C Programming

Lecture 2: Primitive Data Types & Expressions

$$\begin{array}{r} 12 \\ + \ \text{-}13 \\ \hline \text{-}1 \end{array} \Leftrightarrow \begin{array}{r} 00001100 \\ + \ 11110011 \\ \hline 11111111 \end{array}$$

Lecturer: *Dr*. Wan-Lei Zhao

Spring Semester 2022

Email: wlzhao@xmu.edu.cn,

Outline

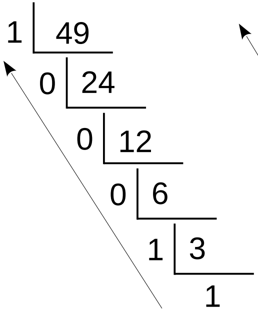
Everything is binary code in computer (1)

- Everything in computer is in **binary** form
- Data: integers, real numbers and strings
- Instructions
- Addresses: sequential numbers for the memory cells
- It is therefore necessary to know how the binary code is produced
- In addition, for convenience
- **Octal** and **Hexadecimal** numbers are also used for display

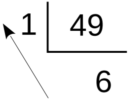
Everything is binary code in computer (2)

- Anyone watched this movie?

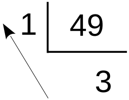
Decimal to binary



Decimal to octal



Decimal to hexadecimal



10 → A
11 → B
12 → C
13 → D
14 → E
15 → F

- Binary code: $110001_{(2)}$
- Octal code: $61_{(8)}$
- Hexadecimal code: $31_{(16)}$
- Can you figure out the relation between them

Decimal to Binary, Octal and Hexadecimal (2)

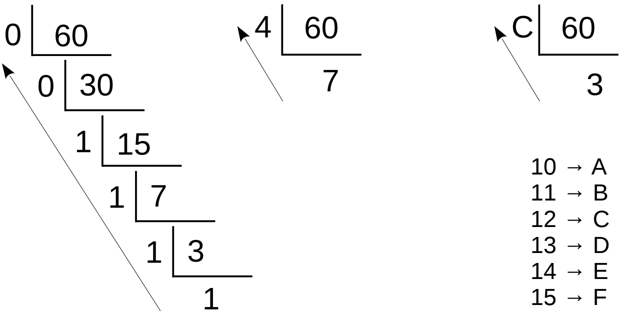
- Try it by yourself to convert **60** to
  - Binary code:
  - Octal code:
  - Hexadecimal code:

Decimal to Binary, Octal and Hexadecimal (2)

- Try it by yourself to convert **60** to
- Binary code: $111100_{(2)}$
- Octal code: $74_{(8)}$
- Hexadecimal code: $3C_{(16)}$

Decimal to Binary, Octal and Hexadecimal (3)

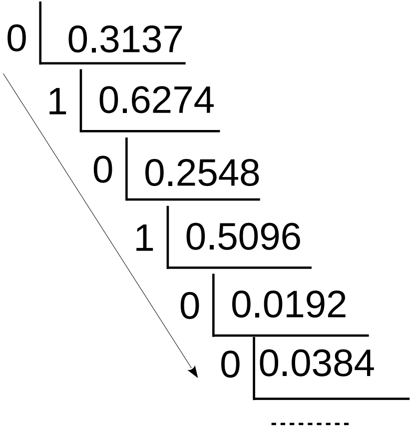
Decimal to binary   Decimal to octal   Decimal to hexadecimal



10 → A
11 → B
12 → C
13 → D
14 → E
15 → F

Decimal to Binary, Octal and Hexadecimal (4)

Decimal fraction to binary



$$0\times2^{-1}+1\times2^{-2}+0\times2^{-3}+1\times2^{-4}=0.3125\approx0.3137$$

Binary, Octal and Hexadecimal to Decimal

- Binary code: $111100_{(2)}$
- Octal code: $74_{(8)}$
- Hexadecimal code: $3C_{(16)}$

$$1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 60$$

$$7 \times 8^1 + 4 \times 8^0 = 60$$

$$3 \times 16^1 + 12 \times 16^0 = 60$$

- Data in the memory is kept in binary form
- Given an integer **49**, its binary code is $110001_{(2)}$
- It is kept in following form

