Just For Func!

프로그래밍 언어론의 개요

프로그래밍 언어

- 프로그래밍 언어(Programming Language)
- 계산 과정을 기술하기 위한 표기법
  - 다른 언어와 다르게 사람→기계의 일방향 언어

- 프로그래밍 언어의 중요성
- 언어의 구조가 사고의 범위를 지배
  - 언어에 따라 사고방식에 차이가 발생
- 소프트웨어의 발전은 프로그래밍 언어를 매개로 하여 발전
- "언어와 사랑에 빠지지 마라!"
  - 다양한 언어의 장단점을 이해하고 목표에 따라 선택

'25/08	프로그래밍 언어	Ratings
1	Python	26.14%
2	C++	9.18%
3	C	9.03%
4	Java	8.59%
5	C#	5.52%
6	JavaScript	3.15%
7	Visual Basic	2.33%
8	Go	2.11%
9	Perl	2.08%
10	Delphi/Object Pascal	1.82%

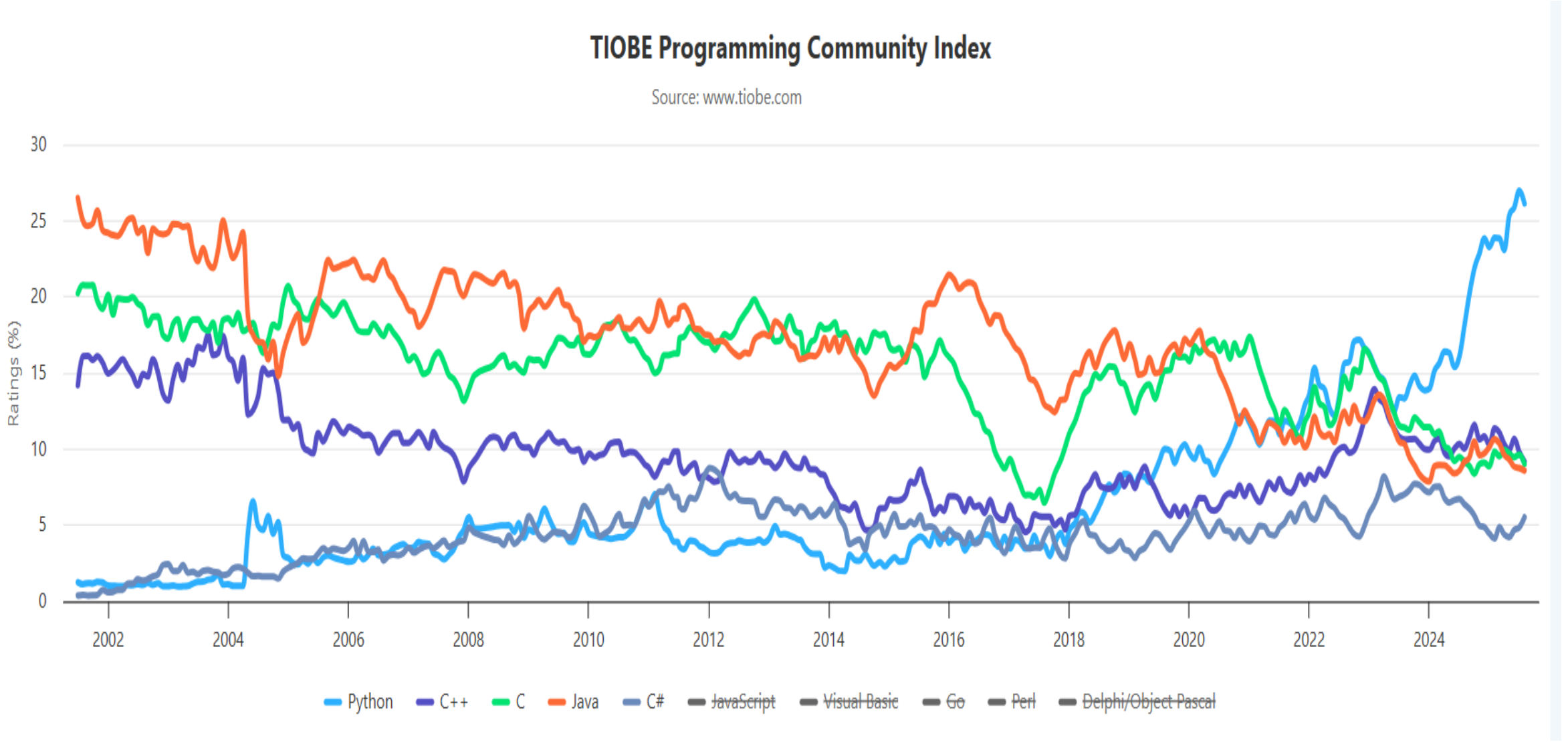
< TIOBE index >

[https://www.tiobe.com/tiobe-index/]

프로그래밍 언어

11	10	⌄	F	Fortran	1.75%	-0.03%
12	7	⌄⌄	SQL	SQL	1.72%	-0.49%
13	30	⌃⌃	Ada	Ada	1.52%	+0.91%
14	19	⌃⌃	R	R	1.37%	+0.26%
15	13	⌄	php	PHP	1.27%	-0.19%
16	11	⌄⌄		MATLAB	1.19%	-0.53%
17	20	⌃		Scratch	1.15%	+0.06%
18	14	⌄⌄		Rust	1.13%	-0.15%
19	18	⌄		Kotlin	1.10%	-0.04%
20	17	⌄	ASM	Assembly language	1.03%	-0.19%

프로그래밍 언어


TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
2022
2024
Python
C++
C
Java
C#
JavaScript
Visual Basic
Go
Perl
Delphi/Object Pascal

프로그래밍 언어론

- 프로그래밍 언어론
- 프로그램의 기술 방법, 코드의 의미 및 구현과 실행에 대해 연구하는 분야
- 주요 주제
  - 구문법(Syntax): 토큰/문장 구조, BNF/EBNF, AST, 파싱, 스코프 및 바인딩 기초, ...
  - 의미론(Semantics): 작동적 의미, 지시적 의미, 공리적 의미, 정당성/완전성, ...
  - 타입 시스템(Type system): 정적/동적, 다형성, 타입 추론, ...
  - 패러다임(Paradigm): 명령형(절차형), 함수형, 논리, 객체지향 등
  - 구현 및 실행: 컴파일/인터프리트/JIT, 가비지 컬렉션(Garbage Collection, GC), 메모리 모델, ...

프로그래밍 언어 분류

- 프로그래밍 언어의 분류
- 추상화 수준
  - 저급 언어: 컴퓨터 구조에 종속적(기계어, 어셈블리어)
  - 고급 언어: 컴퓨터 구조에 독립적(대부분의 언어)
- 목적/범용성
  - 범용(General Purpose Language, GPL): C, Java, Python, Rust 등
  - 도메인 특화(Domain Specific Language, DSL): SQL, HTML/CSS, Verilog, R, MatLab 등
- 실행 방식
  - 컴파일형: C/C++, Rust, Go 등
  - 인터프리트형: Python, Ruby, Lua 등
  - 하이브리드형: Java(JVM), C#(.NET), JavaScript(V8) 등
- 패러다임
  - 명령형/절차형, 함수형, 논리, 객체지향형

프로그래밍 패러다임

- 프로그래밍 패러다임(Programming Paradigm)
- 프로그램의 상태/제어 흐름/자료 추상화등을 규정하는 설계 원칙의 형식적 체계
- 명령형 프로그래밍(Imperative Programming)
  - 문제를 해결하는 명령(절차)를 기술하는 방식의 프로그래밍
  - Ex) C, Pascal, Ada, Python 등
- 함수형 프로그래밍(Functional Programming)
  - 프로그램의 계산 과정을 수학 함수 형태로 프로그래밍
  - Ex) Lisp, Scheme, ML, Haskell 등
- 논리 프로그래밍(Logic Programming)
  - 정형 논리(Formal logic)를 기반으로 한 프로그래밍
  - Ex) Prolog 등
- 객체지향 프로그래밍(Object-Oriented Programming)
  - 객체 개념을 기반으로 하는 프로그래밍
  - 다른 패러다임과 함께 적용
  - Ex) C++, Java, C#, Objective-C, Swift 등

프로그래밍언어와 컴파일러

변수와 데이터 타입

변수(1)

- 변수(Variable)
  - 값을 저장할 수 있는 메모리 영역
  - 변수의 속성
    - 이름, 주소, 값, 데이터 타입

식별자(Identifier)

- 프로그램에서 어떤 요소를 식별하기 위한 문자열
- 변수, 함수, 클래스, 객체, 상수, 예약어등을 구분

- 이름(Name)
  - 변수를 식별하기 위한 식별자
  - 각 언어에서 정의한 고유한 명명 규칙(Naming Rule)이 존재
  - 자주 사용되는 일반 규칙
    - 문자, 숫자, 밑줄(_) 사용만 가능 (특수문자 사용 불가)
    - 첫 글자는 숫자 불가
    - 언어에서 미리 정의된 예약어(Reserved word)는 사용 불가 (if, else, for...)
  - 언어별로 확인이 필요한 규칙
    - 대소문자 구별 여부 (SQL, Visual Basic, HTML등은 구별하지 않음)
    - 일부 특수문자 사용 가능 (Ruby, Perl, Lisp...)

변수(2)

- 주소(Address)
- 메모리의 특정한 영역에 부여된 주소
  - 모든 메모리 영역은 바이트(Byte) 단위의 고유한 주소를 가짐
  - 2B 이상의 값을 저장하는 경우, 시작주소를 의미

- 값(Value)
- 변수의 주소에 저장된 내용
- 일반적으로 배정문(=)에 의해 저장
  - l-value = r-value
  - 배정문 좌측(l-value) / 배정문 우측(r-value)
  - r-value는 **하나의 값으로 평가되어,** l-value가 가리키는 메모리 주소에 저장
  - Ex) x = 3+5; // 3+5를 연산한 후에, x라는 이름의 변수가 가리키는 주소에 저장

변수(3)

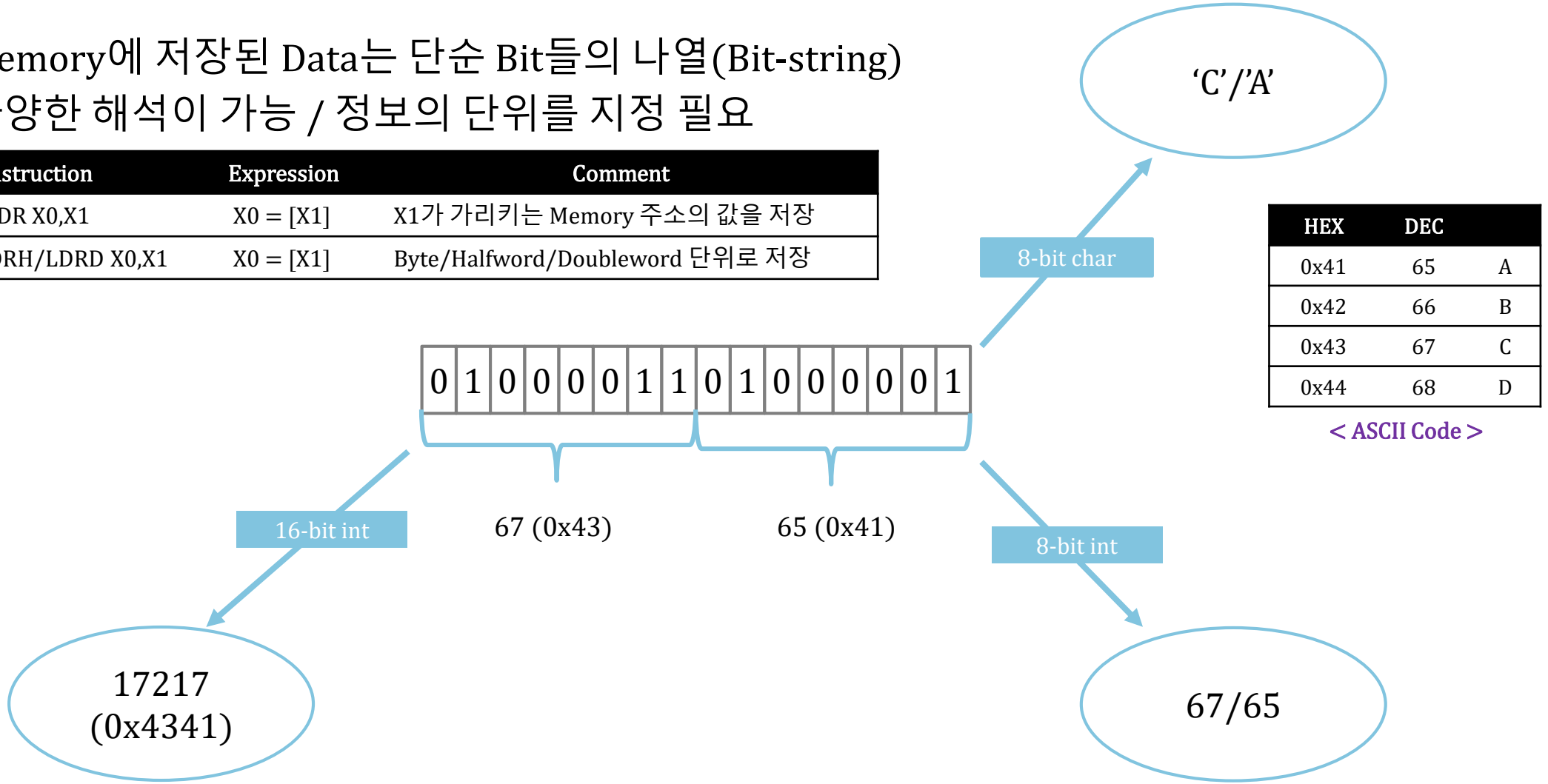
- 데이터 타입(Data Type)
- 변수의 크기, 값의 범위, 수행할 수 있는 연산의 집합
  - Ex) 자바의 주요 변수 타입

타입	크기	범위	주요 연산
byte	1바이트	$-2^7 \sim 2^7-1$	산술연산(+, -, *, /, %)
char	2바이트	$0 \sim 2^{16}-1$	산술연산(+, -, *, /, %)
int	4바이트	$-2^{31} \sim 2^{31}-1$	산술연산(+, -, *, /, %) 또는 비교연산(==, !=, <, >, <=, >=)
boolean	1바이트	true / false	논리연산(&&, ||, !)

변수 타입

- Memory에 저장된 Data는 단순 Bit들의 나열(Bit-string)
  - 다양한 해석이 가능 / 정보의 단위를 지정 필요

Instruction	Expression	Comment
LDR X0,X1	X0 = [X1]	X1가 가리키는 Memory 주소의 값을 저장
LDRB/LDRH/LDRD X0,X1	X0 = [X1]	Byte/Halfword/Doubleword 단위로 저장



HEX	DEC	
0x41	65	A
0x42	66	B
0x43	67	C
0x44	68	D

< ASCII Code >

바인딩

- 바인딩(Binding)
  - 변수의 속성이 구체적으로 부여

Ex) C/C++
int x; // x라는 이름을 4B 메모리 공간에 정수 타입으로 바인딩
x = 3; // x가 가리키는 메모리 공간에 3이라는 값을 바인딩

- 바인딩 시점

분류	시점	의미	예시
정적 바인딩	언어 정의	프로그래밍 언어를 정의할 때	타입의 의미, 연산자의 의미를 결정
	언어 구현	컴파일러를 구현할 때	타입의 범위, 수의 표기법을 결정
	컴파일 시점	소스 프로그램을 컴파일 할 때	변수의 타입을 결정
	링크 시점	프로그램을 라이브러리와 링크할 때	라이브러리 함수의 호출부와 코드부를 연결
	로드 시점	프로그램 실행을 위해 메모리에 적재될 때	전역 변수 또는 정적 변수의 주소를 결정
동적 바인딩	실행 시점	프로그램이 실행될 때	변수에 값을 할당

선언

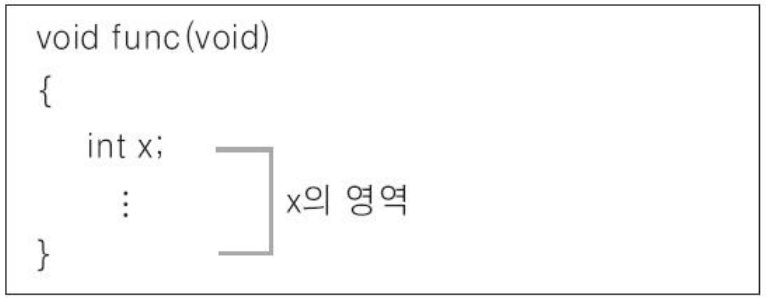
- 선언(Declaration)
- 변수 이름에 다른 속성을 바인딩
- 명시적 선언
  - 선언문을 사용하여 변수 이름에 속성을 바인딩
  - Ex) C/C++: int x; // x라는 이름에 정수 타입과 해당 메모리 주소를 바인딩
- 묵시적 선언
  - 별도의 선언문 없이 변수에 속성을 바인딩
  - Ex) FORTRAN: 변수명이 I~N으로 시작하는 변수는 자동으로 정수 타입을 바인딩
  - Ex) Perl: 변수명이 $로 시작하면 스칼라 타입, @로 시작하면 배열 타입을 바인딩
  - Ex) Python: 대입할 값의 타입으로 바인딩, 동적 바인딩

```
Ex) Python

x = 3.14     //실수 타입으로 자동 바인딩
x = “abc”    // 실수 타입 바인딩이 해제되고 문자열 타입으로 자동 바인딩
```

