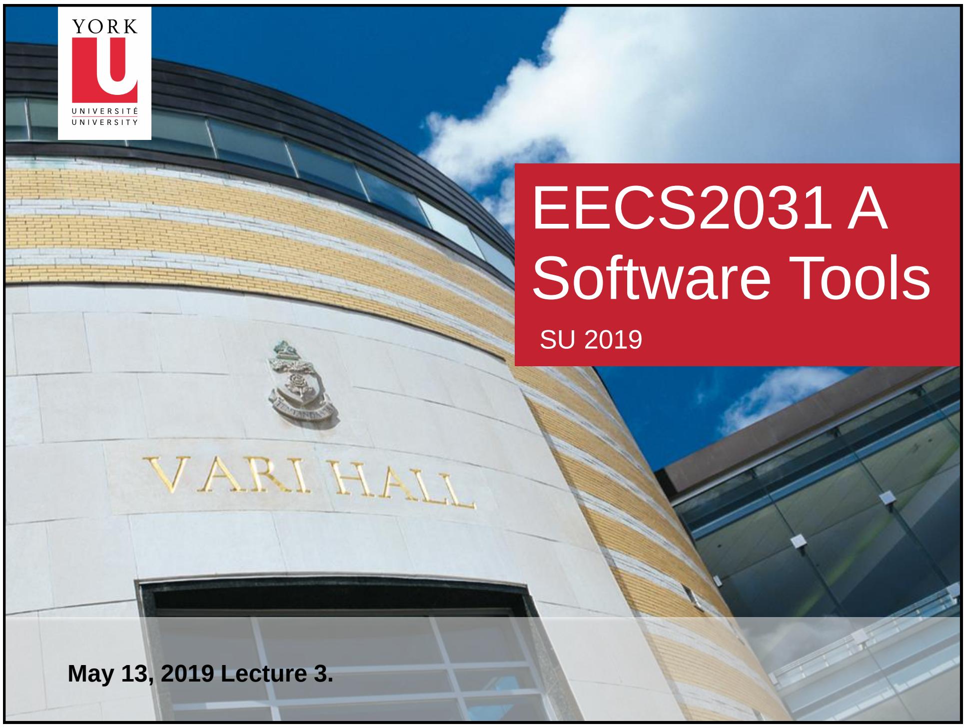


1

Statements

- Program to execute
  - Ended with a ;
- Expression statement (ch2)
  - `i+1;   i++;   x = 4;`
- Function call statement (ch4)
  - `printf("the result is %d");`
- Control flow statement (ch3)
  - `if else, for(), while,  do while, switch`

Same in Java

2

2

Summary and future work

- Type, operators and expressions (Chapter 2) :
  - Types and sizes
    - Basic types, their sizes and constant values (literals)
      - ✓ char: x > 'a' && x < 'z'; x > '0' && x < '9'
      - ✓ int: 122, 0122, 0x12F convert between Decimal, Bin, Oct, Hex
    - Arrays (one dimension) and strings (Ch1.6,1.9)
      - ✓ "hello" has size 6 byte | H | e | l | l | o | \0 |
  - Expressions
    - Basic operators (arithmetic, relational and logical)
      - ✓ y=x++; y=++x;
      - ✓ if (x = 2)
    - Type conversion and promotion
    - Other operators (bitwise, bit shifting , compound assignment, conditional)
      - ✓ Bit: |, &, ~. ^, << >>
      - ✓ Compound: x += 10; x >>= 10; x += y + 3
    - Precedence of operators
- Functions and Program Structure (Chapter 4)

Last week

today

3

3

Expression

- Formed by combining **operands** (variable, constants and function calls) using **operators** (`+ - * % > < == !=`)
- Has return values -- always
  - `x+1`
  - `i < 20` false: 0 true: 1 `printf("%d", i<20);`
  - `sum (i+j)`
  - `x = 5` `= is an operator in C (and Java)! Return value 5`
  - `x = k + sum(i,j)` `printf("%d", x=5);`

"whenever a value is needed, any expression of the same type will do"

- `printf("sum is %d\n", i*y+2)`
- `printf("sum is %d\n", sum(i+j))`

YORK U UNIVERSITÉ UNIVERSITY

4

Text book:
4 basic types: char, int, float, double

3 qualifiers: short, long, unsigned

C Types and sizes

- Variables and values have types
- There are two basic types in ANSI-C: integer, and floating point

  - **Integer type**
    - **char** - **character**, single byte (8 bits)
    - **short (int)** - **short integer**, 1 or 2 bytes (8 or 16 bits)
    - **int** - **integer**, usually 2 or 4 bytes (16 or 32 bits)
    - **long (int)** - **long** integer, usually 4 or 8 bytes (32 or 64 bits)
  - **Floating point**
    - **float** - single-precision, usually 4 bytes (32 bits)
    - **double** - double-precision, usually 8 bytes (64 bits)
    - **long double** - extended-precision

