 7 +` | 9 - 2 = 7; push 7 on stack |
| 14 7 | `/ 5 * 7 +` | 14 / 7 = 2; push 2 on stack |
| 2 | `5 * 7 +` | push 5 on stack |
| 2 5 | `* 7 +` | 2*5 = 10; push 10 on stack |
| 10 | `7 +` | push 7 on stack |
| 10 7 | `+` | 10+7=17; push 17 on stack |
| 17 | | end-of-input; answer = 17 |

It is not hard to refine this algorithm and convert it to C++. This is left as an exercise.

### 6.1.2 Converting Infix to Postfix

We can also use a stack to convert infix expressions to postfix[4]. This time, though, the stack will store the *operators*, not the operands. Let us assume that the input string is well-formed and can contain any of the operators `+`, `-`, `*`, and `/` as well as parentheses, and that the operands are valid tokens, such as numeric literals and variables. We will use the term *simple operand* to refer to things like numeric literals and variables, and *compound operand* to refer to an operand made up of at least one operator or enclosed in parentheses, such as `(a)`.

[4]This is what most modern compilers do in order to create the machine instructions for these expressions

**Some Observations**

- The order of the operands is the same in postfix as it is in the infix expression. In other words, if `b` precedes `c` in the infix expression then b precedes `c` in the postfix as well. The operators may move relative to the operands, but not the operands with respect to each other. (This can be proved by induction on the size of the infix expression.)
- If, in an infix expression, the operand `b` precedes the operator `op`, then in the postfix expression, `b` will also precede `op`. Another way to say this is that the operators can only move to the right relative to the operands, never to the left.
- While the infix expression has parentheses, the postfix does not.
- For every '`(`' in the infix expression, there is a matching '`)`'. The sub-expression within these parentheses is a valid infix expression consisting of a left operand, an operator, and a right operand. These operands may be infix expressions that are not tokens, but regardless of whether they are simple or compound, the operands of all operators within the parentheses are contained within the parentheses also. This implies that the infix expression contained within a set of matching parentheses can be processed independently of what is outside those parentheses.

Since the order of the operands remains the same in the postfix as it was in the infix, the problem boils down to deciding where to put the operators in the postfix. These observations lead to a set of rules for making these decisions.

We initialize an empty postfix expression before we begin. When we see a token that is a simple operand, we append a copy of it immediately to the right end of the output postfix expression. When we see an operator, we know that its right operand begins at the very next token, and so we scan across the infix string looking for the end of the right operand. When we find the end of the right operand, we append the operand to the output postfix and we put the operator after it.

The problem is complicated by the fact that the operators have varying precedence. Therefore, the end of the right operand will depend on the precedence of the operators that we find along the way. Thus, the expression

```
a+b*c
```

has a postfix of the form `abc*+` whereas the expression

```
a+b-c
```

has a postfix of the form `ab+c-`. In the first case, the right operand of the `+` is `b*c`, but in the second case, it is just `b`. When we read the `+`, we push it onto the stack, copy the `b` to the output postfix, and then we look at the next operator. If the precedence of the next operator is less than or equal to that of the `+`, we know that the `b` is the end of the right operand of `+` and we can pop the `+` and put it after the `b`. But if the operator has greater precedence, then the right operand is a compound operand whose end we have not yet found. For example, when reading the infix expression

```
a+b*c*d*e
```

after reading the `b`, we will find that every operator has greater precedence than `+`. The right operand of the `+` is the entire expression `b*c*d*e`, and the end of the right operand will be the `e`.

It will be easier to design the algorithm if we begin by assuming that the infix expressions do not have parentheses. In this case they are of the form

$$a_1 o_1 a_2 o_2 a_3 o_3 \cdots o_{n-1} a_n$$

where the $o_j$ are operators and the $a_j$ are simple operands. Assume that each time we read an operand, we append it to the right end of the output postfix string. Given a string like this, having just read an operator $o_j$from the string, how do we recognize the end of the right operand of $o_j$? We have found the end of its right operand if and only if

1. the token we have just read is an operator whose precedence is less than or equal to precedence($o_j$), or
2. we reached the end of the input string.

In either case, the last operand that was appended to the right end of the output postfix string is the end of the right operand of $o_j$. We use the stack as a holding pen for the operators while we search for their right operands. To be precise, the invariant that will always be true is:

> If the stack is not empty, then the operator on top of the stack will have the property that its left operand is the most recently appended operand in the output postfix, but the end of the right operand has yet to be found.

Therefore, when we find the end of the right operand of the operator on the top of the stack, we pop that operator off of the stack and append it to the output postfix after its right operand. (We can argue by induction on the size of the stack that when we do this, the operator that is now at the top of the stack still satisfies the invariant.) This leads to the following algorithm:

```
While there are tokens to read from the infix string {

    Read a token from the infix string.
    If the token is an operand,
        append a copy to the postfix string.
    Otherwise, {
        if the token is an operator, then
            check if the stack is empty or not.
            If the stack is empty, {
                push the token onto the stack.
                // Notice that the invariant is now satisfied because this
                // is an operator whose right operand is not yet found
                // and its left operand was just appended to the postfix
            }
            Otherwise, {
                While the stack is not empty and
                    precedence(token) ≤ precedence(top)  {
                    pop the operator on top of the stack and
                    append it to the output postfix
                    // each operator that is popped has the property that we
                    // found the end of its right operand and its left operand
                    // is the most recently appended to the postfix
                }
                push the token onto the stack.
               (Notice that the invariant is now satisfied.)
            }
    }

}
While the stack is not empty {
        pop the operator on top of the stack and
        append it to the output postfix
    }
```

You can verify that the invariant will always be true at the end of each iteration of the outermost while-loop. When that loop ends, by Rule 2, the end of the right operand of every operator in the stack has been found, and these must be popped and appended to the postfix. It is because of the invariant that the topmost belongs to the left of the operators beneath it, because each time we pop an operator and append it, it forms a compound postfix expression that is the right operand of the operator now on top of the stack. (Think this through carefully.)

The following table demonstrates the above algorithm at work on the input string `a+b*c*d-e`.

| Output Postfix | Stack | Input Infix Expression |
|---|---|---|
| | | a+b*c*d-e |
| a | | +b*c*d-e |
| a | + | b*c*d-e |
| ab | + | *c*d-e |
| ab | +* | c*d-e |
| abc | +* | *d-e |
| abc* | + | *d-e |
| abc* | +* | d-e |
| abc*d | +* | -e |
| abcd** | + | -e |
| abcd**+ | | -e |
| abcd**+ | - | e |
| abcd**+e | - | |
| abcd**+e- | | |

This algorithm will work fine as long as there are no parentheses in the string. Adding parentheses is not much more work. Suppose that a pair of parentheses encloses a part of the infix expression, as in

```
a+b*(c+d-e*f)+g
```

Since the expression is assumed to be well-formed, the expression contained by the parentheses is also well-formed. Thus, we can treat this substring as a separate string that starts after the left parenthesis and that ends at the right parenthesis, and we can process it independently, treating the right parenthesis like the end-of-input character. Rules 1 and 2 apply to it as well, so we know that when we see the right parenthesis, we have found the end of the right operand of the operator on top of the stack.

We could, if we wanted, create a separate stack each time we found a left parenthesis, and process the infix contained within in that stack and then discard it. But this is inefficient. Instead we can use the single stack to process the parenthesized sub-expression, but putting a marker into the stack to denote the new "bottom". We can use the left parenthesis itself as a marker. Therefore, we put the left parenthesis into the stack when we find it. Our algorithm above will be modified so that the loop that compares the top of stack to the current token stops if, when it pops the stack, it sees the left parenthesis, since that implies we are within a parenthesized sub-expression. Additionally, when we see a right parenthesis, it will act like an end-of-string character for the current sub-expression, and so we will pop each operator off of the stack and append it to the output postfix until we see the left parenthesis, which will terminate that loop.

All of these considerations lead to the following algorithm:

```
create an empty stack to hold operands
for each token tok in the input infix string {
    switch (tok) {
        case operand:
            append tok to right end of postfix;
            break;
        case '(' :
            stack.push(tok);
            break;
        case ')':
            while ( stack.top() != '(' ) {
                append stack.top() to right end of postfix;
                stack.pop();
            }
            break;
        case operator:
            while ( !stack.empty() && stack.top() != '('
                    && precedence(tok) <= precedence(stack.top()) ) {
                append stack.top() to right end of postfix;
                stack.pop();
            }
            stack.push(operator);
            break;
    }
}
while ( !stack.empty() ) {
    append stack.top() to right end of postfix;
    stack.pop();
}
```

**Example**

The following table shows how the infix expression `a-(b+c*d)/e` would be processed by the above algorithm.

| Output Postfix | Stack | Input Infix Expression |
|---|---|---|
| | | `a-(b+c*d)/e` |
| `a` | | `-(b+c*d)/e` |
| `a` | `-` | `(b+c*d)/e` |
| `a` | `-(` | `b+c*d)/e` |
| `ab` | `-(` | `+c*d)/e` |
| `ab` | `-(+` | `c*d)/e` |
| `abc` | `-(+` | `*d)/e` |
| `abc` | `-(+*` | `d)/e` |
| `abcd` | `-(+*` | `)/e` |
| `abcd*` | `-(+` | `)/e` |
| `abcd*+` | `-(` | `)/e` |
| `abcd*+` | `-` | `/e` |
| `abcd*+` | `-/` | `e` |
| `abcd*+e` | `-/` | |
| `abcd*+e/` | `-` | |
| `abcd*+e/-` | | |

# Index

# Queues

## 1 Introduction

A queue is a very intuitive data type, especially among civilized societies. In the United States, people call "lines" what the British call *queues*. In the U.S., people stand "in line" for services such as purchasing a ticket for one thing or another, paying for merchandise, or boarding a train, bus or plane. The British stand in queues to do the same thing. What characterizes these queues is that arriving customers always go to the "end of the line" and the next customer to be served is always taken from the "front" of the line. As customers are served, the other customers steadily get closer to the front of the line, in the order in which they arrived. The idea that people are served in the order in which they arrive, or to put it another way, that *the first one in is the first one out*, is the notion of fairness implicit in queues.

Formally, in computer science terminology, a queue is a list in which all insertions take place at the end of the list and all deletions and accesses take place at the front of the list. The end of the list is called the ***rear***, or sometimes the ***back***, of the queue. The front keeps its name. Because this leads to a first-in, first-out behavior, a queue is known as a ***FIFO list***.

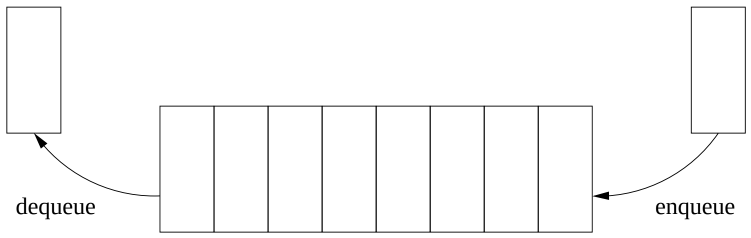


Figure 1: Inserting into and deleting from a queue

## 2 The Queue ADT

Operations on queues are analogous to operations on stacks. There is a one-to-one correspondence between them. The only difference is that the push operation of the stack appends an item to the front of the stack (which we call the top), whereas the ***enqueue*** operation appends to the rear. Popping a queue is called ***dequeuing*** the queue. Other than its having a different name, dequeuing a queue is the same as popping a stack. The single difference between stacks and queues, namely which end of the list new items are inserted, has a major consequence in terms of how the queue abstract data type behaves. See Figure 1.

The queue methods are:

**create()** - Create an empty queue.

**destroy()** - Destroy a queue.

**bool empty()** - Return true if the queue is empty and false if not.

**bool enqueue([in] item)** - Append an item to the rear of the queue, returning true if successful, false if not.

**bool dequeue([out] item)** - Remove the item from the front of the queue, and provide it to the caller, returning true if successful and false otherwise.

**bool dequeue()** - Remove the item from the front of the queue, returning true if successful and false otherwise.

**item front()** - Return the item at the front of the queue to the caller without removing it

**size_type size()** - Return the number of items in the queue.

**Notes.**

- There is no access to an item that is not at the front.
- The `dequeue()` operation is analogous to the stack's `push()` operation, and it is convenient to have two forms of it: one that removes the front item without retrieving it, and another that removes it but also copies it into an out parameter to make it available to the caller.
- `enqueue()` is called `push` in the C++ standard queue template class.
- `dequeue()` is called `pop` in the C++ standard queue template class.
- The `front()` operation returns the item at the front of the queue. This is the way that most people expect it to be. The problem is how to deal with calling `front()` when the queue is empty.

## 3 Refining the Queue ADT

The above descriptions are informal. We now provide a UML description of the queue abstraction, continuing to use our names for the enqueue and dequeue operations.

```
+empty():boolean  {query}
// returns true if the queue is empty, false if not

+enqueue([in] new_item:QueueItemType): boolean throw queue_exception
// appends new_item to the rear of the queue, returning true if successful
// if it fails, it throws a queue_exception and also returns false

+dequeue(): boolean  throw queue_exception
// removes the front element from the queue, returning true if successful
// if it fails, it throws an exception and returns false

+dequeue([out] front_item:QueueItemType) throw queue_exception
// same as dequeue() above but stores the item removed into front_item,
// throwing an exception if it fails

+front():QueueItemType {query} throw queue_exception
// returns  the item at the front of the queue,
// throwing an exception if it fails.
```

As with stacks, these descriptions are an abstraction; a specific interface may choose, for example, to define `front()` so that it passes the item from the front as a parameter rather than returning it. In this way it could return a boolean to indicate success or failure.

## 4 The C++ Queue Template Class Interface

C++ provides a `queue` class template, which can be accessed by including the `<queue>` header file in the application. It makes the following methods available:

```
bool empty() const;
size_type size() const;
value_type& front();
value_type& back();
void push(const value_type& _X);
void pop();
```

**Notes.**

- As noted above, there is no enqueue or dequeue operation; these are named `push` and `pop` respectively. Do not confuse them with the stack's operations. They are functionally the same as enqueue and dequeue. C++ implements queues and stacks as classes that encapsulate a more basic container class template and which restrict access to the embedded container. Classes whose implementations are built on broader, more general classes but which provide restricted methods are called ***adaptor containers***.
- Notice that this interface includes a `back()` method, which accesses the item in the rear of the queue. This is not consistent with the definition of a queue as an abstract data type. You will often encounter queue implementations that provide methods other than those that define the queue as a data abstraction. For example, you maye also find operations that can remove elements from the interior of a queue, which is a gross violation of what makes it a queue.
- Notice too that the `front()` and `back()` methods return a reference to the item at the respective locations. This implies that the caller can modify these elements within the queue. The interface de defined above returns by value, not by reference.

## 5 Queue Applications

Queues have many applications in computer science, partly because they act as first-in-first-out ***buffers***. A buffer can be thought of as a storage area for data that is in a state of transit. There are many circumstances in which one process generates data for another process, but the two processes run independently and at possibly different rates of speed. The process that generates the data puts it into a buffer, and the one that uses that data removes it from the buffer in the order in which it was placed. For example, when you burn a music CD with data from a hard drive, one process delivers that data to the CD burner, and the CD burner burns it to the CD. If the process reading from the hard drive and sending the data sends it too quickly and the CD burner is busy burning data, the new data would be lost if there were no buffer for it. On the other hand, if the data is delivered too slowly and there is no buffered data, the CD burner may reach the next track with no data to burn and will create gaps inadvertently. The buffer allows the two processes to have intermittent pauses or bursts without resulting in missing or duplicated final data. If the buffer is not large enough, there can still be overruns or underruns that could lead to failure in the burning process, depending on how the two processes are designed.

Almost all services within the machine use a queue for their waiting tasks. The queue acts as a buffer of jobs that have yet to be completed. In particular,

- the printer spooler stores print jobs in a queue;
- the various network servers maintain queues of pending service requests; and

- the operating system maintains a complex of queues for a variety of services, such as access to the CPU, allocation of memory or disk space, and even the processing of keyboard characters delivered to the machine from a keyboard device.

Queues are also used in applications outside of computer science, especially in simulations of all kinds. One can use a queue to model air traffic at an airport, vehicular traffic at a toll plaza, or customers waiting for available cashiers at a retail store, for example.

## 5.1 Example Application

An application lets a user enter three commands on the keyboard interactively: `e` to enter data items, `p` to print the data items entered so far, 4 per line, and `q`, to quit. The data items are positive integers. There is no limit to the number of items they can enter at a time; they stop entering by typing `-1`. When the print command is issued, all items entered but not yet displayed are printed. Because this is a buffering problem, a queue can be used.

The program will use the C++ `queue` class, instantiating it to hold items of type `data`, which is `int` in this case.

```
int main()
{
    typedef  int  data;
    queue<data>   buffer;
    data          item;
    int           count;
    char       c;
    bool          done = false;
    while ( !done ) {
        cout << “Enter command:”;
        cin  >> c;
        switch ( c ) {
        case ‘e’:
            cout << "Enter positive numbers;
                 << terminate with -1:\n";
            while ( cin >> item ) {
                if ( item != -1 )
                    buffer.push(item);
                else
                    break;
            }
            break;
        case ‘p’:
            count = 4;
            while (!buffer.empty() ) {
                if (count == 4) {
                    count = 0;
                    cout << endl;
                }
                cout << buffer.front() << “ “;
                buffer.pop();
                count++;
            }
            cout << endl;
            break;
            case ‘q’:
                done = true;
                break;
            }
        }
    }
```

## 6 Queue Implementations

This time we relied on an existing implementation of a queue to solve a problem, namely the one from the C++ library. Once again, the power of data abstraction is demonstrated. But you should know how to implement a queue, because it is an important data structure. To start, we define the interface we will implement and a class of queue exceptions that can be thrown as needed. If you are not familiar with exceptions, refer to the notes on exception handling.

The following exception class will be used by all queue implementations.

```
#include <stdexcept>
#include <string>
using namespace std;
class queue_exception: public logic_error
{
public:
    queue_exception(const string & message="")
         : logic_error(message.c_str())    {}
};
```

The `logic_error` exception is defined in `<stdexcept>`. Its constructor takes a string, which can be printed using the `what()` method of the class. The next logical step would be to derive separate exception types such as `queue_overflow`, or `queue_underflow`, as illustrated below.

```
class queue_overflow: public queue_exception
{
public:
    queue_overflow(const string & message=""): queue_exception(message) {}
};

class queue_underflow: public queue_exception
{
public:
    queue_underflow(const string & message=""): queue_exception(message) {}
};
```

The following is the interface we will implement:

```
typedef data_item QueueItemType;
class Queue
{
public:
    // constructors and destructor:
    Queue();               // default constructor
    Queue(const Queue &);  // copy constructor
    ~Queue();              // destructor

    // queue operations:
    bool empty() const;
    // Determines whether a queue is empty.
    // Precondition: None.
    // Postcondition: Returns true if the queue is empty;
    // otherwise returns false.

    int size() const;
    // returns the size of the queue.
    // Precondition: None.
    // Postcondition: None

    void enqueue(const QueueItemType& new_item) throw(queue_exception);
    // Adds an item to the rear of a queue.
    // Precondition: new_item is the item to be added.
    // Postcondition: If the insertion is successful, new_item
    // is at the rear of the queue.
    // Exception: Throws queue_exception if the item cannot
    // be placed in the queue.

    void dequeue() throw(queue_exception);
    // Removes the front of a queue.
    // Precondition: None.
    // Postcondition: If the queue is not empty, the item
    // that was enqueued least recently is removed. However, if
    // the queue is empty, deletion is impossible.
    // Exception: Throws queue_exception if the queue is empty.

    void dequeue(QueueItemType& front_item) throw(queue_exception);
    // Retrieves and removes the front of a queue.
    // Precondition: None.
    // Postcondition: If the queue is not empty, front_item
    // contains the item that was enqueued least recently and the
    // item is removed. However, if the queue is empty,
    // deletion is impossible and front_item is unchanged.
    // Exception: Throws queue_exception if the queue is empty.

    QueueItemType front() const  throw(queue_exception);
    // Retrieves the front of a queue.
    // Precondition: None.
    // Postcondition:
    // Returns: If the queue is not empty, front_item
    //          otherwise, the return value is defined and a
    //          queue_exception is thrown.

private:
   ...
};
```

**Notes.**

- There is very little difference between a stack and a queue with respect to the potential implementations, and the advantages and disadvantages are the same.
- Since a queue is just a special type of list, a queue can be implemented with a list. In this case, you can make a list a private member of the class and implement the queue operations by calling the list methods on the private list.
- It is more efficient to implement a queue directly, avoiding the unnecessary function calls. Direct methods include using an array, or using a linked representation, such as a singly-linked list.
- A queue can be implemented as an array, but not in the manner you might first envision. The trick is to implement a circular array and put the queue within it. This will make the `enqueue()` and `dequeue()` operations efficient, but still the array size might be exceeded. This can be handled by a resizing operation when this happens.
- A queue implemented as a linked list overcomes the problem of using a fixed size data structure. It is also relatively fast, since the `enqueue()` and `dequeue()` operations do not require many pointer manipulations. The storage is greater because the links take up four bytes each for every queue item.

