Chapter 1. Basic Concepts

Chapter 1. Basic Concepts

1.1 Overview: System Life Cycle

1.2 Pointers and Dynamic Memory Allocation

1.3 Algorithm Specification

1.4 Data Abstraction

1.5 Performance Analysis

1.6 Performance Measurement

1.1 OVERVIEW: System Life Cycle

- Sysem Life Cycle for Moblie phone

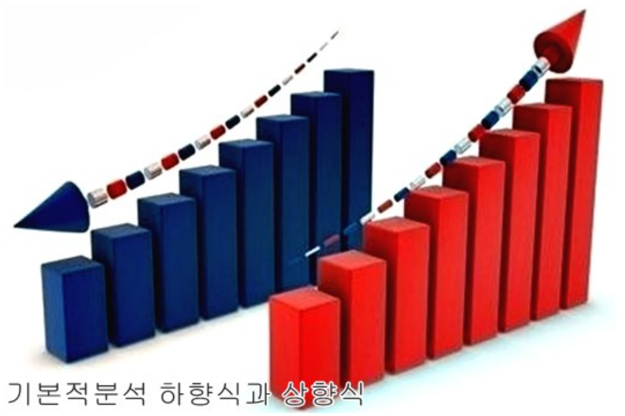
Requirement specification
Analysis
Design
Implementation
Testing

1.1 OVERVIEW: System Life Cycle

- Software development process
  - Requirement specification
    - A set of specifications that define the purpose of the project
    - Describe input and output

1.1 OVERVIEW: System Life Cycle

- Analysis
  - top-down
    - breaking down of a system to gain insight into its compositional sub-systems
  - Bottom-up
    - the piecing together of systems to give rise to more complex systems



1.1 OVERVIEW: System Life Cycle

- Design
  - *Data objects* that the program needs
    - *Abstract data type*
  - *Operations* performed on the data objects
    - *algorithm specifications*
  - Language independent

- Refinement & Coding
  - Choose representations for the data objects
  - Write algorithms for each operation on them
  - The order is crucial
    - ➔ because a data object's representation can determine the efficiency of the algorithms related to it.
- Verification
  - Correctness proofs
  - Testing
  - Error removal

1.2 Pointers and Dynamic Memory Allocation

- Pointers
  - & : the address operator
  - * : the dereferencing (or indirection) operator

- usage
  - int i, *pi;
  - pi = &i;
  - i = 10; *pi = 10;
  - if (pi == NULL) …   /* equals   if (!pi)   */

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

Program 1.1: Allocation and deallocation of memory

A more robust version for *malloc()*

```
if ((pi = (int *) malloc(sizeof(int))) == NULL ||
     (pf = (float *) malloc(sizeof(float))) == NULL)
{fprintf(stderr, "Insufficient memory");
 exit(EXIT_FAILURE);
}
```

or by the equivalent code

```
if (!(pi = malloc(sizeof(int))) ||
     !(pf = malloc(sizeof(float))))
{fprintf(stderr, "Insufficient memory");
 exit(EXIT_FAILURE);
}
```

- A convenient way of using *malloc()*

```
#define MALLOC(p,s) \
    if (!((p) = malloc(s))) {\
        fprintf(stderr, "Insufficient memory"); \
        exit(EXIT_FAILURE);\
    }


 MALLOC(pi, sizeof(int));
 MALLOC(pf, sizeof(float));
```

1.3 Algorithm Specification

- Definitions of algorithm
  - An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task.
  - All algorithms must satisfy the following criteria:

    a. input (≥ 0)

    b. output (> 0)

    c. definiteness

    d. finiteness

    e. effectiveness

How to describe an algorithm?

- Use a natural language like English
- Use graphic representations called flowcharts
- Pseudo Code
- Program language

