2. Introduction to C Programming

[ECE10002] C Programming

Agenda

- Background
- Compiling and Executing
- Structure of C Programs
- Variables and Types
- Constants
- Formatted Input/Output
- Identifiers
- Comments

Why C Language ?

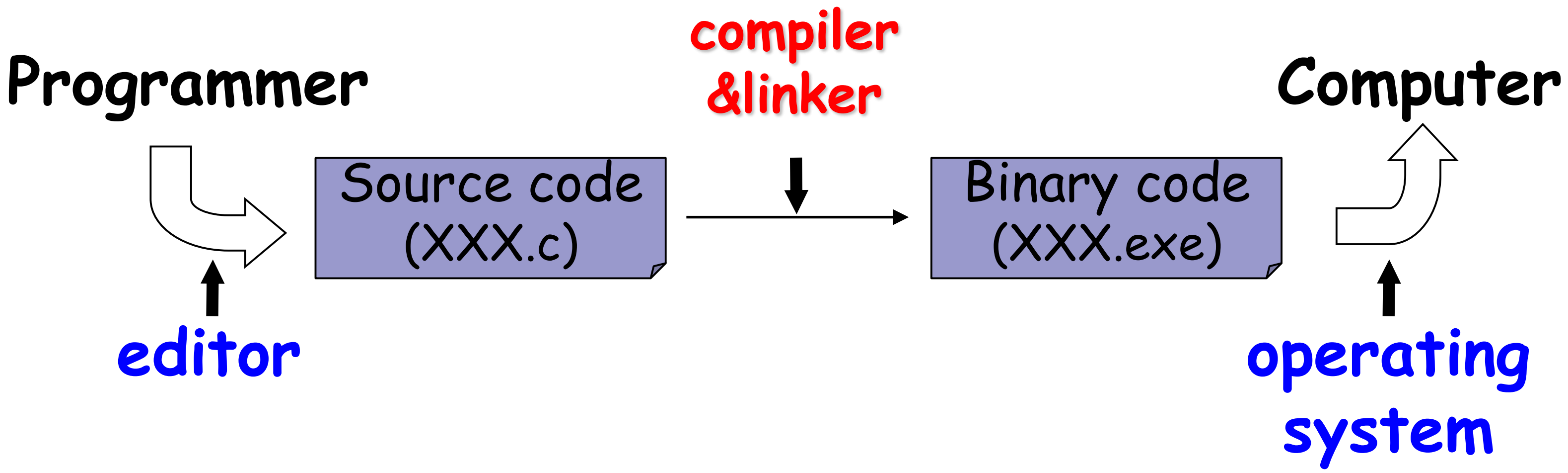
- Good to write high-performance S/W
  - Effective, simple, practical
  - C is a language for experts rather than for beginners.
- Some aspects of low level programming language
  - Control of fine details of low-level elements
    - Pointers, bit-wise operators
- World-wide popular programming languages

Agenda

- Background
- <u>Compiling and Executing</u>
- Structure of C Programs
- Variables and Types
- Constants
- Formatted Input/Output
- Identifiers
- Comments

Developing a Program in C

- Developing process



```
#include <stdio.h>

int main(void)
{
   printf("Hello World!₩n");
   return 0;
}
```

```
0000020 00b8 0000 0000 0000 0040 …
0000040 0000 0000 0000 0000 0000 …
0000060 0000 0000 0000 0000 0000 …
0000100 1f0e 0eba b400 cd09 b821 …
0000120 7369 7020 6f72 7267 6d61 …
0000140 2074 6562 7220 6e75 6920 …
0000160 6f6d 6564 0d2e 0a0d 0024 …
0000200 4550 0000 014c 0007 be3c …
0000220 079f 0000 00e0 0107 010b …
                  …
```

Popular C Compilers

- cc: default compiler of UNIX
- gcc: the most popular free compiler
  - UNIX/Linux: gcc
  - Windows: MinGW, DJGPP, Dev C++ …
- Visual Studio
- Turbo C++ / C++ builder, …

Example: A Greeting Program

- Hello.c: a program that prints out a greeting message.
  - Open a text editor and type the following codes
  - Save as “Hello.c”

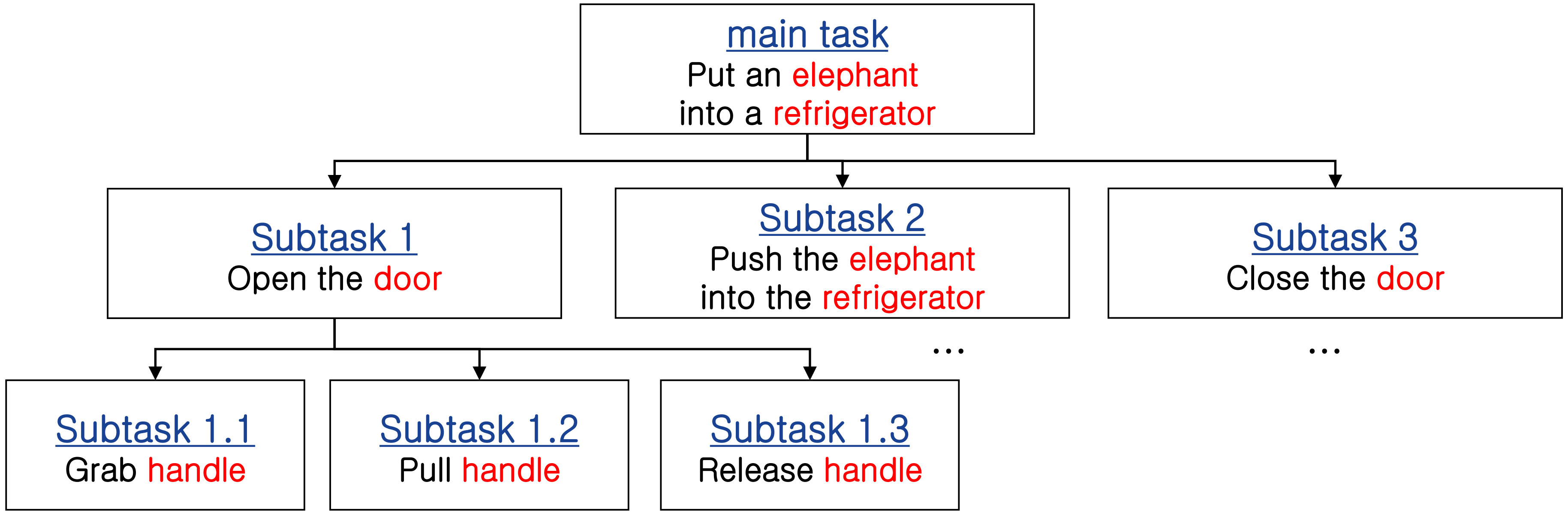
```
#include <stdio.h>


int main()
{
    printf(“Hello World!₩n”);
    return 0;
}
```

Agenda

- Background
- Compiling and Executing
- <u>Structure of C Programs</u>
- Variables and Types
- Constants
- Formatted Input/Output
- Identifiers
- Comments

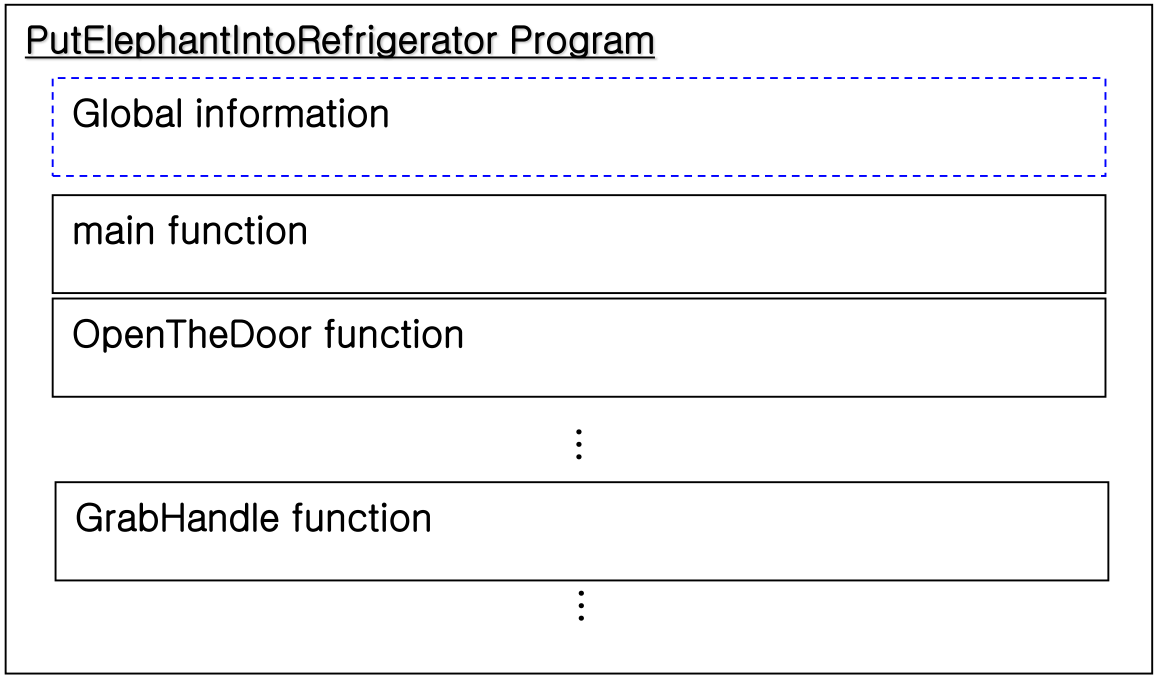
Structure of C Programs

- Observation
  - A large main task can be divided into one or more subtasks
    - If some subtasks are still large, each of them can be divided into even smaller subtasks.



