

Chapter 3
Basic Data Types



Outline

- Data types
- Operators and expressions



Data Types



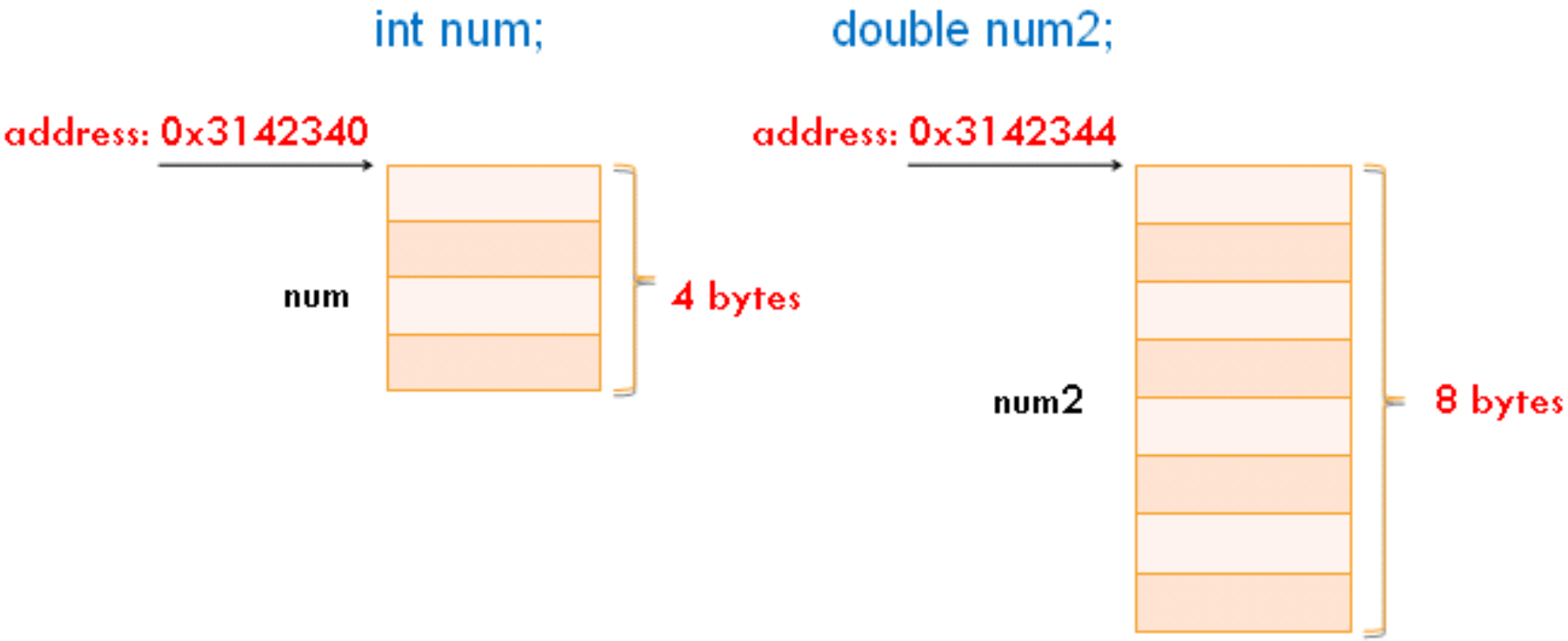
Variable and Constant

- Variable
  - Correspond to locations of the computer's memory
  - Has a name and type, which are fixed after declaration
  - Has a value, which can be updated
- Constant
  - Does not change during the execution of the program