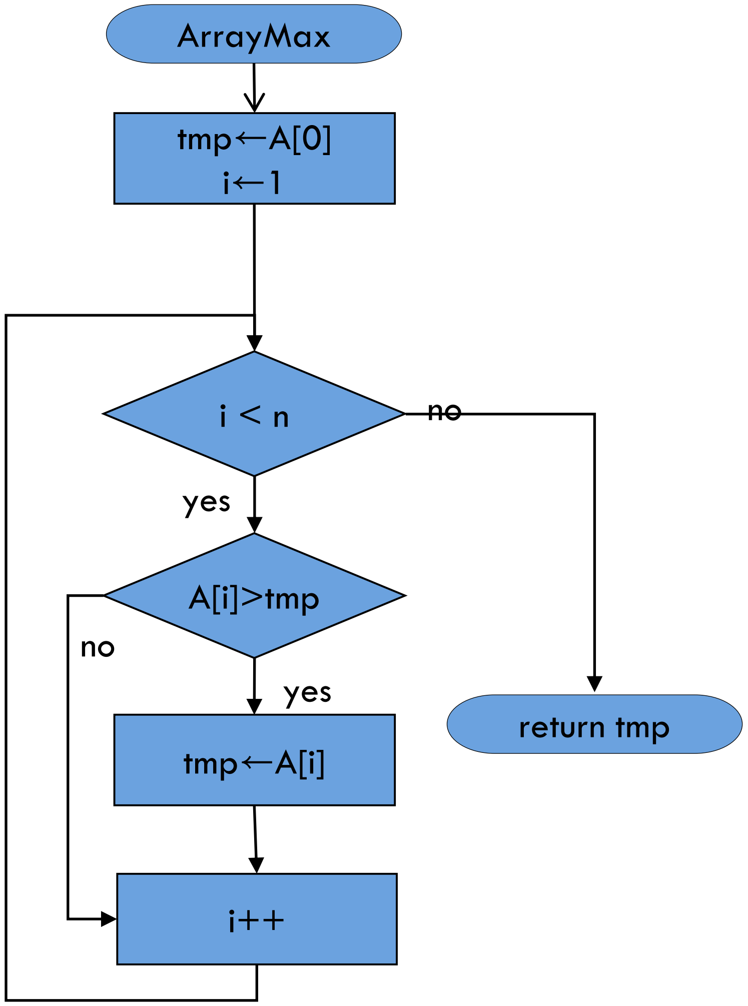
- In this text,
  - Mostly in C
  - Occasionally in a combination of English and C

Find Max in Array

ArrayMax (A, n)

1. Copy the first element of array A to variable tmp
2. The following elements in array A are compared to tmp in turn, and if larger, copied to tmp
3. Returns tmp if all elements in array A are compared

Find Max in Array



```
ArrayMax(A,n)

  tmp ← A[0];
  for i←1 to n-1 do
            if tmp < A[i] then
                      tmp ← A[i];
  return tmp;
```

Translating a Problem into an Algorithm

Example 1.1 [Selection sort]

- Description of a problem
  - Devise a program that sorts a set of $n \geq 1$ integers.

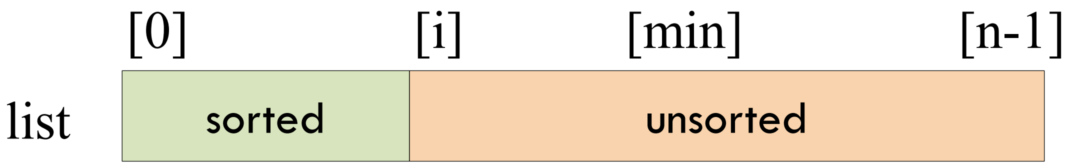
- A simple solution
  - From those integers that are currently unsorted, find the smallest and
  - place it next in the sorted list.

First attempt at deriving a solution

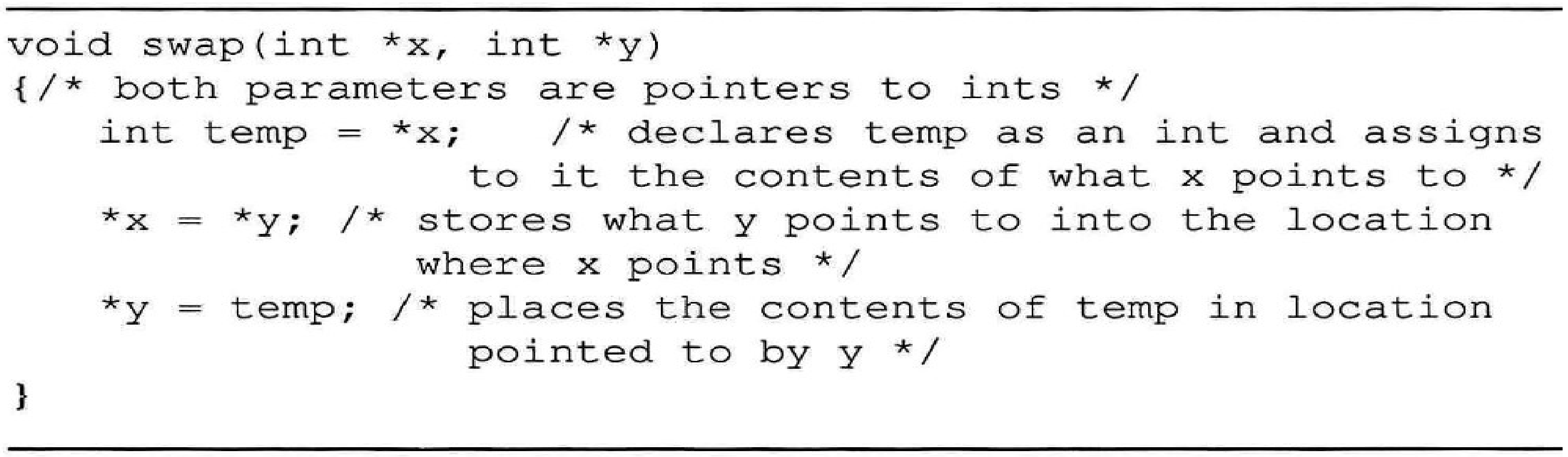
```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.2: Selection sort algorithm



- Interchanging *a* and *b*
  - function