블록

- 블록(Block)
- 문장들의 집합으로 자체적인 선언을 가질 수 있는 범위
  - Ex) C/C++:{ }, Pascal: begin ~ end, Python, Occam: 들여쓰기 등

- 영역(Scope)
- 이름의 사용이 허락되고 있는 범위
  - Ex) 변수 x의 영역은 선언된 지점부터 func 함수의 끝까지
- 영역 탐색
  - x의 값을 사용하는 경우, 블록 내에서 선언문(명시적 선언) 또는 할당문(묵시적 선언)을 찾고, 없으면 바깥쪽 블록으로 x에 대한 선언문 또는 할당문 찾기를 시도



할당

- 할당(Allocation)
- 기억장소 할당
  - 변수에 메모리 공간을 바인딩
- 회수(Deallocation)
  - 변수로부터 바인딩이 해제된 메모리 공간을 가용 공간으로 돌려주는 과정

- 수명(Lifetime)
- 변수가 특정 메모리 주소에 바인딩되어 있는 시간
  - 변수의 수명은 변수가 메모리 공간에 바인딩될 때 시작되며 회수될 때 종료
- 수명에 따라 정적 할당, 스택 기반 할당, 동적 할당으로 분류

변수의 종류 및 수명

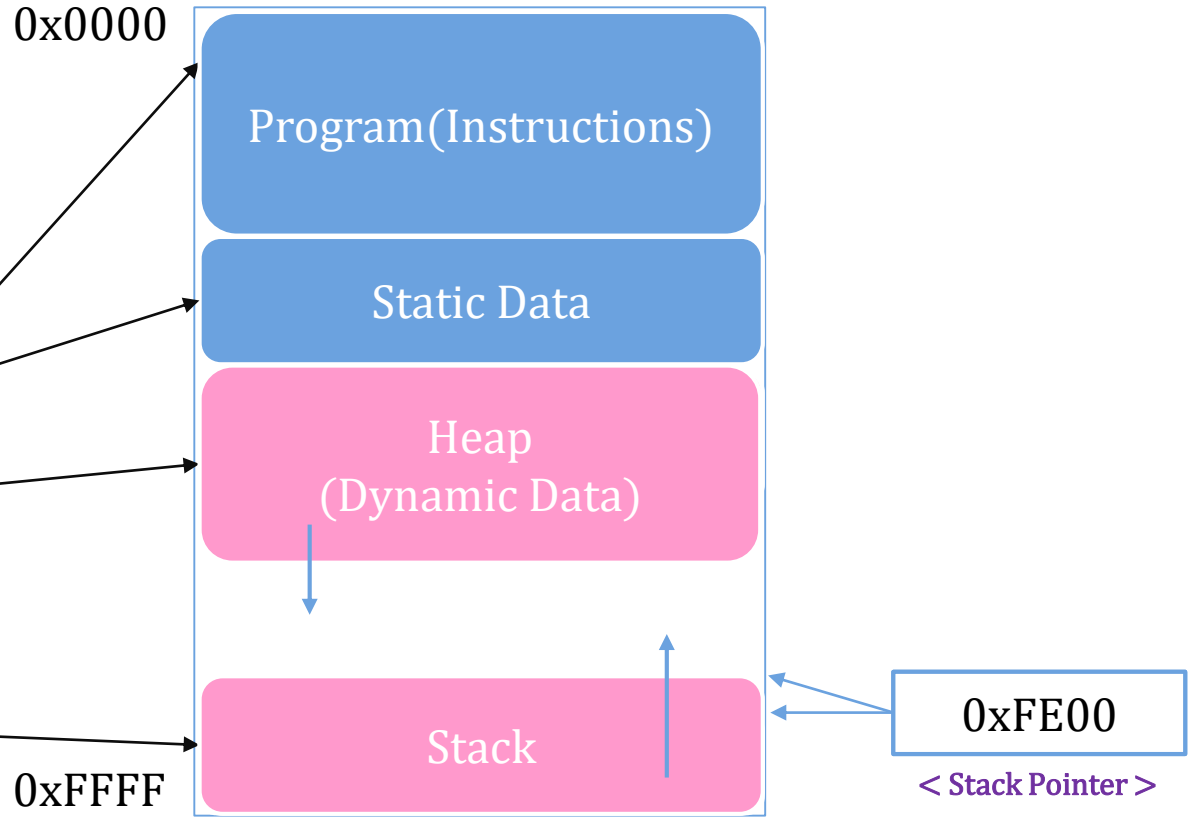
- 변수의 종류 및 수명
- 지역 변수
  - 영역이 블록 내로 한정
  - 블록의 실행이 끝나면 해당 변수 사용이 불가능
  - 인자를 받는 매개 변수는 지역 변수의 일종
- 전역 변수
  - 프로그램 전체에서 접근 가능
  - 프로그램이 끝날 때까지 수명이 유지
- 정적(Static) 변수
  - 영역이 블록 내로 한정
  - 프로그램이 끝날 때까지 수명이 유지

```
int SUM = 0;            // 전역 변수

void main() {
   int a = 3;           // 지역 변수
   int b = 4;           // 지역 변수
   int c = sum(a,b);    // 지역 변수
   printf("%d",c);
}

int sum(int x, int y){   // 지역 변수이고 매개변수
   static int cnt = 0;   // 정적 변수
   cnt++;
   int ret = x+y;        // 지역 변수
   return ret;
}
```

메모리 구조

- 메모리 구조(Memory Layout)
- 프로그램을 실행시 로드 시점에서 크게 4종류의 영역으로 구별
- 프로그램 영역
  - 명령어
- 정적 데이터 영역
  - 전역 변수, 정적 변수, 문자열
- 힙(Heap)
  - 동적 데이터 영역(Malloc, New, ...)
- 스택(Stack)
  - 함수 호출시 필요한 데이터 영역

```
int *a = mallc(1024);
static int cnt;
for (cnt=0;cnt<1024;cnt++) {
   a[cnt] = 0;
}
```



할당 방식(1)

- 정적 할당
- 변수에 메모리 공간이 정적으로 할당
  - 한번 할당되면 프로그램 실행이 종료될 때까지 할당 상태가 그대로 유지
- 정적 할당이 이루어지는 메모리 공간은 정적 영역
  - Ex) 전역 변수 또는 정적 변수

- 스택(Stack) 기반 할당 또는 자동 할당(Automatic allocation)
- 변수의 타입은 정적으로 할당되지만 메모리 공간은 실행 시간 중에 할당
- 스택 기반 할당이 이루어지는 메모리 공간은 스택
  - Ex) 지역 변수

할당 방식(2)

- 동적 할당
  - 명시적인 명령어에 의해 실행 시간에 할당
  - 동적 할당이 이루어지는 메모리 공간은 힙(Heap)
  - 동적으로 할당된 영역은 포인터나 참조 변수를 통해서 참조 가능

Ex) C

```
int *ptr = (int *)malloc(sizeof(int));
```

Ex) Java

```
int[] arr = new int[5];
```

이름 상수

- 이름 상수(Constant)
- 프로그램 전반에 걸쳐 고정된 값을 가지는 식별자
  - 변수와는 달리 값이 변경될 수 없음
- 판독성과 프로그램의 신뢰성을 증진
  - Ex) C/C++: const double PI = 3.14159;
  - Ex) Java: final float PI = 3.14159;
- 관례상 모두 대문자로 명명
  - Ex) int arr[ARRAY_SIZE];

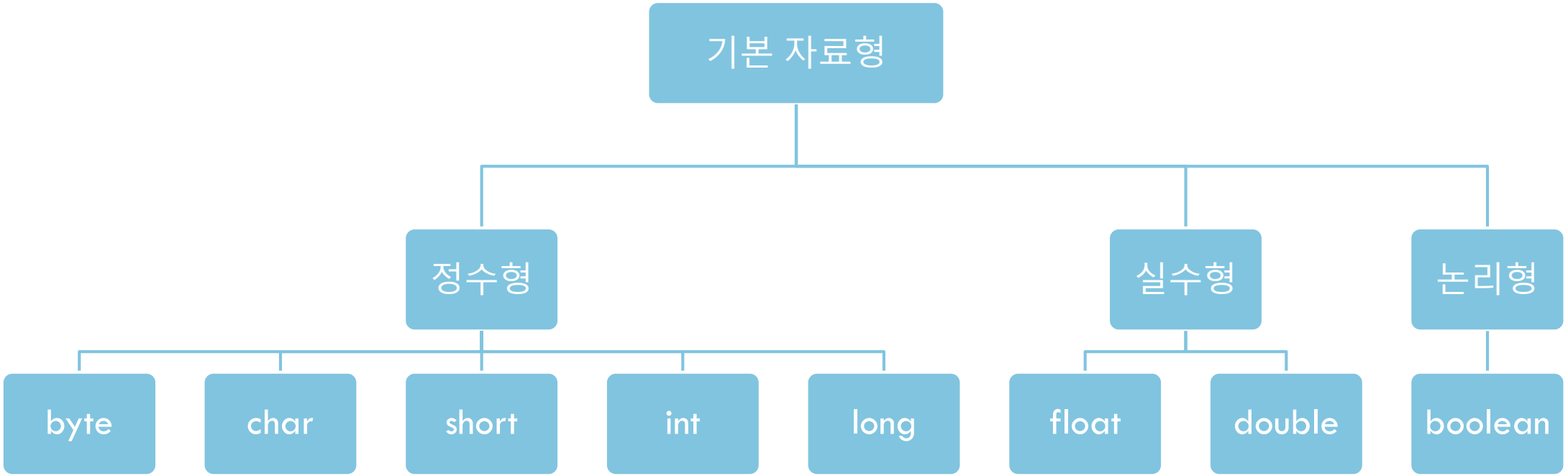
데이터 타입

- 데이터 타입(Data Type) or 자료형
- 변수가 가질 수 있는 값들의 집합

- 데이터 타입의 분류
- 언어 내재 여부
  - 기본 데이터 타입(Primitive Data Type): 정수형, 논리형 등과 같이 해당 언어에서 기본적으로 제공
  - 사용자 정의 데이터 타입(User-defined Data Type): 클래스와 같이 기본 데이터 타입을 이용하여 사용자가 생성
- 데이터 요소의 형태로 분류
  - 단순 데이터 타입(Simple Data Type): 하나의 데이터로만 구성된 타입 (Ex) int, float, bool, enum 등)
  - 복합 데이터 타입(Structure Data Type): 데이터들의 구조로 구성된 타입 (Ex) 배열, 구조체, 리스트, 클래스 등)

자바의 기본 자료형

- 기본 자료형
  - 8개



정수형(1)

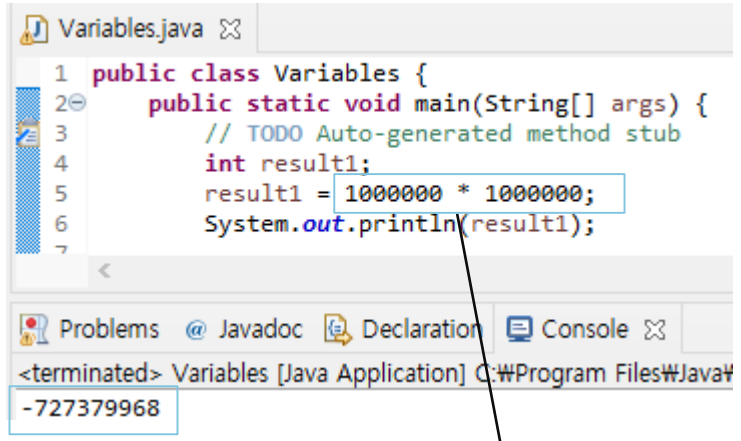
- 정수형(Integer Type)
- 정수 값을 표현
- 같은 크기라도 부호가 있는 범위(Signed)와 부호가 없는 범위(Unsigned)로 구분
  - n비트의 표현 범위: $-2^{n-1} \sim 2^{n-1}-1$ 또는 $0 \sim 2^{n}-1$
- Ex) 대표 정수형

크기	부호	C/Java 변수형	범위
8비트 (1B)	Signed	char(C) / byte(Java)	$-2^7(-128) \sim 2^7-1(127)$
	Unsigned	unsigned char(C)	$0 \sim 2^8-1(255)$
16비트 (2B)	Signed	short	$-2^{15}(-32768) \sim 2^{15}-1(32767)$
	Unsigned	unsigned short(C)	$0 \sim 2^{16}-1(65535)$
32비트 (4B)	Signed	int	$-2^{31}(-2147483648) \sim 2^{31}-1(2147483647)$
	Unsigned	unsigned int(C)	$0 \sim 2^{32}-1(4294967296)$
64비트 (8B)	Signed	long	$-2^{63} \sim 2^{63}-1$
	Unsigned	unsigned long(C)	$0 \sim 2^{64}-1$

정수형(2)

- 범위
- 오버플로우(Overflow)
  - 주어진 자료형의 최대 범위보다 큰 값을 처리하려고 할 때 발생
- 언더플로우(Underflow)
  - 주어진 자료형의 최소 범위보다 작은 값을 처리하려고 할 때 발생

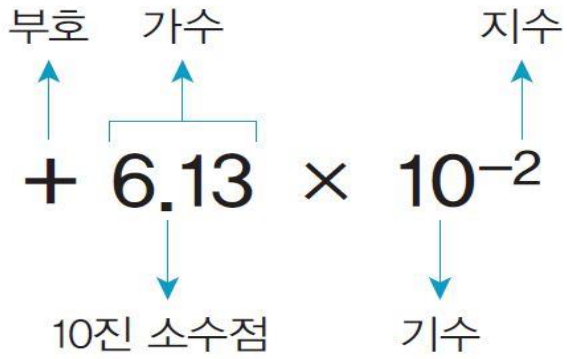
- 관련 연산
- 사칙연산(+,-,*,/), 나머지 연산(%,mod)
- 비트 연산(>>, <<, &, |)
- 비교연산(<, >, ==, !=)



실수형(1)

❖ 부동소수점(Floating Point Type)

- 실수 값을 표현
- IEEE-754 표준
  - 부호(Sign), 지수(Exponent), 가수(Mantissa)로 표현
  - $N = (-1)^S * M * 2^E$
  - 정밀도에 따라 단정도(Single precision) / 배정도(Double precision)로 구분
  - 변수 선언시 일반적으로 단정도는 float형, 배정도는 double형으로 정의



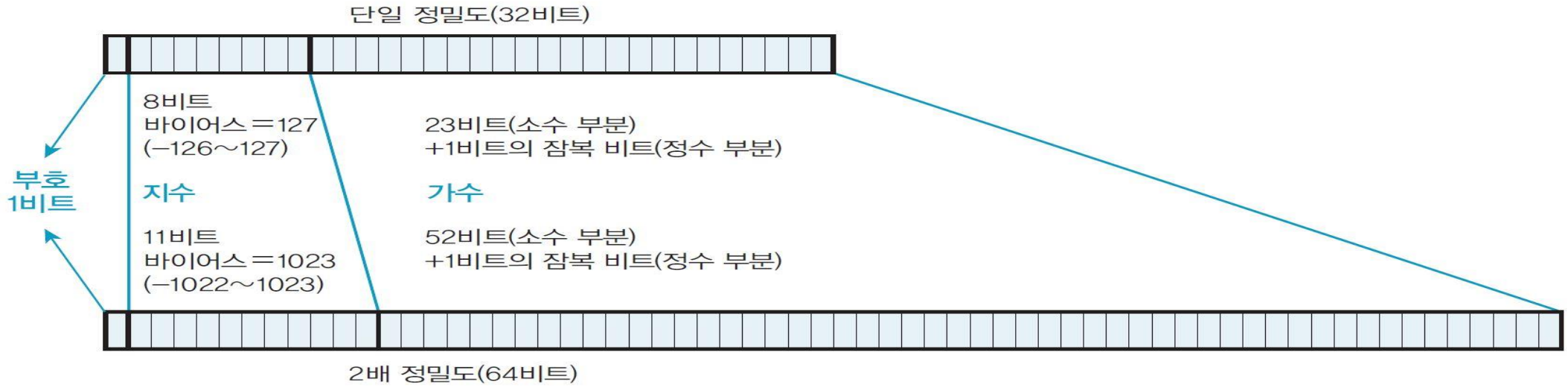
❖ 고정소수점(Fixed Point Type)

- 정수 값과 소수 값을 분리해서 표현
- 10진수 표현을 중시하는 일부 언어에서 지원
  - Ex) SQL - DECIMAL(5,1) // 1234.5와 같이 5자리 숫자 중 뒤에 1자리가 소수점
  - Ex) COLOL - PIC 9(5)V99 // 123.45와 같이 5자리 숫자 중 뒤에 2자리가 소수점

실수의 표현(2)

- IEEE-754 표준
- Ex) 12.5f = 0100 0001 0100 1000 0000 0000 0000 0000
  - Sign = 0 → 양수
  - Exponent = 10000010 (130) → 130 − 127 = 3
  - Fraction = 10010000000000000000000 → 실제 값 = 1 + 0.5 + 0.0625 = 1.5625
  - 최종 값 = 1.5625 × 2^3 = 12.5
  - Ex) 25.0은?

