Computer Programming using C

Fundamental Data Tapes

Instructor: HOU, Fen

2025

The Importance of Data Types

- Variable Declarations:
  - Reserve an appropriate amount of memory.
  - Ensure correct operations on variables.
- Expressions:
  - Combinations of constants, variables, and function calls.
  - Have value and type.

Fundamental data types

Fundamental data types: Long Form		
`char`	`signed char`	`unsigned char`
`float`	`signed short int`	`unsigned short int`
`double`	`signed int`	`unsigned int`
`long double`	`signed long int`	`unsigned long int`

- Abbreviations:
  - `signed int` ⇔ `int`
  - `short int` ⇔ `short`
  - `long int` ⇔ `long`
  - `unsigned int` ⇔ `unsigned`

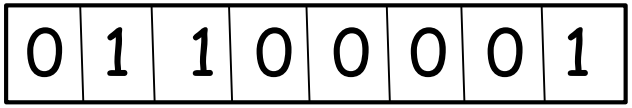
Fundamental data types

Fundamental data types: Short Form		
`char`	`signed char`	`unsigned char`
`float`	`short`	`unsigned short`
`double`	`int`	`unsigned`
`long double`	`long`	`unsigned long`

Fundamental data types grouped by functionality			
Integral Types	`char`	`signed char`	`unsigned char`
	`short`	`int`	`long`
	`unsigned short`	`unsigned`	`unsigned long`
Floating Types	`float`	`double`	`long double`
Arithmetic Types	***`Integral types + Floating types`***		

Fundamental Data Types

- The fundamental data types can be grouped according to functionality.
- The integral types: They are types that can be used to hold integer values.
- For example, char variable 'a' is store as a decimal integer value 97, which is the integer value of the char 'a' in ASCII Table.



- The floating types: They are whose can be used to hold real values.
- All types belong to arithmetic types.

1. The Data Type char

- What is **`char`**?
  - One of the fundamental data types in C.
  - Representing a single character.
  - Enclosed inside `' '`.
  - Examples are **`'A'`**, **`'d'`**, **`'9'`**, **`'+'`**, **`'='`**, ...

- Internal Representation of a char
  - A char can be thought of as an integer value (compatible with BUT **not the `int` data type**).
  - For example,
    - **`'A'`** is stored as integer value 65.
    - **`'+'`** is stored as integer value 43.

The Data Type char (con't)

- What integer value represents what character?
  - The ASCII table (shown on the next page) defines the standard mappings between characters to their corresponding integer values.

The 7-bit ASCII Table

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	NL	VT	NP	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	SP	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	I	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL		

ASCII

- ASCII stands for "American Standard Code for Information Interchange."
- Some are printable characters:

  `'A'`, `'='`, `':'`, …
- Some are non-printable characters: newline (NL), bell (BEL), tab (HT, VT), …

How does a byte store a `char`?

- Each character is stored in one byte.
- One byte → 8 bits → 2^8 possibilities
- 256 distinct values can be represented.
- The full range is 0 – 255. ASCII uses 0 – 127.

0	0	0	0	0	0	0	0	0	NUL
0	0	0	0	0	0	0	1	1	SOH
0	0	0	0	0	0	1	0	2	STX
...									
0	1	0	0	0	0	0	1	65	A
0	1	0	0	0	0	1	0	66	B
...									
0	1	1	1	1	1	1	1	127	DEL

Some character constants and their integer ASCII values	
ASCII of Lowercase Letters	**'a' 'b' 'c' … 'z'** **97 98 99 122**
ASCII of Uppercase Letters	**'A' 'B' 'C' … 'Z'** **65 66 67 90**
ASCII of Digits	**'0' '1' '2' … '9'** **48 49 50 57**
ASCII of Other Characters	**' ' '#' '*' '+' …** **32 38 42 43**

Nonprinting or hard-to-print characters		
Name of character	**Written in C**	**ASCII Value**
alert	`\a`	7
backslash	`\\`	92
backspace	`\b`	8
carriage return	`\r`	13
double quote	`\"`	34
form feed	`\f`	12
horizontal tab	`\t`	9
newline	`\n`	10
null character	`\0`	0
single quote	`\'`	39
vertical tab	`\v`	11

