

Jin-Soo Kim  
([jinsoo.kim@snu.ac.kr](mailto:jinsoo.kim@snu.ac.kr))

Systems Software &  
Architecture Lab.  
Seoul National University

Spring 2023

# Fundamental Data Types



# Declaration, Expression, Assignment

- Variables and constants are the objects that a program manipulates
- All variables must be declared before they can be used

```
#include <stdio.h>

int main(void)
{
    int a, b, c;           /* declaration */
    float x, y = 3.3, z = -7.7; /* declaration */

    printf("Input two integers: "); /* function call */
    scanf("%d%d", &b, &c); /* function call */
    a = b + c;           /* assignment */
    x = y + z;           /* assignment */
    ...
}
```

# Declarations

- Associate a type with each variable declared
- This tells the compiler to set aside an appropriate amount of **memory space to hold values** associated with variables
- This also enables the compiler to instruct the machine to perform specified operation correctly
  - `b + c`      `/* integer addition */`
  - `y + z`      `/* floating-point addition */`

# Expressions

- Meaningful combinations of constants, variables, operators, and function calls
  - A constant, variable, or function call itself is also an expression
    - `a + b`
    - `sqrt(7.333)`
    - `5.0 * x - tan(9.0/x)`
  - Most expressions have a value
    - `i = 7;`
    - `printf("hello, world\n");`
    - `3.777;`
    - `a + b;`
- Perfectly legal, but they are not useful*

# Assignment

- Assignment statement: *variable* = *expr*;

## Mathematical equation

$x + 2 = 0$   
 $x = x + 1$  (meaningless)

## Assignment expression

$x + 2 = 0$  /\* wrong \*/  
 $x = x + 1$

- Although they look alike, the assignment operator in C and the equal sign in mathematics have different meaning
- In C, think of it as: *variable* ← *expr*;

# Basic Data Types

Type	Data types	# Bytes in 32-bit	# Bytes in 64-bit
Integers	(signed) char	1	1
	unsigned char	1	1
	(signed) short (int)	2	2
	unsigned short (int)	2	2
	(signed) int	4	4
	unsigned (int)	4	4
	(signed) long (int)	4	8
	unsigned long (int)	4	8
Integers (C99)	(signed) long long (int)	8	8
	unsigned long long (int)	8	8
Floating-points	float	4	4
	double	8	8
	long double	10/16	10/16

# char (1)

- A variable of type char can be used to hold small integer values
- 1 byte (8 bits) in memory space
  - $2^8$  or 256 distinct values (signed: -128 ~ 127, unsigned: 0 ~ 255)
- Most machines use ASCII codes to represent a character in bits
  - A character encoding scheme
  - A character constant has its corresponding integer value
  - No particular relationship between the value of the character constant representing a digit and the digit's intrinsic integer value (e.g., '2' != 2)

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

# char (2)

- Nonprinting and hard-to-print characters require an escape sequence using \ (backslash) character
  - Used to escape the usual meaning of the character that follows it

```
#include <stdio.h>

int main(void)
{
    printf("%c", '\a');
    printf("\"abc\"");
    printf("%cabc%c", '\'', '\'');
}
```

Special Characters		
Name	Written in C	Value
null character	\0	0
alert	\a	7
newline	\n	10
double quote	\"	34
single quote	\'	39
backslash	\\	92



# char (3)

- Characters are treated as small integers

```
#include <stdio.h>

int main(void)
{
    char c = 'a';           /* stored in memory as 01100001 */
    int i;

    printf("%c\n", c);     /* a is printed */
    printf("%d\n", c);     /* 97 is printed */
    printf("%c%c%c\n", c, c+1, c+2); /* abc is printed */

    for (i = 'a'; i <= 'z'; i++)
        printf("%c", i);   /* abc...z is printed */
    for (c = '0'; c <= '9'; c++)
        printf("%d ", c);  /* 48 49 ... 57 is printed */
}
```

# getchar() / putchar()

```
/* Macros defined in stdio.h */  
  
#include <stdio.h>  
  
int getchar(void);          /* reads a character from the keyboard */  
  
int putchar(int c);        /* prints a character on the screen */
```

- EOF (end-of-file character)
  - `getchar()` returns EOF on end of file (or end of input by **ctrl-d**)
  - EOF is often defined as the integer value -1

```
#define EOF (-1)
```