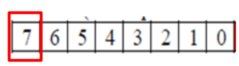
YORK UNIVERSITÉ UNIVERSITY

5

5

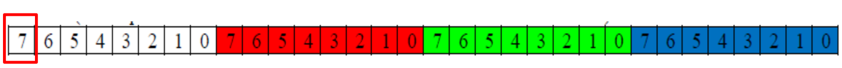
Qualifiers (modifiers) for integer type

- signed, unsigned qualifiers can be applied to integer types
  - Signed: default. Left most bit signifies sign 0: positive 1: negative
  - Unsigned: positive. Left most bit contributes to magnitude




```
▪ (signed) char
▪ (signed) int
▪ (signed) short int
▪ (signed) long  int

▪ unsigned char
▪ unsigned int
▪ unsigned short int
▪ unsigned long  int
```

Java: no direct support for unsigned int. Always signed



`(signed) int`	$-2^{31} \sim 2^{31}-1$	-2145483648~ 2147483647	2^{32} values
`unsigned int`	$0 \sim 2^{32}-1$	0~ 4294967295	2^{32} values

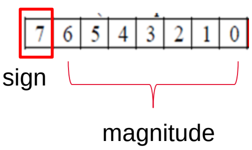
6

Max: signed 0111111….11111 Unsigned: 1111111….11111

6

Qualifiers (modifiers) for integer type

- **signed/unsigned** can be applied to char
  - **signed** char $-2^7 \sim 2^7-1$ /* -128 ~~ 127 */
  - **unsigned** char $0 \sim 2^8-1$ /* 0 ~~ 255 */



Unsigned potentially save bits
E.g., Count # student in our class (about 150)
- If declared `signed short`, max 127, 8 bits not enough
- If declared `unsigned`, then 8 bits are enough.
  `unsigned short counter;`

signed value

Bits	Unsigned value	2's complement value
00000000	0	0
00000001	1	1
00000010	2	2
01111110	126	126
01111111	127	127
10000000	128	−128
10000001	129	−127
10000010	130	−126
11111110	254	−2
11111111	255	−1
	$0 \sim\sim 2^n-1$	$-2^{n-1} \sim\sim 2^{n-1}-1$
	2^n=256 values	2^n=256 values

7

7

Summary

- Interger types:
  - `signed char`   `unsigned char`
  - `(signed) short`   `unsigned short`
  - `(signed) int`   `unsigned int`
  - `(signed) long`   `unsigned long`
- There are three types of floating points:
  - `float        /* single-precision */`
  - `double       /* double precision */`
  - `long double  /* extended-precision */`
- C99 added：
  - `(signed) long long int`
  - `unsigned long long intbo`

Java defines

Type
int
short
long
byte
float
double
char
boolean

YORK U UNIVERSITÉ UNIVERSITY

8

8

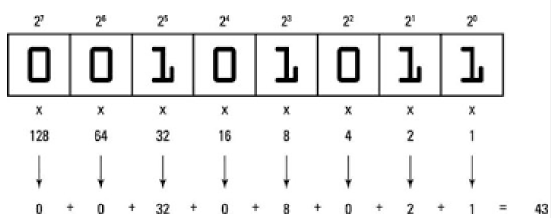
Outline

- Types and sizes
  - Types
  - **Constant values (literals)**
    - **char**
    - int
    - float
- Array and “strings”
- Expressions
  - Basic operators
  - Type promotion and conversion
  - Other operators
  - Precedence of operators

9

9

Internal representation of characters




```
int i =  43;

char a = 'A';
```

How to represent 'A' using 0s and 1s

10

10

01100101 01101100 01101100 01101111 00000000

Internal Representation of Characters

Dec	Hx	Oct	Char
0	0	000	NUL (null)

- '0' - '9' are encoded consecutively (48~57)
- 'A' - 'Z' are encoded consecutively (65~90)
- 'a' - 'z' are encoded consecutively (97~122)
- Upper letters before lower. Index/encoding difference of 'a' and 'A' is 32, so does 'b' and 'B', 'c' and 'C', …
- 8 bits is enough
- Java uses a bigger character set table Unicode, 0~127 are same

Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
32	20	040		Space	64	40	100	@	@	96	60	140	`	`
33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

11

Characters

- **chars** are treated in C as small integers, **char** variables and constants are identical to **int** in arithmetic expressions:
  - **char c** is converted to its encoding (index in the character set table)

46	2E	056	.	.
47	2F	057	/	/
48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9
58	3A	072	:	:
59	3B	073	;	;
60	3C	074	<	<
61	3D	075	=	=
62	3E	076	>	>

```
char aChar = '5';    // encoding 53
aChar + 12        // expression with value 53+12 = 65
```

same in Java

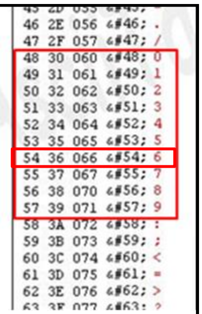
- Same for other expressions. In relational expression, characters can be compared directly, comparing their indexes/encodings

```
aChar == 'H'     // index == 72?  ➔ expr with value 0 (false)

aChar == '/n'   // index = 10?  ➔ exp with value 0 (false)

'5' < 'H'   // 53 < 72?  Earlier in table? ➔ expr with 1 (true)
```

