W.P.Hwang
CS-CCS-NCTU

Lecture – Expressions

Arithmetic expressions

- Operators
  Unary `+ -`
  Binary `+ - * / %`

- Integral `/` integeral = integral; otherwise,= floating
  Both operands of `%` must be of integral type.

- (a`/`b)`*`b `+` a`%`b = a

  If a and b are non-negative, so is a`%`b

  If either a or b is negative, there are two ways to divide:

  −5/2 = −2 −5%2 = −1 ⋯ (1)
  −5/2 = −3 −5%2 = 1 ⋯ (2)

  In either case, the equality (−5/2)*2 + −5%2 = −5 holds.

  Mathematically, sign(a`%`b) = sign(b) and so the result is (2).
  C99 rounds the quotient toward zero (as most CPUs do) and so the result is (1).
  C89 and C++ don't specify the result.

- Example

  For *T* = signed or unsigned

```
bool even(T n) { return n%2==0; }
bool odd(T n) { return n%2!=0; }
```

  For *T* = unsigned only

```
bool odd(T n) { return n%2==1; }
```

- Division by zero
  Integer division by zero yields an error.
  Real division by zero yields an infinity.

```
int n=0; double x=0.0;
printf("%d",5/n);     // runtime error
printf("%d",5/0);     // compile time error
printf("%f",5/x);     // implementation-defined rep. of infinity
```

Relational expressions

- Operators
  Binary **`>  >=  <  <=  ==  !=`**

- Example
  Check if $0 \le n \le 9$
  `0<=n&&n<=9` // ok
  `0<=n<=9` // no, always true

Logical expressions

- Operators
  Unary **`!`**
  Binary **`&&  ||`**

- **`&&`** and **`||`** are short-circuit and hence non-commutative.
  *exp*1 **`&&`** *exp*2 = *exp*1 **`?`** *exp*2 **`:`** false
  *exp*1 **`||`** *exp*2 = *exp*1 **`?`** true **`:`** *exp*2

- Example

```
int m,n=0;
n!=0&&m/n>5             // false
m/n>5&&n!=0             // error
```

- Example – Sequential search, check if $k \in \mathrm{a}[0..n-1]$

```
bool find(int k,int a[],int n)     // int a[7];
{                                  // find(5,a,7)
   int i;
   for (i=0;i<n;i++) if (a[i]==k) break;
   return i<n;
}
```

or

```
bool find(int k,int a[],int n)
{
   int i;
   for (i=0;i<n&&a[i]!=k;i++);   // watch the order
   return i<n;
}
```

Conditional expressions

- Operator
  Ternary **?:**

- Example – Find max(a,b,c)

```
max = a>b? a>c? a: c: b>c? b: c;
```

cf.

```
max=a;
if (max<b) max=b;
if (max<c) max=c;
```

or, more generally,

```
int max(int a[],int n)
{
    int m=a[0];
    for (int i=1;i<n;i++)
        if (m<a[i]) m=a[i];
    return m;
}
```

- Given

  *exp*1**?** *exp*2**:** *exp*3

  if *exp*2 and *exp*3 are of different type, they are brought to a common type, and that is the type of the result.
  For example,

```
(1==1? -1: 1) < 0      // true
(1==1? -1: 1u) < 0     // false
```

  To see why, consider

  (*exp*1**?** *exp*2**:** *exp*3**)** + *exp*4

  In order to type-check the addition, the compiler has to know the type of the conditional expression.

Bitwise operators

- Operators

  Unary **~**
  Binary **&** **|** **^** **<<** **>>**

  Comments

  1 The operands shall be of integral or enumeration type.
  2 There are also assignment operators: **&=**, **|=**, **^=**, **<<=**, **>>=**.

