# CHAPTER 1 BASIC CONCEPTS

# 1.1 Overview : System Life Cycle

## a solid foundation in data abstraction and encapsulation, algorithm specification, and performance analysis and measurement

### provide the necessary methodology to design and implement large-scale S/W

## regard programs as systems that contain many complex interacting parts

## a development process : system life cycle requirements, analysis, design, coding, verification

# Requirements

## begin with a set of specifications that define the purpose of the project

## must develop rigorous input and output descriptions

# Analysis

## break the problem down into manageable pieces

## two approaches to analysis: bottom-up and top-down

### 1) the bottom-up approach

#### emphasis on the coding fine points

#### result in many loosely connected, error-ridden segments

### 2) the top-down approach

#### divide the program into manageable segments

#### refined to take into account low-level details

#### preferred approach for developing complex software systems

# Design

## perspectives of the data objects that the program needs and the operations performed on them

### lead to the creation of abstract data types

### require the specification of algorithms and a algorithm design strategies

## specify the information required for each data object

### ignore coding details

### deferring implementation issues

### create a system that could be written in several programming languages

### have time to pick the most efficient implementation

# Refinement and Coding

## choose representations for data objects and write algorithms for each operation

### a data object's representation determines the efficiency of the algorithm

### should write the algorithms that are independent of the data object

## if good design, then can absorb changes easily

# Verification

## develop correctness proofs for the program

## test the program with a variety of input data

## remove errors

# 1.2 Object-Oriented Design

## using the philosophy of divide-and-conquer

### break up a complex software design project into # of simpler subobjects

### tackle simpler subobjects individually

# 1.2.1 Algorithmic Decomposition Versus O-O encapsulation

## algorithmic or functional decomposition

### view software as a process

### decompose the software into modules(steps of the process)

### modules : implemented by procedures or functions

## O-O decomposition

### view software as a set of well-defined objects that model entities in the application domain

### objects interact with each other to form a software system

### address functional decomposition after defining objects

## advantage of O-O decomposition

### reuse of software

### flexible software systems that can evolve as system requirements change

----------------------- skip -------------------------------------------

# 1.2.2 Concepts of O-O Programming

# Def. of object

## an entity that performs computations and has a local state

## viewed as a combination of data and procedural elemnets

# Def. of O-O Programming

## objects are the fundamental building blocks

### encapsulation

## each object is an instance of type

### classification

## inheritance relationships

### is-a (generalization, specialization)

# Def. of O-O language

## support objects

## require objects to belong to a class

## support inheritance

## object-based language : do not support inheritance

# 1.2.3 Evolution of C++

## four generations of higher order programming languages

### 1) 1st generation: FORTRAN

#### ability to evaluate mathematical expressions

### 2) 2nd generation: Pascal and C

#### on effectively expressing algorithms

### 3) 3rd generation: Modula and Ada

#### introduce the concept of abstract data types

##### object-based language

### 4) 4th generation: O-O language (C++, Objective C, Smalltalk)

#### the use of inheritance

## design of C++ : Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980s

### influenced by Simula67 and Algol68

## the reason that C is widely-used in industry

### efficient: # of low-level features, which utilize hardware

### flexible: used to solve problems in most application areas

### available for most computers

## C++: improve on C

### implement data abstraction or inheritance

### other improvements

-------------------------------------------------------------------

# 1.3 Data Abstraction and Encapsulation

# Def. of data encapsulation or information hiding

## the concealing of the implementation details of a data object

# Def. of data abstraction

## the separation between the specification of a data object and its implementation

## abstraction, encapsulation, information hiding result in better quality programs and more efficient programming techniques

