Computer Programming

Variable

Hung-Yun Hsieh

September 13, 2022

Computer Programming

Literal

Literal *of, relating to, or expressed in letters*

- A literal is a notation for representing a given (fixed) value in the source code
  - Integers, floating-point numbers (real numbers), characters, and strings
- Literals can be used to initialize (*give initial value to*) variables or constants
  - Literals are invariants whose values are implied by their representations
  - ☞ Variables are identifiers that can take on any of a class of fixed values (e.g. a *character variable* can take any *character literal as its value*)
  - ☞ Constants are variables whose value cannot be changed

① Numerical Literal

- Integer number
  - 100 → specify a number in base 10
  - 0100 → in base 8 = 64
  - 0x100 (or 0X100) → in base 16 = 256

```
cout << "The number is: " << 100;
```

The E notation can also be used to specify an integer number

Multiple messages can be cascaded and sent to `cout` in one statement

- Floating-point number
  - 123.0 → specify a floating-point number
  - 1.23e2 (or 1.23E2, 1.23e+2, 1.23e+02, ...) → 123.0
  - 8.33e-4 (or 8.33E-4) → 0.000833

```
cout << "The number is : " << 1.23e2;
```

No space before and after 'e'

8.3e0.5 is not valid!

Recall Base-8 and Base-16 Notations

- Octal (base-8) and hexadecimal (base-16) numbers
  - ☞ 8 < 10 < 16

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binary	Hex	Decimal
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

octal (base-8)

1 nibble = 4 bits

lower case is okay

Output of Different Bases

We will learn more on details of the manipulator later in this course

- Output manipulator
  - Manipulate how output is formatted

    cout << manipulator

    - cout manipulators are special identifiers provided along with cout that make it possible to control the output stream
    - Examples: show an integer value in base-8 (oct), base-16 (hex), or base-10 (dec)

Try showbase and uppercase

```
#include <iostream>
using namespace std;

int main( )
{
    cout << "hex=" << hex << 100 << "\n";
    cout << "oct=" << oct << 100 << "\n";
    cout << "dec=" << dec << 100 << "\n";
}
```

\n is a special character that causes the cursor to move to the beginning of next line on the screen

These manipulators are 'sticky' meaning that the output (cout) is changed hereafter in later statements until specified otherwise

Output of Floating Numbers

- Manipulation of the floating-point notation
  - `scientific`: one digit before the decimal point followed by the e notation
  - `fixed`: position of the decimal point is fixed

```
cout.unsetf(ios::fixed|ios::scientific);
```

The default floating-point notation is set to none (neither `fixed` nor `scientific`)

```
#include <iostream>
using namespace std;
#define VAL 31.4159
int main( )
{
    cout <<                31.4159 << "\n" << 3.0 << "\n";
    cout << fixed <<       31.4159 << "\n" << 3.0 << "\n";
    cout << scientific << 31.4159 << "\n" << 3.0 << "\n";
}
```

VAL

```
31.4159
3
```

```
31.415900
3.000000
```

```
3.141590e+01
3.000000e+00
```

② Character Literal

A *multi-character* literal has integer and *implementation-defined value* (i.e. **unexpected value across compilers**)

- Specification of a character
  - Enclosed in single quotes ' and '
  - 'g' → the character g (the quotes are *must*)
  - g → an identifier that "could" be the name of a variable

The correct way is to have *only ONE character* inside a pair of single quote

```
cout << "The character is: " << 'g';
```

```
cout << 'no';
```

```
28271
```

```
cout << "Two characters are: " << 'n' << 'o';
```

- How to specify the single quote character?
- How about special characters that cannot be typed directly from the keyboard?