# echo2.c

```
#include <stdio.h>

int main(void)
{
    int c;
    while ((c = getchar()) != EOF)
    {
        putchar(c);
        putchar(c);
    }
    return 0;
}
```

# upper.c

```
#include <stdio.h>

int main(void)
{
    int c;

    while ((c = getchar()) != EOF)
        if (c >= 'a' && c <= 'z')
            putchar(c + 'A' - 'a');
        else
            putchar(c);
    return 0;
}
```

# int

- Represents integers

- Stored in 4 bytes (= 32 bits) on 32-bit or 64-bit CPUs →  $2^{32}$  distinct values
- Signed integers:  $-2^{31}$  (= -2,147,483,648) ~  $2^{31}-1$  (2,147,483,647)
- Unsigned integers:  $0 \sim 2^{32}-1$  (4,294,967,295)

- Can result in **overflow** during computation

```
#include <stdio.h>
#define BIG 2000000000 /* 2 billion */

int main(void)
{
    int a, b = BIG, c = BIG;
    a = b + c;
    printf("%d\n", a);
}
```

# short / long / long long

- short: 2 bytes
- long: 4 bytes on 32-bit CPUs, 8 bytes on 64-bit CPUs
- long long: 8 bytes
- Suffixes can be appended to an integer constant to specify its type

Type	Suffix	Example
unsigned	u or U	a = 4000000024 <u>u</u> ;
long	l or L	b = 2000000022 <u>l</u> ;
unsigned long	ul or UL	c = 4000000000 <u>ul</u> ;
long long	ll or LL	d = 9000000000 <u>ll</u> ;
unsigned long long	ull or ULL	e = 9000000000001 <u>ull</u> ;

# Octal and Hexadecimal Constants (I)

## ■ Octal constants

- An integer constant that begins with 0
- $075301 \Leftrightarrow 7 \times 8^4 + 5 \times 8^3 + 3 \times 8^2 + 0 \times 8^1 + 1$

## ■ Hexadecimal constants

- Use characters '0' to '9' and 'A' to 'F'
- $0x2A$  (or  $0x2a$ )  $\Leftrightarrow 2 \times 16^1 + 10 = 42$
- $0x5B3$  (or  $0x5b3$ )  $\Leftrightarrow 5 \times 16^2 + 11 \times 16^1 + 3 = 1459$

## ■ Decimal constants

- First digit must not be 0 (except zero)

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# Octal and Hexadecimal Constants (2)

```
#include <stdio.h>

int main(void)
{
    printf("%d %x %o\n", 19, 19, 19);           /* 19 13 23 */
    printf("%d %x %o\n", 0x1c, 0x1c, 0x1c);    /* 28 1c 34 */
    printf("%d 0x%x 0%o\n", 017, 017, 017);    /* 15 0xf 017 */
    printf("%d %#x %#o\n", 017, 017, 017);     /* 15 0xf 017 */
    printf("%d\n", 11 + 0x11 + 011);           /* 37 */
    printf("%#010x\n", 2097151);               /* 0x001ffffff */
    printf("%d\n", 0x1FFFFFF);                 /* 2097151 */
    return 0;
}
```



# float / double

- IEEE Standard for Floating-Point Arithmetic (IEEE 754)

- Hold (approximated) real values
- float: 4 bytes (32 bits)  $\pm 1.4 \times 10^{-45} \sim 3.4 \times 10^{38}$
- double: 8 bytes (64 bits)  $\pm 4.9 \times 10^{-324} \sim 1.8 \times 10^{308}$

- Floating-point constants

- Decimal notation: 123456.7      0.0000123      1.
- Exponential notation: 1.234567e5      1234567e-1      0e0

- NOT all real numbers are representable

- Floating-point arithmetic operations differ from mathematical ones

# Assignment Conversions

- For assignment operations, the value of the **right side** is converted to the type of the **left**
- **double** → **float** conversion is implementation-dependent (rounded or truncated)
- **float** → **int** causes truncation of any fractional part
- Longer integers are converted to shorter ones (e.g., **int** → **short**) by dropping the excess **high-order** bits