Memory Concepts

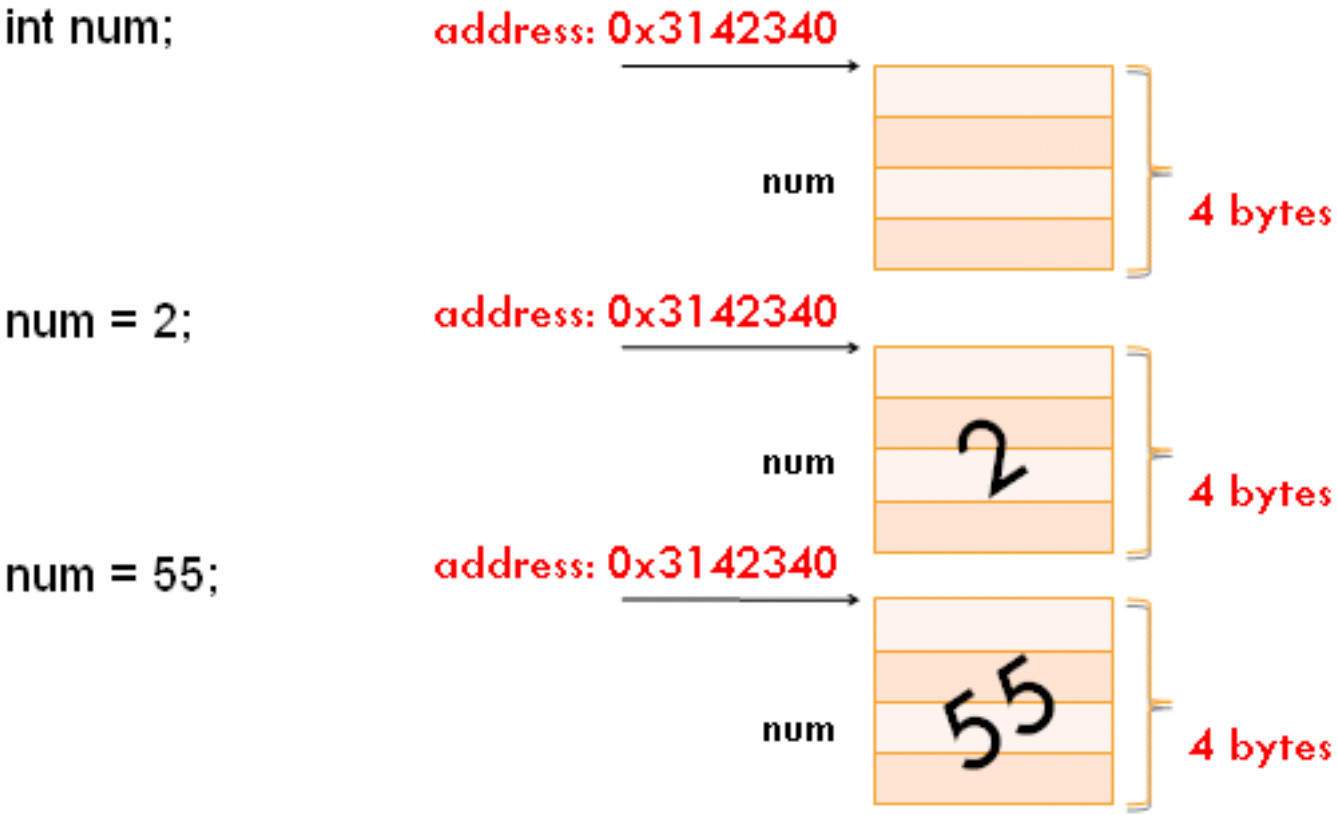
- Declare a variable
  - Allocate memory to a variable





Memory Concepts (Cont.)

- Assign a value to a variable





Data Types

Data Type	Description	# of bytes	Range
int	integer	4	-2147483648 ~ 2147483647
long int	long integer	4	-2147483648 ~ 2147483647
short int	short in	2	-32768 ~ 32767
char	character	1	0 ~ 255
float	floating point	4	1.2e-38 ~ 3.4e38
double	double floating point	8	2.2e-308 ~ 1.8e308

Note: The size of each data type may vary from different compilers



Overflow

- Definition:
  - The value assigned to a variable is larger than or smaller than the range of the corresponding data type.