☞ Use of the *escape sequence*

ASCII Table

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

33 control characters

Escape Sequence

- Escape sequence (code)
  - A character preceded by \ (backslash) for special purpose
  - ☞ Escape code can specify a character in the ASCII code
    - '\x41' or '\101' → the character with ASCII code=65 (0x41, 0101)

hex   oct

  - ☞ Some commonly used escape code

Escape Code	Description
`\n`	Newline. Move the cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the cursor to the next tab stop.
`\a`	Bell. Generate an audible sound (platform-dependent).
`\b`	Backspace. Move the cursor backward for one character.
`\\`	Backslash. Used to print a backslash character.

  - ☞ Single quote needs to be escaped when specified as a character literal (i.e. use `'\''` to specify the character)

More on New Line

Difference between `\n` and `endl`?

```
#include <iostream>
using namespace std;

int main( )
{
    cout << "This is";
    cout << "C++!" << '\n';

    cout << "A new line." << endl;
    return 0;
}
```

`\n` is a character that causes the cursor to move to the beginning of next line on the output screen

`endl` is a manipulator declared in `std` that, when sent to **`cout`**, causes a new line to be created

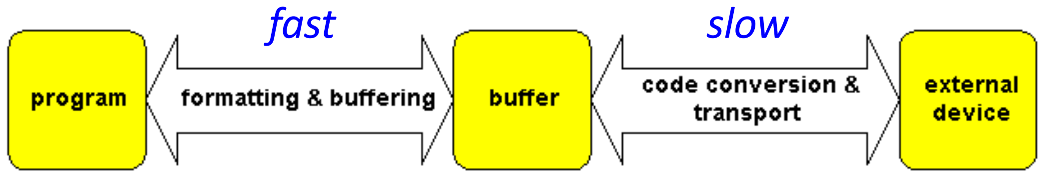
`std::endl`

```
This isC++!
A new line.
```

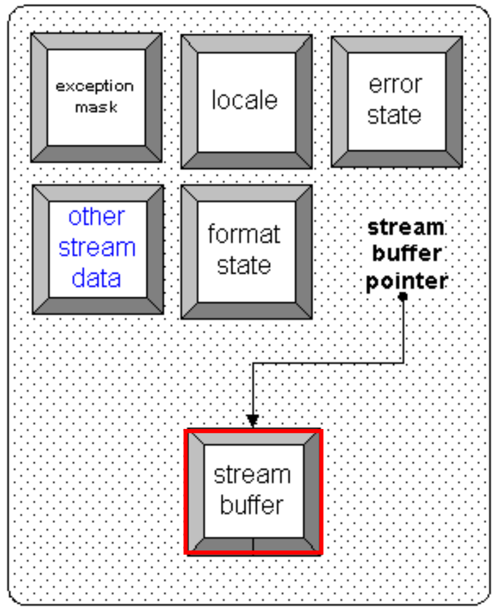
`endl` *inserts* a new line character and then ***flushes* the output stream** for display on the screen

T	h	i	s		i	s	C	+	+	!	\n	A		n	e	w		l	i	n	e	.	\n	

I/O Buffer

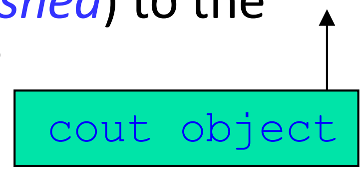


- I/O buffering
  - I/O operations often have high latencies (the time between the initiation of an I/O process and its completion)
  - *I/O buffers* are provided to alleviate the bottleneck
    - Temporarily storing data that is passing between a processor and a peripheral
    - Each output (write) routine simply tacks data onto the buffer, until it is filled, at which point the buffer contents are sent to the peripheral (*full buffering*)



☞ User can request data to be immediately sent (*flushed*) to the peripheral, rather than being cached in the buffer

