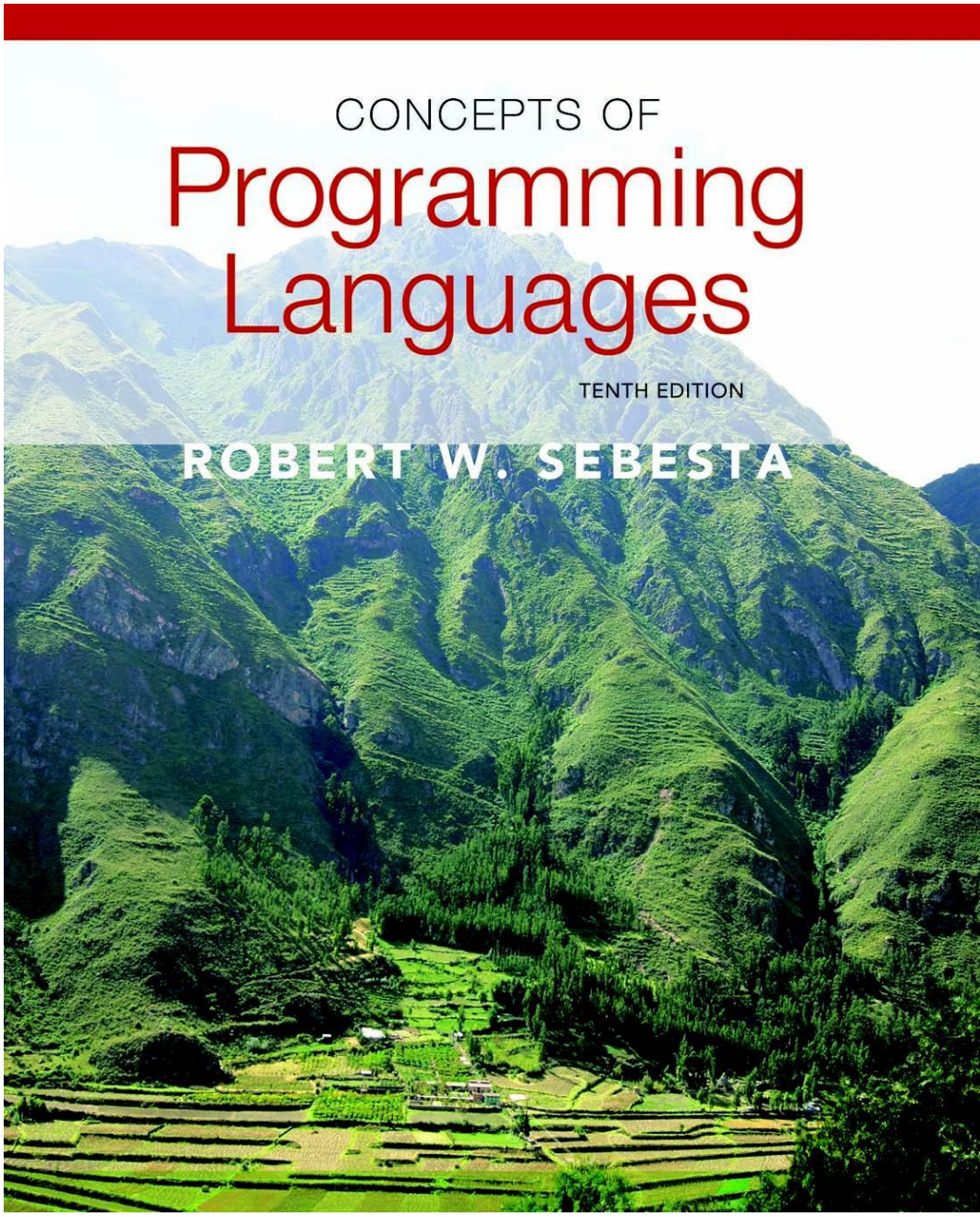
Chapter 6

Data Types

노트에 필기한 내용 가져올것!