- Escape character (\)
- Escape sequence (e.g. `\n`)

Some Examples

`printf ("%c", 'a');`	
`printf ("%d", 'a');`	
`printf ("%c", 97);`	
`printf ("%d", 'a' * 2 - 7);`	
`printf ("H\tello");`	
`printf ("\07"); // in octal!`	***beep***
`printf ("\"Hello!\"");`	

Some Examples

`printf ("%c", 'a');`	`a`
`printf ("%d", 'a');`	`97`
`printf ("%c", 97);`	`a`
`printf ("%d", 'a' * 2 - 7);`	***187***
`printf ("H\tello");`	`H ello`
`printf ("\07"); // in octal!`	***beep***
`printf ("\"Hello!\"");`	`"Hello!"`

2. The Data Type `int`

- …, -3, -2, -1, 0, +1, +2, +3, …
- Computers can store only a finite portion of natural numbers. That is, upper and lower limits exist.
- An int is stored in 2 bytes, 4 bytes, or even larger.

2-byte `int`

- 2 bytes , 16 bits
- 2^{16} = 65536 distinct values can be stored.
- Half for negative integers, and half for non-negative integers.

  $-(2^{15}), -(2^{15}) + 1, \ldots , -2, -1, 0, 1, 2, \ldots, 2^{15} - 1$

- That is,

  -32768, -32767, ., -2, -1 0, 1, 2, …, 32767

4-byte `int`

- 4 bytes , 32 bits
- 2^{32} = 4294967296 distinct values can be stored.
- Half for negative integers, and half for non-negative integers.

  $-(2^{31}), -(2^{31}) + 1, \ldots , -2, -1, 0, 1, 2, \ldots, 2^{31} - 1$
- That is,
  -2147483648, …, -2, -1, 0, 1, 2, …, 2147483647

Finding out the size of an integer

`sizeof(int)`

- Can be used to find out the number of bytes used for storing an int.
- Example below.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("size of int   = %d\n", sizeof(int) );
6      return (0);
7  }
```

```
size of int   = 4
```

The Overflow Problem

```
1   #include <stdio.h>
2   #define BIG 2000000000
3
4   int main(void)
5   {
6       int a, b=BIG, c=BIG;
7
8       printf("Size of int = %d bits\n", sizeof(int) * 8);
9
10      a = b + c;
11      printf ("a=%d, b=%d, c=%d\n", a, b, c);
12      return(0);
13  }
```

- `b + c` > $2^{31} - 1 \approx 4$ billions
- An 32-bit integer variable cannot hold this large value.
- Variable `a` overflows.

How to solve the overflow problem?

```
Size of int = 32 bits
a=-294967296, b=2000000000, c=2000000000
```

`int` variation – `short int`

- May take less number of bytes than **`int`**.
- May thus store a smaller integer value.
- For example,
  - Suppose your computer uses 4-byte **`int`**, ranging from -2147483648 to +2147483647.
  - A **`short int`** may occupy 2 bytes, ranging from -32768 to +32767.

`int` variation – `long int`

- May take more number of bytes than **`int`**.
- May thus store a larger integer value.
- For example,
  - Suppose your computer uses 4-byte **`int`**, ranging from -2147483648 to +2147483647.
  - A **`long int`** may occupy 8 bytes, ranging from $-(2^{63})$ to $+(2^{63} - 1)$.

`int` variation – `unsigned int`

- Take same number of bytes as int.
- But stores non-negative integers only.
- Range: $0 – 2^{int\_length} – 1$
- 2-byte unsigned int:
  - 0 to $2^{16} - 1$
  - 0 to 65535.
- 4-byte unsigned int:
  - 0 to $2^{32} - 1$
  - 0 to 4294967295.

`int` variation – `unsigned int`

- Note that combinations of **`short`** and **`long`** with **`unsigned`** are possible, leading to:
  - **`unsigned short int`**
  - **`unsigned long int`**

- Note the use of unsigned in declaring variable **a**.
- Note the use of **%u** in the **printf**.
- Overflow is avoided. Why?