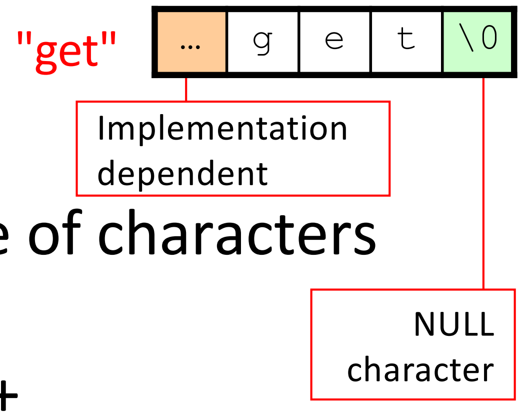
☞ *endl (cout)*



③ String Literal

It is wrong to write
`"This "string" is wrong"`

- Specification of a string
  - Enclosed in double quotes " and "
  - String can be considered as a sequence of characters *ended with a special (NULL) character*
  - ☞ "g" and 'g' mean different things to C++



  - ☞ Double quote needs to be escaped when it is to be specified in a string (i.e. use \" inside double quotes)
  - String can extend to more than a single line by putting a backslash sign (\) at the end of each unfinished line
  - Several string literals (separated by one or several white space characters) are concatenated into one string

Example on String

```
#include <iostream>
using namespace std;
int main( )
{
    cout << "Double and single quotes - ' \" \n";
    cout << "We can connect \
    strings on two lines.\n";

    cout << "We can also connect "
    "strings this" "way!";
}
```

"a\nb" and "a\tb" are both valid strings

There will be a compiler error if no \ is placed here to "escape" the new line

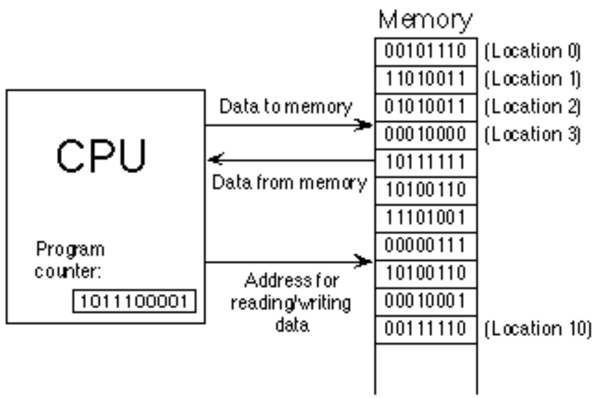
Note that the space is also part of the string

```
Double and single quotes - ' "
We can connect     strings on two lines.
We can also connect strings thisway!
```

Computer Programming

Variable

Concept of a Variable



- Computer processes information
  - Information is stored in memory
  - Need to be able to specify the information stored at a particular location (i.e. memory address)
    - ☞ Store *a* number, and then increase *the* number by 1
  - A variable can be considered as a portion of memory to store a determined value
    - Type of the information stored
    - Size of the information stored
  - Variable name (identifier)
    - Each variable needs an identifier that distinguishes it from others in the same namespace
    - ☞ It is necessary to declare the use of a variable before using it

A variable is a shortcut to a location in memory where value can be stored for use by the program

Declaring Variables

```
signed int tank_id;
unsigned int tank_id;
```

- Declare variables before use

`int tank_id;` — variable name: `tank_id`; variable type: `int`

```
cout << "size of int=";
cout << sizeof(int);
```

It is also okay to use `sizeof(tank_id)`

- Variable type
  - `char`, `int`, `short`, `long`, `float`, `double`, ...
  - ① Integral type: char, short, int, long, long long, ...
  - ② Floating-point type: float, double, long double, ...
  - ☞ Additional `signed` and `unsigned` for integral type

`short` = `short int`

`long` = `long int`

- Knowing the size of a variable (memory occupied)
  - ☞ Use "`sizeof(type)`" or "`sizeof(name)`"

Naming Variables

Wrong variable names:
`2people`, `rate%`, `id#`, `operator`, `my name`

- Variable name
  - Only alphabets (a-z, A-Z), digits (0-9), and underscores (_) can be used
  - Cannot start with a digit (0-9)
  - Case sensitive
  - Mixing cases or underscores
    - MaximumLength, maximum_length, …
  - ☞ Do not use C++ keywords as variable names
    - int, double, long, if, while, new, true, class, …

