Outline

- Types and sizes
  - Types
  - Constant values (literals)
    - char
    - int
    - float
- Array and “strings” (Ch1.6,1.9)
- Expressions
  - Basic operators (arithmetic, relational and logical)
  - **Type promotion and conversion**
  - Other operators (bitwise, bit shifting , compound assignment, conditional)
  - Precedence of operators

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

Type conversion – scenario 1

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

- What is the type of expression **`x/y`** or **`y/x`**
- What is the result of expression **`x/y`** or **`y/x`**

- What is the type of **`x/f`** or **`f/x`**?
- What is the result of **`x/f`** or **`f/x`**?

94

94

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`**

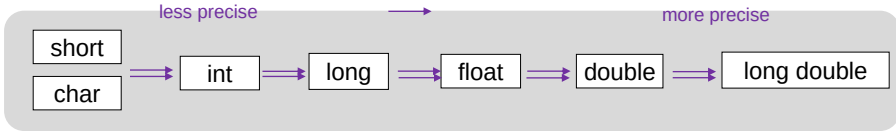
95

95

Type Promotion

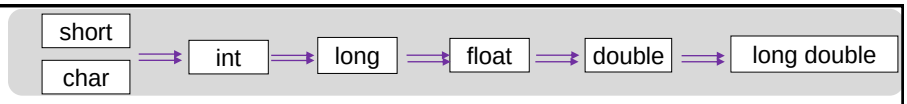
converts to a more precise type

- 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:

96



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

97

97

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 512.000 */
        /* return type float, return value 512.000 */
```

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

```
java:10: error: incompatible types: possible lossy conversion from float to int
int i =f;
      ^
```

```
char x = 68;
```

OK in Java

98

98

Type Conversion - Examples

arithmetic (scenario1) and assignment (scenario2)

same in Java

```
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=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

2 conversions

OK in Java

99

99

Scenario 3,4 Conversions across function

- arguments
- returns

```
#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

100

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)

101

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

```
#include <stdio.h>

/* function declaration */
double aFun();

main()
{
  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()
{
  aFun();  // return type double, value 7.0
}

/* 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
```

104

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

Needed in Java

Another way:

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

a / b * 1.0 ?
```

4.0

105

Outline

- Types and sizes
  - Types
  - Constant values (literals)
    - char
    - int
    - float
- Array and “strings” (Ch1.6,1.9)
- **Expressions**
  - Basic operators (arithmetic, relational and logical)
  - Type promotion and conversion
  - **Other operators (bitwise, bit shifting, compound assignment, conditional)**
  - Precedence of operators

106

106

Bitwise operators

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

- bitwise **`& | ^ ~`**

01100101	01101100	01101100	01101111	00000000
01001000	01100101	01101100	01101100	01101111

And

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

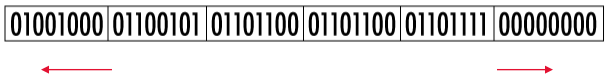
Or

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 **`<< >>`**



107

107

Bitwise Operators

Or

p	q	p ∨ q
T	T	T
T	F	T
F	T	T
F	F	F

Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	1	1	1

| - bitwise "**or**"

- Calculates the "or" of all bits in both operands
- 0 | 0 ➔ 0 0 | 1 ➔ 1 1 | 0 ➔ 1 1 | 1 ➔ 1
  - either bit is 1, result is 1 (on). Both bits must be 0 to result in 0

0: keep whatever other,

- e.g.

```
int z = 145 | 41
```

```
145= 000…10010001
41 = 000…00101001
```

z = x | y does not change x and y

same in Java

Bitwise Operators

And

p	q	p · q
T	T	T
T	F	F
F	T	F
F	F	F

Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	0	0	1

& - bitwise "**and**"

- Calculates 'and' of all bits in both operands
- 0 & 0 ➔ 0 0 & 1 ➔ 0 1 & 0 ➔ 0 1 & 1 ➔ 1
  - either bit is 0, result is 0 (off). Both bits must be 1 to result in 1

- e.g.

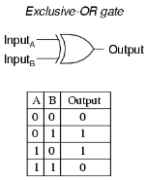
```
int z = 145 & 41
```

```
145= 000…10010001
41 = 000…00101001
```

z = x & y does not change x and y

same in Java

Bitwise Operators



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	1	1	0

^ - "**xor**" ("exclusive-or")

- If two bits are different, the result is 1; otherwise 0.
  - like "or" except when both bits are 1, the result is 0

- e.g.