부호 비트	지수 필드	가수 필드

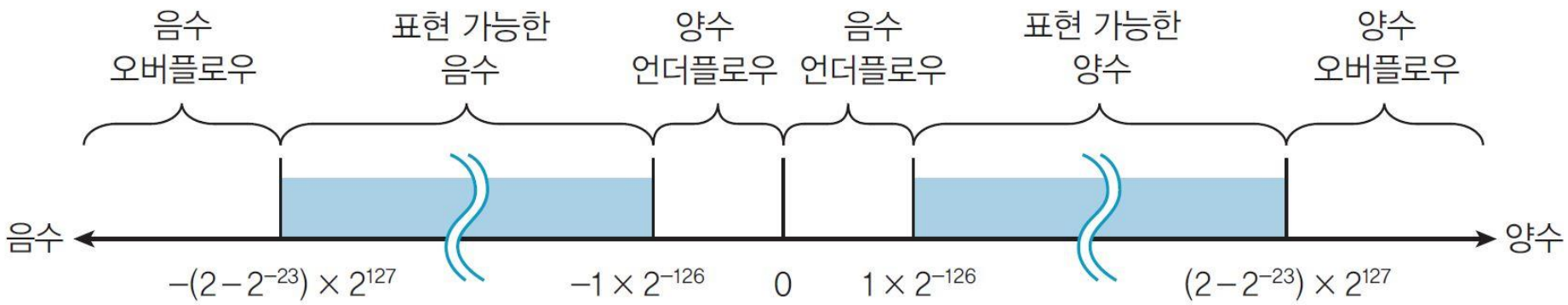


< 그림 5-6 단일 정밀도와 2배 정밀도 >

실수형(2)

- 관련 연산
  - 사칙연산(+,-,*,/), 비교연산(<, >, ==, !=)

- 부동소수점의 정밀도
  - 정밀도
    - 유효숫자를 정확히 저장할 수 있는 정도
    - 배정도(Double precision)의 경우 범위 뿐만 아니라 정밀도도 큼
  - 오버플로우(Overflow)와 언더플로우(Underflow)



실수형(3)

- 정밀도 제한 문제
- 10진수 실수는 2진수로는 근사값으로 표현됨
  - 실수끼리 또는 실수와 정수간의 등호 비교는 부정확
- Ex) $0.1 + 0.2 \neq 0.3\ (= 0.30000000000000004)$
  - $0.1 = (-1)^0 * 00111101110011001100110011001101_{(2)} * 2^{-4}$
  - $0.2 = (-1)^0 * 00111110010011001100110011001101_{(2)} * 2^{-3}$
  - $0.1 + 0.2 = \ 0.30000000000000004 = (-1)^0 * 00111110100110011001100110011010_{(2)} * 2^{-2}$

```python
#python

num1 = 0.1
num2 = 0.2
sum = num1 + num2
print(f"sum = {sum}")
print(sum == 0.3)
```

```
sum = 0.30000000000000004
False
```

논리형

- 논리형 또는 불린형(Boolean Type)
  - true와 false라는 두 개의 값으로 구성

- 관련 연산
  - 논리연산(and, or, not, &&, ||, !)

- 정수형과 호환
  - C, C++, Python, JavaScript 등 일부 언어에서는 정수형 연산이 가능 (0/1 ↔ false/true)
    - Ex) Python (True == 1)

문자형

- 문자형(Character Type)
- 하나의 문자를 표현
  - 일반적으로 1B (ASCII 코드)or 2B (Unicode 코드)으로 구분
- 실제로 숫자로 저장되나, 프로그램에서 사용할 때 해당 코드의 문자로 인식

- ASCII(American Standard Code for Information Interchange) 코드
- 미국 국립 표준 연구소(ANSI)가 제정한 정보 교환용 미국 표준 코드
- 1바이트로 구성

- 유니코드(Unicode)
- 전 세계의 모든 문자를 다루도록 설계된 문자 처리 방식
  - 가장 많이 쓰이는 다국어 기본 평면(BMP) 2바이트
- 한글은 U+로 시작 (U+AC00 : “가”)
  - 주로 U+AC00 ~ U+DA7F에 존재

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

코드

- 코드(Code) or 부호
  - 문자를 컴퓨터에 저장하는 방식
  - Ex) Binary Coded Decimal(BCD) / Excess-3 Code / etc...

- 그레이 코드(Gray code)
  - 연속된 값들 간의 코드 차이는 1비트
  - 데이터 전송시 오류 발생 확률 감소

정수	2진수	그레이 코드
0	$000_{(2)}$	000
1	$001_{(2)}$	001
2	$010_{(2)}$	011
3	$011_{(2)}$	010
4	$100_{(2)}$	110
5	$101_{(2)}$	111
6	$110_{(2)}$	101
7	$111_{(2)}$	100

< 코드 >

문자 코드(1)

- 문자 코드(Character code)
- 모든 문자는 숫자로 변환되어 저장됨
  - 문자 집합(Character set)과 인코딩(Encoding)방식이 필요

- ASCII(American Standard Code for Information Interchange) 코드
- 미국 국립 표준 연구소(ANSI)가 제정한 정보 교환용 미국 표준 코드
  - 영문 알파벳을 사용하는 문자 처리 방식
- 1바이트로 구성
  - 많은 글자를 표현할 수 없음

- 한글 코드
- 문자 집합 - KS X 1001, KS X 1002, …
- 인코딩 방식 - EUC-KR, CP949, MS949 …

0x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
B0A0		가	각	간	갇	갈	갉	갊	감	갑	값	갓	갔	강	갖	갗
B0B0	같	갚	갛	개	객	갠	갤	갬	갭	갯	갰	갱	갸	갹	갼	걀
B0C0	걋	걍	걔	걘	걜	거	걱	건	걷	걸	걺	검	겁	것	겄	겅
B0D0	겆	겉	겊	겋	게	겐	겔	겜	겝	겟	겠	겡	겨	격	겪	견
B0E0	겯	결	겸	겹	겻	겼	경	곁	계	곈	곌	곕	곗	고	곡	곤
B0F0	곧	골	곪	곬	곯	곰	곱	곳	공	곶	과	곽	관	괄	괆	

< KS X 1001 >

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

< ASCII Code >

문자 코드(2)

- 유니코드(Unicode)
- 전 세계의 모든 문자를 다루도록 설계된 문자 처리 방식
  - Unicode Consortium에서 제정
  - 총 21비트
- 평면(Plain) 단위로 구별
  - 가장 많이 쓰이는 다국어 기본 평면(BMP): 16비트로 표현 가능
- 한글은 U+로 시작 (U+AC00 : “가”)
  - 주로 U+AC00 ~ U+DA7F에 존재

다국어 기본 평면 BMP		다국어 보충 평면 SMP		상형 문자 보충 평면 SIP		상형 문자 제3 평면 TIP	특수 목적 보충 평면 SSP
U+0000	U+8000	U+10000	U+18000	U+20000	U+28000	U+30000	U+E0000
U+1000	U+9000	U+11000		U+21000	U+29000	U+31000	
U+2000	U+A000	U+12000		U+22000	U+2A000		
U+3000	U+B000	U+13000	U+1B000	U+23000	U+2B000		
U+4000	U+C000	U+14000		U+24000	U+2C000		
U+5000	U+D000		U+1D000	U+25000	U+2D000		
U+6000	U+E000	U+16000	U+1E000	U+26000	U+2E000		
U+7000	U+F000	U+17000	U+1F000	U+27000	U+2F000		

< Unicode Plain >

U+	0	1	2	3	4	5
AC00	가	각	갂	갃	간	갅
AC10	감	갑	값	갓	갔	강
AC20	갠	갡	갢	갣	갤	갥
AC30	갰	갱	갲	갳	갴	갵
AC40	걀	걁	걂	걃	걄	걅
AC50	걔	걕	걖	걗	개	객
AC60	걠	걡	걢	걣	갱	갶

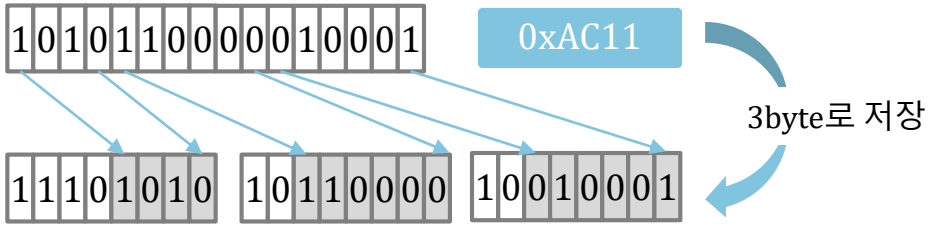
< Unicode >

문자 코드(3)

- 인코딩 방식
- 유니코드를 메모리에 저장하는 방식
  - 대표적으로 UTF-8, UTF-16
- UTF-8
  - 가장 많이 쓰이는 인코딩 방식
  - 기존 ASCII 코드와 호환성 유지(ASCII 문자의 MSB가 0)
- UTF-16
  - BMP(기본 평면) 문자는 16비트단위로 저장
  - Ex) Java의 기본 인코딩 방식

코드 범위	UTF-8	설명
0x0 ~ 0x7F	0xxxxxxx	ASCII와 동일
0x80 ~ 0x7FF	110xxxxx 10xxxxxx	첫 바이트는 110 or 1110 / 나머지는 10으로 시작
0x800 ~ 0xFFFF	1110xxxx 10xxxxxx 10xxxxxx	

< UTF-8 >



< UTF-8 예제 >

열거형

- 열거형(Enumeration Type)
- 가질 수 있는 모든 값들은 타입 정의에서 정해지는 이름 상수
- 가독성을 향상시키기 위해 사용

Ex) C

```
enum day {SUN, MON, TUE, WED, THU, FRI, SAT};
// 열거형 정의

enum day currentDay;
//열거형 변수  선언

currentDay = MON;   // 열거형 변수 사용 예
currentDay = 1;
```

Ex) Pascal

```
type
  DayOfWeek = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
// 열거형 정의

var
  today : DayOfWeek;
//열거형 변수  선언

today := Wed;   // 열거형 변수 사용 예
```

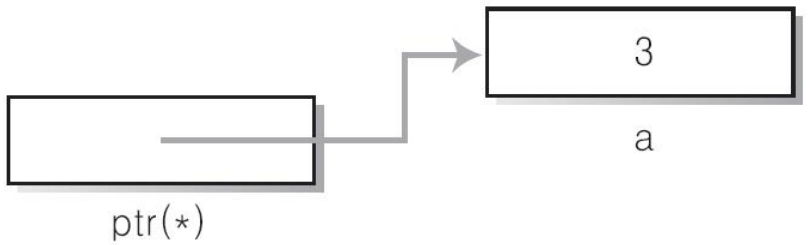
포인터형(1)

- 포인터형(Pointer Type)
- 변수가 임의의 객체를 참조하기 위해 메모리 주소를 값으로 갖는 타입
- 기억장소의 동적 관리 목적

- 관련 연산
- 포인터 선언 및 접근
  - Ex) C: *
- 변수의 주소를 반환
  - Ex) C: &
- 변수의 주소로 구조체를 접근
  - Ex) C: ->

Ex) C

```
int a=3;
int *ptr;
ptr = &a;
```

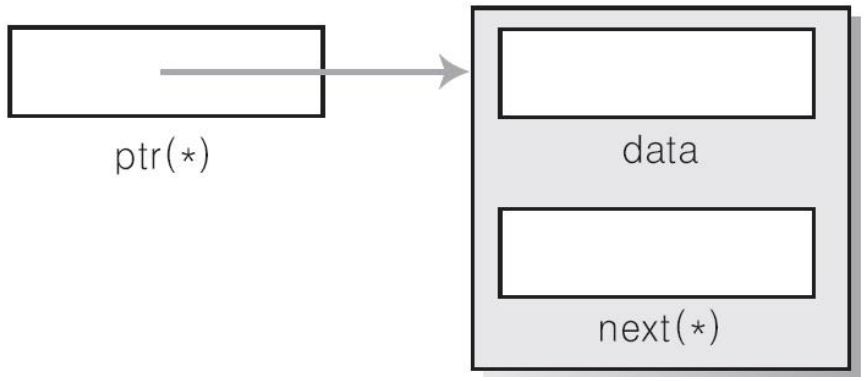


포인터형(2)

- 포인터형의 크기
- 포인터형 변수는 메모리 주소를 저장하므로, 모든 포인터형 변수의 크기는 동일
  - * 연산자 앞의 데이터 타입은 해당 변수가 가리키는 메모리 주소의 데이터 타입
  - Ex) int *ptr; long *ptr, struct list *ptr; // 모든 포인터형 변수의 크기는 동일
  - 변수가 임의의 객체를 참조하기 위해 메모리 주소를 값으로 갖는 타입

Ex) C

```
struct list {
   int data;
   struct list *next;
};

struct list *ptr;

ptr = (struct list *)malloc(sizeof(struct list));
ptr->data = 10;
```



포인터형(3)

- 주소(Address)
- 메모리의 특정한 영역에 부여된 시작주소
  - 모든 메모리 영역은 바이트(Byte) 단위의 고유한 주소를 가짐
- N Byte 메모리에는 $\log_2 N$비트의 주소 저장 공간이 필요
  - $M = \log_2 N$, $N = 2^M$
  - Ex) 4GB($= 2^{32}$B) 는 32 bit(4B) 주소 레지스터가 필요
  - Ex) 8B면 2^{64}B까지 지원
  - 현대 대부분의 컴퓨터는 64 비트 레지스터를 지원하지만, 실제 여러 이유로 메모리는 이보다 적은 크기로 한정됨

포인터형(4)

- Ex) long은 8B크기이며, 메모리는 4GB로 가정

```
Ex) C
void func() {
   long a;    // a는 func 호출시 0x8010_0004 주소에 바인딩
              // 크기가 8B이므로, 0x8010_0004~0x8010_000B까지 할당

   long *ptr_a; // ptr_a는 func 호출시 0x8010_000C 주소에 바인딩
             // 크기가 4B이므로, 0x8010_000B~0x8010_000F까지 할당
             // 지역 변수들은 함수 호출시 주소가 최종 결정됨

   a = 0x12345678;   // 0x8010_0004~에 0x12345678이 저장

   ptr_a = &a;   // 0x8010_000C에 0x8010_0004이 저장됨
}
```

주소	값
0x8010_0000	
0x8010_0001	
0x8010_0002	
0x8010_0003	
0x8010_0004	0x78
0x8010_0005	0x56
0x8010_0006	0x34
0x8010_0007	0x12
0x8010_0008	0
0x8010_0009	0
0x8010_000A	0
0x8010_000B	0
0x8010_000C	0x04
0x8010_000D	0x00
0x8010_000E	0x10
0x8010_000F	0x80

주소	값
0xA030_0010	
0xA030_0011	
0xA030_0012	
0xA030_0013	
0xA030_0014	
0xA030_0015	
0xA030_0016	
0xA030_0017	
0xA030_0018	
0xA030_0019	
0xA030_001A	
0xA030_001B	
0xA030_001C	
0xA030_001D	
0xA030_001E	
0xA030_001F	

< 4GB 메모리 >

포인터형(5)

- Ex) byte은 1B크기이며, 메모리는 4GB로 가정

```
Ex) C
void func1() {
   byte b;    // b는 func1 호출시 0xA030_00014 주소에 바인딩
              // 크기가 1B이므로, 0xA030_0014만 할당

   byte *ptr_b; // ptr_b는 func1 호출시 0xA030_0018 주소에 바인딩
             // 크기가 4B이므로, 0xA030_0018~0xA030_001B까지 할당
             // 지역 변수들은 함수 호출시 주소가 최종 결정됨

   b = 0x11;  // 0xA030_0014에 0x11이 저장

   ptr_b = &b;  // 0xA030_0018에 0xA030_0014이 저장됨
}
```

주소	값
0x8010_0000	
0x8010_0001	
0x8010_0002	
0x8010_0003	
0x8010_0004	0x78
0x8010_0005	0x56
0x8010_0006	0x34
0x8010_0007	0x12
0x8010_0008	0
0x8010_0009	0
0x8010_000A	0
0x8010_000B	0
0x8010_000C	0x04
0x8010_000D	0x00
0x8010_000E	0x10
0x8010_000F	0x80

주소	값
0xA030_0010	
0xA030_0011	
0xA030_0012	
0xA030_0013	
0xA030_0014	0x11
0xA030_0015	
0xA030_0016	
0xA030_0017	
0xA030_0018	0x14
0xA030_0019	0x00
0xA030_001A	0x30
0xA030_001B	0xA0
0xA030_001C	
0xA030_001D	
0xA030_001E	
0xA030_001F	

< 4GB 메모리 >

참조형(1)

- 참조형(Reference Type)
- 문맥에 따라 다음의 2가지 의미로 사용
- 변수의 별명(Alias)
  - 변수 자체를 가리키는 또 다른 이름
  - Ex) C++, Ada, Swift 등
- 객체의 참조(Reference)
  - 객체를 간접적으로 다루는 자료형
  - 객체 자체는 별도로 저장(주로 힙(Heap))
  - Ex) Python, Java, Ruby 등

- 관련 연산
- 참조 연산자: 별명을 만드는 연산
  - Ex) C++: &
- 비교 연산: 값 비교와 참조 비교를 구분
  - Ex) Java, Python 등

