Lecture 4

- Character constants
- Enumerations
- Variable declarations
- Operators

Floating Point Constants

- **floating point** constants contain a decimal point (**`123.4`**) or an exponent (**`1e-2`**) 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 e = 2.71828182845905;
long double half = 0.5L;
```


Character Constants

```
char newline = '\n';
char digit1  = '0' + 1; //='1'
char hex     = '\x7f';  //=127
```

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

Enumeration Constants

```
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, successor values are incremented
- Names in different enumerations must be **distinct**. Values need not.

Variable Declaration Examples

```
int lower, upper, step;
char c;         // all values undefined
float f = 0;    // initialization
int i = c + 1; // undefined! compiler complains?
const float PI = 3.1415926535;
PI = 0;         // compiler complains!(const)
```


Variable Declarations

- Variables must be **declared** before being used
- Declarations define the type of data to be stored in a variable
- Syntax: <type> <ident1>,<ident2>,... or
  <type> <ident1>=<exp1>,...
- **Value of uninitialized variable is undefined!**
- **const-qualifier** makes variables read-only

Arithmetic and Relational Operators

```
int x1 = x0 + delta;
float c = (a+b)*(a-b);
bool v1_eq_v2 = (v1 == v2);
bool x_ge_0 = (x >= 0);
```

- **+ - * / %**
  - **x % y** computes the remainder when x is divided by y (can not be applied to floating-point values)
  - int / int rounds towards 0
  - sign of % result for negative operands is machine dependent, as is the action on overflow
- **> >= < <= == !=** : result bool
  watch out: == is equality test, = is assigment!
  - in integer expression interpreted as 0 (false) or 1 (true)
  - 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)

Mixing integers and floating point values

- int operator int : integer operation
  - **Careful! (4/5) = 0 !**
  - Division result is rounded towards 0
- One integer and one floating point value:
  - 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**

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): **(exp1) && (exp2)** stops when **exp1** evaluates to false
  - **||** (shortcut or): **(exp1) || (exp2)** stops when **exp1** evaluates to true
- **!** : Boolean negation !false = true, !true = false (can also be applied to ints: !5 = 0, !0 = 1)

Increment & Decrement Operators

```
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 save, split expression!

- **++** : adds 1 to its operand
- **--** : subtracts 1
- can be either prefix (**++n**) or postfix (**n++**)
- **++n** increments n **before** value is used
- **n++** increments it **after** the value is used

Bitwise Operators

- Useful for manipulating individual bits or groups of bits in integers
- & : bitwise **AND**
- **|** : bitwise **inclusive OR**
- ^ : bitwise **exclusive OR**
- << : left shift
- >> : right shift
- ~ : one's complement (unary)

Bitwise Operations

- Work on bit **sequences**
  e.g. int = 32 bits, (b31,...b1,b0)
- ~x: result is all bits in x inverted (0->1 1-> 0)
  x=01010110
  ~x=10101001
- z = x & y , z= x | y , z = x ^ y
  - apply operator (& | ^) to pairs of bits
  - zi = xi & yi (i=0..31), xi | yi , xi ^ yi
- z = x >> y (x << y) shift operations
  - roughly: “shift bits to the right (left) y places”