```
1   #include <stdio.h>
2   #define BIG 2000000000
3
4   int main(void)
5   {
6       unsigned int a, b=BIG, c=BIG;
7
8       printf("Size of int = %d bits\n", sizeof(int) * 8);
9
10      a = b + c;
11      printf ("a=%u, b=%u, c=%u\n", a, b, c);
12      return(0);
13  }
```

```
Size of int = 32 bits
a=4000000000, b=2000000000, c=2000000000
```

3. The Floating Types

- What is a Floating-Point value?
  - Store real values such as 0.1, -2.4, 3.14159.
  - Exponential notation:
  - **1.234567e5 = 1.234567 × 10^5**
    = 123456.7
  - **1.234567e-3 = 1.234567 × 10^{-3}**
    = 0.001234567

4. The `sizeof` Operator

- Used to find the number of bytes needed to store a piece of data.
- General form,

```
sizeof(data_type_or_name)
```

- **`data_type_or_name`** can be a valid data type (such as **`int`**, **`float`**) or a variable name that has already been declared.
- e.g.

```
sizeof( long double )
sizeof( i )
```

```c
#include <stdio.h>

int main(void)
{
  printf("\n");
  printf("Here are the sizes of some fundamental types:\n\n");
  printf("       char:%3d byte \n", sizeof(char));
  printf("      short:%3d bytes\n", sizeof(short));
  printf("        int:%3d bytes\n", sizeof(int));
  printf("       long:%3d bytes\n", sizeof(long));
  printf("   unsigned:%3d bytes\n", sizeof(unsigned));
  printf("      float:%3d bytes\n", sizeof(float));
  printf("     double:%3d bytes\n", sizeof(double));
  printf("long double:%3d bytes\n", sizeof(long double));
  printf("\n");

  return (0);
}
```

```
Here are the sizes of some fundamental types:

        char:  1 byte
       short:  2 bytes
         int:  4 bytes
        long:  4 bytes
    unsigned:  4 bytes
       float:  4 bytes
      double:  8 bytes
long double: 16 bytes
```

Note: Execution results vary with different machines and compilers.

5. Data Type Conversions

Arithmetic Conversions?

- An arithmetic expression has both a value and a type.
- For example, if both x and y have type int, the expression x + y also has type int.
- When an expression contains operands of different types, *unification* on data type is needed.
- This is achieved **implicitly** *by the compiler* using *arithmetic conversions*.

The Integral Promotion

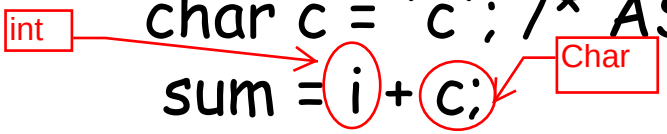
- A char or short (either signed or unsigned) can be used in any expression where an int or unsigned int may be used.

- In any expression, you can always use a variable whose type ranks lower than int in place of an oprand of type int or unsigned int. In these cases, the complier applies integer promotion: that is, any operand whose type ranks lower than int is automatically converted to the type int, provided int is capable of representing all values of the operand's original type. If int is not sufficient, the operand is converted to unsigned int.

The Integral Promotion

- What is the result shown on the screen after the following code is compiled and executed?

```
#include <stdio.h>
main()
{
int sum, i = 17;
char c = 'c'; /* ASCII value is 99 */
 sum = i + c;
printf("Value of sum : %d\n", sum );
getchar();

}
```



The Integral Promotion

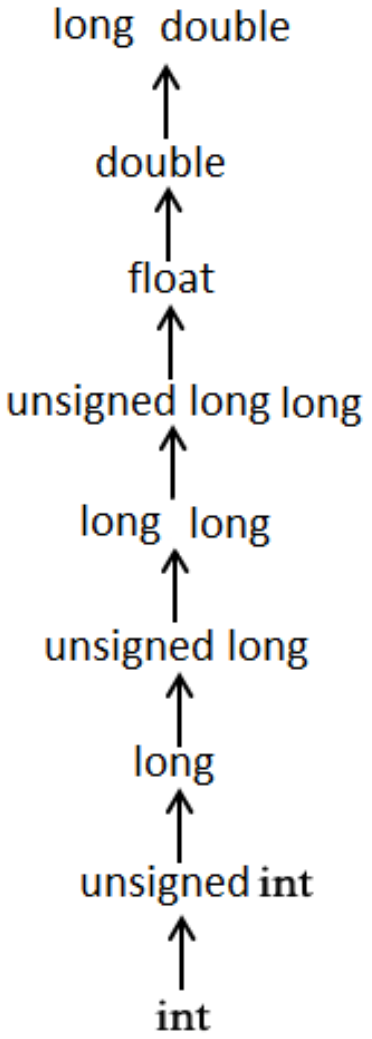
- When the following code is compiled and executed, it produced the following result.