0	1	1	0	0	0	1

- Given an integer -**49**, its binary code is $1110001_{(2)}$
- It is kept in following form

1	1	1	0	0	0	1

- The highest bit is reserved for sign
- This is true for **real** numbers later we will see
- We use 8 bits (1 byte), 2 bytes or more bytes to keep a number

1	0	1	1	0	0	0	1

Data in the memory (2)

- Data in the memory is kept in binary form
- Given we have several numbers to be kept
- They are kept one after another (assume we use 1 byte for one number)

0000	1	0	1	1	0	0	0	1
0001	0	0	1	1	0	0	1	1
0002	0	1	0	1	1	1	0	1
0003		.	.					

Data in the memory (3)

- Now, think about an important issue
- Given 1 byte, how many different numbers we can represent
- Assuming no sign bit

00	0	0	0	0	0	0	0	0
01	0	0	0	0	0	0	0	1
02	0	0	0	0	0	0	1	0
				⋮				
FF	1	1	1	1	1	1	1	1

- With 1 byte, there are $2^8 = 256$ numbers
- Since our memory are limited, we can only represent a limited range of numbers

Data in the memory (4)

- Now, think about how many different numbers we have if one bit is reserved for sign
- ????

Data in the memory (5)

- Now, think about how many different numbers we have if one bit is reserved for sign
- $2 \times 2^7 - 1 = 255$
- Only 127 positive numbers ($1 \sim 127$)
- 127 negatives (-1 $\sim$ -127)
- Some numbers can only be approximately represented by binary code
- For example, **3.3137**
- $11.0101_{(2)}$

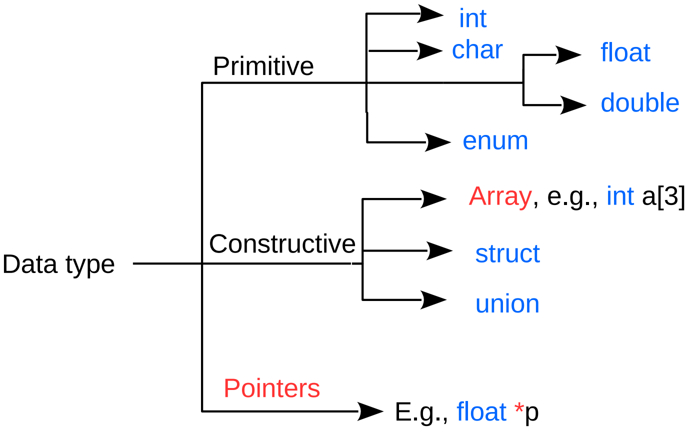
One's complement and Two's Complement

Original	bits	One's Complement	Two's Complement
23	00010111	00010111	00010111
-23	10010111	11101000	11101001
33	00100001	00100001	00100001
-33	10100001	11011110	11011111

- One's complement and two's complement of positive numbers are the same as original code
- For negative number, we do not inverse its sign bit
- Why we do so??
  - It is very convenient when we do substraction
  - Substraction is converted to add operation
- Now please work out one's complement and two's complement of **-17**

Outline

Data Types Supported in C



Integer numbers

- Keywords: int, short, long
- Can be *signed* (default) or *unsigned*
- Actual size of *int*, *short*, *long* depends on architecture