## 6.1 Linked Implementation

We begin with the linked implementation. LIke that of the stack, a linked list implementation of a queue does not pre-allocate storage; it allocates its nodes as needed. Each node will have a data item and a pointer to the next node. Because the linked list implementation hides the fact that it is using linked nodes, the node structure is declared within the private part of the class. The private data members of the `Queue` class include a pointer to the node at the front of the queue and a pointer to the node at the rear of the queue. If the queue is empty, the front pointer and rear pointer are both `NULL`. The private part of the `Queue` class would therefore be:

```
private:
   struct QueueNode
   {
      QueueItemType item;
      QueueNode    *next;
   };
   QueueNode *front_p;     // pointer to front of queue
   QueueNode *rear_p;      // pointer to rear of queue
   int       num_items;    // to keep track of the size
};
```

**Remarks.**

- The constructor sets `front_p` and `rear_p` to `NULL` and `num_items` to 0; the test for emptiness can check whether either is `NULL` or num_items is zero.
- The copy constructor is similar to that of the stack; it has to traverse the linked list of the queue passed to it. It copies each node from the passed queue to the queue being constructed. It allocates a node, fills it, and attaches it to the preceding node. We could implement the copy constructor by repeated calls to `enqueue()`, but it is faster to implement it directly.
- Because the queue allocates its storage dynamically, it must have a user-defined, deep destructor. The destructor repeatedly calls `dequeue()` to empty the list.

- The `enqueue()` operation inserts a node at the rear of the queue. There is a special case when the list is empty because the `front_p` pointer has to be set in this case.
- The two `dequeue()` operations remove the node at the front of the queue, throwing an exception if the queue is empty. One provides this item in the parameter. For simplicity, the latter is implemented by calling the former.
- `front()` simply returns the item at the front of the queue, throwing an exception if the queue is empty.
- In theory it is possible that the operating system will fail to provide additional memory to the object when it calls `new()` in both the copy constructor and the `enqueue()` method. This implementation shows how to handle this within a `try` block.

```
// constructor
Queue::Queue() : front_p(NULL), rear_p(NULL), num_items(0) { }

// copy constructor
Queue::Queue(const Queue& aQueue)
{
    if (aQueue.num_items == 0) {
        front_p = NULL;
        rear_p  = NULL;
        num_items = 0;
    }
    else {
        // set num_items
        num_items = aQueue.num_items;
        // copy first node
        front_p       = new QueueNode;
        front_p->item = aQueue.front_p->item;
        rear_p        = front_p;

        // copy rest of queue
        QueueNode *newPtr = front_p;   // new list pointer
        QueueNode *origPtr = aQueue.front_p; // start at front of list

        while ( origPtr != aQueue.rear_p ) {
            origPtr      = origPtr->next;  // move to next node
            newPtr->next = new QueueNode;  // create new node in new queue
            newPtr       = newPtr->next;   // advance in new queue
            newPtr->item = origPtr->item;  // fill with item
        }
        rear_p       = newPtr;  // set rear to last node inserted
        rear_p->next = NULL;    // set rear node's next to NULL
   }
}

// destructor, which empties the list by calling dequeue() repeatedly
Queue::~Queue()
{
    // dequeue until queue is empty
    while ( num_items > 0 )
        dequeue();
}
```

```
bool Queue::empty() const
{
    return front_p == NULL;
}

bool Queue::size() const
{
    return num_items;
}

void Queue::enqueue(const QueueItemType& newItem) throw(queue_exception)
{
    try {
        // create a new node
        QueueNode *newPtr = new QueueNode;

        // set data portion  of new node
        newPtr->item = newItem;
        newPtr->next = NULL;

        if ( 0 == num_items )
            front_p = newPtr;
        else
            // insert the new node at the rear of the queue
           rear_p->next = newPtr;

        // in either case set rear to point to new node and increment num_items
        rear_p = newPtr;
        num_items++;
    }
    catch (bad_alloc error)
    {
        throw queue_exception(“queue_exception: cannot alloc mem”);
    }
}

void Queue::dequeue() throw(queue_exception)
{
    if ( 0 == num_items )
        throw queue_exception("queue_exception: empty queue");
    else {
        // queue is not empty;
        QueueNode *temp = front_p;
        if ( front_p == rear_p ) {
            front_p = NULL;
            rear_p  = NULL;
        }
        else
            front_p = front_p->next;
        temp->next = NULL;
        delete temp;
        num_items--;
    }
}
```

```
void Queue::dequeue(QueueItemType& front_item) throw(queue_exception)
{
    if ( 0 == num_items )
        throw queue_exception("queue_exception: empty queue");

    else {
        // queue is not empty; retrieve front
        front_item = front_p->item;
        dequeue();
   }
}

void Queue::front(QueueItemType& stackTop) const throw(queue_exception)
{
   if ( 0 == num_items )
      throw queue_exception("queue_exception: empty queue");
   else       // queue is not empty; retrieve front
      front_item = front_p->item;
}
```

## 6.2 Array Implementation

The first implementation that comes to mind, which we will call the ***naive implementation***, is very inefficient. In this implementation the idea would be to set the front of the queue to be index 0 and the rear to the highest index in use. Then enqueuing an item would increment `rear` and insert the item into that position in the array. But what about dequeuing an item?

We could increment `front` to dequeue an item, setting it to 1 instead of 0, then to 2 the second time, and so on. As we continue to dequeue the queue, the front will increase in value, which is called ***rightward drift***. Since enqueuing does not alter the value of `front`, it will steadily increase, eventually exceeding the maximum sixe of the array. This is therefore not a solution. In order for it to stay at index 0, we would have to shift all array items downward with each `dequeue()` operation, and change the value of the `rear` index as well. This means that the computational cost of a `dequeue()` operation is proportional to the number of items in the queue. This is not an acceptable solution.

You should see that an array implementation of a queue is not as simple as that of a stack. Do not be discouraged, however, because one small change will convert an array into a viable implementation of a queue. We will turn the array into a circular array. In a circular array, we think of the last array item as preceding the first item. To illustrate, suppose that the array is of size 8:

Imagine that we bend the array so that it forms a circle, with 0 followed by 7 in this case. In other words, after we reach the item with index 7, the next item after that will be the item with index 0. This is accomplished by using modulo arithmetic when indexing through the array; if `current_index` is an index that sequences through the array, then we would use the following to compute its next index:

```
current_index = (current_index +1) % array_size
```

or equivalently,

```
current_index = (current_index == array_size-1)? 0: current_index+1;
```

Let us assume that the array is named `items` and is of size `MAX_QUEUE`. Now imagine that we maintain two variables called `front_idx` and `rear`_idx, and that `rear_idx` is the index of the last item in the queue, and

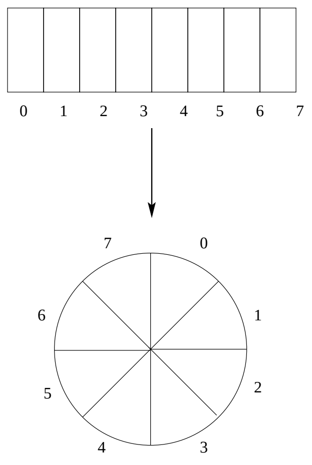


Figure 2: Circular array of size 8

`front_idx` is the index of the first item in the queue. Henceforth let us just say that `front_idx` "points to" the first item and `rear_idx` "points to" the last item. Then to enqueue an item, we would use the instructions

```
rear_idx = (rear_idx + 1) % MAX_QUEUE;
items[rear_idx] = item_to_insert;
```

If `rear_idx` had the value `MAX_QUEUE-1` beforehand, then it would have the value `0` after, since `MAX_QUEUE % MAX_QUEUE = 0`. This is how it behaves circularly. Similarly, to dequeue an item (from the front of the queue) and return it to the caller, we would use

```
front_item = items[front_idx];
front_idx = (front_idx + 1) % MAX_QUEUE;
```

Notice that neither `front_idx` nor `rear_idx` is ever decreased; they just march forward endlessly, but because their world is now round, they do not fall off of it.

The last problem is to decide on the initial values of the front and rear indices. We can arbitrarily set `front_idx` to the first index in the array, 0. It does not matter. What matters is where `rear_idx` is in relation to it. If the invariant assertion about `front_idx` and `rear_idx` are that `front_idx` points to the first item and `rear_idx` points to the last, when `front_idx == rear_idx`, this should mean that the queue has a single item in it. This in turn implies that when the queue is empty, `rear_idx` is the index before `front_idx`, or stated mathematically, `front_idx == (rear_idx + 1) % MAX_QUEUE`. Therefore, we set the initial value of `rear_idx` to `MAX_QUEUE-1`.

We will maintain the size of the queue in the `num_items` member variable as we did with the linked implementation, so the tests for emptiness remains the same. We will also need a test for fullness, because it is possible that the array reaches capacity. The test for fullness will be that `num_items == MAX_QUEUE`.

It is worth pointing out that when the queue is full, every array element contains an item. This implies that `rear_idx` is the index before `front_idx`, or that `front_idx == (rear_idx + 1) % MAX_QUEUE`. But this is exactly the same condition as occurs when the queue is empty. This is why it is especially important that we use the `num_items` variable to test whether the queue is empty or full.

With the preceding discussion in mind, the array-based queue implementation is given below.

```
class Queue
{
public:
    // same interface as above plus
    bool full() const;

private:
    QueueItemType  items[MAX_QUEUE];   // array of MAX_QUEUE many items
    int            front_idx;          // index of front of queue
    int            rear_idx;           // index of rear of queue
    int            num_items;          // number of items in the queue
};
```

**Remarks.** The implementation is very straightforward:

- Because the array is a statically allocated structure, the copy constructor and destructor are simple and omitted here.
- The test for emptiness when the counter variable `num_items` is present reduces to `num_items == 0`.
- The remaining operations are all as described above.

The implementation:

```
Queue::Queue(): front_idx(0), rear_idx(MAX_QUEUE-1), num_items(0) { }
 // end default constructor

bool Queue::empty() const
{
    return num_items == 0);
}

bool Queue::full() const
{
    return num_items == MAX_QUEUE);
}

void Queue::enqueue(QueueItemType new_item) throw(queue_exception)
{
   if (num_items == MAX_QUEUE)
       throw queue_exception("queue_exception: queue full on enqueue");
   else {
       // queue is not full; insert item
       rear_idx = (rear_idx+1) % MAX_QUEUE;
       items[rear_idx] = new_item;
       ++num_items;
   }
}

void Queue::dequeue() throw(queue_exception)
{
    if ( 0 == num_items )
        throw queue_exception("queue_exception: empty queue, cannot dequeue");
    else {
            // queue is not empty; remove front
            front_idx = (front_idx+1) % MAX_QUEUE;
            --num_items;
        }
    }

    void Queue::dequeue(QueueItemType& queueFront) throw(queue_exception)
    {
        if ( 0 == num_items )
            throw queue_exception("queue_exception: empty queue, cannot dequeue");
        else {
            // queue is not empty; retrieve and remove front
            queueFront = items[front_idx];
            front_idx  = (front_idx+1) % MAX_QUEUE;
            --num_items;
        }
    }

    void Queue::front(QueueItemType& queueFront) const throw(queue_exception)
    {
        if ( 0 == num_items )
            throw queue_exception("queue_exception: empty queue, cannot get front");
        else
            // queue is not empty; retrieve front
            queueFront = items[front_idx];
    }
```

# Classes Revisited: Templates and Inheritance

## 1 Introduction

The purpose of these notes is to introduce basic concepts of templates and inheritance, not to explain either topic in an exhaustive way. Templates are a complex subject and inheritance is even more complex. We begin with concepts about the different types of constructors and destructors, then move on to templates, and then move into an introduction to inheritance.

## 2 Constructors, Copy Constructors, and Destructors

This is an overview of the basics of constructors and destructors and things related to them. First, a refresher about constructors:

- Constructors have no return values and have the same name as the class. They cannot contain a return statement.
- Constructors can have arguments.
- A **default constructor** is a constructor that can be called with no arguments. If a class does not have a user-supplied default constructor, the compiler generates one at compile time. We call this a **compiler-generated constructor**.
- If a class must initialize its data members, then the class needs a user-defined constructor because the compiler-generated constructor will not be able to do this.
- A class should always have a default constructor, i.e., one that requires no arguments.

### 2.1 Copy Constructors

A **copy constructor** is a kind of constructor for a class. It is invoked in the following situations:

1. When an object needs to be constructed because of a declaration with initialization from another object of a type that can be converted to the object's class, as in

```
ScreenData C;
ScreenData B = C; // copy constructor called here
ScreenData B(C);  // copy constructor called here
```

2. When an object is passed by value to a function. For example, if the function `redraw_screen` has the signature

```
void redraw_screen( ScreenData S );
```

and it is used in the following code snippet:

```
ScreenData new_screen;
// code to fill in new_screen
redraw_screen(new_screen);
```

then `new_screen` will be copied into `S`.

3. When an object is returned by value from a function. If the function `get_old_screen()` has the signature

   ```
   ScreenData get_old_screen( ScreenData S);
   ```

   then with the call

   ```
   old_screen = get_old_screen(C);
   ```

   the `ScreenData` object returned by `get_old_screen()` will be copied into `old_screen` by the copy constructor.

4. *Note that it is not invoked here*:

   ```
   ScreenData screen2 = ScreenData(params);
   ```

   The ordinary constructor is used to create the object `screen2` directly. The right-hand side is not a call to a constructor. The compiler arranges for screen2 to be initialized from a constructor that is given the parameters `params`.

To illustrate, suppose that `Date` is the following class:

```
class Date
{
public:
    Date       ( int y = 1970, int m = 1, int d = 1 );
    Date       ( const Date &date );
    string get ( );
private:
    short year;
    short month;
    short day;
};
```

It has three private members that store the date as a day, month, and year, and three member functions. The first constructor serves as both a default constructor and a constructor that can be supplied values for the private members. The second is a copy constructor. A copy constructor has a single argument which is a reference to an object of the class. We would provide a definition of this copy constructor such as

```
Date::Date  ( const Date &date )
{
    year = date.year;
    month = date.month;
    day = date.day;
}
```

Because the parameter is not being changed, it is always safer to make it a `const` reference parameter.

## 2.2 Copy Assignment Operators

The copy constructor is called only when an object is being created for the first time, not when it already exists and is being assigned a new value. In this case the **copy assignment operator** is invoked. The *operator=* is the *copy assignment* operator. It is invoked when one object is assigned to an existing object.

Unlike constructors, the copy assignment operator must return a value, and not just any value: it must return the object on which it is invoked. For example, if we were to add a copy assignment operator to the `Date` class, it would be defined as follows:

```
Date Date::operator=  ( const Date &date )
{
    this->year = date.year;
    this->month = date.month;
    this->day = date.day;
    return *this;
}
```

This function returns a `Date` object by value, not reference. It first copies the values out of the argument passed to it, into the object on which it is called. It then returns this object. The value of any assignment operation is always the value assigned to the left-hand side of the assignment; returning the object is required to maintain this semantics.

## 2.3 Destructors

A **destructor** is called when an object goes out of scope or is deleted explicitly by a call to `delete()`. The compiler supplies destructors if the user does not. The reason to supply a destructor is to remove memory allocated by a call to `new()`, to close open files, and so on. If your object does not dynamically allocate memory, directly or indirectly, you do not have to write a destructor for the class; it is sufficient to let the compiler create one for you.

The form of a destructor definition is similar to that of a constructor, except that

- it has a leading tilde ('~'), and
- it has no parameters.

As with constructors, it has no return type.

Some, but not all, of the situations in which a destructor is called are

- when the program terminates, for objects with static storage duration (e.g. global variables or static variables of functions),
- when the block in which an object is created exits, for objects with automatic storage duration (i.e., for local variables of a function when the function terminates),
- when `delete()` is called on an object created with `new()`.

The following listing is a program that writes messages to standard output showing when the constructors and destructors are called. Running it is a good way to explore what the lifetimes of various types of objects are in C++. The class and main program are in a single file here, for simplicity.

```
#include <string>
#include <iostream>
using namespace std;

class MyClass
{
public:
   MyClass( int id=0, string mssge = "" );   //constructor
   ~MyClass();                               // destructor

private:
   int     m_id;        // stores object's unique id
   string m_comment;   // stores comment about object
};  // end of MyClass definition

// Constructor and destructor definitions
MyClass::MyClass( int id, string comment )
{
    m_id = id;
    m_comment = comment;

    cout << "     MyClass CONSTRUCTOR: id = " << m_id
         << ": " << m_comment << endl;
}

MyClass::~MyClass()
{
    cout << "     MyClass DESTRUCTOR:  id = " << m_id
         << ": " << m_comment << endl;
}

/****************************** main program file ***********************/

MyClass Object1(1, "global, static variable");

void foo();
void bar();

int main()
{
    cout << "main() started.\n";

    MyClass Object2(2, "local automatic in main()");
    static MyClass Object3( 3, "local static in main()");

    cout << "calling foo() ...\n";
    foo();
    cout << "foo() returned control to main() ...\n";
    cout << "main() is about to execute its return statement\n";
    return 0;
}

void foo()
{
    cout << "foo() started.\n";

    MyClass        Object5(5, "local automatic in foo()");
    static MyClass Object6(6, "local static in foo()");

    cout << "foo() is calling bar() ...\n";
    bar();
    cout << "bar() returned control to foo().\n";
    cout << "foo() ended.\n";
}

void bar()
{
    cout << "bar() started.\n";
    MyClass        Object7(7, "local automatic in bar()");
    static MyClass Object8(8, "local static in bar()");
    cout << "bar() ended.\n";
}
```

## 3 Templates

One difference between C and C++ is that C++ allows you to define templates for both classes and functions. It is easier to understand class templates if you first understand function templates, and so we start with these.

Suppose that in the course of writing many, many programs, you find that you need a swap function here and there. Sometimes you want a swap function that can swap two integers, sometimes you want one that can swap two doubles, and sometimes you need to swap objects of a class. In C, you have to write a swap function for each type of object, or you can reduce the work by writing something like this[1]:

```
    typedef int elementType;

    void swap ( elementType *x, elementType *y)
    {
        elementType temp = *x;
        *x = *y;
        *y = temp;
    }
```

and you would call this with a call such as

```
    int a, b;
    a = ... ; b = ... ;
    swap(&a, &b);
```

In C, the parameters need to be pointers to the variables to be swapped, and their address must be passed. If you wanted to swap doubles, you would change the `typedef` by replacing the word "`int`" by "`double`."

In C++, you could do the same thing using reference parameters:

[1]There are other methods as well, but these are the two principal approaches.

```
typedef int elementType;

void swap ( elementType &x, elementType &y)
{
    elementType temp = x;
    x = y;
    y = temp;
}
```

and you could call this with code such as

```
int a, b;
a = ... ; b = ... ;
swap(a, b);
```

Although you do not have to write a separate swap function for each different element type, it is inconvenient. The C++ language introduced function templates as a way to avoid this.

## 3.1 Function Templates

A **function template** is a template for a function. It is not an actual function, but a template from which the compiler can create a function if and when it sees the need to create one. This will be clarifed shortly. A template for the swap function would look like this:

```
template <class elementType>
void swap ( elementType &x, elementType &y)
{
    elementType temp = x;
    x = y;
    y = temp;
}
```

The word "class" in the template syntax has nothing to do with classes in the usual sense. It is just a synonym for the word "type." All types in C++ are classes. The syntax of a (single-parameter) function template definition is

```
template <class type_parameter> function-definition
```

where function-definition is replaced by the body of the function, as `swap()` above demonstrates. The syntax for a (single-parameter) function template declaration (i.e., prototype) is

```
template <class type_parameter > function-declaration
```

You need to repeat the line

```
template <class type_parameter>
```

before both the declaration and the definition. For example:

```
// Declare the function template prototype
template <class T>
void swap( T & x, T & y );

int main()
{
     int n= 5;
     int m= 8;
     char ch1 = 'a', ch2 = 'b';
     // more stuff here
     swap(n,m);
     swap(ch1, ch2);
     // ...
}

// Define the function template declared above
template <class T>
void swap( T & x, T & y )
{
    T temp = x;
    x = y;
    y = temp;
}
```

You will often see just the letter "T" used as the type parameter in the template.

When the compiler compiles the main program above, it sees the first call to a function named `swap`. It is at that point that it has to create an instance of a function from the template. It infers from the types of its parameters that the type of the template's parameter is `int`, and it creates a function from the template, replacing the type parameter by `int`. When it sees the next call, it creates a second instance whose type is `char`.

Because function templates are not functions, but just templates from which the compiler can create functions, there is a bit of a problem with projects that are in multiple files. If you want to put the function prototype in a header file and the function definition in a separate `.cpp` file, the compiler will not be able to compile code for it in the usual way if you use that function in a program. To demonstrate, suppose that we create a header file with our swap function prototype, an implementation file with the definition, and a main program that calls the function.