Ex) C++

```
int a = 10;
int &r = a;   // r은 a의 별명
r = 20;       // a도 20으로 바뀜
```

Ex) Java

```
int[] arr = {1, 2, 3};  // arr은 {1,2,3}을 가리키는 변수
```

Ex) Python

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print(a == b)   # True  (값이 같음)
print(a is b)     # False (다른 객체)

print(a == c)   # True  (값이 같음)
print(a is c)     # True  (같은 객체)
```

참조형(2)

- 포인터형의 문제
- 메모리 누수(Memory Leak)
  - 프로그래머에 의해서 수동으로 할당된 메모리 영역은 반드시 명시적으로 해제 필요
  - 버그가 있을 경우 동적으로 할당된 영역을 회수하지 못해 메모리가 낭비
- 안전성 문제
  - 이미 해제된 메모리를 다시 참고하거나(Dangling pointer), 다른 메모리 영역을 침범(Buffer overflow)

- 참조형 vs 포인터형
- 참조형은 주소 연산과 메모리 관리 책임을 언어/런타임이 대신 처리
  - 가비지 컬렉션(GC)으로 대부분 자동 해제하여 메모리 누수 문제 해결
- 프로그래머는 객체의 참조만 다루고, 내부의 참조값(주소)은 직접 제어 불가
  - 안전성 확보

배열(1)

- 배열(Array)
- 동질형 데이터의 모임
- 배열의 원소(Element)는 배열 이름과 첨자(Subscript)에 의해 참조
  - 첫번째 원소의 상대적 위치로 원소를 식별

- 배열 선언
- 이름, 크기, 차원, 데이터 타입을 정의
  - Ex) C/C++, Java: int A[5];
  - Ex) Ada: A: array (-2..2) of integer

- 배열 연산자
- 배열을 선언하거나 첨자를 나타내는 연산자
  - Ex) C/C++, Java: [], FORTRAN/Ada: ()

배열(2)

- 배열 차원(Dimension)
  - 배열의 원소 접근에 사용되는 첨자의 개수
    - Ex) FORTRAN: INTEGER I(5, 2) // 2차원 배열

A[2][2]	
1	2
3	4

< 행우선 저장 >

- 다차원 배열
  - 차원의 크기가 2이상인 배열
    - Ex) C/C++: int A[3][2];
  - 다차원 배열의 저장 방식
    - 행우선 저장: 하나의 행에 모두 저장한 후에 다음 행에 저장
    - 열우선 저장: 하나의 열에 모두 저장한 후에 다음 행에 저장
  - 다차원 배열의 구현 방식
    - 연속 메모리 모델(True Multidimensional Array): 물리적으로 연속된 공간을 할당
    - 배열의 배열 모델(Array of Arrays, Jagged Array): 상위 배열의 각 원소가 다른 배열에 대한 참조로 구현

A[2][2]	
1	3
2	4

< 열우선 저장 >

배열(3)

- 배열 시작 첨자
- 배열의 시작 첨자를 0부터 사용
  - Ex) C/C++, Java 등 대다수의 언어
- 배열의 시작 첨자를 1부터 사용
  - Ex) FORTRAN, Matlab, R, Julia 등
- 배열의 시작 첨자를 지정가능
  - Ex) Ada: A: array (-2..2) of integer

Ex) R

```
m <- array(1:6, dim = c(2, 3))

print(m)
print(m[1,2])
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[1] 3
```

Ex) Ada

```
procedure Custom_Index_Array is
  type My_Array is array (5 .. 7) of Integer;

  A : My_Array := (5 => 10, 6 => 20, 7 => 30);
begin
  for I in A'Range loop
    Put_Line("A(" & Integer'Image(I) & ") = " &
Integer'Image(A(I)));
  end loop;
end Custom_Index_Array;
```

```
A( 5) = 10
A( 6) = 20
A( 7) = 30
```

배열(4)

- 연속 메모리 모델 (True Multidimensional Array)
- 모든 원소가 메모리에 연속적으로 저장
  - Ex) C/C++, Ada, FORTRAN 등
  - 모든 배열은 1차원 배열처럼 취급 가능
- 2차원 배열(행주소 우선)의 주소 계산
  - Ex) C: A[i][j]의 주소 = A[0][0]의 주소 + i * 한 행의 크기 * 데이터 타입의 크기 + j * 데이터 타입의 크기
  - Ex) C: A[i][j]의 주소 == A[i*한 행의 크기 + j]

Ex) C/C++

```
int A[2][3] = {{1,2,3}, {4,5,6}};
int *p = A;
for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 3; j++) {
      printf("A[%d][%d] = %d, 주소 = %p\n",
         i, j, A[i][j], &A[i][j]);
      }
   }
printf("A[1][1] = p[1*3 + 1] = p[4] = %d\n", p[4]);
```

```
A[0][0] = 1, 주소 = 0x8000
A[0][1] = 2, 주소 = 0x8004
A[0][2] = 3, 주소 = 0x8008
A[1][0] = 4, 주소 = 0x800C
A[1][1] = 5, 주소 = 0x8010
A[1][2] = 6, 주소 = 0x8014
A[1][1] = A[1*3 + 1] = A[4] = 5
```

배열(5)

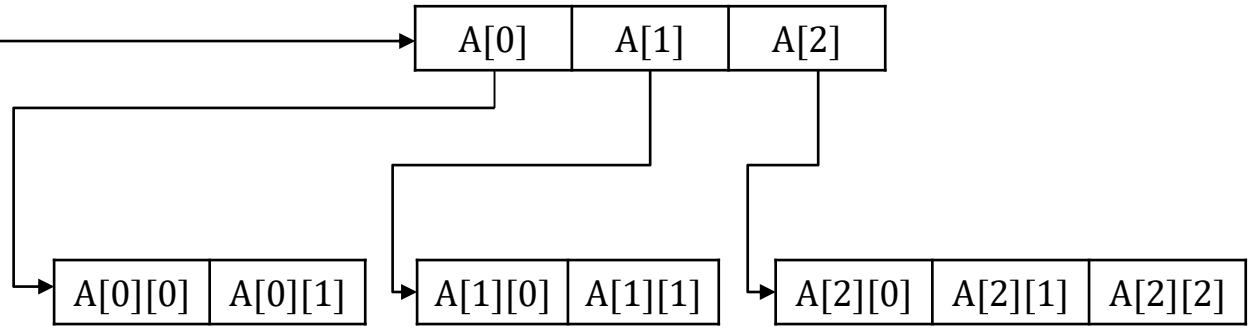
- 배열의 배열 모델(Array of Arrays, Jagged Array)
- 다차원 배열은 1차원 배열들의 배열로 구현
  - Ex) Java, Python, JavaScript 등 (C/C++에서 포인터의 배열과 유사)
  - 1차원 배열만 물리적으로 연속이고, 상위 차원은 하위 차원 배열들의 참조로 이루어진 1차원 배열
- 각 차원마다 배열의 크기를 다르게 구성할 수 있음

Ex) Java

```
int[][] A = new int[3][];
A[0] = new int[2];  // 길이 2
A[1] = new int[2];  // 길이 2
A[2] = new int[3];  // 길이 3
```

Ex) C

```
int* A[3];        // 포인터 배열
A[0] = malloc(2 * sizeof(int));
A[1] = malloc(2 * sizeof(int));
A[2] = malloc(3 * sizeof(int));
```



레코드(1)

- 레코드(Record)
- 집합체의 원소를 이름으로 식별하는 데이터들의 모임
  - 각 원소를 필드(Field)라고 함
- 배열과 다르게 다른 데이터 타입의 변수들로 구성이 가능하며, 숫자가 아닌 이름으로 접근
  - Ex) Pascal: record, C/C++: struct

Ex) Pascal

```
type
   student = record
     name: packed array[1..20] of char;
     number: integer;
     address: packed array[1..30] of char
   end;
// name, number, address 데이터를 하나의 레코드로 묶음

var A: student;  // student 타입의 변수 A 선언
```

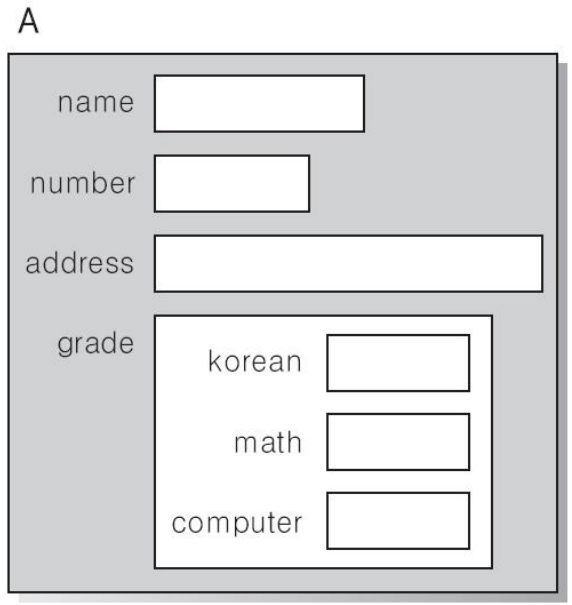
Ex) C/C++

```
 struct student {
   char name[20];
   int number;
   char address[30];
 };

struct student A;
```

레코드(2)

- 필드 연산자
- 레코드의 필드에 접근하는 연산자
  - Ex) C/C++, Pascal: . // A.number = 10;
  - Ex) COBOL: OF // number of A

- 중첩된 레코드
- 다른 레코드도 필드로 구성 가능
  - 중첩된 레코드는 필드 연산자로 연결하여 접근 가능
  - Ex) C: A.grade.math



공용체

- 공용체(Union)
- 모든 필드가 같은 메모리를 공유하면서 필요로 따라 한 필드만 사용 가능
  - 같은 데이터를 다른 데이터 타입으로 접근 가능
  - Ex) C/C++, Rust, Pascal, Ada 등

Ex) C

```
union Data {
   int i;
   char c;
};

int main() {
   union Data data;

   data.i = 10;
   printf("data.i = %d\n", data.i);    // 10
   data.c = 'A';
   printf("data.i = %d\n", data.i);    // 65 ('A'의 아스키코드)
}
```

Ex) Ada

```
type Shape (Kind : Character) is record
  case Kind is
    when 'C' => Radius : Float;
    when 'R' => Width, Height : Float;
  end case;
end record;
```

연관 배열(1)

- 연관 배열(Associative Array)
- 키 집합 K에서 값 집합 V로의 매핑을 제공하는 자료형
  - 키(Key)로 값(Value)을 저장
  - Ex) Python: dict, C++: map, Java, Map<K,V>, Lua: table 등
- 다른 복합 데이터 타입과의 차이
  - 배열은 인덱스가 정수형이지만, 연관 배열은 의미 있는 키를 인덱스로 사용
  - 구조체는 필드 이름은 미리 정의된 것만 사용하나, 연관 배열은 접근 이름을 동적으로 변경 가능
- 장단점
  - 키 추가 및 삭제가 자유로워 가독성과 유연성이 높음
  - 해시(Hash)/트리(Tree) 기반 구현으로 배열보다 접근 속도가 느릴 수 있음

연관 배열(2)

- 관련 연산
- 삽입(Insert/Put)
  - A[k]←v: 키 k에 값 v를 저장
- 검색(Lookup/Get)
  - v=A[k] : 키 k에 해당하는 값 v를 반환
- 탐색(Iteration)
  - 모든 키 집합 또는 (키, 값) 쌍을 순회

Ex) Python

```
text = "the quick brown fox jumps over the lazy dog the fox"
words = text.split()

count = {}
for w in words:
   count[w] = 0

for w in words:
   count[w] = count[w] + 1

print(count)
```

```
{'the': 3, 'quick': 1, 'brown': 1, 'fox': 2, 'jumps': 1, 'over': 1, 'lazy': 1, 'dog': 1}
```

문자열형(1)

- 문자열형(String Type)
- 문자열은 사람들은 자주 사용하는 데이터지만 기본 데이터 타입으로 표현이 어려움
  - 문자열은 문자들의 집합이라서 단일 값으로 치환이 어려움
  - 숫자(int, float)는 단일 이진수 값으로 표현이 가능
  - 문자는 보통 1바이트(또는 유니코드 2~4바이트)로 표현 가능

- 구현 방식
  - 문자열을 문자 배열 또는 전용 타입으로 제공
  - 문자열 조작을 위한 함수 및 연산자를 언어 차원에서 지원
  - 구현 방식은 언어마다 매우 상이하므로, 새로운 언어 학습시 주의 필요

문자열형(2)

- 문자열 지원 요소
- 문자열 전용 데이터 타입
  - Ex) Java: String str = “programming”;
- 문자열 함수
  - Ex) C: <string.h> 라이브러리 (strlen(), strcpy())
- 문자열 연산자
  - Ex) Python: “abc”+”dec” -> “abcdef”
- 문자열 상수
  - 불변(Immutable) 상수로 지원

형변환(1)

- 형변환(Type Conversion)
- 하나의 데이터 타입으로 표현된 값을 다른 데이터 타입으로 바꾸는 과정
  - 서로 다른 데이터 타입의 연산이나 대입 시 형변환이 필요
- 묵시적 형변환(Implicit Conversion, Type Coercion)
  - 언어가 자동으로 데이터 타입을 변환
  - Ex) C: 정수 + 실수 → 실수
  - 코드가 간결해지지만, 의도하지 않은 변환이 일어나 오류가 발생할 여지가 있음
- 명시적 형변환(Explicit Conversion, Type Casting)
  - 강제로 데이터 타입을 바꾸는 것
  - 보통 cast 연산자 사용
  - Ex) Ruby, Haskell, OCaml, Scala 등 (강타입 언어들)

형변환(2)

- 묵시적 형변환의 문제
  - 일반적으로 산술 계산에서는 작은 타입 → 큰 타입으로 자동 승격
  - Ex) 정수+실수 → 실수
  - 변환 규칙에 따라 예상하지 않은 결과가 나올 수 있음

Ex) C

```
unsigned int x = 1;
int y = -2;

printf("y=%u\n",y);
if (y < x) {                // y가 unsigned로 자동 변환
   printf("y < x\n");
} else {
   printf("y >= x\n");
}
```

```
y=4294967294
y >= x
```

Ex) JavaScript

```
console.log(10 + 2.32);
console.log("10" + 2);
console.log("10" * 2);
```

```
12.32
102
20
```

형변환(3)

- 명시적 형변환
  - 강타입(Strong Type)은 잘못된 타입 결합을 자동으로 허용하지 않음
  - Ex) 정수+문자열은 오류, 반드시 변환 함수 사용
  - 명시적 형변환이어도 의도가 분명하지 않으면 결과가 달라짐

Ex) Haskell

```
x = 5 :: Int
y = 2.5 :: Float
-- z = x + y                  -- 오류! (암시적 변환 없음)
z = fromIntegral x + y   -- 명시적 변환 필요
```

Ex) Python

```
print("10"+3)          # 오류
print(int("10")+3)     # 13
print("10"+str(3))     # "103"
```

Ex) C++

```
double a = 3.14;
int b = 2;
cout << a + b << endl;                 // 5.14
cout << (int)a + b << endl;          // 5
cout << a + (double)b << endl;    // 5.14
```

프로그래밍언어와 컴파일러

식과 문장

식

- 식(Expression)
- 연산자(Operator)와 피연산자(Operand)로 구성되어 값을 나타내는 표현
- 프로그램이 실행 도중 평가(Evaluate)되어 **하나의 값으로 수렴**
- 단순 상수부터 복잡한 함수 호출까지 모두 포함
- Ex) 식의 예시
  - 상수 표현식: 42, 'hello', 3.14
  - 변수 참조: x, user_name
  - 함수 호출: max(3, 5), len("hello")
  - 데이터 접근: arr[0], dict['key']
- 문장(Statement)과 다름
  - 문장은 보통 명령을 수행하며, 값을 반환하지 않을 수도 있음
  - Ex) if, while, for, ...

연산자(1)

- 연산자(Operator)
- 값(피연산자)에 대해 계산 또는 처리를 수행하는 기호 또는 키워드
  - Ex) +, -, *, /, and, ==, not 등

- 피연산자의 개수
- 단항 연산자(Unary): 피연산자 1개를 처리
  - Ex) -x, not x, ~x
- 이항 연산자(Binary): 피연산자 2개를 처리
  - Ex) x + y, a > b, a and b
- 삼항 연산자(Ternary): 피연산자 3개를 처리
  - Ex) (i%2) ? "odd" : "even“

연산자(2)

- 표기법
- 중위 표기법(Infix): 피연산자 사이에 연산자 위치
  - Ex) a + b (가장 일반적인 표기법)
- 전위 표기법(Prefix): 연산자가 앞에 위치
  - Ex) + a b (함수형 언어)
- 후위 표기법(Postfix): 연산자가 뒤에 위치
  - Ex) a b + (스택 기반 계산기)

- 연산자 평가 순서
- 여러 개의 연산자로 이루어진 식에서의 평가 순서
- 괄호 > 우선순위(Precedence) > 결합 규칙(Associativity)
- 대부분의 언어가 비슷하지만, 가독성을 위해 괄호를 명시하는 것이 권장
  - Ex) a ? b : c ? d : e → a ? b : (c ? d : e)

연산자(3)

- 연산자 우선순위(Precedence)
- 우선순위가 높은 연산을 먼저 실행
- 같은 우선순위의 경우, 선행 연산자를 먼저 실행
  - Ex) 7 - 3 * 2 = 7 - (3 * 2) = 1
- 대부분의 언어가 비슷한 우선순위를 가지지만, 일부 연산자는 주의 필요
  - Ex) C는 == 가 & 보다 우선이지만 Python에서는 반대
  - Ex) Python에서는 **가 단항 -보다 우선이지만, JavaScript에서는 반대
  - 우결합 연산자(대입, 삼항, 거듭제공등), 단항 연산자, 비트 연산자, 비교 연산자는 괄호 사용을 추천

Ex) C
```
int a = 1, b = 2, c = 2;
int r = a & b == c;   // r = a & (b == c) → 1
```

Ex) Python
```
a, b, c = 1, 2, 2
r = a & b == c    # r = (a & b) == c → False
```

Ex) Python
```
-2**2   #  -(2**2) → -4
```

Ex) JavaScript
```
-2 ** 2     // SyntaxError (왼쪽에 단항 부호 결합 금지)
-(2 ** 2)  // -4
(-2) ** 2  // 4
```

연산자(4)

- 연산자 결합 규칙(Associativity)
- 좌결합(Left-to-right)
  - 같은 우선순위 연산자가 여러 개 있으면 왼쪽부터 계산
  - 대부분의 연산자가 좌결합
  - Ex) a - b - c → (a - b) - c
- 우결합(Right-to-left)
  - 같은 우선순위 연산자가 여러 개 있으면 오른쪽부터 계산
  - Ex) a = b = c → a = (b = c)
  - 대표적인 우결합 연산자: 단항 연산자, 거듭제곱(**), 대입(=), 삼항(?:)

Ex) Python