```
#include <stdio.h>
main()
{
int sum, i = 17;
char c = 'c'; /* ASCII value is 99 */
 sum = i + c;
printf("Value of sum : %d\n", sum );
getchar();
}
```

Shown on the screen: Value of sum : 116

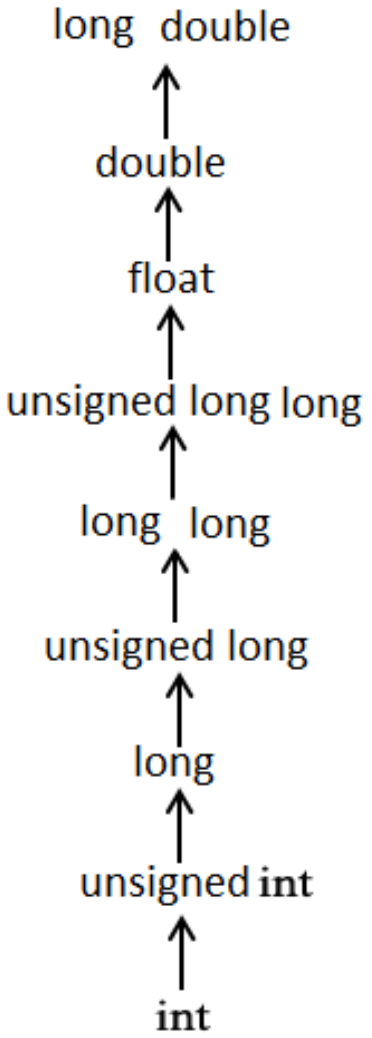
Arithmetic Conversion

- The usual arithmetic conversions are implicitly performed to cast their values to a common type. The complier first performs integer promotion; if the operands still have different types, then arithmetic conversion is used to convert these operands to the type that appears highest in the following hierarchy.
- Hierarchy of Type



Arithmetic Conversion

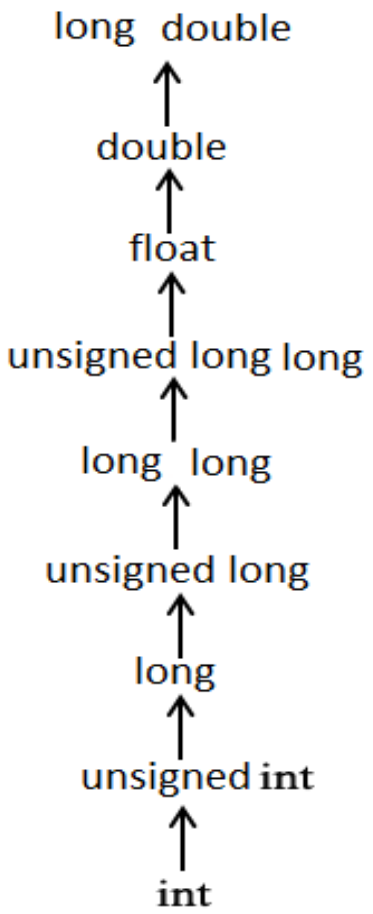
- The usual arithmetic conversions are not only performed for mixed expressions, but also performed across an assignment.



Arithmetic Conversion

- The arithmetic conversion in the following code.

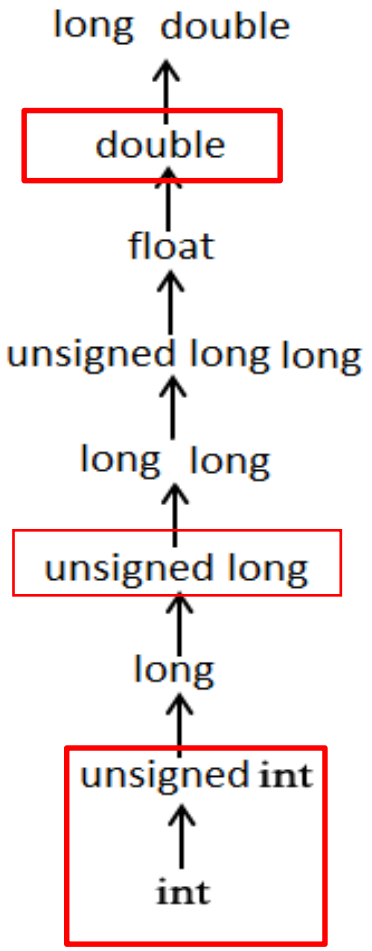
```
#include <stdio.h>
int main(void)
{
float       fVal;
double      dVal;
int          iVal;
unsigned long ulVal;
dVal = iVal * ulVal;