- Example

  `unsigned x=0x5,y=0x6;`

x	000...101		unsigned
y	000...110		
x&y	000...100	AND	
x|y	000...111	inclusive OR	
x^y	000...011	exclusive OR	
~x	111...010	NOT	
x<<1	000..1010	left shift	
x>>1	000...010	right shift	

  Cf.

x&&y	000...001	logical AND	bool
x||y	000...001	logical OR	
!x	000...000	logical NOT	

- Properties of **&**, **|**, and **^**

$$\begin{array}{r} x\ x \\ \underline{\&\ 0\ 1} \\ 0\ x \end{array} \qquad \begin{array}{r} x\ x \\ \underline{|\ 0\ 1} \\ x\ 1 \end{array} \qquad \begin{array}{r} x\ x \\ \underline{\hat{}\ 0\ 1} \\ x\ \bar{x} \end{array} \qquad \Leftarrow \text{ mask}$$

- Properties of << and >>

  x << n insert 0 from the right; $= x \times 2^n$
  x >> n
  unsigned x insert 0 from the left; $= x/2^n$
  signed x implementation dependent
  insert 0 from the left or sign extension

- Properties of **<<** and **>>** (Con't)

  Cf. 98 << 2 = 9800 = 98×10^2
  9876 >> 2 = 98 = $9876/10^2$

  N.B. If $n < 0$ or $n \geq$ 8*sizeof(x), the behavior is undefined.

- Example

  For *T* = signed or unsigned

```
bool even(T n) { return (n&1)==0; }
bool odd(T n) { return (n&1)==1; }
bool even(T n) { return n>>1<<1==n; }
bool odd(T n) { return n>>1<<1!=n; }
```

- Example

```
n*10
(n<<2)+ n<<1
(n<<3)+(n<<1)
```

  All are equivalent, but multiplication is expensive.

- Example

  Method 1: Varied mask

	140	113	235	131	
&	0	FF	0	0	mask
	0	113	0	0	
	0	0	0	113	

```
void showip(unsigned ip)
{
    for (int i=3;i>=0;i--) {
        printf("%u",(ip&0xFF<<8*i)>>8*i); // *
        if (i!=0) printf(".");
    }
    printf("\n");
}
```

  Method 2: Fixed mask

	140	113	235	131	
	0	0	140	113	
&	0	0	0	FF	mask
	0	0	0	113	

  Replace the starred line by

```
printf("%u",ip>>8*i&0xFF);
```

- Example – Find 2's complement representation of an integer

Version C – Bit vector

```
void twoscomp(int n)
{
   const int bits=8*sizeof(int);
   unsigned a=0;
   unsigned m=abs(n);
   int i=0;
   while (m>0) {
      a|=(m&1)<<i; m>>=1; i++;
   }
   if (n<0) {
      for (i=0;i<bits;i++) a^=1<<i;            ①
      i=0;
      while ((a&1<<i)!=0) { a^=1<<i; i++; }   ②
      a^=1<<i;                                 ③
   }
   for (i=bits-1;i>=0;i--) printf("%u",a>>i&1);
   printf("\n");
}
```

i=3 ↓

a	0...00xxx
(m&1)<<i	0...0?000
a|=(m&1)<<i	0...0?xxx

i=3 ↓

a	x...xxxxx
1<<i	0...01000
a^=1<<i	x...x̄xxx
a&1<<i	0...0x000

① change 1 to 0 and 0 to 1
or, `a = ~a;`
or, `a ^= -1;`
or, `a ^= 0xFFFFFFFF;` assuming that sizeof(int)=4

② change 1 to 0
or, `a &= ~(1<<i);`

③ change 0 to 1
or, `a |= 1<<i`

i=3 ↓

a	x...x1xxx
~(1<<i)	1...10111
a&=~(1<<i)	x...x0xxx

i=3 ↓

a	x...x0xxx
1<<i	0...01000
a|=1<<i	x...x1xxx

Increment and decrement operators

- Operators
  Unary `++` `--`

- Prefix and postfix increment
  Let e_0 be the current value of the expression *exp*, then

Exp	Value	Side effect
exp++	e_0	increment *exp* by 1
++*exp*	e_0+1	〃

- The operand must be a modifiable (i.e.non-const) lvalue.

  address (lvalue) value (rvalue)

```
x = x+1;
```

  e.g. `7++; const x=2; x++;` // illegal

Assignment expressions

- Operators
  Binary `=`
  op= for arithmetic and bitwise binary operator *op*

- *exp*1 = *exp*2
  Value the value of *exp*1 after executing the assignment
  Side effect the value of *exp*2 is stored in *exp*1

```
int z;
printf("%d",z=3.9);
```

- The left operand must be a modifiable lvalue.