## the fundamental data types of C++

### char, int, float, double

### modifiers: short, long, signed, unsigned

### support types derived from the fundamental data types

#### include pointer and reference types

## three mechanisms for grouping data in C++

### the array

### structs

### classes

# Def. of data type

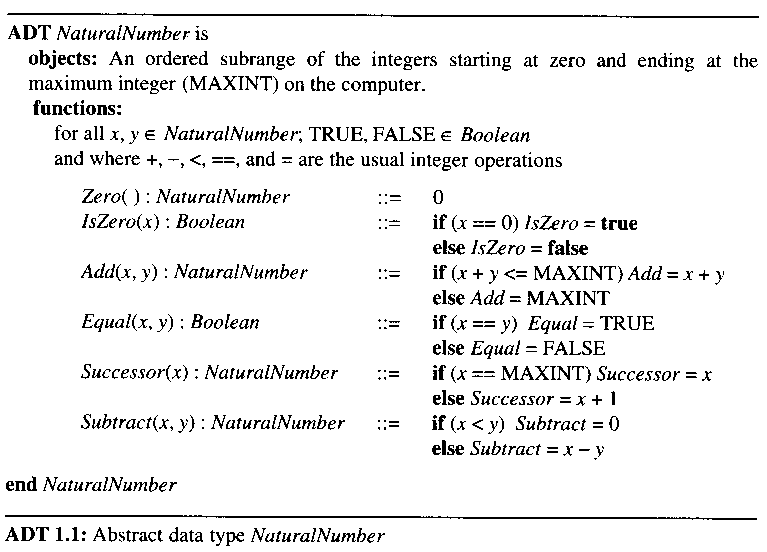
## a collection of objects and a set of operations that act on those objects

# Def. of abstract data type

## separate the specification of the objects and the specification of the operations from the representation of the objects and the implementation of the operations

## emphasize the distinction between specification and implementation

# Example 1.1 [Abstract Data Type NaturalNumber]



## Two main sections in the defintion: the objects and the functions

### define the objects in terms of the integers

### makes no explicit reference to their representation

### makes use of functions that are defined on the set of integers, namely plus, minus, equal, less than ..

## develop well-designed programs by using data abstraction and encapsulation

# Simplification of a software development

## use the concept of data abstraction

### facilitate the decomposition of the complex task into # of simpler subtasks

## a top-down review of the problem

### three data types A, B, C will be used

### some additional code(called glue) to facilitate interactions among the three data types

### provide the specifications of each data type

## scenario 1: a team of 4 programmers

### assign a programmer to each of the three data types

#### implement his data type according to the specifications

### no needs to know how the other programmers implement their portion of code

## scenario 2: a single programmer

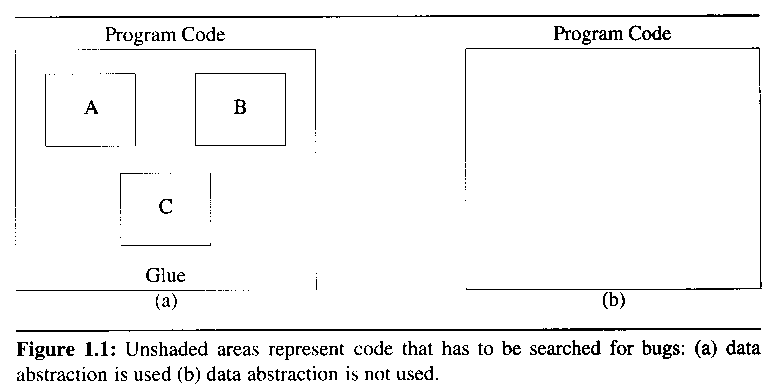
### implement each data type one by one according to the specifications

### help reduce # of things the programmer has to keep in mind at any time

# Testing and Debugging

## can test and debug each data type separately

### simplify testing and debugging



# Reusability

## make it easier to extract the code for a data structure and its operations from a software system and use it in another software system