12

12

Characters



46	2E	056	.	.
47	2F	057	/	/
48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9
58	3A	072	:	:
59	3B	073	;	;
60	3C	074	<	<
61	3D	075	=	=
62	3E	076	>	>

- Since **chars** are just small integers, **char** variables and constants are identical to **int** in arithmetic expressions:
  - **char c** is converted to its encoding (index in the character set table)

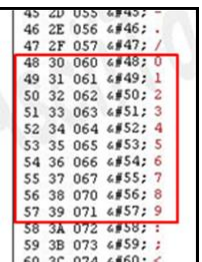
```
char aCh = '6';  // same as   char aCh = 54;
printf("value is %c\n", aCh ); // char 6
printf("value is %d\n", aCh ); // numerical 54
                               // print encoding

printf("value is %d\n", aCh + 2 ); //numerical 56
printf("value is %c\n", aCh + 2 ); // char 8

printf("value is %d\n", aCh-'0' ); // 54-48
                                   //numerical 6
```

13

same in Java

13

Characters



46	2E	056	.	.
47	2F	057	/	/
48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9
58	3A	072	:	:
59	3B	073	;	;
60	3C	074	<	<

- Since **chars** are just small integers, **char** variables and constants are identical to **int**s in arithmetic expressions: take advantage of this

```
if(c >= '0' && c <= '9') /*index 48~57,is a digit */
                          Located after '0' and before '9'

if(c >='a' && c <= 'z') /* low case letter*/

if(c >='A' && c <= 'Z') /*upper case letter*/


if(c >='0' && c <= '9')  // c<= 48 c>=57  isdigit(c)
  printf("c is a digit\n");
  printf("numerical value is %d\n", c-'0');
```

14

same in Java

14

Example

- Upper case letters before lower case letters.
- Encoding difference of 'a' and 'A' is 32, so does 'b' and 'B', 'c' and 'C', 'd' and 'D'…

64	40	100	@	@	96	60	140	`	`
65	41	101	A	A	97	61	141	a	a
66	42	102	B	B	98	62	142	b	b
67	43	103	C	C	99	63	143	c	c
68	44	104	D	D	100	64	144	d	d
69	45	105	E	E	101	65	145	e	e
70	46	106	F	F	102	66	146	f	f
71	47	107	G	G	103	67	147	g	g
72	48	110	H	H	104	68	150	h	h
73	49	111	I	I	105	69	151	i	i
74	4A	112	J	J	106	6A	152	j	j
75	4B	113	K	K	107	6B	153	k	k
76	4C	114	L	L	108	6C	154	l	l
77	4D	115	M	M	109	6D	155	m	m
78	4E	116	N	N	110	6E	156	n	n
79	4F	117	O	O	111	6F	157	o	o
80	50	120	P	P	112	70	160	p	p
81	51	121	Q	Q	113	71	161	q	q
82	52	122	R	R	114	72	162	r	r
83	53	123	S	S	115	73	163	s	s
84	54	124	T	T	116	74	164	t	t
85	55	125	U	U	117	75	165	u	u
86	56	126	V	V	118	76	166	v	v
87	57	127	W	W	119	77	167	w	w
88	58	130	X	X	120	78	170	x	x
89	59	131	Y	Y	121	79	171	y	y
90	5A	132	Z	Z	122	7A	172	z	z
91	5B	133	[	[	123	7B	173	{	{
92	5C	134	\	\	124	7C	174			|

```
#include<stdio.h>

/*copying input to output with
converting upper-case to lower-case letters */
main(){
    int c; int lowC;
    c= getchar();
    while (c != EOF)
    {
        if (c >= 'A' && c <= 'Z') /* 65~90 upper case letter*/
            lowC = c + 'a'- 'A';   /* c + 'b' - 'B'   */
                                   /* c + 'c' - 'C'   */
        putchar(lowC);             /* c = tolower(c)   */

        c = getchar(); // read again
    }
    return 0;
}
```

`c + 32` works in the lab but not good for portability. Avoid that!

15

15

Outline

- Types and sizes
  - Types
  - **Constant values (literals)**
    - **char** treated as small int
    - **int** different bases
    - float
- Array and "strings"
- Expressions
  - Basic operators
  - Type promotion and conversion
  - Other operators
  - Precedence of operators

16

16

Integer Constants

0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

Stored always binary 2^2 2^1 2^0

- Integer constants can be expressed in three different ways:
  1. **Decimal** [base 10]
     - `int x = 31`

     same in Java
  2. **Octal** [base 8]
     - Start with zero **0**
     - `int x = 037` (31 in decimal)

     same in Java
  3. **Hexadecimal** [base 16]
     - Start with **0x** or **0X**
     - `int x = 0x1F` (31 in decimal)

     same in Java

Ways for people to write numbers. No effect on how the numbers are stored –- always binary.

