Type conversion

EECS 2031

Gias Uddin
guddin@yorku.ca
https://giasuddin.ca/

Acknowledgement

- The covered materials are based on previous EECS2031 offerings:
  - Song Wang, Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

More Bitwise operators

C/C++ (and Java,Python) allows us to easily manipulate individual bits in **integer** types (**`char, short, int, long`**)

- bitwise `& | ~ ^`

01001000	01100101	01101100	01101100
01100101	01101100	01101100	01101111

And

p	q	$p \cdot q$
T	T	T
T	F	F
F	T	F
F	F	F

Or

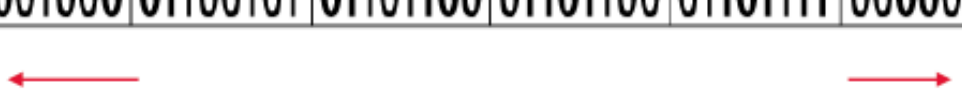
p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Not

p	$\sim p$
T	F
F	T

- bit shifting `<< >>`

01001000	01100101	01101100	01101100	01101111	00000000



~ flips all bits in its operand

e.g.

```
int z= ~145;
```

```
000 ..... 10010001
-------------------
111 ..... 01101110   = -146
```

z= ~x does not change x

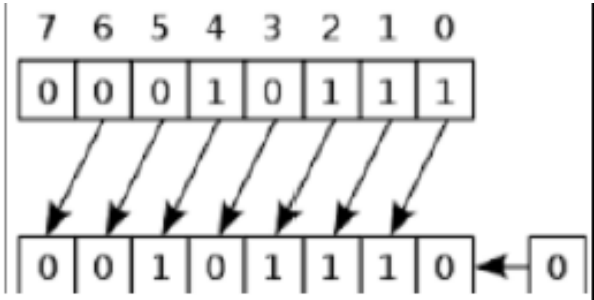
Bit Shifting

- Shifting bits: << (left shift), >> (right shift)
  - `x << n` means "take *x* and shift it *n* bits to the left"
  - `x >> n` means "take *x* and shift it *n* bits to the right"
  - Result is an int value (but does not change `x`)

What goes out? bits pushed "off the end" on the end

What comes in? >> << different

Bit Shifting <<

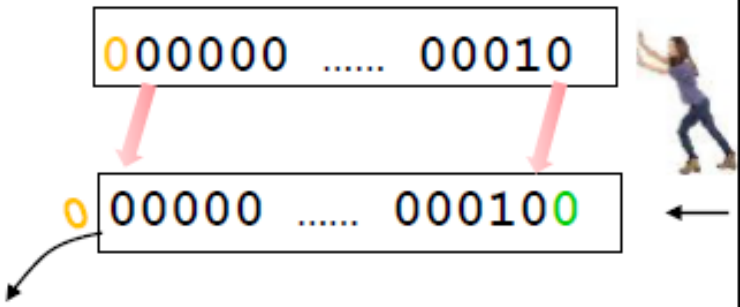


- Suppose `z` is an int
  - e.g.

    ```
    int z= 2 << 1;
    ```

    shift left 1 bits

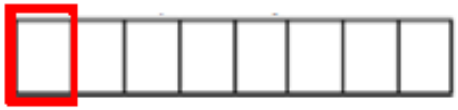
    `z: 4`



What goes out? bits pushed "off the end" on the left end

What comes in? we add 0 on the right

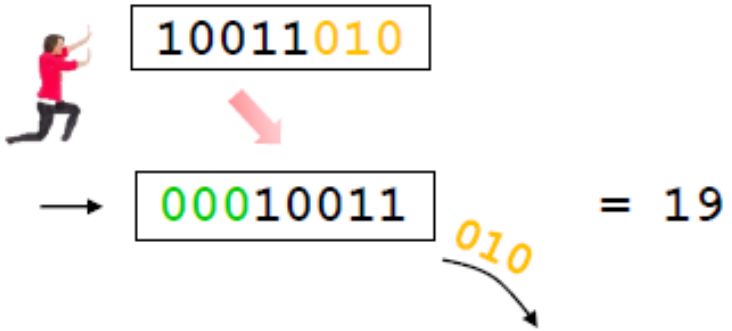
Bit Shifting >>



- What if we shift right? >> complicated.
- For "unsigned" types – all bits are magnitude -- add 0 on left