## data abstraction and encapsulation: lead to implement data structures as distinct entities of a S/W system

# Modifications to the representation of a data type

## by information hiding, the implementation of a data type is invisible to the other program

## a change in the internal implementation of a data type without affecting the rest of the program

--------------------------------- skip -----------------------------

# 1.4 Basics of C++

## "a better C" : the expression is used to describe the C++ language

### C and C++ having common features

### C++ has # of features not associated with data abstraction and inheritance

# 1.4.2 Scope in C++

# File scope

## declarations that are not contained in function scope or class scope

# Function scope

## use labels anywhere within the function definition in which they are declared

### only labels have function scope

### only used as the targets of go to statements

# Local scope

## a name declared in a block : belong to a local scope consisting of that block

# Class scope

## declarations associated with a class definition

## each class represents a distinct class scope

### class의 instance variables

## a variable is visible to a program only from within its scope

### only access a variable defined in a block from within the block

### can access a variable defined at file scope(a global variable) anywhere in the program

## use the scope operator ::

### a local variable reuses a global variable name in a block

#### want to access the global variable

#### use the scope operator to access the global variable

## use extern

### to declare the variable in another file

## use static

### to declare the same global variables in two files

#### means different entities

# 1.4.3 C++ statements and operators

## new and delete

## <<, >> operators

## operator overloading

### allowed to have different functions depending on the types of the operands

#### dynamic binding?

#### polymorphism과의 차이는?

# 1.4.4 data declarations in C++

## referenced types : a feature of C++ that is not a feature of C

### a mechanism to provide an alternate name for an object

int i = 5;

int &j = i;

##### j is a reference type

# 1.4.6 I/O in C++

## cout / cin

## << operator

# 1.4.7 Functions in C++

## two kinds of functions in C++: regular functions and member functions

## member functions : functions that are associated with specific C++ classes

## function 구성:

### a function name, a list of arguments or signature (input), a return type, the body

# 1.4.8 parameter passing in C++

## passed by value

### the default parameter-passing mechanism

### actual arguments are not inadvertently modified

## passed by reference

### need to explicitly declare an argument to be a reference type

### execute faster if the object being passed requires more memory than its address

#### because the overhead of copying the actual argument into the function's local store

## pass *constant references* such as const T& a

### const T&는 타입

### one technique for retaining the advantages of both parameter-passing method

### any attempt to modify a const argument in the function body : a compile-time error

## one exception to the default parameter passing mechanism

### array types are passed by reference

### the array is not copied into the function's local store

### with an array argument a (e.g., f(a))

#### a pointer to the first element of a (i.e., &a[0])

### usually denote arrays by pointers to the appropriate (e.g., f(int \*a))

### explicitly passing the size of the array as a separate parameter of the function

# 1.4.9 Function name overloading in C++

## function overloading

### more than one function with the same name as long as they have different signatures

int max(int, int);

int max(int, int, int);

int max(int \*, int);

int max(float, int);

int max(int, float);

# 1.4.10 Inline functions

## replace any calls to sum by the body of sum

## eliminate the overhead of performing a function call and copying arguments

inline int sum(int a, int b)  
{

return a + b;

}

# 1.4.11 Dynamic memory allocation in C++

int \*ip = new int;

//new: create an object of the desired type

// return a pointer to the data type

. . .

delete ip;

int \*jp = new int[10];

. . .

delete [ ] jp;