This is `swap.h`:

```
#ifndef SWAP_H
#define SWAP_H

template <class T>
void swap( T &x, T &y);
#endif
```

and `swap.cpp`:

```
template <class T>
void swap( T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

and main.cpp:

```
#include "swap.h"
int main ()
{
    int a = 10, b = 5;
    swap(a,b);
    return 0;
}
```

When we run the command

```
g++ -o demo swap.cpp main.cpp
```

we will see the error message

```
/tmp/ccriQBJX.o: In function ‘main’:
main.cpp:(.text+0x29): undefined reference to ‘void swap<int>(int&, int&)’
collect2: ld returned 1 exit status
```

This is because the function named `swap` does not really exist when `main` is compiled. It has a reference only to a function template. The solution is to put the function template implementation into the header file, as unsatisfying as that is because it breaks the wall that separates interface and implementation. This can be accomplished with an `#include` directive:

```
#ifndef SWAP_H
#define SWAP_H

template <class T>
void swap( T &x, T &y);

#include “swap.cpp”
#endif
```

The general rule then, is to put the function template prototypes into the header file, and at the bottom, include the implementation files using an `#include` directive. There will be no problem with multiply-defined symbols in this case when you compile the code.

*Note.* Function templates, and templates in general, can have multiple parameters, and they do not have to be classes, but that is a topic beyond the scope of this introduction. You may also see the word `typename` used in place of the word `class` as the type of the template parameter. For the most part, these are interchangeable, but it is better to use `class` until you know the subtle difference. The interested reader can refer to a good C++ book for the details.

## 3.2 Class Templates

Imagine that you want to implement a list class, such as the one we described in the introduction to this course. If you go back and look at the list ADT, you will find nothing in it that is specific to any particular type of data object, other than the ability to copy objects. For the sorted list ADT, the objects did have to be comparable to each other in some linear ordering, but that was about it, in terms of specific properties. It stands to reason that you should be able to create a generic sort of list, one whose definition does not depend on the underlying element type. This is one reason that C++ allows you to create templates for classes as well. A class template is like a generic description of that class that can be instantiated with different underlying data types.

### Defining Class Templates

As with function templates, a C++ class template is not a class, but a *template for a class.* An example of a simple class template interface is

```
template <class T>
class Container
{
public:
    Container();
    Container( T initial_data);
    void set( T new_data);
    T get() const;
private:
    T mydata;
};
```

Notice that a class template begins with the `template` keyword and template parameter list, after which the class definition looks the same as an ordinary class definition. The only difference is that it uses the type parameter `T` from the template's parameter list. The syntax for the implementations of the class template member functions when they are outside of the interface is a bit more complex. The above functions would have to be defined as follows:

```
template <class T>
void Container<T>::set ( T new_data )
{
    mydata = new_data;
}

template <class T>
T Container<T>::get() const
{
    return mydata);
}
```

Notice the following:

1. Each member function is actually a function template definition.
2. All references to the class are to `Container<T>` and not just Container. Thus, the name of each member function must be preceded by `Container<T>::`.

In general the syntax for creating a class template is

```
template <class T>  class class_name {  class_definition  };
```

and a member function named `foo` would have a definition of the form

```
template <class T>
return_type class_name<T>::foo ( parameter_list ) { function definition }
```

**Declaring Objects**

To declare an object of a class that has been defined by a template requires , in the simplest case, using a syntax of the form

```
class_name<typename> object_name;
```

as in

```
Container<int>    int_container;
Container<double> double_container;
```

If the `Container` class template had a constructor with a single parameters, the declarations would instead be something like

```
Container<int>    int_container(1);
Container<double> double_container(1.0);
```

The following is a complete listing of a very simple program that uses a class template.

Listing 1: A program using a simple class template.

```
#include <iostream>
using namespace std;

template <class T>
class MyClass
{
  public:
      MyClass(  T initial_value);
      void set( T x) ;
      T get( )    ;

  private:
      T val;
};

template < class T >
MyClass < T >:: MyClass (T initial_value)
{
    val = initial_value;
}

template < class T >
void MyClass < T >:: set (T x)
{
    val = x;
}

template < class T >
T MyClass < T >::get ()
{
    return val;
}
```

```
int main ()
{
    MyClass<int>     intobj(0);
    MyClass<double>  floatobj(1.2);

    cout << "intobj value = " << intobj.get()
         << " and floatobj value = "  << floatobj.get() << endl;
    intobj.set(1000);
    floatobj.set (0.12345);
    cout << "intobj value = " << intobj.get()
         << " and floatobj value = "  << floatobj.get() << endl;

    return 0;
}
```

Again, remember that *a class template is not an actual definition of a class, but of a template for a class.* Therefore, if you put the implementation of the class member functions in a separate implementation file, which you should, then you must put an `#include` directive at the bottom of the header file of the class template, including that implementation file. In addition, make sure that you do not add the implementation file to the project or compile it together with the main program. For example, if `myclass.h`, `myclass.cpp`, and `main.cpp` comprise the program code, with `myclass.h` being of the form

```
#ifndef MYCLASS_H
#define MYCLASS_H

// stuff here

#include “myclass.cpp”
#endif // MYCLASS_H
```

and if `main.cpp` includes `myclass.h`, then the command to compile the program must be

```
g++ -o myprogram main.cpp
```

*not*

```
g++ -o myprogram myclass.cpp main.cpp
```

because the latter will cause errors like

```
myclass.cpp:4:6: error: redefinition of ‘void MyClass<T>::set(T)’
myclass.cc :4:6: error: ‘void MyClass<T>::set(T)’ previously declared here
```

This is because the compiler will compile the `.cpp` file twice! This is not a problem with function templates, but it is with classes, because classes are turned into objects.

## 4 Inheritance

Inheritance is a feature that is present in many object-oriented languages such as C++, Eiffel, Java, Ruby, and Smalltalk, but each language implements it in its own way. This chapter explains the key concepts of the C++ implementation of inheritance.

## 4.1 Deriving Classes

Inheritance is a feature of an object-oriented language that allows classes or objects to be defined as extensions or specializations of other classes or objects. In C++, classes inherit from other classes. Inheritance is useful when a project contains many similar, but not identical, types of objects. In this case, the task of the programmer/software engineer is to find commonality in this set of similar objects, and create a class that can serve as an archetype for this set of classes.

**Examples**

- Squares, triangles, circles, and hexagons are all 2D shapes; hence a `Shape` class could be an archetype.
- Faculty, administrators, office assistants, and technical support staff are all employees, so an `Employee` class could be an archetype.
- Cars, trucks, motorcycles, and buses are all vehicles, so a `Vehicle` class could be an archetype.

When this type of relationship exists among classes, it is more efficient to create a class hierarchy rather than replicating member functions and properties in each of the classes. Inheritance provides the mechanism for achieving this.

**Syntax**

The syntax for creating a derived class is very simple. (You will wish everything else about it were so simple though.)

```
class A
{/* ... stuff here ... */};

class B: [access-specifier] A
{/* ... stuff here ... */};
```

in which an *access-specifier* can be one of the words, `public`, `protected`, or `private`, and the square brackets indicate that it is optional. If omitted, the inheritance is `private`.

In this example, `A` is called the **base class** and `B` is the **derived class**. Sometimes, the base class is called the **superclass** and the derived class is called a **subclass**.

**Five Important Points (regardless of access specifier):**

1. The constructors and destructors of a base class are not inherited.
2. The assignment operator is not inherited.
3. The friend functions and friend classes of the base class are not inherited.
4. The derived class does not have access to the base class's private members.
5. The derived class has access to all public and protected members of the base class.

Public inheritance expresses an **is-a** relationship: a B *is a* particular type of an A, as a car *is a* type of vehicle, a manager *is a* type of employee, and a square *is a* type of shape.

Protected and private inheritance serve different purposes from public inheritance. Protected inheritance makes the public and protected members of the base class protected in the derived class. Private inheritance makes the public and protected members of the base class private in the derived class. These notes do not discuss protected and private inheritance.

**Example**

In this example, a `Shape` class is defined and then many different kinds of shapes are derived from it.

```
class Shape {
private:
    Point Centroid;
public:
    void Move(Point newCentroid); // move Shape to new centroid
    /* more stuff here */
};

class Square   : public Shape {/* stuff here */ };
class Triangle : public Shape {/* stuff here */ };
class Hexagon  : public Shape {/* stuff here */ };
/* and so forth */
```

## 4.2 Implicit Conversion of Classes

The C++ language allows certain assignments to be made even though the types of the left and right sides are not identical. For example, it will allow an integer to be assigned to a floating-point type without an explicit cast. However, it will not in general let a pointer of one type be assigned to a pointer of another type. One exception to this rule has to do with assigning pointers to classes. I will begin this section by stating its conclusion. If you do not want to understand why it is true or get a deeper understanding of the nature of inheritance, you can then just skip to the next section.

**Implicit Conversion of Classes**: *The address of an object of a derived class can be assigned to a pointer declared to the base class, but the base class pointer can be used only to invoke member functions of the base class.*

Because a `Square` is a specific type of `Shape`, a `Square` is a `Shape`. But because a `Square` has other attributes, a `Shape` is not necessarily a `Square`. But consider a `Shape*` pointer. A `Shape*` is a pointer to a `Shape`. A `Square*` is a pointer to a `Square`. A `Square*` is not the same thing as a `Shape*`, since one points to `Squares` and the other points to `Shapes`, and so they have no inherent commonality other than their "pointerness."

However, since a `Square` is also a `Shape`, a `Square*` can be used wherever a `Shape*` can be used. In other words, `Squares`, `Triangles`, and `Hexagons` are all `Shapes`, so whatever kinds of operations can be applied to `Shapes` can also be applied to `Squares`, `Triangles`, and `Hexagons`. Thus, it is reasonable to be able to invoke any operation that is a member function of the `Shape` class on any dereferenced `Shape*`, whether it is a `Square`, a `Triangle`, or a `Hexagon`. This argument explains why, in C++, the address of any object derived from a `Shape` can be assigned to a `Shape` pointer; e.g., a `Square*` can be assigned to a `Shape*`.

The converse is not true; a `Shape*` cannot be used wherever a `Square*` is used because a `Shape` is not necessarily a `Square`! Dereferencing a `Square*` gives access to a specialized set of operations that only work on `Squares` and cannot be applied to arbitrary shapes. If a `Square*` contained the address of a `Shape` object, then after dereferencing the `Square*`, you would be allowed to invoke a member function of a `Square` on a `Shape` that does not know what it is like to be a `Square`, and that would make no sense. So this cannot be allowed.

The need to make the preceding argument stems from the undecidability of the **Halting Problem** and the need for the compiler designer to make sensible design decisions. If you are not familiar with the Halting Problem, you can think of it as a statement that there are problems for which no algorithms exist. One consequence of the Halting Problem is that it is not possible for the compiler to know whether or not the address stored in a pointer is always going to be the address of any specific object. To illustrate this, consider the following code fragment, and assume that the `Square` class is derived from the `Shape` class.

```
1. Square* pSquare;
2. Shape* pShape;
3. void* ptr;
4. Shape someShape;
5. Square someSquare;
6. /* .. ..... */
7. if ( some condition that depends on user input )
8.     ptr = (void*) &someShape;
9. else
10.    ptr = (void*) &someSquare;
11. pShape = (Shape*) ptr;
```

The compiler cannot tell at compile time whether the true or false branch of the if-statement will always be taken, ever be taken, or never taken. If it could, we could solve the Halting Problem. Thus, at compile-time, it cannot know whether the assignment in line 11 will put the address of a `Square` or a `Shape` into the variable `pShape`. Another way to say this is that, at compile-time, the compiler cannot know how to bind an object to a pointer. The designer of C++ had to decide what behavior to give to this assignment statement. Should it be allowed? Should it be a compile time error? If it is allowed, what operations can then be performed on this pointer?

The only sensible and safe decision is to allow the assignment to take place, but to play it "safe" and allow the dereferenced `pShape` pointer to access only the member functions and members of the `Shape` class, not any derived class, because those operations are available to both types of objects.

## 4.3 Multiple Inheritance

Suppose that the shapes are not geometric abstractions but are instead windows in an art-deco building supply store. Then what they also have in common is the property of being a window, assuming they are all the same type of window (e.g., double hung, casement, sliding, etc.). Then geometric window shapes really inherit properties from different archetypes, i.e., the property of being a shape and the property of being a window. In this case we need multiple inheritance, which C++ provides:

**Example**

```
class Shape {
private:
    Point Centroid;
public:
    void Move(Point newCentroid); // move Shape to new centroid
    /* more stuff here about being a Shape */
};
class Window
{
   /* stuff here about being a Window */
};
class SquareWin   : public Shape, public Window {/* stuff here */ };
class TriangleWin : public Shape, public Window {/* stuff here */ };
class HexagonWin  : public Shape, public Window {/* stuff here */ };
/* and so forth */
```

Note the syntax. The derived class is followed by a single colon (:) and a comma-separated list of base classes, with the inheritance qualifier (`public`) attached to each base class. The set of classes created by the above code creates the hierarchy depicted in Figure 1.

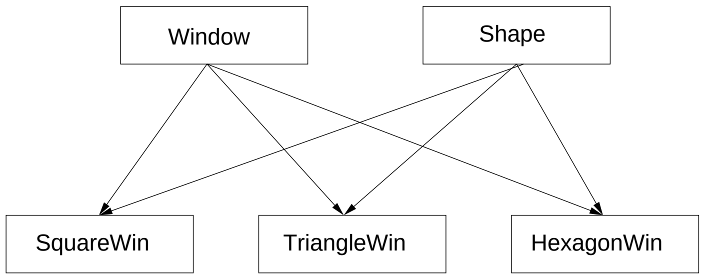


Figure 1: Multiple inheritance.

## 4.4 Extending Functionality of Derived Classes with Member Functions

Inheritance would be relatively useless if it did not allow the derived classes to make themselves different from the base class by the addition of new members, whether data or functions. The derived class can add members that are not in the base class, making it like a subcategory of the base class.

**Example**

The `Rectangle` class below will add a member function not in the base class.

```
class Shape
{
private:
    Point Centroid;
public:
    void Move(Point newCentroid); // move Shape to new centroid
    Point getCentroid() const;
    /* more stuff here */
};

class Rectangle : public Shape
{
private:
    float length, width;
public:
    /* some stuff here too */
    float LengthDiagonal() const;// functionality not in Shape class
};
```

The `Rectangle` class can add a `LengthDiagonal` function that was not in the base class since it did not makes sense for it to be in the base class and only the `Rectangle` class has all diagonals of the same length. Remember though that the member functions of the derived class cannot access the private members of the base class, so the base class must either make protected or public accessor functions for its subclasses if they need access to the private members.

## 4.5 Redefining Member Functions in Derived Classes

Derived classes can redefine member functions that are declared in the base class. This allows subclasses to specialize the functionality of the base class member functions for their own particular needs. For example, the base class might have a function `print()` to display the fields of an employee record that are common to all employees, and in a derived class, the `print()` function might display more information than the `print()` function in the base class.

*Note.* A function in a derived class overrides a function in the base class only if its signature is identical to that of the base class except for some minor differences in the return type. It must have the same parameters and same qualifiers. Thus,

```
void print();
void print() const;
```

are not the same and the compiler will treat them as different functions, whereas

```
void print() const;
void print() const;
```

are identical and one would override the other. Continuing with the `Shape` example, suppose that a `Shape` has the ability to print only its `Centroid` coordinates, but we want each derived class to print out different information. Consider the `Rectangle` class again, with a `print()` member function added to `Shape` and `Rectangle` classes.

```
class Shape
{
private:
    Point Centroid;
public:
    void Move(Point newCentroid); // move Shape to new centroid
    Point getCentroid() const;
    void print() const; // prints just Centroid coordinates
    /* more stuff here */
};

class Rectangle : public Shape
{
private:
    float length, width;
public:
    /* some stuff here too */
    float LengthDiagonal() const;// functionality not in Shape class
    void print() const
    { /* print stuff that Rectangle class has here */ }
};

/* .... */
Shape     myShape;
Rectangle myRect;
myRect.print();
myShape.print();
```

The call to `myRect.print()` will invoke the `print()` member function of the `Rectangle` class, since `myRect` is bound to a `Rectangle` at compile time. Similarly, the call to `myShape.print()` will invoke the `Shape` class's `print()` function. But what happens here:

```
Shape* pShape;
pShape = new Rectangle;
pShape->print();
```

In this case, the address of the dynamically allocated, anonymous `Rectangle` object is assigned to a `Shape` pointer, and referring back to Section 4.2 above, the dereferenced `pShape` pointer will point to the `print()` member function in the `Shape` class, since the compiler binds a pointer to the member functions of its own class. Even though it points to a `Rectangle` object, `pShape` cannot invoke the `Rectangle`'s `print()` function. This problem will be overcome below when virtual functions are introduced.

## 5 Revisiting Constructors and Destructors

### 5.1 Constructors

Let us begin by answering two questions.

***When does a class need a constructor?***

If a class must initialize its data members, then the class needs a user-defined constructor because the compiler-generated constructor will not be able to do this.

***If a class is derived from other classes, when does it need a user-defined constructor?***

To understand the answer, it is necessary to understand what happens at run time when an object is created. Class objects are always constructed from the bottom up, meaning that the lowest level base class object is constructed first, then any base class object immediately derived from that is constructed, and so on, until the constructor for the derived class itself is called. To make this concrete, suppose that four classes have been defined, and that they form the hierarchy depicted in Figure 2, in which `D` is derived from `C`, which is derived from `B`, which is derived from A.

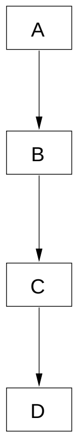


Figure 2: Class hierarchy.

Then when an object of class `D` is created, the run time system will recursively descend the hierarchy until it reaches the lowest level base class (`A`), and construct `A`, then `B`, then `C`, and finally `D`.

From this discussion, it should be clear that a constructor is required for every class from which the class is derived. If a base class's constructors require arguments, then there must be a user-supplied constructor for that class, and any class derived from it must explicitly call the constructor of the base class, supplying the arguments to it that it needs. If the base class has at least one default constructor, the derived class does not need to call it explicitly, because the default constructor can be invoked implicitly as needed. In this case, the derived class may not need a user-defined constructor, because the compiler will arrange to have the run time system call the default constructors for each class in the correct order. But in any case, when a derived class object is created, a base class constructor must always be invoked, whether or not the derived class has a user-defined constructor, and whether or not it requires arguments. The short program that follows demonstrates the example described above.

**Example**

```
class A  {
public:
    A() {cout << "A constructor called\n";}
};

class B : public A  {
public:
    B() {cout << "B constructor called\n";}
};

class C : public B  {
public:
    C() {cout << "C constructor called\n";}
};

class D : public C  {};
void main() {
    D  d;
}
```

This program will display

```
A constructor called
B constructor called
C constructor called
```

because the construction of `d` implicitly requires that `C`'s constructor be executed beforehand, which in turn requires that `B`'s constructor be executed before `C`'s, which in turn requires that `A`'s constructor be executed before `B`'s. To make this explicit, you would do so in the initializer lists:

```
class A {
public:
    A() {}
};

class B : public A  {
public:
    B(): A() {}
};

class C : public B  {
public:
    C(): B() {}
};

class D : public C  {
public:
    D(): C() {}
};
```

This would explicitly invoke the `A()`, then `B()`, then `C()`, and finally `D()` constructors.

#### Summary

The important rules regarding constructors are:

- A base class constructor is ***ALWAYS*** called if an object of the derived class is constructed, even if the derived class does not have a user-defined constructor.
- If the base class does not have a default constructor, then the derived class must have a constructor that can invoke the appropriate base class constructor with arguments.
- The constructor of the base class is invoked before the constructor of the derived class.
- If the derived class has members in addition to the base class, these are constructed after those of the base class.

### 5.2 Destructors

Destructors are slightly more complicated than constructors. The major difference arises because destructors are rarely called explicitly. They are invoked for only one of two possible reasons:

1. The object was not created dynamically and execution of the program left the scope containing the definition of the class object, in which case the destructors of all objects created in that scope are implicitly invoked, or
2. The delete operator was invoked on a class object that was created dynamically, and the destructors for that object and all of its base classes are invoked.

#### Notes

- When a derived class object must be destroyed, for either of the two reasons above, it will always cause the base class's destructor to be invoked implicitly.
- Destructors are always invoked in the reverse of the order in which the constructors were invoked when the object was constructed. In other words, the derived class destructor is invoked before the base class destructor, recursively, until the lowest level base class destructor is called.
- If a class used the `new` operator to allocate dynamic memory, then the destructor should release dynamic memory by called the `delete` operator explicitly.
- From the preceding statements, it can be concluded that the derived class releases its dynamic memory before the classes from which it was derived.

To illustrate with an example, consider the following program. The derived class has neither a constructor nor a destructor, but the base class has a default constructor and a default destructor, each of which prints a short message on the standard output stream.