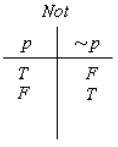
```
int z = 145 ^ 41
```

```
145= 000…10010001
41 = 000…00101001
```

z = x ^ y does not change x and y

same in Java

Bitwise Operators



Rhs	0	1
Result	1	0

~ one's complement (bit inversion)

- flips all bits in its operand

- e.g.

```
int z= ~145
```

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

z= ~x does not change x

same in Java

Bit Shifting

01001000	01100101	01101100	01101100	01101111	00000000

- 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`**)

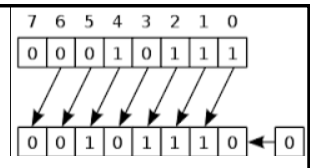
What goes Out? bits pushed "off the end" on the end

What comes in? >> << different

113

113

Bit Shifting <<

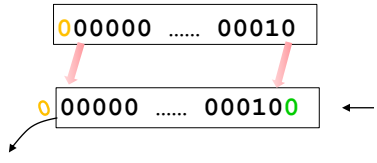


- Suppose **`z`** is an int
  - e.g.

    `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

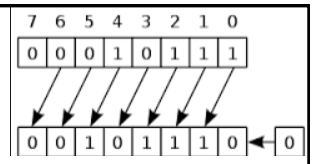
114

z = x << 3 does not change x

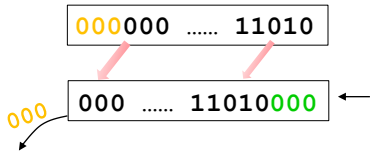
114

Bit Shifting <<

Another example



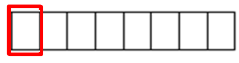
- Suppose `z` is an int
  - e.g.
    - `int z= 26 << 3`
    - shift left 3 bits
    - `z: 208`
      `0320`
      `0XD0`



What goes Out? bits pushed "off the end" on the left end
What comes in? we add 0 on the right

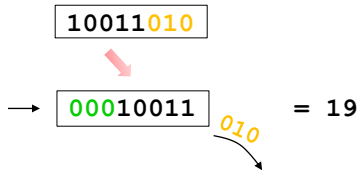
z = x << 3 does not change x

Bit Shifting >>



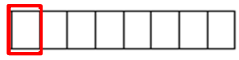
- What if we shift right? >> complicated.
- For "unsigned" types – all bits are magnitude -- add 0 on left
- e.g. (assume these are all unsigned)

```
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)

117

```
printf("%d", -1 >> 3);
```

YORK UNIVERSITÉ UNIVERSITY

117

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…

118

```
System.out.printf("%d\n", -1 >> 3); // -1
System.out.printf("%d\n", -1 >>> 3); // 536870911
```

YORK UNIVERSITÉ UNIVERSITY

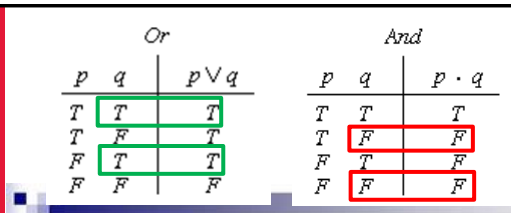
118

But What Is It Useful?

- A common use: flags, masks
  - A flag is a Boolean value (off=0, on=1) which describes a state, e.g., switches
  - We could use an "**`int`**" to describe a flag, but an **`int`** has a minimum of 16 bits (65536 values) - far more than we need
  - We can use bitwise operators to efficiently represent flags - each bit can be a flag
    - so one **`int`** can represent at least 16 flags.

00000100 01011000 00001100 11101111

One int – 16 or 32 'Boolean' flags



Masking for bit manipulation

- Masking uses AND and OR operators
- Use OR ('|') to set bits to '1' with 1
- Use AND ('&') to set bits to 0 with 0

- | 1: turn on (set 1)
- & 0: turn off (set 0)
- | 0: keep value
- & 1: keep value

OR		
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

AND		
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Flags (some idioms)

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

- `int flags;`
  - `flags = flags & (1<<5)`
    - 0..00100000.

    ?..????????
    0..00100000 &
    0..00?00000
  - `flags = flags | (1<<5)`
    - 0..00100000.

    ?..????????
    0..00100000 |
    ?..??1?????
  - `flags = flags & ~(1<< 5)`
    - 11..11011111.

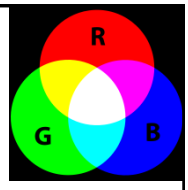
    ?..????????
    1..11011111 &
    ?..??0?????
  - `flags = flags & 0177`
    - 00.. 001 111 111

    ?..?????????
    0..001111111 &
    0..00???????
  - `flags = flags & ~077`
    - 00..000111111->11..11000000.

    ?..?????????
    1..11000000 &
    ? ..??000000

Practice in the lab. Revisit next time

Some examples



- In Java, getRGB() packs 3 +1 values (0~255) into a 32 bit (4 bytes) int

- 10101100 11111101 01001000 10001011

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

java.awt.image
Class BufferedImage
java.lang.Object
java.awt.Image
java.awt.image.BufferedImage

java.awt
Class Color
java.lang.Object
java.awt.Color

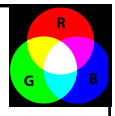
getRGB

public int getRGB (int x, int y)

Returns the RGB value representing the color in the default sRGB `ColorModel`. (Bits 24-31 are alpha, 16-23 are red, 8-15 are green, 0-7 are blue).

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)

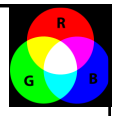
Need to keep lower 8 bits, turn off others.
How?

00000000 00000000 00000000 10001011 **139 (decimal)**

123

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)

&

00000000 00000000 00000000 11111111 **255 0377 0xFF**

Turn off keep value

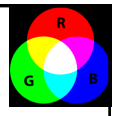
00000000 00000000 00000000 10001011 **139 (decimal)**