#### subscript operator ([ ]) : to inform the compiler that the object being created or deleted is an array

----------------------------------------------------------------------

# 1.5 Algorithm Specification

# 1.5.1 Introduction

# Def. of algorithm

## a finite set of instructions that accomplishes a particular task

## satisfy the following criteria

### input

#### zero or more quantities are supplied

### output

#### produce at least one quantity

### definiteness

#### clear and unambiguous instruction

### finiteness

#### the algorithm terminates after a finite number of steps

### effectiveness

#### must be feasible

##### every instruction must be basic enough to be carried out, by a person using only pencil and paper

## differences between an algorithm and a program, in computational theory

### a program does not have to satisfy finiteness criteria

### an example : operating system

# Example 1.2 [selection sort]

## devise a program that sorts a collection of N integers

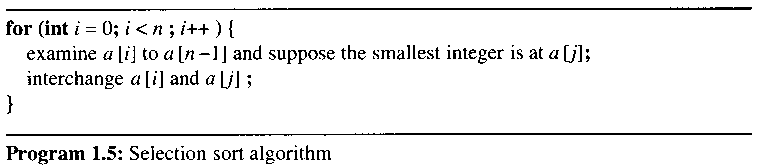
## a simple solution

### "From those integers that are currently unsorted, find the smallest and place it next in the sorted list"

#### is not an algorithm -> not tell where and how the integers are initially stored or where we should place the result

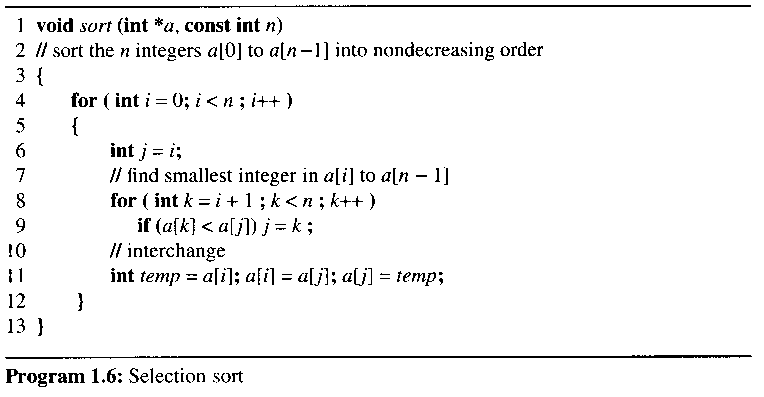
#### not-effective and/or not-definite ?

##### 예: place it next: unclear instruction



#### turn Program 1.5 into a real C++ program

##### called refinement



# Example 1.3 [Binary search]

### n>=1 distinct integers that are already sorted in a[0], . . ., a[n-1] determine if the integer x is present and if so to return j (x = a[J]); otherwise return -1

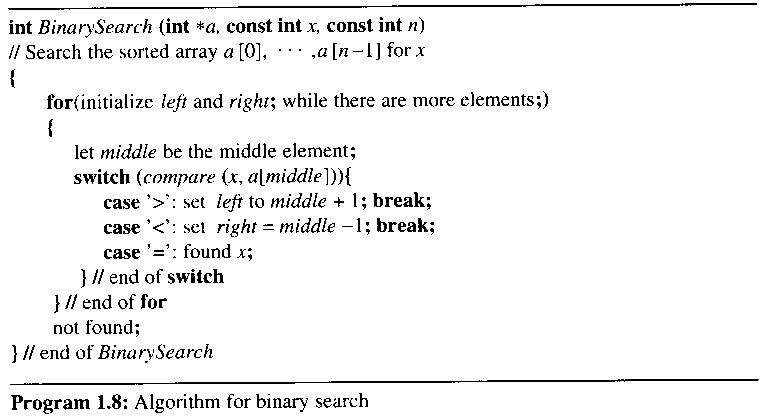
left = 0, right = n-1, middle = (left + right) / 2

compare a[middle] with x

1) if x < a[middle] then set right = middle - 1

2) if x == a[middle} then return middle

3) if x > a[middle] then set left = middle + 1



# 1.5.2 Recursive algorithms

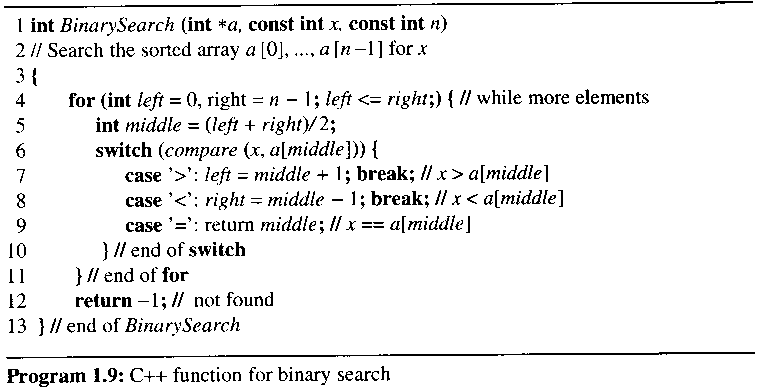
## a recursive function requires a terminating condition

## any program written using assignment, if-else statement, while statement can also be written using assignment, if-else, and recursion