```
class A
{
public:
    A() {cout << "base constructor called.\n";}
    ~A() {cout << "base destructor called.\n";}
};

class B : public A  // has no constructor or destructor
```

```
{ };

void main()
{
    B* pb;       //pb is a derived class pointer
    pb = new B;  // allocate an object of the derived class
    delete pb;   // delete the object
}
```

This program will display two lines of output:

```
base constructor called.
base destructor called.
```

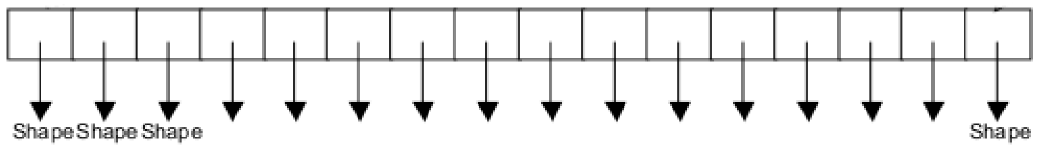
This confirms that the base class destructor and constructor are called implicitly.

## 6 Virtual Functions

I stressed above that the compiler always binds pointers to the member functions of the class to which they are declared to point to in their declarations (i.e., at compile time.) This is not a problem if derived objects are never created dynamically. In this case, inheritance is really only buying a savings in the amount of code needed in the program; it does not give the language polymorphism. Polymorphism occurs when objects can alter their behavior dynamically. This is the reason for the virtual function specifier, "virtual."

**Example**

Suppose that you have an application that draws different types of shapes on the screen. The number and type of shapes that may appear on the screen will vary over time, and you decide to create an array to store and process them. The array will therefore need to access `Shape` objects of all kinds. Since you do not know in advance which cells of the array will access which objects, the array must be able to change what it can hold dynamically, i.e., at run time. If the array element type is `Shape*`, then each array element can point to a `Shape` of a different class. However, because the pointers are of type `Shape*`, the program will only be able to access the member functions of the `Shape` class through this pointer, not the member functions of the derived classes. The following figure illustrates this idea.



What is needed is a way to allow the pointer to the base class to access the members of the derived class. This is the purpose of a **virtual function**. Declaring a function to be virtual in the base class means that if a function in a derived class overrides it and a base class pointer is dereferenced, that pointer will access the member function in the derived class. To demonstrate, consider the following program.

```
class CBase
{
```

```
public:
virtual void print()
{ cout<< "Base class print() called. " << endl; }
};

class CDerived : public CBase
{
public:
void print()
{ cout<< "Derived class print function called." << endl; }
};

void main()
{
CBase*  baseptr;
baseptr = new CDerived;
baseptr->print();
}
```

When this program is run, even though the type of `baseptr` is `CBase*`, the function invoked by the dereference "`baseptr->print()`" will be the `print()` function in the derived class, `CDerived`, because the run time environment bound `baseptr->print()` to the derived object when it was assigned a pointer of type `CDerived` and it knew that `print()` was virtual in the base class.

## 6.1 Virtual Destructors and Constructors

Constructors cannot be virtual. Each class must have its own constructor. Since the name of the constructor is the same as the name of the class, a derived class cannot override it. Furthermore, constructors are not inherited anyway, so it makes little sense.

On the other hand, destructors are rarely invoked explicitly and surprising things can happen in certain circumstances if a destructor is not virtual. Consider the following program.

```
class CBase {
public:
CBase()
{ cout << "Constructor for CBase called." << endl;}
~CBase()
{ cout << "Destructor for CBase called." << endl;}
};

class CDerived: public Cbase {
public:
CDerived()
{ cout << "Constructor for CDerived called." << endl;}
~CDerived()
{ cout << "Destructor for CDerived called." << endl;}
};

void main() {
CBase *ptr = new CDerived();
delete ptr;
}
```

When this is run, the output will be

```
Constructor for CBase called.
Constructor for CDerived called.
Destructor for CBase called.
```

The destructor for the `CDerived` class was not called because the destructor was not declared to be a virtual function, so the call "`delete ptr`" will invoke the destructor of the class of the pointer itself, i.e., the `CBase` destructor. This is exactly how non-virtual functions work. Now suppose that we make the destructor in the base class virtual. Even though we cannot actually override a destructor, we still need to use the virtual function specifier to force the pointer to be bound to the destructor of the most-derived type, in this case `CDerived`.

```
class CvirtualBase {
public:
CVirtualBase()
{ cout << "Constructor for CVirtualBase called." << endl; }
virtual ~CVirtualBase()  // THIS IS A VIRTUAL DESTRUCTOR!!!
{ cout << "Destructor for CVirtualBase called." << endl; }
};

class CDerived: public CvirtualBase {
public:
CDerived()
{ cout << "Constructor for CDerived called." << endl; }
~CDerived()
{ cout << "Destructor for CDerived called." << endl; }
};

void main() {
CVirtualBase *ptr = new CDerived();
delete ptr;
}
```

The output of this program will be:

```
Constructor for CVirtualBase called.
Constructor for CDerived called.
Destructor for CDerived called.
Destructor for CVirtualBase called.
```

This is the correct behavior.

In summary, a class `C` must have a virtual destructor if both of the following conditions are true:

- A pointer `p` of type `C*` may be used as the argument to a `delete` call, and
- It is possible that this pointer may point to an object of a derived class.

There are no other conditions that need to be met. You do not need a virtual destructor if a derived class destructor is called because it went out of scope at run time. You do not need it just because the base class has some virtual functions (which some people will tell you.)

### 6.2 Pure Virtual Functions

Suppose that we want to create an `Area()` member function in the `Shape` class. This is a reasonable function to include in this base class because every closed shape has area. Every class derived from the `Shape` class can override this member function with its own area function, designed to compute the area of that particular shape. However, the `Area()` function in the base class has no implementation because a `Shape` without any particular form cannot have a function that can compute its area. This is an example of a pure virtual function. A **pure virtual function** is one that has no possible implementation in its own class.

To declare that a virtual function is **pure**, use the following syntax:

```
virtual return-type function-name( parameter-list) = 0;
```

For example, in the `Shape` class, we can include a pure `Area()` function by writing

```
class Shape
{
private:
     Point Centroid;
public:
    void Move(Point newCentroid);
    Point getCentroid() const;
    virtual double Area() = 0; // pure Area() function
};
```

## 7 Abstract Classes

A class with at least one pure virtual function cannot have any objects that are instances of it, because at least one function has no implementation. Such a class is called an **abstract class**. In contrast, a class that can have objects is called a **concrete class**. An abstract class can serve as a class interface that can have multiple implementations, by deriving classes from it that do not add any more functionality but provide implementations of the pure virtual functions. It can also serve as an abstraction that is extended in functionality by deriving more specific classes from it.

### 7.1 Abstract Classes as Interfaces

An abstract class can act like a class interface, without divulging the "secret implementation." The following code demonstrates this idea.

```
class List // an abstract List  class
{
public:
   List();  // default constructor
   ~List(); // destructor
   virtual bool is_empty() const = 0;
   virtual int length()    const = 0;
   virtual void insert(int new_position,
           list_item_type new_item, bool& Success) = 0;
   void delete(int position, bool& Success) = 0;
   void retrieve(int position, list_item_type & DataItem,
                 bool& Success) const = 0;
};
```

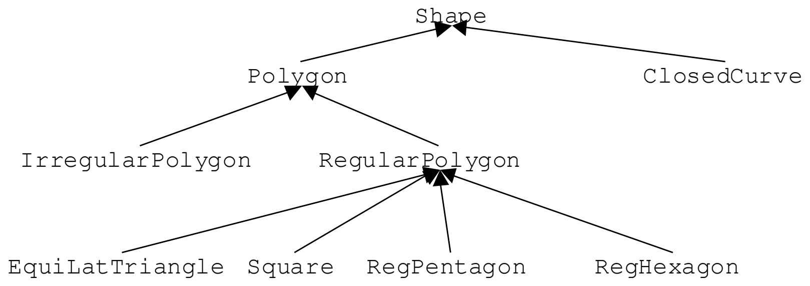


Figure 3: An abstract class hierarchy

We do not have to specify in this abstract class how the `List` is actually represented, such as whether an array stores the list, or a vector, or some linked representation. There is no private data in this class, and it has no implementation. We can derive a new class from it and let the derived class implement it. Any such derived class must conform to the abstract base class's interface, but it is free to add private members.

## 7.2 Abstract Class Hierarchies

When a class is derived from an abstract class and it does not redefine all of the pure virtual functions, it too is an abstract class, because it cannot have objects that represent it. This is often exactly what you want.

Using the `Shape` example again, imagine that we want to build an extensive collection of two-dimensional shapes with the `Shape` class at the root of a tree of derived classes. We can subdivide shapes into polygons and closed curves. Among the polygons we could further partition them into regular and irregular polygons, and among closed curves, we could have other shapes such as ellipses and circles. The class `Polygon` could be derived from the `Shape` class and yet be abstract, because just knowing that a shape is a polygon is still not enough information to implement an `Area` member function. In fact, the `Area` member function cannot be implemented until the shape is pretty much nailed down. This leads to a multi-level hierarchy that takes the shape of a general tree with the property that the interior nodes are abstract classes and the leaf nodes are concrete classes. A portion of that hierarchy is shown in Figure 3.

# Algorithm Efficiency and Sorting

## 1 Introduction

The first objective of this chapter is to develop a system for evaluating and comparing the performance of *algorithms*. We are not interested in measuring the performance of *programs*. For this reason, we do not measure performance differences between two implementations of the same algorithm, nor are we concerned with precise calculations of running times, since these depend on the implementation, the compiler, the operating system, and the machine architecture. Instead, in this chapter we devise a means of evaluating an actual algorithm, independent of its implementation. To do this, we must work at a higher level of abstraction.

While one can actually "time" the execution of a program, this does not always help us to understand the cost of running the program with larger-sized input data. But on the other hand, one cannot "time" an algorithm, since an algorithm is not executed on hardware. Instead of timing an algorithm, we mathematically analyze its running time to assess its efficiency.

The running time of an algorithm depends on the input it is given. For some inputs it might be faster than for others. We are usually interested in knowing its worst case behavior, because then we can plan conservatively. Sometimes though, we want to know the "average" running time. The idea of "average" is a bit vague and also a bit useless, because "average" running time treats all inputs as equally likely to occur, which is never true. What is more useful is the *expected running time*, which is the weighted average, i.e., each input is weighted by its probability of occurrence. This is also a little impractical, as we rarely are able to weight the inputs realistically, because we do not know their respective probabilities of occurring. Therefore, people generally use the average running time, knowing that it is only an approximation. Nonetheless, expected case is sometimes used and we therefore define it here.

**Definition 1.** Let the set of all possible inputs to an algorithm be denoted $D$ ($D$ for domain.) Let$x$ be any element of the domain. Let $C(x)$ be a cost function that assigns a cost, such as the running time, to each input$x$, and let $p(x)$ be the probability that $x$ will be chosen as an input to the algorithm on an arbitrary run of it. Then

$$E[C] = \sum_{x \in D} p(x) \cdot C(x)$$

is the *expected value of the cost function* $C(x)$.

Because we do not have the tools yet to describe the running time of algorithms, we will use a program to illustrate how to apply this definition.

**Example 2.** Suppose an interactive program has a set of eight possible input commands, and market analysis has shown that their probabilities of being issued to the program are as in the table below. Suppose that cost function for each command has also been determined experimentally as a number of seconds it takes to execute the command and it too is in the table below. For simplicity, the commands are labeled c1, c2, ..., c8.

| $x$ | c1 | c2 | c3 | c4 | c5 | c6 | c7 | c8 |
|---|---|---|---|---|---|---|---|---|
| *p(x)* | 0.1 | 0.05 | 0.1 | 0.05 | 0.2 | 0.05 | 0.25 | 0.2 |
| *C(x)* | 5.0 | 2.0 | 3.0 | 4.0 | 8.0 | 2.0 | 1.0 | 3.0 |

Then the expected running time in seconds for this program with the given input distribution is

$$
\begin{aligned}
E[C] &= \sum_{x\in D} p(x)\cdot C(x)\\
&= (0.1\cdot 5.0)+(0.05\cdot 2.0)+(0.1\cdot 3.0)+(0.05\cdot 4.0)+(0.2\cdot 8.0)+(0.05\cdot 2.0)+(0.25\cdot 1.0)+(0.2\cdot 3.0)\\
&= 0.5+0.1+0.3+0.2+1.6+0.1+0.25+0.6\\
&= 3.65
\end{aligned}
$$

There are actually three different measures of the running time of any algorithm:

**Best case:** The running time under the best of all possible conditions.

**Expected case:** The running time averaged over all possible conditions, using some probability distribution of the input domain.[1]

**Worst case:** The running time in the worst of all possible conditions.

For each of these, the measure is expressed as a function of the size of the input, the idea of which will be explained shortly. Best case analysis is rarely used because one usually does not want to know how an algorthm will behave in the best possible situation. The expected case is useful only when the following three conditions are true:

1. It is possible to postulate a probability distribution of the input domain that reflects actual usage.
2. It is mathematically tractable to calculate.
3. Nothing catastrophic can result from using the algorithm under worst case conditions without having planned for it.

Because Condition 1 is unlikely to be true, a *uniform distribution* is often assumed. A uniform distribution is one in which every input is equally likely to occur. This makes the analysis less meaningful, since it may not reflect actual conditions. Condition 3 is usually the determining factor; if an algorithm absolutely must perform within a fixed running time, as when it is being used in an embedded system with real-time constraints, one cannot base an assessment on any kind of averaging of the running times.

Worst case analysis is almost always what people use to decide on the running time of an algorithm, because it provides an upper bound on how "bad" it can be.

## 2 Input Size

Every input is assumed to have an integer size that depends on the particular problem to be solved. For example, the input to an algorithm that sorts a list of items is a list. The number of elements in the list is the size of the input, not the lengths of the items to be sorted, or their combined lengths, because usually the sorting algorithm will depend on the number of elements, not how big each element might be. The input to an algorithm that searches for the occurrence of one string in a second string, would be the two strings, and so there may be two integers that influence the running time: the length of the first string and the length of the second. Different algorithms may depend on their sum, or product, or some other function of the two sizes.

Every algorithm has a particular set of input data, and the size of that data will depend on the particular problem. In any case, the running time will always be viewed as a function of the size of the input.

[1]If you have not yet had a course in probability, then the way to view this is as follows. Imagine that the set of all possible inputs to the algorithm is finite. Suppose that S is the name of the set, for argument's sake. Imagine that the program will be run in all kinds of environments, by all differ types of users. Some users are more likely to input a particular value than others. This leads to the idea that certain elements of S are more "likely" to be inputs to the program than others. The "likeliness" of these elements is called the probability distribution of S.

**Examples**

- For an algorithm that sorts an array of items, the input size is the number of items in the array.
- For an algorithm that searches for a given item within an array, the input size is the number of items in the array.
- For an algorithm that merges the lines of two different files in a particular order, the input size is two values, the number of lines in each file.
- For an algorithm that evaluates a polynomial, the input size is the number of nonzero terms of the polynomial.

## 3 Operator Counts as a Cost Function

The running time of an algorithm is obtained by assuming that each primitive operation takes the same amount of execution time and counting the total number of operations that the algorithm requires for a given input. For example, the `retrieve()` list member function may, in the worst case have to look at every element of the list in order to complete its task. The list is the input in this case, and the number of operations will be proportional to the length of the list.

Typically, all arithmetic operators, logical operators, control-flow operators, array subscripting operators, pointer dereferencing operators, and other operators similar to these take a single "step." Function calls, returns, and parameter transmission each count as a single step, even though they actually involve more instructions than simple assignment operators. The simplest way to approximate the number of steps is to count the operators, and the number of operands of each function call. Treat the parentheses of all conditions as a single operator.

**Example.** Consider the code

```
if ( z == f(x) )
    y = 0;
else if ( z > f(x) )
    y = 1;
else
    y = -1;
```

Executing this code, if `z == f(x)`, will take 6 operations: if-statement (1), relational operator (1), function call with one parameter passed and returned (3), assignment operator (1).

Executing the code when `z < f(x)` will take 11 operations: two if-statements (2), two relational operators (2), two function calls with one parameter passed and returned (6), one assignment (1).

There are simple algebraic rules for counting the number of steps in an algorithm.

### Rule 1. Loops

The running time of a loop is at most the running time of the statements inside the loop times the number of iterations of the loop. It can be less if the loop body has conditional statements within it and the number of statements within the branches varies.

**Rule 2 – Nested Loops**

The total running time of a statement inside nested loops is the running time of the statement multiplied by the product of the sizes of the loops.

```
for ( i = 0; i < n; i++ )
    for ( j = 0; j < m; j++ )
        k = k+1;
```

The body of the nested loops is 1 step if we assume the increment takes place in a register, so the running time is approximated by $n*m$. If we count more accurately, each iteration of a for-loop adds the comparison and the increment/assign, or 2 steps, and the final comparison that fails is 1 step, and the initialization is 1 step. Therefore, each iteration of the inner loop uses $1+3m+1=3m+2$ steps, and the total running time is $1+n\cdot(3m+2)+2n=3mn+4n+1$. We normally do not care about the added constants, nor the constant factors, so it is sufficient to say that the running time is $c_1mn+c_2n$ for some constants $c_1$and $c_2$.

```
for ( i = 0; i < n; i++ )
    for ( j = i; j < n; j++ )
        k = k+1;
```

The body of the nested loops is 1 step if we assume the increment takes place in a register. The inner loop executes $n-i$ times for each $i$, so the running time is approximated by $(1+2+3+...+n=n(n+1)/2$. Again, we would state that the running time is $cn(n+1)=cn^2+cn$ for some constant $c$.

**Rule 3 – Consecutive Statements**

The total running time of a sequence of statements is the sum of the running times of the statements. This makes sense. However, when we introduce order notation, the running time of a sequence of statements will be defined as the maximum of the running times of the individual statements. If, for example, a loop is one of the statements, then its running time may "dominate" the total time.

**Rule 4 – Conditional Statements**

The running time of an if/else statement

```
if (condition)
    S1
else
    S2
```

is at most the running time of the evaluation of the condition plus the maximum of the running times of `S1` and `S2`. Because we do not know which branch is taken in general, when we are interested in worst case analysis, we have to use the maximum, unless we can prove one of the branches is never executed with any inputs.

**Rule 5 – Function Calls**

If a sequence of statements contains function calls, the running time of the calls must be determined first. For recursive functions, the analysis can get difficult. This is a topic that we cover in the sequel to this class.

```
long fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

This is a recursive function. The running time for any $n <= 1$ is the cost of the parameter passing (1), the single if-test (1), the comparison (1), and the return (1), for a total of 4 steps. However, the cost for $n > 1$ is a recursively defined function. Let $T(n)$ be the running time of this function given input $n$. Then $T(0) = T(1) = 4$. For $n > 1$, the total is

$$T(n) = T(n-1) + T(n-2) + 9$$

because there 9 steps in addition to the two recursive calls. We will not solve this recurrence relation here. However, it is worth knowing that the running time is an exponential function of $n$.

### 3.1 Weakness of Counting Operations as a Measure

The idea of using operator counts is at this point questionable. In a modern processor, the dominant cost is not the time to execute the instructions, but the time to fetch data from primary memory. This can be an order of magnitude larger than the time to execute an instruction. However, there are many factors that can influence the number of times that data must be fetched from memory, because of caching within processors. The way in which the particular algorithm accesses its data, the relationship between data sizes and cache block sizes and the type of cache, and the amount of cache in general, all influence the overall time spent accessing memory.

## 4 Big "O" and Asymptotic Rates of Growth

First, we define a method of measuring how "fast" real-valued functions of a single variable can grow. Use your intuition here. Consider the non-decreasing functions

$$f(x) = x^2$$
$$g(x) = x^3$$
$$h(x) = 5x^2$$

Your intuition should tell you that as $x$ gets larger and larger, the function $g(x)$ grows faster and faster than the others. Furthermore, for any value of $x$, $h(x)$ will always be exactly $5f(x)$. So whatever the rate of growth of $f(x)$, $h(x)$ is growing at the same rate as $f(x)$. They stay in a kind of lock step, with $h$ and $f$ proportionally the same as $x$ marches towards infinity. On the other hand, as $x$ increases, clearly $g(x)$ gets larger and larger than both $f(x)$ and $h(x)$. To see this, look at the first six integer cubes: 1, 8, 27, 64, 125, 216 in comparison to the first six integer squares: 1, 4, 9, 16, 25, 36. Even the first six values of $h(x)$ are overtaken by the faster growing $g(x)$: 5, 20, 45, 80, 125, 180. The point is that whatever means we use to measure the relative rates of growth of functions, it ought to ignore constant factors such as the 5 above, and must rank functions like cubics ahead of quadratics.

Table 1 shows the approximate values of selected functions with varying rates of growth. A good method of measuring the rates of growth of functions would rank these functions in the order they appear in the table, because each successive row is growing faster than the preceding one.