- e.g. (assume these are all unsigned)

```
unsigned char c = 154;
z = c >> 3;
```



z = c >> 3 does not change c

Bit Shifting- What Comes In >>

- What about "signed" values?
  - It's undefined, meaning
    - On some platforms it's logical (0's – like unsigned values)
    - On others it's arithmetic (whatever the leftmost bit is)

- e.g.(8-bit signed values using 2's complement)
  - `signed char c = -94;` 10100010
  - `c >> 3;`
    - logical → 00010100 20
    - arithmetic → 11110100 -12 (lab)

Bit Shifting- What Comes In >>

- What about "signed" values?
  - It's undefined, meaning
    - On some platforms it's logical (0's – like unsigned values)
    - On others it's arithmetic (whatever the leftmost bit is)

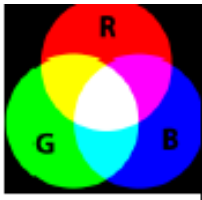
C does not define which method is used
The moral:
Avoid right bit-shifting signed values!

Java address right shift by introducing >>>
>> whatever leftmost is
>>> always 00…

(lab)

Some examples

- | 1: turn on
- & 0: turn off
- | 0: keep value
- & 1: keep value



- In Java, getRGB() packs 3 +1 values into a 32 bit (4 bytes) int
- How to get *blue* value?

