Lecture #13 | A few more C++ syntax

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Previously in Object-Oriented Programming

- Polymorphism: virtual functions
  - Functions with the same name may behave differently, depending on which types (i.e., classes) they are associated with
- Polymorphism: templates

Today's topic

- A few more C++ syntax or library
  - Static member
  - Reference
  - Copy constructor and copy assignment
  - C++ string: `std::string`

`static` member

- *A static member* is a variable that is a part of a class, but is not a part of an object of that class
- *A static member function* is a function that needs access to members of class, yet doesn't need to be invoked for a particular object

Example: `static` member

```
class Static {
public:
    static int s_val; // declaration
    int val;
    Static(int val) { this->val = val; s_val++; }
    static void set_sval(int s_val_) { s_val = s_val_; }
};
int Static::s_val = 0; // definition
int main() {
    Static s1(42); Static s2(23);
    cout << s1.val << " " << s1.s_val << endl;
    cout << s2.val << " " << s2.s_val << endl;
    s1.set_sval(7);
    cout << Static::s_val << endl;
}
```

Revisit: call by value v.s. call by reference

```
void call_by_value(int x) {
    x++;
    cout << "call_by_value: x=" << x << endl;
}
void call_by_reference(int* ptr) {
    (*ptr)++;
    cout << "call_by_reference: *ptr=" << *ptr << endl;
}
int main(void) {
    int x = 23;
    call_by_value(x);       cout << "x=" << x << endl;
    call_by_reference(&x); cout << "x=" << x << endl;
}
```

Revisit: call by reference (C v.s. C++)

```
void call_by_reference(int* ptr) {
    (*ptr)++;
    cout << "call_by_reference: *ptr=" << *ptr << endl;
}
void call_by_reference_cpp(int& x) { // C++ only, will be discussed later
    x++;
    cout << "call_by_reference_cpp: x=" << x << endl;
}
int main(void) {
    int x = 23;
    call_by_reference(&x);    cout << "x=" << x << endl;
    call_by_reference_cpp(x); cout << "x=" << x << endl;
}
```

Reference

- Reference
  – an automatically dereferenced* immutable pointer, or
  – an alternative name (alias) for an object
- Pointer v.s. reference

```
int x = 10;
int* p = &x;
*p = 7;
int x2 = *p;
int* p2 = p;
p = &x2;
```

```
int y = 10;
int&r = y;
r = 7;
int y2 = r;
int& r2 = y2;
r2 = r;
r = &y2; // error
```

Source: Programming, principle and practice using C++

Example: call by reference

```
void call_by_reference_cpp(int& x) {
    x++;
    cout << "call_by_reference_cpp: x=" << x << endl;
}
int main(void) {
    int x = 23;
    call_by_reference_cpp(x); cout << "x=" << x << endl;
}
```

Reference and inheritance

- A pointer of a derived class can be implicitly converted to that of a base class; so can a reference
- Example

```
int main()
{
    Rectangle rectangle {1, 2};

    Shape& r = rectangle;
    cout << r.getArea() << endl;
}
```

Using a reference as a return value

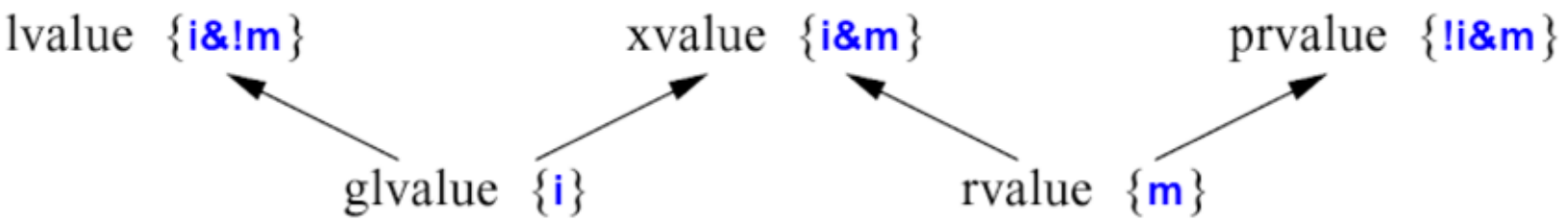
- When a return value is used
  - in an expression: The behavior is similar to the case we return a value itself
  - left side of an assignment: We can assign a value to a variable that is pointed by *a returned reference*
- The same principle holds, i.e., a reference is an alias in either case
- Typically used to return an object as a reference, e.g.,

```
return *this;
```

```
class Simple {
private:
    int val;
public:
    Simple(int val_) : val{val_} {}
    int& getRef() { return val; }
};
int main()
{
    Simple s {42};
    cout << s.getRef() << endl;
    int d = s.getRef();
    cout << d << endl;
    s.getRef() = 63;
    cout << s.getRef() << endl;
}
```

(Advanced) lvalues and rvalues

- lvalue: an expression that refers to an object
- rvalue: a value that is not an lvalue
- (Optionally) More technically*,
  - Had identity: The program has the name of, pointer to, or reference to the object so that it is possible to determine if two objects are the same, whether the value of the object has changed, etc.
  - Is movable: The object may be moved from (i.e., we can move its value to another location and leave the object in a valid but unspecified state, rather than copy)



* Chapter 6.4 “Object and values” from “The C++ programming language”

A few notes on reference

- Use `const` if function parameters are NOT supposed to be changed; In C++, it can be vague whether a specific parameter takes a value or a reference
- There shall be no references to references, no arrays of references, and no pointers to references

