

Introduction to Scientific and Engineering Computation (BIL 104E)

Lecture 3

Data Types and Keywords & Handling Standart (Input/Output)

C Keywords

The C language reserves certain words (**C keywords**) that have special meanings to the language.

You should not use the **C keywords** as an **identifiers** such as variable, constant, or function names.

<u>Keyword</u>	<u>Description</u>
auto	Storage class specifier / modifier
const	
extern	
register	
static	
volatile	

<u>Keyword</u>	<u>Description</u>
default	Label
sizeof	Operator

C Keywords

<u>Keyword</u>	<u>Description</u>
char	Type specifier
double	
float	
register	
int	
long	
short	
signed	
struct	
union	
unsigned	
void	

<u>Keyword</u>	<u>Description</u>
break	Statement
case	
continue	
do	
else	
for	
goto	
if	
return	
switch	
typedef	
while	

All **C keywords** are written in lowercase letters. **int** is considered as a **C keyword**, but **INT** is not. Because C is a case sensitive language

Data Types: char

An object of the char data type represents a single character of the character set used by your computer.

Decimal	Hex	Binary	Value	Explanation
048	30	00110000	0	
049	31	00110001	1	
050	32	00110010	2	
051	33	00110011	3	
052	34	00110100	4	
053	35	00110101	5	
054	36	00110110	6	
055	37	00110111	7	
056	38	00111000	8	
057	39	00111001	9	
058	3A	00111010	:	Colon
059	3B	00111011	;	Semi-colon
060	3C	00111100	<	Less than
061	3D	00111101	=	Equal sign
062	3E	00111110	>	Greater than
063	3F	00111111	?	Question mark
064	40	01000000	@	AT symbol

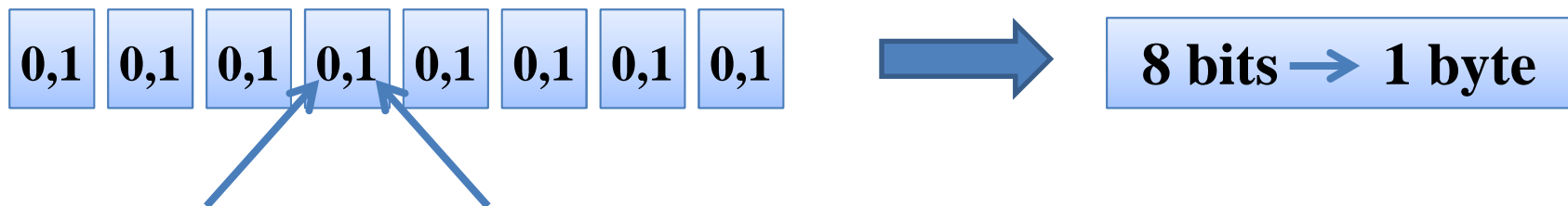
Data Types: char

Decimal	Hex	Binary	Value	Explanation
065	41	01000001	A	
066	42	01000010	B	
067	43	01000011	C	
068	44	01000100	D	
069	45	01000101	E	
070	46	01000110	F	
071	47	01000111	G	
072	48	01001000	H	
097	61	01100001	a	
098	62	01100010	b	
099	63	01100011	c	
100	64	01100100	d	
101	65	01100101	e	
102	66	01100110	f	
103	67	01100111	g	
104	68	01101000	h	

Data Types: char

A computer can only store numeric code (**binary code**). Therefore, characters such as **A**, **a**, **B**, **b**, and so on all have a unique numeric code.

Usually, a **character** takes **8 bits (1 byte)** to store its numeric code.



There are two possibilities (**0 and 1**) for each bit area.

$2^8 = 256$ Different characters can be represented by 1 byte.

Data Types: Character Variables

A **variable** that can represent different **characters** is called a **character variable**

C syntax

```
char    variable_name;
```

C syntax

```
char    variable_name1;  
char    variable_name2;  
char    variable_name3;
```

C syntax

```
char    variable_name1, variable_name2, variable_name3;
```

For instance:

```
char MyCharacter = 'A';
```

This statement declares **MyCharacter** and sets it to 'A'.

```
char x, y, z;
```

```
x = 'A';
```

```
y = ';';
```

```
z = '7';
```

x

65

y

59

z

55

This statements declare **x**, **y**, and **z** as **character** variables and then assign values to them.