dVal = ulVal + fVal;



getchar();
return (0);
}
```



Arithmetic Conversion

- The arithmetic conversion in the following code.

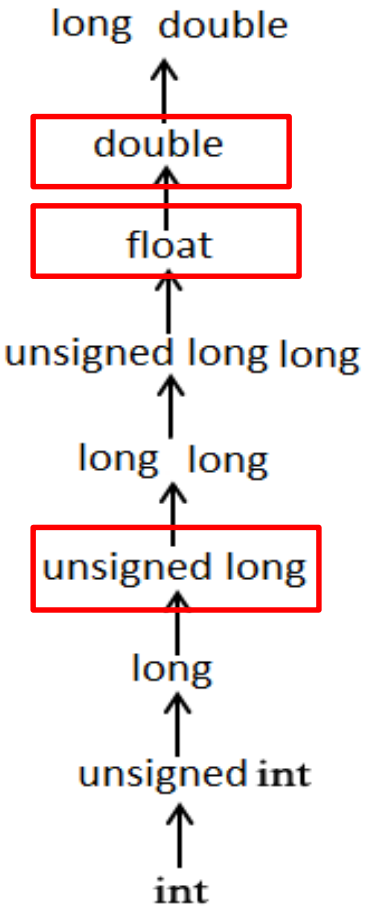
```
#include <stdio.h>
int main(void)
{
float       fVal;
double      dVal;
int          iVal;
unsigned long ulVal;
dVal = iVal * ulVal; /* iVal converted to unsigned long
                      * Result of multiplication converted to double
                      */
dVal = ulVal + fVal; /* ulVal converted to float
                      * Result of addition converted to double
                      */
getchar();
return (0);
}
```



Arithmetic Conversion

- The arithmetic conversion in the following code.

```
#include <stdio.h>
int main(void)
{
float       fVal;
double      dVal;
int          iVal;
unsigned long ulVal;
dVal = iVal * ulVal; /* iVal converted to unsigned long
                      * Result of multiplication converted to double
                      */
dVal = ulVal + fVal; /* ulVal converted to float
                      * Result of addition converted to double
                      */
getchar();
return (0);
}
```



Declaration		
`char c;`	`short s;`	`int i;`
`unsigned u;`	`unsigned long ul;`	`float f;`
`double d;`	`long double ld;`	`long l;`

Expression	Type	Expression	Type
`c - s / i`		`u * 7 - i`	
`u * 2.0 - i`		`f * 7 - i`	
`c + 3`		`7 * s * ul`	
`c + 5.0`		`ld + c`	
`d + s`		`u - ul`	
`2 * i / l`			

Declaration		
`char c;`	`short s;`	`int i;`
`unsigned u;`	`unsigned long ul;`	`float f;`
`double d;`	`long double ld;`	`long l;`

Expression	Type	Expression	Type
`c - s / i`	int	`u * 7 - i`	unsigned
`u * 2.0 - i`	double	`f * 7 - i`	float
`c + 3`	int	`7 * s * ul`	unsigned long
`c + 5.0`	double	`ld + c`	long double
`d + s`	double	`u - ul`	unsigned long
`2 * i / l`	long		

Casting

- **Explicit** type conversion ***of a value*** by the programmer.

Cast the value of i. That is, convert the value of i to another type.

- Example,

