[CSED233-01] Data Structure

Basic C++ Programming (2)

Jaesik Park

POSTECH

For Loop

- Many loops involve three common elements:
  - an initialization
  - a condition under which to continue execution
  - and an increment to be performed after each execution of the loop's body
- A *for loop* conveniently encapsulates these three elements.

`for` (*initialization* ; *condition* ; *increment*)
loop body statement

For Loop

- Here is a simple example, which prints the positive elements of an array, one per line
  - `i` was declared as **`int`** `i = 0`
  - Before each iteration, the loop tests the condition "`i < NUM ELEMENTS`" and executes the loop body only if this is true
  - At the end of each iteration the loop uses the statement `i++` to increment the loop variable `i` before testing the condition again

```
const int NUM ELEMENTS = 10;
double b[NUM ELEMENTS] = {1.34,2.58,-3.61,-4.60,-
5.03,5.23,4.38,3.34,-2.46,-1.56};

for (int i = 0; i < NUM ELEMENTS; i++) {
      if (b[i] > 0)
            cout << b[i] << '\n';
}
```

Break Statements

- C++ provides statements to change control flow
- A break statement is used to "break" out of a loop statement
  - When it is executed, it causes the flow of control to immediately exit the innermost loop.
- The following example:
  - Sums the elements of an array until finding the first negative value.

```
int a[10] = {1,2,-3,-4,-5,5,4,3,-2,-1};
int sum = 0;

for (int i = 0; i < 10; i++) {
    if (a[i] < 0) break;
    sum += a[i];
}

cout << "sum : " << sum << endl;
```

Break Statements

- The following example - equivalent
  - Sums the elements of an array until finding the first negative value.

```
int a[10] = {1,2,-3,-4,-5,5,4,3,-2,-1};
int sum = 0;

for (int i = 0; a[i] > 0 && i < 10; i++) {
    sum += a[i];
}

cout << "sum : " << sum << endl;
```

Continue Statement

- The continue statement can only be used inside loops
- The continue statement causes the execution to skip to the end of the loop, ready to start a new iteration

```
int a[10] = {1,2,-3,-4,-5,5,4,3,-2,-1};
int sum = 0;

for (int i = 0; i < 10; i++) {
    if (a[i] < 0) continue;
    sum += a[i];
}

cout << "sum : " << sum << endl;
```

Functions

- A *function* is a chunk of code that can be called to perform some well-defined task
  - Return type
    - This specifies the type of value or object that is returned by the function
  - Function name
    - This indicates the name that is given to the function
  - Argument list
    - This serves as a list of placeholders for the values that will be passed into the function
  - Function body
    - This is a collection of C++ statements that define the actual computations to be performed by the function

```
bool evenSum(int a[], int n) {                    // function declaration
        int sum = 0;
        for (int i = 0; i < n; i++)               // sum the array elements
                sum += a[i];
        return (sum % 2) == 0;                    // returns true if sum is even
}
```

Function declaration vs definition

- Function specifications in C++ typically involve two steps
  - A function is *declared*, by specifying three things: the function's return type, its name, and its argument list. This three-part combination is called the function's *signature* or *prototype*

```
bool evenSum(int a[], int n);                    // function declaration
```

  - the function is *defined*. The definition consists both of the function's signature and the function body

```
bool evenSum(int a[], int n) {                   // function declaration
        int sum = 0;
        for (int i = 0; i < n; i++)              // sum the array elements
                sum += a[i];
        return (sum % 2) == 0;                   // returns true if sum is even
}
```

Argument Passing

- Sometimes it is useful for the function to modify one of its arguments. To do so, we can explicitly define a formal argument to be a *reference type*

```
void f (int value, int& ref ) {   // one value and one reference
        value++;                  // no effect on the actual argument
        ref ++;                   // modifies the actual argument

        cout << value << endl;    // outputs 2
        cout << ref << endl;      // outputs 6
}
int main() {
        int cat = 1;
        int dog = 5;
        f(cat, dog);              // pass cat by value, dog by ref
        cout << cat << endl;      // outputs 1
        cout << dog << endl;      // outputs 6
        return 0;
}
```

Function Overloading

- Two or more functions can be defined with the same name but with different argument lists
- Such definitions are useful in situations where we desire two functions that achieve essentially the same purpose, but do it with different types of arguments

```
void print(int x)                              // print an integer
{ cout << x; }

void print(const Passenger& pass) {      // print a Passenger
        cout << pass.name << " " << pass.mealPref;
        if (pass.isFreqFlyer)
                cout << " " << pass.freqFlyerNo;
}
```

Class

- Similar to Structure (normally used for grouping data)
- Class is often used for data abstraction and reuse
- There are more detailed differences about private, public, …

```
struct Passenger {
    string     name;
    MealType   mealPref;
    Bool       isFreqFlyer;
    string     freqFlyerNo;
};
```

```
class Passenger {
    string     name;
    MealType   mealPref;
    Bool       isFreqFlyer;
    string     freqFlyerNo;
};
```

More About C++

- Much more than what we learned ☺
  - Template
  - Switch
  - Over-riding
  - Inheritance
  - …
- Learn from examples!
- If you become an expert in C++, you will also be able to write high-quality code in other programming languages!

References

- Further reading list and references
  - https://www.w3schools.com/cpp/

- Slide credit
  - Seung-Hwan Baek
  - Jaesik Park
  - Jong-Hyeok Lee