***Order of magnitude*** analysis is used for comparing the relative rates of growth of the running times of algorithms. Order of magnitude will ignore constant multiples and differences of functions. Instead of making a statement such as

| | $n$ | | | | |
|---|---|---|---|---|---|
| $f(n)$ | 10 | 100 | 1000 | 10000 | $10^6$ |
| $log_2(n)$ | 3 | 6 | 9 | 13 | 19 |
| $n$ | 10 | 100 | 1000 | 10000 | $10^6$ |
| $n\log_2 n$ | 30 | 664 | 9965 | $10^5$ | $10^7$ |
| $n^2$ | $10^2$ | $10^4$ | $10^6$ | $10^8$ | $10^{12}$ |
| $n^3$ | $10^3$ | $10^6$ | $10^9$ | $10^{12}$ | $10^{18}$ |
| $2^n$ | $10^3$ | $10^{30}$ | $10^{301}$ | $10^{3010}$ | $10^{301030}$ |

Table 1: Approximate values of selected functions of n

"Algorithm A runs in $5n^2$ steps for inputs of size $n$."

we will say that

"Algorithm A runs in time proportional to $n^2$ for inputs of size $n$."

Suppose algorithm A solves a problem in time proportional to $n^2$ and algorithm B solves the exact same problem in time proportional to $n$ for inputs of size $n$. Then even if B's more accurate running time were $100n$, as $n$ increases, A's running time would overtake B's.

What matters most is how fast the function grows as $n$ increases. In the above example, the rate of growth of A's running time, $n^2$, is greater than the rate of growth of B's running time, $n$.

**Definition 3.** Algorithm A is order $f(n)$, denoted $O(f(n))$, if for any implementation of algorithm A, there is a positive constant $c$ and a constant $n_0$ such that for all $n \geq n_0$, A requires no more than $c \cdot f(n)$ time units to solve a problem of size $n$.

The requirement that $n \geq n_0$ in the definition is the mathematical way to say that, for inputs that are sufficiently large, the running time is bounded by a constant multiple of $f(n)$. You pronounce the notation $O(f(n))$ as "big-O of f(n)". This is called ***order notation*** and the "big-O" is a way to characterize the *order of magnitude* of the rate of growth of the function.

**Example 4.** Suppose the running time of an algorithm with input size $n$ is $3n^2+5n-6$. Then the algorithm is $O(n^2)$ because there is a constant $c = 8$ and an integer $n_0 = 0$ such that for all $n \geq n_0$, $3n^2+5n-6 \leq cn^2$. If you substitute the value of $c$, you will see that

$$3n^2 + 5n - 6 \leq 8n^2 \quad iff \quad 0 \leq 5n^2 - 5n + 6$$

which is true for any value of $n$ greater than or equal to 0.

**Example 5.** Suppose the running time of an algorithm with input size $n$ is $n^4 + 100n^3 + 100$. Then the algorithm is $O(n^4)$ because for all $n \geq 2$, $n^4 + 100n^3 + 100 \leq 101n^4$. To see this, note that

$$\begin{aligned} n^4 + 100n^3 + 100 &\leq 101n^4 \qquad iff \\ 0 &\leq 100n^4 - 100n^3 - 100 \qquad iff \\ 0 &\leq n^4 - n^3 - 1 \end{aligned}$$

and since for any $n > 1$, $n^4 - n^3 - 1 \geq 0$. Therefore, the constant $c = 101$ and the constant $n_0 = 2$.

Although we have applied the definition strictly in these examples, there are rules that make it easy in many cases to show that the running is big-O of some function. For example, you should see that if an algorithm is $O(f(n))$ for some function $f(n)$ , then it is $O(k \cdot f(n))$ for any constant $k$.

*Proof.* Suppose that the algorithm is $O(f(n))$ for some function $f(n)$, and let $k$ be a positive constant. By the definition of big-O there is a positive constant $c$ such that the algorithm runs in at most $cf(n)$ steps for sufficiently large $n$. Let $b = c/k$. Since $b \cdot k \cdot f(n) = (c/k) \cdot k \cdot f(n) = c \cdot f(n)$, there is a constant $b$ such that the algorithm runs in at most $b \cdot kf(n)$ steps for all sufficiently large $n$, or stated in terms of big-O, the algorithm is $O(kf(n))$. $\square$

Another way to state the preceding observation is that, for any function $f(n)$ and any positive constant $k$,

$$O(f(n)) = O(k \cdot f(n))$$

In short, constant multiples of functions have the same order of magnitude as each other, so there is never a need to multiple a function by a constant greater than 1.

*Remark.* You will see in almost every context that authors will write statements such as

$$T(n) = O(f(n))$$

or

$$g(n) = O(f(n))$$

to mean that the function $T(n)$ is big-O of $f(n)$ or that $g(n)$ is big-O of $f(n)$. The use of the equality symbol is misleading and technically incorrect. The notation $O(f(n))$ is actually describing a set. It is the set of all functions $g(n)$ such that there are constants $c$ and $n_0$ such that $g(n) \leq c \cdot f(n)$ for $n \geq n_0$. A better way to write the preceding statements, one that does not mislead you, is

$$T(n) \in O(f(n))$$

or

$$g(n) \in O(f(n)).$$

If you realize that $O(f(n))$ is a set of functions that grow no faster than $f(n)$, then it will make the following statements easy to understand.

1. If $g(n) \in O(f(n))$, then if $h(n) \in O(f(n))$ it is also true that $h(n) \in O(f(n) + g(n))$. Another way to say this is that adding a slower growing function to $f(n)$ does not change the set of functions that grow no faster than it.
2. A corollary is that adding any constant to$f(n)$ does not change the set of functions in $O(f(n))$, since all constants are $O(f(n))$ for any function $f(n)$. (Why?)
3. For any constant $c$, $O(c) = O(1)$. This is the set of constant functions.
4. $O(f(n)) \bigcup O(g(n)) = O(f(n) + g(n))$. In other words, the set of functions that grow no faster than the sum of $f(n)$ and $g(n)$ is the union of the sets of functions that grow no faster than each. (Why?)

Last, in case you do not remember the basics about logarithms, it does not matter what the base of the log function is in terms of the rate of growth, because, for any positive numbers $a$ and $b$ there is a constant $c$ such that

$$\log_a n = c \cdot \log_b n$$

Just take $c = \log_a b$. To see this, let $m = \log_a n$. Then $n = a^m = a^{(c)^{m/c}} = a^{(\log_a b)^{m/c}} = b^{m/\log_a b}$. Taking $\log_b$ of both sides, we get

$$\log_b n = \frac{m}{\log_a b} = \frac{\log_a n}{\log_a b}$$

implying the result. This means that for any $a$ and $b$, $\log_a(n) \in O(\log_b n)$.

### 4.1 A Hierarchy of Functions

We can define a binary comparison operation on two orders of magnitude, $O(f(n))$ and $O(g(n))$ as follows. We will write

$$O(g(n)) < O(f(n))$$

if and only if

$$O(g(n)) \subsetneq O(f(n))$$

In other words, if $O(g(n)) < O(f(n))$ then all functions that grow no faster than $g(n)$ grow no faster than $f(n)$, but there are functions that grow faster than $g(n)$ that do not grow faster than $f(n)$. For example, $O(n^2) < O(n^3)$. If you need a function that acts like a "witness" to this, notice that if $h(n) = 2n^3$ then $h(n) \in O(f(n))$ but $h(n) \notin O(g(n))$. It should be clear that any function that is in $O(n^2)$ must be in $O(n^3)$. (Why?)

More generally, $O(n^k) < O(n^m)$ whenever $k < m$. This can also be proved easily.

This "less-than"-looking relation is a transitive relation. The following is a hierarchy of functions of successively greater rate of growth:

$$O(1) < O(\log_2(n)) < O(\log^c(n)) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^b) < O(2^n) < O(3^n) < \cdots < O(n!)$$

where $c$ is any number greater than 1 and $b$ is any number greater than 3.

### 4.2 Practical Matters

You will find, when we start to analyze the various algorithms that we will encounter, that certain rates of growth will be prevalent, among which are

$O(\log n)$**.** Algorithms such as binary search have ***logarithmic*** running time.

$O(n)$**.** An algorithm that whose running time is $O(n)$ is said to have ***linear*** running time.

$O(n \log n)$**.** Algorithms that use divide and conquer strategies often have running times that are $O(n \log n)$. We will see that certain sorting algorithms have worst case behavior that is $O(n \log n)$.

$O(n^2)$**.** An algorithm whose running time is $O(n^2)$ is said to have ***quadratic*** running time. Many inefficient sorting algorithms have quadratic running time in their worst cases.

$O(c^n)$**.** Algorithms whose running time is $O(c^n)$ for some $c > 1$ have ***exponential*** running time. There are classes of problems for which it is generally believed that any algorithm that solves them must have exponential running time. Such problems are called ***intractable*** problems.

## 5 Efficiency of Search Algorithms

Unless otherwise stated, the size of the input to all problems will be denoted by $n$. For search algorithms, this means that they search through a container (an array, a vector, a list, etc.) that contains $n$ items.

### 5.1 Linear Search

Linear search is a method of searching a list of key-value pairs one after the other, starting at the beginning of the list, until either the key is found or the end of the list is reached. It does not matter whether the list is a linked list or an array; the algorithm is the same. The only difference is in how the elements are accessed and how the structure is traversed.

We can use the term *iterator* to refer to an object that, at any time, references a particular element in the structure. So, for example, if the list is stored in an array, then the array index will be the value of the iterator. If the list is stored in a linked list, then a node pointer will be the value of the iterator. With this in mind, linear search is of the form

```
list.iterator = list.reference_to_first_value;
while ( list.iterator refers to a valid item ) {
    if ( list.iterator.key matches search_key )
        return list.iterator.value;
    else
        list.iterator++;
}
return item_not_in_list;
```

Basically this starts at the beginning of the list and compares the search key to the key in the list. On a match it quits, otherwise it continues to the next item, until the entire list is searched.

What is the running time of this algorithm as a function of the list size, in the worst case?

The worst case is when no item in the list matches the key. In this case, it has to compare the search key to every item's key. Thus, for searches that fail it performs n comparisons and the worst case is therefore $O(n)$.

What about the average case running time?

The number of comparisons made if the search key matches the first item is 1, if it matches the second item, 2, if it matches the $k^{th}$ item, $k$. If it is equally likely to stop at any of these nodes, or not be there at all, then the expected case running time is the average, which is $(1+2+3+\cdots+n)/(n+1) = n(n+1)/(2n+2) = n/2$, which is $O(n)$. We divide by $n+1$ instead of $n$ because there are $n+1$ cases – matching any of the $n$ list items, and not matching any of them.

### 5.2 Binary Search

Binary search can only be applied to data that is sorted and that is in a structure with efficient random access, such as an array. There is no advantage to performing binary search on a data structure that only provides sequential access, since each access requires traversing the structure repeatedly. Therefore we will assume the data is in an array. Binary search of an array is of the form

```
Let the search region be the entire array.
while the item is not found and the search region is not empty {
    let middle be the index at the middle of the search region;
    if the key is smaller than the middle
        make the search region the lower half and search in there;
    else if the key is larger than the middle
        make the search region the upper half and search in there;
    else
        return middle;
}
return item_is_not_found;
```

What is the worst case running time as a function of $n$?

The worst case will occur when the item is not in the array, or when, as luck will have it, it is not in a computed middle element until the search region is size 1. Each iteration of the loop divides the size of the search region in half, approximately. In the worst case, it stops when the search region reaches size 1. At that point , if the item is not found in the middle element, the search region will become empty (that is in the code details, which are omitted in the above pseudo-code), and the loop will exit. If it is found, the loop will exit anyway.

In each iteration, at least one key comparison is made. (It could be two, but we will ignore that fact for the moment.) How many iterations will it take for the size of the search region to become 1 if it is initially of size $n$, which we will first assume is a power of 2, and is repeatedly divided in half? The size starts out at $n = n/2^0$, then becomes $n/2$, then $n/(2^2)$, then $n/(2^3)$ and so on until it has been divided $k$ times and $n/(2^k) = 1$, which implies that $n = 2^k$, or that $k = \log_2(n)$. Therefore, there are $\log_2(n) + 1$ iterations. Because we make a comparison for each value of $k$ from 0 to $\log_2(n)$, we make $\log_2(n) + 1$ comparisons. If $n$ is not a power of 2, $k$ would not be an integer, and the more accurate answer is that we make $\lfloor \log(n) \rfloor + 1$ comparisons, or equivalently, $\lceil \log_2(n + 1) \rceil$ comparisons. This shows that the number of comparisons will be proportional to $\log_2(n)$ or that it is $O(\log_2 n)$. Because we ignore constants of proportionality, the fact that two comparisons might be made in each iteration does not affect this answer.

In conclusion, binary search is much faster than linear search, but it requires that the data be maintained in sorted order. Therefore, one has to factor in the cost of keeping the array sorted to have a meaningful comparison of efficiency.

## 6 Efficiency of Sorting Algorithms

### 6.1 Preliminaries

Sorting is the act of rearranging data so that it is in some pre-specified order, such as ascending or descending order. The exact ordering does not really matter, as long as it is a linear, or total ordering, which means that any two elements can be ordered. For simplicity, we will assume that all sorting algorithms put their elements in ascending order.

Sometimes we sort scalar objects such as numbers, and other times we might want to sort non-scalar objects such as records with multiple members. In the latter case, one must designate a specific member to be used as the ***sort key***. A ***sort key*** is part of the data item that determines the ordering relation used in the sort.

Sometimes the data to be sorted is so large that it cannot fit in memory. The sorting algorithms in this case have very different requirements, since only a fraction of the data can be examined at a time. Algorithms that sort data that cannot all be stored in memory are called ***external sorting algorithms***. Those that sort data that can fit entirely in memory are called ***internal sorts***.

Sometimes the data to be sorted does not have unique keys and multiple elements can have the same key. It may be important for a sorting algorithm to preserve the relative order of elements with equal keys. Sorts that preserve the order of equal keys are called ***stable sorts***. To give an example, suppose that data is sorted by last name, and if there are any elements with equal last names, then ties are broken by sorting by first name. One way to do this is to first sort all of the data by first name. Then, the data can be sorted by last name, as long as it has the property that elements with equal keys stay in the same relative order. This will work because if there are two people with the same last name but different first names, then the second sort will move elements with different last names relative to each other, but those with equal last names will stay in the same relative order, and since they are already sorted by first name, those with smaller first names and equal last names will precede those with larger first names and equal last names. E.g.,

```
zachary abigail
smith harry
```

```
jones mary
smith sam
```

becomes

```
jones mary
smith harry
smith sam
zachary abigail
```

When data is sorted, mostly what takes place is that keys are compared and data items are moved around. When measuring the performance of sorting algorithms, sometimes it is only the number of key comparisons that are of interest. However, if the data items are very big, then moving a data item can be a costly operation. In this case we may also be interested in how many data movement operations take place as a function of input size. The number of key comparisons might be small but if the data movements are frequent, then it may not perform as well as an algorithm with fewer data movements and more key comparisons. One measure of a sort is how much data movement it performs when the initial data set is already sorted. Sorts that do not move the data much in that case are called ***natural sorts***.

The algorithms that we will examine are all internal sorting algorithms that act on data stored in arrays. We begin with selection sort.

## 6.2 Selection Sort

Selection sort proceeds in stages. At the start of each stage, the array will be partitioned into a two disjoint regions. The upper region will contain those elements already in sorted order. The lower region will contain those elements that are not necessarily sorted. (They might be by chance.) The upper region will also have the property that every element in it is greater than or equal to any element in the unsorted region.

The action performed during each stage is therefore to find the largest key in the unsorted region and put it into the lowest position of the sorted region. Since it was in the unsorted region, it is less than or equal to every key in the sorted region. By placing it in the lowest position in the sorted region it preserves the ordering of that region. In addition, the size of the unsorted region is reduced by one. This implies that after n steps, where n is the number of elements, the unsorted region will be size zero, or that the array has been sorted.

### Algorithm

The following function defines selection sort of an array `A` of size `n`. `T` is the underlying element type. The `index_of_maximum_key()` function performs a linear search of its array argument from index 0 to index `last` for the key with maximum value and returns its index. The `swap()` function swaps its arguments.

```
void SelectionSort ( T A[ ], int n)
{
    // Precondition:  A[0..n-1] contains the data to be sorted
    // Postcondition: A[0..n-1] is sorted in ascending order.
    int largest;      // index of largest item in unsorted part of array
    int last;         // index of last item in unsorted part of array
    for ( int last  = n-1; last >= 1; last--) {
        //  Invariant:   A[last+1...n-1] is already sorted and
        //               if j <= last and k > last, A[j] <= A[k]
        largest = index_of_maximum_key(A, last);
        swap( A[largest], A[last]);
    }
}
```

**Analysis**

During each iteration of the loop, `index_of_maximum_key()` is called once with an array of size `last`+1. To find the maximum element in an array of size n, n-1 comparisons must be performed. (Why not n?) Therefore, the function performs `last` comparisons when its second argument is `last`. Since `last` runs from n-1 down to 1, the total number of key comparisons is the sum of the numbers 1, 2, 3,..., n-1, which is $n(n-1)/2$. Notice that the total number of key comparisons performed by selection sort is the same in all cases – it does not depend on the data.

The `swap()` function performs a constant number of data exchanges. To swap two elements requires three assignments. It is called once in each iteration of the loop. Since there are $(n-1)$ iterations, there are $3(n-1)$ data exchanges caused by `swap()`. As there is no other data movement in the algorithm, the total number of data exchanges is $3(n-1)$. This too is independent of the original state of the data.

Expressing these results in order notation, we can see that selection sort has a worst case, average case, and best case running time that is $O(n^2)$.

## 6.3 Insertion Sort

Insertion sort also splits an array into a sorted and an unsorted region, but unlike selection sort, it repeatedly picks the lowest index element of the unsorted region and inserts it into the proper position in the sorted region. Hence its name.

Although it makes no difference whether the sorted region is below the unsorted one or not, in this description the sorted region initially consists of the single element A[0] and grows upward. An array of size one is always sorted. Unlike selection sort, the sorted region starts out at non-zero size. The unsorted region is everything from the second array element to the last one.

The algorithm starts at the second position and stops when the element in the last position has been inserted, thereby forcing the size of the unsorted region to zero. Inserting an element into a specific position requires shifting data. The procedure that insertion sort uses is to copy the element to be inserted next into a temporary location, then scan downwards from its initial position while the keys are larger than it. When a key is found that is not larger, the element is inserted above that key and below the one right above it. To make this possible, as the keys are examined during the scan, each element is shifted up one to make room for the inserted element. The algorithm is as follows:

```
void InsertionSort ( T A[ ], int n)
    {
        // Precondition:  A[0..n-1] contains the data to be sorted
        // Postcondition: A[0..n-1] is sorted in ascending order.
        // for each array element A[i], insert it into the sorted region
        for ( int i = 1; i < n; i++) {
            // A[0..i-1] is in sorted order
            tmp = A[i];   // copy
            j = i;
            while ( j > 0 && tmp < A[j-1] ) {
                A[j] = A[j-1];
                j--;
            }
            A[j] = tmp;
            // A[0..i] is now sorted
        }
    }
```

**Analysis**

In the worst case, the next item to be inserted into its position in the sorted region is always smaller than everything in that region. In other words, the initial array is in reverse sorted order. In this case, the inner while loop makes i key comparisons, since it starts comparing when j = i and stops when j = 1. As i runs from 1 to n-1, the total number of comparisons is

$1 + 2 + 3 + \cdots (n-2) = n(n-1)/2.$

Each time that the inner while loop iterates, it makes one data exchange. The number of iterations of that loop is the current value of i. So the number of data exchanges within that while loop in the worst case is the same as the number of comparisons, or $n(n-1)/2$. In addition, it makes one data exchange before and after the loop. Therefore the number of data exchanges in the worst case is $n(n-1)/2 + 2(n-1) = n^2 + 3n - 4$.

In short the worst case behavior of insertion sort is $O(n^2)$. It moves much more data than selection sort, since selection sort moves $O(n)$ data items as compared to insertion sort's $O(n^2)$ data exchanges. But insertion sort is stable; if the array is sorted initially, it will only perform $2(n-1)$ data exchanges.

## 6.4 Quicksort

Quicksort is the fastest known sorting algorithm, on average, when implemented *correctly*, which means that it should be written without using recursion, using a stack instead, and putting the larger partition onto the stack each time. But this will not have much meaning to you now. It was invented, or perhaps we should say "discovered" by C.A.R. Hoare in 1960 when he was just 26 years old, and published in the Communications of the ACM in 1961; it revolutionized computer science. He was knighted in 1980.

Quicksort is a divide-and-conquer sorting algorithm. The general idea is to pick an element from the array and to use that element to partition the array into two disjoint sets: one that consists of all elements no larger than the selected element, and the other consisting of elements no smaller than the selected element. The first set will reside in the lower part of the array and the second in the upper part. The selected element will be placed between the two. This process is recursively repeated on each of the two sets so created until the recursion reaches the point at which the set being partitioned is of size one.