```
double d = 3.14159265;
float  f = (float) 2.0;  // avoid warning
float  g;
g = (float) d;
```

Casting (con't)

- Casts can be applied to expressions.

  (type_name) expression

- The cast operator has higher precedence over the arithmetic operators.

Casting (con't)

- Examples,

```
double   US_rate = 7.78932;
unsigned exchange;
long     approximate;

exchange = (unsigned) (US_rate * 109700);
approximate = (long) US_rate;
```

- Examples,

```
printf("%f\n", 4/2);
printf("%f\n", 4.0/2);
printf("%f\n", (float) 4/2);
```

Casting (con't)

- Examples,

```
double   US_rate = 7.78932;
unsigned exchange;
long     approximate;

exchange = (unsigned) (US_rate * 109700);
approximate = (long) US_rate;
```

- Examples,

```
printf("%f\n", 4/2);  /* 0.000000 incorrect result*/
printf("%f\n", 4.0/2);                /* 2.000000 */
printf("%f\n", (float) 4/2);         /* 2.000000 */
```

Casting (con't)

- Examples,

```
double   US_rate = 7.78932;
unsigned exchange;
long     approximate;

exchange = (unsigned) (US_rate * 109700);
approximate = (long) US_rate;
```

- Examples,

```
printf("%f\n", 14/3); /* 0.000000 incorrect result */
printf("%f\n", 14.0/3);           /* 4.666667 */
printf("%f\n", (float) 14/3);     /* 4.666667 */
```

Casting (con't)

- Examples,

```
double   US_rate = 7.78932;
unsigned exchange;
long     approximate;

exchange = (unsigned) (US_rate * 109700);
approximate = (long) US_rate;
```

- Examples,

```
printf("%f\n", 14/3); /*0.000000 incorrect result*/
printf("%d\n", 14/3); /* 4 incorrect result */
printf("%f\n", (float) 14/3);     /* 4.666667 */
```

Default Type

- Assumed type of integer constants in a C program is *normally* **`int`**.

  e.g. a = 3 + 109700; // both 3 and 109700 are int

- Assumed type of real number constants in a C program is **`double`**.

  e.g. d = 2.0; // the number 2.0 is a double

Casting

- **Explicit** type conversion ***of a value*** by the programmer.

- Example,

```
double d = 3.14159265;
float  f = (float) 2.0;  // avoid warning
float  g;
g = (float) d;
```

- What is the data type of variable **d** after the casting operation?

Casting (con't)

- For example. Given that i is an int.

```
(double) i
```

- Casts the value of i. That is, convert the value of i to type double.

Casting (con't)

- For example. Given that i is an int.

  (double) i

- Casts the value of i. That is, convert the value of i to type double. However, the variable i itself remains unchanged. It is still the type of int.

Casting (con't)

- For example,

```
#include <stdio.h>
main() {
int sum = 17, count = 5;
double mean;
mean = (double) sum / count;
printf("Value of mean : %f\n", mean );
 }
```

- When the above code is complied and executed, it produces the following result: The value of sum is first converted to type double, and finally it gets divided by count yielding a double value, and assign to the variable mean.

Casting (con't)

- For example,

```
#include <stdio.h>

int main(void)
{int i;
float f, sum;
i = 2, f = 2.012345;
printf("f=%d\n", (int)f);

sum = f + (float)i;
printf("sum=%f\n", sum);

printf("i=%d\n", i);

getchar();
return(0);