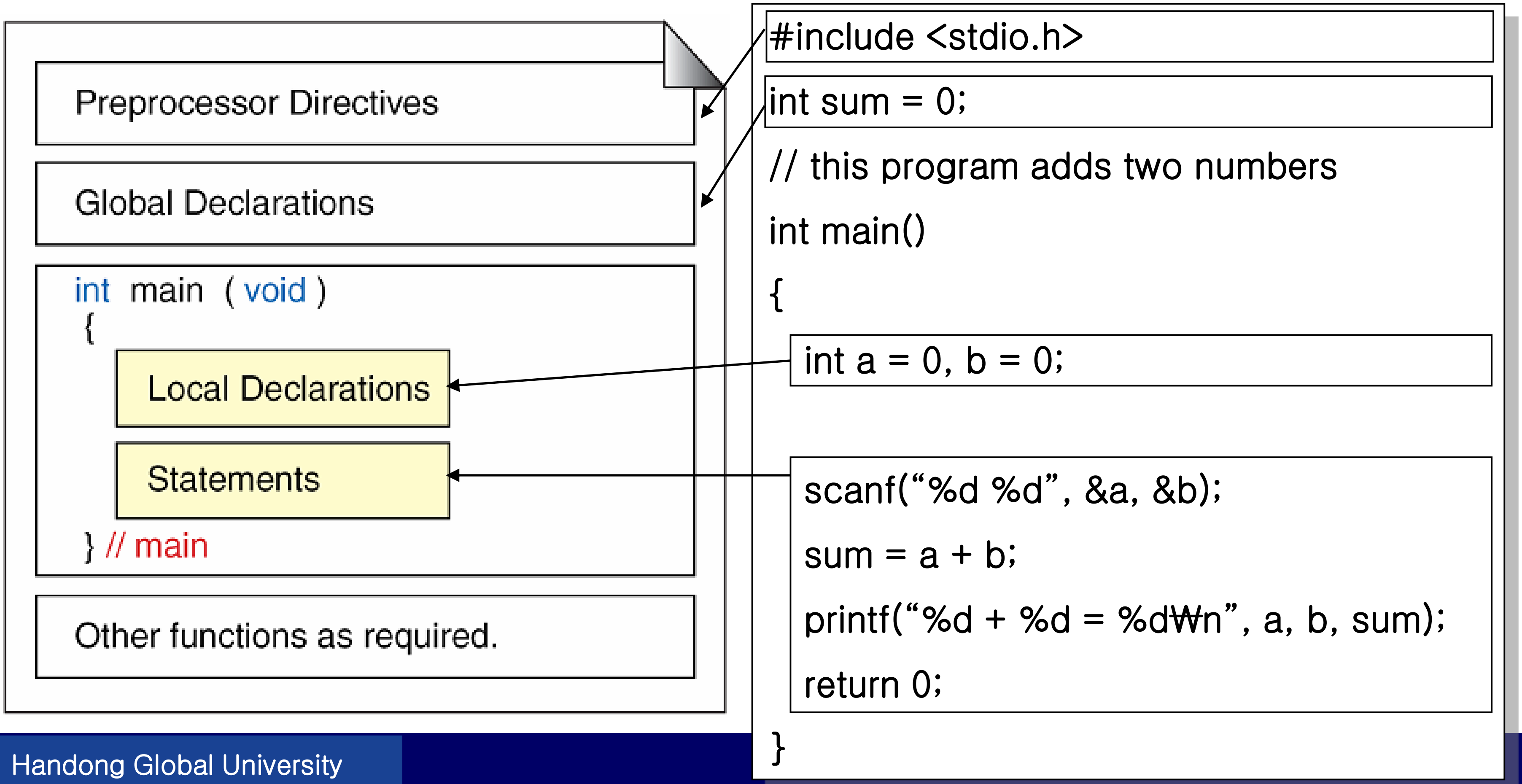
Structure of C Programs

- Main task can be implemented by writing functions for the subtasks and organizing them.
  - ➔ C program is composed of one or more functions
- Function (or subroutine): A portion of program within a larger program.
  - Performs a specific task, relatively independent of the remaining code.



Structure of C Programs

- A C program consists of preprocessor directives, global declarations and functions

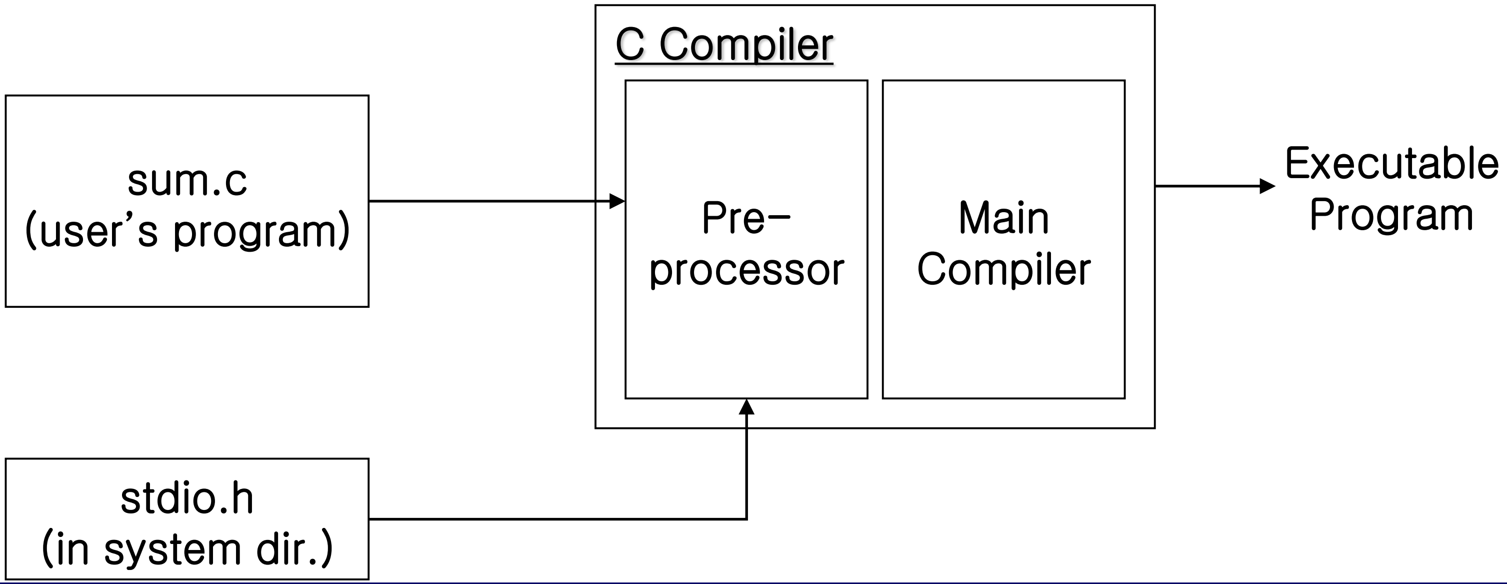


```
#include <stdio.h>
int sum = 0;
// this program adds two numbers
int main()
{
    int a = 0, b = 0;

    scanf("%d %d", &a, &b);
    sum = a + b;
    printf("%d + %d = %d₩n", a, b, sum);
    return 0;
}
```

Structure of C Programs

- Preprocessor directives
  - Indications for preprocessor

    Ex) "#include <stdio.h>" – indicate to include a file "stdio.h" at this position to use *printf* and *scanf* functions
    - This line is substituted by stdio.h
  - Starts with # sign



Declaration and Statement

- Declaration: Specification of identifier, type, and other aspects of variables or functions.
  - Used to announce the existence of a variable or function
  - In C language, all variables and functions should be declared before use.
- Statement: the smallest standalone element that specifies an action.
  - In C language, each statement terminates by semicolon (;).
  - A program or a function is formed by a sequence of one or more statements.