Hungarian notation:
nSize, fMoney, chLetter, lDistance…

☞ Okay to declare *multiple* variables in one statement

```
int tank_id, staff_num, count;
```

Not Just for Variables

We will encounter most of these elements later in this course

- An identifier is a sequence of characters *given by the programmer* to denote one of the following
  - Object or variable name
  - Class, structure, union, or enumeration name
  - Member of a class, structure, union, or enumeration
  - Function or class-member function
  - `typedef` name
  - Macro name
  - …

☞ Naming of any identifier follows the same rule as mentioned before

Assigning Value

Implicit *type conversion* is performed by the compiler during value assignment if type is mismatched

- Storing value to a variable (at some location)
  - Note "=" reads "assign" not "equal"
  - Value can only appear at the right hand side (RHS)
  - Assign a value of *the right data type* to the variable

```
tank_id = 10;
```

Variable name	Variable type	Memory cell address	Variable value
tank_id	int	FFE0	12
diameter	double	FFFE	111.1
pressure	double	FFF6	100.

☞ Declaration and assignment in one statement

```
int tank_id = 10;
double diameter, pressure = 99.3, weight;
weight = 62.5;
```

Uninitialized

Initialize the variable

Assign a value

☞ Assignment during declaration is called *initialization*

Example

If you do not initialize an variable defined inside a function, the variable value is *undefined*, meaning that it can take on whatever value previously resided at that location in memory

```
#include <iostream>
using namespace std;
int main( )
{
    int tank_id, TankId;
    double fDiameter = 111.1;

    tank_id = 12;

    cout << ": tank ID="  << tank_id
         << ", diameter=" << fDiameter
         << ", TankId="   << TankId;
}
```

Declaring variables

Declaring and initializing variables

Assigning values

Need to initialize a variable before retrieving its value

```
Program #1: tank ID=12, diameter=111.1, TankId=251547702
```

Literals Revisited

Use `sizeof(100LL)` and `sizeof(123.0L)` to check the size

- Integral data type
  - ☞ Integer literal by default is stored as type `int`
    - 100u or 100U → stored as `unsigned int`
    - 100l or 100L → stored as `long int`
    - 100ll or 100LL → stored as `long long`
    - 100ul or 100UL → stored as `unsigned long`
    - 100ull or 100ULL → stored as `unsigned long long`
- Floating point data type
  - ☞ Floating point literal by default is stored as `double`
    - 123.0f or 123.0F → stored as `float`
    - 123.0l or 123.0L → stored as `long double`

Example

```
#include <iostream>
using namespace std;
int main( )
{
     cout << "size of 100    = " <<sizeof(100)<< '\n';
     cout << "size of 100l   = " <<sizeof(100l)<< '\n';
     cout << "size of 100ll  = " <<sizeof(100ll)<< '\n';
     cout << "size of 100.0  = " <<sizeof(100.0)<< '\n';
     cout << "size of 100.0f = " <<sizeof(100.0f)<< '\n';
     cout << "size of 100.0l = " <<sizeof(100.0l)<< '\n';
}
```

```
size of 100    = 4
size of 100l   = 4
size of 100ll  = 8
size of 100.0  = 8
size of 100.0f = 4
size of 100.0l = 12
```

① Integral Data Type

- Integral data (no fraction)
  - `char`, `int`, `short`, `long`, `long long`, …
  - `signed` integer & `unsigned` integer
- Unsigned integer
  - Positive integer and zero

  $100111011_2 = 473_8 = 13B_{16} = 315_{10}$
- Signed integer
  - Positive integer, negative integer, and zero

☞ Two's complement representation

① Invert the bit sequence of a positive integer

② Add 1 to the sequence to get the negative integer

One's and Two's Complement

8 bit ones' complement

Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	+0	0
00000001	1	1
...	...	...
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−127	128
10000001	−126	129
10000010	−125	130
...	...	...
11111110	−1	254
11111111	−0	255

- ❶ Invert bit by bit (get complement)