- *exp*1 *op*= *exp*2 // *exp*1 is evaluated once
  *exp*1 = *exp*1 *op* (*exp*2) // *exp*1 is evaluated twice

  E.g.

  `x*=y+2` = `x=x*(y+2)`
  ≠ `x=x*y+2`

- (Cont'd)

  Let **i = 2**, then

```
a[i]+=5      =  a[i]=a[i]+5
a[i++]+=5    ≠  a[i++]=a[i++]+5
    ↑                ↑
a[2]+=5         a[2]=a[3]+5 or a[3]=a[2]+5
i=3             i=4
```

Q: Consider
*exp*1 + *exp*2 ***** *exp*3
f(*exp*1**,***exp*2**,***exp*3**)**
In what order should *exp*1, *exp*2, and *exp*3 be evaluated?
A: This depends on languages.
Operator evaluation order is specified in C/C++.
But, operand (or argument) evaluation order is unspecified.

Q: Some operators in C/C++ have required operand evaluation order. What are they and why?
A: **?: && || ,**
Due to semantics

Lesson
Never write things that depend on argument evaluation order.

Another example

```
int f(void) { printf("Snoopy"); return 2; }
int g(void) { printf("Pluto"); return 3; }
printf("%d",f()+g());        // undefined
```

Depending on the desired output order, it should be written as

```
int x=f();
printf("%d",x+g());
```
or
```
int x=g();
printf("%d",f()+x);
```

Comma expressions

- Operator
  Binary ,

- Comma expression

  *exp*1 , *exp*2

  Evaluate *exp*1 and then *exp*2
  The value of *exp*2 is the value of the comma expression.
  Since the value of *exp*1 is discarded, *exp*1 should be an I/O or assignment expression or an expression of **`void`** type, e.g.

```
x+y,x            // x+y is redundant
x=2,x
scanf("%d",&x),x
p(2),p(3);
```

- Example

```
void p(int);
P(x=2,x);       // error
p((x=2,x));     // ok, same as  x=2; p(x); or  x=2, p(x);
```

- Example

```
while (true) {
   if (n==0) break;       // can't be replaced by a comma
   printf("%d",n%10);     // can be replaced by a comma
   n/=10;
}
```

- Example

```
printf("Enter an integer: ");
while (scanf("%d",&x)!=EOF) {
   ...
   printf("Enter an integer: ");
}
```

⇓

```
while (printf("Enter an integer: "),
                                   scanf("%d",&x)!=EOF) {
   ...
}
```

- Example

```
for (int i=1;i<=9;i++) {
   for (int j=1;j<=9;j++) printf("%4d",i*j);
   printf("\n");
}
```

$\Downarrow$

```
for (int i=1,j=1†;i<=9;j==9?printf("\n"),i++,j=1:j++)
   printf("%4d",i*j);
```

† This isn't a comma expression; cf.

```
int i,j;
for (i=1,j=1;...;...)...;        // comma expression
```

sizeof operator

- Operator
  Unary **`sizeof`**

- Syntax **`sizeof`** *exp*
  `sizeof(`*type***`)`**

- **`sizeof`** is a compile-time operator. In particular, the expression exp isn't evaluated, only its type is analyzed by the compiler.

- Example

  `sizeof x+1` $\neq$ **`sizeof(x+1)`**
  `sizeof x++`   // **`x`** unchanged

- The value of a **`sizeof`** expression is of implementation-defined unsigned type, called **`size_t`**.

