CS 101: Computer Programming and Utilization

Shivaram Kalyanakrishnan
(Abhiram Ranade's slides, borrowed and edited)
Lecture 5

This Lecture

- Rules regarding expressions
- Increment and decrement operators (++, —)
- Blocks and scope

Rules regarding expressions

- Multiplication must be written explicitly.

```
S = u t + 0.5 a t t;  // not legal
```

- Multiplication, division have higher precedence than addition, subtraction
- Multiplication, division have same precedence
- Addition, subtraction have same precedence
- Operators of same precedence will be evaluated left to right.
- Parentheses can be used with usual meaning.
- Spaces can be put between operators and values as you like

```
s=u*t+0.5*a*t*t;   s = u*t + 0.5*a*t*t;  // both OK
```

Examples

```
int x=2, y=3, p=4, q=5;
int r, s, t, u;
r = x * y + p * q;
s = x * (y + p) * q;
t = x - y + p - q;
u = x + w;
```

Evaluating “`A op B`” when A, B have different types

- If `A,B` have different data types: then they will be converted to “more expressive” data type among the two:
  - `int/short/unsigned int` are less expressive than `float/double`
  - shorter types are less expressive than longer.
- The operation will then be performed, and the result will have the more expressive type.

Implication of limited precision

- `float w,y=1.5, avogadro=6.022e23;`
- `w = y + avogadro;`

- "Actual sum" : 602200000000000000000001.5
- Sum will have to be stored in variable `w` of type float.
  - But `w` can only accommodate 23 bits, or about 7 digits.
  - Thus all digits after the 7th will get truncated.
  - To 7 digits of precision `avogadro` is same as y `+avogadro`.
- `w` **will receive the truncated value, i.e.** `avogadro`.
- This addition has no effect!

Other aspects

- `Overflow (due to limited size)`
- `inf (division by 0)`
- `nan (0/0)`

This Lecture

- Rules regarding expressions
- Increment and decrement operators (++, —)
- Blocks and scope

Some additional operators

- The fragment "`i = i + 1`" appears very frequently, and so can be abbreviated as "`i++`".
- ++ : increment operator
- Similarly we may write "`j--`" which means "`j = j - 1`"
- -- : decrement operator

Intricacies of ++ and --

- ++ and -- can be written after the variable, and this also cause the variable to increment or decrement.
- Turns out that expressions such as k = ++i; and k = i++; are legal in C++ and produce different results.
- Such assignments are described in the book for completeness.
- But they are somewhat hard to read, and so it is recommended you do not use them.
- Similarly with --.

Compound assignment

- The fragments of the form “sum = sum + expression;” occur frequently, and so C++ allows them to be shortened to “sum += expression;”
- Likewise you may have *=, -=, …
- Example

```
int x=5, y=6, z=7, w=8;
x += z;    // x becomes x+z = 12
y *= z+w; // y becomes y*(z+w) = 90
```

- Note: `z, w` do not change.

This Lecture

- Rules regarding expressions
- Increment and decrement operators (++, —)
- Blocks and scope

Blocks and Scope

- Code inside {} is called a block.
- Blocks are associated with repeats, but you may create them otherwise too.
- You may declare variables inside any block.

New summing program:

- The variable `term` is defined close to where it is used, rather than at the beginning. This makes the program more readable.
- But the execution of this code is a bit involved.

```
// The summing program
// written differently

main_program{
   int s = 0;
   repeat(10){
     int term;
     cin >> term;
     s = s + term;
   }
   cout << s << endl;
}
```

How definitions in a block execute

Basic rules

- A variable is defined/created every time control reaches the definition.
- All variables defined in a block are destroyed every time control reaches the end of the block.
- “Creating” a variable is only notional; the compiler simply starts using that region of memory from then on.
- Likewise “destroying” a variable is notional.
- New summing program executes exactly like the old, it just reads different (better!).

Scope and Shadowing

- Variables defined outside a block can be used inside the block, if no variable of the same name is defined inside the block.
- If a variable of the same name is defined, then from the point of definition to the end of the block, the newly defined variable gets used.
- The new variable is said to "shadow" the old variable.
- The region of the program where a variable defined in a particular definition can be used is said to be the scope of the definition.
- Why do we care:
  - In a single English language document you might write "Let x denote" in several places, with the understanding that the x on page 5 is different from the x on page 37.
  - If you do not have an intervening "Let x denote .." the two xs might be same.
  - Something similar happens while writing large programs

Example

```
main_program{
  int x=5;
  cout << x << endl;   // prints 5
  {
    cout << x << endl; // prints 5
    int x = 10;
    cout << x << endl; // prints 10
  }
  cout << x << endl; // prints 5
}
```

What if `int x = 10;` was `x = 10;` ?

This Lecture

- Rules regarding expressions
- Increment and decrement operators (++, —)
- Blocks and scope