Chapter 6 Topics(1st)

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types
- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

Chaper 6 Topics (continued)

- Primitive data types
- Enumeration & subrange type
- Structured data types
  - Arrays, associative arrays, records, tuples, lists, and unions
- Pointers & references

Introduction

- A data type defines a collection of data objects and a set of predefined operations on those objects

  - 프로그램의 모든 데이터에는 타입(type)이 있음
  - 예 1 : x = 12 + 3.456; /* 12는 정수 타입, 3.456은 부동 소수점 타입 */
    예 2 : int x; /* 타입 int를 변수 x에 바인딩, 변수 x가 가질 수 있는 값은 -2,147,483,648 ~ 2,147,483,647 (4 바이트)의 정수들의 집합 */

  ➔ **데이터 타입** : 그 타입의 변수가 가질 수 있는 값들의 집합과 이러한 값들에 적용할 수 있는 연산들(operations)의 집합

  ➔ 정수나 부동소수점 타입에 대한 +, -, *, / 등의 연산

  ➔ Primitive data type (int, float, char, …)

  ➔ User-defined data type (COBOL:record, C:structure)

Introduction

- A **descriptor** is the collection of the attributes of a variable (name, address, value, type, lifetime, scope)
  - **An area of memory** that stores the attributes of a variable
  - 속성이 정적/동적인가에 따라 descriptor의 일부 또는 전체의 유지기간이 달라짐

- An **object** represents an instance of a user-defined (abstract data) type (more details in ch.11, 12, OOP)
  - An **object** is an **instance of class** whether predefined or user-defined

- One design issue for all data types:
  What operations are defined and how are they specified?

Primitive Data Types

- Almost all programming languages provide a set of primitive data types
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
  - Most integer types
- Others require only a little non-hardware support for their implementation

Primitive Data Types: Integer (C, C++)

- Almost always an exact reflection of the hardware so the mapping is trivial

- Integer types C, C++

Type	Size	Range
(signed) short (int)	2바이트	-32,768~32,767
unsigned short (int)	2바이트	0~65,536
(signed) int	4바이트	-2,147,483,648~2,147,483,647
unsigned (int)	4바이트	0~4,294,967,295
(signed) long (int)	4바이트	-2,147,483,648~2,147,483,647
unsigned long (int)	4바이트	0~4,294,967,295

Primitive Data Types: Integer (Java)

- Almost always an exact reflection of the hardware so the mapping is trivial

- Integer types in Java

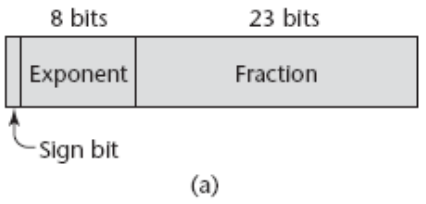
Type	Size	Range
byte	1바이트	-128 ~ 127
short	2바이트	-32,768 ~ 32,767
int	4바이트	-2,147,483,648 ~ 2,147,483,647
long	8바이트	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

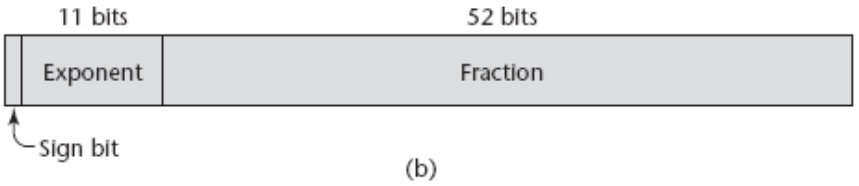
Primitive Data Types: Floating Point

- Model real numbers, but only as approximations (근사값)
- Languages for scientific use support at least two floating-point types (e.g., `float` and `double`; sometimes more
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754 (Figure 6.1)

Figure 6.1

IEEE floating-point formats: (a) single precision, (b) double precision





- 가수부(부호 없는 정수)
- 지수부(부호 있는 정수)

가수 * 밑수지수

Fraction * (base number)^{exponent}

ex) $0.1225 * 10^{-15}$

Primitive Data Types: Complex

- Some languages support a complex type, e.g., C99, Fortran, and Python(현재 C에서는 표준라이브러리 차원에서는 지원하지 않음)
- Each value consists of two floats, the real part and the imaginary part
- Literal form (in Python):

  `(7 + 3j)`, where `7` is the real part and `3` is the imaginary part

```
>>> x = 1 - 2j
>>> type(x)
<class 'complex'>
>>> x.imag
-2.0
>>> x.real
1.0
>>> x.conjugate()
(1+2j)
>>>
```