8 bit two's complement

Binary value	Two's complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
...	...	...
01111110	126	126
01111111	127	127
10000000	−128	128
10000001	−127	129
10000010	−126	130
...	...	...
11111110	−2	254
11111111	−1	255

worth $-1*2^7$

- ❶ Invert bit by bit
- ❷ Add 1 to the result

Two's complement is more popularly used

Using two's complement, 1 byte can represent from -128 to 127

1-2 = -1
= 1 + (-2)

Integer Range

It is important to tell CPU what the bit sequence represents (e.g. signed or unsigned)

- Maximum value of an integer
  - Only 2^n values can be represented for n bits
  - ☞ Use "`sizeof(`type`)`" to know the size of type
  - char: 1 byte, short: 2 bytes, long: 4 bytes, int: 4 bytes, long long: 8 bytes (for 32-bit systems)

```
unsigned char:      0 ~ 255
signed char:      -128 ~ 127
unsigned short:     0 ~ 65535
signed short:     -32768 ~ 32767
```

Check `<climits>` for the max & min values of each data type

- ☞ A note on character and integer
  - A `char` is *internally stored* as an (1-byte) integer
  - The difference is only when it is "displayed" (`c-out`)
  - It is okay to use `char` for numerical calculation

Data Type Models

C header with a name of the form *name.h*: name in the global namespace
C++ header with a name of the form *cname*: name in the standard library (**std**) namespace

- Different data models for 64-bit programs
  - LP64: long and pointer are 64 bits
    - 4/8/8 — sizes of int/long/pointer
    - ☞ Mac OS X, Linux
  - LLP64: long long and pointer are 64 bits
    - 4/4/8
    - ☞ MS Windows
  - ILP64: integer, long, and pointer are 64 bits
    - 8/8/8
    - ☞ Solaris SPARC64

```
int16_t a =10;
int64_t b;
uint8_t c ='x';
```

32-bit model

Data Type	LP32	ILP32	ILP64	LLP64	LP64
char	8	8	8	8	8
short	16	16	16	16	16
int32			32		
int	16	32	64	32	32
long	32	32	64	32	64
long long (int64)				64	
pointer	32	32	64	64	64

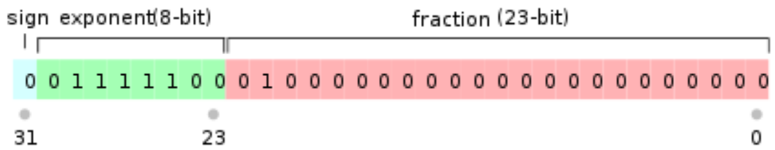
☞ It is a good habit to use
- `intXX_t` & `uintXX_t`
- ☞ XX: `8`, `16`, `32`, `64`

`#include <cstdint>` if there is an error

② Floating Point Data Type

Note that it is possible that *a simple real number can not be accurately represented* (e.g. 0.8)

- An example (for 32 bits)



10111111010000000000000000000000

2^7 2^0 2^{-1} 2^{-23}

sign (1) *exponent (8)* *fraction or significand (23)*

How to express -1313.3125?

Any nonzero number can be normalized as 1.xxx (radix=2)

value = -1.5 x $2^{(126-127)}$ = -1.5 x 2^{-1} = -0.75

Invariant numbers

To allow for positive and negative exponents (-127 ~ 128)

sign is 1 = the number is negative

exponent is 01111110 = 126 (treated as an unsigned integer)

fraction is 100000000000… = 2^{-1} = 0.5 (decimal)

- ☞ Number of bits in exponent affects range
- ☞ Number of bits in fraction affects precision

Floating Point Number Conversion

- From -1313.3125 to the IEEE 32-bit format
  - ① Integer part (treated as an unsigned integer)

    $1313 = 10100100001_2$

    2^0

  - ② Fractional part

    2^{-1}

0.3125	× 2 =	0.625	0	Generate 0 and continue
0.625	× 2 =	1.25	1	Generate 1 and continue with the rest
0.25	× 2 =	0.5	0	Generate 0 and continue
0.5	× 2 =	1.0	1	Generate 1 and nothing remains

