Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 4

- Constants
- Operators

Integer Constants

- An integer constant like 12345 is an int
  `int foo = 12345;`
- Unsigned constants end with u or U
  `unsigned short bar = 60000u;`
- Leading 0 (zero) indicates an octal (base 8) constant (e.g. $037 = 3 \cdot 8 + 7 = 31$)
  `unsigned short file_permissions = 0666;`
- Leading 0x means hexadecimal (base 16).
  E.g. 0x1f = 31, 0x100 = 256, 0xa = 10
  `unsigned int thirty_two_ones = 0xffffffff;`

Floating-Point Constants

- Floating-point constants contain a decimal point (123.4) or an exponent ($2e\text{-}2 = 2 \cdot 10^{-2} = 0.02$) or both
- Their type is double (8 bytes), unless suffixed
- Suffixes f and F indicate float (4 bytes)
- l or L indicate long double (12 bytes)

```
float two = 2.0; // converted to float
float e = 2.71828182845905f;
long double half = 0.5L;
```


Character Constants

```
char charx   = 'x';      // = 120
char newline = '\n';     // = 10
char digit1  = '0' + 1;  // = 49 ('1')
char hex     = '\x7f';   // = 127
```

- Characters within single quotes e.g. `'A'` `'%'`
- Characters are stored as integers using their ASCII code. E.g. `'0'` is represented as 48 (`man ascii`)
- Escape sequences for non-printable characters:
  `'\n'` newline, `'\''` single quote,
  `'\\'` backslash, `'\a'` bell,
  `'\r'` carriage return, `'\xhh'` hexadecimal code

Enumeration Type

```
enum Month { JAN=1, FEB, MAR, APR,...};
// JAN=1 FEB=2 MAR=3 APR=4 ...
Month x, y;
x = JAN; y = APR;
```

- List of names of integer constants, as in
  `enum Answer { NO, YES };`
- First constant has value 0, next 1, etc.
- Values can be assigned, unassigned successor values are incremented
- Names in different enumerations must be distinct. Values need not.

Arithmetic

`+ - * / %` : result type depends on operands

- `x % y` computes the remainder when `x` is divided by `y` (can not be applied to floating-point values)
- Result of `%` for negative operands is machine dependent, as is the action on overflow
- Division `int / int` rounds towards 0

```
int x1 = x0 + delta;
float c = a * b;

int y1 = 8 / 5;  // = 1
int y2 = -8 / 5; // = -1
int y3 = 8 % 5;  // = 3
```


Mixing integers and floating-point values

- Two int operands : integer operation
  - Careful! `(4/5) = 0 !`
  - Division result is rounded towards 0
- One integer and one floating-point operand
  - int value is silently converted into floating-point
  - then the floating-point operator is executed
  - `(4.0/5) = (4/0.5) = 0.8`
- Two floats: floating-point operation
  - `(4.0/5.0) = 0.8`

If `x` and `y` are integers and you want to compute the "exact" floating-point ratio you need to cast like so:

```
double ratio = ((double)x)/y;
```


Relational Operators

Compare two values with `>` `>=` `<` `<=` `==` `!=`
result type bool

Watch out: == is equality test, = is assigment!

```
bool v1_eq_v2 = (v1 == v2);
bool x_ge_0 = (x >= 0);
bool x = 5;        // != 0 -> true
int a = (1 > 0); // true -> 1
```

bool vs. int

- In integer expressions, bool values are interpreted as 0 (false) or 1 (true)
- int values != 0 are interpreted as true, 0 as false

Useful g++ Flags

```
g++ -Wall -Wuninitialized -W -O test.c
```

reports potentially dangerous but valid C++ code such as

```
if (c = 0) ..  // assignment, not equality test
```

or uninitialized variables (for which data-flow analysis is required which is done when optimizing code: -O)

Logical Operators

```
if (a >= 'a' && a <= 'z').. // a is a lower-case letter
if (a < '0' || a > '9')...  // a is *not* a digit
if (!valid) ...             // true iff valid is false
```

&& || : Boolean shortcut operators

- evaluated from left to right
- evaluation stops when truth-value is known
- && (shortcut and): evaluation of (exp1 && exp2) stops when exp1 evaluates to false
- || (shortcut or): evaluation of (exp1 || exp2) stops when exp1 evaluates to true

! : Boolean negation !false = true, !true = false
(can also be applied to ints: !5 = false, !0 = true)

Increment & Decrement Operators

- ++ : adds 1 to its operand
- -- : subtracts 1
- can be either prefix (++n) or postfix (n++)
- ++n increments n, value of expression is that of n after increment
- n++ increments n, value of expression is original value of n

Examples

```
a++;  ++a; // identical (value not assigned)

int x = 5;
int y = x++;  // y=5, x=6
int z = ++x;  // z=7, x=7

int n = 3;
x = n + n++;  // undefined!
y = y && n++; // DANGER!
```

Watch out! If expression terms have side-effects like ++, evaluation order matters! To be safe, split expression!