- Overflow example:
  - short int num = 40000; /* short: -32768~32767 */
  - unsigned short int num = 700000; /* unsigned short: 0~65535 */
  - short int num_a, num_b, sum;
    num_a = 30000;
    num_b = 30000;
    sum = num_a + num_b; /* num_a + num_b = 60000 */

Overflow: Example

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 int main(void)
04 {
05   short int sum, num;          /* declare “num” and “sum” */
06   num = 0x7FFFL;               /* set “num” to 32767 */
07   sum = num + 1;
08   printf(“num + 1 = %d\n“, sum);      /* print variable */
09
10   sum = num + 2;
11   printf(“num + 2 = %d\n”, sum);      /* print variable */
12   system("pause")
13   return 0;
14 }
```

Long integer

0x7FFF=32767

Output:

```
num + 1 = -32768
num + 2 = -32767
```



Overflow: 2's Complement

One's complement: if sign bit = 1, invert from 0 to 1 and from 1 to 0
Two's complement: One's complement + 1
The value of an integer variable: $(-1)^{\text{sign-bit}} * (\text{two's complement})_2$

sign bit Max positive value = $(-1)^0 * (111111..11)_2 = 32767$

0	1	1	1	1	…	1	1	1	1

sign bit Min negative value = $(-1)^1 * (10000…00)_2 = -32768$

1	0	0	0	0	…	0	0	0	0

Two's complement (8 bits)	
00000000	00000000 = 0
00000001	00000001 = 1
….	
01111111	01111111=127
11111111	-(00000001) = -1
….	
10000000	-(10000000) = -128

- If sign bit = 1, the variable is negative
- If sign bit = 0, the variable is non-negative



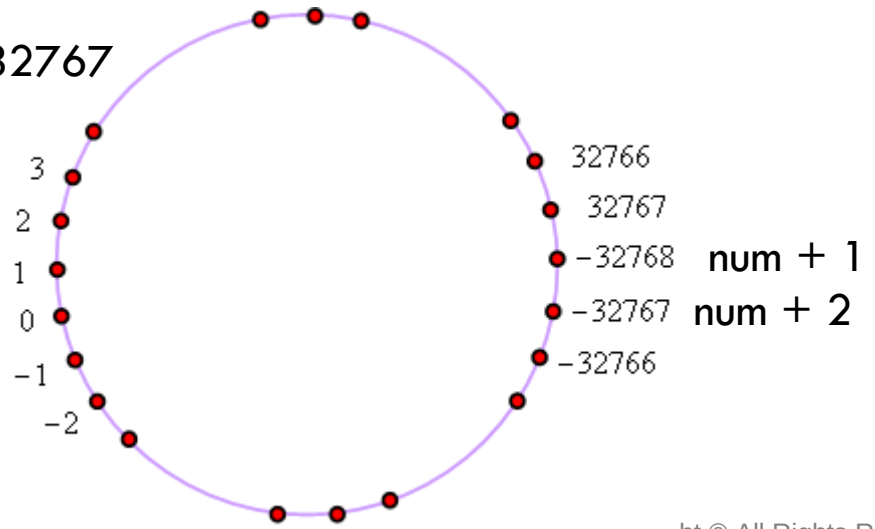
Overflow: Signed Variable

sign bit

Max positive value = $(-1)^0 * (111111..11)_2$

	0	**1**	**1**	**1**	**1**	**...**	**1**	**1**	**1**	**1**
+										1
	1	**0**	**0**	**0**	**0**	**...**	**0**	**0**	**0**	**0**

short int num = 32767





Unsigned Variables

- Let the variable always have a positive value

Data Type	Description	# of bytes	Range
unsigned int	Unsigned integer	4	0 ~ 4294967295
unsigned long int	Unsigned long integer	4	0 ~ 4294967295
unsigned short int	Unsigned short in	2	0 ~ 65535

- Why should we use unsigned variables?
  - Increase the range of a variable if we are very sure that the variable must not be a negative value.
    - Example: The number of students in a class
  - Save the size of a variable (in terms of memory)
    - Example: For a variable ranging from 0 to 40000, we can declare it as int (4 bytes) or unsigned short int (2 bytes)

Data Type: char

- Character, which occupies one byte
  - char ch = ‘1’; /* the character ‘1’ != integer 1 */
    ch = ‘A’;
    ch = ‘a’; /* ‘A’ != ‘a’ */
    ch = 97; /* Assign the character whose ASCII is 10 (‘a’) to ch. */
- There are totally 256 characters
  - Not all characters are printable
  - Check the table mapping each character to its ASCII code (http://en.wikipedia.org/wiki/ASCII)



Print a Character

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 int main(void)
04 {
05   char ch = 'a' , ch2 = 100;          /* declare a character */
06   printf("ch = %c\n", ch); /* print the character ‘a’ */
07   printf("ASCII of ch = %d\n", ch);/* print the ASCII of ‘a’ */
08
09   printf("ASCII 100 = %c\n", ch2);/* print the variable
10                                          whose ASCII is 100 */
12   system("pause");
13   return 0;
14 } /* end of main() */
```

```
ch = a
ASCII of ch = 97
ASCII 100 = d
```