}
```

Casting (con't)

- For example,

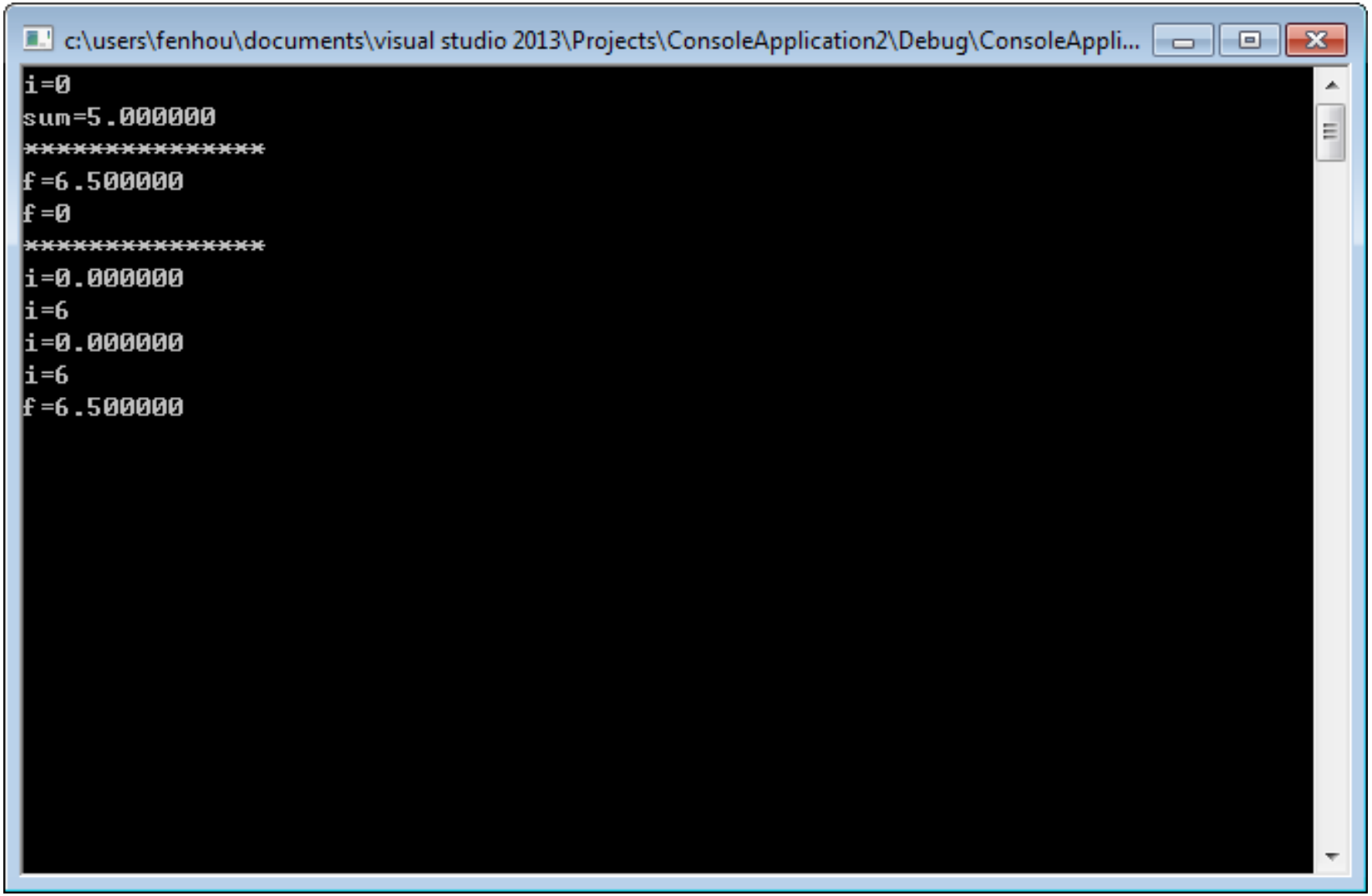
```
#include <stdio.h>

int main(void)
{int i;
float f, sum;
i = 2, f = 2.012345;
printf("f=%d\n", (int)f);

sum = f + (float)i;
printf("sum=%f\n", sum);

printf("i=%f\n", i);

getchar();
return(0);