- **imag** - imaginary number
- **real** - real number
- **conjugate() method** - returns conjugate complex number (켤레)

Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Stored like character strings, using binary codes for the decimal digits (BCD - Binary Coded Decimal)
  - 같은 수를 십진수로 표기하는 것이 이진수로 표기하는 것보다 많은 메모리 차지
  - BCD(이진화 십진법): **이진수 네 자리를 묶어 십진수 한 자리로 사용하는 기수법**
  - 729 → 0111 0010 1001 **(추가적인 연산(bit shift, 조건부 덧셈을 통한 나눗셈)이 필요)**
  - 1011011001_2

십진법	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

- Advantage: accuracy
- Disadvantages: limited range(지수 표현 X), wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements, one for "true" and one for "false"
- Could be implemented as bits, but often as bytes
  - Often stored in the smallest efficiently addressable cell of memory, typically a byte
- Advantage: readability

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: **ASCII(8-bit code)**
  - Uses the values 0 to127 to code 128 different characters
- An alternative, **16-bit coding: Unicode (UCS-2)**, 1991
  - Includes characters from most natural languages
  - Originally used in Java
  - JavaScript, Python, Perl, C# and F# also support Unicode
- 32-bit Unicode (UCS-4 or UTF-32)
  - ISO/IEC 10646, Published in 2000
  - Supported by Fortran, starting with 2003

Character String Types

- Values are sequences of characters

- Design issues:
  - Should strings be
    just a special kind of character array or
    a primitive type?

  - Should strings have static or dynamic length?

Character String Types Operations

- Typical operations:
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Character String Type in Certain Languages

- C and C++
  - Not primitive
  - use **`char`** arrays and a library of functions that provide operations
    - **`char` array: (char str[] = "apples")**
    - **functions : strcpy, strcat, strcmp, and strlen**
  - In C++, programmers should use **string** class from the standard library rather than char arrays and the C string library
    - **Why? The insecurities of the C string library**
    - ex) **strcpy(dest, src)** /* dest :20 bytes, src : 50 bytes */

Character String Type in Certain Languages (continued)

- Java
  - Primitive via the `String` class & `Stringbuffer` class
    - `String` class - constant strings
      - String 클래스를 생성하면, JVM은 내부의 문자열 저장소에 동일한 내용의 문자열이 있는지 확인함
      - 문자열이 있는 경우 : 해당 문자열을 참조하는 레퍼런스 변수만 넘겨줌
      - 문자열이 없는 경우 : 새로운 문자열을 생성하여 문자열 저장소에 저장
        ≫ 한번 생성되면 변하지 않는 문자열(상수 문자열)
    - `Stringbuffer` class - changeable and more like arrays of single characters
      - JVM 내부에 새롭게 문자열을 생성하는 것이 아니라, 메모리 상에서 문자열을 처리하기 때문에 동적으로 문자열의 내용을 바꾸거나 위치를 조정할 수 있음
        (정해진 위치를 지정하여 문자열을 추가 할 수 있음)

Character String Type in Certain Languages (continued)

- Perl, JavaScript, Ruby, and PHP
  - Provide built-in pattern matching, using regular expressions
  - ex) Regular expressions
    - **/ [A-Za-z] [A-Za-z\d]+/**
      - **Matching the typical Name form**
      - **[ ]** – character classes (문자의 유형)을 포함
      - **[A-Za-z]**: all letters (모든 문자의 유형)
      - **[A-Za-z\d]**: all letters and digits (모든 문자와 숫자의 유형)
      - **+**: 해당 유형에 속한 것이 적어도 한번 이상 나와야 함
    - **/ \d+\.? \d* | \.\d+/**
      - **Matching numeric literals**
      - **\d+** : 한 개 이상의 숫자
      - **\.** : 십진수의 소수점
      - **?** : 앞에 오는 것이 0개 또는 1개 가능함 (여기서는 소수점(.))
      - **\d*** : 0개 이상의 숫자가 나와야 함
      - **|** : 두 가지 선택사항의 구분 (두 가지 형태의 스트링 매칭 중 선택)
        - » 한 개 이상의 숫자 → 십진수의 소수점**(optional)** → **0**개 이상의 숫자
        - » 십진수의 소수점 → 한 개 이상의 숫자