Java also has the 4th: binary
`int x = 0b00011111`

17

Others To decimal

- 3 2 8 (10^2 10^1 10^0) ➡ $3*10^2 + 2*10^1 + 8*10^0 =$ 300 + 20 + 8 = 328 — Decimal 328
- 1 0 1 (2^2 2^1 2^0) ➡ $1*2^2 + 0*2^1 + 1*2^0 =$ 4 + 0 + 1 = 5 — 00000000000101 Binary
- 3 4 5 (8^2 8^1 8^0) ➡ $3*8^2 + 4*8^1 + 5*8^0 =$ 192 + 32 + 5 = 229 — Octal 0345
- 3 4 F (16^2 16^1 16^0) ➡ $3*16^2 + 4*16^1 + F*16^0 =$ 3*256 + 4*16 + 15 *1 = 768 + 64 + 15 = 847 — Hex 0x34F 0X34f

You should know these conversions.

YORK UNIVERSITÉ UNIVERSITY

18

Binary to others -- why Hex and Oct

I know I want an int with representation 01001100, how to code it in C?

In Java, can do binary `int a = 0b01001100`

- 0 1 0 0 1 1 0 0
  2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
  → $1*2^6 + 1*2^3 + 1*2^2 =$ 64 + 8 + 4
  Decimal `int a = 76`

- $\underbrace{0\ 1}_{1}\ \underbrace{0\ 0\ 1}_{1}\ \underbrace{1\ 0\ 0}_{4}$ ⇄ `int a = 0114` Octal

- $\underbrace{0\ 1\ 0\ 0}_{4}\ \underbrace{1\ 1\ 0\ 0}_{C}$ ⇄ `int a = 0X4C` `= 0x4c` Hex

19 You should know these conversions (both ways).

YORK UNIVERSITÉ UNIVERSITY

19

Outline

- Types and sizes
  - Types
  - Constant values (literals)
    - char
    - int
    - float
- **Array and "strings" (Ch1.6,1.9)**
- Expressions
  - Basic operators
  - Type promotion and conversion
  - Other operators
  - Precedence of operators

20

YORK UNIVERSITÉ UNIVERSITY

20

Declaring Arrays

- Declare and initialize (how to do in Java?)

```
int k[5];  /* each element get some garble value*/
           -5  122 45623 85 58

int k[5] = {1,5,3,2,25};     1 5 3 2 25

int k[5] = {1,5};            1 5 0 0 0

int k[] = {1,5,3,2,25};      1 5 3 2 25
```

```
sizeof k?   20     // assuming 4 bytes int

sizeof(k)/sizeof(k[0]) = 20/4 = 5
```

21

21

An example involving array and chars

What does this program do?

```
/*counting digits*/
#include <stdio.h>
#define N 10
int main () {
   int c, i;
   int digit[N];

   for (i=0; i< N; i++)
     digit[i]=0;

   c = getchar();
   while (c != EOF)
   {
     if ( c>= '0' && c <= '9' ){
         int pos = c - '0';
         digit[pos] ++;     // digit[c] ++
     }
     c = getchar(); // read again
   }

   for (i=0; i< N; i++)
      printf ("%d: %d ", i, digit[i]);
}
```

45	2D	055	-	-
46	2E	056	.	.
47	2F	057	/	/
48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9
58	3A	072	:	:
59	3B	073	;	;
60	3C	074	<	<

22

22

Strings ⟷ Character Arrays !

```
Dec Hx Oct Char
0   0 000 NUL (null)
1   1 001 SOH (start of heading)
2   2 002 STX (start of text)
```

- There is no separate “string” type in C
- Strings are just arrays of char that end with `'\0'`

```
char s[] = "Hello";
char s[6] = "Hello";
```

'H'	'e'	'l'	'l'	'o'	'\0'
01001000	01100101	01101100	01101100	01101111	00000000

\0 added for you

- What's the size of s in memory? `sizeof (s)?` `6×1 bytes`
  - `char s[5]= "Hello";`
  - `char s[8]= "Hello";` `sizeof s?` `8×1 bytes`

'H'	'e'	'l'	'l'	'o'	'\0'	'\0'	'\0'

- What is the length of s?
  `strlen(s) = 5` later

23

23

Accessing Arrays/Strings

- In C, you can only assign to array members
  - This means you cannot assign to an array:

```
int  i, k[4], j[4];
for (i=0; i<4; i++)
   j[i]= 0;      /* another way? int j[4]={0}   */

k = j;  /* invalid *//* perfectly valid in Java */
```

- **Also cannot compare directly**

```
if (k == j)        /* invalid */
if (k == "quit")   /* invalid, as in Java */

if (c == 'Q')  /* valid, comparing encodings */
while (arr[i] != '\0') /* valid */
```