}
```

Casting (con't)

- For example,

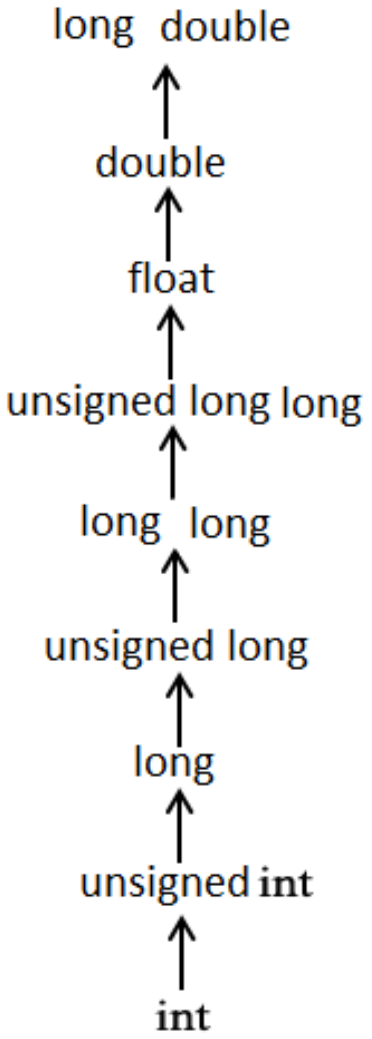
```
#include <stdio.h>

int main(void)
{int i;
float f, sum;
i = 2, f = 2.0123456;
printf("f=%____\n", (int)f);

sum = f + (float)i;
printf("sum=%____\n", sum);

printf("i=%____\n", i);

getchar();
return(0);

}
```

```c
#include <stdio.h>
int main(void)
{int i =5;float f=5;
f = i;
printf("i=%d\n", f=i);
printf("sum=%f\n", f=i);
printf("***************\n");
printf("f=%f\n", f = i + 1.5);
printf("f=%d\n", f = i + 1.5);
printf("***************\n");
printf("i=%f\n", i=f);
printf("i=%d\n", i=f);
printf("i=%f\n", i);
printf("i=%d\n", i);
printf("f=%f\n", f);}
```


c:\users\fenhou\documents\visual studio 2013\Projects\ConsoleApplication2\Debug\ConsoleAppli...
i=0
sum=5.000000
****
f=6.500000
f=0
****
i=0.000000
i=6
i=0.000000
i=6
f=6.500000

Conversion and Casts

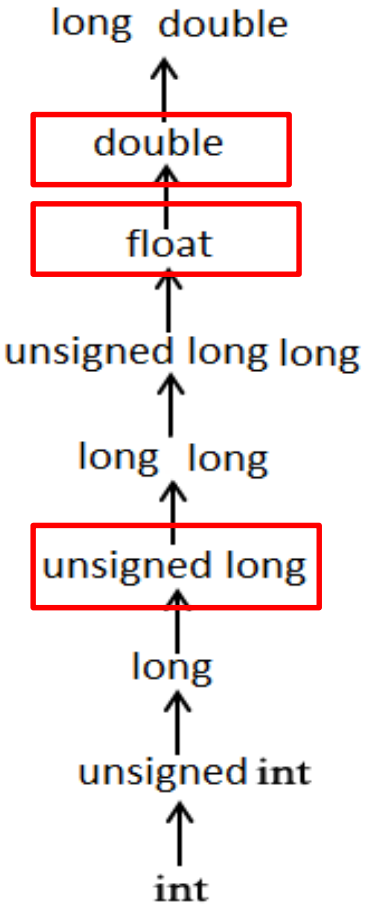
- The usual arithmetic conversions are implicitly performed to cast their values to a common type. The complier first performs integer promotion; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy.
- Hierarchy of Type
- The usual arithmetic conversions are not only performed for mixed expressions, but also performed across an assignment.



Arithmetic Conversion

- The arithmetic conversion in the following code.

```
#include <stdio.h>
int main(void)
{
float       fVal;
double      dVal;
int          iVal;
unsigned long ulVal;
dVal = iVal * ulVal;  /* iVal converted to unsigned long
                       * Result of multiplication converted to double
                       */
dVal = ulVal + fVal;  /* ulVal converted to float
                       * Result of addition converted to double
                       */
getchar();
return (0);
}
```



```
int main(void)
{
        float a = 5, b = 2;
        printf("float number is %f\n", a/b);
        printf("float number is %d\n", (int) (a / b));
        printf("float number is %d\n", (int) a / b);
        printf("float number is %d\n", (int) a / 2);
        getch();
        return(0);
}
```

```
int main(void)
{
        float a = 5, b = 2;
        printf("float number is %f\n", a/b);
        printf("float number is %d\n", (int) (a / b));
        printf("float number is %d\n", (int) a / b);
        printf("float number is %d\n", (int) a / 2);
        getch();
        return(0);
}
```