Ex)

```
...
int i, j;                    // declaration
i = 10;                      // statement
j = i * 2;                   // statement
...
```

Structure of C Programs

- Global declarations
  - Declarations of variables, functions, etc. visible in all functions
    - Indicating compiler some elements will be defined and used
    - Globally declared elements are visible in all functions

- Function definitions
  - Local declarations: declarations of variables, functions, etc. visible only in that function
    - Indicating compiler some elements will be defined and used
    - Locally declared elements are visible only in the corresponding function

  - Statements: actions that function does

Ex)
```
sum = a + b;    // add a and b and store the result into sum
printf("%d + %d = %d₩n", a, b, sum); // print values of a, b and sum
```

Analysis of Hello.c

- Line1: indicates pre-processor to include another file “stdio.h”
  - stdio.h should be include to use printf function in line 4
- Line2: function definition
  - main is the entry function
    - Each program should have a function named as “main”
    - Whenever a program is executed, the program starts from main function
- Line3 and 6: start and end of function main
- Line4: statement to print the greeting message on screen
- Line5: statement to terminate the program execution

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


2. int main(void)
3. {
4.     printf(“Hello World!₩n”);
5.     return 0;
6. }
```

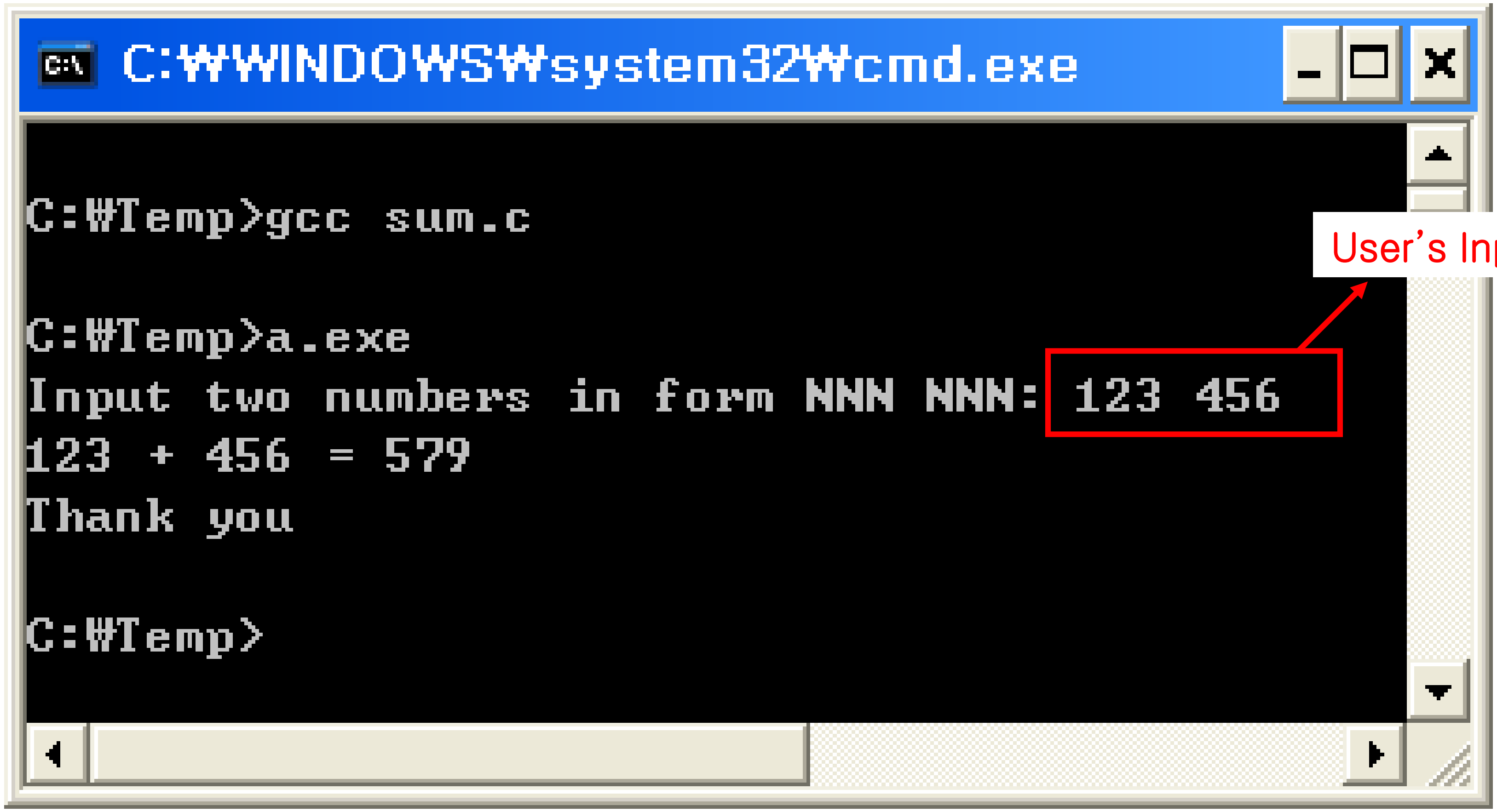
Example: Adding Two Numbers

- Add.c: a program that reads two numbers and prints their sum

```
#include <stdio.h>
int main(void)
{
    int a = 0, b = 0;
    int sum = 0;

    printf(“Input two numbers in form NNN NNN: ”);
    scanf(“%d %d”, &a, &b);
    sum = a + b;
    printf(“%d + %d = %d₩n”, a, b, sum);
    printf(“Thank you₩n”);

    return 0;
}
```

Example: Adding Two Numbers

- Result



```
C:₩Temp>gcc sum.c

C:₩Temp>a.exe
Input two numbers in form NNN NNN: 123 456
123 + 456 = 579
Thank you

C:₩Temp>
```

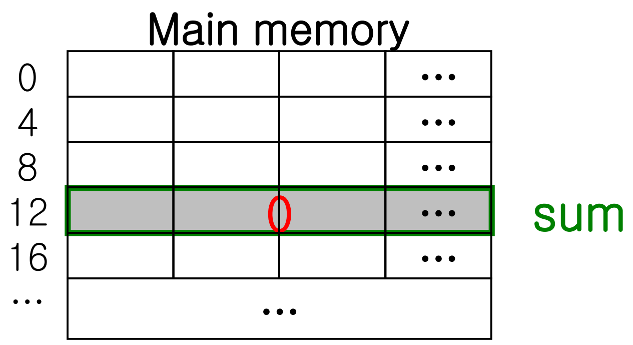
New Elements in Add.c

- Variable declaration (4, 5)
  - int sum = 0;
- Formatted input/output
  - printf(“%d + %d = %d₩n”, a, b, sum); // output (7,10,11)
  - scanf(“%d %d”, &a, &b); // input (8)
- Statement
  - sum = a + b; (9)

Variable Declaration

- Reserve a memory space and name it 'sum'
  - Variable: a memory space with a name

int sum = 0;



  - sum: variable name
  - int: type of *sum* (integer)
  - 0: an integer constant zero
  - = : initialization
    - "Put a value 0 into a variable *sum*"

Types

- Type defines a set of values and operations that can be applied to the values

- Important types in C
  - −1, 5, 7, 152, … ➔ integer numbers (int)
  - 34.2, −53.98, 3.14, … ➔ floating point numbers (float)
  - 'a', 'X', '5', '$' … ➔ single character (char)
  - "Hello", "Welcome", … ➔ string (char [])

    Ex) char mesg[30] = "Hello";

    maximum length

Types

Ex)

```
int i = 10;                        // int type

float f = 3.141592;                // float type

char c = ‘a’;                      // char type

char mesg[30] = “Hello”;           // string type
```

Formatted Input/Output

- Formatted input

```
scanf("%d %d", &a, &b);
```

  - Two integer numbers are read from keyboard and stored in *a* and *b*, respectively
    - & should precede the variable name except for string type

- Formatted output

```
printf("%d + %d = %d₩n", a, b, sum);
```

  - %d's are replaced by value of a, b and sum respectively.

Conversion Specifications

- Keyboard input: scanf
  - Integer
    ```
    int i;
    scanf("%d", &i);
    ```
  - Float
    ```
    float f;
    scanf("%f", &f);
    ```
  - Character
    ```
    char ch;
    scanf("%c", &ch);
    ```
  - String
    ```
    char str[100];
    scanf("%s",str);
    ```
- Display output: printf
  - Integer
    ```
    printf("value = %d₩n", i);
    ```
  - Float
    ```
    printf("value = %f", f);
    ```
  - Character
    ```
    printf("ch = %c₩n", ch);
    ```
  - String
    ```
    printf("str = %s₩n",str);
    ```

%d, %f, %c, %s are conversion specifications

scanf() vs. scanf_s()

- scan_s() is safer version of scanf()
  - New compilers support scanf_s() rather than scanf()
  - Old compilers don't support scanf_s()

- Reading a string

```
char buffer[30];
scanf("%s", buffer);
scanf_s("%s", buffer, 30);
```

Analysis of Add.c

- Analysis
  - 4~5: variable a, b, sum are declared
  - 7: display message
  - 8: receives two numbers from keyboard
  - 9: add content of a and b and store the result in sum
  - 10: display a, b, and sum
  - 11: display ending message

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int a = 0, b = 0;
5       int sum = 0;
6
7       printf(“Input two numbers in form NNN NNN: “);
8       scanf(“%d %d”, &a, &b);
9       sum = a + b;
10      printf(“%d + %d = %d₩n”, a, b, sum);
11      printf(“Thank you₩n”);
12
13      return 0;
14  }
```

Excises

- Write a program that convert Fahrenheit into Celsius.