24

24

An example involving reading char arrays

```
#include<stdio.h>
int length (char []);

main() {
   char my_strg[100];
   int a;

   printf("Enter a word and a int by blank>");
   scanf("%s %d", my_strg,  &a);
   printf("%d", length(my_strg));
}

int length(char arr[]){
    int i = 0;
    while (arr[i] != '\0')
      i++;
    return i;
}
```

No need to give size

No & needed!
A big topic. Talk about that later

No [ ].
Same in Java

No need to give size

25

25

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

26

26

Expressions

- Expressions are made up of *operands* (things we operate upon) and *operators* (things that do the operations: `+ - * % > <`)
  - `x+y/2, i>=0, x==y, i++,…`
- Operands can be constants, variables, array elements, function calls and other expressions
- Every expression has a return value.
  - `x+2` has return value 3 if `x` was 1
  - `i < 20` has return value true or false -- 1 or 0

- In C/Java, `=` is an operator, so assignment is also an expression
  - `variable = expression`
  - `x = 2+3` has return value 5 `printf("%d", x=2+3) // 5`
  - Assignment expression can be an operand in other expressions
    - `y = x = 2;`
    - `while ((c=getchar())!= EOF )`

"whenever a value is needed, any expression of the same type will do" `printf("sum is %d\n", i*y+2);`

27

27

Expressions

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

28

28

Arithmetic (unary) Increment/Decrement Operators

- **++** increment
- **--** decrement

same in Java

- May come before (prefix) or after the operand (postfix)

`++x`	increment x, result of expression is new value	(pre-increment)
`x++`	increment x, result of expression is old value	(post-increment)
`--x`	decrement x, result of expression is new value	(pre-decrement)
`x--`	decrement x, result of expression is old value	(post-decrement)

```
while (x < 10){
  ......
  x++;  // increment later,
        //   before next statement
  ......
}
```

```
while (x < 10){
  ......
  ++x;  // increment immediately
  ......
}
```

Same effects

29

29

Arithmetic (unary) Increment/Decrement Operators

- **++** increment
- **--** decrement

same in Java

- May come before (prefix) or after the operand (postfix)

`++x`	increment x, result of expression is new value	(pre-increment)
`x++`	increment x, result of expression is old value	(post-increment)
`--x`	decrement x, result of expression is new value	(pre-decrement)
`x--`	decrement x, result of expression is old value	(post-decrement)

```
x = 2;
y = x++;  // increment after
          //   assignment
printf("%d %d",x, y);
```

~~x:2 y:3~~ x:3 y:2

```
x = 2;
y = ++x;  // increment before
          //   assignment
printf("%d %d",x, y);
```

x: 3 y:3

30

30

A common use

```
/*initialize to 0 */

#include <stdio.h>
#define N 10

int main () {

   int i=0;
   int digit[N];

   while (i< N)
   {
     digit[i]=0;
     i++;
   }
```

```
// succinct code
while ( i< N)
{
  digit[i++]=0;

}
```

same in Java

31

31

A common use

```
/*copy 4 elements from pos 10 of arrB to arrA */

#include <stdio.h>
#define N 10
int main () {
   int i,j;
   …..

   i=0; j=10;
   while (i<4 && j<14…)
   {
     arrA[i] = arrB[j];
     i++;
     j++;
   }
```

```
// succinct code
while (i<4 && j<14…)
{

  arrA[i++] = arrB[j++];

}
```

same in Java

Stopped here last time

32

32

Summary and future work

- Type, operators and expressions (Chapter 2) :
  - Types and sizes
    - Basic types, their size and constant values (literals)
      - ✓ char: x > 'a' && x < 'z'; x > '0' && x < '9'
      - ✓ int: 122, 0122, 0x12F convert between Decimal, Bin, Oct, Hex
    - Arrays (one dimension) and strings (Ch1.6,1.9)
      - ✓ "hello" has size 6 byte

H	e	l	l	o	\0

  - Expressions
    - Basic operators (arithmetic, relational and logical)
      - ✓ y=x++; y=++x;
      - ✓ if (x = 2)
    - Type conversion and promotion
    - Other operators (bitwise, bit shifting , compound assignment, conditional)
      - ✓ Bit: |, &, ~. ^, << >>
      - ✓ Compound: x += 10; x >>= 10; x += y + 3
    - Precedence of operators
- Functions and Program Structure (Chapter 4)

Last week

today

33

33

Expressions

- Some of the common operators:
  - `+, -, *, /, %, ++,--` (basic arithmetic)
  - `<, >, <=, >=` (relational operators)
  - `==, !=` (equality operators)
  - `&&, ||, !` (logical operators)
  - `= += -=` (assignment & compound assignment)
- Others: bitwise `& | ~`, bit shifting `<< >>`, conditional ? :
  `sizeof`