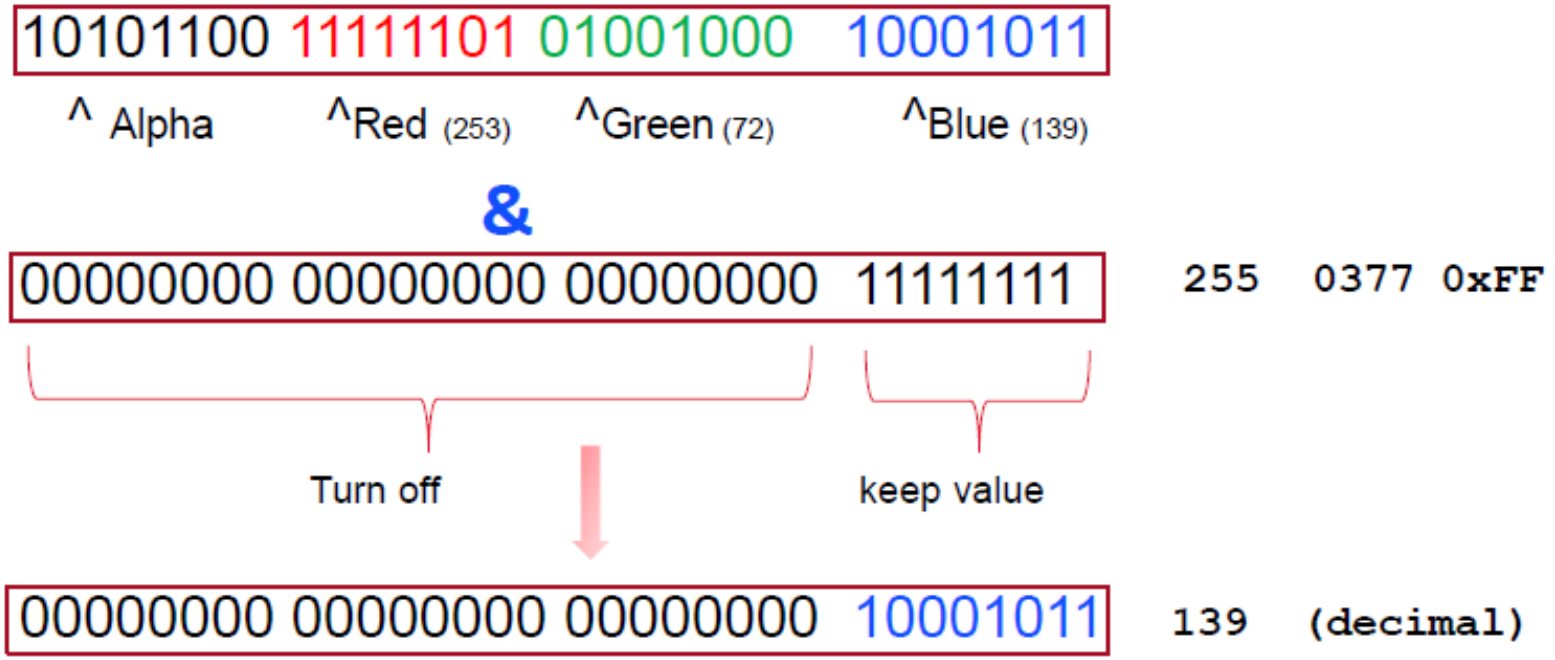
10101100 11111101 01001000 10001011

^ Alpha ^Red (253) ^Green (72) ^Blue (139)

Need to keep lower 8 bits, turn off others.
How?

00000000 00000000 00000000 10001011 139 (decimal)

How to get *blue* value?




```
int blue = rgb  & 255        /* rgb not changed */
         = rgb  & 0377
         = rgb  & 0XFF
```

How to get *red* value?

10101100 11111101 01001000 10101011

^ Alpha ^Red (253) ^Green (72) ^Blue(139)

Need to move “red bits” 11111101 to eight ends, turn off others.
How?

00000000 00000000 00000000 11111101 253 (decimal)

Type conversion –4 scenarios

1. Given an expression with operands of mixed types, C converts (promotes) the types of values to do calculations

```
float f = 3.8;    int i = 3;
f + i;
```