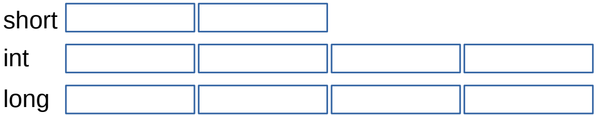
```
int a; /*Range: -2,147,483,648 to 2,147,483,647*/
short b;   /*Range: -32,768 to 32,767*/
long c; /*Range: -2,147,483,648 to 2,147,483,647*/
unsigned int a1;   /*Range: 0 to 4,294,967,297*/

unsigned short b1; /*Range: 0 to 65,535*/
```

- int and long take 4 bytes (32 bits system)
- short takes 2 bytes

Integer numbers

- Keywords: int, short, long
- Can be *signed* (default) or *unsigned*
- Actual size of *int*, *short*, *long* depends on architecture




```
int a; /*Range: −2,147,483,648 to 2,147,483,647*/
short b;    /*Range: −32,768 to 32,767*/
long c; /*Range: −2,147,483,648 to 2,147,483,647*/
unsigned int a1;    /*Range: 0 to 4,294,967,297*/

unsigned short b1; /*Range: 0 to 65,535*/
```

Integer numbers

```
int main()
{
    short a = 0x8000;
    short b = 0x7FFF;
    short c = 0xFFFE;
    char  d = 0x80;
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    printf("d = %d\n", d);
    return 0;
}
```

The Problem of Overflow (1)

- Given following code, anything wrong??

```
int main()
{
    unsigned short b = 65537;
    return 0;
}
```

The Problem of Overflow (2)

- Given following code, anything wrong??

```
int main()
{
    unsigned short b = 65537;
    return 0;
}
```

- **b** will never reach to **65537**
- In this case, it is **65535**
- Guess the value of b in following code

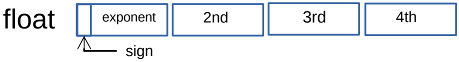
```
int main()
{
    short b = 65537;
    return 0;
}
```

The Problem of Overflow (3)

- The same problem exists for all primitive data types
- Because, we only use limited bytes to represent the data
- Be careful when you assign big value to a variable
- Tricks: estimate how big it could be

Floating point numbers (1)

- Keywords: float, double, long double
- Real numbers: $x \in R$
  - Due to limited memory, only 4 bytes/8 bytes are used for float/double
  - So it will not cover the whole range of R



[3.14159]

```
      0  0000100  11001001000011111001110
  ^        ^                    ^
  |        |                    |
  |        |                    +---- significand = 0.7853975
  |        |
  |        +------------------------- exponent = 4
  |
  +---------------------------------- sign = 0 (positive)
```

Floating point numbers (2)

- Keywords: float, double, long double

```
float x = 0.125;              /*Precision: 7 to 8
    digits*/
double y = 111111.111111;     /*Precision: 15 to 16
    digits*/
```

- Now you should know a very useful operator sizeof(.)

```
#include <stdio.h>
int main()
{
    float x = 0.125;
    double y = 111111.111111;
    printf("float: %d, double: %d", sizeof(x), sizeof(y));
}
```

Characters (1)

- Keyword: char
- Can be signed (default) or unsigned
- Size: 1 Byte (8 bits) on almost every architecture
- Intended to represent a single character
- Stores its *ASCII* number (e.g. 'A' $\Rightarrow$ 65)
- You can define a char either by its ASCII number or by its symbol:

```
char a = 65;
char b = 'A';    /*use single quotation marks*/
```

Characters (2)

- Essentially, char uses 1 byte to represent 255 characters
- Each integer is associated with a character
- American Standard Code for Information Interchange (ASCII)