### 6.4.1 The Basic Idea

Let $S$ be the set of elements in the array. $S$ may have elements with equal keys. Quicksort(S) performs the following steps:

1. If the number of elements in $S$ is 0 or 1, then do nothing and return.
2. Pick any element $v$ in $S$. This element is called the ***pivot***.
3. Partition $S$–$v$ into two disjoint sets $S1 = x \mid x \in S \wedge x \leq v$ and $S2 = x \mid x \in S \wedge x \geq v$. If an element is equal to $v$ it will be placed into one of $S1$ or $S2$, but not both.
4. Form the sorted array from Quicksort(S1) followed by $v$ followed by Quicksort(S2).

To illustrate, suppose an array contains the elements

```
8 1 4 9 0 3 5 2 7 6
```

Suppose that we pick 6 as the pivot. Then after partitioning the array it will look like

```
1 4 0 3 5 2 6  8 9 7
```

where S1 is the set {1,4,0,3,5,2} and S2 is the set {8,9,7}. Space is inserted just to identify the sets. If we recursively repeat this to these sets S1 and S2 (and assume by an induction argument that it does indeed sort them), then after the recursive calls to quicksort S1 and S2, the array will be in the order

```
0 1 2 3 4 5  6   7 8 9
```

### 6.4.2 The Partitioning Algorithm

Quicksort can only be efficient if the array can be partitioned in a single pass, i.e., if the partition step takes time proportional to $n$, the size of the array. Each element should be compared to the pivot once. There are various methods of achieving this. The algorithm described here uses two variables that act like inspectors. These variables start at opposite ends of the array and march towards each other. As they march, they compare the elements they pass on the way. The inspector that starts on the bottom looks for array elements that are greater than or equal to the pivot. If it finds one, it stops on it and waits. The inspector that starts at the top looks for array elements that are less than or equal to the pivot. If it finds one, it stops on it. When each is stopped on an element, the two elements are swapped, because each is in the wrong set[2]. The inspectors then resume their marches. If, when either one is marching ahead, it passes the other one, then the partitioning has finished because it means that everything that they passed on the way to each other has been placed into the proper set.

```
8 1 4 9 0 3 5 2 7 6       initial array
i               j pivot

8 1 4 9 0 3 5 2 7 6       i found 8; j found  2
i             j

2 1 4 9 0 3 5 8 7 6       2 and 8 are swapped, then advance
  i         j

2 1 4 9 0 3 5 8 7 6       i found 9; j found 5
      i     j

2 1 4 5 0 3 9 8 7 6       9 and 5 are swapped, then advance
        i j

2 1 4 5 0 3 9 8 7 6       i advanced past j
        j   i

2 1 4 5 0 3 6 8 7 9       pivot swapped with A[i]
```

In the above example the array elements were all unique. Soon you will see what happens when they are not all unique.

The first step in the algorithm is to pick a pivot element. For reasons of performance, the pivot should never be the very largest or very smallest element. Once way to make this very unlikely is to pick three random elements, sort them, and make the pivot the middle value. The choose_pivot() function will do that and more. It will also rearrange the array so that the first element has the smallest of the three values, the last element has the pivot, and the second-to-last has the largest element. It will look like this:

```
smallest  ............. largest pivot
0                         n-2    n-1
```

[2]The inspectors stop on elements equal to the pivot. They are not in the wrong set. It will be seen that this prevents poor performance. That will be explained shortly.

assuming the array has $n$ elements. The reason for this is that the smallest and largest will act as sentinels, preventing the partitioning step from going out of the array bounds. Since the array has to have at least three elements to do this, quicksort will only be called for arrays of size three or larger.

```
void quicksort( T A[], int left, int right)
{
    // make sure array has at least three elements:
    if ( left + 2 <=  right ) {
        // pick three values, sort them, and put the pivot into A[right],
        // the smallest of the three into A[left] and the largest into A[right-1].
        // assume that the function choose_pivot() does all of this.
        T pivot  = choose_pivot(A, left, right);

        // A[left] <= pivot <= A[right-1] and pivot is A[right]

        // now partition
        int i = left;
        int j = right-1;
        bool done = false;

        while ( ! done ) {
            while( A[++i] < pivot ) { }   // advance i
            while ( pivot < A[--j]) { }   // advance j

            // A[i] >= pivot and A[j] <= pivot
            if ( i < j )
                swap( A[i], A[j] );
            else
                done = true;
        }
        // now j <= i and A[i] >= pivot and A[j] <= pivot
        // so we can swap the pivot with A[i]
        swap( A[i], A[right -1] );

        // Now the pivot is between the two sets and in A[i]
        // quicksort the left set:
        quicksort(A, left, i-1);

        // quicksort the right set:
        quicksort(A,i+1, right);
    }
    else {
        // A has just two elements
        if ( A[left] > A[right] )
            swap(A[left], A[right] );
    }
}
```

### 6.4.3 Analysis

The outer loop of quicksort iterates until i and j pass each other. They start at opposite ends of the array and advance towards each other with each key comparison. Therefore, roughly n key comparisons take place for an array of size n. The function is called twice recursively, once for each set. Suppose that the pivot is chosen very badly each time and that it is always the largest element in the set. Then the upper set will be empty and the lower set will be roughly size n-1. This means that the recursive call to the lower set will make n-1 key comparisons and that to the upper will not even execute. This implies that the recursion will descend down the lower sets, as they get smaller by one each time, from n, to n-1 to n-2 to n-3 and so on until the last call has an array of size 3, when it stops. The total number of key comparisons is therefore roughly

$$3+4+5+\cdots(n-2)+(n-1)+n \approx n(n-1)/2 = O(n^2).$$

Thus, the worst case of quicksort is $O(n^2)$. That is why it is important to choose the pivot properly. A poor choice of pivot will cause one set to be very small and the other large. The result will be that the algorithm achieves its $O(n^2)$ worst case behavior. If the pivot is smaller than all other keys or larger than all other keys this will happen. Quicksort is best when the pivots divide the array roughly in half every time. Since the array is divided in half each time, the depth of recursion is $\log_2 n$, when the array is sorted. It can be proved that each level of the recursion uses $O(n)$ key comparisons, so that at best it takes $O(n \log n)$ steps. It can be proved that the average case is $O(n \log n)$ as well. In fact, it can be proved that the average case requires about 1.39 times the number of comparisons as the best case.

### 6.4.4 Equal Keys

Consider the array

```
1 1 1 1 1 1 1 1 1 1
```

in which all keys are identical. A good implementation of quicksort should take $O(n \log n)$ steps in this case. If you follow the algorithm described above, you will see that it does. Because i and j stop on keys equal to the pivot, they each progress one element at a time and swap, eventually meeting in the center:

```
1 1 1 1 1 1 1 1 1 1        initial array
i               j pivot

1 1 1 1 1 1 1 1 1 1        i and j swap and advance 1 each
  i           j

1 1 1 1 1 1 1 1 1 1        i and j swap and advance 1 each
    i       j

1 1 1 1 1 1 1 1 1 1        i and j swap and advance 1 each
      i   j

1 1 1 1 1 1 1 1 1 1        i and j swap and advance 1 each
        i                  they are now on the same element
        j                  and the partition stops
```

Now the two sets are equal size, and the recursion is balanced. This leads to the best case behavior.

To be efficient, quicksort should be implemented non-recursively. The idea is to use a stack to store the parameters of the recursive call. Instead of making the two recursive calls, the parameters of one of the recursive calls are pushed on the stack and the other call is executed within the while loop again. The stack is popped to retrieve the parameters and simulate the call. To avoid the stack growing too large, the call for the larger partition is pushed onto the stack and the one for the smaller is executed immediately. You can find the details elsewhere.

# Chapter 10: Trees

## 1 Introduction

Trees are an abstraction that can be used to represent hierarchical relationships, such as genealogies, evolutionary trees, corporate structures, and file systems.

Previous data types have been linear in nature. Trees are a way to represent a specific kind of non-linear abstraction in which there is a *parent/child* type of relationship. For the moment, ignore the fact that most animals require two parents to create children. Instead assume *asexual reproduction*, or *agamogenesis*. Then every parent can have multiple children, but each child has but a single parent. Let us think about such a species for now.

Consider the set of all individuals in such a species's population. Define a binary relationship $e$ on this set as follows: for any two members of this set $x$ and $y$, *e(x,y)* is true if and only if $x$ is the parent of $y$. We can represent $x$ and $y$ as little circles on a paper. We call such circles ***nodes***. If *e(x,y)* is true, we can connect $x$ and $y$ with an arrow leading from $x$ to $y$. We call these arrows ***directed edges***. A directed edge from $x$ to $y$ indicates that $x$ is the parent of $y$, or that $y$ is the child of $x$. We can also write a directed edge as an ordered pair *(x,y)*.

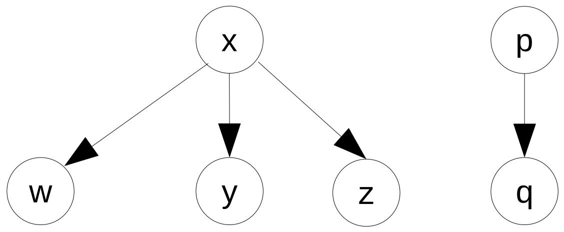


Figure 1: Ordered pairs in a set.

In Figure 1, the directed edges are $(x, y)$, $(x, z)$, $(x, w)$, and $(p, q)$ and nothing else. Thus, x is the parent of w, y, and z, and p is the parent of q. If p and x are both children of a node a, then the population would be represented as in Figure 2.

You can imagine that we could arrange the nodes on a very large piece of paper in such a way that they look like an upside-down tree.

## 2 Trees

There is a lot of terminology to learn, and many definitions are about to follow. There are two basic types of trees, ***general trees*** and ***n-ary trees***. We begin with general trees.

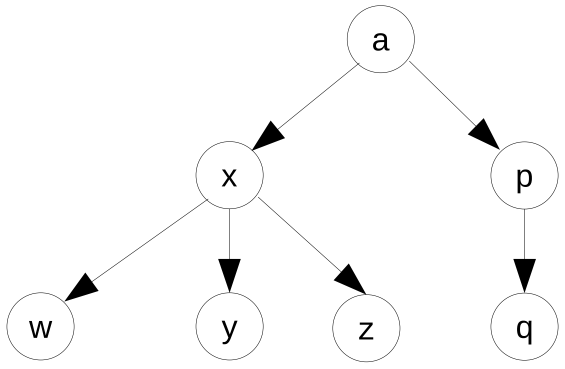


Figure 2: Ordered pairs with common ancestor.

## 2.1 General Trees

**Definition 1.** A ***general tree*** T consists of a possible empty set of nodes. If it is not empty, it consists of a unique node $r$ called the ***root*** of T and zero or more non-empty trees $T_1$, $T_2$, ..., $T_k$ such that there is a directed edge from $r$ to each of the roots of $T_1$, $T_2$, ..., $T_k$. A ***subtree*** of a tree T is any tree whose root is a node of T.

In Figure 2, a is the root of the tree. It has two subtrees, whose roots are x and p. Notice that in a general tree, the subtrees are non-empty – it has to have at least one node – but that there may be zero subtrees, meaning that a general tree can have exactly one node with no subtrees at all. In Figure 2, the nodes w, y, z, and q are the roots of trees with no subtrees, and p is the root of a tree with a single subtree whose root is q.

**Definition 2.** A ***forest*** is a collection of non-empty general trees.

*Remark.* You can always create a tree from a forest by creating a new root node and making it the parent of the roots of all of the trees in the forest. Conversely, if you lop off the root of a tree, what is left is a forest.

### 2.1.1 Applications of General Trees

In a file system, a node represents each file, and if the file is a directory, then it is an internal node whose children are the files contained in the directory. Some file systems do not restrict the number of files per folder, implying that the number of children per node is varying and unbounded.

In computational linguistics, as sentences are parsed, the parser creates a representation of the sentence as a tree whose nodes represent grammatical elements such as predicates, subjects, prepositional phrases, and so on. Some elements such as subject elements are always internal nodes because they are made up of simpler elements such as nouns and articles. Others are always leaf nodes, such as nouns. The number of children of the internal nodes is unbounded and varying.

In genealogical software, the tree of descendants of a given person is a general tree because the number of children of a given person is not fixed.

A tree need not be drawn in a way that looks like a hierarchy and yet it still is a tree. Figure 3 contains a fragment of the evolutionary tree, with its root at the center of a circle and the subtrees radiating away from the center.

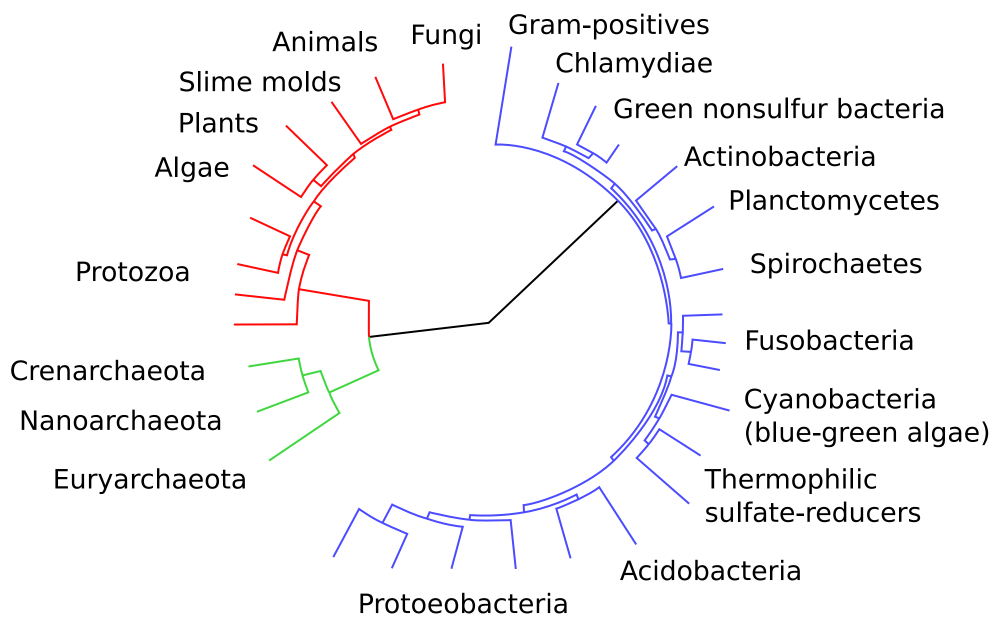


Figure 3: Evolutionary tree: Diagrammatic representation of the divergence of modern taxonomic groups from their common ancestor (from Wikipedia http://en.wikipedia.org/wiki/Phylogenetic_tree.)

## 2.2 Tree Terminology

The remainder of this terminology applies to all kinds of trees, not just general trees.

**Definition 3.** A node in a tree is called a ***leaf node*** or an ***external node*** if it has no children.

In Figure 2, the leaf nodes are w, y, z, and q.

**Definition 4.** A node is called an ***internal node*** if it is not a leaf node.

An internal node has at least one child. In Figure 2, the internal nodes are a, x, and p.

Just as it makes sense to talk about siblings, grandparents, grandchildren, ancestors, and descendants with respect to people, so it is with nodes in a tree.

**Definition 5.** Two nodes are ***siblings*** if they have the same parent.

**Definition 6.** The ***grandparent*** of a node is the parent of the parent of the node, if it exists. A ***grandchild*** of a node is a child of a child of that node if it exists. More generally, a node $p$ is an ***ancestor*** of a node $t$ if either $p$ is the parent of $t$, or there exists a node $q$ such that $q$ is a parent of $t$ and $p$ is an ancestor of $q$. If $p$ is an ancestor of $t$, then $t$ is a ***descendant*** of $p$.

In Figure 2, a has 4 grandchildren.

Notice that the definition of an ancestor is recursive. Many definitions of properties or relationships having to do with trees are recursive because a tree is essentially a recursively defined structure.

**Definition 7.** The ***degree of a node*** is the number of children of that node. The ***degree of a tree*** is the maximum degree of all of the nodes in the tree.

Notice that a tree can have many nodes with small degree, but if one node in it has large degree, then the tree itself has that degree. The degree of a tree is just one of many quantitative properties of a tree.

**Definition 8.** A ***path*** from a node $n_1$ to node $n_k$ is a sequence of nodes $n_1$, $n_2$, ..., $n_k$ such that $n_i$ is the parent of $n_{i+1}$ for $1 \leq i < k$. The ***length*** of a path is the number of edges in the path, not the number of nodes!

*Note* 9. Some books will define the length of a path as the number of nodes in the path, not the number of edges. This will have an effect on the remaining definitions and the details of many theorems and proofs of these theorems. It is important to know which definition is being used. In these notes, it is always the number of edges in the path.

If we start at the root and travel down the paths from the root, we can define a notion of the levels of a tree. The root is at the first level, sometimes labeled 0 and sometimes 1. We will label the level of the root 1. The children of the root are at level 2. More generally, the level of a node n in the tree is the level of its parent plus 1. All nodes at a given level are reachable from the root in the same number of steps.

The height of a tree is another concept that is not universally agreed upon. However, the standard definition (which differs from the one used in the textbook), is as follows.

**Definition 10.** The ***height*** of a node is the length of the longest path from the node to any of its children.

This implies that all leaf nodes have height = 0. The height of the node z in 2 is 0. The height of the node labeled p is 1, and the root has height 2. Again, many books will state that the height of leaf nodes is 1, not 0.

**Exercise.** Height can be defined recursively. Write its definition.

**Definition 11.** The ***height*** of a tree is the height of the root node of the tree. Another way to put it is that it is the length of the longest path in the tree.

The height of the tree in Figure 2 is 2 since the longest paths are of length 2.

**Definition 12.** The ***depth*** of a node is the length of the path from the root to the node. The root has depth 0.

**Exercise.** Depth can also be defined recursively. Write that definition.

## 3 Binary Trees

### 3.1 N-ary Trees

An ***n-ary tree*** is not the same thing as a general tree. The distinction is that, in an $n$-ary tree the degree of any node is bounded by $n$, i.e., it can never be greater than $n$. The formal definition is

**Definition 13.** An ***n-ary tree*** is a set S of nodes that is either empty, or it has a distinguished node called the ***root*** and *n,* possibly empty n-ary subtrees of the root.

A general tree cannot be empty. It always has at least one node, but it might not have any subtrees. In contrast, an $n$-ary tree may be empty, but it always has all of its subtrees, which might be empty. So a general tree with one subtree has one subtree, but a 3-ary tree with one non-empty subtree technically has 3 subtrees, two of which are empty.

We will be interested in just the special case of $n = 2$. When $n = 2$ the tree is called a binary tree.

## 3.2 Binary Trees

A binary tree is not a special kind of tree. General trees do not distinguish among their children, but binary trees do. To be precise,

**Definition 14.** A ***binary tree*** is either empty, or it has a distinguished node called the ***root*** and a ***left*** and ***right*** binary sub-tree.

Notice that in a binary tree, the subtrees are ordered; there is a left subtree and a right subtree. Since binary trees can be empty, either or both of these may be empty. In a general tree there is no ordering of the sub-trees. The root of the left subtree, if it not empty, is called the ***left child*** of the root of the tree, and the root of the right subtree, if it is not empty, is called the ***right child***.

The most important applications of binary trees are in compiler design and in search structures. One use, for example, in compilers is as a representation of algebraic expressions, regardless of whether they are written in prefix, postfix, or infix. In such a tree, the operator is the root and its left operand is its left subtree and its right operand is its right subtree. This applies recursively to the subtrees. The leaf nods of the tree are simple operands.

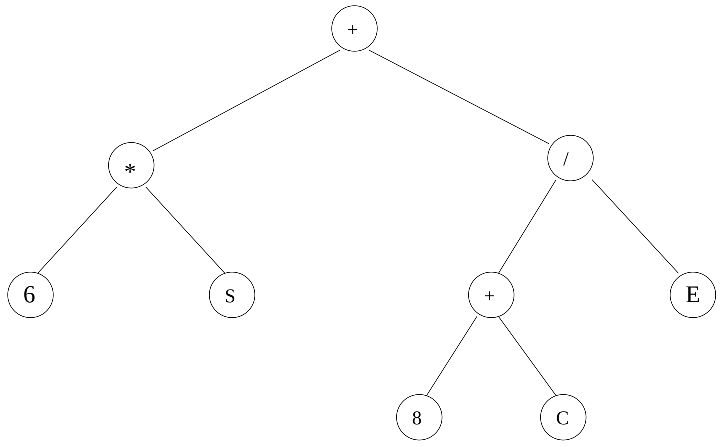


Figure 4: An expression tree representing $6 * S + (8 + C)/E$

The binary tree in 4 is an example of such a tree. It is an unambiguous representation of an algebraic expression. The compiler can use it to construct machine code for the expression.

The other important application of binary trees is as search trees. Informally, a search tree is a tree that is designed to make finding items in the tree "fast". By "fast" we mean better than O(n) on average. Later we will see how this is done.

### 3.2.1 Binary Tree Properties

There are a few interesting questions that we can ask about binary trees, mostly related to the number of nodes they can contain at various heights, and what heights they might be if we know how many nodes they have.

To start, we give a name to a particular shape of binary tree, the one that has as many nodes as it possibly can have for its height.