## when is recursion an appropriate mechanism for algorithm exposition?

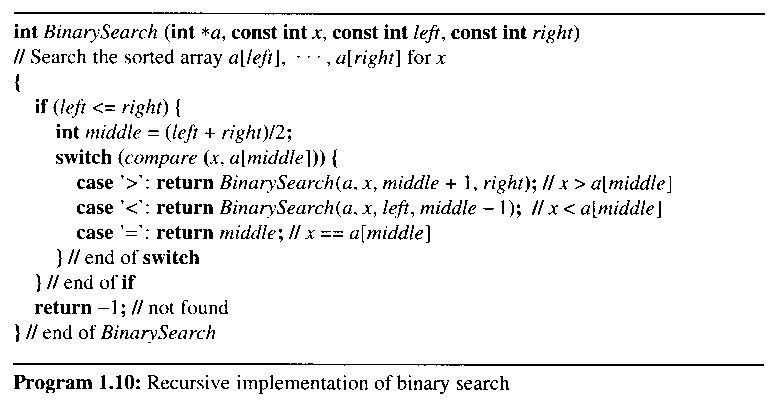
### when the problem itself is recursively defined

### example: factorial computation N! = N \* (N-1)!



# Example 1.4[Recursive binary search]

## pass left and right as parameters in the recursive version



## invoke the recursive function : BinarySearch(a, x, 0, n-1)

# Example 1.5[Permutation generator]

## print all possible permutations of set

## n! different permutations for given n elements

## look at the four elements (a, b, c, d)

a followed by all permutations of (b, c, d)

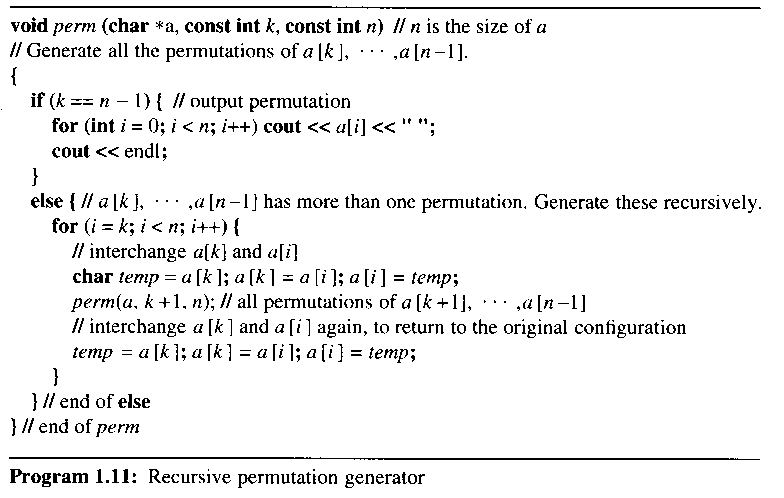
b followed by all permutations of (a, c, d)

c followed by all permutations of (a, b, d)

d followed by all permutations of (a, b, c)

##### "followed by all permutations" : recursion

##### can solve the problem for a set with n elements if we have an algorithm that works on n-1 elements



//I = 0 => permutation a,xxx

//I = 1 => permutation b,xxx

//I = 2 => permutation c,xxx

//I = 3 => permutation d,xxx

Call perm(a, 0, n);

# 1.6 Performance Analysis and Measurement

## many criteria to judge a program

### what to do?

### work correctly according to the original spec.

### documentation that describes how to use and how to works

### are functions created in such a way that they perform logical subfunctions?

### readable code?

## performance criteria for judging programs

# Definition:

## space complexity: the amount of memory a program needs to run to completion

## time complexity: the amount of computer time a program needs to run to completion

## a priori estimates of performance evaluation: performance analysis

## a posteriori estimates of performance evaluation: performance measurement

본 강의 자료의 그림 및 알고리즘 발췌

저자 : HOROWITZ

타이틀 : FUNDAMENTALS OF DATA STRUCTURES IN C++ 2nd Edition (2006)

공저 : SAHNI, MEHTA

출판사 : Silicon Press