34

34

Relational and logical Operators

`<, >, <=, >= == !=` (relational and equality operators)
`&&, ||, !` (logical operators)

- Value of a relational or logical expression is `Boolean`
  - 0 when *false*
  - 1 when *true*

In C,
0 means *false*
non-zero means *true*

```
int x = 3;
x > 4      0        printf("%d", x<4);
x == 3     1
x != 4     1

if (x == 5)    not true
while (1)      if (5)
if (x = 5)       ?
```

35

Relational and logical Operators

- Not as safe as Java -- probably why Java introduce Boolean

```
int x = 2;
if (x = 1)
  ……
else if (x=2)…
  ……
```

```
int x = 2;
while(x = 3)
  ……
```

```
indigo 311 % javac Hello.java
Hello.java:13: incompatible types
found    : int
required: boolean
              if (x = 1){
                    ^
1 error
```

Relational and **logical** Operators (cont.)

And

p	q	p · q
T	T	T
T	F	F
F	T	F
F	F	F

Or

p	q	p ∨ q
T	T	T
T	F	T
F	T	T
F	F	F

- ! Logical negation

  `!0` returns 1, `!` (any non-zero value) returns 0 `!-4`

Not valid in Java

- || logical or, && logical and
  - `&&` returns 1 if both non-zero. Otherwise 0 `3 && -2`
  - `||` returns 1 if either non-zero. Otherwise 0 `3 || 5`
  - **Short-circuit** (lazy) evaluation stops when we have an answer

```
int  x = 1, y = 1
if (x == 0 && y == 0)….;
```

same in Java

Java example:

```
if ( object != null && object.data  > 9)
  ……
```

37

37

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

38

YORK U UNIVERSITÉ UNIVERSITY

38

Type conversion – 4 scenarios

1. Given an expression with operands of mixed types, C converts (promotes) the types of values to do calculations
2. Conversion may happens on assignment
3. May happens on function call arguments
4. May happens on function return type

39

39

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**?

40

40

Scenario 1 -- Type conversion (Promotion)

- 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 type.

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

same in Java

- E.g., for **`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`**

short / char ⟹ int ⟹ long ⟹ float ⟹ double ⟹ long double

Examples:

char, int → operator → int

int, double → operator → double

short → int → long → float → double → long double
char

Mixed type arithmetic

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

- 17 / 5
  - 3
- 'k' + 32
  - 75 + 32 = 107
  - Return type int

same in Java

- 17.0 / 5
  - 17.0 / 5.0 = 3.4
  - Return type double

- 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
- 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

43

2 conversions

Associativity: left to right

3 conversions

43

Scenarion 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  512.0 */
        /* return type float, return value 512.0 */
```

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 = 512.3;
int i = f;  /* f is converted to int 512 */
```

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

44

Not valid in Java

UNIVERSITÉ
UNIVERSITY

44

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'
```

45

45

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
}
```

46

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
}
```

47

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
  - 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) */
```

48

48

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  */
```

49

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  */
```

50

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)  ?

int d = (int)f
```

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

Needed in Java

51

51

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

52

52

Bitwise operators

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

01100101	01101100	01101100	01101111	00000000
01001000	01100101	01101100	01101100	01101111

- bitwise **& | ~ ^**

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

53

53

Bitwise Operators

Or

p	q	$p \vee q$
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
  - either bit is 1 ➔ the result is 1 (set) 0: keep whatever other,

- e.g.

```
z= 145 | 41
                  145= 000…10010001
                  41 = 000…00101001
                       000…10111001 = 185 (decimal)
                                      0271 (oct)
                                      0Xb9 (hex)
```

54

same in Java

54

Bitwise Operators

And

p	q	$p \cdot q$
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
  - (both bits must be 1 to result in 1)
  - Either is 0: 0 (off) 1: keep whatever the other

- e.g.

```
z= 145 & 41       145= 000…10010001
                   41= 000…00101001
z= 1                   000…00000001   = 1 (decimal)
```

55

same in Java

55

Bitwise Operators

Exclusive-OR gate

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

e.g. **z= 145 ^ 41**

```
145 = 000…10010001
 41 = 000…00101001
      000…10111000 = 184  (decimal)
                     0270 (oct)
                     0Xb8 (hex)
```

56

same in Java

56

Bitwise Operators

Not

p	$\sim p$
T	F
F	T

Rhs	0	1
Result	1	0

- **~**
  - one's complement (bit inversion)
  - flips all bits in its operand

- e.g.(assuming unsigned char)