  - ③ Normalize (to 1.xxx)

    $10100100001.0101_2 = 1.01001000010101_2 \times 2^{10}$

    $10+127 = 137 = 10001001_2$

  - ④ Sign, exponent, and mantissa

    $-1313.3125 = 11000100101001000010101000000000_2 = C4A42A00_{16}$

More on the Floating Point Number

- Some special cases

Precision limits on integer values (float)

- Integers in $[-16777216, 16777216]$ can be exactly represented
- Integers in $[-33554432, -16777217]$ or in $[16777217, 33554432]$ round to a multiple of 2
- Integers in $[-2^{26}, -2^{25}-1]$ or in $[2^{25}+1, 2^{26}]$ round to a multiple of 4
- ....

Type	Sign	Exponent field	Significand (fraction field)	Value
Zero	0	0000 0000	000 0000 0000 0000 0000 0000	0.0
Negative zero	1	0000 0000	000 0000 0000 0000 0000 0000	−0.0
One	0	0111 1111	000 0000 0000 0000 0000 0000	1.0
Minus One	1	0111 1111	000 0000 0000 0000 0000 0000	−1.0
Smallest denormalized number	*	0000 0000	000 0000 0000 0000 0000 0001	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	0000 0000	100 0000 0000 0000 0000 0000	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	0000 0000	111 1111 1111 1111 1111 1111	$\pm(1-2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	0000 0001	000 0000 0000 0000 0000 0000	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Largest normalized number	*	1111 1110	111 1111 1111 1111 1111 1111	$\pm(2-2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	1111 1111	000 0000 0000 0000 0000 0000	$+\infty$
Negative infinity	1	1111 1111	000 0000 0000 0000 0000 0000	$-\infty$
Not a number	*	1111 1111	non zero	NaN
* Sign bit can be either 0 or 1 .				

☞ IEEE single-precision (32 bits), double-precision (64 bits), and quad-precision (128 bits) formats

Size of Floating Point Number

Check `<cfloat>` for more information

- Size of floating point number
  - float: 4 bytes, double: 8 bytes, long double: 8~16 bytes
    - `float`: 23 bits for fraction, 8 bits for exponent
    - `double`: 52 bits for fraction, 11 bits for exponent

Range (exponent)
float: 2^{128} ~ 10^{+38}
double: 2^{1024} ~ 10^{+308}

Precision (fraction)
float: 2^{-23} ~ 10^{-7}
double: 2^{-52} ~ $10^{-15.x}$

☞ A note on floating-point number

- A floating-point number may not be precisely represented (and stored) *even for simple values*

```
int n = 4.35 * 100; cout << n;
```

☞ Use *integral type* for fraction if precision really matters

Storing vs. Showing Values

- Integer data type (for storage)

ASCII code of 'g' = **67** (hex)

```
short  x = 'g';
```

x	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	1

```
char   y = 'g';
```

The integer value corresponding to the ASCII code of 'g' is stored in `x` – same as the case for `y`

y	0	1	1	0	0	1	1	1

It is `cout` that makes the output *look different*

- Output formatting (for display)

```
cout << x;
```

Numerical value of the bit sequence stored in `x` is shown

```
cout << y;
```

Character corresponding to the numerical value in `y` is shown

Example

```
#include <iostream>
using namespace std;

int main( )
{
    char c1, c2, c3, c4, c5, c6, c7;
    int  i;

    c1 = 'g';  c2 = '7'; c3 = '<'; c4 = '\n';
    c5 = 63;   c6 = '\101'; c7 = '\x61';
    i  = c1;

    cout << "c1=" <<c1 <<" c2=" <<c2 <<" c3=" <<c3 <<" c4=" <<c4
         <<" c5=" <<c5 <<" c6=" <<c6 <<" c7=" <<c7 <<" i=" <<i <<endl;
}
```

