Lecture #3 | C/C++: Variables and control flows

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Today's topic

- Basic language features
  - Values and statements
  - Data types
  - Variables
  - Operators
  - Control flows: `if`, `while`, `do`, `for`, `switch/case`
  - Functions (definition & declaration)
  - Header files

Variables

- Variable declaration (type variable_name)

```
double d;
int i;
```

- A variable names is
  - Composed of alphabets, numbers, _
  - Starting with alphabets or _
  - Case-sensitive
- Assignment

```
d = 2.3;
i = 2 * 2;
i = 2 * 2.2; // evaluates to 4 (implicit conversion)
```

Variables (cont.)

- A variable can be initialized when it is declared with = or {}
  - = dates back to C, more commonly used
  - {} is new for all data types; it is more flexible and prevents you from conversions that lose information (since C++11)
- Examples

```
double d1 = 2.3;
double d2 {2.3};
double d3 = {2.3}; // = is redundant

int i1 = 7.2;   // i1 becomes 7
int i2 {7.2};   // error: compile time error
```

Operators (not comprehensive)

- Operators act on expressions to form a new expression (e.g., `(4 + 2) / 3`)
- Types of operators
  - Arithmetic: `+, -, *, /, %,`
  - Increment, decrement: `++, --`
  - Comparison (or relational): `==, !=, >, <, >=, <=`
  - Logical: `&&, ||, !`
  - Bitwise: `&, |, ^`
  - Assignment: `=, +=, -=, *=, /=, ...`
  - Member access: `[], *, &, ->, .`
  - Other: `sizeof()`, conditional (`? :`), comma (`,`)

Increment & decrement operators

- Pre-increment: increment the given variable first, and then return the new value
- Post-increment: return the current value (copy it if needed), and increment the given variable

```
int  x, y;
// Increment operators
x = 1;
y = ++x;    // x is now 2, y is also 2
y = x++;    // x is now 3, y is 2
// Decrement operators
x = 3;
y = x--;    // x is now 2, y is 3
y = --x;    // x is now 1, y is also 1
```

Logical operators: &&, ||, !

- The position of operands matters when they have side effects

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

int main(void)
{
    int result, x = 1, y = 2; // not recommended
    ++x > 2 && ++y > 2;
    cout << "x=" << x << ", y=" << y << endl;
    ++x > 2 || ++y > 2;
    cout << "x=" << x << ", y=" << y << endl;
}
```

`sizeof()`: returns the size of a variable/data type

```
#include <iostream>
using namespace std;
int main(void)
{
    cout << "char: " << sizeof(char) << endl;
    cout << "int: " << sizeof(int) << endl;
    cout << "long: " << sizeof(long) << endl;
    cout << "long long: " << sizeof(long long) << endl;
    cout << "float: " << sizeof(float) << endl;
    cout << "double: " << sizeof(double) << endl;
    cout << "long double: " << sizeof(long double) << endl;
}
```

```
char: 1
int: 4
long: 8
long long: 8
float: 4
double: 8
long double: 16
```

This result may differ system by system!!!

Conditional operator: ? :

- *condition* ? *X* : *Y* → If *condition* is true, evaluates to *X*; otherwise *Y*

```
int result;
int i = 5;
int j = 3;

result = i > j ? i : j; // result takes 5

// equivalent code w/o conditional operator
if (i > j)
    result = i;
else
    result = j;
```

Control flows: `if`

- `if (condition) statement;`
- `if (condition) statement; else statement;`

```
int i = 2;
int j = 3;

if (i > j)
    cout << "i is greater than j" << endl;
else {
    cout << "i is NOT greater than j" << endl;
    cout << "i is either smaller than or equal to j" << endl;
}
```

Blocks

- A sequence of statements delimited by curly braces ({ and }) is called a *block* or a *compound statement*
- Since control flows in C/C++ takes only one statement, you need to use blocks to have more than one statement
- Note: Unlike python, white spaces (or indentations) are ignored by compilers. But it is STRONGLY recommended put adequate and consistent indentations in your code!

Control flows: `while` & `do`

- while (*condition*) *statement*;
  → *statement* may not be executed
- do *statement* while (*expression*);
  → *statement* is executed at least once

```
i = 3
while (i > 0) {
    cout << "i=" << i-- << endl;
}
do {
    cout << "i=" << i++ << endl;
} while (i < 5);
```

Control flows: for

- for (*init-statement*; *condition*$_{opt}$; *expression*$_{opt}$) *statement*;

```
for (i = 0; i < 5; i++)
    cout << "i=" << i << endl;

cout << "----" << endl;

// equivalent code
i = 0;
while (i < 5) {
    cout << "i=" << i << endl;
    i++;
}
```

Control flows: switch/case