```
int blue = rgb  & 255          /* rgb not changed */
         = rgb  & 0377
         = rgb  & 0XFF
                & 0b11111111   (Java)
```

124

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 *red* value?

10101100 11111101 01001000 10101011

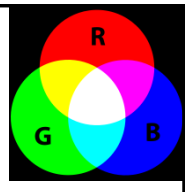
^ Alpha ^Red (253) ^Green (72) ^Blue(139)

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

00000000 00000000 00000000 11111101 **253 (decimal)**

125

Some examples



- 10101100 11111101 01001000 11111111

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

How to get *red* value?

- First shift "red bits" to the right end (how?)

00000000 00000000 10101100 11111101 **rgb >> 16**

or

11111111 11111111 10101100 11111101 **if signed (maybe)**

What's next

00000000 00000000 00000000 11111101

126

Summary and plan

- (Primitive) Types and sizes
  - Types: char, short, int, long, unsigned short, unsigned int, float, double …..
  - Constant values (literals)
    - char
    - int
    - float
- Array and "strings"
- Expressions
  - Basic operators ++ -- && ||
  - Type promotion and conversion
  - Other operators (bitwise, bitshift, compound assignment)
  - Precedence of operators

today

127

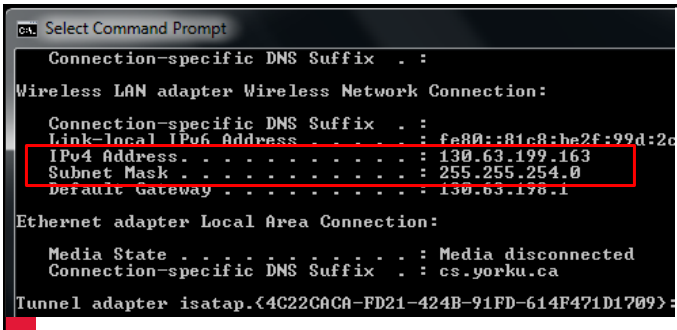
Outline

- Types and sizes
  - Types
  - Constant values (literals)
    - char
    - int
    - float
- Array and “strings” (Ch1.6,1.9)
- **Expressions**
  - Basic operators (arithmetic, relational and logical)
  - Type promotion and conversion
  - **Other operators (bitwise, bit shifting, compound assignment, conditional)**
  - Precedence of operators

128

- Below from ON L4 until end of ch2, in case have time left

129



Example: ip address 192.168.18.55, subnet mask: 255.255.255.0

11000000 10101000 00010010 00110111

11111111 11111111 11111111 00000000

Address: 11000000 10101000 00010010 00110111
Subnet Mask: 11111111 11111111 11111111 00000000
AND -------- -------- -------- --------
Network ID: 11000000 10101000 00010010 00000000

NET_ID is 192.168.18

For your information

130

Somethings to Think About

- | looks similar to || Both do “OR”
- & looks similar to && Both do “AND”

- Can you substitute | for ||?
- Can you substitute & for &&?

- | and & applies to bits, || and && apply to whole values

```
int x=1, y=2;
x && y   ?    1
x &  y   ?    0

x || y   ?    1
X |  y   ?    3
```

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

131

- ~ vs ! Both do “Negation” `~145` `!145`

Expressions

- Some of the common operators:
  - `+, -, *, /, %, ++,--` (basic arithmetic)
  - `<, >, <=, >=` (relational operators)
  - `==, !=` (equality operators)
  - `&&, ||, !` (logical operators)
  - `= += -=` (assignment & compound assignment)

- Others:
  - bitwise `& | ~`, bit shifting `<< >>`,
  - `sizeof`
  - conditional ? :
  - compound assignment

132

Expressions

- sizeof

```
sizeof (int)

int a;
sizeof a;   or  sizeof (a);
```

- Not a function
- Don't use **sizeof** on function array parameter

```
main(){
 char s[] = "Hello";
 printf("%d", sizeof s); // ?
 int a = indexOf(s, 'a');
}

int indexOf (char arr[], char c ) {
    for(i=0; i < sizeof arr; i++)
```

sizeof arr
always 4 or 8
Explain later

```
lab5E.c:66:28: warning: 'sizeof (arr)' will return the size of the pointer, not the array itself
      [-Wsizeof-pointer-div]
    int size = sizeof(arr)/sizeof(int);
```

Some nice compiler (MAC. not lab ☹)

133