```
z= ~145        145= 10010001
                ____________
                01101110  = 110 (decimal)
                            0156 (oct)
                            0X6E (hex)
```

57

same in Java

57

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**)

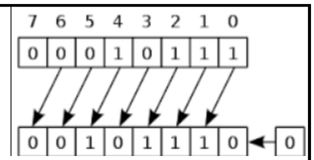
What goes Out? bits pushed "off the end" on the end

What comes in? >> << different

58

58

Bit Shifting <<



- Suppose **z** is an int
  - e.g.
    - **z= 26 << 3**
    - **shift left 3 bits**
    - **z= 208**

      0320

      0XD0

26 = 000 000…..11010

000… . 11010000

000

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

What comes in? we add 0 on the right

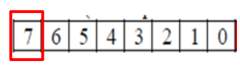
YORK UNIVERSITÉ UNIVERSITY

59

z = x << 3 does not change x

59

Bit Shifting >>



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

  z = 154 >> 3

  shift left 3 bits

154=…. 10011010

… 00010011 = 19

010

z = x >> 3 does not change x

YORK UNIVERSITÉ UNIVERSITY

60

60

Bit Shifting- What Comes In >>

- What about "signed" values?   7 6 5 4 3 2 1 0
  - 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)
  - -94 >>3   -94= 10010010
  - logical   00010010= 18
  - arithmetic   11110010= -14

YORK UNIVERSITÉ UNIVERSITY

61

61

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 >>>
>> what ever leftmost is
>>> always 00…

YORK UNIVERSITÉ UNIVERSITY

62

62

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 -- 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
- Use XOR(^) to toggle bits (0 -> 1 and 1-> 0)

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

OR			AND			XOR		
A	B	Output	A	B	Output	A	B	Output
0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	1	1
1	0	1	1	0	0	1	0	1
1	1	1	1	1	1	1	1	0

Flags (some idioms)

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

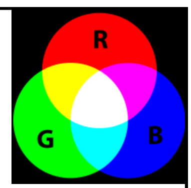
- Unsigned char **`flags`**;
  - **`flags = flags | (1<<5)`**
    - 00100000. Turn 6th bit on (set to 1) (lowest-bit is 1st bit)
  - **`flags = flags & ~(1<< 5)`**
    - 11011111. Turn 6th bit off (set to 0)
  - **`flags = flags & (1<<5)`**
    - 00100000. keep 6th bit only (other off)
  - **`flags = flags & 0177`**
    - …. 001 111 111. Set to zero all but the low-order 7 bits of flag
  - **`flags = flags & ~077`**
    - 000111111->111000000. Set last 6 bits to zero (turn off)

65

Practice in the lab. Revisit next time

65

Some examples



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

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

- 00000000 11111101 01001000 10101011

  ^ Alpha ^Red ^Green ^Blue

java.awt.image

Class BufferedImage

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

getRGB

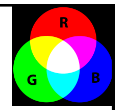
```
public int getRGB()
```

66

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).

66

Some examples



- I 1: turn on
- & 0: turn off
- I 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?

00001010 11111101 01001000 10101011

^ Alpha ^Red ^Green ^Blue

&

00000000 00000000 00000000 11111111 255 0377 0xFF

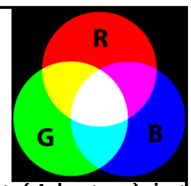
Turn off keep value

00000000 00000000 00000000 10101011

- `int blue = (rgb_pack) & 0377;` // rgb_pack not changed */

67

Some examples



- In Java, getRGB() packS 3 +1 values into a 32 bit (4 bytes) int
- 10101100 11111101 01001000 11111111

^ Alpha ^Red ^Green ^Blue

How to get red value?

- First shift to the right end. 00000000 00000000 10101100 11111101
- Then need **000 ….. 11111111** to turn off 9~32 bits -- mask

68

Select Command Prompt

```
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wireless Network Connection:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::81c8:be2f:99d:2c
   IPv4 Address. . . . . . . . . . . : 130.63.199.163
   Subnet Mask . . . . . . . . . . . : 255.255.254.0
   Default Gateway . . . . . . . . . : 130.63.198.1

Ethernet adapter Local Area Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . : cs.yorku.ca

Tunnel adapter isatap.{4C22CACA-FD21-424B-91FD-614F471D1709}:
```

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

YORK UNIVERSITÉ UNIVERSITY

69

Expressions

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

YORK UNIVERSITÉ UNIVERSITY

70

70

Somethings to Think About

- | looks similar to || Both do "OR"
- & looks similar to &&   Both do "AND"

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

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

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

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

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

71

71

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

72

YORK UNIVERSITÉ UNIVERSITY

72

(Compound) Assignment Operators

- C (and Java) provides other "short-hand" assignment operators (we've seen `++` and `--`)
- e.g.
  - `x += 5;      <-->     x = x + 5`
  - `x *= 5;      <-->     x = x * 5`

73

YORK U UNIVERSITÉ UNIVERSITY

73

Assignment Op. & Expressions

- Assignment operator: "**op=**"
  exp1 op= exp2 is equivalent to
  exp1 =(exp1) op (exp2)
  - exp1 and exp2 are expressions
- **op** can be:
  `+ - * / % << >> & ^ |`
- Thus, we can have
  `+=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=`

`flags = flags | (1<<5)` ⇔ `flag |= (1 << 5)`

74

YORK U UNIVERSITÉ UNIVERSITY

74

Assignment Op. -- Examples

- `x *= y + 1` is equivalent to `x = x * (y +1)`
  - Because `*=` has low precedence than `+`

  - `x=2; y=2; x *= y + 1 + 5;`
    `x` has value `2*(2+1+5) = 16`

same in Java

unsigned int x;

- `x =24; x >>=2;` 00011000 24 -> 6
  x is 00000110

- `x =24; x <<=2;` 00011000 24 -> 96
  x is 01100000

- `x =24; x |= 0x02;` 00011000 |
  00000010
  x is 00011010 24->26

75

Turn on 2nd bit

75

Conditional operator

- `exp1 ? exp 2: exp 3`
- If `exp1` is true, the value of the conditional expression is `exp2`; otherwise, `exp3`