2. May happen on assignment

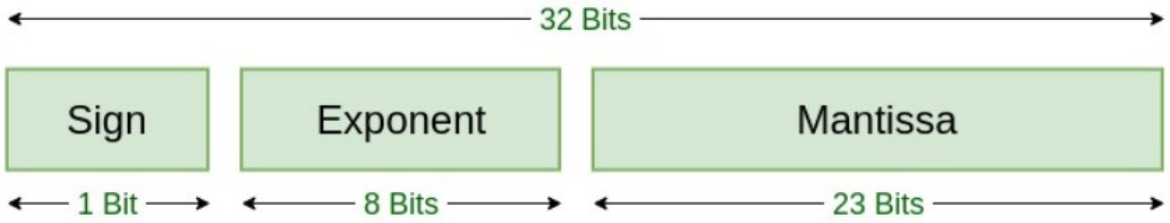
```
float f = 3;      int i = 3.8;
```

3. May happen on function call arguments
4. May happen on function return type

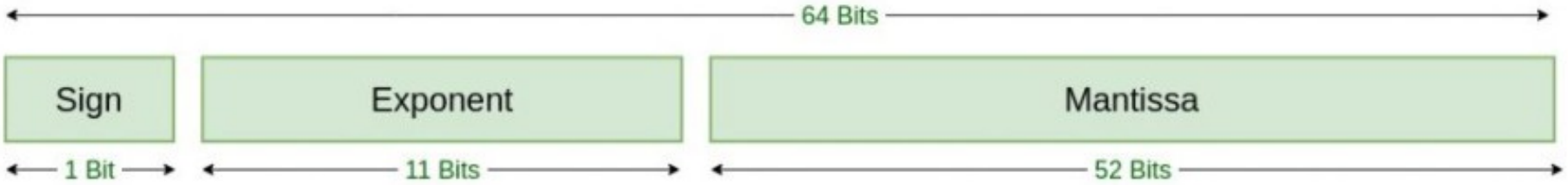
Memory Representation

IEEE Standard 754 for Floating-Point Arithmetic

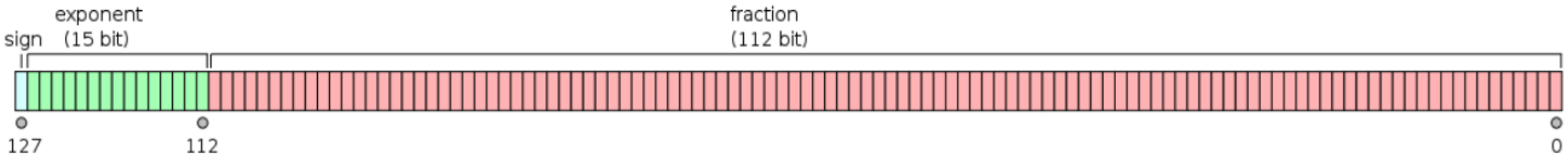
float

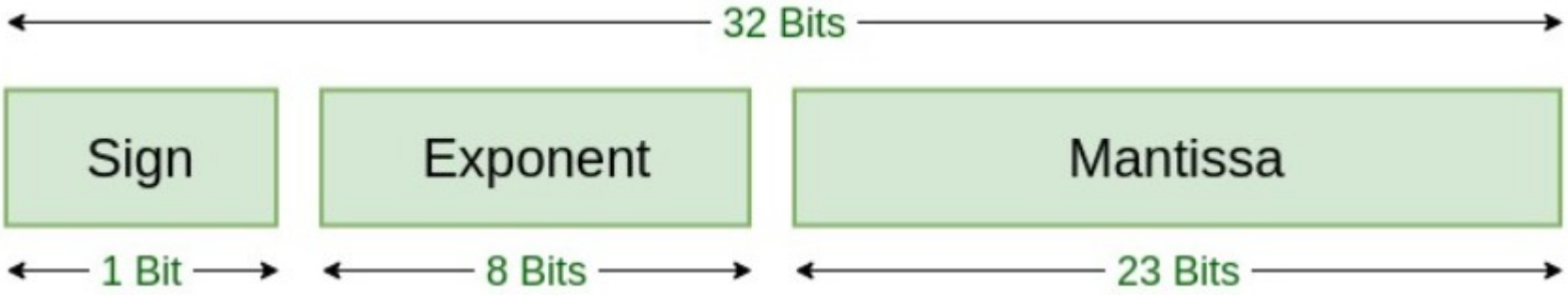


double



long double





65.125 = 1000001.001
= 1.000001001 x 10^6

Normalized Mantissa = 000001001

Biased (+Offset) exponent = 127 + 6
= 10000101

Signed bit is = 0 (positive)

0 10000101 00000100100000000000000

Scenario 1–mixed types in arithmetic

Given an expression with operands of mixed types, C converts (promotes) the types of values to do calculations

- Promotes: converts to a more precise type
- Result is the promoted (more precise) type.

```
int x = 5, y = 2;     x/y = ?
float f = 2.0F;
```

same in Java

for expression **x/f** x is int, f is float

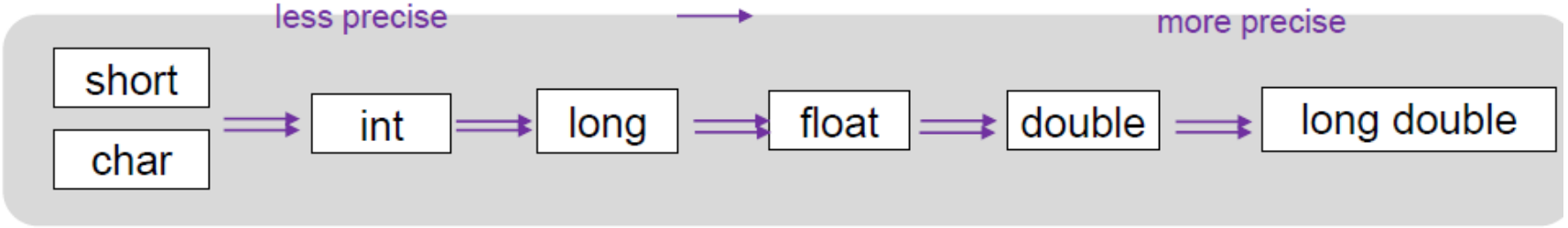
x's value is read, converted to a float and then used in division

(i,e., 5 ⟹ 5.0)

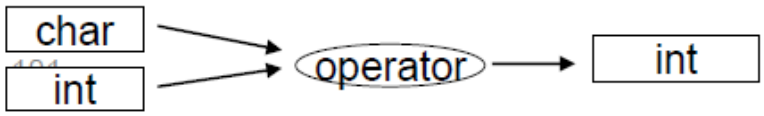
- 5 / 2.0 = 5.0 / 2.0 = 2.5
- return type **float**

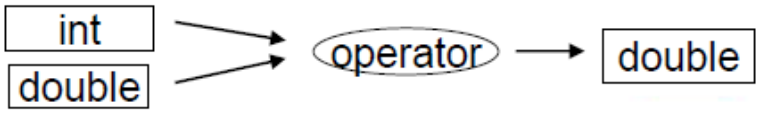
Type Promotion: converts to a more precise type