Character String Length Options

- **Static length string** (Java's `String` class)
  - The length of string values can be static and set when the string is created
  - String class in Java, C#, Python, Ruby
- **Limited dynamic length string** (C and C++)
  - In these languages, a special character('\0') is used to indicate the end of a string's characters, rather than maintaining the length (고정된 최대 길이 내에서 가변적 길이의 허용)
  - C strings & C-style strings of C++
- **Dynamic length String** (no maximum)
  - Perl, JavaScript, the standard C++ library
  - 최대의 길이 제한 없이 가변적 길이의 허용
- Ada supports all three string length options

Character String Type Evaluation

- String types are important to the writability of a language
- As a **primitive type** with static length, they are inexpensive to provide--why not have them?
  - Predefined function (standard library ?)
  - ex) strcpy in C

  - 스트링 타입을 기본 타입으로 제공할 때 compiler에 미치는 cost는 크지 않음
  - 스트링 타입을 기본 타입으로 제공하지 않는 언어는 스트링 조작 함수를 제공

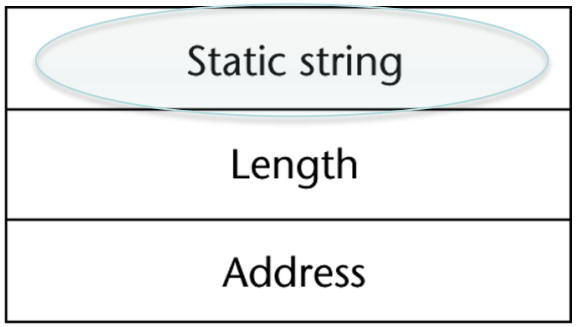
- Dynamic length is nice, but is it worth the expense?

Character String Implementation

- **Static length**: compile-time descriptor
- **Limited dynamic length**: may need a run-time descriptor to store both the fixed maximum length and the current length
  - but not in C and C++, compile-time descriptor)
- **Dynamic length**: need run-time descriptor; allocation/deallocation is the biggest implementation problem
  - Store the current length

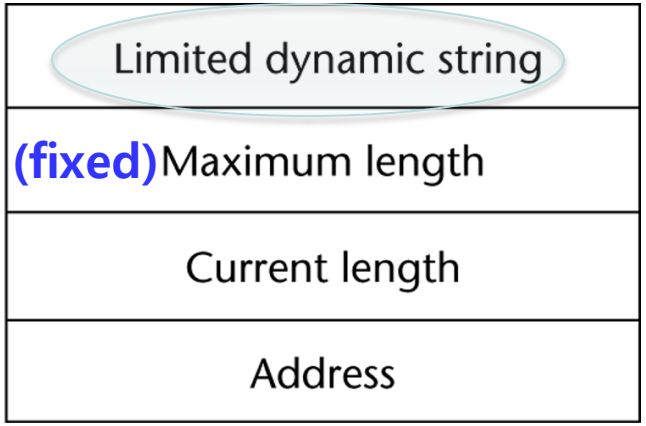
Compile- and Run-Time Descriptors

Compile-time descriptor for static strings



타입 이름

Run-time descriptor for limited dynamic strings



※ C, C++에서 limited dynamic strings: 문자열의 끝을 Null 문자('₩0')로 표시하여 run-time descriptor가 필요 없음. 배열 참조 시 색인(index) 값의 범위를 검사하지 않기 때문에 최대 값이 필요 없음. 컴파일러는 최대 길이를 위한 충분한 공간을 바인딩 (한 번만 할당)

Character String Implementation (continued)

- **Dynamic length**: need run-time descriptor; allocation/deallocation is the biggest implementation problem
  - Linked list에 저장
    - 많은 기억장소를 요구
    - 느린 스트링 연산 속도
    - 할당과 회수가 용이
  - Heap에 할당된 각 문자들을 가리키는 포인터들의 배열로 저장
    - 여분 메모리 요구
    - 연결 리스트보다 빠른 접근
  - 인접한 메모리 셀(adjacent storage cell)에 저장
    - 적은 기억장소를 요구
    - 빠른 스트링 연산 속도
    - 스트링 길이가 증가될 때 재 할당 문제
    - 회수과정이 복잡