# Usual Arithmetic Conversions

- For binary operations with two operands of different types, the "lower" type is promoted to the "higher" type before operation proceeds
  - (long double, any) → (long double, long double)
  - (double, any) → (double, double)
  - (float, any) → (float, float)
  - Perform **integral promotion**: (unsigned) char, (unsigned) short → int
  - (unsigned long int, any) → (unsigned long int, unsigned long int)
  - (long int, unsigned int) → (long int, long int) on 64-bit CPUs  
(unsigned long int, unsigned long int) on 32-bit CPUs
  - (long int, any) → (long int, long int)
  - (unsigned int, any) → (unsigned int, unsigned int)
  - Otherwise, both operands have type int

# Conversions: Examples

Declarations			
char c; unsigned u;	short s; unsigned long ul;	int i; float f;	long l; double d;
Expression	Type	Expression	Type
c - s / i	int	2 * i / l	long
u * 2.0 - i	double	u * 7 - i	unsigned
c + 3	int	f * 7 - i	float
c + 5.0	double	7 * s * ul	unsigned long
d + s	double	u > ul	unsigned long

- **d = i; Widening**
  - The value of **i** is converted to a double and then assigned to **d**
- **i = d; Narrowing**
  - Loss of information. The fraction part of **d** will be discarded

# Type Casting

- Explicit conversion using (*type*)
- `(double) i`
  - Converts the value of `i` so that the expression has type `double`
  - The variable `i` itself remains unchanged

## Examples

```
l = (long) ('A'+1.0);  
f = (float) ((int) d + 1);  
d = (double) i / 3;
```

## Wrong example

```
(double) x = 77;
```

- The cast operator (*type*) is an unary operator
  - `(float) i + 3`  $\Leftrightarrow$  `((float) i) + 3`

# The `sizeof` Operator (I)

- `sizeof(object)`

- A compile-time unary operator to find the # of bytes needed to store an *object*
- *object* can be a type such as `int` or `float`, or an expression such as `a + b`

```
#include <stdio.h>

int main(void)
{
    printf("char:\t%lu byte\n", sizeof(char));
    printf("short:\t%lu bytes\n", sizeof(short));
    printf("int:\t%lu bytes\n", sizeof(int));
    printf("long:\t%lu bytes\n", sizeof(long));
    printf("float:\t%lu bytes\n", sizeof(float));
    printf("double:\t%lu bytes\n", sizeof(double));
}
```

# The `sizeof` Operator (2)

- `sizeof(char) = 1`
- `sizeof(char) < sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)`
- `sizeof(signed) = sizeof(unsigned) = sizeof(int)`
- `sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)`
- `sizeof(object)` looks like a function, but it is not. An operator.
- The type returned by the operator `sizeof` is typically `unsigned`

# The Use of typedef

## ■ typedef

- Allows the programmer to explicitly associate a type with an identifier
- To abbreviate long declarations
- To have type names that reflect the intended use

```
typedef char      uppercase;
typedef int      INCHES;
typedef unsigned long  size_t;

int main(void)
{
    uppercase    u;
    INCHES      length, width;
    ...
}
```



# Example 1: What's Wrong?

```
#include <stdio.h>

int main(void)
{
    unsigned i;

    for (i = 10; i >= 0; i--)
        printf("%u\n", i);

    return 0;
}
```

# Example 2: What's Wrong?

```
#include <stdio.h>

int main(void)
{
    unsigned char c;

    while ((c = getchar()) != EOF)
        putchar(c);

    return 0;
}
```

# Example 3

```
#include <stdio.h>

int main()
{
    int n = 123456789;
    int nf, ng;
    float f;
    double g;

    f = (float) n;
    g = (double) n;
    nf = (int) f;
    ng = (int) g;
    printf("nf=%d ng=%d\n", nf, ng);
    return 0;
}
```

# Example 4

```
#include <stdio.h>

int main(void)
{
    double d;

    d = 1.0 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
        + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

    if (d == 2.0)
        printf("TRUE\n");
    else
        printf("FALSE\n");
    return 0;
}
```

# Example 5

```
#include <stdio.h>

#define PI      3.14
#define BIG     1e20

int main(void)
{
    float f1 = (PI + BIG) - BIG;
    float f2 = PI + (BIG - BIG);

    if (f1 == f2)
        printf("TRUE\n");
    else
        printf("FALSE\n");
    return 0;
}
```