  Hint: celsius = (fahr-32) * 5 / 9

- Write a program that read a name and print greeting message.

  Ex) What's your name?

  ```
  Shrek        → User's Input
  Nice to meet you, Shrek!
  ```

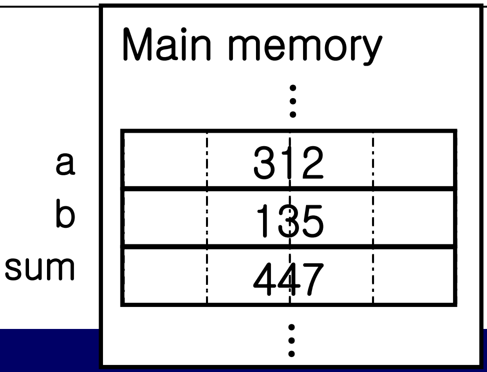
Agenda

- Background
- Compiling and Executing
- Structure of C Programs
- <u>Variables and Types</u>
- Constants
- Formatted Input/Output
- Identifiers
- Comments

Variables

- Variable: named memory location to store or read data of a particular type

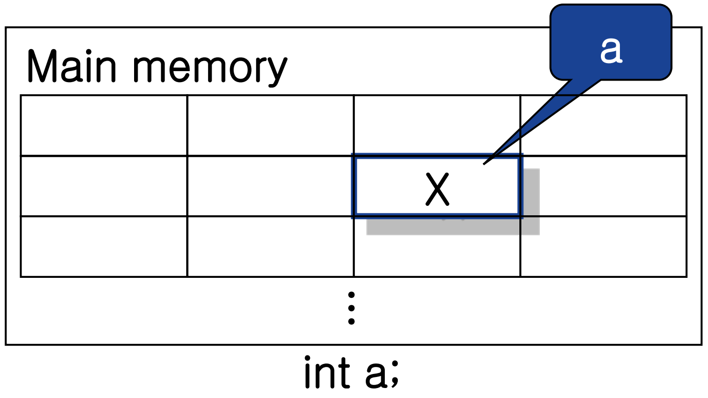
```
#include <stdio.h>

int main(void)
{
   int a = 0, b = 0, sum = 0;

   scanf("%d %d", &a, &b);                // input numbers into a and b
   sum = a + b;                           // add a and b and store it into sum
   printf("%d + %d = %d₩n", a, b, sum);   // print sum
   return 0;
}
```

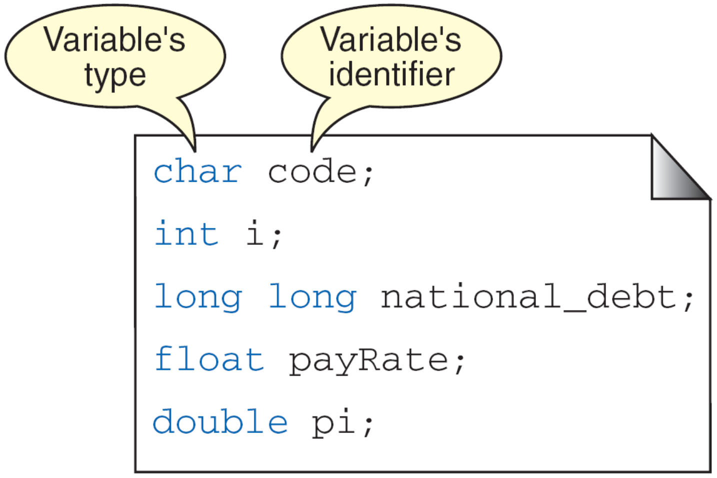


Variable Declaration

- Each variable should be declared before use
  - Variables are created when they are declared.
    - Reserves memory
    - Defines a symbolic name

```
Ex) int a;          // declaring a variable
    int b, c;       // declares two variables in a line
```



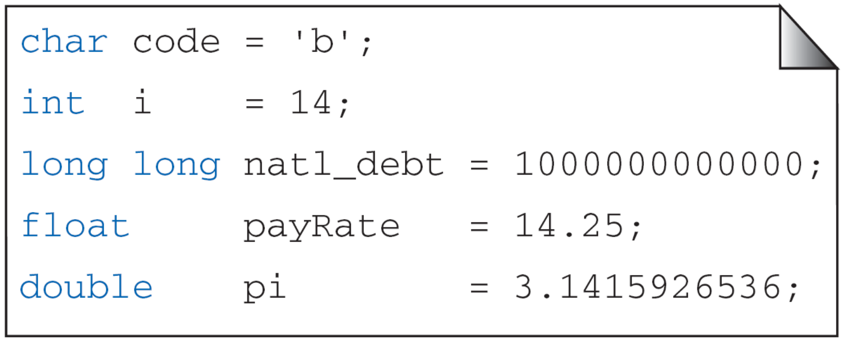


```
char code;
int i;
long long national_debt;
float payRate;
double pi;
```

Program

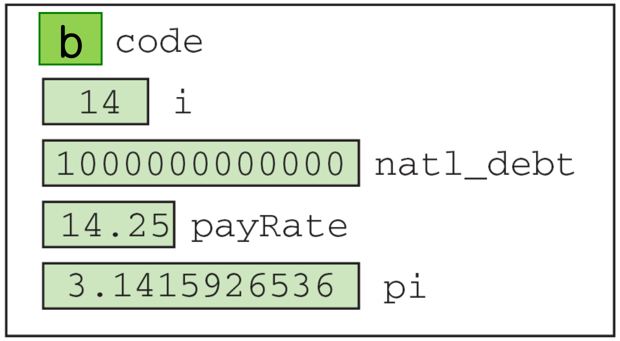
Variable Initialization

- Initialization: first assignment of a value to a variable



```
char code = 'b';
int  i    = 14;
long long natl_debt = 1000000000000;
float     payRate   = 14.25;
double    pi        = 3.1415926536;
```

Program



Memory

Note! If a variable is not initialized, its initial value is undefined. (It might have a random value)

➔ Not initialized variable can cause error extremely difficult to find.

Variable Initialization

- Not initialized variable

```
int main()
{
	int i = 0, j = 0;

	i = 100;		// What if this line is missed?
	j = i + 2;
	printf("j = %d₩n", j);

	return 0;
}
```

Recommendation

- To build a stable program, all variables should be initialized to when they are declared.
  - Valid, but not desirable

```
int a = 10, b = 20;
int sum;                  // not initialized
...
sum = a + b;               // what happens this line is omitted?
MyFunc(sum);
```

→ Result may depend on undefined value

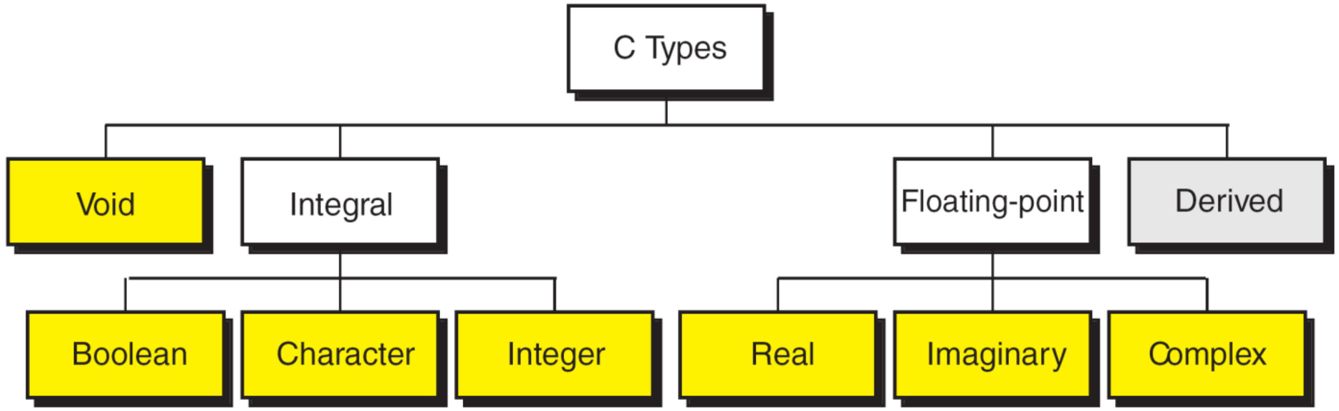
  - Redundant, but safe

```
int a = 10, b = 20;
int sum = 0;              // initialization is not necessary,
                          // but desirable for safety
...
sum = a + b;              // first assign
MyFunc(sum);
```