Escapes

Escape sequence	Description	ASCII
\a	Alert	7
\b	Backspace	8
\n	New line	10
\r	Carriage return	13
\0	Null character	0
\t	Tab	9
\\	Backslash (\)	92
\’	Single quote (‘)	39
\”	Double quote (“)	34

Data type: float

- Declare a float variable
  - float f1 = 123.45**F**;
  - float f1 = 1.2345E2;
  - float f1 = 0.00123;
  - float f1 = 1.23E-3**F**;

float

float

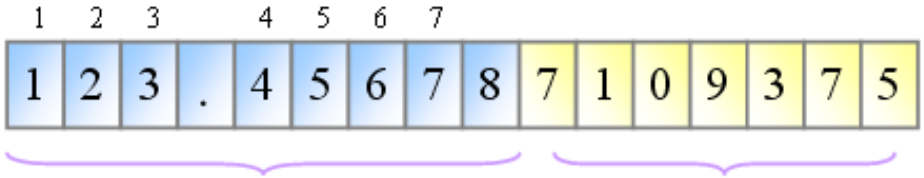
- Print a floating variable
  - **printf(”f1 = %f\n”, f1);**



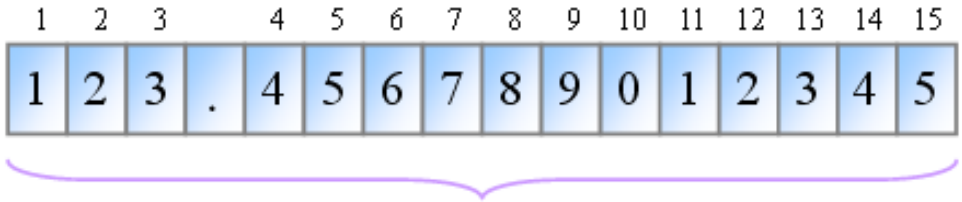
Data type: double

- 2.2E-308 ~ 1.8E308
- Precision of float: 7~8 bits
- Precision of double: 15~16 bits

```
float num1=123.456789012345F;
```




```
double num2=123.456789012345;
```



Must be abandoned due to the limited memory size



Library Function *sizeof()*

- Calculate the size of data types
  - sizeof(int); // 4 bytes
  - sizeof(double); // 8 bytes
- Calculate the size of a variable
  - int num;
    sizeof(num); // 4 bytes
  - sizeof(2L); // 4 bytes
- Unit: ***byte***



Type Conversion

- Convert a variable to another type:

(new type) variable

```
01 int num = 12;
02 float total;
03 total = (float) num;
```