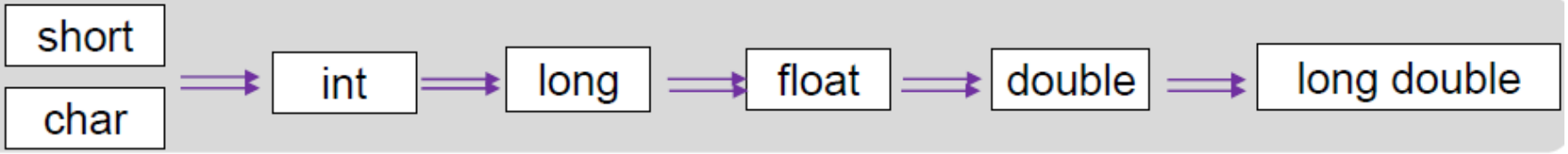
- Informal rules (from K&R p. 44)
  - **if either operand is "long double"**
    - **convert to "long double"**
  - **else if either operand is "double"**
    - **convert to "double"**
  - **else if either operand is "float"**
    - **convert to "float"**
  - **else**
    - **convert char and short to int**
    - **if either operand is long, convert to long**



Examples:







Mixed type arithmetic

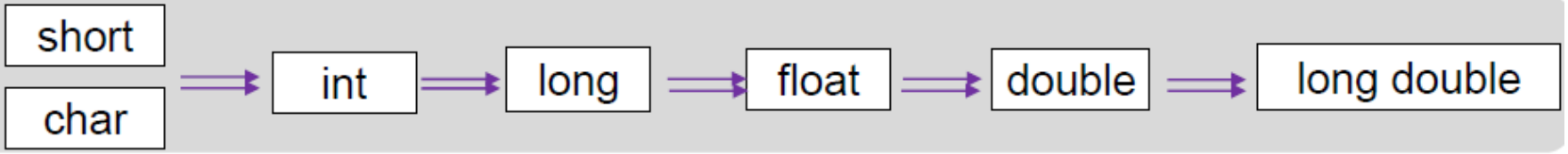
- Given an expression with operands of mixed types, C converts (promotes) the types of values to do calculations

- 17 / 5
  - 3

0 conversion

- 'K' + 32
  - 75 + 32 = 107
  - Return type int

1 conversion

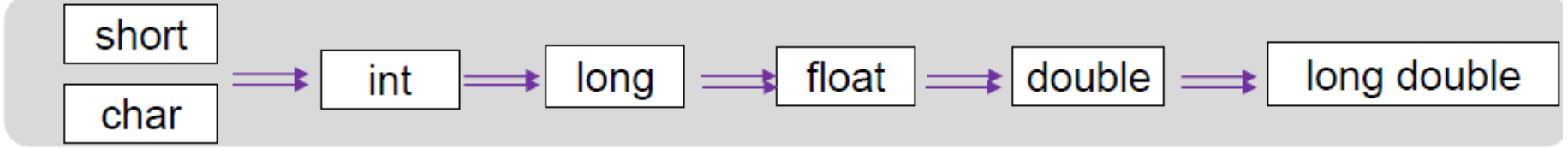


Mixed type arithmetic

- Given an expression with operands of mixed types, C converts (promotes) the types of values to do calculations

- 17 / 5
  - 3 — 0 conversion
- 'K' + 32
  - 75 + 32 = 107
  - Return type int — 1 conversion
- 17.0 / 5
  - 17.0 / 5.0 = 3.4 — 1 conversion
  - Return type double

same in Java



Mixed type arithmetic

- Given an expression with operands of mixed types, C converts (promotes) the types of values to do calculations

- 17 / 5
  - 3 — 0 conversion
- 'K' + 32
  - 75 + 32 = 107
  - Return type int — 1 conversion
- 17.0 / 5
  - 17.0 / 5.0 = 3.4 — 1 conversion
  - Return type double

same in Java

- 9 / 2 * 3.0 / 4
  - 9/2 = 4 type int
  - 4*3.0 = 4.0*3.0 = 12.0 double
  - 12.0/4 = 12.0/4.0=3.0 double