Character String Implementation (continued)

- **Dynamic length**: need run-time descriptor; allocation/deallocation is the biggest implementation problem
  - To store **complete strings** in **adjacent storage cells**
    - 문제점
      - String이 늘어날 경우 인접한 메모리 셀을 확보 할 수 없음
    - 해결방안
      - 새로운 String 전체를 저장할 수 있는 새로운 메모리 영역을 찾아 할당함
      - 이전의 String 전체를 위해 사용했던 메모리 영역은 회수함

User-Defined Ordinal Types

- Ordinal type
  - one in which the range of possible values can be easily associated with the set of positive integers
  - 값의 범위가 양의 정수의 집합
  - Examples of primitive ordinal types in Java
    - `Integer, char, boolean`

- User-defined Ordinal type
  - Enumeration, subrange types

Enumeration Types

- All possible values, which are named constants, are provided in the definition

- C, C++, C# example (enumeration constant)

```
enum days {mon, tue, wed, thu, fri, sat, sun};  // implicitly assigned
enum days {mon=1, tue, wed, thu, fri, sat, sun}; // explicitly assigned
```

- Design issues
  - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
  - Are enumeration values coerced to integer?
  - Any other type coerced to an enumeration type?

Enumeration Types (Designs)

- C++는 C의 열거 타입을 포함

```
enum colors {red, blue, green, yellow, black};

colors myColor = blue, yourColor = red;
```

  - 열거 상수를 한 개의 열거 타입 정의에서만 허용
  - 정수타입으로 강제변환 : myColor++; // legal and would assign green to my Color
  - 다른 타입 값은 열거 타입으로 **강제 변환되지 않음** : `myColor = 4; // illegal`

- Java 5.0에 열거 타입을 추가
  - Subclasses of the predefined class `enum` (모든 열거 타입은 Enum의 부 클래스)
  - 클래스이므로 field, constructor, method 포함

- C#의 열거 타입은 정수로 강제 변환되지 않는 것을 제외하고, C++와 동일

Evaluation of Enumerated Type

- Aid to readability, e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
  - operations (don't allow colors to be added)
  - No enumeration variable can be assigned a value outside its defined range
  - Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

❖ **Ada, C#, F#, Java5.0**
- ✓ 열거타입의 변수는 산술연산 안됨: week + day // illegal
- ✓ 정의된 범위를 벗어난 값의 할당 안됨 : 내부 값으로 0..9를 사용할 경우 9보다 큰 수는 enum type 변수에 할당될 수 없음

❖ **C**
- ✓ Enumeration variable를 integer variable로 취급
- ✓ **enum** days {mon=20, tue, wed, thu=30, fri, sat, sun}; // legal

Subrange Types (skip~)

- An ordered contiguous subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type
- Ada's design

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2:= Day1;
```

Subrange Evaluation (skip~)

- Aid to readability
  - Make it clear to the readers that variables of subrange can store only certain range of values
- Reliability
  - Assigning a value to a subrange variable that is outside the specified range is detected as an error

Implementation of User-Defined Ordinal Types

- Enumeration types are implemented as integers

For providing increase in reliability

Restriction on range of values and operations

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- Can arrays be initialized when they have their storage allocated?
- Are any kind of slices supported?

Array Types

- **Structured data type**
  - 여러 데이터를 묶어서 하나의 단위로 처리하는 데이터 타입
  - **Arrays**, associative arrays, records, tuples, lists, and unions
- **An array**
  - a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
  - 개개의 데이터 원소들은 동일한 타입
  - 각각의 원소(individual element)는 첫 번째 원소의 상대적인 위치에 의해 식별
  - 원소에 대한 참조를 위해 첨자(subscript)사용 : **int a [25];**
  - 첨자 식에 변수가 포함되면 참조되는 메모리 위치의 주소를 결정하기 위해 추가적인 실행 시간의 계산(additional run-time calculation)이 필요함