→ Produces consistent result

Types

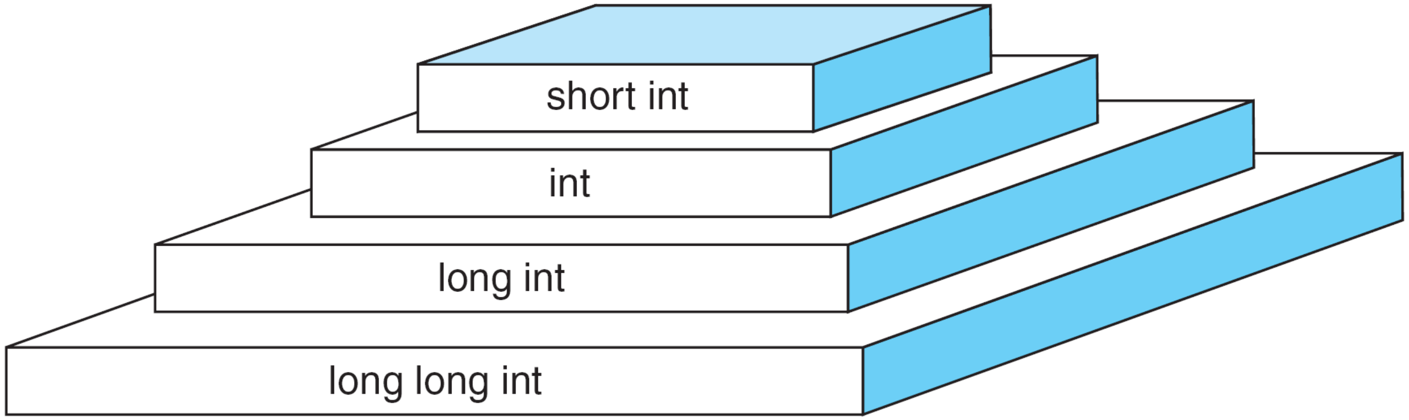
- Types in C language



- Each type is different from others in actual representation, size, and operations applicable to it.

Integral Types

- Integer types: types for numbers without fraction parts
  - short int (= short)
  - int
  - long int (= long)
  - long long int (= long long)



Integral Types

- Types of integers determine the size of the storage and the range of values

  Ex) assume int type takes 4 bytes

  ➔ it can represent 2^{32} different values (-2^{31} ~ $2^{31}-1$)

- Relative sizes of integer types
  - Note: actual size of each type depends on H/W and compiler
  - sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)

Type	Byte Size	Minimum Value	Maximum Value
short int	2	–32,768 (-2^{15})	32,767 ($2^{15}-1$)
int	4	–2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31}-1$)
long int	4	–2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31}-1$)
long long int	8	–9,223,372,036,854,775,807 (-2^{63})	9,223,372,036,854,775,806 ($2^{63}-1$)

Character Type

- Character types: types for letters

  Ex) char c = 'a';

  In computer, each letter is represented by a number.

  - 'a': 0x61, 'b': 0x62, …,'e': 0x65, …, 'h': 0x68, …, 'l': 0x6C, …, 'o': 0x6F 'z': 0x7A, …

  Ex) "hello" ➔ 68 65 6C 6C 6F 00 (in hex)

  cf. 'a' + 1 makes 'b'

  'a' + 1 = 61 + 1 = 62 = 'b'

```
printf("char = [%c]₩n", 'a');      // char = [a]
printf("code = %d₩n", 'a');        // code = 97 ( = 0x61)
```

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	TAB	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	{	|	}	~	

- Each character is represented by 7 bits
  - 0x00~0x1f: control characters
  - 0x20~0x7f: printable characters

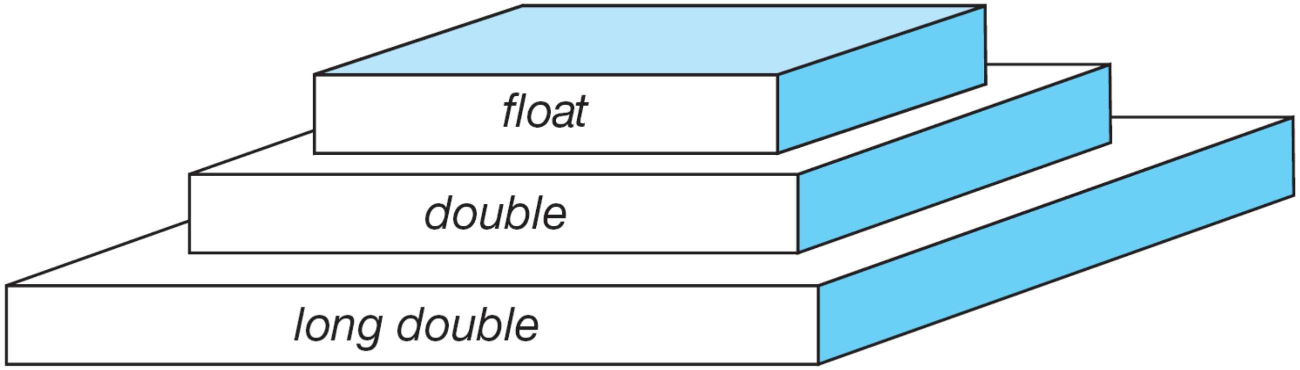
    Ex) 'a' = 0x61, '0' = 0x30, '-' = 0x2D

Floating-Point Types

- Floating-point types: types for fractional numbers

  Ex) float f = 0.5;

  - float, double, long double
- Relative sizes of floating-point types
  - Actual size of each type depends on machine and compiler
  - sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)



Other Types

- unsigned integer
  - A variable of unsigned types can store only non-negative values, but it can store larger numbers.
    - short i; // $-32768(=-2^{15}) \sim 32767(=2^{15}-1)$
    - unsigned short u; // $0 \sim 65535(=2^{16}-1)$
- bool: type for binary value true or false (C99)

  Ex) bool b = true;
  - int type is still more frequently used than boolean type
- complex/imaginary (C99)
- void: no value, no operation

Agenda

- Background
- Compiling and Executing
- Structure of C Programs
- Variables and Types
- <u>Constants</u>
- Formatted Input/Output
- Identifiers
- Comments

Constants

- Constant: data values that cannot be changed during execution
  - Integer constants

    Ex) 0, 100, +123, -378
  - Floating-point (real) constants

    Ex) 0.5, 3.141592, -5.0, 10., .78
  - Character constants

    Ex) 'A', 'b', '0', '+'
  - String constants

    Ex) "Hello", "h", "HOW ARE YOU?"

```
Ex) int i = 10;          // initialize i with integer constant 10
    float f = 10.5;      // initialize f with real constant 10.5
```

Postfixes for Integer/Float Constants

- Postfixes to specify a particular integral type
  - Integer number without postfix: int
  - U or u: unsigned
  - L or l: long
  - LL or ll: long long int

  Ex) +123, -32271L, 76542LU, 12789845LL

- Postfixes to specify a particular floating-point type
  - floating-point number without postfix: double
  - F or f: float
  - L or l: long double

  Ex) 2., .0, 3.1416, -2.0f, 3.1415926536L

  Recommendation: avoid to use l (use L instead)

Character/String Constants

- Character constants: single character enclosed by single quotes
  Ex) 'A', 'b', '0', '+'
- String constant: a sequence of zero or more characters enclosed in double quotes.
  Ex) "Hello₩n", "HOW ARE YOU?", "h", ""

- Backslash (\ or ₩) has a special use to represent non-graphical characters
  - printf("Hello, World₩n");
  - ➔ Hello, World↵
  - How to print [He said "Hello."]?
    - printf("He said "Hello."₩n");   // Error!!
    - printf("He said ₩"Hello.₩"₩n");   // OK!

Character/String Constants

- Special characters represented by escape sequence

ASCII Character	Symbolic Name
null character	`'\0'`
alert (bell)	`'\a'`
backspace	`'\b'`
horizontal tab	`'\t'`
newline	`'\n'`
vertical tab	`'\v'`
form feed	`'\f'`
carriage return	`'\r'`
single quote	`'\''`
double quote	`'\"'`
backslash	`'\\'`

