Lecture 5

- Operators continued
- Expressions
- Flow control

Illustration

```
int a,b,c,f;

a = a & 0xff; // clears all but the lowest 8 bits
              // ( 0&0=0, 0&1=0, 1&0=0, 1&1=1 )

b = b | 5;    // sets the lowest and third lowest bit
              // in b
              // ( 0|0=0, 0|1=1, 1|0=1, 1|1=1 )

c = c ^ 0xffff0000; // inverts the highest 16 bits
                    // in c
                    // ( 0^0=0, 0^1=1, 1^0=1, 1^1=0 )

f = ~f;       // negates all bits in f
              //~0=0xffffffff, ~0x55555555=0xaaaaaaaa
```


Shift Operations

```
int d,e,x;
// set d,e,x

d = d >> x;   // shifts all bits in d x places to
              // the right. The lower x bits are
              // lost, x must be >= 0 and < 32
              // E.g. 8 >> 1 = 4, 10 >> 2 = 2

     // IMPORTANT: if d is a signed variable
     // the most significant bit is not changed.
     // If d is unsigned, a 0 comes in from the left

e = e << x;   // shifts all bits in e x (0<=x<32) places to
              // the left. x 0s come in from the right,
              // E.g. 8 << 1 = 16, 10 << 2 = 40
```


Expressions

```
(a+b)*(a-b)          // OK
)a+b(                // not OK
(a2*x+a1)*x+a0       // OK
a + b + c            // OK, a + b first
a + b * c            // OK, * first
(a >= b) || (c != 1) // Boolean expression
```

- Built from variables, constants, operators, and ()
- infix notation
- () used for explicit evaluation order, must be **balanced**
- Operators have fixed arity, associativity & precedence

Conditional Expressions

```
int x,a,b;

x = (a > b) ? a : b;  // x = max(a,b)
```

- **exp1 ? exp2 : exp3**
- **exp1** is evaluated first. If it is true, then **exp2** is evaluated and this is the value of the conditional expression
- if not, **exp3** is the value of the expression

Comma Operator

```
result = (first=1, second=first+1);

// result is 2
```

- Evaluate many expressions in one statement
- Syntax: <exp_1> , <exp_2> , ... <exp_n>
- Semantics:
  - expression evaluated **from left to right**
  - value of last expression is returned
  - , has lowest operator precedence of all

Assignment Operators

```
int a,b,c;
float d;

a = a + 4;      a += 4;     // equivalent
b = b >> x;     b >>= x;    // equivalent
c = c | 3;      c |= 3;     // equivalent
d = d * (a+1);  d *= a+1;   // equivalent
```

- Set/change value of variable
- Syntax: <variable> = <expression> ;
- **i OP= c** equivalent to **i = i OP c**, where **OP** is one of **+ - * / % << >> & ^ |**

Type Conversions

- Types of variable and expression must be compatible
- Value is converted to type of variable

```
int a; double b;

// Implicit type casting
b = a; // OK, int fits into double
a = b; // not OK, warning should be issued

// Explicit type casting
a = (int)b;              // oldest C style
a = int(b);              // older C++ style
a = static_cast<int>(b); // new C++ style
```

Explicit casts supress warnings, but precision may be lost! Floating point numbers are truncated when converted to integers, not rounded!

Associativity and Precedence

```
() [] -> .                                  ltr high
! ~ ++ -- +(1) -(1) *(1) &(1) (type) sizeof   rtl
* / %                                       ltr
+(2) -(2)                                   ltr
<< >>                                       ltr
< <= > >=                                   ltr
== !=                                       ltr
&(2)                                        ltr
^                                           ltr
|                                           ltr
&&                                          ltr
||                                          ltr
?:                                          rtl
= += -= *= /= %= &= |= <<= >>=              rtl
,                                           ltr low
```

rtl = right to left, **ltr** = left to right, unary +-* higher precedence than binary ops.

Precedence Examples

```
a = b + c * d;         // * before + before =

a = b >= 5 && c <= 6;  // >= (before <=)before =

a = b = c+1;           // right to left evaluation
                       // c+1 before b= before a=
```


Program Flow Control

- if-then-else
- goto
- loops
- functions

If-Statements

```
if (y > x) x = y; // x = max(x,y)

if (x < 0) {
  sign = -1;
} else if (x > 0) {
  sign = +1;
} else
  sign = 0;
```

- if (<expr>) <statement>
- if (<expr>) <then-statement> else <else-statement>
- if (<expr>) <statement-1> else if (<expr>) <statement-2>
- <statement> : any statement, including statement list enclosed in **{}** and additional if-statements!
- <expr> is evaluated; if true, <then-statement> is executed otherwise – if there is an <else-statement> it is executed

Goto Statement

```
...
goto label;

...

label: ;  // resume here
```

- Control flow resumes at a specific location marked by a label (syntax: <**identifier**>**:**)
- **Use rarely!** goto code is hard to understand and maintain ~ **"Spaghetti code"**

Loops

- Repeat execution of statements until a condition is met
- Three forms:
  - **while** (<test-expr>) <statement>
  - **do** <statement> **while** (<test-expr>) ;
  - **for** (<init>; <test-expr>; <update>) <statement>

while Loop

- **while** (<test-expr>) <statement>
- while expression evaluates to true execute statement

```
// add values 1..100

int sum=0, i=1;

while (i <= 100) { sum += i; i++; }
```


do Loop

- **do** <statement> **while** <test-expr> ;
- first execute statement and loop if expression evaluates to true

```
int s=0, i=1;

do { s = s+i; i = i+1; } while (i <= 100);
```

for Loop

- **for** (<init> ; <test-expr> ; <update>) <statement>

  is equivalent to:

  <init> ;
  while (<test-expr>) { <statement>; <update>; }

```
int s=0;

for (int i=1; i <= 100; ++i) s += i;
```


Loop Control

- break; exits loop immediately
- continue; skips loop body

```
while (...) {
  ...
  break;
  // equivalent to
  // goto break_loc;
  ...
}

break_loc: ;
```

```
while (...) {
  ...
  continue;
  // equivalent to
  // goto cont_loc;
  ...

  cont_loc: ;
}
```

In for loops, continue resumes with the update