  Motivation:

  T **`x=sizeof(int);`**

  How should we declare *T*?

- The type **`size_t`** is defined in **`<stddef.h>`**, say

  `typedef unsigned int size_t;`

  Note: **`typedef`** doesn't introduce new types.

```
size_t x; unsigned y;
x=y;              // no type conversion required
```

Precedence and associativity

	Precedence		Associativity
1	`::`		
2	`. -> [] ()`		
	`++ --` postfix		
	`typeid static_cast` etc		
3	`+ - ++ --` prefix	unary	right
	`! ~ & *`		
	`sizeof new delete (`*type*`)`		
4	`.* ->*`		
5	`* / %`		`2+3*4` precedence
6	`+ -`		`2-3-4` associativity
7	`<< >>`		
8	`< <= > >=`		
9	`== !=`		
10	`&`		
11	`^`		
12	`|`		
13	`&&`		
14	`||`		
15	`?:`		right
16	`=` *op*`=`		right
17	`throw`		
18	`,`		

Remarks

1 The shaded operators and precedence levels are for C++ only.
2 Unless stated otherwise, all the operators are left-associative.
3 In C++, `(`*type*`)` is in the 3rd precedence level; but in C, it is in the 4th precedence level.

Examples

Example 1 – Compute a^n for integer $n \geq 0$

Algorithm A – $O(n)$ divide and conquer algorithm

$$a^n = \begin{cases} 1 & n = 0 \\ a * a^{n-1} & n > 0 \end{cases}$$

```
int pow(int a,unsigned n)
{
   int r=1;
   while (n>0) { r*=a; n--; }
   return r;
}
```

Algorithm B – $O(\log n)$ divide and conquer algorithm

$$a^n = \begin{cases} 1 & n = 0 \\ a * a^{n-1} & n > 0 \text{ is odd} \\ (a^2)^{n/2} & n > 0 \text{ is even} \end{cases}$$

[

Version 1

```
int pow(int a,unsigned n)
{
   int r=1;
   while (n>0)
      if (n%2==1) { r*=a; n--; }
      else  { a*=a; n/=2; }
   return r;
}
```

Comments

1 For $n = 2^k$, the loop is executed $k + 1 = \log_2 n + 1$ times, for $n = 2^k \to 2^{k-1} \to 2^{k-2} \to \cdots \to 2^1 \to 2^0 \to 0$

2 Better algorithms reduce the order of time-complexity function. Better coding reduces the coefficient of time-complexity function.

Example 1 (Cont'd)

Version 2

```
int pow(int a,unsigned n)
{
   int r=1;
   while (n>0) {
      if (n%2==1) r*=a;    ①
      a*=a; n/=2;          ②
   }
   return r;
}
```

① `n--` is removed

② Version 2 does one more `a*=a` than version 1.

Version 3

```
int pow(int a,unsigned n)
{
   int r=1;
   while (n>0) {
      while (n%2==0) { a*=a; n/=2; }
      r*=a; n--;
   }
   return r;
}
```

Q: How many multiplications are done by Version 1 or 3?

multiplication	n	bit pattern
1	11	1011
2	10	1010
3	5	101
4	4	100
5	2	10
6	1	1
	0	0

# of multiplications
= # of 0-bits + 2 × # of 1-bits – 1
= # of bits + # of 1-bits – 1

Example 1 (Cont'd)

Let

$t(n)$ = the number of multiplications done by Version 1 or 3

$\text{bits}(n)$ = the number of bits in binary representation of n

$\text{ones}(n)$ = the number of 1-bits in binary representation of n

Then,

$$t(n) = \text{bits}(n) + \text{ones}(n) - 1$$

Observe that

$$\text{bits}(n) - 1 \leq t(n) \leq 2\text{bits}(n) - 1$$

Now, suppose that $\text{bits}(n) = k$, then

$$2^{k-1} \leq n < 2^k \Rightarrow k - 1 \leq \log_2 n < k \Rightarrow k = 1 + \lfloor \log_2 n \rfloor, \quad n \geq 1$$

or, $2^{k-1} < n + 1 \leq 2^k \Rightarrow k = \lceil \log_2(n+1) \rceil, \ n \geq 0$