```
void swap(int *x, int *y)
{/* both parameters are pointers to ints */
   int temp = *x;    /* declares temp as an int and assigns
                        to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
                      where x points */
   *y = temp; /* places the contents of temp in location
                        pointed to by y */
}
```

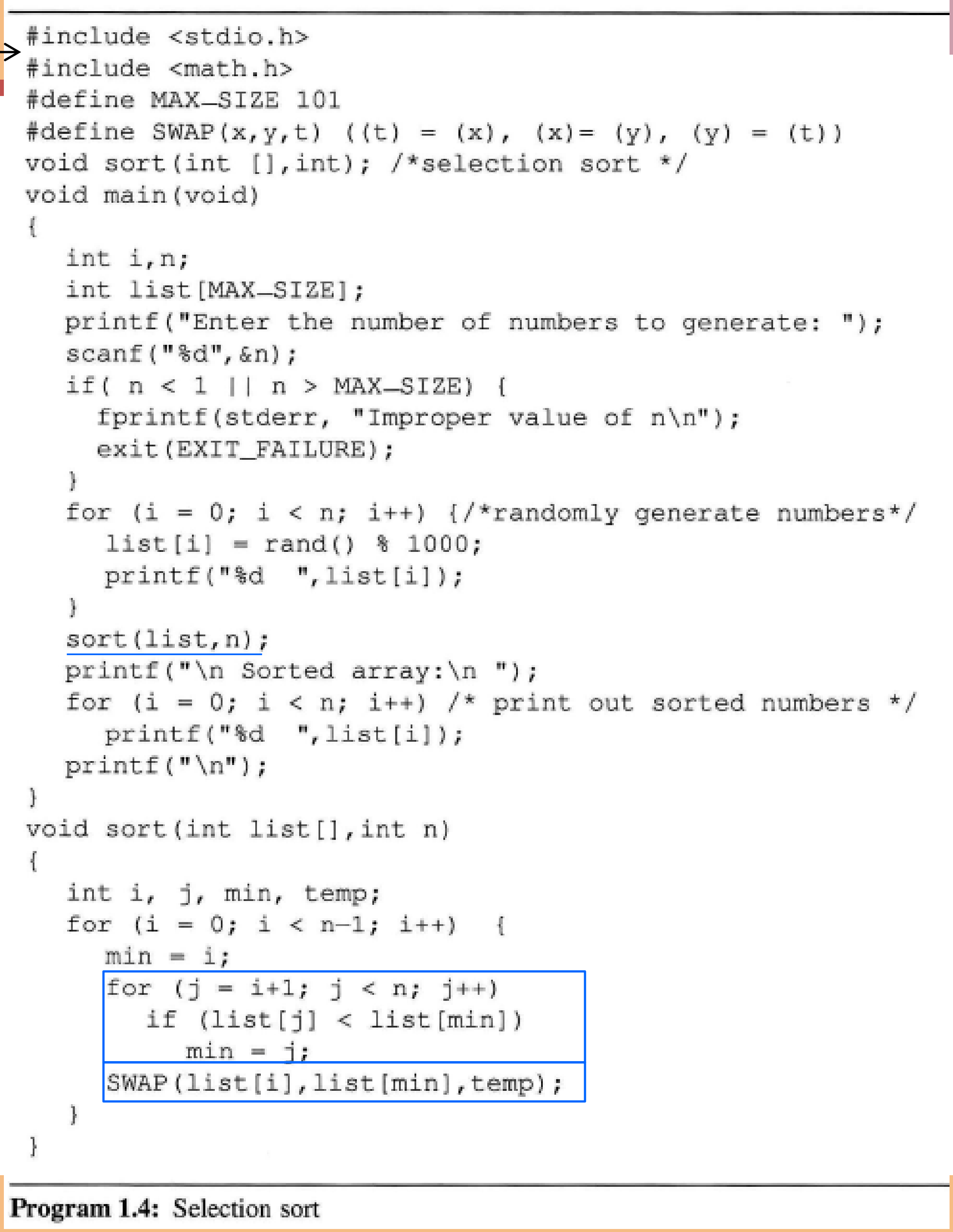
Program 1.3: Swap function

