

MH1402 Algorithms & Computing II

Lecture 2 Scope and Operators

Wu Hongjun

Overview

- **Data overflow**
- **Type conversion and casting**
- **Statement, Statement Block, Scope**
- **Operators**
- **Debugging**

Data Overflow

- Each type of data has a range

char [-128,127] unsigned char [0, 255]

int [-2147483648, 2147483647] unsigned int [0, 4294967295]

double \pm [2.2E-308, 1.79E308]

- For a variable of a certain type, if a value which is not in the range of that type is assigned to that variable, there is overflow.
- Data overflow may cause **serious programming error!**

Overflow of unsigned char

- only the integer value modulo 256 would be stored.
- Reason: the memory size of unsigned char is only one byte.

```
unsigned char x1 = 127; cout << int(x1) << endl;  
unsigned char x2 = 128; cout << int(x2) << endl;  
unsigned char x3 = 129; cout << int(x3) << endl;  
unsigned char x4 = 130; cout << int(x4) << endl;  
unsigned char x5 = 131; cout << int(x5) << endl;  
unsigned char x6 = 254; cout << int(x6) << endl;  
unsigned char x7 = 255; cout << int(x7) << endl;  
→ unsigned char x8 = 256; cout << int(x8) << endl;  
unsigned char x9 = 257; cout << int(x9) << endl;  
unsigned char x10 = 258; cout << int(x10) << endl;
```

```
127  
128  
129  
130  
131  
254  
255  
0  
1  
2
```

- **Overflow of char:**

- only one byte of value would be stored (modulo 256).
- If the value is larger than 127, negative.

```
char y1 = 127; cout << int(y1) << endl;
```

```
→ char y2 = 128; cout << int(y2) << endl;
```

```
char y3 = 129; cout << int(y3) << endl;
```

```
char y4 = 130; cout << int(y4) << endl;
```

```
char y5 = 131; cout << int(y5) << endl;
```

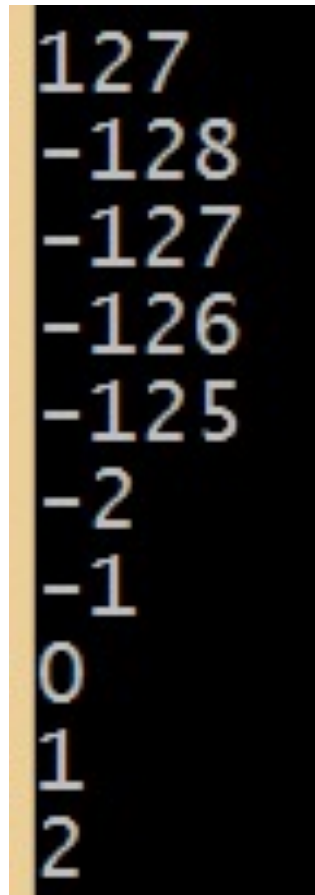
```
char y6 = 254; cout << int(y6) << endl;
```

```
char y7 = 255; cout << int(y7) << endl;
```

```
→ char y8 = 256; cout << int(y8) << endl;
```

```
→ char y9 = 257; cout << int(y9) << endl;
```

```
char y10 = 258; cout << int(y10) << endl;
```



```
127
-128
-127
-126
-125
-2
-1
0
1
2
```

- **int overflow is similar to char overflow, except that the range of int is much larger than that of char**

```
int x = 2000;
```

```
x = 1000*x;  cout << x << endl;
```

```
x = 1000*x;  cout << x << endl;
```

```
x = 1000*x;  cout << x << endl;
```

```
x = 1000*x;  cout << x << endl;
```

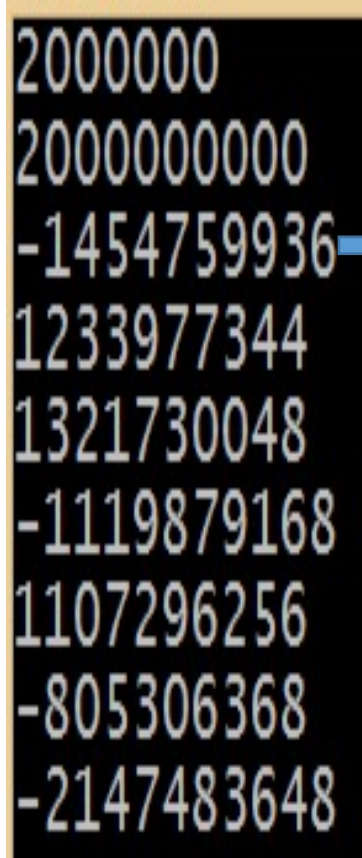
```
x = 1000*x;  cout << x << endl;
```

```
x = 1000*x;  cout << x << endl;
```

```
x = 1000*x;  cout << x << endl;
```

```
x = 1000*x;  cout << x << endl;
```

```
x = 1000*x;  cout << x << endl;
```