Hence, $t(n) = O(\log_2 n) = \mathrm{O}(\log n)$

Notice that the base of logarithm is immaterial in big-*O* notation, for

$$\log_a n = \frac{\log_b n}{\log_b a}$$

```
unsigned bits(unsigned n)
{
   return n==0? 1: 1+floor(log(n)/log(2));
// return n==0? 1: ceil(log(n+1)/log(2));
}
```

Remarks

1 The call to **`floor`** may be omitted.

2 The natural logarithm **`log`** may be replaced by the common logarithm **`log10`**.

3 In C99, we may also use the **`float`** or **`long double`** versions, such as **`logf`**, **`logl`**, **`ceilf`**, **`ceill`**, etc.

4 In C++, the calls to **`log`** must have floating-point arguments, e.g. **`log((double)n)/log(2.0)`**

Example 1 (Cont'd)

Version 1 – $O(\text{bits}(n))$

```
unsigned ones(unsigned n)
{
   unsigned r=0;
   while (n>0) { r+=n&1; n>>=1; }
   return r;
}
```

Version 2 – $O(\text{ones}(n))$

```
unsigned ones(unsigned n)
{
   unsigned r=0;
   while (n>0) { n&=n-1; r++; }
   return r;
}

int pow(int a,unsigned n)  // for fun
{
   int r=1;
   for (int i=1;i<=bits(n)+ones(n)-1;i++)
      if (n%2==1) { r*=a; n--; }
      else { a*=a; n/=2; }
   return r;
}
```

Example 2 – Find the gcd of integers $a \geq 0$ and $b \geq 0$

Algorithm A – Euclid's algorithm

$$\gcd(a,b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b > 0 \end{cases}$$

Convention: $\gcd(0,0) = 0$

```
unsigned gcd(unsigned a,unsigned b)
{
   while (b>0) {
      unsigned c=a%b;
      a=b; b=c;
   }
   return a;
}
```

Euclid's algorithm needs only a small number of iterations.

	a	b	
	1024	24	
42	1008	16	1
	16	8	
2	16		
	0		

a	b	k
1024	24	1
512	12	2
256	6	3
128	3	
64	3	
32	3	
16	3	
8	3	
4	3	
2	3	
1	3	
1	2	
1	1	

Algorithm B – Binary method

$$\gcd(a,b) = \begin{cases} a & a = b \text{ both odd} \\ \gcd(a-b, b) & a > b \text{ both odd} \\ \gcd(a, b-a) & a < b \text{ both odd} \\ \gcd(a/2, b) & a \text{ even}, b \text{ odd} \\ \gcd(a, b/2) & a \text{ odd}, b \text{ even} \\ 2 * \gcd(a/2, b/2) & \text{both even} \end{cases}$$

In general, binary method requires more iterations than Euclid's algorithm, but the iterations in binary method have greater speed. Empirical studies show that binary method is about 20% faster.

Example 2 (Cont'd)

Version 1

```
#define even(n) (((n)&1)==0)     ①
inline bool odd(unsigned n)      ②
{
   return (n&1)==1;
}
unsigned gcd(unsigned a,unsigned b)
{
   unsigned k=0;
   while (!(odd(a)&&odd(b)&&a==b))
      if (odd(a)&&odd(b))
         if (a>b) a-=b; else /*if (a<b)*/ b-=a; ③
      else if (even(a)&&even(b)) {
         k++; a>>=1; b>>=1;
      } else if (even(a)) a>>=1;
      else b>>=1;
   return a<<k;
}
```

① It is wrong to define the macro as

```
#define even(n) (n&1)==0
```

because the macro call **even(2⊕3)⊗ 4**
will be expanded into **(2⊕3&1)==0 ⊗ 4**,
which isn't what one would expect if the precedence of ⊕ is lower than that of **&**, or the precedence of ⊗ is higher than that of **==**.