```
swap( &a, &b)
```

  - macro

```
#define SWAP(x, y, t) ((t)=(x), (x)=(y), (y)=(t))
SWAP(a, b, temp) // Macro works with any data type.
```

#include <stdlib.h> →



```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
  int i,n;
  int list[MAX_SIZE];
  printf("Enter the number of numbers to generate: ");
  scanf("%d",&n);
  if( n < 1 || n > MAX_SIZE) {
    fprintf(stderr, "Improper value of n\n");
    exit(EXIT_FAILURE);
  }
  for (i = 0; i < n; i++) {/*randomly generate numbers*/
    list[i] = rand() % 1000;
    printf("%d  ",list[i]);
  }
  sort(list,n);
  printf("\n Sorted array:\n ");
  for (i = 0; i < n; i++) /* print out sorted numbers */
    printf("%d  ",list[i]);
  printf("\n");
}
void sort(int list[],int n)
{
  int i, j, min, temp;
  for (i = 0; i < n-1; i++)  {
    min = i;
    for (j = i+1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    SWAP(list[i],list[min],temp);
  }
}
```

Program 1.4: Selection sort

Example 1.2 [Binary search]

- Problem description
  - Assume that we have $n \geq 1$ distinct integer that are *already sorted* and stored in the array list. That is, $list[0] \leq list[1] \leq \ldots \leq list[n-1]$.
  - We must figure out if an integer *searchnum* is in this list. If it is, we should return an index, *i*, such that $list[i] = searchnum$. If it is not present, we should return -1.

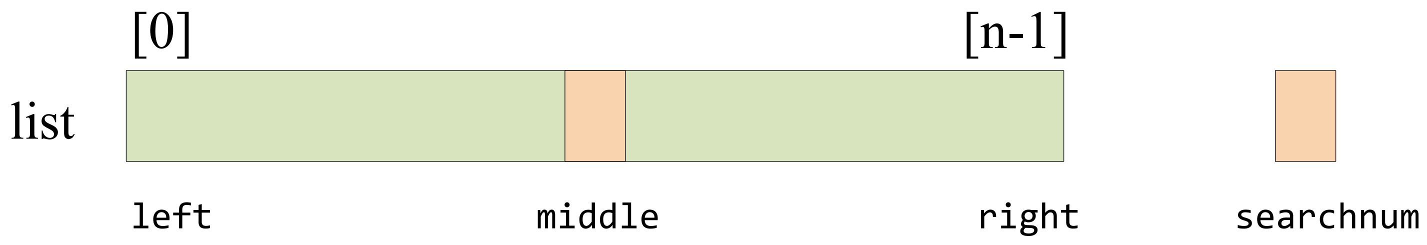
Binary search

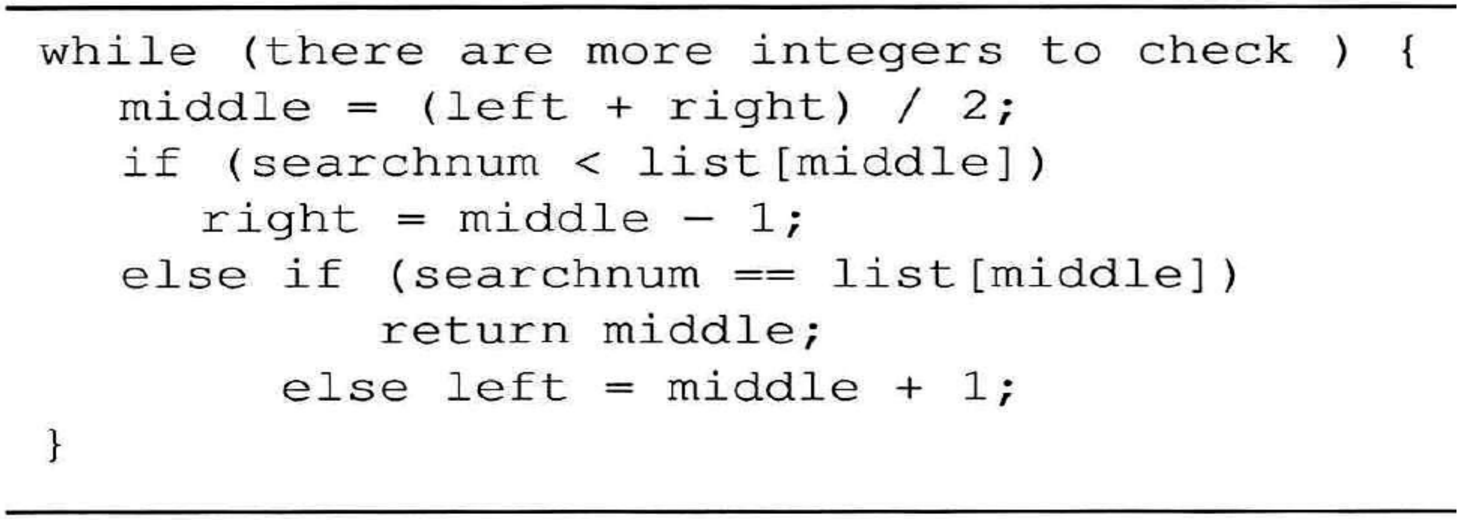
- A solution
  - Compare *list*[*middle*] with *searchnum*

(1) searchnum < list[middle] : set *right* to *middle*-1

(2) searchnum = list[middle] : return *middle*

(3) searchnum > list[middle] : set *left* to *middle*+1





