What we have learnt so far in CS1

Prof. Zhang

February 27, 2014

1 Nuts and Bolts of programming

We have learnt different parts that made up a C++ program. Please review them often.

1.1 C++ Program Structure

Below is a sample C++ program, with comments inside to explain the key parts.

```
/* This is a sample program
   Author: Xiaolan Zhang
   Last modified on: 2/8/2014
*/
#include <iostream>  //include this file in order to use cin and cout
using namespace std;

int main()   // the start of main function
{
   //add comment whenever you think it will help you or other programmers to
   //understand your code better
   statement1;   //; ends a statement
   statement2;

   statement3
   statement3 (cont'd)
   statement3 (cont'd);   //a statement can span multiple lines ...

   //comment for the next group of statements...
   ...
   statementn;
}
```

You can break a long statement into multiple lines, only the last line ends with semi-colon:

```
cout <<"The total number of students is " << studentNum
     <<"Avarage age is " << age  <<"\n";

double amount_due = pizza12_num * PIZZA_12_PRICE     //subtotal 1
   + pizza14_num * PIZZA_14_PRICE;   //subtotal 2
```

Note that you CANNOT break a statement inside a quoted string, or within a variable name:

```
cout <<"Hello world,
       my name is C++\n";  //WRONG !!

double amount_due = pizza12  //WRONG!!
     _num * PIZZA_12
     _PRICE;
```

1.2 Constants

There are two types of constants.

- **literal constant**. When you directly write the constant value in your code, you are using literal constants. For example,

  ```
  if (response=='Y')   //constant chars are quoted with single quotation mark
    cout <<"You choose Y\n";  //the string constant are quoted with double quotation mar

  area = 3.14*radius*radius;  //3.14 is a literal constant

  leapyear= true; //true and false are the two values for bool type
  ```

- **named constant**, or **constant variable** is used to increase readability of our programs. For example,

  ```
  const double PIZZA_12_PRICE=13.99; //PIZZA_12_PRICE is a named constant
  ```

1.3 Variables

- Variable: (name, type, value)-tuple, use a box labeled with name, filled with its value
- Variable declaration:

```
type name=initial_value;
```

- Basic type and user defined type
  - Each type uses a fixed amount of memory (sizeof function returns the number of bytes in a type); supports a set of operations
  - Basic type: int, short, long, float, double, char, bool, long long,
  - User defined type (need to include header files): string,
- Variable scope: local scope (block scope), global scope, eclipse effect
- Conversion between type: When assigning a floating value to int-type variable, some information is lost, i.e., the decimal parts of the value are thrown away:

  ```
  int a=13.999;   // a value will be 13

  //To get rounding behavior in the conversion:
  int a=13.999+0.5;  //a value will be 14
  int b=13.49+0.5; //b value will be 13
  ```

1.4 Expressions

An *expression* is made up of operators and operands. C++ *operators* are listed below, and *operands* can be variables, constants, other expressions. For example, the following are expressions.

```
2+3, (a+sqrt(b))/3,
i=0, x%2 == 0
```

We have learnt two kinds of **constants**:

- literal constant: in which we write the value of the constant directly in the code, for exmaple,

  ```
  area = 3.1415*radius*radium;   //3.1415 is a literal constant
  cout <<"area is" << area << "\n";
  // here "area is" and "\n" are two constant strings
  ```

- named constant (i.e., constant variable), for example,

  ```
  const double PI=3.1415; //declare and initialize a named constant (use capital letters
  area = PI*radius*radius;  //use the named constant.
  ```

The following are operators that we have learnt so far:

- arithmetic operators:

  ```
  +, -, *, /, % (only for int, long, short)
  ```

  Note that for division (/) operation, if both operands are **int** or **long** types, it stands for **integer division**, the division where the fractional part is discarded. On the other hand, if one or both operands are **double** pr **float** type, then it stands for regular division, where the fractional part is kept. For example, the following expression for converting Fahrenheit to Celsius:

  ```
  double F=100,C;
  C = 5/9*(F-32);  // this will assign 0 to C, as 5/9 is 0
  ```

  To fix this problem, you can do the following:

  ```
  double F=100,C;
  C = 5.0/9*(F-32);  // 5.0 is not an int, so 5.0/9 is not integer division...
  ```

- assignment operator: =
- shorthand operator: ++, –, +=, -=, *=, /=
- comparision operators (i.e., relational operators): $<, <=, >, >=, ==, ! =$
- boolean operators:

  ```
  &&, ||, !
  ```

The complete version of C++ operators precedence can be found at:

`http://en.cppreference.com/w/cpp/language/operator\_precedence`

We have so far learnt the following:

Precedence	Operators	Description	Association
1	$++, --$	suffix increment and decrement	left-to-right
2	$++, --$	prefix increment and decrement	right-to-left
	!	logic NOT	
3	* / %	multiplicaiton, division, and remainder	left-to-right
4	+ -	Addition and subtration	left-to-right
5	$<\leq>\geq$	Relation operators	left-to-right
6	$==, ! =$	relational operators	left-to-right
7	&&	logical AND	left-to-right
8	||	logic OR	left-to-right
9	=	assignment	right-to-left

For examples, let's consider the expression $1 < a < 10$. A common mistake made by beginngers is to interpret this as meaning "a is larger than 1 and smaller than 10". Now let's use the above table to dissect this expression:

The relational operator $<$ is associated from left to right, so we first evaluate $1 < a$, which gives a boolean value. This value (true is 1, false is 0) is then compared to 10, which is always true.

Now, let's take a look at another expression $a = b = 10$. This expression assigns value 10 to both a and b. Why?

The assignment operator $=$ is associated from right-to-left, therefore 10 is assigned to b, and the subexpression $b = 10$ is evaluated to 10, this value is then assigned to a.

1.5 Statement

C++ program is made up of statements. All statements end with a semicolon (;). All statements in the *main*() body is executed in order.

The following is a list of statements we have learnt:

- Variable declaration statements: for example,

  ```
  double total_due;
  ```

- Expression: for example, $a = 10;$, $cout << "Helloworld";$

- Empty statement: the following is a statement that does nothing

  ```
     ;
  ```

  It's useful in some situation when used in a loop or if/else statement, for example

  ```
  if (x>0)
     ; //if x>0, do nothing
  else
     cout <<"wrong value\n";   //otherwise, display error info.
  ```

- Block statement: group a sequence of statments together to form a single statement

  ```
  {
     statement_1;
     statement_2;
     ...
     statement_n;
  }
  ```

- **if-else** statement and **if** statement. **if-else statement** is used to implement a two-way branch.

  ```
  if (boolean_expression)  //1. boolean_expression is first evaluated
     yes_statement   //2. This yes_statement is executed if the boolean_expression is tr
  else
     no_statement   //3. This no_statement is executed if the boolean expression is fals

  next_statement   //4. Both branches merge here...
  ```

  If statement is used to implement *conditional* execution.

  ```
  if (boolean_expression)
     yes_statement  //this statement is executed if boolean expression is true
  ```

  Note that the yes_statement and no_statement above can be any statement, for example, it can be an **if statement** or **if-else statement** itself. So we might have:

  ```
  if (boolean_expression_1)
     if (boolean_expression_2)
         yes_statement_2;  // this yes_statement_2 can be an if statement, or if/else st
     else
         no_statement_2;
  else
     if (boolean_expression_3)
         yes_statement_3;
  ```

  This is like a nested doll, and the nested level can be very deep, 3, 5, 10.

- **for** statement, **while** statement, and **do-while** statement

- **return** statement and **break** statement

  return statement leaves a function. For now, all our programs are written in the **main** function, so whenever you have a **return** statement, it will take you out of the **main** function, and therefore your program.

  ```
  return 0; //exit from current function/program...
  ```

  break statement leaves a loop, even if the condition for the loop's end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. If **break** statement is used in a nested loop, it leaves the inner-most loop only.

```
while (true)
{
   cout <<"Guess a number:";
   cin >> guess;
   if (guess==secretnum)
       break;
}
```

2 Coding Style

Please refer to the document "How programs are graded?" for coding styles.

3 When things go wrong...

We have learnt that C++, as a programming language, has certain grammer rules (just like English). For example, the following are grammar rules that you have learnt:

- variable names need to start with English letters or underscore, and cannot contain space.
- Reserved words (such as main, int, return) cannot be used for variable names.
- Variables must be declared before referenced (used).
- Statements must end with semicolon (;) (except block statement)

If your program breaks grammar rules, compiler will be able to report them, such errors are therefore referred to as *syntax errors*, or *compilation errors*.

Sometimes your program passes compilation, but the execution result is wrong because you have expressed your intentions wrong. This is usually caused by so called *logic errors*. For example, the following code contains a logic error:

```
if (x=y)   //from the context, we can see that this actually should be x==y
  cout <<"x equals to y\n";
else
  cout <<"x does not equal to y\n";
```

The following are common logic errors:

- Confusing = and ==
- *Accidental* Empty loop body or yes_statement
- Forget to use {} to group statements

- Access variables before it is initialized with a value

The last category of errors also happen at runtime (i.e., when you execute the program), it's sometimes called *fatal error*, or *runtime errors*. For example,

```
int scores=500;
int num = 0;
int avg = scores / num;
```

The program will crash, and reports error messages such as

```
floating exceptions
```