`cout` can display the value of the variable intelligently depending on its data type

In C++, 'g' and "g" are different. You cannot assign a string literal to a character variable

Wrong usage:
`c1 = "g";`

The numeric value of `c1` (ASCII code) is **stored as an integer** at `i`

```
c1=g c2=7 c3=< c4=
 c5=? c6=A c7=a i=103
```

Implicit *type conversion* (if necessary) is performed by the compiler during value assignment

Another Example

```
#include <iostream>
using namespace std;

int main( )
{
    int id = 12, itype;
    char type = 'c';
    double diameter = 13411.11;
    cout << "Tank ID=" << id << ", diameter=" << diameter << "\n";
    cout << "Tank type=" << type << "\n";

    itype = type;
    cout << "Tank type in ASCII code=" << itype
         << " (HEX=" << hex << itype << ")\n";
}
```

```
Tank ID=12, diameter=13411.1
Tank type=c
Tank type in ASCII code=99 (HEX=63)
```

We will see the advantages (e.g. extensibility) of using `cout` for output formatting later

`cout` "intelligently" shows the character for a `char` variable

Output Manipulator Revisited

- Parameterized manipulator

  `cout << manipulator(argument)`

  - Include header `<iomanip>`

- `setprecision()`
  - Set the precision for displaying real numbers
  - *Max number* of significant digits (not including leading 0's) in a number for the default floating-point notation
  - Use the `fixed` notation for setting the exact number "after the decimal point" (even they are trailing zeros)

Use `cout.unsetf(ios::fixed);` to reset to the default (none) notation

The `fixed` manipulator specifies non-scientific float-field notation (*cf*. `scientific`) for formatting a floating-point number

```
cout << setprecision(20) << 10.45  << endl;
cout << fixed << setprecision(20) << 10.45  << endl;
cout << fixed << setprecision(20) << 10.45f << endl;
cout << fixed << setprecision(20) << 10.45l << endl;
```

User-Input Values

```
#include <iostream>
using namespace std;

int main( )
{
    double income, expense;
    int month;

    cout << "What month is it?" << endl;
    cin  >> month;
    cout << "You have entered month=" << month << endl;

    cout << "Enter your income and expenses" << endl;
    cin  >> income >> expense;
    cout << "Entered income=$" << income << ", expenses=$" << expense;
}
```

Errors can be generated due to improper input from the user (more on this later)

Reads one integer value from the keyboard

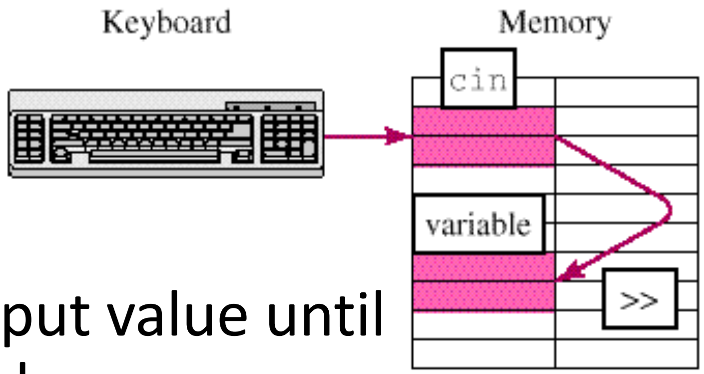
Reads two floating-point values from the keyboard

`cin` can read `0.8` as well as `.8` as valid input for a floating-point number

The `cin` Object (`std::cin`)

- Standard input
  - `cin` is linked to the keyboard

    ```
    cin >> variable;
    ```



  - The program waits for user to input value until the "Enter key" has been pressed
  - White space characters are skipped (not read)
  - Reading multiple values using one statement