```
while (there are more integers to check ) {
  middle = (left + right) / 2;
  if (searchnum < list[middle])
     right = middle - 1;
  else if (searchnum == list[middle])
          return middle;
       else left = middle + 1;
}
```

Program 1.5: Searching a sorted list

Two subtasks :

(1) determining if there are any integers left to check

(2) comparing *searchnum* to *list*[*middle*]

Comparison

```
int compare(int x, int y)
{/* compare x and y, return -1 for less than, 0 for equal,
    1 for greater */
  if (x < y) return -1;
  else if (x == y) return 0;
       else return 1;
}
```

Program 1.6: Comparison of two integers

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

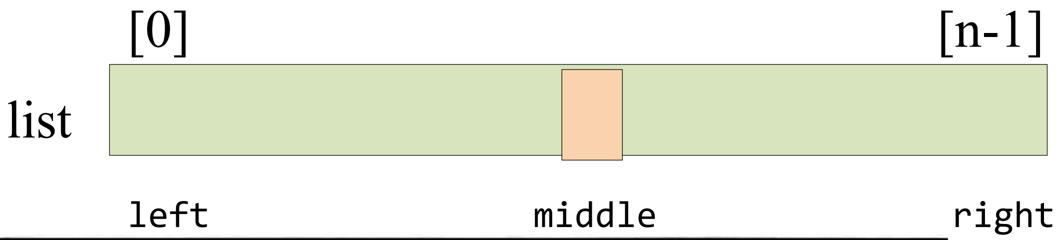
Binary search

```
int binsearch(int list[], int searchnum, int left,
                                          int right)
{/* search list[0] <= list[1] <=  . . .  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

Binary search(Recursive)