```
z = (a > b) ?  a : b; /* z = max (a,b) */
if (a>b)
   z=a;
else z=b;
```

76

Java vs. C, types and operators

	Java	C
Boolean	boolean	int 0/1
Integer types	byte // 8 bits short // 16 bits int // 32 bits long // 64 bits	char unsigned char short unsigned short int unsigned int long unsigned long
String type	String s1 = "Hello"; String s2 = new String("hello");	char s1[] = "Hello"; char s2[6]; strcpy(s2, "hello");

String concatenate	s1 + s2	#include <string.h>
		strcat(s1, s2);

Logical	&&, ||, !	&&, ||, !
Compare	=, !=, >, <, >=, <=	=, !=, >, <, >=, <=
Arithmetic	+, -, *, /, %, unary -	+, -, *, /, %, unary -
Bit-wise ops	>>, <<, >>>, &, |, ^	>>, <<, &, |, ^
Assignments	=, *=, /=, +=, -=, <<=, >>=, >>>=, =, ^=, |=, %=	=, *=, /=, +=, -=, <<=, >>=, =, ^=, |=, %=

77

77

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**

YORK UNIVERSITÉ UNIVERSITY

78

78

Precedence

- How do we interpret:
  - `a && b || c && d`
  - `i << 2 +1`   `flag | 1 << 4`
  - `i *= y+1`
  - `(int) f1/f2`

- Rules of precedence tell us what gets evaluated first:
  - `a && b || c && d`
  - `i << 2 + 1`   `flag | 1 << 4`
  - `i *=  y + 1`
  - `(int) f1 / f2`

- Precedence should be familiar from basic math:
  - Given "**x+y*5**", you evaluate "**y*5**" first:
    - **x + (y*5)**

Precedence

```
#include <stdio.h>

main(){
 int c;
 c = getchar();
 while(c != EOF)
 {
   putchar(c);
   c = getchar();/*read next*/
 }
}
```

Succinct code

```
#include <stdio.h>

main(){
 int c;

 while( c = getchar() != EOF )
 {
   putchar(c);

 }
}
```

?

Precedence and Associativity p53

- Observe that:
  - Parentheses first
  - Negation(!,~) next
  - Arithmetic before Relational
  - Arithmetic: /, *, % before +-
  - Relational before Logical
  - Logical: && before ||
  - Bit shift << >> before bitwise & ^ |
  - Assignment = += very low

```
if ( a&&b || c && d)
while((c=getchar()) == EOF)
i << 2 + 1     // i = i << 3
if (a==2 || a<4)
i *= y + 1  // i=i*(y+1)
(*p).data
```

- When in doubt – use parentheses
  - Also for clarity

- Don't need to memorize
  - Will be provided in tests
  - But know how to use them

Similar in Java

Operator Type	Operator	
Primary Expression Operators	() [] . -> expr++ expr--	
Unary Operators	* & + - ! ~ ++expr --expr (typecast) sizeof	
Binary Operators	* / %	arithmetic
	+ -	arithmetic
	>> <<	bit shift
	< > <= >=	relational
	== !=	relational
	&	bitwise
	^	bitwise
	|	bitwise
	&&	logical
	||	logical
Ternary Operator	?:	
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= |=	
Comma	,	

Summary and future work

- Type, operators and expressions (Chapter 2) :
  - Types and sizes
    - Basic types, their size and constant values (literals)
      - ✓ char: x > 'a' && x < 'z'; x > '0' && x < '9'
      - ✓ int: 122, 0122, 0x12F convert between Decimal, Bin, Oct, Hex
    - Arrays (one dimension) and strings (Ch1.6,1.9)
      - ✓ "hello" has size 6 byte | H | e | l | l | o | \0 |
  - Expressions
    - Basic operators (arithmetic, relational and logical)
      - ✓ y=x++; y=++x;
      - ✓ if (x = 2)
    - Type conversion and promotion
    - Other operators (bitwise, bit shifting , compound assignment, conditional)
      - ✓ Bit: |, &, ~. ^, << >>
      - ✓ Compound: x += 10; x >>= 10; x += y + 3
    - Precedence of operators
- Next: Functions and Program Structure (Chapter 4)