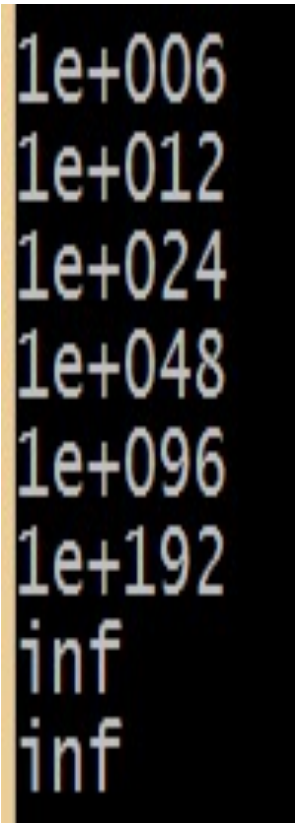
```
2000000
2000000000
-1454759936
1233977344
1321730048
-1119879168
1107296256
-805306368
-2147483648
```

Overflow:

$2,000,000,000,000 > 2^{32}-1;$
 $2,000,000,000,000 \bmod 2^{32}$
 $= 2840207360 > 2^{31}$
so this integer is negative.

double (floating-point) overflow (different from integer overflow)

```
double w = 1000.0;  
w = w*w; cout << w << endl;  
w = w*w; cout << w << endl;  
w = w*w; cout << w << endl;  
w = w*w; cout << w << endl;  
w = w*w; cout << w << endl;  
w = w*w; cout << w << endl;  
w = w*w; cout << w << endl;  
w = w*w; cout << w << endl;
```



```
1e+006  
1e+012  
1e+024  
1e+048  
1e+096  
1e+192  
inf  
inf
```

Overflow: $E394 > 1.79E308$

The symbol `inf` stands for infinity

On computer, for a floating-point, if the exponent bits are all 1, and the fractional bits are all 0, the computer treats the value of the floating point as `inf`.

Type Conversion and casting

- How to convert one data type into another type?
- Type conversion in data assignments:
 x = y; // suppose that x, y are different types,
 // the value of y is converted to the type of x automatically,
 // then the converted value is assigned to x

int x;

x = 23.45; // the same as x = 23; the fractional part gets truncated.

Type conversion and casting

- Type conversion with an operator (we use addition operator to illustrate) :

$x + y$

if x and y are the same data type, then $(x+y)$ is that data type;

if x and y are different types, for example:

x is int, y is double,

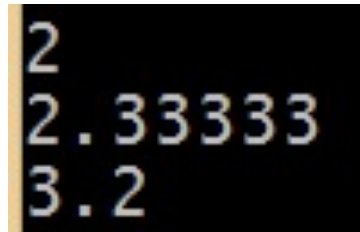
then the value of x is represented as double in the addition,

and $(x+y)$ is double

```
cout << 7/3;
```

```
cout << 7/3.0;
```

```
cout << 7/3 + 1.2;
```



```
2  
2.33333  
3.2
```

Type conversion and casting

- **Type casting: actively change one type of data into another type**

- **Examples:**

```
int x = 7, y = 2;
```

```
cout << double(x)/2 << endl; // prints 3.5 to the screen
```

```
// the data type and value of x are not affected
```

```
cout << int('A') << endl; // prints 65 to the screen
```

In C++, we use `int(x)` `double(x)`

In C, we use `(int)x` `(double)x`

The type casting in C can be used in C++

Statement and Statement Block

- A statement is a unit of code that does something, and it is a basic building block of a program

- A C++ statement must be **ended with semicolon ;**

Examples:

```
int x;           // a statement to declare an integer variable x
```

```
cout << 245;    // a statement to print 245 on the screen
```

- Note that the line starting with `#` is not a program statement, so semicolon is not needed

Examples: `#include <iostream>` //use the library iostream

```
#define WEIGHT 3500 //define WEIGHT as a constant
```

Statement and Statement Block

- **Statement block:** You can enclose several (or many) statements between a pair of curly braces, { }, in which case they're referred to as a statement block.