Macro expansion

The macro calls of **even** within

```
if (even(a)&&even(b)) ...
```

are expanded to

```
if (((((a)&1)==0)&&(((b)&1)==0)) ...
```

Example 2 (Cont'd)

② Inline functions are supported by C99 and C++, but not C89. The inline function calls of **odd** within

```
if (odd(a)&&odd(b)) ...
```

are compiled to something like

```
unsigned n;
if (((n=a,(n&1)==0)&&(n=b,(n&1)==0)) ...
```

In short, inline function calls don't have the runtime overhead of non-inline function calls and are nearly as efficiency as macro calls.

Why macros or inline functions, rather than non-inline functions? (for "small" functions only. Large functions will bloat the code.)

To reduce runtime overhead
Macros
The preprocessor does macro expansions for macro calls.
Inline functions
The compiler generates code inline for inline-function calls. (But, the compiler may choose to ignore inline requests.)

Prefer inline functions to macros

Macros
1 Must parenthesize the macro body properly
2 Arguments may be evaluated more than once
3 Type flexible but unsafe

Inline functions
1 Needn't parenthesize the function body
2 Arguments are evaluated only once
3 Type inflexible but safe

For point 2, consider

```
#define square(x) ((x)*(x))
```

The argument is evaluated twice.

```
square(2+3)  ⇒  ((2+3)*(2+3))     // inefficient
square(++x)  ⇒  ((++x)*(++x))     // incorrect
```

Example 2 (Cont'd)

② Without side effects, it is a matter of efficiency; but with side effects, it is a matter of correctness.

In contrast, consider

```
inline int square(int x) { return x*x; }
```

The argument is evaluated once.

```
square(2+3)  ⇒  square(5)          // efficient
square(++x)  ⇒  ++x,square(x)      // correct
```

For point 3, consider again

```
#define square(x) ((x)*(x))
```

This is flexible, as the argument may be of any numeric type:

```
square(2)    ⇒  ((2)*(2))
square(3.4)  ⇒  ((3.4)*(3.4))
```

In contrast, consider again

```
inline int square(int x) { return x*x; }
```

The argument had better be of type `int`.

```
square(2)    ⇒  4
square(3.4)  ⇒  9
```

To remedy this problem, we may define square functions of other types using other names, say

```
inline double squared(double x) { return x*x; }
```

N.B. C++ offers a better solution.

The other side of the coin: tradeoff between flexibility and safety
Consider

```
#define writeln(n) printf("%d\n",n)
writeln(7.7);  ⇒  ???
```

```
inline void writeln(int n) { printf("%d\n",n); }
writeln(7.7);  ⇒  7
```

③ The dangling else problem

```
         (1)
if (e1) if (e2) s1; else s2;
         (2)
```

C and C++ adopt (2), called the nearest unmatched approach. For (1), we have to write

```
if (e1) if (e2) s1; else; else s2;
if (e1) { if (e2) s1; } else s2;
```

In the example code, if the pair of comment marks is removed, we have to write:

```
if (odd(a)&&odd(b))
   if (a>b) a-=b; else if (a<b) b-=a; else;
else if (even(a)&&even(b)) ...;
```

or

```
if (odd(a)&&odd(b))
   if (a>b) a-=b; else { if (a<b) b-=a; }
else if (even(a)&&even(b)) ...;
```

or

```
if (odd(a)&&odd(b))
   { if (a>b) a-=b; else if (a<b) b-=a; }
else if (even(a)&&even(b)) ...;
```

Version 2

```
unsigned gcd(unsigned a,unsigned b)
{
   unsigned k=0;
   while (even(a)&&even(b)) {
      k++; a>>=1; b>>=1;
   }
   while (a!=b) {
      while (even(a)) a>>=1;
      while (even(b)) b>>=1;
      if (a>b) a-=b; else if (a<b) b-=a;
   }
   return a<<k;
}
```