Copy of built-in data types

- For built-in data types, the following operations are supported
  - Copy the value of one variable to another variable (assignment)
  - Initialize a new variable with the value of another variable

```
int d1 = 42;
int d2;
d2 = d1;
int d3 {d1};
```

Revisit: `class IntList` (partial)

```
class IntList {
private:
    int n;
    int* elem;
public:
    IntList(int n_ = 0);
    ~IntList();
    void set(int index, int value) {
        elem[index] = value;
    }
    int get(int index) { return elem[index]; }
    void display() {
        for (int i = 0; i < n_; ++i)
            cout << elem[i];
        cout << endl;
    }
};
```

```
IntList::IntList(int n_)
{
    n = n_;
    elem = new int[n];

    for (int i = 0; i < n; i++)
        elem[i] = 0;
}

IntList::IntList(int n_, int* a)
{
    elem = new int[n];

    for (n = 0; n < n_; n++)
        elem[n] = a[n];
}
IntList::~IntList()
{
    delete[] elem;
}
```

Copy constructor and assignment with classes

- Copy of a `class X` is defined by two operations
  - Copy constructor: `X(const X&)`
  - Copy assignment: `X& operator=(const X&)`
- Default copy constructor or assignment copies only the values of member variables
- Example

```
int main() {
    IntList v1 {2};
    IntList v2 = v1;
    IntList v3(v1);  // or IntList v3 {v1};
}
```

Problems of default copy constructor or assignment

```
int main()
{
    IntList list1 {2};
    list1.set(0, 42);
    list1.set(1, 23);
    IntList list2 = list1;
    IntList list3 {list1};

    list1.set(0, 63);
    list1.display();
    list2.display();
    list3.display();
}
```

```
$ ./intlist
63 23
63 23
63 23
IntList with 2 elements is
destroyed.
intlist (84698,0x7fff982253c0)
malloc: *** error for object
0x7f9d8cd00000: pointer being freed
was not allocated
*** set a breakpoint in
malloc_error_break to debug
Abort trap: 6
```

Implementation of copy constructor and assignment

```
class IntList {
    ...
    IntList(IntList& list);
    IntList& operator=(const IntList& list);
};
IntList::IntList(IntList& list) {
    n = list.n;
    elem = new int[n];
    for (int i = 0; i < n; i++)
        elem[i] = list.elem[i];
}
IntList& IntList::operator=(const IntList& list) {
    n = list.n;
    delete[] elem;
    elem = new int[n];
    for (int i = 0; i < n; i++)
        elem[i] = list.elem[i];
    return *this;
}
```

After implementing copy constructor and assignment

```
int main()
{
    IntList list1 {2};
    list1.set(0, 42);
    list1.set(1, 23);
    IntList list2 = list1;
    IntList list3 {list1};

    list1.set(0, 63);
    list1.display();
    list2.display();
    list3.display();
}
```

```
$ ./intlist
63 23
42 23
42 23
```

`std::string`

- C string is very... primitive, and there should be a better way to handle string
- `std::string` class provides convenient features, so that you don't have worry about detailed implementation (an example of abstraction)
- Templates are used to provide additional string types, e.g.,
  - `std::wstring`
  - `std::u16string`
  - `std::u32string`
- But using Korean characters (a.k.a., 한글) may be tricky
  - You many need to know character encodings (or locale) of your development and user environment
  - Does C++ support unicode? Yes and No

Example: std::string

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1; // variable declaration
    // variable declaration with initialization
    // string support many initialization methods
    string s2 = "Hello, world!";
    string s3("Hello, world!");
    string s4 {"Hello, world!"};

    // string operations
    s1 = s2 + s3;
    cout << s1 + " " + s1[0] << s1[1] << s1.at(2) << endl;
}
```

Example: `std::string` (cont.)

```
 // string with cin
string name;
string age;
// what would happen if you type "John Doe"
cout << "Enter your full name: ";
cin >> name;
cout << "Enter your age: ";
cin >> age;
cout << name << " is " << age << " years old.\n";

// right way to handle multiple words into one string
// but be careful when you use this right after using cin
cout << "Enter your full name: ";
getline(cin, name); // read a full line
cout << name << endl;
```

Example: `std::string` (cont.)

```
// which one do you prefer, C-string or C++-string?
string arr[] =
    {"I ", "am ", "familliar ", "with ", "C/C++ ", "pointers!"};
int n_words = sizeof(arr) / sizeof(string);
string buffer {};

for (int i = 0; i < n_words; i++)
{
    cout << "length of \"" + arr[i] << "\": " << arr[i].length() << endl;
    buffer += arr[i];
}
cout << buffer << "\n";
```

Example: `std::string` (cont.)

```
// may work, or may not work, depending on your system
string hangul = "한글은 과학적 창제원리에 따라 만들어졌습니다.";
cout << hangul << endl;
```

Reading list

- Learn C++
  - `this`, static member variables/functions: Ch. 8.8-12
  - Reference: Ch. 6.10-12, Ch. 7.3-4
  - string: Ch. 4.4b
- Reference on character encodings
  - Unicode in C++: https://youtu.be/MW884pluTw8
    - History and details of character encodings and C/C++ string
    - A little bit long (1:29:40, yes 1.5 hours, not 1.5 minutes)
  - 한글 인코딩의 이해
    - http://d2.naver.com/helloworld/19187
    - http://d2.naver.com/helloworld/76650

ANY QUESTIONS?