Bitwise Operators

Useful for manipulating individual bits or groups of bits in integers

Is fast (parallel computation) and can save space

- ~ : complement
- & : bitwise AND
- | : bitwise inclusive OR
- ^ : bitwise exclusive OR (XOR)
- << : left shift
- >> : right shift

Complement

Think of `int x` as a 32-bit sequence: $x_{31}..x_1x_0$
x_0: least-significant bit, x_{31}: most-significant bit

`z = ~x`

- invert bits (0->1 1->0)
- $z_i = \neg x_i \ (i = 0..31)$

```
 x = 0..[0]1010110
~x = 1..[1]0101001
```


AND

`z = x & y`

- apply operator $\wedge$ (Boolean AND) to pairs of bits
- $z_i = x_i \wedge y_i \ (i = 0..31)$
- $0 \wedge 0 = 0, \ 0 \wedge 1 = 0, \ 1 \wedge 0 = 0, \ 1 \wedge 1 = 1$

```
  x   = 0..[0]1111100
  y   = 1..[1]0101001
  ------------------
 x&y  = 0..[0]0101000
```


OR

`z = x | y`

- apply operator $\vee$ (Boolean OR) to pairs of bits
- $z_i = x_i \vee y_i \ (i = 0..31)$
- $0 \vee 0 = 0, \ 0 \vee 1 = 0, \ 1 \vee 0 = 0, \ 1 \vee 1 = 1$

```
 x   = 0..[0]1111100
 y   = 1..[1]0101001
 ------------------
 x|y = 1..[1]1111101
```

XOR

z = x ^ y

- apply operator $\oplus$ (Boolean XOR) to pairs of bits
- $z_i = x_i \oplus y_i \ (i = 0..31)$
- $0 \oplus 0 = 0, \ 0 \oplus 1 = 1, \ 1 \oplus 0 = 1, \ 1 \oplus 1 = 0$

```
  x   = 0..|0|1111100
  y   = 1..|1|0101001
 ------------------
 x^y  = 1..|1|1010101
```

N.B.: Don't confuse x^y (XOR) with x^y (exponentiation)! There is no exponentiation operator in C++ (have to use pow(x,y) function call instead).

Left Shift

z = x << k

- shift all bits in x k places to the left and set k least-significant bits to 0 ($0 \leqslant k < 32$)
- k most-significant bits are lost
- $z_{i+k} = x_i \ (i = 0..31 - k)$, $z_0..z_{k-1} = 0$
- x << 1 $\equiv$ x * 2

```
  x    = 0..01011111          x    = 0..00001011111
-------------------        ----------------------
x<<1   = 0..10111110        x<<3   = 0..01011111000
```


Right Shift

z = x >> k

- shift all bits in x k places to the right ($0 \leqslant k < 32$)
- k least-significant bits are lost
- x unsigned: clear k most-significant bits
- x signed: clone most-significant bit k times
- $z_{i-k} = x_i \ (i = k..31)$, $z_{31}..z_{32-k}$ = 0 or x_{31}
- unsigned x >> 1 $\equiv$ x / 2

unsigned x:

```
  x    = 11..10111
-----------------
x>>3   = 00011..10
```

signed x:

```
  x    = 11..10111
-----------------
x>>3   = 11111..10
```


Example 1

```
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
```

Example 2

```
#include <iostream>
using namespace std;

// write number in octal format to standard output

int main()
{
  unsigned char n = 65; // 8-bit unsigned integer
  int d0, d1, d2;

  d0 = n & 7;           // all bits but the lowest 3 set to 0
  d1 = (n >> 3) & 7;    // middle 3-bit group of n
  d2 = (n >> 6) & 7;    // left 3-bit group of n

  cout << "octal(" << n << ") = " << d2 << d1 << d0 << endl;
}

output: octal(65) = 101
```


Example 3

```
#include <iostream>
using namespace std;

// write k-th least-significant bit of x to standard output

int main()
{
  unsigned int x = 257, k = 8;
  bool bit = (x & (1 << k)); // shift 1 left to k-th position
                             // and mask all other bits out

  cout << "bit " << k << " of " << x << " = " << bit << endl;
}

output: bit 8 of 257 = 1
```