```
# Python 연산자 우선순위
# & (비트 AND) > 비교 연산자 (==) > not (논리 부정) > or (논리 OR)

a, b, c = 2, 3, 3
not a == b or a & b == c              # True
((not (a == b)) or ((a & b) == c))  # True
```

연산자(5)

❖ Ex) C의 연산자 우선순위 및 결합 규칙

순위	종류	설명	예	결합 규칙
1	()	함수 호출	func()	좌결합
	[]	배열 인덱스	a[1]	좌결합
	-> , .	구조체/포인터 멤버 접근	pointer->x	좌결합
	++ , --	후위 증감 연산	a++	좌결합
2	~ , !	부정(NOT)	0 == !1	우결합
	+ , -	양수/음수	+3, -3	우결합
	&, *	주소 참조/포인터	int *x = &y	우결합
	(type)	데이터 타입 변환	(float)y	우결합
	++ , --	전위 증감 연산	++a	우결합
3	* , / , %	산술 연산	1*3, 4/2	좌결합
4	+ , -	산술 연산	1+3	좌결합
5	<< , >>	비트 시프트(Shift)	2 >> 2	좌결합

순위	종류	설명	예	결합 규칙
6	< , > , <= , >=	비교 연산자(대소)	3 > 2	좌결합
7	==, !=	비교 연산자(동등)	3 == 2	좌결합
8	&	비트 AND	x & y	좌결합
9	^	비트 XOR	x ^ y	좌결합
10	|	비트 OR	x | y	좌결합
11	&&	논리 AND	x && y	좌결합
12	||	논리 OR	x || y	좌결합
13	? :	삼항 연산자	x ? y : z	우결합
14	=, +=, -=, ...	대입 및 복합 대입 연산자	x = y x += z	우결합
15	,	쉼표 연산자	x = y, a = b	좌결합

피연산자

- 피연산자(Operand)
  - 연산자의 대상이 되는 값, 변수, 식

- 피연산자 평가 순서
  - 실제 실행 시 어떤 피연산자가 먼저 계산되는지에 대한 규칙
    - 연산자 우선순위나 결합 규칙과는 별개의 개념
  - 대부분의 언어가 좌측 우선(Left-to-right)이지만, C/C++은 피연산자 평가 순서가 미정
    - C/C++은 컴파일러 구현에 따라 우측 피연산자가 먼저 실행 가능

```
Ex) C
int i = 1;
int x = i + i++;
// 좌측을 먼저 평가하면, x = 2 , https://rextester.com/l/c_online_compiler_gcc (clang)
// 우측을 먼저 평가하면, x = 3 , https://rextester.com/l/c_online_compiler_gcc (gcc)
```

단락 평가

- 단락 평가(Short-Circuit Evaluation)
- 모든 피연산자와 연산자를 평가하지 않고서도 식의 결과를 결정
  - Ex) true or x → x의 값에 관계없이 true
- 대부분의 언어에서 지원하나 Pascal, VB, Ada의 일부 연산자나 SQL등 일부 언어에서 미지원
- 필요 없는 피연산자를 건너뛰어 계산량 감소
  - Ex) Python: x and f(x) → x가 참일 때만 f 실행
- 오류 발생 코드가 뒤에 있어도 실행되지 않음
  - Ex) C: ptr != NULL && ptr->value == 10 → NULL 포인터 접근 방지
- 부작용(Side Effect)을 포함하는 식의 경우 주의
  - Ex) C: (x > y) || (x++ % 2)
  - 두 번째 식은 x<=y인 경우에만 평가되므로 x 값 역시 x<=y인 경우에만 증가

실습문제(1)

- 1) 다음 식은 후위 표기법(Postfix)으로 표현되고 있다. 연산 결과는?
  - 모든 연산자는 연산자를 만나면 앞의 2개의 값을 계산해서 하나의 값으로 변환

```
5 1 2 + 4 * + 3 -
```

- 2) 다음 식의 연산자가 C의 우선순위를 가질 때 어떤 순서로 평가되는가?

```
int r = x << y + 1 == z ? u = v : w += 2;
```

실습문제(1)

- 1) 다음 식은 후위 표기법(Postfix)으로 표현되고 있다. 연산 결과는?
  - 모든 연산자는 연산자를 만나면 앞의 2개의 값을 계산해서 하나의 값으로 변환

```
5 1 2 + 4 * + 3 -
```

  - A) ((5 (((1 2 +) 4 *) +) 3 -) = ((5 (3 4 *) +) 3 -)) = ((5 12 +) 3 -) = 17 3 - = 14

- 2) 다음 식의 연산자가 C의 우선순위를 가질 때 어떤 순서로 평가되는가?

```
int r = x << y + 1 == z ? u = v : w += 2;
```

  - A) (int r = (((x << (y + 1)) == z) ? (u = v) : (w += 2));)

문장

- 문장(Statement)
- 프로그램을 구성하는 실행 단위
  - 독립적으로 실행되어 프로그램의 흐름이나 상태를 변화
  - 세미콜론(;)이나 줄바꿈등으로 구분
  - 문장은 식을 포함할 수 있지만, 식이 항상 문장이 되는 것은 아님

- 문장의 종류
- 선언문/정의문
  - 이름을 도입하거나 새로운 대상을 정의하는 문장
  - Ex) int x;
- 실행문
  - 어떤 동작을 실행하는 문장 (대입, 함수 호출 등)
  - Ex) x = x +1;
- 제어문
  - 흐름을 바꾸는 문장 (조건문, 반복문, 분기문, 예외 처리등)

선언문/정의문

- 선언(Declaration)과 정의(Definition)
- 선언은 프로그램 내에서 식별자가 어떤 타입 혹은 형식을 가질 것임을 알리는 문법적 구성 요소
  - 구체적인 구현(메모리 할당, 본문, 초기값)은 포함하지 않을 수 있음
  - 선언문은 이름을 도입하는 문장
- 정의는 선언에서 약속한 형식에 따라, 실체를 구현하거나 생성하는 문법 요소
  - Ex) 메모리 할당 (변수 정의), 함수 본문 제공 (함수 정의), 클래스/모듈 내용 제공 (클래스/모듈 정의)
  - 이름에 실제 의미를 구현하는 문장
- 정의는 선언을 포함하지만, 선언은 선언만 가능

Ex) C
```
extern int a;        // 선언
int a = 10;          // 정의

int f(int x, int y) {  // 함수 정의 (함수 선언을 포함)
   return x+y;
}
```

Ex) Java
```
interface Adder {
   int add(int a, int b);   // 선언 (시그니처)
}
class MyAdder implements Adder {
   public int add(int a, int b) {   // 정의
      return a+b;
   }
}
```

실행문

- 실행(Execution)
- 식을 하나의 실행 단위로 만든 것
- 부작용(Side Effect) 발생 가능

- 부작용(Side Effect)
- 값을 계산하는 것 외에 프로그램의 상태나 외부 세계에 변화가 발생
  - Ex) x = f(y); // 변수 x값의 변경은 프로그램 상태의 변화로 부작용. 함수 f에 y가 입력되어 값이 계산되는 부분까지는 부작용이 아니지만, 그 결과가 x에 저장되는 부분은 부작용
  - Ex) x = (i++) + (i++); // x와 i가 모두 변경되어 부작용
- 이름과 달리 실제 유용한 동작(상태 업데이트, 입출력 등)은 대부분 부작용으로 구현됨
- 순수성이 깨져 예측, 검증, 디버깅이 어려우며, 병렬화 및 최적화 방해
  - 함수형 프로그래밍 기법이 도입되는 이유

조건문(1)

❖ 조건문

- 조건에 따라 둘 또는 그 이상의 실행 경로 중에서 하나를 선택할 수 있는 수단을 제공하는 문장
  - 조건이 참이냐 거짓이냐에 따라 선택하는 if 문
  - 조건에 따라 여러 경로 중 하나를 선택하는 case(or switch) 문

❖ if 문

- FORTRAN 77에서 블록 if 문 도입
  - IF (식) THEN ... ELSE ... ENDIF
- Dangling else
  - if-then-else 구문에서 else가 어느 if에 속하는지 문법적으로 애매해지는 문제
  - else는 가장 가까운 '짝이 없는(if without else)' if에 결합 (C/C++, Java, JavaScript 등)
  - begin...end 블록표시를 강제하거나 end if 같은 종결 키워드로 구문 경계를 명시(Algol, Pascal, Ada 등)

```
// Dangling else 문제
if a then
  if b then S
else T            // T는 if a의 else인가? if b의 else인가?
```

조건문(2)

- switch 문
- ALGOL W에서 처음 도입
- Fall-through 모호성
  - C/C++, Java 등은 case 블록이 끝나면 자동으로 다음 case로 흘러감(Fall-through)
  - break를 쓰지 않으면 다음 case 실행까지 이어짐
  - 이 때문에 의도된 것인지, 실수인지 문법적으로 구분 불가능
  - 암묵적 break 기본화: 각 case가 끝나면 자동으로 분기가 종료(Swift, Kotlin, Pascal, Ada 등)
  - switch 자체를 재설계: switch 결과를 값으로 반환(Scala, Rust, Haskell 등)

Ex) Ada

```
case score is
   when 90..100 => grade := 'A';
   when 80..89 => grade := 'B';
   when 70..79 => grade := 'C';
   when 60..69 => grade := 'D';
   when 0..59 => grade := 'E';
end case;
```

Ex) Rust

```
let result = match x {
   1 | 2 => "OneOrTwo",
   3     => "Three",
   _     => "Other",
};
```

반복문(1)

- 반복문
- 특정 부분을 반복해서 실행되게 하는 문장
  - while/do-while문
  - for 문
  - FORTRAN에서 DO 문으로 도입

- while/do-while 문
- while 문의 조건식이 참인 동안 문장을 반복해서 실행
- do-while 문은 본문 최소 1회 실행 보장

Ex) Ada
```
while index <= 10 loop
   result := result + index;
   index := index + 1;
end loop;
```

Ex) JavaScript
```
let input;
do {
  input = prompt("Enter a number greater than 10:");
} while (Number(input) <= 10);
console.log("Valid input:", input);
```

반복문(2)

- for 문
- 반복 변수, 조건 검사, 반복 후 갱신의 세 요소를 문법적으로 통합하여 제공
  - Ex) Kotlin: for (i in 1..5) { ... }
- for 문의 발전
  - foreach: 모든 원소를 순회한다는 의도를 코드가 바로 표현 (C# foreach, JS for…of)
  - 컴프리헨션: 새로운 시퀀스를 생성하기 위한 선언적 구문, 중간변수 및 상태 최소화 (Python 리스트 컴프리헨션)
  - 제너레이터/이터레이터: 지연 평가를 통한 대용량 스트림 및 파이프라인 처리에 최적 (C# yield return)

Ex) Kotlin (foreach)
```
val list = listOf("a", "b", "c")
for (elem in list) {
   println(elem)
}
```

Ex) Haskell (컴프리헨션)
```
[x*x | x <- [1..10], even x]
-- 결과: [4,16,36,64,100]
```

Ex) JavaScript (제너레이터)
```
function* counter() {
  let i = 0;
  while (true) {
    yield i++;
  }
}
let gen = counter();
console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
```

무조건 분기문

- 무조건 분기문 (GOTO 문)
- 프로그램의 실행 순서를 특정 위치로 바꾸는 문장
  - 어셈블리 언어의 JMP와 동일한 기능을 고급언어에 도입
- 초기언어에서 프로그램 흐름 제어에 필수였으나, for, if-else 등의 구문 등장으로 필요성 감소
- 대부분의 현대 언어에서 제거
  - 프로그램 흐름이 비구조적이어서 유지보수가 어려움
  - C/C++에 여전히 존재하지만 권장되지 않음

Ex) FORTRAN

```
      PROGRAM LOOPGOTO
      INTEGER I
      I = 1
10    IF (I .GT. 5) GOTO 100  ! I > 5면 루프 종료
      PRINT *, 'I = ', I
      I = I + 1
      GOTO 10
100   CONTINUE
      END
```

Ex) BASIC

```
10 INPUT "Enter X:"; X
20 IF X < 0 THEN GOTO 100
30 IF X = 0 THEN GOTO 200
40 PRINT "Positive"
50 GOTO 300
100 PRINT "Negative"
110 GOTO 300
200 PRINT "Zero"
300 PRINT "Done"
310 GOTO 10
```

예외 처리(1)

- 예외(Exception)
- 비정상적인 사건
  - Ex) 프로그램 실행 중의 오버플로나 언더플로, 0으로 나누기, 배열 첨자 범위 이탈 오류, EOF(end-of-file) 조건 등

- 예외 처리(Exception Handling)
- 예외가 탐지되었을 때 프로그램의 중단 없이 다시 정상적으로 실행되도록 하는 기능
  - 예외를 처리하는 부분을 예외 처리기(Exception Handler)라고 함
  - 인터럽트 처리 기능을 프로그래밍 언어에서 흉내낸 것
- 오류를 일반 제어 흐름과 분리하여 핵심 부분과 오류 처리 부분을 명확히 구분
  - 예외 처리가 없으며 프로그래머가 예외에 대한 검사를 사전에 수행해야 함
  - 예외에 대한 조건 처리와 정상적인 조건 처리를 구분하기가 힘듦
- PL/I나 CLU에서 도입되었으며, Ada에서 본격적으로 표준화
  - Ex) C++, Python, Swift,Haskell 등 대부분의 현대 언어에서 지원

예외 처리(2)

- Ada 예외 처리 예제

```
procedure Main is
  procedure Division_Test is
    X, Y, Z : Integer;
  begin
    X := 10;  Y := 0;
    Z := X / Y;  -- 예외 발생후 프로그램 종료
    Put_Line("Result = " & Integer'Image(Z));
  end Division_Test;
begin
  Division_Test;
  declare
    A, B : Integer := 5;
  begin
    Put_Line("A + B = " & Integer'Image(A + B));
  end;
end Main;
```

```
raised CONSTRAINT_ERROR
```

```
procedure Main is
  procedure Division_Test is
    X, Y, Z : Integer;
  begin
    X := 10;  Y := 0;
    Z := X / Y;
    Put_Line("Result = " & Integer'Image(Z));
 exception   -- 예외 처리후 정상적으로 프로그램 실행
   when Constraint_Error =>
    Put_Line("예외 발생: 0으로 나눌 수 없습니다.");
  end Division_Test;
begin
  Division_Test;
  declare
    A, B : Integer := 5;
  begin
    Put_Line("A + B = " & Integer'Image(A + B));
  end;
end Main;
```

```
예외 발생: 0으로 나눌 수 없습니다.
10
```

구조적 프로그래밍(1)

- 구조적 프로그래밍(Structured Programming)
- 제어 흐름을 잘 정의된 구조적 제어문(순차, 선택, 반복)만으로 표현하는 프로그래밍 패러다임
- “Go To Statement Considered Harmful” (CACM, 1968)
  - Edsger W. Dijkstra가 발표한 에세이
  - GOTO 사용은 프로그램의 구조적 이해를 방해 → 프로그램은 순차, 선택, 반복 세 가지 구조만으로 작성을 주장
  - 구조적 프로그래밍 운동을 촉발하여 후대 언어 설계(Java, Pascal, Ada 등)에 지대한 영향
- 특징
  - 제어 흐름 제한: GOTO 대신 if–then–else, while, for 같은 구조적 구문 사용
  - 블록 구조: 코드가 중첩 블록으로 구성 → 지역 변수, 스코프 명확
  - 정형적 검증(Formal Verification) 용이: 제어 흐름이 예측 가능 → 프로그램의 수학적 증명, 논리적 추론이 가능
  - 가독성과 유지보수성 향상: 프로그램 논리가 트리(Tree)나 그래프(Graph) 구조로 표현 가능 → 디버깅 용이

구조적 프로그래밍(2)

- 구조적 프로그래밍(Structured Programming)

Ex) BASIC