```
switch (i) {
case 1:
    cout << "i is 1" << endl;
    break;
case 2:
case 3:
    cout << "i is 2 or 3" << endl;
    break;
default: // default is NOT required
    cout << "i is neither 1, 2, nor 3" << endl;
}
```

Functions

- What is function in C/C++?
  - A reusable sequence of statement(s) designed to a particular job
- Why define your own function?
  - Readability: `sqrt(5)` is clearer than copy-pasting in an algorithm to compute the square root
  - Maintainability: To change the algorithm, just change the function (vs changing it everywhere you ever used it)
  - Code reuse: Lets other people use algorithms you've implemented
- `main()` is called (or invoked) after initialization of non-local objects, i.e., the entry point of program execution

Function definition and declaration

- Function definition

```
return_type function_name(parameters) {
    statement;
}
```

- *parameters*

```
void
[data_type1 param1[, data_type2 param2[, ...]]]
```

- Function declaration: parameter names can be omitted

```
return_type function_name(parameters);
```

Example: function

```
int raise_to_power(int base, int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; ++i)
        result *= base;
    return result;
}

int main(void)
{
    cout << "3^4 is " << raise_to_power(3, 4) << endl;
    return 0;
}
```

Function overload

- When two or more different declarations are specified for a single name in the same scope, that name is said to be *overloaded* (only in C++*)
- Overloaded functions should have different parameters, i.e., number/type of parameters; functions with different return types cannot be overloaded

```
void print(int arg) {
    cout << "int value:" << arg << endl;
}

void print(double arg) {
    cout << "double value:" << arg << endl;
}

int main(void) {
    print(1);
    print(1.0);
}
```

* C11 supports similar function using _Generic()

Example: function declaration

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

int cube(int x) {
    return x * square(x);
}
int square(int x) {
    return x * x;
}
void main(void) {
    cout << "2^3" << cube(2) << endl;
}
```

```
$ g++ func_cube_square.cpp
func_cube_square.cpp:5:16:
error: use of undeclared
identifier 'square'
    return x * square(x);
               ^
1 error generated.
```

Example: function declaration (cont.)

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

int cube(int x) {
    return x * square(x);
}
int square(int x) {
    return x * x;
}
void main(void) {
    cout << "2^3" << cube(2);
    cout << endl;
}
```

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

int square(int x);
int cube(int x) {
    return x * square(x);
}
int square(int x) {
    return x * x;
}
void main(void) {
    cout << "2^3" << cube(2);
    cout << endl;
}
```

Header files: mostly function and class declarations*

```
/* func_cube_square.h
 * with function prototypes
 */
int cube(int);
int square(int);
```

- Only data types of return value and parameters matter
- But it is recommended to provide "meaningful" parameters names

```
// func_cube_square.cpp
#include <iostream>
#include "func_cube_square.h"
using namespace std;

int cube(int x) {
    return x * square(x);
}
int square(int x) {
    return x * x;
}
void main(void) {
    cout << "2^3" << cube(2) << endl;
}
```

* or prototypes

Scope

- Scope: a portion of program text that a particular name (e.g., variable, function) is valid
- Global variable: end of a file (or whole files when used with `extern`)
  - Using global variables is DISCOURAGED!!!
  - Initialized when a program begins
  - Destroyed when a program exits
- Local variable
  - Valid until the end of block or function
  - Allocated (and initialized) when a block/function starts
  - Destroyed when a block/function exits

Scope: global variable

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

int n_count = 6;
void func(void)
{
    n_count++;
}
int main(void)
{
    cout << "n_count=" << n_count << endl;
    func(); func();
    cout << "n_count=" << n_count << endl;
}
```

- Global variable can be accessed from everywhere
- If no assignment is provided, global variables are initialized as the basic value of the type, e.g.,
  - char/int/long/…: `0`
  - double/float/…: `0.0`
  - string: null string

Scope: local variable

- Local variable can be accessed within only its local scope
- Local variable is destroyed when a block/function exits, thus the value in local variable is NOT preserved for the next loop/invocation
- Note: in python, the scope of a local variable range from its definition to the end of the function

```
int n_calls;
int pow(int base, int exponent)
{
    n_calls++;
    int result = 1;
    for (int i = 0; i < exponent; i++)
        result *= base;
    return result;
}
int max(int num1, int num2)
{
    n_calls++;
    int result = num1 > num2 ? num1 : num2;
    return result;
}
int main(void)
{
    int result = max(pow(2,10), pow(10, 3));
    cout << result << " " << n_count << endl;
}
```

Scope: local variable (cont.)

```
int main(void)
{
    int result = 1024;
    {
        int result = 42;
        cout << "result=" << result << endl;
    }
    cout << "result=" << result << endl;
    for (int i = 0; i < 2; i++)
    {
        int result = 0;
        result++;
        cout << "result=" << result << endl;
    }
}
```

Scope: recursion

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}

int main(void)
{
    cout << "5!=" << factorial(3) << endl;
}
```

- Whenever a function is called, a new scope is generated
- The same rule holds even though a function calls itself (recursion)
- This is the case for most programming languages (C, C++, python, Java, …)

Reading List

- Learn C++
  - Chapter 1: 3-4, 7, 9
  - Chapter 2: 1-4
  - Chapter 4: 1a, 2, 2a, 3a, 3b
    (skip linkage-related explanations)

ANY QUESTIONS?