Example of a statement block:


```
{           // { indicates the beginning of a statement block
    x = 3;
    y = 45;
    z = x + 32;
}           // } indicates the end of a statement block
```

Scope

- The scope of an identifier (for example, variable name) is that part of the program where it can be used.
- Variables cannot be used before they are declared, so their scopes begin where they are declared.
- If a variable is declared within a statement block, then its scope **starts from the declaration, and ends at the end of that statement block**
 - Identifying the scope of a variable is important in C++

Scope

```
int main ()  
{  
    x = 45;           // error: x was not declared in this scope  
    int x;  
    x = 45;  
    {  
        x = 23;       // correct.  
    }  
    cout << x;  
    int y = 5;  
    return 0;  
}
```



Scope of x

Scope

```
int main ()  
{
```

```
{
```

```
    int x = 23;  
    int y = 34, z;  
    y = x + y;  
    z = y;
```

```
}
```

```
    cout << x;  
    y = 5;  
    return 0;
```

```
}
```



Scope of x

```
// error: x was not declared in this scope  
// error: y was not declared in this scope
```

Scope

```
int main ()  
{  
    int x;  
    int y;  
    int x = 45;    // error: re-declaration of x  
    cout << x;  
    return 0;  
}
```


Operators

http://en.wikipedia.org/wiki/Operators_in_C_and_C++

- An operator is a symbol that performs an action
- Arithmetic Operator
 - +, -, *, / and parentheses have their usual mathematical meanings
 - % is the integer modulus operator, and it gives the remainder of a division. For example, 6%5 yields 1 and not 1.2

Note: A C++ operator for exponentiation does not exist, **different from MATLAB**

For example: `int x = 2^3;` does not assign 8 to x

(^ is the bitwise XOR operator in C++)

To perform exponentiation in C++, we should use the function `pow` declared in `cmath` (more on it when we learn C++ functions)

Operators

- **Arithmetic Operator (cont.)**

- **Increment operator ++**

- Adding 1

- That is the reason this programming language is called C++

post-increment (example: c++) vs. pre-increment (example: ++c)

**post- version: the current variable value is used in the surrounding context,
then add its value by 1;**

**pre- version: add the value of the current variable by 1;
then use the modified value in the surrounding context.**

Operators

- **Arithmetic Operator (cont.)**
 - **Increment operator ++ (cont.)**

```
int main ()
{
    int m, n;
    m = 44;
    n = ++m; // the value of m is incremented, then the value of m is assigned to n
    cout << m << endl; 45
    cout << n << endl; 45

    m = 44;
    n = m++; // the value of m assigned to n, then the value of m is incremented
    cout << m << endl; 45
    cout << n << endl; 44
}
```

Operators

- **Arithmetic Operator (cont.)**
 - **Decrement operator --**
 - Subtracting 1
 - **Post-decrement (example: `c--`) vs. pre-decrement (example: `--c`) :**
similar to that of increment

Operators

- **Composite assignment operators:** `+=` `-=` `*=` `/=` `%=`

Examples:

```
int m = 3,n=4,p=5,q = 6;
```

```
m += n;      // m = m + n;
```

```
m *= n + p;  // m = m*(n+p);
```

```
m /= n*p+q;  // m = m/(n*p+q);
```

```
m -= n;      // m = m - n;
```

Operators

- Bitwise operators http://en.wikipedia.org/wiki/Bitwise_operations_in_C
 - Manipulate the binary representation of numbers
 - & ^ | ~
- Shift operators http://en.wikipedia.org/wiki/Bitwise_operations_in_C
 - << >>

(Bitwise operators and shift operators are not the focus of this course, but I strongly encourage the students who are good at programming to learn these operators)

- Comparison & Logical Operator ➔ next

Comparison operators

- **Comparison operators: test a relation between two values, and return a Boolean value of either true (1) or false (0)**

(recall that the data type of this Boolean value is bool)

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

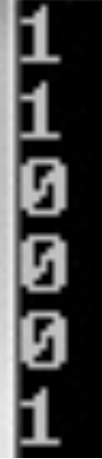
(Note that in MATLAB, not-equal-to operator is `~=`)