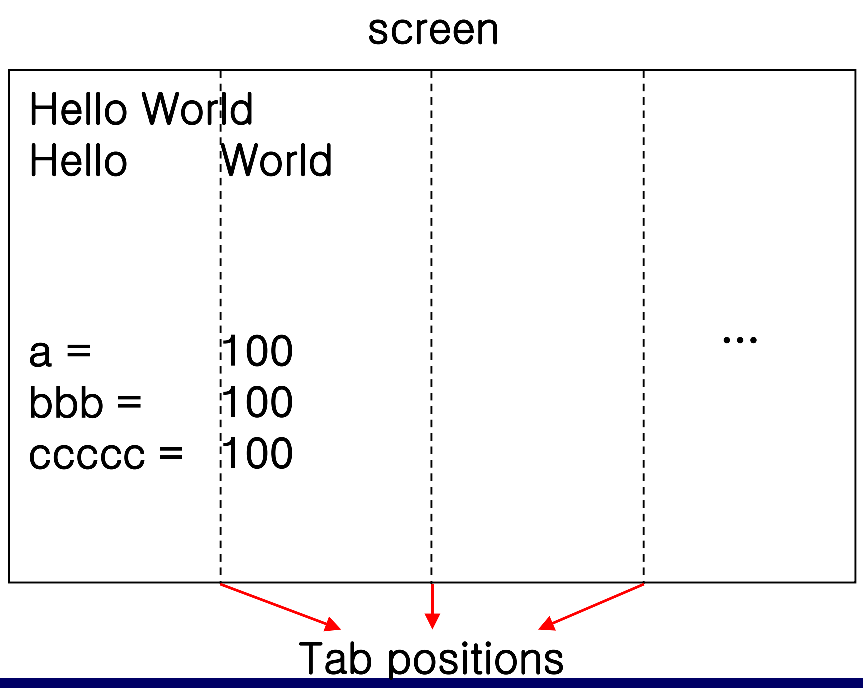
Character/String Constants

- Tab character '₩t'
  - Conceptually, the screen is vertically divided into tabs.
    - Tab size depends on systems but typically 8 or 4 columns
  - '₩t' indicates to print the following characters from the next tap position.

```
Ex) printf("Hello World₩n");
    printf("Hello₩tWorld₩n");
```

  - Useful to make an alignment

```
printf("a =₩t%d₩n", a);
printf("bbb =₩t%d₩n", b);
printf("ccccc =₩t%d₩n", c);
```



```
Hello World
Hello   World

a =     100
bbb =   100
ccccc = 100
```

Alternative Categorization of Constants

- Literal constants: unnamed constant to specify data

  Ex) a = b + 5;

- Defined constants

  ```
  Ex) #define SALES_TAX_RATE .0825
      float tax_rate = SALES_TAX_RATE;
  ```

  → Every occurrence of SALES_TAX_RATE is replaced by 0.0825

  Note! #define is preprocessor directive

- Memory constants: similar to variables except its value cannot be changed
  - Syntax: const *type identifier = value;*
    - Note! A memory constants has its own type

  ```
  Ex) const float cPi = 3.141592;        // cPi cannot be changed
      float Pi_2 = cPi / 2;              // Pi_2 can be changed
  ```

Agenda

- Background
- Compiling and Executing
- Structure of C Programs
- Variables and Types
- Constants
- <u>Formatted Input/Output</u>
- Identifiers
- Comments

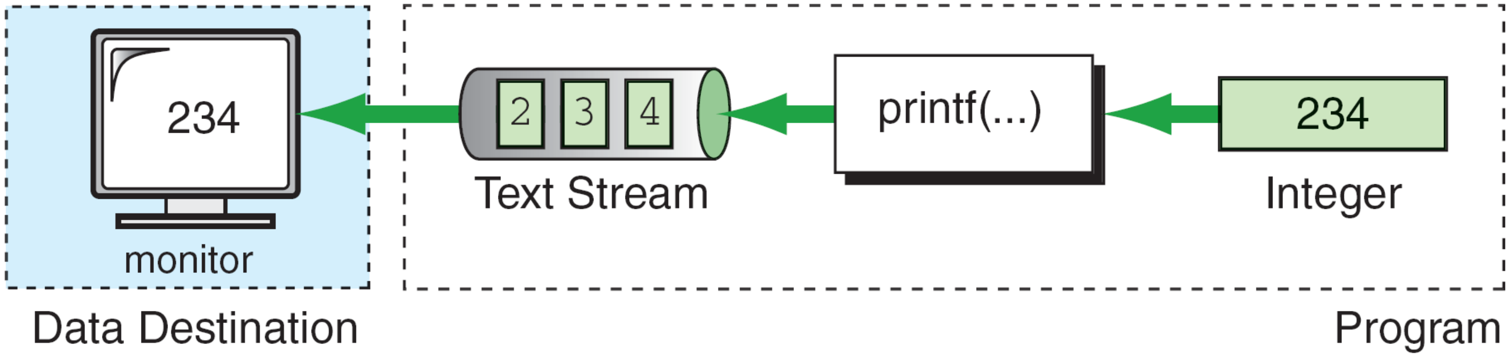
Example

```
#include <stdio.h>

int main()
{
    int i = 0;
    char ch = 0;
    char str[100];

    printf("Input an integer: ");
    scanf(“ %d", &i);
    printf(“₩tInteger value = %d₩n", i);

    printf("Input a character: ");
    scanf(“ %c", &ch);
    printf(“₩tCharacter value = %c₩n", ch);

    printf("Input a string: ");
    scanf(“ %s", str);
    printf(“₩tString value = %s₩n", str);

    return 0;
}
```

Formatted Output

- Formatted output: printf
  - Monitor can display only text characters
    - Text data can be displayed directly, but numeral data requires formatting.
  - Formatting: converting values to text stream

    ex) 234 (integer) → '2', '3', '4' (character sequence)



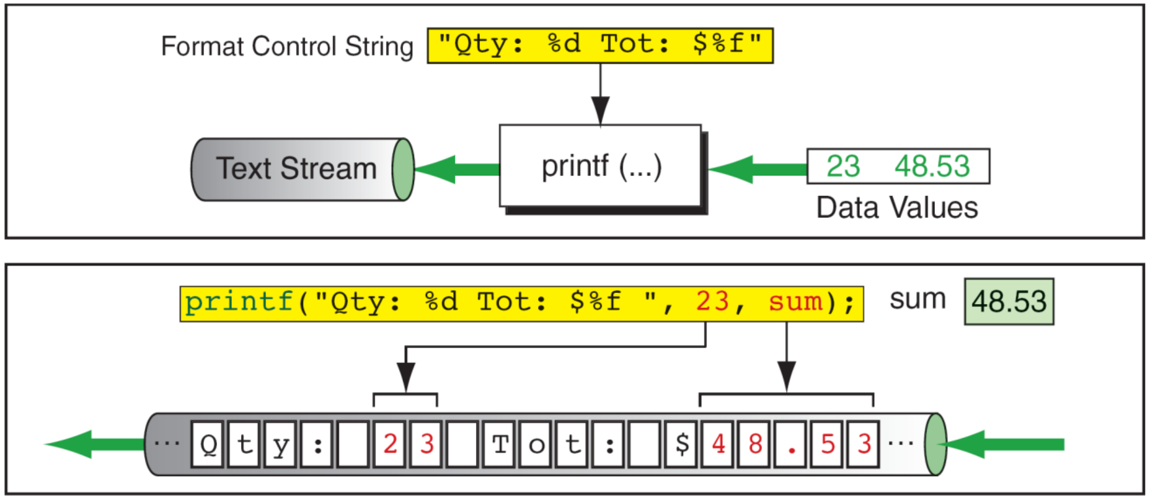
Formatted Output

- Syntax: printf(format_string, v_0, v_1, …);
  - format_string: text string containing zero or more conversion specifications
  - Each of conversion specifications is replaced by v_i's
    - v_i: value to replace i-th conversion specification

Ex) printf("Qty: %d Tot: $%f", 23, sum);   // sum = 48.53F

→ Qty: 23 Tot: $48.53

(a) Basic Concept

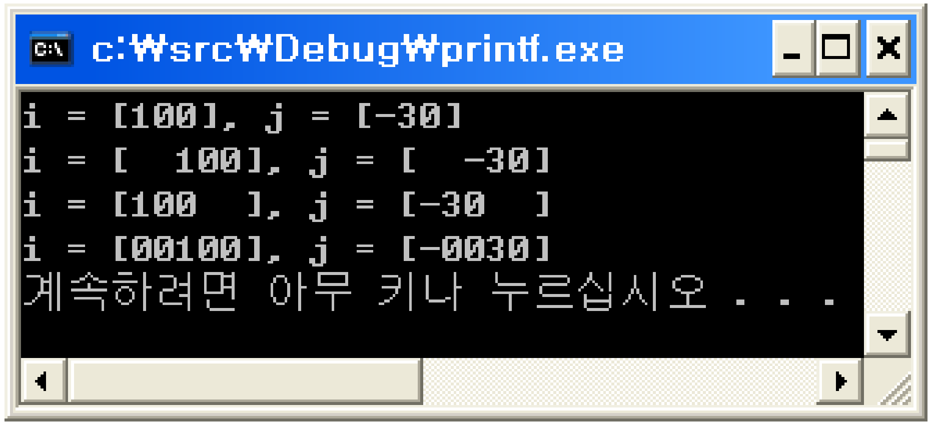


(b) Implementation

Example

- print.c

```
#include <stdio.h>

int main()
{
    int i = 100, j = -30;

    printf("i = [%d], j = [%d]₩n", i, j);
    printf("i = [%5d], j = [%5d]₩n", i, j);
    printf("i = [%-5d], j = [%-5d]₩n", i, j);
    printf("i = [%05d], j = [%05d]₩n", i, j);

    system("PAUSE");

    return 0;
}
```



```
i = [100], j = [-30]
i = [  100], j = [  -30]
i = [100  ], j = [-30  ]
i = [00100], j = [-0030]
계속하려면 아무 키나 누르십시오 . . .
```