    ```
    cin >> variable_1 >> variable_2;
    ```

  ☞ Cascading is similar to `cout`

  ☞ Show prompts before `cin` statements so the user knows what is going on

Reading Characters

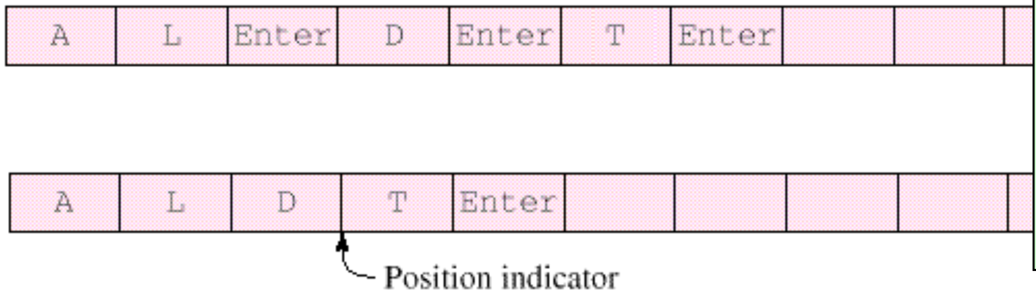
- Character input
  - Similar to number input

We will discuss how to read a word or an entire line from the user later

```
char c1, c2, c3, c4;
cout << "Enter your first, middle, and last initials: " << endl;
cin >> c1 >> c2 >> c3;
cout << "You entered: " << c1 << c2 << c3 << endl;

cout << "Enter one more character: " << endl;
cin >> c4;
cout << "You entered: " << c4 << endl;
```

  - Input buffer



If the input buffer *is not empty* when you call `cin`, then `cin` uses the data already in the buffer instead of waiting for more from the user

Constants (*Constant* Variables)

- Constant
  - It is easier to refer to an invariant value through a name (identifier) rather than directly through its value
  - Variable

```
double pi = 3.14159265;
```

The value of a variable may inadvertently be changed by other statements

  - Constant variable

```
const double pi = 3.14159265;
```

Wrong usage:
```
const double pi;
pi=3.14159265;
```

    - A constant variable *must be explicitly initialized* and its value *cannot be changed once it is initialized*
  - Preprocessor directive

```
#define pi 3.14159265
```

    - The preprocessor performs replacement *before compiling*

Constant Variables

After processing of the preprocessor

```cpp
#include <iostream>
using namespace std;
#define PI 3.1415926
#define LF '\n'
int main( )
{
    double Pi = PI;
    const double pi = PI;

    cout<<"Pi= "<<Pi<<LF;
    cout<<"pi= "<<pi<<LF;
    cout<<"PI= "<<PI<<LF;

    Pi = 3.14;
    cout<<"Pi= "<<Pi<<LF;
}
```

```cpp
// content of the file: iostream
using namespace std;


int main( )
{
    double Pi = 3.1415926;
    const double pi = 3.1415926;

    cout<<"Pi = "<<Pi<<'\n';
    cout<<"pi = "<<pi<<'\n';
    cout<<"PI = "<<3.1415926<<'\n';

    Pi = 3.14;
    cout<<"Pi = "<<Pi<<'\n';
}
```

Variable Reference

We will explain in details the use of variable reference later

- Alias of a variable

```
int count = 1;
int & cref = count;
```

`cref` can be considered as an "**alias**" of the variable `count`

- The variable `cref` is a "reference" to another integer variable `count`
- In essence, `cref` and `count` indicate the **same location** in memory (no additional space is allocated for `cref`)

☞ Reference variable *must be explicitly initialized* (to another variable that it references) in the declaration

```
cref = 2;
cout << "count=" << count << endl;
```

```
count=2
```

Wrong usage:
`int & cref;`
`cref = count;`

Variable in a Namespace

Ambiguous call of variables due to the use of "`using namespace` …" will be stopped by the compiler

```
#include <iostream>
using namespace std;

namespace first
{
  int var = 5;
}

namespace first { int num = 7; }

namespace second { double var = 3.1416; }

int main( )
{
    cout << first::var << first::num << endl;
    cout << second::var;
}
```

```
57
3.1416
```