Comparison operators

- Example:

```
int x = 6, y = 2;
```

```
cout << (x > y) << endl; // x > y returns true
cout << (x >= y) << endl; // x >= y returns true
cout << (x < y) << endl; // x < y returns false
cout << (x <= y) << endl; // x <= y returns false
cout << (x == y) << endl; // x == y returns false
cout << (x != y) << endl; // x != y returns true
```



1
1
0
0
0
1

Comparison operators

- It is **risky to compare two floating-points** (for example, double) when their values are very close (or almost the same)
 - Reason: **the round-off error may cause programming error**
- Example of round-off error in comparison:

```
double x = 1.0/351.0;
```

```
x = 1.0 - 351.0*x;
```

```
cout << (x > 0) << endl; // return false
```

```
cout << (x < 0) << endl; // return true
```

```
cout << (x == 0) << endl; // return false
```

```
cout << (x != 0) << endl; // return true
```

```
0  
1  
0  
1
```

```
// The reason is:
```

```
cout << "The value of x is " << x << endl;
```

```
The value of x is -4.98733e-017
```

Logical Operators

- The *logical operators operate on Boolean type values* (true or false)
 - The logical operators return true or false
- Logical operators: `&&` `||` `!`
- logical AND operator `&&`
 - it operates according to the following rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Logical Operators

- Logical OR operator `||`
 - it operates according to the following rules of logic:

a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

Logical Operators

- **Logical NOT operator !**
 - It operates according to the following rule of logic

a	!a
true	false
false	true

- **Logical NOT operator is a unary operator**
 - Unary operator takes only one argument
- In MATLAB, this operator is **~**

Logical Operators

- Examples:

```
int x = 6, y = 2;
```

```
cout << ( (x > y) && false ) << endl; // return false
```

```
cout << ( (x > y) && (y > 0) ) << endl; // return true
```

```
cout << ( (x < y) || (y > 1) ) << endl; // return true
```

```
cout << ( !(x == y) ) << endl; // return true
```



Logical Operators

- Another example:

```
int t = 2;
```

```
cout << ((t > 3) || 5) << endl; // returns true
```

1

Why?

Logical OR operator takes two Boolean arguments, one argument is 5. In C++, Boolean false is represented by a value 0; **anything that is not zero (integer/floating-point) is Boolean true.**

Logical Operators

- A **common programming mistake** is to confuse the equality “==” and assignment “=” operators
 - Example: we want to test whether r is equal to 9, and to test whether w is less than 6; if either of these two tests is true, return true

```
int r = 3, w = 7;  
cout << ( (r = 9) || ( w < 6) ) << endl;  
cout << "The value of r is " << r << endl;
```

```
1  
The value of r is 9
```

Why? The reason is that `r = 9` assigns the value 9 to `r`, so we are evaluating `(9 || (w < 6))`. A nonzero (here 9) is always treated as Boolean value true.

Order of Comparison and Logical Operators

- The relative precedence levels of comparison operators and logical operators

()	Highest
< <= >= >	
== !=	
&&	
	Lowest

- Examples:

(4 > 3 3 > 5 && 7 > 8)	returns true	why?
((4 > 3 3 > 5) && 7 > 8)	returns false	

- It is always a good idea to use brackets to enforce the order of computation if you are uncertain about the order of operators

Debugging

- Two types of C++ programming errors: compilation errors and runtime errors.
- Compilation errors are problems raised by the compiler, generally resulting from violations of the syntax rules or misuse of types. These are often caused by typos and the like.
 - Example: `int x; X = 4;`
 - Relatively easy to fix
- Runtime errors are problems that you only spot when you run the program: your program doesn't do what you wanted it to.
 - Usually more tricky to catch, since the compiler won't tell you about them.
- **Debugging is an important programming skill**
 - Learn it in practice

Debugging

- **Debugger (it will not be tested)**

- Suppose that your program can get compiled, but it does not work as expected (it gives wrong outputs or it crashes), you may debug your program using debugger. An introduction on debugger is available at:

<https://en.wikipedia.org/wiki/Debugger>

- You may use the Code::Blocks debugger to debug your program:

http://wiki.codeblocks.org/index.php/Debugging_with_Code::Blocks

The Code::Blocks debugger does not work for single file, so you need to create a project. To create a new project, you may read the following link:

http://wiki.codeblocks.org/index.php/Creating_a_new_project