0	*NUL*	16	*DLE*	32	SPC	48	0	64	@	80	P	96	`	112	p
1	*SOH*	17	*DC1*	33	!	49	1	65	A	81	Q	97	a	113	q
2	*STX*	18	*DC2*	34	"	50	2	66	B	82	R	98	b	114	r
3	*ETX*	19	*DC3*	35	#	51	3	67	C	83	S	99	c	115	s
4	*EOT*	20	*DC4*	36	$	52	4	68	D	84	T	100	d	116	t
5	*ENQ*	21	*NAK*	37	%	53	5	69	E	85	U	101	e	117	u
6	*ACK*	22	*SYN*	38	&	54	6	70	F	86	V	102	f	118	v
7	*BEL*	23	*ETB*	39	'	55	7	71	G	87	W	103	g	119	w
8	*BS*	24	*CAN*	40	(	56	8	72	H	88	X	104	h	120	x
9	*HT*	25	*EM*	41	)	57	9	73	I	89	Y	105	i	121	y
10	*LF*	26	*SUB*	42	*	58	:	74	J	90	Z	106	j	122	z
11	*VT*	27	*ESC*	43	+	59	;	75	K	91	[	107	k	123	{
12	*FF*	28	*FS*	44	,	60	<	76	L	92	\	108	l	124	|
13	*CR*	29	*GS*	45	-	61	=	77	M	93	]	109	m	125	}
14	*SO*	30	*RS*	46	.	62	>	78	N	94	^	110	n	126	~
15	*SI*	31	*US*	47	/	63	?	79	O	95	_	111	o	127	DEL

- There are some frequently used ones you should know

ASCII	value	ASCII	value
0∼9	48∼57	A∼Z	65∼90
a∼z	97∼122	␣␣␣	32
\n	10	\t	9

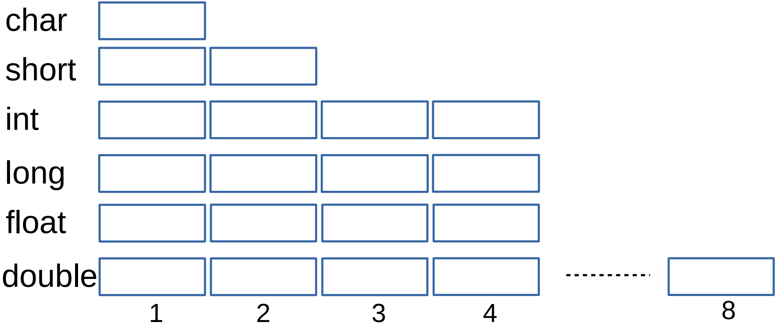
[Code]

```
#include <stdio.h>
int main()
{
    printf("A: %d %c\n", 'A', 'A');
    printf("1: %d %c\n", '1', '1');
    printf("B: %d %c\n", 66, 66);
    printf("2: %d %c\n", 50, 50);
}
```

[Output]

```
A: 65 A
1: 49 1
B: 66 B
2: 50 2
```

Data type and its size



- You should clearly know what is the use of your data
- One should not define data in double/long double just for convenience
- It wastes a lot of memory
- String: an **array** of chars

Outline

Variable: valid identifiers (1)

- Consist of English letters (a-z, A-Z), numbers (0-9) and underscore (_)
- Start with a letter (a-z, A-Z) or underscore (_)
- Are case sensitive (**number** differs from **Number**)
- Must not be reserved words (e.g int, return)
- Check which are valid identifiers

distance
milesPerHour
x-ray
2ndGrade
$amount
_2nd
two&four
_hi
return

Variable: valid identifiers (1)

- Consist of English letters (a-z, A-Z), numbers (0-9) and underscore (_)
- Start with a letter (a-z, A-Z) or underscore (_)
- Are case sensitive (**number** differs from **Number**)
- Must not be reserved words (e.g int, return)
- Check which are valid identifiers

distance	√
milesPerHour	√
x-ray	×
2ndGrade	×
$amount	×
_2nd	√
two&four	×
_hi	√
return	×

- Recommended style
  - Stay in one language (English recommended)
  - Decide whether to use camelCaseIdentifiers or snake_case_identifiers
  - When nesting blocks, indent every inner block by one additional tab!

```
#include <stdio.h>
int main()
{
    float width = 3.0, height = 5.0, area = 0.0;
    area = width*height;
    printf("Area is: %f\n", area);
    return 0;
}
```

