Lecture #8 | Class

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

What we have covered so far…

- C/C++ common syntax
  - Data types
  - Variables/operators
  - Control flows (e.g., `if`, `while`, `for`, `switch`)
  - Functions
  - Standard inputs/outputs

What will be covered in the rest of this course?

- Introduction to object-oriented programming
- C++ syntax to make object, a.k.a., `class`
- Standard libraries based on classes, incl. standard template libraries (STLs)
- More on C++ syntax/idiom such as memory management, etc.
- Main features of OOP paradigm
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
- Design patterns: from small, but poorly designed code to well-designed code

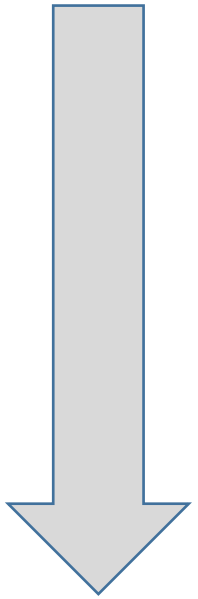
What are classes or objects?

- User-defined data type with data and functions
  - Data*: member variable, field, attributes, properties, characteristics
  - Functions*: member functions, methods, responsibilities

* Names for data and functions may differ by programming languages, books, etc.

But why…?

- Programming languages have evolved to support a higher level of abstraction
  - Binary code
  - Assembly Language
  - Procedural languages (e.g., C, fotran)
  - Object-oriented languages (e.g., C++, Java, python)
  - Declarative (or functional) languages (e.g., Haskell, ML, prolog, Scala)
  - ?



Level of abstraction

Why do we need higher level of abstraction?

- Ease of maintenance
  - Maintaining large applications is very difficult due to the inter-dependency of codes; if you change one part, other parts may be affected
  - OOP aids development of modular applications, which ease code maintenance
- Reusability
  - A piece of code is easier to be reused by the same author or other people
  - Not just code, but designs can be reused; these solutions are called design patterns
- Extendibility
  - To support application-specific requirements, existing codes can be extended, meaning adding/modifying the behavior of the original codes

class v.s. object

- A class is a user defined type with associated methods
- An object (or an instance) is a specific realization of class
  - Memory allocation for all the data (member variables) of the given class
- These terms are often used interchangeably
- Example (with basic type):

```
int p;
int q;
```

  - `int`: a type or a class
  - `p, q`: instances or objects
  - `p` and `q` are instances (or objects) of `int` type

Class declaration v.s. definition

- Class declaration
  - Tells C++ compilers the member variables & functions of a specific class
  - Usually resides in .h files
- Class definition
  - Tells C++ compilers what statements are executed when a member function is called
  - Usually resides in .cpp (or .cxx, …) files
- Note: it is common to have separate .h/.cpp files per class

We need to model what we want to implement

- Let's implement a class which mimics python list
- Member variables
  - `n`: number of stored elements
  - `elem`: an array that stores elements
- Member functions
  - `int len()`: return `n`, i.e., number of stored elements
  - `void set(int index, int value)`: set the `index`-th element as value
  - `int get(int index)`: return `index`-th element
  - `void append(int value)`: add `value` at the end of store elements
  - Constructors: will discuss later

* Letter and numeric grade should be consistent

Class declaration*

```
constexpr int max_list = 1024;
class IntList
{
private:
    int n;
    // need to replace with dynamic mem mgmt
    int elem[max_list] = {0, };
public:
    IntList(int n_ = 0);
    int len();
    void set(int index, int value);
    int get(int index);
    void append(int value);
};
```

- class name
- member variables
- member functions
- constructor: invoked when a class is instantiated
- access control
  - public
  - private
  - protected

* All the codes in this slide does NOT contain any error checking as in python

Class definition: constructor

- Constructors are member functions which have the same name with the class
- A constructor is called when an instance is created
  - If you don't write any constructor, a default constructor would be called
  - A class may have multiple constructors, but only one of them is called
- Constructors cannot have return value → But we don't need to add `void`
- Example

```
IntList::IntList(int n_)
{
    n = n_; // assign to a member variable
    //this->n = n_; // this is similar to self in python
    for (int i = 0; i < n_; i++)
        elem[i] = 0;
}
```

Class definition: member functions

```
int IntList::len()
{
    return n;
}

void IntList::set(int index,
                  int value)
{
    elem[index] = value;
}
```

```
int IntList::get(int index)
{
    return elem[index];
}

void IntList::append(int value)
{
    elem[n++] = value;
}
```

Examples of using `IntList` class

```
int main()
{
    IntList list(3);
    for (int i = 0; i < list.len(); i++)
        cout << list.get(i) << ' ';
    cout << endl;
    list.append(42);
    for (int i = 0; i < list.len(); i++)
        cout << list.get(i) << ' ';
    cout << endl;
}
```

Member function definition within class body

```
class IntList
{
private:
    int n;
    // need to replace with dynamic mem mgmt
    int elem[max_list] = {0, };
public:
    IntList(int n_ = 0);
    IntList(int n_, int* a);
    int len() { return n; }
    void set(int index, int value) {
        elem[index] = value;
    }
    int get(int index) { return elem[index]; }
    void append(int value) { elem[n++] = value; }
};
```

- Simple member functions are typically implemented in the declaration
- When declared in the class definition, member functions are defined as inline functions