Formatted Output

- Conversion specification of printf

%	Flag	Minimum Width	Precision	Size	Code

Ex) %d, %ld, %f, %Lf, %c, %s

- Format codes for output

Type	Size[a]	Code	Example
char	None	c	%c
short int	h	d	%hd
int	None	d	%d
long int	l	d	%ld
long long int	ll	d	%lld
float	None	f	%f
double	None	f	%f
long double	L	f	%Lf

* Integer in hexadecimal format: %x
* string : %s,   * pointer: %p

Formatted Output

%	Flag	Minimum Width	Precision	Size	Code

- Size modifier: specifies type of conversion type
  - h(short), l(long), ll(long long), L(long double)

  Ex) printf("%ld", 7382949L);

  printf("%Lf", 314159265.3578L);

- Width modifier: specifies minimum width

  Ex) printf("[%5d]", 123); // [□□123]

  - If date requires more space, then printf overrides width modifier

  Ex) printf("[%3d]", 12345); // [12345]

Formatted Output

%	Flag	Minimum Width	Precision	Size	Code

- Precision modifier: specifies # of decimal places (for floating-point numbers)
  - Syntax: *n.m* (m decimal digits among n total positions)
    - n: width modifier (# of total positions)
    - m: precision modifier (# of decimal digits)

```
Ex) printf("[%7.2f], 123.456);      // [□123.46]
```

Formatted Output

%	Flag	Minimum Width	Precision	Size	Code

- Flag modifier: justification, padding, sign, etc.
  - Justification
    - printf(“[%10d]₩n”, 123); // [□□□□□□□123]
    - printf(“[%−10d]₩n”, 123); // [123□□□□□□□]
  - Padding
    - printf(“[%d010d]₩n”, 123); // [0000000123]
  - Sign
    - printf(“[%10d]₩n”, 123); // [□□□□□□□123]
    - printf(“[%+10d]₩n”, 123); // [□□□□□□+123]
    - printf(“[%□d]₩n”, 123); // [□123]
    - printf(“[%□d]₩n”, −123); // [−123]

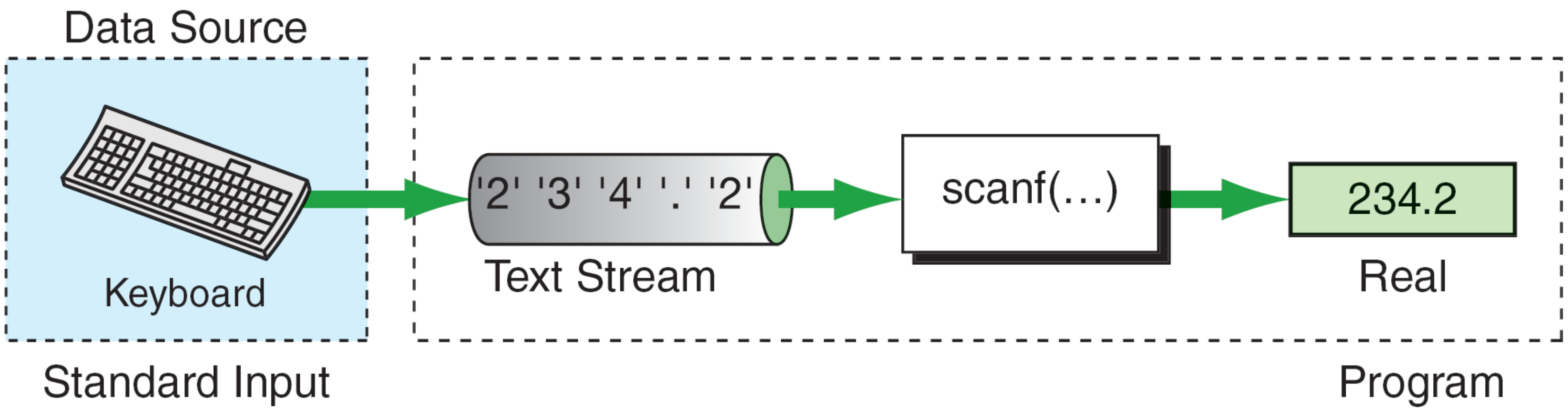
Examples

- Write the outputs of the following sentences
  - printf(“%d%c%f”, 23, ‘z’, 4.1);
  - printf(“This number is %6d.”, 23);
  - printf(“The tax is %08.2f this year.”, 233.12);
  - printf(“₩”%8c   %d₩””, ‘h’, 23);
- Describe the problem of each sentence and say what the result would be like.
  - printf(“%d %d %d₩n”, 44, 55);
  - printf(“%d %d₩n”, 44, 55, 66);
  - float x = 123.45;
    printf(“The data are: %d₩n”, x);

Formatted Input

- Formatted input: scanf
  - Inputs from the keyboard are sequences of characters
    - Value should be extracted from the text stream

    Ex) ‘2’, ‘3’, ‘4’, ‘.’, ‘2’ (character sequence) → 234.2 (float)
  - Function of scanf is the reverse of printf



Formatted Input

- Syntax: scanf(format_string, a_0, a_1, …);

  Ex) scanf("id = %d", &i);

  - format_string: data to be extracted from stream and reformatted
    - format_string contains zero ore more conversion specifications
  - a_0, a_1: variable addresses for conversion specifications
  - The portion matched to a conversion specification is converted and stored in the corresponding variable address

    Ex) scanf("id = %d", &i);

    input: "id = 392"

    → integer value 392 will be stored in variable i

Formatted Input

- Variable address: location of memory occupied by the variable
  - Address of a variable can be acquired by attaching address operator & before a variable name

    ```
    Ex) int a, b;  // addresses of a and b are &a and &b, respectively
        char c;    // address of c is &c
        scanf("%d %d %c", &a, &b, &c);
    ```

  - Array names (including char string) can be used for variable address as itself

    ```
    Ex) char inputString[100];
        scanf("input = %s", inputString);      // & is not required
        Given an input "input = Thanks", "Thanks" will be stored in
      inputString
    ```

→ Variables should be prefixed with & to get an input through *scanf*. But character string is an exception.

Variable Address

- Example: address.c

```
#include <stdio.h>

int main()
{
  int i = 0;
  printf("i = %d, &i = %p₩n", i, &i);

  return 0;
}

// result: i = 0, &i = 0012FF44
```

Formatted Input

- Requirement for successful conversion
  - Non-conversion specification characters must be exactly matched by the characters in input stream.

    Ex) To be matched with ***scanf("id = %d", &i);***, the input should be *"id = XXX"*

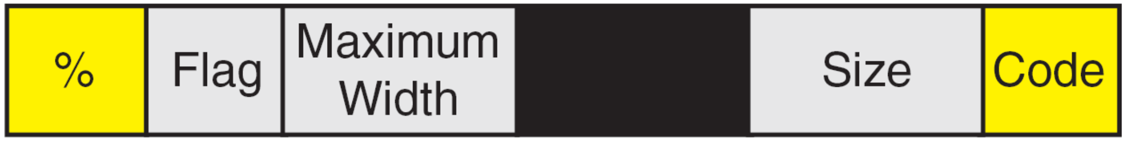
  - A whitespace in format string can match with zero or more white spaces.

    Ex) ***scanf("□id = %d", &i);*** accepts *"□□□id = XXX"*

  - Action of conversion is determined by the conversion specification and type of the variable.