Speaking identifiers

```
/* calculate volume of square pyramid */
int a, b, c;
a = 3;
b = 2;
c = (1 / 3) * a * a * b;
```

⇓

```
/* calculate volume of square pyramid */
int length, height, volume;
length = 3;
height = 2;
volume = (1 / 3) * length * length * height;
```

Constants

- Put key word 'const' before and type of variable definition
- The variable(s) become(s) constant(s)
- Constant means that you are not allowed to change the value after the definition

```
const int a = 5, b = 6;
const float c = 2.1;
```

```
#include <stdio.h>
int main()
{
        const float PI = 3.14159;
        float r = 3.0, area = 0.0;
        PI = 3.14;    /*Invalid*/
        area = PI*r*r;    /*'area' has been updated here*/
}
```

Variables and Constants

int a;

int a=3;

const int a=3;

Outline

- **printf("%d ...%f ...%ld", d1, d2, d3)**
- A function **pre-defined** by C
- It is in charge of print things onto screen
- You should organize your things in special format

[Codes]

```
#include <stdio.h>
int main()
{
    int a = 1;
    float b = 3.1;
    char c = 'h';
    printf("a: %d\n", a);
    printf("b: %f\n", b);
    printf("c: %c\n", c);
    printf("a: %d, c: %c\n", a, c);
}
```

[Output]

```
a: 1
b: 3.1
c: h
a: 1, c: h
```

printf() with placeholders (2)

- "%x" is called placeholder
- It **holds/occupies** the place that is replaced by output data
- Different output data require different placeholders
- The **order** of placeholders corresponds to the order of output
- The **number** of placeholders corresponds to the number of output

[Codes]

```c
#include <stdio.h>
int main()
{
  int a = 3;
  int b = 5;
  float c = 7.4;
  printf("a: %d\nb: %d\nc: %f\n", a, b, c);
}
```

[Output]

```
a: 3
b: 5
c: 7.4
```

Supported placeholders

- The placeholder determines how the value is interpreted.

type	description	type of argument
%c	single character	char, int (if $<=$ 255)
%d	decimal number	char, int
%u	unsigned decimal number	unsigned char, unsigned int
%x	hexadecimal number	char, int
%ld	long decimal number	long
%f	floating point number	float, double
%lf	double number	double

printf() by example

- **printf("%d ...%f ...%ld", d1, d2, d3)**
- A function **pre-defined** by C

[Codes]

```
#include <stdio.h>
int main()
{
   int a = 79;
   char b = 'n';
   printf("a: %d, b: %d\n", a, b);
   printf("a: %c, b: %c\n", a, b);
   printf("a: %x, b: %x\n", a, b);
}
```

[Output]

- **printf("%d ...%f ...%ld", d1, d2, d3)**
- A function **pre-defined** by C

[Codes]

```
#include <stdio.h>
int main()
{
   int a = 79;
   char b = 'n';
   printf("a: %d, b: %d\n", a, b);
   printf("a: %c, b: %c\n", a, b);
   printf("a: %x, b: %x\n", a, b);
}
```

[Output]

```
a: 79, b: 110
a: O, b: n
a: 4f, b: 6e
```

Escape Character in ASCII (1)

- There are some special character to be print out
  - "Tab", "Enter", "backspace"
- We want to express it by one character in ASCII
  - But....
  - All characters have their own use
- If we want to use them to express different meaning
  - We use '\'

Escape Character in ASCII (2)

- All characters have their own use
- If we want to use them to express different meaning
  - We use '\'

ESC	their charactor
'\t'	Tab
'\b'	back one character
'\r'	return to the start if a line
'\n'	go to the next line
'\\'	\
'\''	single quote: '
'\"'	double quote: "

- Remember that it is one character: \"
- It is valid: '\b'