```
10 LET SUM = 0
20 INPUT "Enter number: ", N
30 IF N < 0 THEN GOTO 60
40 LET SUM = SUM + N
50 GOTO 20
60 PRINT "Final Sum = "; SUM
```

Ex) Python

```
total = 0

while True:
    n = int(input("Enter number: "))
    if n < 0:
        break
    total += n

print("Final Sum =", total)
```

프로그래밍언어와 컴파일러

부프로그램

부프로그램(1)

- 부프로그램(Subprogram)
- 프로그래밍 언어에서 재사용 가능한 코드의 묶음
- 부프로그램 선언
  - 부프로그램의 인터페이스를 명시
  - 컴파일러/인터프리터에 이름, 매개변수, 반환형등의 정보를 전달
  - 부프로그램의 머리부(Header)
- 부프로그램 정의
  - 선언된 부프로그램의 본체(Body)를 구현
- 부프로그램 호출
  - 부프로그램을 실행하라는 명령
- 매개변수(Parameter) or 형식 매개변수
  - 부프로그램에서 입력값을 받기 위해 선언된 변수
  - 함수의 헤더(Header) 안에 나타남
- 인수(Argument) or 실매개변수
  - 호출자가 함수를 호출할 때 실제로 전달하는 값

Ex) Pascal

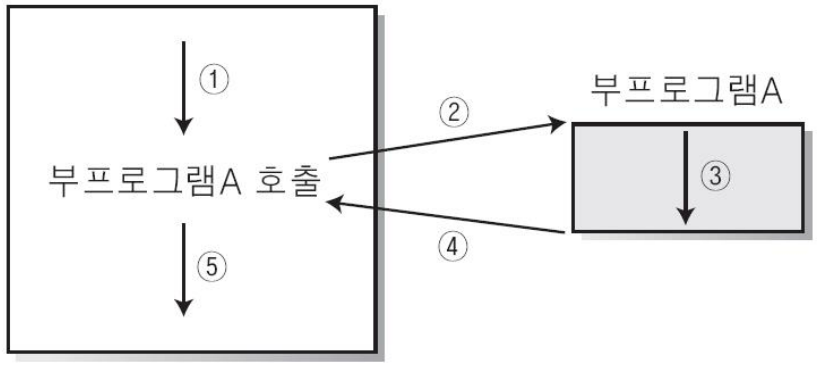
```
function Add(x, y: Integer): Integer; {선언}
{ x, y 는 매개변수 }

function Add(x, y: Integer): Integer; { 정의 }
begin
  Add := x + y;
end;

begin
  writeln(Add(3, 5)); { 호출: 3, 5는 인수 }
end.
```

인자

- 매개변수와 인수를 모두 포괄하는 용어
- "add 함수의 인자는 int i" → 매개변수를 의미
- "add 함수로 전달된 인자는 3" → 인수를 의미

부프로그램(2)

- 부프로그램의 실행 순서
- 1) 부프로그램 호출
- 2) 현재 상태 저장
  - 매개변수, 지역변수, 레지스터 내용, 복귀 주소(Retrun Address) 등
  - 복귀 주소는 부프로그램 호출 명령 다음 주소
- 3) 부프로그램 실행
  - 부프로그램의 시작 주소로 분기(Branch)
- 4) 반환값 전달 및 원래 상태로 복귀
- 5) 원래 프로그램 수행
  - 저장된 복귀 주소로 분기(Branch)



부프로그램(3)

- 부프로그램의 구성
- 코드부(Code Segement)
  - 부프로그램의 명령어
  - 일반적으로 프로그램 수행 중 변화하지 않음
- 활성 레코드(Activation Record)
  - 부프로그램 수행에 필요한 정보를 저장하는 공간
  - 매개변수, 지역변수, 정적/동적 링크, 복귀 주소, 반환값 등을 저장
  - 호출시 내용이 동적으로 변화

부프로그램(4)

- 부프로그램의 분류
- 함수(Function)
  - 입력된 값을 바탕으로 연산을 수행하여 값을 반환하는 부프로그램
  - 수학적 함수 개념과 유사
- 프로시저(Procedure)
  - 명시적 반환값이 없으며, 부작용(Side Effect) 발생을 목적으로 호출
  - 특정 작업(입출력, 화면 출력, 파일 처리 등)을 수행하는 것이 목적
  - 초기 언어에서는 서브루틴(Subroutine)이라고 함
- Ada, FORTRAN등 일부 언어에서는 함수와 프로시저를 문법적으로 구분
  - 대부분 현대 언어에서는 함수와 프로시저를 구분하지 않고, 프로시저를 반환값이 없는 함수로 구현

Ex) Ada
```
function Add(X,Y:Integer) return Integer;  -- 함수
procedure PrintSum(X,Y:Integer);  -- 프로시저
```

Ex) FORTRAN
```
function add(x,y) result(r)  ! 함수
subroutine printSum(x,y)    ! 서브루틴 (프로시저)
```

Ex) C
```
int add(int x, int y);  // 함수
void printx(int x);    // 함수지만 프로시저 역할
```

중복 부프로그램

- 중복 부프로그램(Overloaded Subprogram)
- 이름을 동일하지만 매개변수의 매개변수 시그니처(Parameter Signature) - 개수, 순서, 자료형 - 이 다른 복수의 부프로그램
  - 호출 시점에 제공된 인수의 유형과 개수에 따라 정적으로 적절한 부프로그램이 선택
  - Ex) C++, C#, Swift, Java, Ada, Scala 등 (Python, JavaScript, Ruby 등은 지원 안함)
- 기본값을 사용시 주의 필요

Ex) Swift

```
func greet(name: String) {
   print("Hello, \(name)!")
}
func greet(times: Int) {
   for _ in 1...times {
      print("Hello!")
   }
}
greet(name: "Alice")   // → Hello, Alice!
greet(times: 3)        // → Hello! (3번 출력)
```

Ex) Swift

```
func greet(name: String, times: Int = 1) {
   for _ in 1...times {
      print("Hello, \(name)!")
   }
}
func greet(name: String) {
   print("Hi, \(name)!")
}

greet(name: "Alice")     // 어떤 함수가 호출될지 혼란
```

포괄 부프로그램

- 포괄 부프로그램(Generic Subprogram)
- 다양한 타입의 매개변수를 허용하는 부프로그램
  - 호출 시점에 서로 다른 자료형 인수 사용 가능
  - 동일한 의미적 동작을 하는 연산을 동일한 이름으로 다양한 자료형에 맞게 제공하여 가독성과 일관성 향상
  - Ex) C++, Java, Swift, Ada 등 (Python, JavaScript, Ruby 등은 지원 안함)

Ex) Java

```
public class Main {
   static <T extends Number> double add(T a, T b) {
      return a.doubleValue() + b.doubleValue();
   }

   public static void main(String[] args) {
      System.out.println(add(3, 5));       // int → 8.0
      System.out.println(add(2.5, 4.1));   // double → 6.6
      System.out.println(add(10L, 20L));   // long → 30.0
   }
}
```

Ex) Go

```
func Add[T int | float64](a, b T) T {
   return a + b
}

func main() {
   fmt.Println(Add(3, 5))       // int → 8
   fmt.Println(Add(2.5, 4.1))   // float64 → 6.6
}
```

매크로 함수

- 매크로 함수(Macro Function)
- 함수 호출이 아닌 소스 코드 치환을 통해 구현
  - 전처리기(Preprocessor)에 의해 처리되는 매크로 정의의 한 형태
  - Ex) C/C++의 #define, Lisp의 defmacro, Rust의 macro_rules! 등
- 함수 호출은 메모리 할당 등의 오버헤드가 발생
  - 짧은 길이의 함수는 아예 매크로 함수로 만드는 것이 효율적일 수 있음
- 단순 텍스트 치환이므로 의도와 다르게 적용 가능하므로 사용에 주의 필요
  - 특히, 인수에 연산자가 포함된 경우 주의

Ex) Lisp

```
(defun square-fn (x)
  (* x x))

(print (square-fn (+ 1 2)))    ; 9
```

Ex) C

```
#define SQUARE(x) x * x

void main() {
   int i = 3;
   printf("%d\n", SQUARE(i + i));
   // 의도 6 *6 , 결과 : 3 + 3 * 3 + 3 = 15
}
```

인라인 함수

- 인라인 함수(Inline Function)
- 컴파일러가 함수 본문을 호출 지점에 직접 삽입
  - 매크로와 달리 컴파일 단계에서 처리
  - Ex) C/C++의 inline, Ada의 pragma Inline, Fortran의INLNE 등
  - 대부분의 현대 언어들은 컴파일러에서 자동 지원
- 매크로 함수와 달리 일반 함수처럼 타입 검사, 스코프 규칙, 인수 평가 등을 지원
  - 되도록 인라인 함수로 사용

Ex) Rust

```
#[inline]           // Inlining을 권장하는 힌트 키워드
fn square(x: i32) -> i32 {
   x * x
}

fn main() {
   let n = 3;
   println!("square = {}", square(n));
}
```

Ex) C

```
inline int SQUARE(int x) { return x * x; }

void main() {
   int i = 3;
   printf("%d\n", SQUARE(i + i));
   // 의도대로 6 *6
   printf("%d\n", SQUARE(i++));
   // 의도대로  3 * 3
```

매개변수 대응 방식(1)

- 위치에 의한 대응
- 매개변수의 위치 순서에 따라 매개변수와 대응
  - Ex) C/C++, Java, Pascal 등 대부분의 언어에서 지원

- 키워드에 의한 대응
- 매개변수의 이름을 명시하여 대응하고, 순서는 고려하지 않음
  - Ex) Ada, Fortran 90, Python, Swift, R 등
- 일반적으로 일부는 위치로, 일부는 키워드로 지정 가능한 혼합 방식도 지원

Ex) Ada
```
procedure Print_Sum(X, Y : Integer) is
begin
   Put_Line(Integer'Image(X + Y));
end Print_Sum;

Print_Sum(Y => 5, X => 3);   -- 순서 바꿔도 X=3, Y=5
```

Ex) Fortran 90
```
subroutine printSum(x, y, z)
   integer :: x, y, z
   print *, x + y + z
end subroutine

! 호출
call printSum(1, z=3, y=2)   ! x=1, y=2, z=3
```

매개변수 대응 방식(2)

- ❖ 매개변수 기본값(Default Value)
- 매개변수에 디폴트 값을 부여하여 호출 시 해당 인자 생략 가능
  - Ex) C++, Python, Ruby, PHP, JavaScript 등 대부분의 언어
  - Ex) 고전 C, Java, Go, Rust, Haskell 등에서는 직접적으로는 사용 불가

- ❖ 가변 매개변수(Variadic Parameter)
- 매개변수의 개수가 가변적일 수 있도록 정의하는 매개변수
  - Ex) C/C++, Ruby, Perl, PHP 등 대부분의 언어
  - Ex) Ada, Rust, Go, Pascal, Haskell 등에서는 배열인자 등으로 간접적으로 구현

Ex) Ada
```
procedure Greet(Name : String := "Guest") is
begin
  Put_Line("Hello, " & Name);
end Greet;

Greet;                  -- "Hello, Guest"
Greet("Alice");         -- "Hello, Alice"
```

Ex) Ruby
```
def sum(*args)
  args.reduce(0) { |total, x| total + x }
end

puts sum(1,2,3,4)   # 10
puts sum()          # 0
```

매개변수 전달 방식(1)

- 값 전달(Call by Value)
- 실 매개변수의 값을 형식 매개변수에 복사하고 이를 부프로그램의 지역 변수로 사용
  - Ex) C/C++, Pascal, Java의 기본형, JavaScript의 기본형, Ada에서 in 등
- 형식 매개변수의 변화가 실 매개변수에는 영향을 미치지 않음
- 큰 데이터를 전달하는 경우 인수 전달이 시간적, 공간적으로 비효율적
  - Ex) 100x100 행렬을 전달하는 경우

Ex) C

```
void sub(int x)
{
   printf("%d", ++x);   // 11를 출력
}

int a=10;
sub(a);
printf("%d", a);          // 10을 출력
```

Ex C)

```
typedef struct { int a[100][100]; } Mat;
int sum_elements(Mat A) {
   int s=0;
   for (int i=0;i<100;i++)
      for (int j=0;j<100;j++)
         s += A.a[i][j];
   return s;
}

sum_elements(A);  // 배열을 포함한 구조체 A를 복사
```

매개변수 전달 방식(2)

- 참조 전달(Call by Reference or Call by Sharing)
- 실 매개변수의 주소를 형식 매개변수로 보내는 방식
  - Ex) Python, Ruby, Java의 객체, C++에서 참조 연산자(&) 등
- 형식 매개변수는 실 매개변수의 별명(Alias)
  - 형식 매개변수의 변화가 실 매개변수에 반영
- 용량 데이터 처리에 효율적이지만, 별명 문제로 안전성이 떨어짐

Ex) C++

```
void sub(int &x)
{
   cout << ++x;   // 11를 출력
}

int a=10;
sub(a);
cout << a;          // 11를 출력
```

Ex) Python

```
def duplicate_list(lst):
   copy = lst              # 참조만 복사
   copy[0] = 999
   return copy

a = [1, 2, 3]
b = duplicate_list(a)

print("a =", a)  # a = [999, 2, 3] ← 원본까지 바뀜!
print("b =", b)  # b = [999, 2, 3]
```

매개변수 전달 방식(3)

- 값 전달과 포인터
- 포인터는 주소값을 직접 다룰 수 있는 변수
  - Ex) C/C++, Pascal, Ada 등
- 구조체나 배열 같은 큰 데이터를 값으로 복사하면 성능과 메모리 낭비가 큼
  - 포인터를 활용해서 값 전달을 참조 전달 처럼 사용

Ex) C

```
#define SIZE 100000   // 100,000개 원소
typedef struct {
   int data[SIZE];
} BigStruct;

void processByValue(BigStruct bs) {
   bs.data[0] = 42;  // 복사본만 수정
}

BigStruct big;       // 약 400KB (int=4바이트 × 100,000개)
processByValue(big);    // 함수 호출 시 400KB 복사 발생
```

Ex) C

```
#define SIZE 100000   // 100,000개 원소
typedef struct {
   int data[SIZE];
} BigStruct;

void processByPointer(BigStruct *bs) {
   bs->data[0] = 99; // 원본 수정
}

BigStruct big;       // 약 400KB (int=4바이트 × 100,000개)
processByPointer(&big); // 주소(8바이트)만 전달
```

매개변수 전달 방식(4)

- Call by Reference
- 엄밀한 의미의 Call by Reference는 진짜 별명(Alias) 방식
  - 변수명이 아예 주소값 상수로 대체되어, 새롭게 정의 불가
  - 별도의 주소 공간을 만들지 않음
  - Ex) C++의 & 등

Ex) C

```
void sub(int x)
{
   printf("%d", ++x);   // 11를 출력
}

int a=10;
sub(a);
printf("%d", a);          // 10을 출력
```

주소	변수	값
0x2000	sub의 x	11
0x3000	a	10

Ex) C++

```
void sub(int &x)  // x는 a 주소값을 나타내는 상수로 취급
{
   printf("%d", ++x);   // 11를 출력
}

int a=10;
sub(a);
printf("%d", a);          // 11을 출력
```

주소	변수	값
0x3000	a, sub의 x	11

매개변수 전달 방식(5)

- Call by Reference
- 포인터 vs 참조
  - 포인터 변수는 그 자체로 하나의 변수이고, 변수값 자체의 변화가 가능
  - 참조는 상수값의 치환이므로 참조값 자체는 변화 불가
  - Ex) C의 배열이름은 배열의 참조값

Ex) C++

```
int a[2] = {10,20};
int b[2] = {50,60};

int *c = &a[0];
int &d = b[0];

c++;
d++;

cout << *c << endl;  // 20이 출력
cout << d << endl;   // 51이 출력
```

주소	변수	값
0x2000	a or a[0]	10
0x2004	a[1]	20
0x3000	b or b[0] or d	50
0x3004	b[1]	60

주소	변수	값
0x2000	a or a[0]	10
0x2004	a[1]	20
0x3000	b or b[0] or d	50
0x3004	b[1]	60
0x4000	c	0x2000

주소	변수	값
0x2000	a or a[0]	10
0x2004	a[1]	20
0x3000	b or b[0] or d	51
0x3004	b[1]	60
0x4000	c	0x2004

매개변수 전달 방식(6)

- Call by Sharing
- 별명이 아닌 객체의 참조값을 전달
  - 참조값을 저장하는 별도의 주소 공간이 필요
  - 포인터 연산자(*)나 참조 연산자(&)와 같은 별도의 연산자 불필요하고 자동으로 참조값으로 접근
  - 직접 주소값 연산 불가

Ex) Python

```
def f(lst):
   lst.append(99)

a = [10,20]
f(a)
print(a)  # [1,2,99]
```

참조값	변수	값
0x1000	a	0x2000
0x2000	a[0]	10
0x2024	a[1]	20

참조값	변수	값
0x1000	a	0x2000
0x2000	a[0]	10
0x2024	a[1]	20
0x1084	a[2]	99
0x3000	lst	0x2000

* 주의! Python의 리스트는 메모리에 연속적으로 저장되지 않음

매개변수 전달 방식(7)

- 이름 전달(Call by Name)
- 함수 호출 시, 인수 표현식 자체를 함수 본문에 그대로 대입하여 실행
- 일부 함수형 언어에서 지연 평가(Lazy Evaluation)를 위해 제한적으로 지원
  - Ex) Simula, Scala, Eiffel 등
- 키워드 대응 방식과는 다른 개념이므로 주의 필요
- 표현식의 평가 시점을 정확히 파악하지 않으면 오류 발생 위험 존재

Ex) Scala