Formatted Input

- Conversion specification of *scanf*



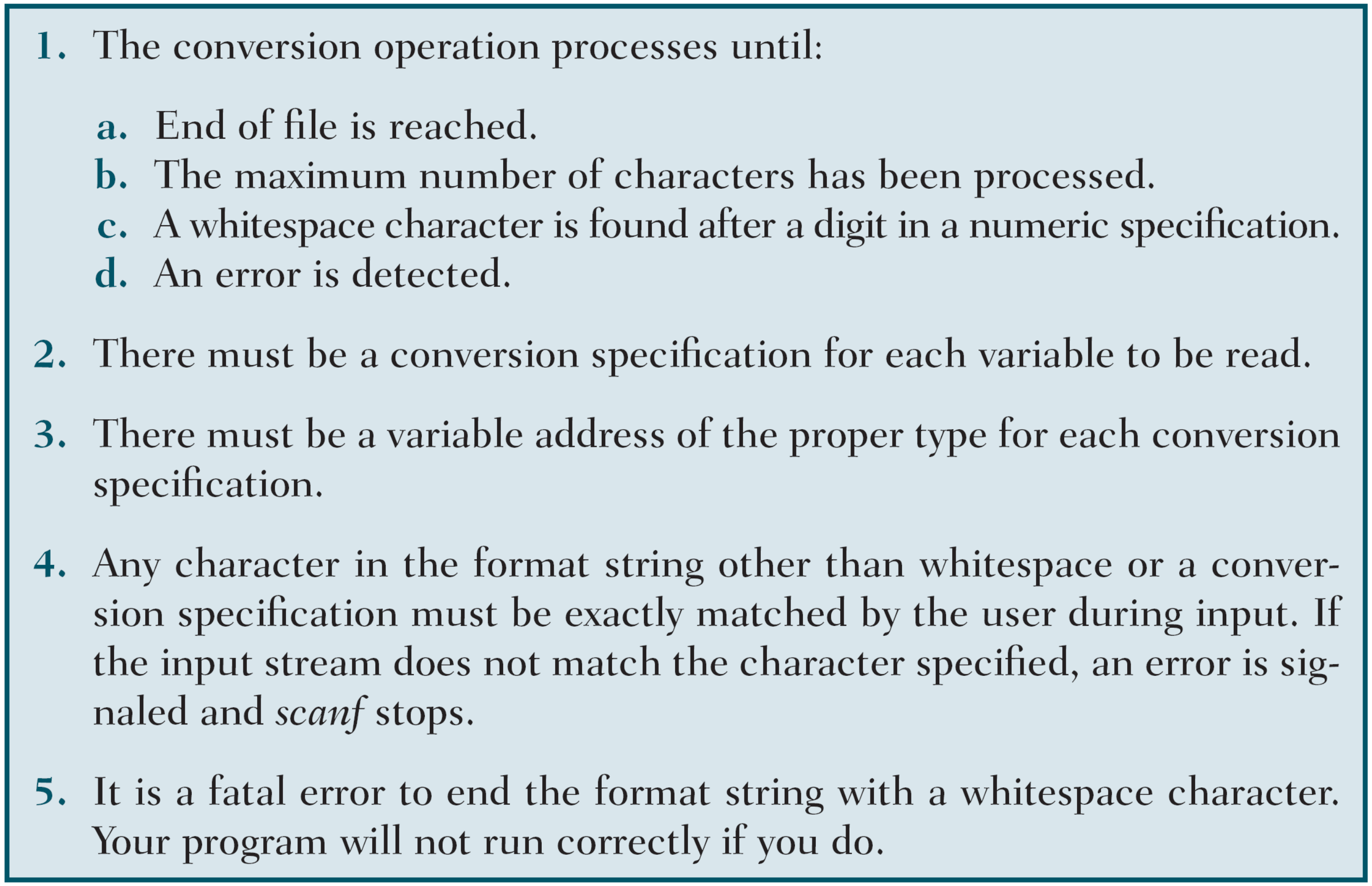
- Similar to that of printf but three differences
  - Precision modifiers are not allowed
  - Only one flag modifier: assignment suppression flag (*)

    Ex) scanf("%d %*c %f", &x, &y);

    // portion of input text matched with %c is just matched and discarded
  - Width modifier does not represent minimum, but **maximum width**

Formatted Input

- Matching rule of *scanf*



1. The conversion operation processes until:
   a. End of file is reached.
   b. The maximum number of characters has been processed.
   c. A whitespace character is found after a digit in a numeric specification.
   d. An error is detected.
2. There must be a conversion specification for each variable to be read.
3. There must be a variable address of the proper type for each conversion specification.
4. Any character in the format string other than whitespace or a conversion specification must be exactly matched by the user during input. If the input stream does not match the character specified, an error is signaled and *scanf* stops.
5. It is a fatal error to end the format string with a whitespace character. Your program will not run correctly if you do.

Examples

- Describe the problem of each sentence and how to fix it.

  -
    ```
    int a = 0;
    scanf(“%d”, a);            // input: “234”
    printf(“%d”, a);
    ```

  -
    ```
    scanf(“%d %d %d”, &a, &b);
    ```

  -
    ```
    float a = 2.1;
    scanf(“%5.2f”, &a);        // input: “74.35”
    printf(“%5.2f”, a);
    ```

Examples

- Write the result of the following sentences and input.
  - scanf(“%d%d%d%c”, &a, &b, &c, &d);
    - input: “214 156 14Z”
    - input: “214 156 14 Z”   // space is not discarded by %c
  - scanf(“%d %d %f”, &a, &b, &c);
    - input: “2314 15 2.14”
  - scanf(“%d-%d-%d”, &year, &month, &date);
    - Input: “2006-09-01”

Formatted Input

- Leading whitespaces

```
// whitespace.c
#include <stdio.h>

int main()
{
    char c1 = 0;
    char c2 = 0;

    scanf(" %c", &c1);
    printf("c1 = %c₩n", c1);
    scanf(" %c", &c2);
    printf("c2 = %c₩n", c2);

    return 0;
}
```

Agenda

- Background
- Compiling and Executing
- Structure of C Programs
- Variables and Types
- Constants
- Formatted Input/Output
- Identifiers
- Comments

Identifiers

- Identifiers: user-specified names of functions, variables, tags or members of structures/unions, enumeration constants, type names and objects

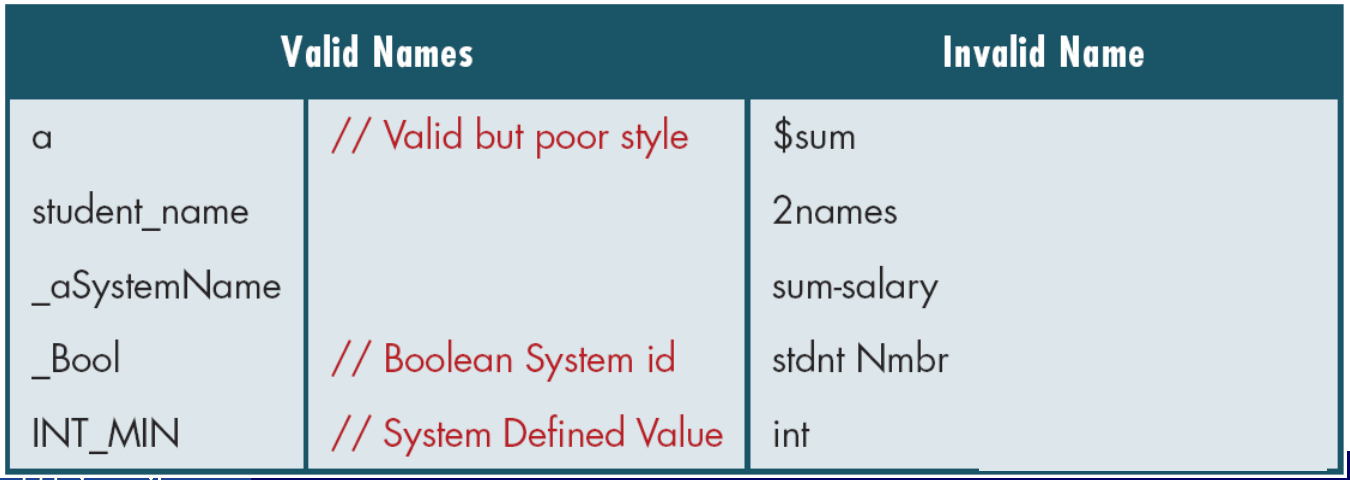
  Cf. keywords (reserved words) syntactical words whose meanings are pre-defined in C language (if, while, int, void, …)

```
#include <stdio.h>
int main()
{
   int a, b;

   scanf("%d %d", &a, &b);
   printf("%d + %d = %d₩n", a, b, a + b);
   return 0;
}
```

Identifiers

- Syntactic rules for valid identifier
  - First character must be alphabetic character or underscore
    - ab10: O, _ab10: O, 10ab: X
  - Must consist only of alphabetic characters, digits, or underscore
  - First 63 characters of identifier are significant.
  - Cannot duplicate a keyword.

    Ex) int is not allowed
- Examples of valid and invalid identifiers



Valid Names		Invalid Name
a	// Valid but poor style	$sum
student_name		2names
_aSystemName		sum-salary
_Bool	// Boolean System id	stdnt Nmbr
INT_MIN	// System Defined Value	int

Comments

- Comment: internal documentation for programmers.
  - Comments are meaningful only to human. (compilers ignore comments)
  - Line comment / block comment

- Line comment
  - Starts with double slash (//) and ends at the end of line

Ex)

```
// This is a whole line comment

a = 5;          // This is a partial line comment
```

```
#include <stdio.h>
int sum = 0;

// this program adds two numbers
int main(void)
{
   int a, b;

   scanf("%d %d", &a, &b);
   sum = a + b;
   printf("%d + %d = %d₩n", a, b, sum);
   return 0;
}
```

Comments

- Block comment: starts with /* and ends with */
  - All characters between /* and */ are ignored by the compiler

Ex)

```
/* This is a block comment that
   covers two lines.             */


/*
** It is a very common style to put the opening token
** on a line by itself, followed by the documentation
** and then the closing token on a separate line. Some
** programmers also like to put asterisks at the beginning
** of each line to clearly mark the comment.
*/
```

  - Note! Nested comments are not allowed
    - Once the compiler sees /*, it ignores all chars until it sees */

Ex) /* =============================== */

Ex) /* ========= /* ------- */ ========= */

ignored

Error!