Data Types: Character Constants

A **character** enclosed in single **quotes** (') is called a **character constant**.

Character constants are always surrounded by single **quote** characters (') while a string of more than one **character** uses the double **quote** (")

For instance:

```
x = 'A';
```

```
x = 65 ;
```

```
y = 'a' ;
```

```
y = 97 ;
```

Two assignment statements are equivalent for each declaration couple.

```
y = a ;
```

This statement assigns value contained in variable **a** into variable **y**.

Data Types: The Escape Character (\)

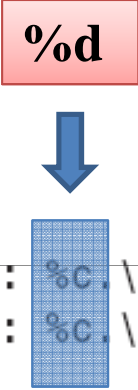
The **backslash** (\) is called the **escape character** in the **ASCII character** set.

The **escape character** is used in the C language to tell the computer that a special **character** follows.

<i>Character</i>	<i>Description</i>
<code>\b</code>	The backspace character; moves the cursor to the left one character.
<code>\f</code>	The form-feed character; goes to the top of a new page.
<code>\r</code>	The return character; returns to the beginning of the current line.
<code>\t</code>	The tab character; advances to the next tab stop.

Data Types: Printing Characters

```
1:  /* 04L01.c: Printing out characters */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      char c1;
7:      char c2;
8:
9:      c1 = 'A';
10:     c2 = 'a';
11:     printf("Convert the value of c1 to character: %c.\n", c1);
12:     printf("Convert the value of c2 to character: %c.\n", c2);
13:     return 0;
14: }
```



The diagram illustrates the conversion of character literals to integer values. A red box containing the format specifier `%d` has a blue arrow pointing down to a blue box containing the character `'a'`. This indicates that the character `'a'` is being converted to its corresponding integer value (97) when printed using the `%d` format specifier.

Convert the value of c1 to character: A.
Convert the value of c2 to character: a.

Convert the value of c1 to character: 65.
Convert the value of c2 to character: 97.

Data Types: int

The **int** keyword is used to specify the type of a **variable** as an **integer**.

Integer numbers are also called whole numbers, which have no **fractional** part or **decimal** point.

An **integer** is 32 bits long, which means that the range of an **integer** is from **2147483647** to **-2147483648**.

$$(2^{31} - 1) > \text{int } number > -(2^{31})$$

$$2147483647 > \text{int } number > -2147483648$$

Data Types: Integer Variables

A **variable** that can represent different **integers** is called a **integer variable**

C syntax

```
int    variable_name;
```

C syntax

```
int    variable_name1;  
int    variable_name2;  
int    variable_name3;
```

C syntax

```
int    variable_name1, variable_name2, variable_name3;
```

For instance:

```
int MyInteger = 2314;
```

This statement declares **MyInteger** as an **integer** variable and assigns it a value.

```
int A, a, B, b;
```

```
A = 37;
```

```
a = -37;
```

```
B = -2418;
```

A

37

a

-37

B

-2418

This statements declare **A**, **a**, and **B** as **integer** variables and then assign values to them.

Data Types: float

The **float** keyword is used to specify the type of a **variable** as a **floating-point** number .

A **floating-point** number contains a decimal point. A **floating-point** number is also called a **real number**.

For instance, 7.01 is a floating-point number.

A **floating-point** number is represented by taking 32 bits (1 bit for sign, 8 bits for **exponent**, 23 bits for **mantissa**).

$$3.402823466e + 38 > \text{float } number > 1.175494351e - 38$$

A **floating-point** number in C is at least six digits of **precision**.

Unlike an integer division, a **floating-point** division produces another **floating-point** number.

Data Types: Floating-point Variables

A **variable** that can represent different **floating-points** is called a **float variable**

C syntax

```
float    variable_name;
```

C syntax

```
float    variable_name1;  
float    variable_name2;  
float    variable_name3;
```

C syntax

```
float    variable_name1, variable_name2, variable_name3;
```

For instance:

```
float MyFloat = 3.14;
```

This statement declares **MyFloat** as a **float** variable and assigns it a value.

```
float a, b, c;
```

```
a = 10.38;
```

```
b = -32.7;
```

```
c = 12.0f;
```

Float constant declaration, Otherwise compiler assumes this constant is double.

This statements declare **a**, **b**, and **c** as **float** variables and then assign values to them.