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for
    searchnum. Return its position if found. Otherwise
    return -1 */
   int middle;
   if (left <= right) {
       middle = (left + right)/2;
       switch (COMPARE(list[middle], searchnum)) {
            case -1: return
                binsearch(list, searchnum, middle + 1, right);
            case 0 : return middle;
            case 1 : return
                binsearch(list, searchnum, left, middle - 1);
       }
   }
   return -1;
}
```



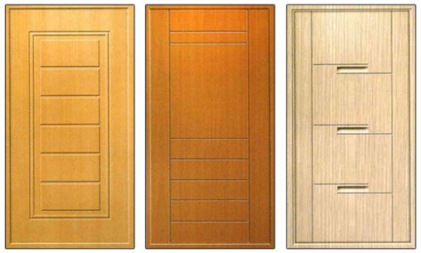
Program 1.8: Recursive implementation of binary search

What is Data Structure?

Assume you make a house

Raw materials

Processed materials



1.4 Data Abstraction

- [Definition] A *data type* is a collection of *objects* and a set of *operations* that act on those objects.

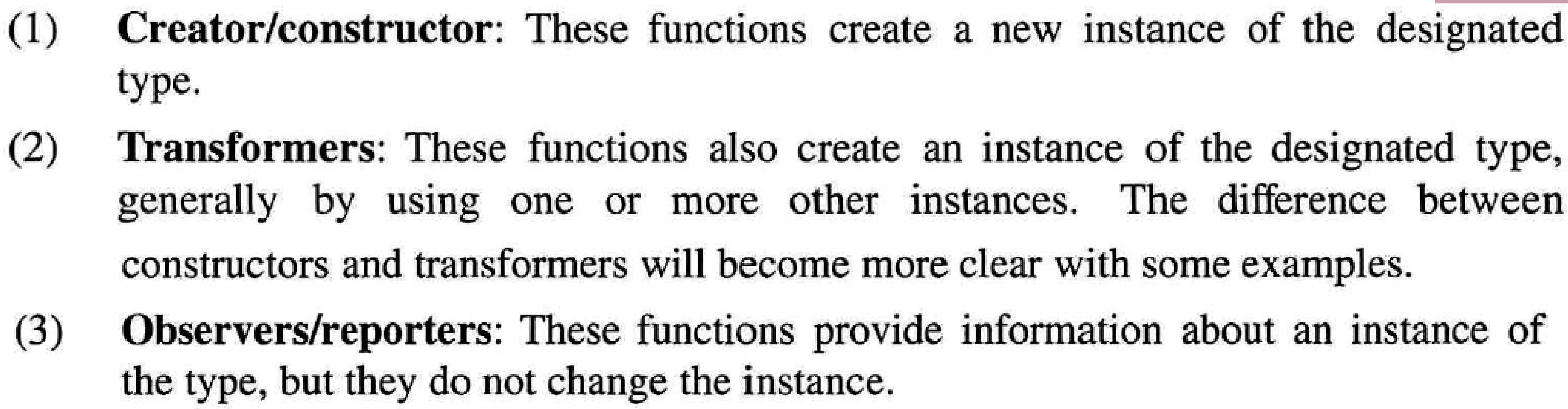
- All *programming languages* provide at least a minimal set of *predefined data types*, plus the ability to construct new, or *user defined data type*.

- Ex) data type int in C
  - Objects
    - { 0, +1, -1, …, INT_MAX, INT_MIN}
    - Representation : 2 byte or 4 bytes of memory
  - Operations
    - Arithmetic operators { +, -, *, /, % }
    - Testing for equality/inequality { ==, >, >= , <, <= }
    - Operation that assigns an integer to a variable {=}

- [Definition] An *abstract data type* is a data type that is organized in such a way that
  - the specification of the objects and the specification of the operations on the objects *is separated from* the representation of the objects and the implementation of the operations.

- An abstract data type is *implementation-independent.*

Categories of operations of an ADT



(1) **Creator/constructor**: These functions create a new instance of the designated type.

(2) **Transformers**: These functions also create an instance of the designated type, generally by using one or more other instances. The difference between constructors and transformers will become more clear with some examples.

(3) **Observers/reporters**: These functions provide information about an instance of the type, but they do not change the instance.

Creator/constructor :

Int I;

Transformers :

4+5

Observers/reporters

Sizeof(i)

A[10] //Array index operation

ADT *NaturalNumber* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all $x, y \in$ *NaturalNumber*; *TRUE*, *FALSE* $\in$ *Boolean*
and where +, −, <, and == are the usual integer operations

```
NaturalNumber Zero( )            ::=  0
Boolean IsZero(x)                ::=  if (x) return FALSE
                                      else return TRUE
Boolean Equal(x, y)              ::=  if (x == y) return TRUE
                                      else return FALSE
NaturalNumber Successor(x)       ::=  if (x == INT_MAX) return x
                                      else return x + 1
NaturalNumber Add(x, y)          ::=  if ((x + y) <= INT_MAX) return x + y
                                      else return INT_MAX
NaturalNumber Subtract(x, y)     ::=  if (x < y) return 0
                                      else return x − y
```

end *NaturalNumber*

ADT 1.1: Abstract data type *NaturalNumber*

- Nicklaus Wirth
  - data structure = data type + storage structure
  - program = data structure + algorithm

- Overview: System Life Cycle
- Pointers and Dynamic Memory Allocation
- Algorithm Specification
- Data Abstraction