```
for (i=0; i<5; i++) {
        if (i >= 0 && i < 5) {
            sum += arScore[i];  /* a variable in subscript ?? */
        }
}
```

Array Indexing

- **Indexing** (or **subscripting**) is a mapping from indices to elements

  array_name (index_value_list) → an element

- Index Syntax
  - Fortran and Ada use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
  - Most other languages use brackets

※**( )**: parentheses, **{ }**: brace, **[ ]**: square bracket,
< >: angle bracket(꺾쇠괄호)

Arrays Index (Subscript) Types

- C, Java
  - integer only
  - 정수형 변수, 정수형 상수, 정수로 계산되는 수식: [2*i]

- Index range checking
  - An important factor in the reliability of languages
  - C, C++ do not specify range checking
  - Java, C# specify range checking

Subscript Binding and Array Categories

- **Static:** subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (**no dynamic allocation or deallocation**)
  - Disadvantage: storage for the array is **fixed for the entire execution time** of the program
  - Ex) ***C, C++:*** `static int a[10];`

Subscript Binding and Array Categories

- **Fixed stack-dynamic**: subscript ranges are statically bound, but the allocation is done at declaration time
  - Advantage: **space efficiency** (동일 공간을 여러 프로시저의 배열이 사용할 수 있음)
  - Disadvantage: **run-time overhead** of allocation and deallocation
  - Ex) *C, C++*: `an array declared in a function`

Subscript Binding and Array Categories (continued)

- **Stack-dynamic**: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
  - Advantage: flexibility (the size of an array need not be known until the array is to be used)
  - Ex) ***Array in Ada***

```
GET (LIST_LEN);
declare
LIST : array (1 .. LIST_LEN) of INTEGER;
```

Subscript Binding and Array Categories (continued)

- **Fixed heap-dynamic**: similar to fixed stack-dynamic: storage binding is dynamic but **fixed after allocation** (i.e., binding is done when requested and storage is allocated from **heap**, not stack)
  - 기억장소가 할당된 후 첨자범위와 기억장소가 고정
  - **Explicit allocation/deallocation**
  - **C: malloc, free,**

C++: new, delete

What are the differences in subscipt ranges and stroage bindings between fixed -stack-dynamic and fixed heap-dynamic array?
answer: The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the storage is allocated from the heap, than the stack

사용자 프로그램 요청이 왔을때 스토리지 바인딩, 첨자 바인딩이 진행됨

fixed statck -dynamic의 경우 ????질문

What is the difference in subscript binding between fixed stack-dynamic and fixed heap-dynamic array?

Stack or heap??

Subscript Binding and Array Categories (continued)

C- malloc, free

예) 문자 100개를 저장 할 수 있는
메모리 할당

```
char *ptr;

ptr = malloc(sizeof(char)*100);

…

ptr[0] = 'a';
Ptr[1] = 'b'

…

free(ptr);
```

C++ - new, delete

예) 문자 100개를 저장 할 수 있는
메모리 할당

```
char *ptr;

ptr = new char[100];

…

ptr[0] = 'a';
ptr[1] = 'b';

…

delete[] ptr; // array 전체 메모리 해제

// delete ptr 의 경우 ptr[0]에 대한
   소멸자 호출 및 메모리 해제만 이루어 짐
```

Subscript Binding and Array Categories (continued)

- **Heap-dynamic**: binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (**arrays can grow or shrink during program execution**)
  - Examples

Java: ArrayList class (Generic class)

```
ArralyList intlist = new ArrayList();
 ...
intList.add(nextOne); //객체에 element 추가
```

C#: Generic※ heap-dynamic array

```
List<String> stringList = new List<String>();
 ...
stringList.add("Michael");
```

※ Generic: 클래스에 사용할 타입을 디자인 시에 지정하는 것이 아니라
클래스를 사용할 때 지정한 후 사용하는 기술 (C++에서는 template으로 많이 쓰임)

Array Examples - 1