Type Conversion

- Example 1

float

```
01 float f1 = 3.1, f2 = 3.2F;
02 printf (“f1 = %f, f2 = %f\n“, f1, f2);
03 printf (“f1 = %d, f2 = %d\n“, (int)f1, (int)f2);
```

Output:

```
f1 = 3.1, f2 = 3.2
f1 = 3, f2 = 3
```

```
01 int num = 5;
02 printf (“num/2 = %d\n“, num/2);
03 printf (“float: num/2= %f\n“, (float)num/2);
```

Output:

```
num/2 = 2
float: num/2 = 2.500
```



Operators and Expressions

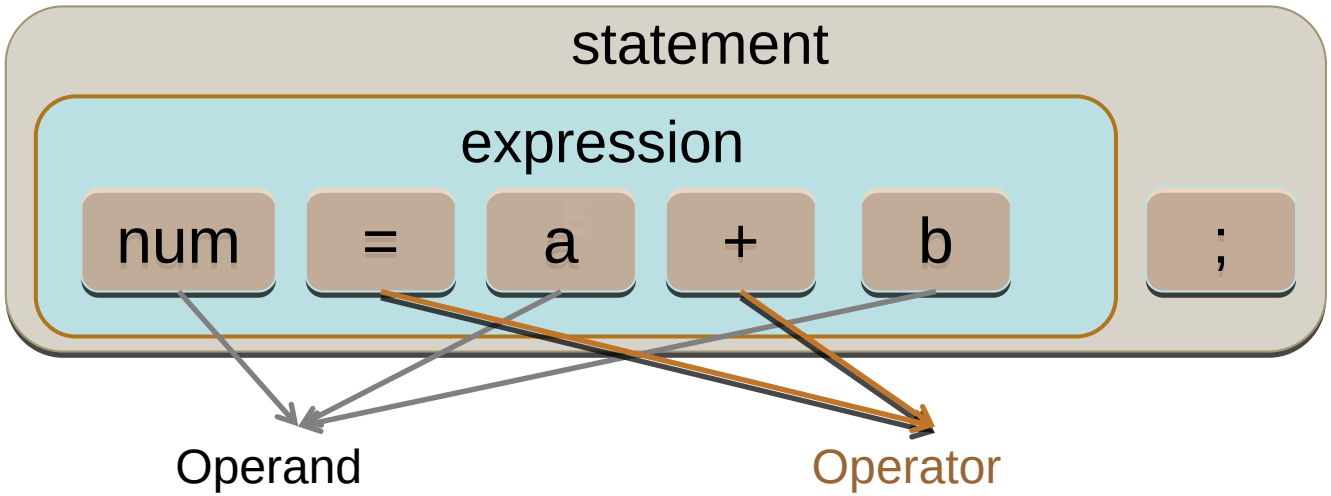


Expression

- Expression is composed of operands and operators
  - ***Operand***: *variables* or *constant*, such as num, 10, etc.
  - ***Operator***: +, -, *, /, %, =, >, <, &, |, !, (,)
  - Example:

num = a + b;

age = age + 1;





Arithmetic Operator

Operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	a + b	a + b
Subtraction	-	a – b	a – b
Multiplication	*	ab	a * b
Division	/	a / b	a / b
Remainder	%	a mod b	a % b

- Division
  - (int) / (int) = (int) (example: 5 / 2 = 2)
  - (float) / (int) = (float) (example: 5.0 / 2 = 2.5)
  - (int) / (float) = (float) (example: 5 / 2.0 = 2.5)
- Remainder
  - 10 % 3 = 1
  - Print %: use ***%%*** (example: print("10%%3 = %d\n", 10 % 3);



Operator Precedence

- Some arithmetic operators act before others.
  - Multiplication and division before addition and subtraction.
  - Use ***parenthesis*** to specify precedence.
    - Example: computer the average of a, b, and c
    - **Do not use**: a + b + c / 3
    - **Use**: (a + b + c) / 3

Operator	Precedence
( )	Evaluated first; Inner to outer; Left to right
*, /, %	Evaluated second; Left to right
+, -	Evaluated last; Left to right