**Definition 15.** A ***full binary tree*** is a tree that, for its height, has the maximum possible number of nodes.

This means that every node that is not a leaf node has two children, and that all leaf nodes are on the same level in the tree. If there were an internal node with just one child, we could add another child without making the tree taller and this would be a tree with more nodes than the maximum, which is impossible. If not all leaf nodes were at the same level, then there would be one either higher or lower than the remaining leaf nodes. If it were lower, we could add children to it without increasing the height, making a tree with more nodes, again impossible. If it were at a higher level, then we could pick any leaf node at a lower level and add a child to it without increasing the height of the tree and this tree would have more nodes, again impossible. Figure 5 depicts a full binary tree of height 3.

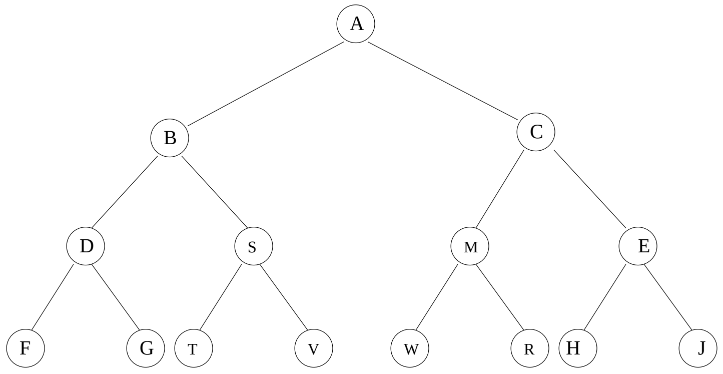


Figure 5: Full binary tree

How many nodes are at depth $d$ in a full binary tree of height $h$, for $d \leq h$? Let $f(d)$ denote the number of nodes at depth $d$ in a full binary tree. We claim that $f(d) = 2^d$.

*Proof.* Assume $h \geq 0$. Let $d = 0$. There is a single root node, so $f(0) = 1 = 2^0$. Assume $h > 0$, and that for $d < h$, the hypothesis is true. Then at depth $d$, there are $2^d$ nodes. Since each of these nodes has exactly 2 children, there are $2 \cdot 2^d = 2^{d+1}$nodes at depth $d+1$. If $d = h$ then there are no nodes at depth $d+1$. Thus, for all $d \leq h$, $f(d) = 2^d$. □

It follows from this that the number of leaf nodes in a full binary tree of height $h$ is $2^h$.

A full binary tree is a good thing. Why? Because all algorithms that do things to binary trees start at the root and have to visit the other nodes by traveling paths from the root. If the tree is packed densely, it means the average path length is smaller than if the same number of nodes were not in a full tree. How many nodes are in a full binary tree of height $h$? Since each level $d$ has $2^d$nodes, there are $1 + 2^1 + 2^2 + 2^3 + \cdots + 2^h = (2^{h+1} - 1)/(2 - 1) = 2^{h+1} - 1$ nodes in a full binary tree. This is important enough to state as a theorem:

**Theorem 16.** *A full binary tree of height $h$ has $2^{h+1} - 1$ nodes.*

Thus, the number of nodes in full binary trees of ever increasing heights are 1, 3, 7, 15, 31, 63, and so on, corresponding to trees of heights 0, 1, 2, 3, 4, and 5 respectively.

Certain types of binary trees that are "nearly" full are also given a name. One such type of binary tree is called a ***complete binary tree***.

**Definition 17.** A binary tree of height $h$ is ***complete*** if the subtree of height $h - 1$ rooted at the root of the tree is a full binary tree, and if a node at depth $h - 1$ has any children, then all nodes to the left of that node have two children, and if it has only one child, that child is a left child.

Figure 6 depicts a complete binary tree of height 3. Notice that it is like a full tree with part of its bottom level removed, from the right towards the left. This is how a complete tree must appear. Complete binary trees can be used to implement a special data structure called a ***priority queue***.

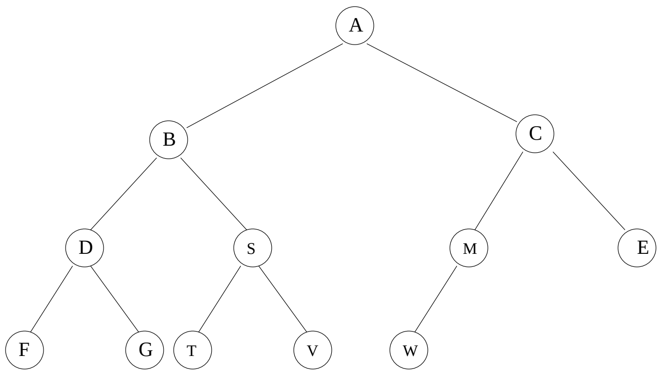


Figure 6: Complete binary tree of height 3.

**Definition 18.** A ***degenerate binary tree*** is a binary tree all of whose nodes except the leaf node has exactly one child.

The tree in Figure 7 is a degenerate tree of height 3. Notice that if we "straightened out" the edges, it would look just like a list. This is what characterizes degenerate trees – they are essentially lists. A degenerate tree has exactly one edge for every node except the single leaf node, so a degenerate tree with $n$ nodes has height$n-1$.

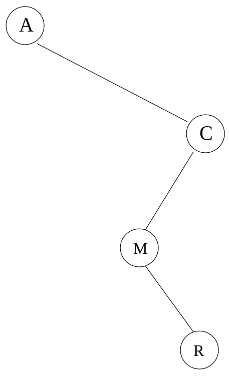


Figure 7: Degenerate binary tree of height 3.

Now we can answer some questions about binary trees.

*What is the maximum height of a binary tree with n nodes?*

The tallest binary tree with $n$ nodes must be degenerate. If this were false, it would mean there is at least one node with two children. We could remove a child and attach it to a node in the bottom level of the tree, making the tree taller, which contradicts the assumption that this was the tallest tree possible with $n$ nodes. The height of a degenerate tree with $n$ nodes is $n-1$, so we have proved:

**Theorem 19.** *The maximum height of a binary tree with n nodes is n-1.*

*What is the minimum height of a binary tree with n nodes?*

Before we derive the answer mathematically, first consider an example. Suppose n=24. The tallest full binary tree with less than or equal to 24 nodes is the one that has 15 nodes and is of height 3. Because we cannot add another node to this tree without making it taller, it is clear that the minimum height of a binary tree with 24 nodes must be greater than 3. Now consider the shortest binary tree that has at least 24 nodes. The tree with 31 nodes is that tree, and its height is 4. Can a binary tree with 24 nodes have height

4? It certainly can. We just have to remove 7 nodes from the full tree of height 4 to create such a tree. We can use this argument more generally.

Let $h$ be the smallest integer such that $n \leq 2^{h+1} - 1$. This implies that $n > 2^h - 1$, because if it were false, then $n \leq 2^h - 1$ would be true and therefore $h - 1$ would be an integer smaller than $h$ for which $n \leq 2^h - 1 = 2^{(h-1)+1} - 1$. This $h$ is therefore the unique integer for which $2^h - 1 < n \leq 2^{h+1} - 1$. Stated in terms of binary trees, $h$ is the height of the smallest full binary tree that has at least $n$ nodes, and $h - 1$ is the height of the tallest full binary tree that has strictly less than $n$ nodes. This implies that a tree with n nodes must be strictly taller than h-1, but may be of height $h$, because a binary tree of height $h$ can have up to $2^{h+1} - 1$ nodes.

It follows from the preceding argument that the minimum height of a binary tree with $n$ nodes is the smallest integer $h$ for which $n \leq 2^{h+1} - 1$. We can determine h as a function of n as follows:

$$\begin{array}{rcc} 2^h - 1 & < n & \leq 2^{h+1} - 1 \\ \iff 2^h & < n+1 & \leq 2^{h+1} \\ \iff h & < \log_2(n+1) & \leq h+1 \end{array}$$

If $\log_2(n+1) = h+1$ then $h+1 = \lceil \log_2(n+1) \rceil$, and $h = \lceil \log_2(n+1) \rceil - 1$. If $\log_2(n+1) < h+1$, then $h = \lceil \log_2(n+1) \rceil - 1$. This proves

**Theorem 20.** *The minimum height of a binary tree with n nodes is* $\lceil \log_2(n+1) \rceil - 1$.

## 4 The Binary Tree ADT

There are several operations that a binary tree class should support. There are the usual suspects, such as creating empty trees, destroying trees, and perhaps getting properties such as their height or the number of nodes they contain. In addition, we need methods of inserting new data, removing data, and searching to see if a particular data item is in the tree. These are the same types of methods that lists supported, and we will include them in our interface.

Unlike the abstract data types based on lists, binary trees also need to provide traversals – methods of visiting every node in the tree, perhaps to print out their contents, to replicate a tree, or to modify the data in all nodes. In general, a binary tree should allow the client to supply a processing function to a traversal so that as the traversal visits each node it can apply that function to the node. Languages like C and C++ allow function parameters, so this is possible, and we will see how to do this later.

For now we start by exploring the different ways to traverse a binary tree, and only after that do we flesh out the interface for a binary tree abstract data type.

### 4.1 Binary Tree Traversals

There have to be systematic, i.e., algorithmic, methods of processing each node in a binary tree. The type of processing is irrelevant to the algorithm that traverses the tree. It might be retrieving a value or modifying a value in some specific way. We can assume that there is some specific function, named `visit()`, that is applied to each node as it is visited by the traversal algorithm.

There are three important algorithms for traversing a binary tree: ***in-order***, ***pre-order***, and ***post-order***. They are easy to describe recursively because binary trees are essentially recursively defined structures.

All traversals of a binary tree must visit the root and all nodes in its left and right subtrees. If these traversals are described recursively, then there are three different steps that can take place:

1. visit the root
2. visit the left subtree (in the same order as the tree rooted at the root is visited)
3. visit the right subtree (in the same order as the tree rooted at the root is visited)

Because these three actions are independent, there are six different permutations of them:

```
1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1
```

In three of these the right subtree is visited before the left subtree, and the traversals is otherwise analogous to the three in which the left subtree is visited before the right subtree. Because visiting the left subtree before the right subtree is usually more useful, the three orderings in which the right is visited before the left are not usually discussed. The remaining ones, with left preceding right, are the ones customarily used. They are:

```
1, 2, 3
2, 1, 3
2, 3, 1
```

and these three permutations correspond to the following:

#### 4.1.1 Pr-Order:

- visit the root
- visit the left subtree
- visit the right subtree

#### 4.1.2 In-Order:

- visit the left subtree
- visit the root
- visit the right subtree

#### 4.1.3 Post-Order:

- visit the left subtree
- visit the right subtree
- visit the root

In each of these it is implicit that the visits to the subtrees are carried out recursively using the same permutation of actions as at the top level. For example, an in-order traversal of the tree in Figure 4 would first go left from the root to the node labeled "*". It does not apply the `visit()` function to this node yet, but would instead apply recursively and go left again to the node labeled "6". Since this node has no left child, the attempt to go left returns back to the 6, visits 6, attempts to visit the right child of 6, which is empty, and therefore backs up to the parent of 6, which is *. It now visits * and then goes to the right child of *, which is S. After processing S, it backs up to * and then backs up to the root node +, which it now visits. The same logic happens on the right hand side of the tree. We will work through some complete examples shortly.

If we assume for the moment that a tree node has the representation

```
struct tree_node
{
    item_type   item;
    tree_node   *left;
    tree_node   *right;
};
```

and that a `visit()` function is defined using the following `typedef` statement[1]

```
typedef void (*visit_function)(item_type data);
```

then the three traversals can be defined precisely by the following functions

```
void in_order ( binary_tree *t, visit_function visit )
{
    if ( t != NULL ) {
        in_order( t->left,  visit);
        visit(t->item);
        in_order( t->right, visit);
    }
}

void pre_order ( binary_tree *t, visit_function visit )
{
    if ( t != NULL ) {
        visit(t->item);
        pre_order( t->left,  visit);
        pre_order( t->right, visit);
    }
}
void post_order ( binary_tree *t, visit_function visit )
{
    if ( t != NULL ) {
        post_order( t->left,  visit);
        post_order( t->right, visit);
        visit(t->item);
    }
}
```

[1]If this `typedef` is new to you, the way to read it is that the name of the function is what is being defined. The `typedef` tells the compiler that the symbol "`visit_function`" is the name of a function type, and that this function type has a signature consisting of a `void` return type and a single parameter of type `item_type`. Any place where an object is declared to be of type `visit_function`, the compiler will expect that object to be used as a function with this signature.

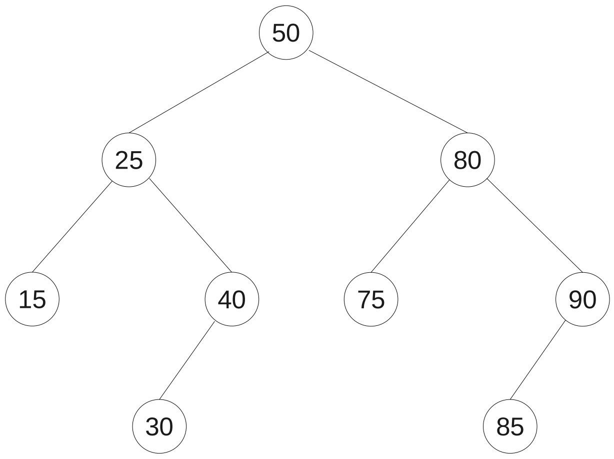


Figure 8: A binary tree

### 4.1.4 Examples

Given the binary tree in Figure 8, the pre-order, in-order, and post-order traversals are as follows. Work through them carefully to understand how they are applied.

**Pre-order:** 50, 25, 15, 40, 30, 80, 75, 90, 85

**In-order:** 15, 25, 30, 40, 50, 75, 80, 85, 90

**Post-order:** 15, 30, 40, 25, 75, 85, 90, 80, 50

**Observations**

- Notice that among them, the in-order traversal visits the nodes in such a way that they are in sorted order. This is not a coincidence; it it because the tree in Figure 8 is a ***binary search tree***. In Section 6, we will see that an in-order traversal of a binary search tree always visits the nodes in ascending order.
- Also note that each of these traversals visits each node exactly once and that therefore, they are $O(n)$ algorithms, where $n$ is the number of nodes in the tree.
- The function that is called to process each node, the `visit()` function, must use only the exposed operations of the binary tree. If the traversals are member functions of a binary tree class, then they provide a means for client code to traverse the tree, but not the means to access private data.

## 4.2 The Binary Tree ADT Interface

We can now return to designing an interface. The following is a list of operations that ought to be exposed.

1. Create a new, empty binary tree.
2. Create a one-node binary tree. This is a convenient function to give to clients.
3. Destroy a binary tree, i.e., deallocate all of its resources.

4. Insert data into the root of the tree.
5. Get data from the root of the tree.
6. Append an existing binary tree as the left (or right) subtree of the root.
7. Get the left (or right) subtree of the root.
8. Detach the left (or right) subtree and save as a new binary tree.
9. Copy one binary tree to another.
10. Return the height of the tree.
11. Return the number of nodes in the tree.
12. Traverse the tree in-order.
13. Traverse the tree pre-order.
14. Traverse the tree post-order.

There may be other useful methods, but they can be derived from these. For example, it may be useful to provide a method that constructs a binary tree from the data for a root and two existing binary trees, one left and one right. That can be derived using methods 2, 4, and 6 above. The above list does not contain a method to attach a single node as the left or right child of the root. That operation can be carried out by creating a one node binary tree and using method 6. A formal ADT for a binary tree is specified as follows.

```
create_binarytree([in] item_type) throw tree_exception
// Create a one-node tree whose root contains the specified item.
// Throw an exception if this fails.
// This is  a constructor.

create_binarytree([in] item_type new_item,
                  [inout] binary_tree left_subtree,
                  [inout] binary_tree right_subtree) throw tree_exception
// Create a binary whose root contains the new_item and whose
// left and right subtrees are left_subtree and right_subtree respectively.
// Throw an exception if this fails.
// On return release the references to the left and right subtrees.
// This is a constructor.

destroy_binarytree()
// Destroy a binary tree.
// This is a destructor. It deallocates all memory belonging to the tree.

int height() const
// This returns the height of the tree.
int size() const
// This returns the number of nodes in the tree.

void get_root_item([out] item_type root_item) const throw tree_exception
// Returns the item stored in the root of the tree.
// Throws an exception if the tree is empty.

void set_root-item([in] item_type new_item )  throw tree_exception
// Stores  new_item into the root of the tree, replacing any value if it exists.
// Throws an exception if the tree is empty.

void attach_left_subtree( [inout] binary_tree left_tree ) throw tree_exception
// If the tree is not empty and the root does not have a left child,
// the left_tree is attached as the left subtree of the root, and the
// reference to it is removed.
// Otherwise an exception is thrown.

void attach_right_subtree( [inout] binary_tree right_tree ) throw tree_exception
// If the tree is not empty and the root does not have a right child,
// the right_tree is attached as the right subtree of the root, and the
// reference to it is removed.
// Otherwise an exception is thrown.

void detach_left_subtree( [out] binary_tree left_tree ) throw tree_exception
// If the tree is not empty, the left subtree is detached from the root
// and a reference to it is returned in the left_tree parameter. If the
// left subtree is empty a NULL reference is returned.
// Otherwise an exception is thrown.

void detach_right_subtree( [out] binary_tree right_tree ) throw tree_exception
// If the tree is not empty, the right subtree is detached from the root
// and a reference to it is returned in the right_tree parameter. If the
// right subtree is empty a NULL reference is returned.
// Otherwise an exception is thrown.

binary_tree get_left_subtree() const throw tree_exception
// If the tree is not empty, a copy of the left subtree is returned.
// Otherwise an exception is thrown.

binary_tree get_right_subtree() const throw tree_exception
// If the tree is not empty, a copy of the right subtree is returned.
// Otherwise an exception is thrown.

binary_tree copy() const
// Return a copy of the tree.

void pre_order_tree ([in] visit_function visit);
// Traverse the tree using the pre-order algorithm, applying the function
// visit() to each node as it is visited.

void in_order_tree  ([in] visit_function visit);
// Traverse the tree using the in-order algorithm, applying the function
// visit() to each node as it is visited.

void post_order_tree([in] visit_function visit);
// Traverse the tree using the post-order algorithm, applying the function
// visit() to each node as it is visited.
```

## 5 Implementation of a Binary Tree

Although it is possible to implement a binary tree with an array, we will focus on a pointer-based implementation. We begin with the basic building block: the tree node.

Modifying the preceding definition of a tree node that was based on a C struct, adding a private constructor (which will be justified shortly), and declaring the binary tree class to be a friend class:

```
class tree_node
{
private:
    tree_node ()   {};
    // Non-default constructor:
    tree_node (const  item_type  & node_item,
               tree_node *left_tree  = NULL,
               tree_node *right_tree = NULL):
                    item(node_item),
                    left (left_tree),
                    right (right_tree) {}
    item_type   item;
    tree_node   *left;
    tree_node   *right;
    friend class binary_tree;
};
```

The `tree_node` class is not exposed to any client software. It has no public members, not even a constructor. Without a public constructor, no code can create an instance of it, unless that code belongs to a friend of this class. By making the `binary_tree` class a friend of the `tree_node` class, we allow the `binary_tree` class to create instances of tree nodes, which is how the tree can grow.

## 5.1 Binary Tree Class Interface

The `binary_tree` class interface is given below. The protected methods are not declared as private, so that a class that is publicly derived from derived from this class can access those methods. In addition, the pre- and post-conditions for each method are omitted to save space, because it is essentially a refinement of the ADT described earlier.

```
// this declares a function type that can be applied to the item stored in a tree node
typedef void (*visit_function)(item_type& );

class binary_tree
{
public:
    // constructors:

    binary_tree  ();                             // default
    binary_tree  ( const item_type & root_item); // param-1 constructor given item
    binary_tree  ( const item_type & root_item,
                   binary_tree    & left_tree,
                   binary_tree    & right_tree); // param-3 constructor
    binary_tree  ( const binary_tree    & tree); // copy constructor

    // destructor:
     ~binary_tree();

    // binary tree operations:
    int         size()       const;
    int          height()     const;
    item_type    get_root()   const                           throw(tree_exception);
    void         set_root(const item_type & new_item)         throw(tree_exception);

    void         attach_left_subtree (binary_tree & left_tree)  throw(tree_exception);
    void         attach_right_subtree(binary_tree & right_tree) throw(tree_exception);

    void         detach_right_subtree(binary_tree & left_tree)  throw(tree_exception);
    void         detach_right_subtree(binary_tree & right_tree) throw(tree_exception);

    binary_tree get_left_subtree () const;
    binary_tree get_right_subtree() const;
    // traversals: (just wrappers)
    void         pre_order  (visit_function visit);
    void         in_order   (visit_function visit);
    void         post_order (visit_function visit);

protected:
    binary_tree (tree_node *nodePtr);  // private constructor

    // A function to copy the tree to a new tree
    void          copy_tree  (tree_node *tree_ptr,
                              tree_node *& new_ptr) const;
    // private function called by public destructor
    void          destroy    (tree_node *& tree_ptr);

    // returns pointer to root
    tree_node   *get_root_ptr() const;

    // set root pointer to given value
    void         set_root_ptr (tree_node *new_root);

    // Given a pointer to a node, retrieve pointers to its children
    void         get_children (tree_node *node_ptr,
                              tree_node *& left_child,
                              tree_node *& right_child) const;

    // Given a pointer to a node, set pointers to its children
    void         set_children (tree_node *nodePtr,
                              tree_node *left_child,
                              tree_node *right_child);
    // Helpers for size and height
    int          get_height( tree_node *tree) const;
    int          get_size( tree_node *tree)   const;

    // The three private traversal functions. These are recursive
    // whereas the public ones are just wrappers that call these.
    // These need access to pointers whereas public cannot have it.
    void         pre_order    (tree_node *treePtr,
                              visit_function visit);

    void         in_order     (tree_node *treePtr,
                              visit_function visit);
```

```
        void        post_order   (tree_node *treePtr,
                                   visit_function visit);
    private:
        // pointer to root of tree
        tree_node *root;
    };
```

## 5.2 Implementation of Methods

The textbook contains the implementations of all of these functions. In general, they are all very simple. They all are either recursive, or perform a simple task such as getting or setting the private members of the object. What makes them a bit more than trivial is that they all handle the various error conditions that can arise by throwing exceptions. The error handling code is the majority of the code. Another bit of complexity has to do with holding and releasing references, which is explained below.