```
#include <stdio.h>
#include <math.h>

#define IMAX 1000000
#define JMAX 2000
#define ZMAX 2

int main(void)
{
  int JLS[ZMAX];
  int IGS[JMAX];
  double Y[JMAX];
  double X[IMAX];
  double F[IMAX];

  int i,j,k;

  for (i = 0; i < IMAX; i++){
    IGS[i] = 0;
    F[i] = 0;
    X[i] = 0;
  }
  return 0;
}
```

- Compilation → Segmentation fault!!

- Cause
  - ✓ `int IGS[JMAX];` // stack area
    - main 함수에 선언된 모든 배열(5개)
  - ✓ 함수의 스택영역 변수(지역변수)의 크기를 너무 크게 잡으면 메모리 구조가 깨짐(segmentation fault)

- Solution
  - ✓ `int *IGS = malloc(sizeof(int) * IMAX);`
    - ✓ IGS[i]가 차지할 메모리 공간의 확보
    - ✓ 충분한 공간이 있는 Heap 영역 사용

Array Examples - 2

```
view plain  copy to clipboard  print  ?
01.  #include <iostream>
02.  usng namespace std;
03.
04.  void function (int);
05.
06.  int main()
07.  {
08.      int size;
09.      cin>>size;
10.      function(size);
11.      return 0;
12.      //메인함수에서 요구되어 지는 메모리 공간의 크기는 컴파일 타임에 크기가 결정 가능하다.
13.          //이런 것을 올리기 위한 영역이 Stack이다.
14.  }
15.
16.  void function(int i)
17.  {
18.      static int i = 10; //funtion이라는 함수가 처음 호출될때 i라는 변수가 초기화 된다.
19.      int array[i];
20.  }
```

- 에러 발생
- int array[i]; // stack에 올리라는 의미. 불가능함!
- 첨자 i의 값이 compile-time에 그 크기가 계산되어 결정되지 않음
- run-time에 프로그램이 실행되면서 size의 크기가 결정된 후, i의 값이 결정
- 위 코드에서 array 배열을 run-time에 크기가 결정되는 구조
- 따라서, malloc을 통해 heap 영역을 사용해야 함
  → int *array = malloc(sizeof(int) * 10);

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without static modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- Java includes a generic class similar to C#'s `List`, `ArrayList` that provides heap-dynamic
  - Subscripting is not supported (In List, supported)
  - `get` and `set` methods must be used to access the elements

- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array in C – Tip1

- **초기화 되지 않은 변수의 경우 그 변수의 저장 위치에 따라 초기화 되는 조건이 다름**
- **지역 자동변수(local auto variable) - Stack 영역**
  - 초기화 되지 않은 변수 값은 어떤 값이 될지 예측할 수 없음(일반적으로 garbage value라 표현)
  - 함수가 호출되는 순서나 시점에 따라 직전에 호출 되었던 함수가 지역 자동변수를 저장하기 위해 사용했던 공간을 그대로 다시 사용하기 때문에 어떤 값이 저장되었을 지 예측이 불가능함
  - 일반적으로 Stack은 **여러 함수에서 공유해서 사용**
  - **초기화 되지 않은 자동변수의 값을 바로 읽는 코드를 작성**
    - 상황에 따라 다른 결과를 보이는 프로그램이 만들어지게 되며,
      이러한 버그는 찾아내기가 아주 어렵기 때문에 가능하면 초기화 하는 습관을 들이는 것이 좋음
- **Static data 영역을 사용하는 전역변수**
  - Static 변수는 초기화 하지 않았다면 compile time에 컴파일러가 0으로 초기화 해줌

Array in C – Tip2

- **함수 내 선언 및 사용되는 배열**
  - 경우 배열 자체가 STACK 영역에 생성되고, **함수가 호출될 때 마다 해당 배열을 위한 메모리를 새로 생성하고 초기화 하는 작업**을 하기 때문에 배열의 크기가 큰 경우에는 다른 방법을 사용해야 함
  - 배열을 초기화하는 경우 함수호출 때마다, 값을 채워 넣어야 하므로 프로그램의 수행 속도가 느려짐 (**특히나 자주 호출되는 함수인 경우**에는 프로그램의 속도를 치명적으로 느리게 만듦)
  - **이론적으로 스택** -- 지역 자동변수가 저장되는 영역의 크기는 제한되지 않지만, 시스템에 따라 하드웨어적인 제한이나 STACK 운영의 효율을 높이기 위해 STACK의 크기를 제한함. (함수 내에서 선언되는 배열의 크기가 과도하게 큰 경우 **STACK OVERFLOW** 에러가 발생하고 프로그램이 강제 종료 될 수 도 있음)
  - 컴파일러의 종류나 옵션에 따라서는 컴파일된 프로그램의 크기가 커질 수 있음
- **함수 내 배열의 크기가 크다고 생각되는 경우**
  - STATIC AREA를 사용하는 변수(**전역 변수나 스태틱 변수**)를 사용
  - alloc() 계열의 함수를 이용하여 **HEAP AREA**을 할당하여 사용

Heterogeneous Arrays

- A heterogeneous array (이형배열) is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby
- JavaScript example
  -
```
myArray[0]=Date.now;      //variable object
myArray[1]=myFunction(); // function object
myArray[2]=myCars;        // another array object
```

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Array Initialization

- **C-based languages**
  - `int list [] = {1, 3, 5, 7}`
  - `char *names [] = {"Mike", "Fred", "Mary Lou"};`
- Ada
  - `List : array (1..5) of Integer :=`
    `(1 => 17, 3 => 34, others => 0);`
- Python
  - List comprehensions

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```

puts `[0, 9, 36, 81]` in `list`

Arrays Operations (skip~)

- The C-based languages do not provide any array operations, except through the methods of Java, C++, and C#