Operator Precedence: Example

1. y = 2 * 5 * 5 + 3 * 5 + 7;
2. y = 10 * 5 + 3 * 5 + 7;
3. y = 50 + 3 * 5 + 7;
4. y = 50 + 15 + 7;
5. y = 65 + 7;
6. y = 72;



Equality and Relationship Operator

Operator	Condition	Meaning
Equality operator		
==	x == y	x is equal to y
!=	x != y	x is not equal to y
Relationship operator		
>	x > y	x is larger than y
<	x < y	x is smaller than y
>=	x >= y	x is larger than or equal to y
<=	x <= y	x is smaller than or equal to y

- Return **"truth" (1)** if the condition is true; otherwise, **"false" (0)**.
- Note that **==** (equality operator) is different from = (assignment operator)



Logical Operator

Operator	Condition	Meaning
&&	x && y	x **and** y
||	x || y	x **or** y

Truth Table

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

F: 0

T: any non-zero value

10 && 0 → **False**

5 || 0 → **Truth**

Used in flow control; can not be put in the statement (ex: **a = b && c**)



Concatenation Operator

Operator	Condition	Meaning
&	x & y	x **and** y (bit-wise and)
|	x | y	x **or** y (bit-wise or)
~	~x	**not** x (bit-wise inverse)

12 & 4 = 4 $(00001100)_2$ & $(00000100)_2$ = $(00000100)_2$
12 & 2 = 0 $(00001100)_2$ & $(00000010)_2$ = $(00000000)_2$
12 | 2 = 14 $(00001100)_2$ | $(00000010)_2$ = $(00001110)_2$
~12 = -13 ~$(00001100)_2$ $(11110011)_2$

```
printf("%d %d %d %d", 12&4, 12&2, 12 | 2, ~12);
```

Output:

```
4 0 14 -13
```



Unary Operator

- Need only one operand
  - +5 /* positive 5 */
  - -a /* = -1 * a */
  - !a /* not operator; if a = 0, !a = 1; otherwise, if a ≠ 0, !a = 0 */

```
01 int a = +5;
02 int b = -5;
03 int c = !a;
04 a = 0;
05 int d = !a;
06 int e = !b;
07 printf("c = %d, d = %d, e = %d\n", c, d, e);
```

Output:

c = 0, d = 1, e = 0



++ and --

Operator	Condition	Meaning
++	x++	x = x + 1
	++x	x = x + 1
--	x--	x = x - 1
	--x	x = x - 1

- x++: execute the statement first, and then add x by 1
  - int x = 5;
    int y = (x++) + x + 5; (result: y = 5 + 5 + 5 =15; x = 6)
- ++x: add x by 1, and then execute the statement
  - int x = 5;
    int y = (++x) + x + 5; (result: y = 6 + 5 + 5 =17; x = 6)



Compound Operator

Operator	Condition	Meaning
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5

No space before “=“



Operator Precedence

Precedence	Operators	Associative
1	++, --	Right to left
2	(), []	Left to right
3	!, -(negative), ~	Right to left
4	*, /, %	Left to right
5	+, -	Left to right
6	<<, >>	Left to right
7	>, >=, <. <=	Left to right
8	==, !=	Left to right
9	&	Left to right
10	^	Left to right
11	|	Left to right
12	&&	Left to right
13	||	Left to right
14	?:	Right to left
15	=	Right to left



Example

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

int main(void)
{
 int a = 2;
 int b = 5;
 int c = 10;
 int e;
 e= ++a + ++c/b+(c-2*b)|a&5 ;
 printf("a = %d, e = %d", a,e);
 system("pause");
}
```

Output:

```
a = 3, e=7
```



Lab 03

- Write a program to print the ASCII codes of ‘a’, ‘&’ and ‘\n’.
- Use ***sizeof()*** to show the size of the data types “char”, “short”, “int”, “float”, and “double”.