Variable input

- **scanf("%d...%f", &a, &b)** is another useful function
- Like **printf**(), it is declared in **stdio.h**
- Like **printf**(), it has a format string with placeholders
- You can use it to read values of primitive datatypes from the command line

Example:

```
int i;
scanf("%d", &i);
```

- Notice that there is "&" before the variable
- This **operator** takes the address of the variable
- When buy goods online, you should put your the address
- The postman will transfer the **goods** (value) to your **mailbox** (variable)

Notes for scanf

- **scanf**() uses the same placeholders as **printf**()
- You must type an *&* before each variable identifier
- If you read a number (using %d, %u etc.), interpretation
  - Starts at first digit
  - Ends before last **non digit** character
  - E.g: **2 2.3**
- If you use %c, the first character of the user input is taken

- **scanf("%d ...%f ...%ld", &d1, &d2, &d3)**
- A function **pre-defined** by C

[Codes]

```
#include <stdio.h>
int main()
{
    int a = 79;
    float b = 0.1;
    printf("a: %d, b: %f\n", a, b);
    printf("Input a and b: ");
    scanf("%d%f", &a, &b);
    printf("a: %d, b: %f\n", a, b);
}
```

[Output]

```
a: 79, b: 0.1
Input a and b: xx xx.
    xx
a: xx, b: xx.xx
```

Outline

Overview about Expressions

- Legal expressions consist of legal combinations of
  - Constants: const float PI = 3.14
  - Variables: int a, b;
  - Operators: +,-
  - Function alls, printf("%d", a)

- Operators
  - Arithmetic: +,-,*, /, %
  - Relational: ==, !=, >,<, <=, >=
  - Logical: &&, !, ||
  - Bitwise: &, —, ^, ~
  - Shift: <<, >>

Arithmetic Operators in C

- Rules for operator precedence

Operator	Operation	Precedence
()	Parenthese	Evaluated first
*,/ or %	multiplication, division	evluated second
+ or -	addition, substraction	evaluated last

- Take average of three numbers
- 1+2+4/3 ??

Precedence Example

$$(2+3+5)/3$$
$$5*((2+6) \quad (1)$$

```
int avg = 2 + 3 + 5/3;
float x=5*2+6%2;
```

```
int avg = (2 + 3 + 5)
    /3;
float x=5*((2+6)%2);
```

- Try to use "()" to clarify, if you are uncertain about the precedence

Division Operator (1)

- Generates a result that is the same data type of the largest operand used in the operation
- Dividing two integers yields an integer result

```
5/2
17/5
```

[Result]

```
2
3
```

Division Operator (2)

- Generates a result that is the same data type of the largest operand used in the operation
- Dividing two integers yields an integer result

```
5.0/2
17.0/5
```

[Result]

```
2.5
3.4
```

Modulus Operator %

- Modulus Operator % returns the remainder
- Dividing two integers yields an integer result

```
5%2
17%5
12%3
```

[Result]

```
1
2
0
```

- See whether you can work out the answer

```
11/2
11%2
11/2.0
5.0/2
```

[Result]

Evaluating Arithmetic Expressions (2)

- Check your answer

```
11/2
11%2
11/2.0
5.0/2
```

[Result]

```
5
1
5.5
2.5
```

Arithmetic Expressions (1)

[Arithmetic Expression]

$$\frac{a}{b}$$

$$2x$$

$$\frac{x-7}{2+3y}$$

[Expression in C]

```
a/b
2*x
(x-7)/(2+3*y)
```

Arithmetic Expressions (2)

[Arithmetic Expression]

```
2 * (−3)
4 * 5 − 15
4 + 2 * 5
7/2
7 / 2.0
2 / 5
2.0 / 5.0
2 / 5 * 5
2.0 + 1.0 + 5 / 2
5 % 2
4 * 5/2 + 5 % 2
```

Arithmetic Expressions (3)