```
public class ArrayTest {
    public static void main(String[] args) {

    int[] onjoraclejava  = new int[]{5, 4, 6, 9, 7, 9};
        int[] target = {10, 20, 30, 40, 50, 60,70, 80};

        System.arraycopy(onjoraclejava, 2, target, 3, 4);

        for(int i=0; i<target.length; i++)
        {
            System.out.println("target["+i+"] : " + target[i]);
        }
    }
}
```

- System.arrayCopy()메서드 이용

```
public static void arraycopy(Object src , int src_position ,
                             Object dst ,int dst_position ,int length)
```

Object src : 원본배열
int src_position : 원본배열의 시작위치

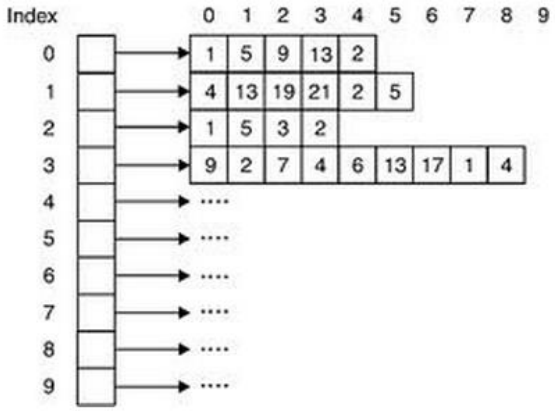
Object dst : 복사할 배열
int dst_position : 복사할 배열의 시작위치
int length : 복사할 개수.

[결과]

```
target[0] : 10
target[1] : 20
target[2] : 30
target[3] : 6
target[4] : 9
target[5] : 7
target[6] : 9
target[7] : 80
```

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
  (가변적인 columns(세로줄-열)의 수를 갖는 row(가로줄-행))
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays
  - myArray[rows][columns]



- C, C++, and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays (C# also supports jagged arrays)

Rectangular and Jagged Arrays - Example (C#)

```
// arrays.cs
using System;
class DeclareArraysSample
{
    public static void Main()
    {
        // Single-dimensional array
        int[] numbers = new int[5];

        // Multidimensional array
        string[,] names = new string[5,4];

        // Array-of-arrays (jagged array)
        byte[][] scores = new byte[5][];

        // Create the jagged array
        for (int i = 0; i < scores.Length; i++)
        {
            scores[i] = new byte[i+3]; // length of row i is set to (i+3)
        }                              // scores.Length → 5
                                       // value is initialized to zero
        // Print length of each row
        for (int i = 0; i < scores.Length; i++)
        {
            Console.WriteLine("Length of row {0} is {1}", i, scores[i].Length);
        }
    }
}
```

<output>

```
Length of row 0 is 3
Length of row 1 is 4
Length of row 2 is 5
Length of row 3 is 6
Length of row 4 is 7
```

Ref) https://msdn.microsoft.com/ko-kr/library/aa288453(v=vs.71).aspx

To the 2nd Topic