Data Types: The Floating-Point Format Specifier (%f)

```
1:  /* 04L04.c: Integer vs. floating-point divisions */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      int int_num1, int_num2, int_num3;    /* Declare integer
7:      float flt_num1, flt_num2, flt_num3; /* Declare floating
8:      -point variables */
9:
10:     int_num1 = 32 / 10;    /* Both divisor and dividend are
11:     integers */
12:     flt_num1 = 32 / 10;
13:     int_num2 = 32.0 / 10; /* The divisor is an integer */
14:     flt_num2 = 32.0 / 10;
15:     int_num3 = 32 / 10.0; /* The dividend is an integer */
16:     flt_num3 = 32 / 10.0;
17:
18:     printf("The integer divis. of 32/10 is: %d\n", int_num1);
19:     printf("The floating-point divis. of 32/10 is: %f\n", flt_num1);
20:     printf("The integer divis. of 32.0/10 is: %d\n", int_num2);
21:     printf("The floating-point divis. of 32.0/10 is: %f\n", flt_num2);
22:     printf("The integer divis. of 32/10.0 is: %d\n", int_num3);
23:     printf("The floating-point divis. of 32/10.0 is: %f\n", flt_num3);
24:     return 0;
25: }
```

```
The integer divis. of 32/10 is: 3
The floating-point divis. of 32/10 is: 3.000000
The integer divis. of 32.0/10 is: 3
The floating-point divis. of 32.0/10 is: 3.200000
The integer divis. of 32/10.0 is: 3
The floating-point divis. of 32/10.0 is: 3.200000
```

Data Types: double

A **floating-point** number can also be represented by another data type, called the **double** data type.

A **double floating-point** number is represented by taking 64 bits (1 bit for sign, 11 bits for **exponent**, 52 bits for **mantissa**).

$$1.7976931348623157e+308 > \text{double } number > 2.2250738585072014e-308$$

A **double floating-point** number in C is at least ten digits of **precision**.

Data Types: Scientific Notation

The C language uses **scientific notation** to help you write lengthy **floating-point** numbers.

In **scientific notation**, a number can be represented by the combination of the **mantissa** and the **exponent**.

The format of the notation is that there is an **e** or **E** between the **mantissa** number and the **exponent** number.

Notation: mantissa**e**exponent and mantissa**E**exponent

5000  5e3 in scientific notation.

-300  -3e2 in scientific notation.

0.0025  2.5e-3 in scientific notation.

The format specifier, **%e** or **%E**, is used to format a floating-point number in scientific notation.

Handling Standard Input and Output

The C language provides many functions to manipulate file reading and writing (**I/O**).

The header file **stdio.h** contains the **declarations** for those functions.

You should always include the header file **stdio.h** in your C program before doing anything with **file I/O**.

- ❖ The C language treats a file as a series of bytes.
- ❖ A series of bytes is also called a **stream**.
- ❖ The C language treats all **file streams** equally.

Handling Standard Input and Output

There are three file streams that are pre-opened for you:

- **stdin:** The standard input for reading.
- **stdout:** The standard output for writing.
- **stderr:** The standard error for writing error messages.

The standard input (**stdin**) file stream links to your keyboard, while the standard output (**stdout**) and the standard error (**stderr**) file streams point to your terminal screen.

For instance:

When you called the **printf()** function, you were actually sending the output to the default file stream, **stdout**, which points to your screen.



Standard Input and Output: `getc()` Function

The `getc()` function reads the next **character** from a **file stream**, and returns the **character** as an **integer**.

Syntax Entry

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

```
int getc(FILE *stream);      (stdin is used for FILE *stream )
```

FILE *stream declares a **file stream** (a variable). The function returns the numeric value of the **character** read.

If an **end-of-file** or **error** occurs, the function returns **EOF**.

EOF is a **constant**. **EOF** stands for end-of-file. Usually, the value of **EOF** is **-1**.

Standard Input and Output: getc() Function

Reading in a Character Entered by the User

```
1:  /* 05L01.c: Reading input by calling getc() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      int ch;
7:
8:      printf("Please type in one character:\n");
9:      ch = getc( stdin );
10:     printf("The character you just entered is: %c\n", ch);
11:     return 0;
12: }
```

Please type in one character:

H

The character you just entered is: H

Standard Input and Output: getchar() Function

The **getchar()** function is equivalent to **getc(stdin)**.

Syntax Entry

```
#include <stdio.h>  
int getchar(void);
```

void indicates that no argument is needed for calling the function.