```
def loopWhile(cond: => Boolean)(body: => Unit): Unit =
  while (cond) body

var x = 3
loopWhile(x > 0) {
  println(x)
  x -= 1
}
```

Ex) Scala

```
def twice(x: => Int): Int = x + x

var c = 0
def tick(): Int = { c += 1; c }

println(twice(tick()))  // tick() 두 번 실행 → c가 2 증가
println(c)              // 2
```

활성 레코드(1)

- 활성 레코드(Activation Record)
- 부프로그램 수행에 필요한 정보를 저장하는 공간
  - 매개변수, 지역변수, 정적/동적 링크, 복귀 주소, 반환값 등
- 컴파일시 활성 레코드의 구조를 결정
  - 인터프리트형 언어에서는 호출시 동적으로 결정

```
Ex) C
int add(int x, int y)
{
   return x+y;
}

int main()
{
   int c;
   c = add(3, 5);
   return c;
}
```

역할
지역 변수
매개변수
기타 정보
복귀 주소
반환값

< 기본 활성 레코드 구조 >

역할	값
지역 변수 c	8
기타 정보	...
복귀 주소	?
반환값	1

< Main 함수의 활성 레코드 >

역할	값
매개변수 x	3
매개변수 y	5
기타 정보	...
복귀 주소	return c의 주소
반환값	8

< add 함수의 활성 레코드 >

활성 레코드(2)

- 활성 레코드의 관리 방식
- 정적 구조
  - 메모리 할당이 적재 시간에 모든 변수들의 위치가 고정
  - 부프로그램 중첩 및 재귀 호출 불가
  - Ex) 고전 FORTRAN, 고전 COBOL 등
- 스택(Stack) 기반 구조
  - 부프로그램이 활성화될 때 활성 레코드가 스택에 동적으로 생성
  - Ex) C, Pascal 등 대부분의 컴파일형 언어
- 동적 구조
  - 활성 레코드나 변수들이 필요할 때마다 힙(Heap)에서 동적으로 생성
  - 클로저, 코루틴, 비동기 함수 등 유연한 함수 구현 가능
  - 성능상 오버헤드가 발생하며, 가비지 컬렉션이 필요
  - Ex) Lisp, Scheme, JavaScript (클로저), Python 등 함수형 언어 또는 함수형 프로그래밍 지원 언어

스택 기반 구조(1)

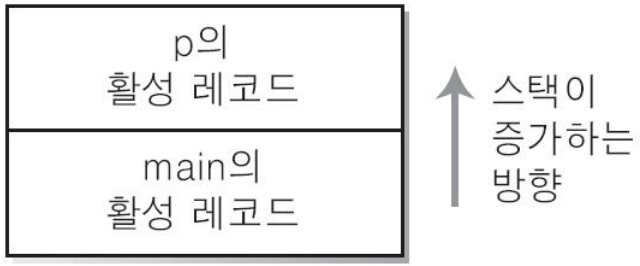
- 스택 기반 구조
- 부프로그램이 활성화될 때 활성 레코드가 스택에 동적으로 생성되고 종료시 해제
- 메모리 활용이 효율적이고 구현이 간단함
  - 호출/복귀 시 스택 포인터 조정만으로 활성 레코드 생성과 해제 가능
  - Heap의 경우는 별도로 공간 회수 필요
- 재귀 함수 지원
- 지역 변수 보호
  - 각 함수마다 독립된 활성 레코드를 가지므로, 함수 간 변수 충돌 없음
- 유연한 함수 사용이 불가능하거나 비효율적
  - 클로저(Closure), 코루틴(Coroutine), 비동기 함수(Aysnc), 중첩 함수등과 같이 유연한 함수 구현이 어려움
- 스택 오버플로우(Stack Overflow) 위험
  - 재귀 깊이가 깊거나 지역 변수가 너무 크면 스택 용량 초과 발생

스택 기반 구조(2)

❖ Ex) C에서 함수 p와 q를 호출 및 복귀 예

- 운영체제에서 main() 함수 호출로 프로그램을 시작
  - 스택에 main의 활성 레코드 생성

- p() 호출
  - 스택에 p의 활성 레코드 생성



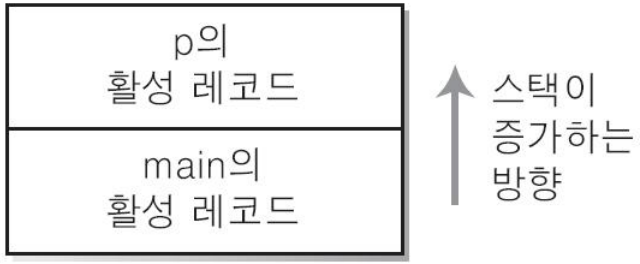

```
Ex) C
void q(void)
{
   ⋮
}
void p(void)
{
   q();
    ⋮
}
int main(void)
{
   p();
    ⋮
}
```

스택 기반 구조(3)

❖ Ex) C에서 함수 p와 q를 호출 및 복귀 예

- q() 호출
  - 스택에 q의 활성 레코드 생성

- q() 종료
  - 스택에서 q의 활성 레코드 제거




```
Ex) C
void q(void)
{
   ⋮
}
void p(void)
{
   q();
    ⋮
}
int main(void)
{
   p();
    ⋮
}
```

스택 기반 구조(4)

❖ Ex) C에서 함수 p와 q를 호출 및 복귀 예

- p() 종료
  - 스택에서 p의 활성 레코드 제거

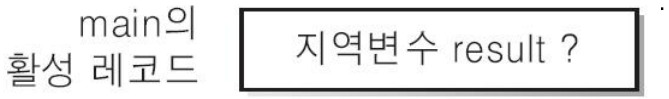
- main() 종료
  - 스택에서 main의 활성 레코드 제거
  - 프로그램 종료
  - 운영체제로 제어권 복귀

```
Ex) C
void q(void)
{
   ⋮
}
void p(void)
{
   q();
    ⋮
}
int main(void)
{
   p();
    ⋮
}
```

동적 링크(1)

- 동적 링크(Dynamic Link)
- 활성 레코드 제거시 범위 지정 필요
  - 어디까지가 활성 레코드인지 확인 필요
  - 호출자의 활성 레코드의 시작 부분 주소를 저장

- Ex) 재귀형 factorial 함수
- 프로그램의 시작
  - 운영체제에서 main() 함수 호출

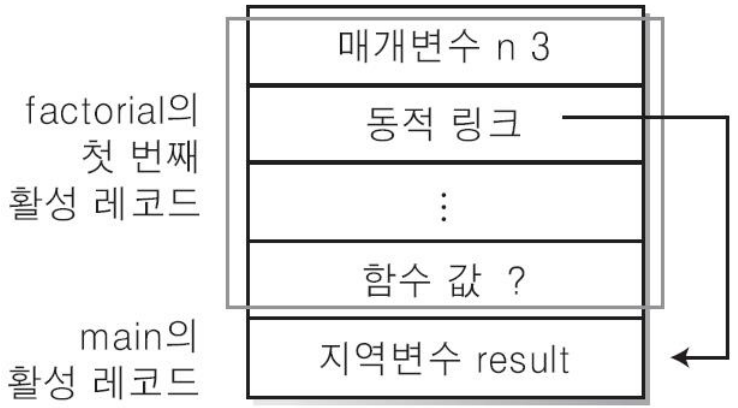


Ex) C

```
int factorial(int n)
{
   if(n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
int main(void)
{
   int result;
   result = factorial(3);
   return 0;
}
```

동적 링크(2)

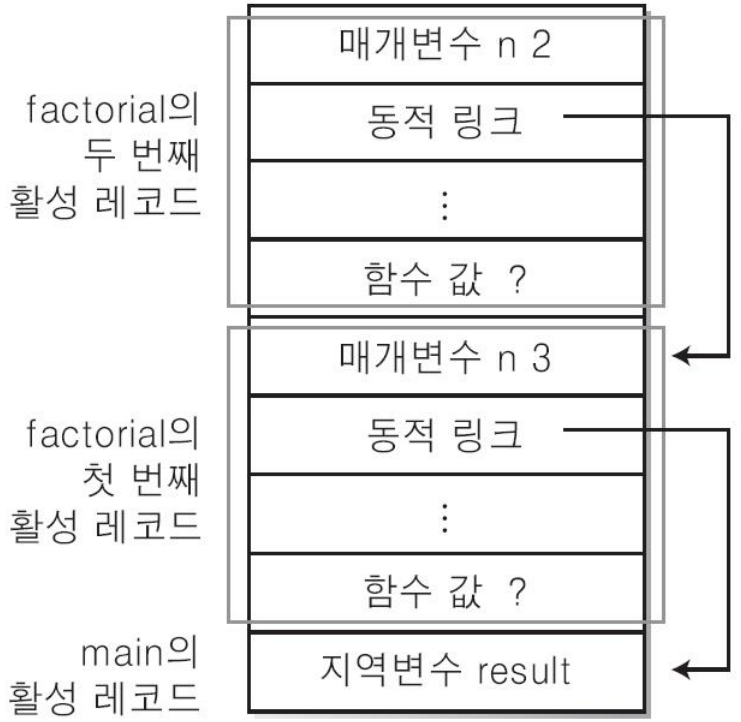
- Ex) 재귀형 factorial 함수
- factorial(3) 호출
  - 매개변수 전달(n=3)
  - 동적 링크로 main 의 활성 레코드를 연결
  - 함수의 반환값은 정해지지 않음




```
Ex) C
int factorial(int n)
{
   if(n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
int main(void)
{
   int result;
   result = factorial(3);
   return 0;
}
```

동적 링크(3)

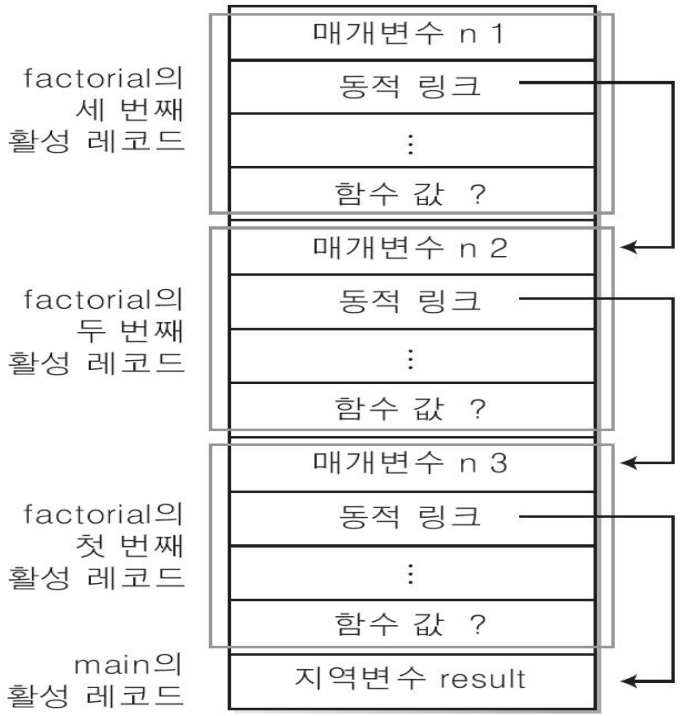
- Ex) 재귀형 factorial 함수
- factorial(2) 호출
  - 매개변수 전달(n=2)
  - 동적 링크로 첫번째 함수의 활성 레코드를 연결
  - 함수의 반환값은 정해지지 않음




```
Ex) C
int factorial(int n)
{
   if(n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
int main(void)
{
   int result;
   result = factorial(3);
   return 0;
}
```

동적 링크(4)

- Ex) 재귀형 factorial 함수
- factorial(1) 호출
  - 매개변수 전달(n=1)
  - 동적 링크로 두번째 함수의 활성 레코드를 연결
  - 함수의 반환값은 정해지지 않음

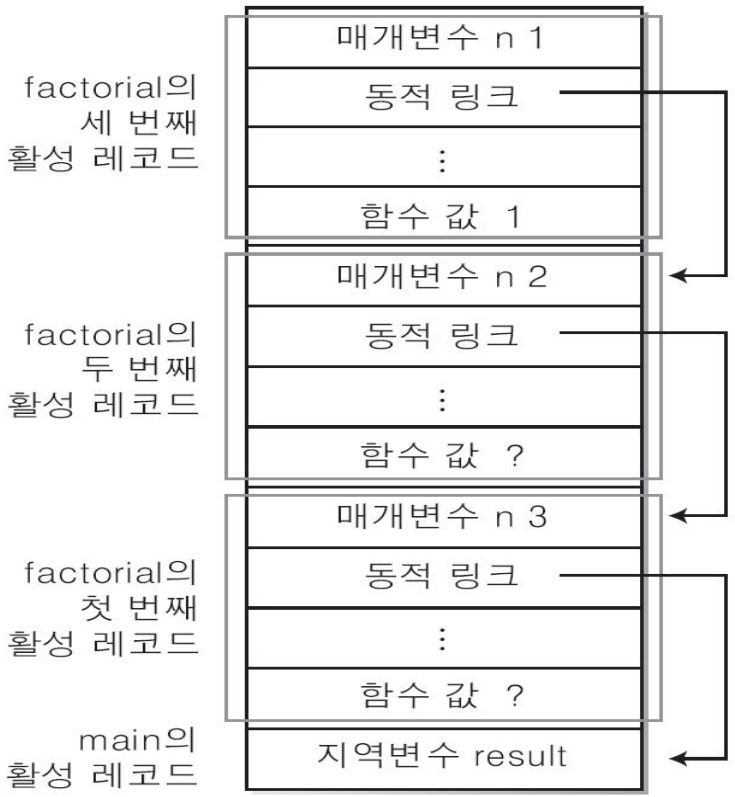


Ex) C

```
int factorial(int n)
{
   if(n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
int main(void)
{
   int result;
   result = factorial(3);
   return 0;
}
```

동적 링크(5)

- Ex) 재귀형 factorial 함수
- factorial(1) 종료
  - 함수의 반환값이 1로 결정

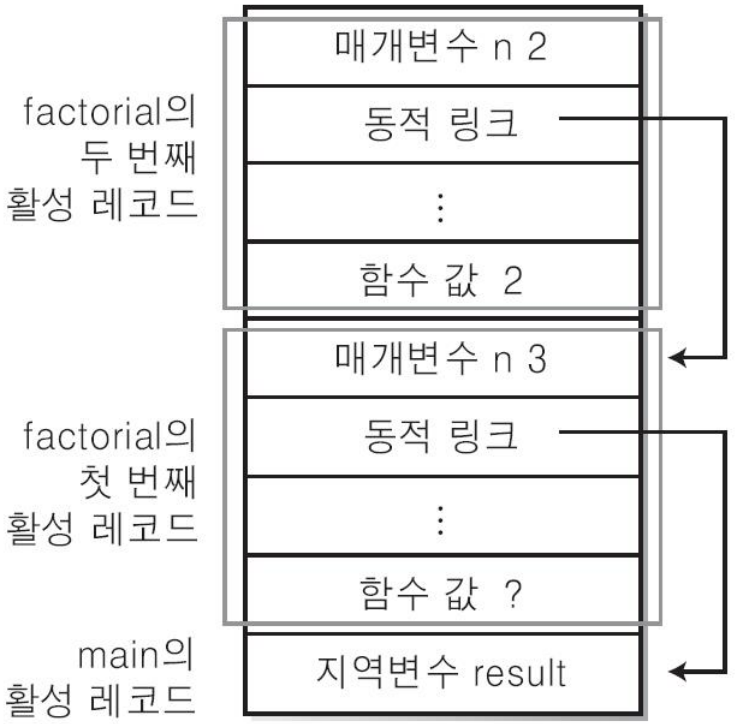


Ex) C

```
int factorial(int n)
{
   if(n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
int main(void)
{
   int result;
   result = factorial(3);
   return 0;
}
```

동적 링크(6)

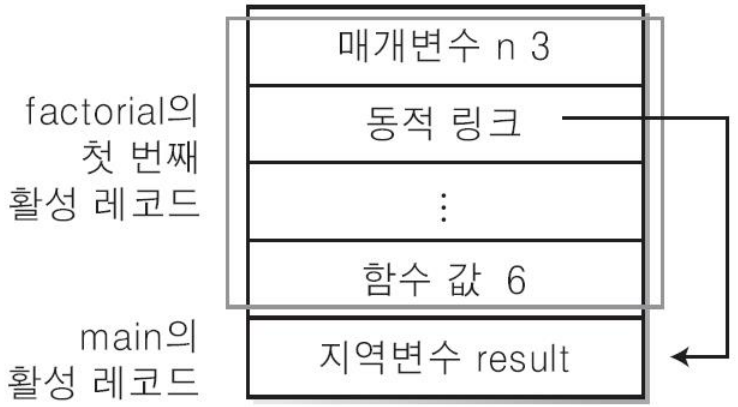
- Ex) 재귀형 factorial 함수
- factorial(2) 종료
  - 함수의 반환값이 n * factorial(1)로 결정 (2*1)