[Arithmetic Expression]	[Results]
2 * (−3)	−6
4 * 5 − 15	5
4 + 2 * 5	14
7/2	3
7 / 2.0	3.5
2 / 5	0
2.0 / 5.0	0.4
2 / 5 * 5	0
2.0 + 1.0 + 5 / 2	5.0
5 % 2	1
4 * 5/2 + 5 % 2	11

- Assig value to variable in accordance with its type

```c
int main()
{
    int a;
    a = 2.99;
    printf("a = %d", a);
}
```

[Output]

```
a = 2
```

- Comments: above expression is valid, but NOT suggested

Shortcut assignment Operators (1)

Assignment	Shortcut
d = d - 4	d-=4
e = e*5	e *= 5
f = f/3	f /= 3
g = g%9	g %=9
m = m*(5 + 3)	m *= 5+3
k = k/(5 + 1)	m /= 5+1
k = k/(5*7)	k /= 5*7

Shortcut assignment Operators (2)

```c
a += 4;        /* a = a + 4; */
a -= 4;        /* a = a - 4; */
a *= b;        /* a = a * b; */
b /= 4+2;      /* b = b / (4+2); */
b %= 2+3;      /* b = b % (2+3); */
```

Shorthand Operators (1)

- Incremental operator: ++
  - **i++** equivalent to **i = i+1**
- Decremental operator: –
  - **i–** equivalent to **i = i-1**
- When they are used alone
  - **i++** and **++i** behave the same as
  - **i = i+1**
  - Similar comment applies to –

Shorthand Operators (2)

- When they appear in a compound expression, things are different
- **a=i++** will be different from **a=++i**
- In **a=i++**, **i** contributes its value to **a** first, then self-increments
- In **a=++i**, **i** self-increments first, then contributes its value to **a**
- Similiar comments apply to **i–** and **–i**

```
int main()
{
    int a, b, i = 4;
    a = i++;
    b = ++i;
}
```

```
int main()
{
    int a, i = 4;
    a = i;
    i = i + 1;
    i = i + 1;
    b = i;
}
```

Shorthand Operators (3)

- Now verify how much you understand

```c
int main()
{
    int a, b, i = 4;
    a = i--;
    b = --i;
    printf("a = %d, b = %d\n", a, b);
}
```

[Output]

```
a = ?, b = ?
```

- Conditional Operator: logic_exp1?exp2:exp3
- Three operands
- If logic_exp1 is none zero, takes **exp2**
- If logic_exp1 is zero, takes **exp3**

```
int main()
{
    int a = 2, b = 3, i = 4;
    a = b>i?b:i;
    b = b==3?2:1;
    printf("a = %d, b = %d\n", a, b);
}
```

[Output]

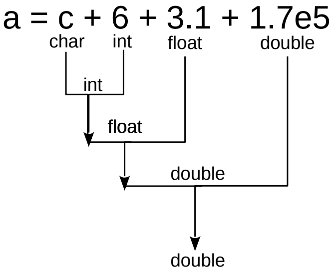
```
a = 4, b = 2
```

Outline

Implicit Data Type Casting

- See whether you can work out the answer

```
char c = 'x';
double a = c + 5 + 1.3 + 1.73e4;
```



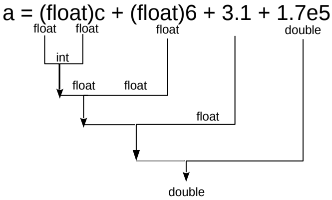
- Above type castings are done automatically (implicitly)
- Code below is risky, rear part will be truncated

```
int a = 0;
a = 5.1;
```

Explicit (forceful) Data Type Casting

- See whether you can work out the answer

```
char c = 'x';
double a = (float)c + (float)5 +
    1.3 + 1.73e4;
```



- Above type castings are done forcefully
- Again it is risky sometimes

```
int a = 0;
float b = 5.4;
a = (int)b;
```

```
int a = 0;
float b = 5.4;
a = (int)round(b);
```