The function returns the numeric value of the **character** read.

If an **end-of-file** or **error** occurs, the function returns **EOF**.

Standard Input and Output: getchar() Function

Reading in a Character by Calling getchar()

```
1:  /* 05L02.c: Reading input by calling getchar() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      int ch1, ch2;
7:
8:      printf("Please type in two characters together:\n");
9:      ch1 = getc( stdin );
10:     ch2 = getchar( );
11:     printf("The first character you just entered is: %c\n", ch1);
12:     printf("The second character you just entered is: %c\n", ch2);
13:     return 0;
14: }
```

Please type in two character together:

Hi

The first character you just entered is: H

The second character you just entered is: i

Standard Input and Output: `putc()` Function

The **`putc()`** function writes a character to the specified file stream which is the standard output pointing to your screen.

Syntax Entry

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

```
int putc(int c, FILE *stream); (stdout is used for FILE *stream)
```

The first argument, **int c**, indicates that the output is a character saved in an **integer variable c**.

The second argument, **FILE *stream**, specifies a file stream.

If successful, **`putc()`** returns the character written; otherwise, it returns **EOF**.

Standard Input and Output: `putc()` Function

Putting a Character on the Screen

```
1:  /* 05L03.c: Outputting a character with putc() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      int ch;
7:
8:      ch = 65;    /* the numeric value of A */
9:      printf("The character that has numeric value of 65 is:\n");
10:     putc(ch, stdout);
11:     return 0;
12: }
```

```
The character that has numeric value of 65 is:
A
```

Standard Input and Output: putchar() Function

The **putchar()** function writes a character without any file stream.

Syntax Entry

```
#include <stdio.h>  
int putchar(int c);
```

int c is the argument that contains the numeric value of a character

If successful, **putchar()** returns the character written; otherwise, it returns **EOF**.

Standard Input and Output: putc() Function

Outputting Characters with putchar().

```
1:  /* 05L04.c: Outputting characters with putchar() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      putchar(65);
7:      putchar(10);
8:      putchar(66);
9:      putchar(10);
10:     putchar(67);
11:     putchar(10);
12:     return 0;
13: }
```

A

newline character (\n).

B

C

A
B
C

Standard Input and Output: printf() Function

Syntax Entry

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

```
int printf(const char *format-string, . . .);
```

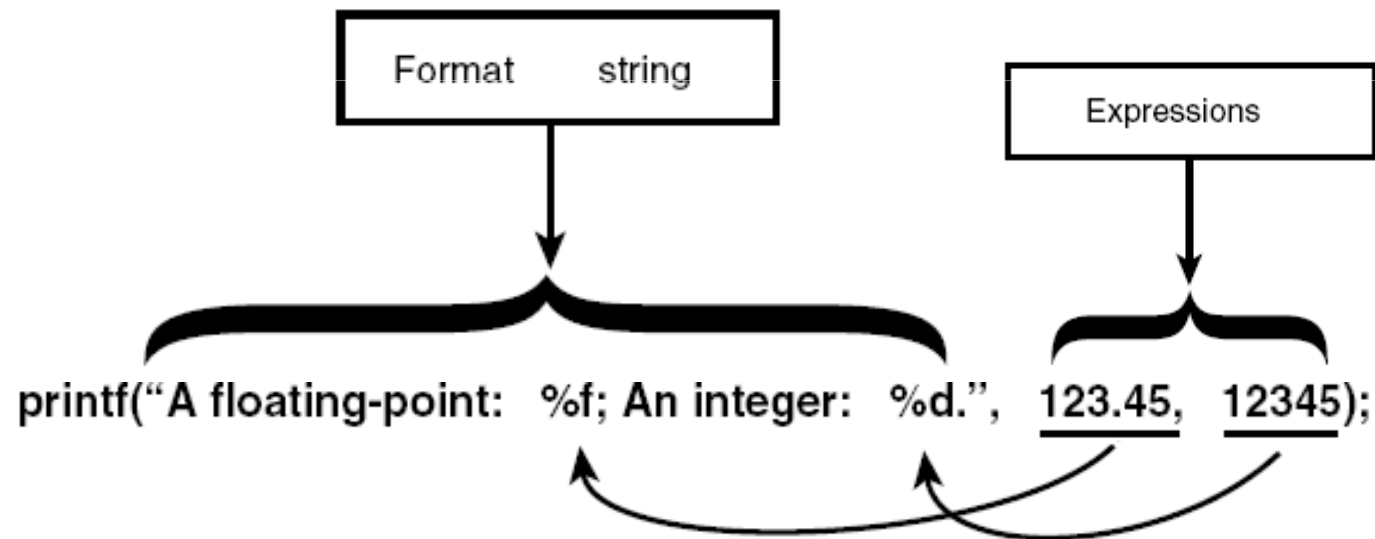
const char *format-string is the first argument that contains the format specifier(s).