Most tree algorithms are expressed easily using recursion. On the other hand, to do so requires that the parameter of the algorithm is a pointer to the root of the current node, but this is a problem, because one does not want to expose the node pointers to the object's clients . Nodes should be hidden from the object's clients. Put another way, the clients should not know and not care how the data is structured inside the tree; in fact, they should not even know that it is a tree! The binary tree should be a black box with hooks to the methods it makes available to its clients, and no more. Therefore, the sensible solution is to create a "wrapper" method that the client calls that wraps a call to a recursive function. This is exactly how the traversals, the copy function, the destructor, and the functions that get subtrees work. Although I omit the implementations of most of these functions, you should not skip them; you have to make sure you understand how this class implementation works!

What follows demonstrates a typical method that wraps a private (or protected) recursive method; in particular we implement the destructor. First we define a recursive, protected `destroy()` function that uses a post-order traversal to delete all of the nodes of the tree. It has to be post-order because it first has to delete the nodes of the subtrees, and only then can it remove the root of the tree. If we deleted the root first, we could not get to the subtrees, unless we saved the pointers to them. This is inefficient.

```
    binary_tree::destroy (tree_node *& tree_ptr)
    {
        // postorder traversal
        if (tree_ptr  !=  NULL) {
            destroy (tree_ptr->left);  //
            destroy (tree_ptr->right);
            delete tree_ptr;
            tree_ptr = NULL;
        }
    }
```

Having defined this hidden, recursive function, the destructor is simply

```
    binary_tree::~binary_tree()
    {
        destroy(root);
    }
```

As a second example, we implement the copy constructor. First we define the recursive, hidden function that copies a tree:

```
void binary_tree::copy_tree(tree_node *tree_ptr,
                            tree_node *& new_ptr) const
{
    // preorder traversal
    if (tree_ptr != NULL) {
        // copy node
        new_ptr   = new tree_node(tree_ptr->item, NULL, NULL);
        copy_tree  ( tree_ptr->left,  new_ptr->left);
        copy_tree  ( tree_ptr->right, new_ptr->right);
    }
    else
        new_ptr = NULL;  // copy empty tree
}
```

Unlike the `destroy()` function, this uses a pre-order traversal, because it first has to create the root node of the new tree and only then can it create its subtrees. Having defined this function, the public copy constructor is trivial:

```
binary_tree::binary_tree( const binary_tree & tree); // copy constructor
{
    copy_tree(tree);
}
```

Before we look at some more challenging functions, we dispense with the problem of getting the tree's height. Again we wrap the recursive helper function by the public method. The recursive height-computing function is

```
int  binary_tree::get_height( tree_node *tree) const
{
    if ( NULL == tree  )
        return 0;
    else
        return 1 + max (get_height(tree->left), get_height( tree->right));
}
```

In other words, the height of a binary tree is defined recursively; it is one more than the heights of the larger of its two subtrees. The public method is just

```
int binary_tree::height( const binary_tree & tree) const
{
    return get_height(root);
}
```

**Exercise 21.** The size of a tree is also defined recursively. How?

The harder methods are those that attach and detach subtrees. An implementation of a method to attach a left subtree to a root node checks first for whether the root is a `NULL` pointer. If so it throws an exception. If not, it checks whether there exists a left subtree already. If so, it throws a different exception. If not, it sets the left child pointer of the root to point to the root of the argument tree, and then it sets the argument tree pointer to `NULL`, to prevent the client code from being able to manipulate the subtree internally after it has been attached to the tree:

```
void binary_tree::attach_left_subtree (binary_tree & left_tree) throw(tree_exception);
{
    if (0 == size() )
        throw tree_exception("Tree Exception: Empty tree");
    else if (root->left != NULL)
        throw tree_exception ("TreeException: Cannot overwrite left subtree");
    else {
        root->left = left_tree.root; // the pointer, not the node
        left_tree.root = NULL;       // prevent client from accessing tree
   }
}
```

To be clear, the argument is passed by reference. The `left_tree` parameter is not a copy of the tree to be attached but the actual tree. The assignment

```
root->left = left_tree.root
```

makes the `left` pointer in the binary tree on which this is called get a copy of the `root` pointer of the `left_tree` passed to the function. Now `root->left` points to the tree's root node. After the call we want to make sure that the client code cannot access this tree anymore. Setting `left_tree.root` to `NULL` ensures that this tree's nodes can no longer be accessed because `left_tree` was passed by reference.

Detaching the left subtree and providing it as a standalone tree to the caller requires first checking that the tree is not empty, and if not, using the private binary tree constructor (the one that is given a pointer to a root and constructs a tree from it) and assigning the newly created tree to the argument, after which the left child pointer is set to `NULL`, to detach it from the tree:

```
void binary_tree::detach_left_subtree (binary_tree & left_tree)  throw(tree_exception)
{
    if (is_empty())
        throw tree_exception("TreeException: Empty tree");
    else    {
       left_tree = binary_tree(root->left);
       root->left = NULL;
   }
}
```

In both of these functions, what happened was that a reference (i.e., a pointer) to a subtree was transferred either from client to tree or vice versa. The total number of references remained one: either the client lost it and the tree gained it or vice versa. The idea of making sure that no more than a single code entity holds a reference to an object is a way to reduce errors in code.

Providing a copy of a left or right subtree is a different story. In this case, there is no need to remove a reference because the data structure is being replicated. The code for this follows. Notice that the copy function is the private copy function and that its arguments are pointers to tree nodes. The function has to use a constructor to convert the root pointer to a tree in the final return statement.

```
binary_tree binary_tree::get_left_subtree() const
{
    tree_node *subtree;
    if (is_empty())
        return binary_tree();
        else    {
            copy_tree(root->left, subtree);
            return binary_tree(subtree);
        }
    }
```

Lastly, this is a small piece of code to demonstrate how to return pointers in the private code. Notice that in order to pass the value of a pointer back to the caller, the pointer itself must be passed by reference (in C++, or as a double pointer in C).

```
    void get_children (tree_node *node_ptr,
                       tree_node *& left_child,
                       tree_node *& right_child) const
    {
        left_child  = nodePtr->left;
        right_child = nodePtr->right;
    }
```

## 6 Binary Search Trees

Let S be a set of values upon which a total ordering relation, <, is defined. For example, S can be a set of numbers or strings. A ***binary search tree*** T for the ordered set (S,<) is a binary tree with the following properties:

- Each node of $T$ has a value called its *label*. If $p$ and $q$ are nodes, then we write $p < q$ to mean that the label of $p$ is less than the label of $q$.
- For each node $n \in T$, if $p$ is a node in the left subtree of $n$, then $p < n$.
- For each node $n \in T$, if $p$ is a node in the right subtree of $n$, then $n < p$.
- For each element $s \in S$ there exists a node $n \in T$ such that $s = n$.

Binary search trees are binary trees that store elements in such a way that insertions, deletions, and search operations never require more than $O(h)$ operations, where $h$ is the *height* of the tree. Minimally, a binary search tree class should support insertion, deletion, search, a test for emptiness, and a find-minimum operation. A find-minimum is useful because very often one needs to know the smallest value in a set. It would also be useful to support a list-all operation that lists all elements in sorted order. Since a binary search tree is a container, it also needs to provide methods for creating an empty instance, for making a copy of an existing tree, and for destroying instances of trees.

### 6.1 Examples

There can be many different binary search trees for the same data set. The topology depends upon the insertion algorithm, the deletion algorithm, and the order in which the keys are inserted and deleted. The three trees shown in Figure 9 each represent the set $S = \{15, 20, 25, 30, 40, 50, 75, 80\}$. The tree in Figure 9(a) is of height 3, that in Figure 9(b) has height 4, and the one in Figure 9(c) has height 2. You will notice that there cannot be a binary search tree of height less than 2 for this set because it has 7 elements, and a full tree of height 1 has 3 elements, so the shortest tree that can contain 7 elements is the full tree of height 2. Of course, there could be an even worse tree than the ones shown, of height 6. There can be many of these, actually. (How many?)

How many different binary search trees are there for a set of $n$ unique elements? Could it be related to the number of permutations of the set? In other words, does each distinct permutation of the set correspond to a unique binary search tree?

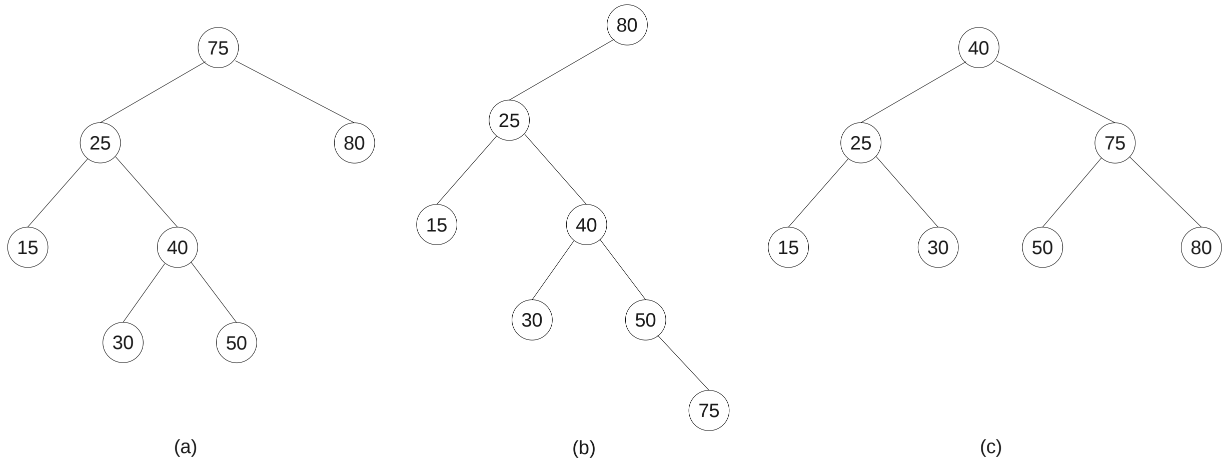


Figure 9: Three different binary search trees for the same data set.

## 6.2 Algorithms

The algorithms we need to consider are the find operation (searching), insertion, deletion, and find_minimum.

### Searching

Consider first of all how to search for an item in a binary search tree. A recursive algorithm is of the form

```
search( current_node, item)
{
    if ( the current node is empty )
        return an indication that the item is not in the tree;
    else {
        if (  item <   current node's item )
            search ( left subtree, item);
        else if (  item >  current node's item )
            search ( right subtree, item);
        else
            the item is in the current node;
    }
}
```

The test for emptiness must be done first of course. If the node is empty, it means that the search has descended the tree to the place where the item should be found if it were in the tree, but that it is not there. For example, if we searched the tree in Figure 8 for the key 77, we would compare 77 to 50 and descend the right subtree, comparing 77 to 80 next. Because $77 < 80$, we descend the left subtree and compare it to 75. Because $77 > 75$, we descend the right subtree of 75, which is empty. At this point we discover that 77 is not in the tree.

This search algorithm descends a level each time it compares the key to a node in the tree. Therefore, in the worst case, it will compare the key to $h$ values (up to $2h$ comparisons), where $h$ is the height of the tree. Since the height of the tree can be proportional to the number of elements in the worst case, this search will take $O(n)$ steps in the worst case.

### Insertion

To insert an element in a tree, we should search for it, and if we do not find it, then the place at which we should have found it is the place at which we want to insert it. The tricky part is that, once we have found it is not there, we have lost a pointer to the node whose child it should be. Careful programming can prevent this. The following pseudo-code includes some actual C++ code so that you can see how this is handled.

```
void insert(tree_node * & current, item_type new_item )
{
    if ( current is empty )
         current = new node containing new_item ;

    else if ( new_item < current->item )
         insert( current->left, new_item );

    else if ( t->element < x )
         insert( current->right, new_item );

    else
         ;
}
```

The `insert` function is passed by reference a pointer to the root of a tree in which to search for the item. If that pointer is `NULL`, then a new node is allocated and the address is stored in that call-by-reference pointer. Because it is passed by reference, in the recursive calls, the pointers `current->left` and `current->right` can contain a pointer to the newly allocated node when one of them is `NULL`. This is how the new node is attached to the tree in the correct place.

The `insert` function is the means by which trees can be created. Start with an empty tree and call insert on it to fill it with the data from the input source. Since the data is always added at the end of an unsuccessful search, the insertions essentially add new levels to the tree. In other words, the first item is place dat the root, the second is either the left or right child of the root, the third might be a grandchild or the child that the previous item did not become. Each successive insertion either fills out an existing level or starts a new one.

### Deletion

Deleting an item is more complex than insertion, because of the possibility that the item to be deleted is not a leaf node. In other words, when the item to be deleted is a leaf node, it is pretty simple – just delete that node. If the item has a single child, there is also a relatively simple task to be performed: delete the node and make the only child the child of the node's parent (i.e., let the parent of the deleted node adopt that node's child). If the item, however, has two children, then it is more complex: find the smallest node in the node's right subtree and copy it into the node, effectively deleting that element. Then delete the node that it came from. That node cannot possibly have two children because if it did, one would have to be smaller than the node, contradicting the assumption that it was the smallest node in the right subtree.

A pseudo-code version of the deletion algorithm is

```
void delete(tree_node * & current, item_type item_to_delete )
{
    if ( current is empty )
         return;  // the item was not found

    else if ( item_to_delete < current->item )
```

```
             delete( current->left, item_to_delete );

        else if ( item_to_delete > current->item )
             delete( current->right, item_to_delete );

        else {
            // item is equal to the item in the node; it is found
            // Check how many children it has
            if ( current->left != NULL && current->right != NULL ) {
                // It has two children. We need to replace it by the
                // smallest item in the right subtree. Assume there
                // is a function, find_min() that returns a pointer
                // to the smallest item in a tree.

                // get the pointer to the smallest item in right subtree
                temp_ptr = findMin( current->right );

                // Copy the item into the current node
                current->item = temp_ptr->item;

                // Recursively call delete to delete the item that was just
                // copied. It is in the right subtree.
                delete( current->right, current->item  );
            }
            else {
                // The current node has at most one child. Copy the value of
                // current temporarily
                old_node = current;

                // If the left child is not empty, then make the left child the
                // child of the parent of current. By assigning to current this
                // achieves that.
                // If the left child is empty, then either the right is empty or it is not
                // In either case we can set current to point to its right child.
                if ( current->left != NULL )
                    current = current->left;
                else
                    current = current->right;

                // Delete the node that current used to point to
                delete old_node;
            }
        }
    }
```

The above deletion algorithm depends on a find_min function. Finding the minimum in a tree is a relatively easy task, since it is always the leftmost node in the tree. By "leftmost", we mean that it is reached by traveling down the left-child edges until the left-child edge is NULL. The node whose left-child pointer is NULL is the minimum in the tree. This can be expressed recursively as follows:

```
    find_min( tree_node *current )
    {
        if ( current == NULL )
            return NULL;
        if ( current->left == NULL )
            return current;

        return find_min( current->left );
    }
```

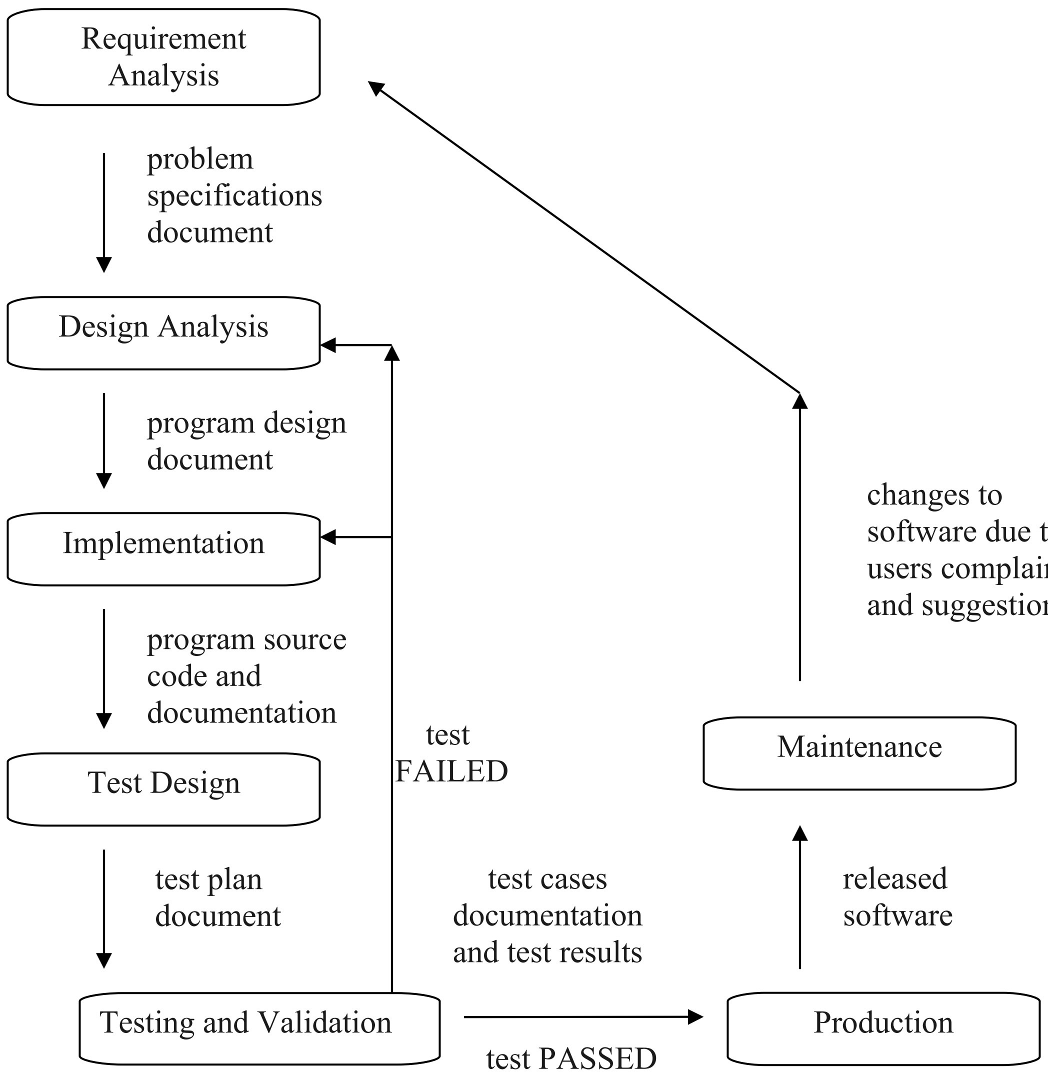
This could be done iteratively as well.

I pointed out in an earlier example that the in-order traversal of a binary search tree always results in visiting the nodes in sorted, ascending order. We can formalize this:

**Theorem.** *An in-order traversal of a binary search tree visits the nodes in ascending sorted order.*

*Proof.* This can be proved by induction on the height of the tree. It is trivially true for a tree of height 0. Suppose it is true of all binary search trees of height at most h. Let T be a BST of height h+1. Then T consists of a root, a left subtree, and a right subtree. Since the left and right subtrees are each at most height h, an in-order traversal of them visits their nodes in sorted order. Since the in-order visits the left subtree first, then the root, and then the right subtree, and since the nodes in the left subtree are all smaller than the root, which is smaller than all nodes in the right subtree, the sequence of nodes visited is in ascending order. □

*"Stop the life cycle—I want to get off!"*


Requirement Analysis
problem specifications document
Design Analysis
program design document
Implementation
program source code and documentation
Test Design
test plan document
Testing and Validation
test FAILED
test cases documentation and test results
test PASSED
Production
released software
Maintenance
changes to software due to users complaints and suggestions