2 conversions

- 3.0 * 9 / 2 /4
  - 3.0*9 = 3.0*9.0=27.0 double
  - 27.0/2.0 = 13.5 double
  - 13.5/4.0 = 3.375 double

3 conversions

Associativity: left to right

Scenario 2: Conversions across assignments

- The value of the right side is converted to the type of the left, which is the type of the result =

```
int i = 512;
float f;
f = i;   /*value of i is converted to float 512.00 */
         /* return type float, return value 512.00 */
```

same in Java

Scenario 2: Conversions across assignments

- The value of the right side is converted to the type of the left, which is the type of the result =

```
int i = 512;
float f;
f = i;   /*value of i is converted to float 512.00 */
         /* return type float, return value 512.00 */
```

same in Java

- If the left side is of smaller range or precision, information may be lost (should avoid)
  - Longer integers converted to shorter ones or chars by dropping the excess high-order bits
  - float/double to int truncates any fractional part.

```
float f = 512.993f;
int i = f;   /* f is converted to int 512 (no rounding) */
```

Not valid in Java

Type Conversion –Examples arithmetic (scenario1) and assignment (scenario2)

```
int x=5, y=2;
double q = 2;          // conversion on assignment q=2.0

int w = x/y;           //  no conversions w=2

double z = x/y;        //  z=2.0 conversion on assignment

double z = x/q;        // z=5.0/2.0=2.5 conversion on /

int w = x/q;
//  conversion on / and then on assignment
//  w = 5.0/2.0 = 2.5 = 2

char x = 'K' + 32:   // conversion on + and then on =
                     // x = 75 + 32 = 107 = 'k'
```

2 conversions

Scenario 3,4 Conversions across function

- arguments
- returns

```c
#include <stdio.h>

/* function declaration */
double sum(double, double);

main()
{
  int x = 4; double y= 3.9;
  double su = sum(x,y); // sum receives 4 → 4.0 and 3.9
  printf("Sum is %f\n", su); // 7.9
}

/* function definition */
double sum (double i, double j){
   return i+j;    // 4.0 + 3.9
}
```

double i = x call-by-value
1 conversion -- on (implicit) assignment

Scenario 3,4 Conversions across function

- arguments
- returns

```
#include <stdio.h>

/* function declaration */
int sum(int, int);

main()
{
   int x = 4; double y= 3.9;
   int su = sum(x,y); // sum receives 4, and 3.9 → 3
   printf( "Sum is %d\n", su); // 7
}

/* function definition */
int sum (int i, int j){
     return i+j;   // 4 + 3
}
```

int j = y call-by-value
1 conversion (on assignment)

Scenario 3,4 Conversions across function

- arguments
- returns

```
type function (){
    return expr;
}
```

- If **expr** is not of type **type**, compiler
  - produces a warning
  - converts **expr** (as if by assignment) to the return **type** of the function (the contract to user)
  - should avoid

```
int function (){
  double x;
  return x;   /* return (int)x if you have to
  }              tell the complier you know
                 what you are doing (losing) */
```

Scenario 3,4 Conversions across function

- arguments
- returns

```c
#include <stdio.h>

/* function declaration */
double aFun();

main()
{
  printf("%f", aFun());  // return type double, value 7.0
}

/* function definition */
double aFun (){
  int i = 3;
  int j = 4;
  return i + j;    /* i+j of type int, converted to double*/
}                  /*  7 → 7.0  */
```

Scenario 3,4 Conversions across function

- arguments
- returns

```
#include <stdio.h>

/* function declaration */
int aFun();

main()
{
  printf("%d", aFun());  // return type int, value 7
}

/* function definition */
int  aFun (){
  double i = 3.6;
  int j = 4;
  return i + j;    /* i+j of type double, converted to int */
}                   /*  7.6 → 7  */
                                    2 conversions
```

Explicit Conversion (Type Casting)

- We can also explicitly change type
- Type cast operator; **`(type-name)operand`**

```
int a = 9, b = 2;
float f;

f = a / b;              /* f is 4.0 */
f = a /(float) b;       /* f is 4.5 */
f = (float)a/b;         /* f is 4.5 */

f = (float)(a/b); ? /* f is 4.0 */

int d = (int)f;
```

Doesn't change the value of b, Just changes the type to float

Another way:

```
1.0 * a / b
a * 1.0 / b
```