... indicates the expression section that contains the expression(s) to be formatted according to the format specifiers.

The function returns the number of expressions formatted if it succeeds. It returns a negative value if an error occurs.

Standard Input and Output: printf() Function

The first argument to the **printf()** function is a **string** (a series of characters surrounded with double **quotes**) with some format specifiers inside.



You should use exactly the same number of expressions as the number of format specifiers within the format string.

Standard Input and Output: Format Specifiers

<code>%c</code>	The character format specifier.
<code>%d</code>	The integer format specifier.
<code>%i</code>	The integer format specifier (same as <code>%d</code>).
<code>%f</code>	The floating-point format specifier.
<code>%e</code>	The scientific notation format specifier (note the lowercase <code>e</code>).
<code>%E</code>	The scientific notation format specifier (note the uppercase <code>E</code>).
<code>%g</code>	Uses <code>%f</code> or <code>%e</code> , whichever result is shorter.
<code>%G</code>	Uses <code>%f</code> or <code>%E</code> , whichever result is shorter.
<code>%o</code>	The unsigned octal format specifier.
<code>%s</code>	The string format specifier.
<code>%u</code>	The unsigned integer format specifier.
<code>%x</code>	The unsigned hexadecimal format specifier (note the lowercase <code>x</code>).
<code>%X</code>	The unsigned hexadecimal format specifier (note the uppercase <code>X</code>).
<code>%p</code>	Displays the corresponding argument that is a pointer.
<code>%n</code>	Records the number of characters written so far.
<code>%%</code>	Outputs a percent sign (%).

Standard Input and Output: Specifying the Minimum Field Width

When %10f is used as a format specifier, 10 is a **minimum field width** specifier that ensures that the output is at least 10 character spaces wide.

```
1:  /* 05L06.c: Specifying minimum field width */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      int num1, num2;
7:
8:      num1 = 12;
9:      num2 = 12345;
10:     printf("%d\n", num1);
11:     printf("%d\n", num2);
12:     printf("%5d\n", num1);
13:     printf("%05d\n", num1);
14:     printf("%2d\n", num2);
15:     return 0;
16: }
```

```
12
12345
    12
00012
12345
```

Standard Input and Output:

Aligning Output

By default, all output is placed on the right edge of the field.

It is necessary to prefix the minimum field specifier with the minus sign (-) to justify the output from the left edge of the field.

```
1:  /* 05L07.c: Aligning output */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      int num1, num2, num3, num4, num5;
7:
8:      num1 = 1;
9:      num2 = 12;
10:     num3 = 123;
11:     num4 = 1234;
12:     num5 = 12345;
13:     printf("%8d %-8d\n", num1, num1);
14:     printf("%8d %-8d\n", num2, num2);
15:     printf("%8d %-8d\n", num3, num3);
16:     printf("%8d %-8d\n", num4, num4);
17:     printf("%8d %-8d\n", num5, num5);
18:     return 0;
19: }
```

1	1
12	12
123	123
1234	1234
12345	12345

Standard Input and Output: Using the Precision Specifier

The **precision** specifier is used to determine the number of decimal places for **floating-point** numbers or to specify the maximum field width (or length) for integers or strings.

For instance: %10.3f, means minimum field width is specified as 10 characters long, and the number of decimal places is set to 3.

```
1:  /* 05L08.c: Using precision specifiers */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:      int int_num;
7:      double flt_num;
8:
9:      int_num = 123;
10:     flt_num = 123.456789;
11:     printf("Default integer format:    %d\n", int_num);
12:     printf("With precision specifier:  %2.8d\n", int_num);
13:     printf("Default float format:      %f\n", flt_num);
14:     printf("With precision specifier:  %-10.2f\n", flt_num);
15:     return 0;
16: }
```

Default integer format:	123
With precision specifier:	00000123
Default float format:	123.456789
With precision specifier:	123.46

Original number is rounded to 2 decimal places.