```
Ex) C
int factorial(int n)
{
   if(n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
int main(void)
{
   int result;
   result = factorial(3);
   return 0;
}
```

동적 링크(7)

- Ex) 재귀형 factorial 함수
- factorial(3) 종료
  - 함수의 반환값이 n * factorial(3)로 결정 (3*2)

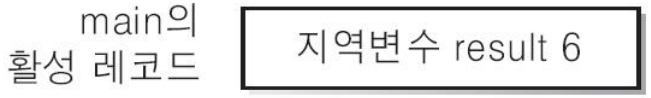


Ex) C

```
int factorial(int n)
{
   if(n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
int main(void)
{
   int result;
   result = factorial(3);
   return 0;
}
```

동적 링크(8)

- Ex) 재귀형 factorial 함수
- factorial(3) 종료
  - main 활성 레코드의 지역변수 result가 6이 됨




```
Ex) C
int factorial(int n)
{
   if(n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
int main(void)
{
   int result;
   result = factorial(3);
   return 0;
}
```

스택 기반 구조 - Revistied(1)

- 컴퓨터 구조의 지원
- 일반적으로 컴퓨터 구조에서는 이를 지원하기 위한 별도의 레지스터들이 존재
  - SPR(Special Purpose Register)로 존재
- 스택 포인터(Stack Pointer, SP)
  - 현재 스택의 가장 상단 주소를 저장
  - Push나 Pop이 발생하면 이 위치의 값에 저장되거나 반환되고, SP의 값이 변화
- 프레임 포인터(Frame Pointer, FP)
  - 현재 활성 레코드의 기준 주소
  - 지역 변수, 매개변수, 반환 주소를 일정한 오프셋으로 접근하기 위해 사용되는 기준점
- 복귀 주소(Return Address)
  - 복귀 주소를 저장하는 레지스터
  - 함수 호출시 저장되며, 복귀시 해당 레지스터의 값을 PC로 복사
  - Ex) ARM: LR(Link Register), MIPS: $ra
  - 별도의 레지스터가 아닌 활성 레코드에 저장되기도 함

스택 기반 구조 - Revistied(2)

- 함수 호출 절차 예제
- 현재 PC는 0x3000
  - sum 함수 호출
- sum의 함수의 활성 레코드는 16 Bytes
  - 매개변수 2개, 이전 FP, 반환값
  - 4 Bytes * 4 = 16 Bytes(0x10)

주소	내용
0x3E8	
0x3EC	
0x3F0	
0x3F8	
0x3F8	
0x3FC	
0x400~	Main의 활성 레코드

< 메모리>

주소	역할	값
FP - 12	매개 변수 x	3
FP - 8	매개 변수 y	4
FP -4	이전 FP	?
FP	지역 변수 ret	??

< sum 함수의 활성 레코드 >

SPR	값	SPR	값
PC	0x3000	FP	0x420
SP	0x400	LR	??

< 주요 SPR >

```
void main() {
   int a = 3, b = 4;
   int c = sum(a,b);
}

int sum(int x, int y){
   return x+y;
}
```

스택 기반 구조 - Revistied(3)

- 컴퓨터 구조의 지원
- 일반적으로 컴퓨터 구조에서는 이를 지원하기 위한 별도의 레지스터들이 존재
  - SPR(Special Purpose Register)로 존재
- 스택 포인터(Stack Pointer, SP)
  - 현재 스택의 가장 상단 주소를 저장
  - Push나 Pop이 발생하면 이 위치의 값에 저장되거나 반환되고, SP의 값이 변화
- 프레임 포인터(Frame Pointer, FP)
  - 현재 활성 레코드의 기준 주소
  - 지역 변수, 매개변수, 반환 주소를 일정한 오프셋으로 접근하기 위해 사용되는 기준점
- 복귀 주소(Return Address)
  - 복귀 주소를 저장하는 레지스터
  - 함수 호출시 저장되며, 복귀시 해당 레지스터의 값을 PC로 복사
  - Ex) ARM: LR(Link Register), MIPS: $ra
  - 별도의 레지스터가 아닌 활성 레코드에 저장되기도 함

스택 기반 구조 - Revistied(4)

- 함수 호출 절차 예제
- 1) 활성 레코드의 크기 및 위치 계산
  - 크기: 4 Bytes * 4 = 16 Bytes(0x10)
  - 매개변수 2개, 이전 FP, 반환값
  - 위치: FP를 기준으로 지정
  - x는 FP - 12, 반환값은 FP

주소	내용
0x3E8	
0x3EC	
0x3F0	
0x3F4	
0x3F8	
0x3FC	
0x400~	Main의 활성 레코드

< 메모리>

주소	역할	값
FP - 12	매개변수 x	?
FP - 8	매개변수 y	?
FP -4	이전 FP	?
FP	반환값	?

< sum 함수의 활성 레코드 >

SPR	값	SPR	값
PC	0x3000	FP	0x420
SP	0x400	LR	??

< 주요 SPR >

sum 함수 호출
PC는 0x3000으로가정

sum 함수 코드
시작 주소는 0x4000

```
void main() {
   int a = 3, b = 4;
   int c = sum(a,b);
}

int sum(int x, int y){
   return x+y;
}
```

스택 기반 구조 - Revistied(5)

- 함수 호출 절차 예제
- 1) 활성 레코드의 크기 및 위치 계산
  - 크기: 4 Bytes * 4 = 16 Bytes(0x10)
  - 매개변수 2개, 이전 FP, 반환값
  - 위치: FP를 기준으로 지정
  - x는 FP - 12, 반환값은 FP
- 2) 활성 레코드의 정보를 저장
  - sum의 활성 레코드 시작 위치를 지정
  - FP ← 0x3FC
  - 매개변수를 저장(3,4) 및 이전 FP의 값을 저장(여기서는 무시 가능)

주소	내용
0x3E8	
0x3EC	
0x3F0	3
0x3F4	4
0x3F8	0x420
0x3FC	??
0x400~	Main의 활성 레코드

< 메모리>

주소	역할	값
FP - 12	매개변수 x	3
FP - 8	매개변수 y	4
FP -4	이전 FP	0x420
FP	반환값	??

< sum 함수의 활성 레코드 >

SPR	값	SPR	값
PC	0x3000	FP	0x3FC
SP	0x400	LR	??

< 주요 SPR >

```
void main() {
   int a = 3, b = 4;
   int c = sum(a,b);
}

int sum(int x, int y){
   return x+y;
}
```

스택 기반 구조 - Revistied(6)

- 함수 호출 절차 예제
- 1) 활성 레코드의 크기 및 위치 계산
  - 크기: 4 Bytes * 4 = 16 Bytes(0x10)
  - 매개변수 2개, 이전 FP, 반환값
  - 위치: FP를 기준으로 지정
  - x는 FP - 12, 반환값은 FP
- 2) 활성 레코드의 정보를 저장
  - sum의 활성 레코드 시작 위치를 지정
  - FP ← 0x3FC
  - 매개변수를 저장(3,4) 및 이전 FP의 값을 저장(여기서는 무시 가능)
- 3) 활성 레코드 만큼 SP를 차감
  - SP ← SP - #0x10

주소	내용
0x3E8	
0x3EC	
0x3F0	3
0x3F4	4
0x3F8	0x420
0x3FC	??
0x400~	Main의 활성 레코드

< 메모리>

주소	역할	값
FP - 12	매개변수 x	3
FP - 8	매개변수 y	4
FP -4	이전 FP	0x420
FP	반환값	??

< sum 함수의 활성 레코드 >

SPR	값	SPR	값
PC	0x3000	FP	0x3FC
SP	0x3F0	LR	??

< 주요 SPR >

```
void main() {
   int a = 3, b = 4;
   int c = sum(a,b);
}

int sum(int x, int y){
   return x+y;
}
```

스택 기반 구조 - Revistied(7)

- 함수 호출 절차 예제
- 1) 활성 레코드의 크기 및 위치 계산
  - 크기: 4 Bytes * 4 = 16 Bytes(0x10)
  - 매개변수 2개, 이전 FP, 반환값
  - 위치: FP를 기준으로 지정
  - x는 FP - 12, 반환값은 FP
- 2) 활성 레코드의 정보를 저장
  - sum의 활성 레코드 시작 위치를 지정
  - FP ← 0x3FC
  - 매개변수를 저장(3,4) 및 이전 FP의 값을 저장(여기서는 무시 가능)
- 3) 활성 레코드 만큼 SP를 차감
  - SP ← SP - #0x10
- 4) 복귀 주소를 해당 레지스터에 저장
  - LR ← PC+4

주소	내용
0x3E8	
0x3EC	
0x3F0	3
0x3F4	4
0x3F8	0x420
0x3FC	??
0x400~	Main의 활성 레코드

< 메모리>

주소	역할	값
FP - 12	매개변수 x	3
FP - 8	매개변수 y	4
FP -4	이전 FP	0x420
FP	반환값	??

< sum 함수의 활성 레코드 >

SPR	값	SPR	값
PC	0x3000	FP	0x3FC
SP	0x3F0	LR	0x3004

< 주요 SPR >

```
void main() {
   int a = 3, b = 4;
   int c = sum(a,b);
}

int sum(int x, int y){
   return x+y;
}
```

스택 기반 구조 - Revistied(8)

- 함수 호출 절차 예제
- 1) 활성 레코드의 크기 및 위치 계산
  - 크기: 4 Bytes * 4 = 16 Bytes(0x10)
  - 매개변수 2개, 이전 FP, 반환값
  - 위치: FP를 기준으로 지정
  - x는 FP - 12, 반환값은 FP
- 2) 활성 레코드의 정보를 저장
  - sum의 활성 레코드 시작 위치를 지정
  - FP ← 0x3FC
  - 매개변수를 저장(3,4) 및 이전 FP의 값을 저장(여기서는 무시 가능)
- 3) 활성 레코드 만큼 SP를 차감
  - SP ← SP - #0x10
- 4) 복귀 주소를 해당 레지스터에 저장
  - LR ← PC+4
- 5) 함수의 시작 위치로 분기
  - PC ← 0x4000

주소	내용
0x3E8	
0x3EC	
0x3F0	3
0x3F4	4
0x3F8	0x420
0x3FC	??
0x400~	Main의 활성 레코드

< 메모리>

주소	역할	값
FP - 12	매개변수 x	3
FP - 8	매개변수 y	4
FP -4	이전 FP	0x420
FP	반환값	??

< sum 함수의 활성 레코드 >

SPR	값	SPR	값
PC	0x4000	FP	0x3FC
SP	0x3F0	LR	0x3004

< 주요 SPR >

sum 함수 호출
PC는 0x3000으로가정

sum 함수 코드
시작 주소는 0x4000

```
void main() {
   int a = 3, b = 4;
   int c = sum(a,b);
}

int sum(int x, int y){
   return x+y;
}
```

스택 기반 구조 - Revistied(9)

- 함수 복귀 절차 예제
- 1) 반환값을 활성 레코드에 저장
  - 7을 저장
- 2) SP와 FP를 원래 상태로
  - FP ← 0x420
  - SP ← SP + 0x10
- 3) 복귀 주소로 분기
  - PC ← LR

주소	내용
0x3E8	
0x3EC	
0x3F0	3
0x3F4	4
0x3F8	0x420
0x3FC	7
0x400~	Main의 활성 레코드

< 메모리>

주소	역할	값
FP - 12	매개변수 x	3
FP - 8	매개변수 y	4
FP -4	이전 FP	0x420
FP	반환값	7

< sum 함수의 활성 레코드 >

SPR	값	SPR	값
PC	0x3004	FP	0x420
SP	0x400	LR	0x3004

< 주요 SPR >

- 이 예제는 활성 레코드 부분만 단순화 한 것
  - GPR/SPR 복구 방법, 반환값 복사 시점 결정등의 약속이 필요
- ABI(Application Binary Interface)
  - 프로그램이 이진(Binary) 수준에서 운영체제/하드웨어와 상호작용하는 규칙
  - 컴파일된 코드끼리 서로 호환되도록 정해둔 약속

```
void main() {
   int a = 3, b = 4;
   int c = sum(a,b);
}

int sum(int x, int y){
   return x+y;
}
```

실습문제(8)

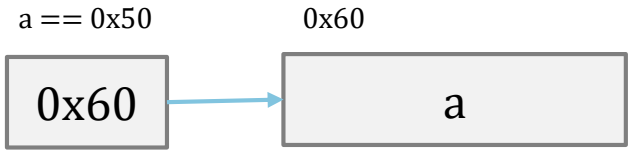
```
#include <stdio.h>

void login() {
   char id[20] = "JustForFunc";
   char password[20] = "SMUStudy";
   printf("Login...\n");
}

void hacker() {
   unsigned char garbage[20];
   printf("[Hacker]: \n");
   for(int i = 0; i < 8; i++) {
      printf("  garbage[%d]: %c \n", i, garbage[i]);
   }
}
int main() {
   hacker();
   login();
   hakcer();
   return 0;
}
```

```
[Hacker]:
  garbage[0]: ▒
  garbage[1]: B
  garbage[2]: (
  garbage[3]: L
  garbage[4]:
  garbage[5]:
  garbage[6]:
  garbage[7]:
Login...
[Hacker]:
  garbage[0]: S
  garbage[1]: M
  garbage[2]: U
  garbage[3]: S
  garbage[4]: t
  garbage[5]: u
  garbage[6]: d
  garbage[7]: y
```

함수의 매개 변수 전달 방식(1)

- 실질적으로 모두 Call by value
- 중요한 건 매개 변수의 접근 방식!
  - 매개변수를 직접 접근할 것인가? vs 매개변수의 값을 주소로 활용할 것인가?
- []는 메모리에 접근하는 연산
  - a[i] => *(a+i x 4) (int형인 경우)
  - 매개변수를 메모리 접근 연산으로 사용하는 경우 call by reference로 동작

- 모든 것이 객체! 라는 것의 진짜 의미
  - 모든 변수는 객체를 가리키는 포인터로 취급
  - class AA = new a();



함수의 매개 변수 전달 방식(2)

- 자바의 매개 변수 전달 방식
  - 기본 자료형은 Call by value
  - 다른 자료형은 Call by reference
    - ✓ 단, 정확한 이해가 필요

```
public class Main {
   public static void func(int a, int[] b, int[] c) {
      a = 100;
      b = new int[]{41, 42};
      c[0] = 53;
   }

   public static void main(String[] args) {
      int a = 11;
      int[] b = {21, 22};
      int[] c = {31, 32};

      func(a, b, c);

      System.out.println("a=" + a);          // ???
      System.out.println("b=" + Arrays.toString(b)); // ???
      System.out.println("c=" + Arrays.toString(c)); // ???
   }
}
```

함수의 매개 변수 전달 방식(2)

- 자바의 매개 변수 전달 방식
  - 기본 자료형은 Call by value
  - 다른 자료형은 Call by reference
    - ✓ 단, 정확한 이해가 필요

```
public class Main {
   public static void func(int a, int[] b, int[] c) {
      a = 100;
      b = new int[]{41, 42};
      c[0] = 53;
   }

   public static void main(String[] args) {
      int a = 11;
      int[] b = {21, 22};
      int[] c = {31, 32};

      func(a, b, c);

      System.out.println("a=" + a);          // 11
      System.out.println("b=" + Arrays.toString(b)); // [21, 22]
      System.out.println("c=" + Arrays.toString(c)); // [53, 32]
   }
}
```

함수의 매개 변수 전달 방식(3)

- 파이썬의 매개 변수 전달 방식
  - 기본 자료형은 Call by value
  - 다른 자료형은 Call by reference
    - ✓ id는 객체 고유값(포인터와는 다름)

```
def func1(a):
    a = 100
    print("2. Address of a =" + hex(id(a)))

a = 11
print("1. Address of a ="+hex(id(a)))
func1(a)
print("3. Address of a ="+hex(id(a)))
print("a=",a)
```

```
1. Address of a =0x1b3ea8c6a70
2. Address of a =0x1b3ea8f55d0
3. Address of a =0x1b3ea8c6a70
a= 11
```

함수의 매개 변수 전달 방식(4)

- 파이썬의 매개 변수 전달 방식
  - 기본 자료형은 Call by value
  - 다른 자료형은 Call by reference
    - ✓ id는 객체 고유값(포인터와는 다름)

```
def func1(a):
    a[1] = 25
    print("2. Address of a =" + hex(id(a)))
    print("2-1. Address of a =" + hex(id(a[0])))
    print("2-2. Address of a =" + hex(id(a[1])))
    print("2-3. Address of a =" + hex(id(a[2])))

a = [0,1,2]
print("1. Address of a ="+hex(id(a)))
print("1-1. Address of a ="+hex(id(a[0])))
print("1-2. Address of a ="+hex(id(a[1])))
print("1-3. Address of a ="+hex(id(a[2])))
func1(a)
print("3. Address of a ="+hex(id(a)))
print("a=",a)
```

```
1. Address of a =0x257ef824100
1-1. Address of a =0x257ef2f6910
1-2. Address of a =0x257ef2f6930
1-3. Address of a =0x257ef2f6950
2. Address of a =0x257ef824100
2-1. Address of a =0x257ef2f6910
2-2. Address of a =0x257ef2f6c30
2-3. Address of a =0x257ef2f6950
3. Address of a =0x257ef824100
a= [0, 25, 2]
```