Reference: inline functions

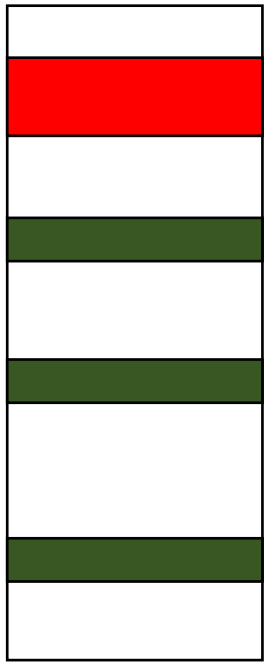
- inline function: When the compiler inline-expands a function call, the function's code gets inserted into the caller's code stream (similar to what happens with a #define macro).
- Benefits and pitfalls (see the reading list)
  - Program may run faster or slower
  - Compile code may be larger or smaller

```
int max(int x, int y);
inline int max(int x, int y)
{
    return x > y ? x : y;
}
```

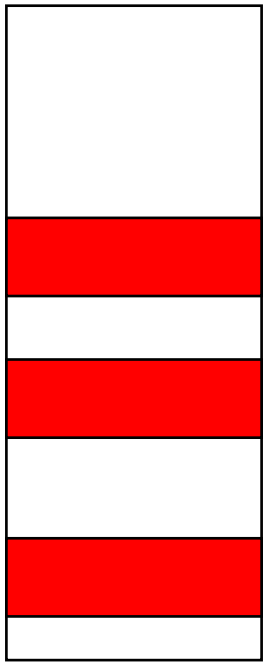
```
class SimpleClass {
public:
    int func(void);
};
inline SimpleClass::func(void) {
}
```

Regular functions v.s. inline functions

- If you define a function `func()`, and call it 3 times in your code
  - Red: function body, Green: code necessary for function call
- Regular function



- Inline function



You may define more than one constructors

```
// intlist.h
class IntList
{
   ...
public:
    IntList(int n_ = 0);
    IntList(int n_, int* a);
    ...
};
```

```
// intlist.cpp
IntList::IntList(int n_)
{
    n = n_;
    for (int i = 0; i < n_; i++)
        elem[i] = 0;
}
IntList::IntList(int n_, int* a)
{
    for (n = 0; n < n_; n++)
        elem[n] = a[n];
}
```

Constructors with default values*

```
// intlist.h
class IntList
{
public:
    IntList(int n_ = 0);
    IntList(int n_, int* a);
    ...
};
/*
Parameters with a default
value should appear after ones
without a default value
*/
```

```
// intlist.cpp
IntList::IntList(int n_)
{
    n = n_;
    for (int i = 0; i < n_; i++)
        elem[i] = 0;
}
IntList::IntList(int n_, int* a)
{
    for (int n = 0; n < n_; n++)
        elem[n] = a[n];
}
```

* Any function can take default values

Constructors with member initializer list*

- The following codes assign n_ to the member variable n

```
// intlist.h
// Old C++ style
class IntList
{
public:
    IntList(int n_): n(n_)
    {
        ...
        for (int i = 0; i < n_; i++)
            elem[i] = 0;
    };
    ...
};
```

```
// intlist.cpp
// Modern C++ style
class IntList
{
public:
    IntList(int n_): n {n_}
    {
        ...
        for (int i = 0; i < n_; i++)
            elem[i] = 0;
    };
    ...
};
```

* You can specify more than more initializer separated by comma

Default constructor

- Default constructor: a constructor that can be called without an argument
- A default constructor is used if no arguments are specified or if an empty initializer list is provided
- The built-in types are considered to have default and copy constructors*

```
class Vector {
public:
    Vector(); // default constructor; no elements
    // ...
};

Vector v1;    // OK
Vector v2 {}; // OK
```

* Will be discussed later

Destructor

- A destructor is invoked when an object is destroyed, e.g.,
  - When a local variables goes out of scope
  - When an object on the free store is deleted*
- The name of a destructor is ~ followed by the class name (‘~’ means complement’)
- Typically used to release a resource such as allocated memory, close file handle*

```
class IntList
{
    ...
public:
    ~IntList() {
        cout << "IntList with" << n
             << "elements is destroyed.”
             << endl;
    }
    ...
};
int main()
{
    IntList int_list(3);
}
```

* Will be covered with dynamic memory allocation

Access control with `private` and `public`

- Public member variables/functions can be accessed outside of the class
- Private member variables/functions can be accessed only in the class member functions

```
// intlist.h
class IntList
{
private:
    int n;
    int elem[max_list] = {0, };
public:
    int len() { return n; }
    ...
};
```

```
#include <intlist.h>

int main()
{
    IntList list(3);
    cout << list.len();
    cout << list.n; // error
}
```

Reference: code documentation

- Many programming languages provide (and strongly recommends to use) language-specific documentation, e.g.,
  - javadoc
  - pydoc
- C++ does not provide any language-specific documentation
- Doxygen (http://www.stack.nl/~dimitri/doxygen/index.html)
  - De facto standard for C++ documentation
  - Provides javadoc-like features for C++
- Code annotation (or documentation) in assignment1.cpp follows doxygen-style documentation

Reading list

- Learn C++
  - class: Ch. 8.1-5
- Inline functions: https://isocpp.org/wiki/faq/inline-functions
  - Some parts require understanding of compiler or call stack
  - You may skip the parts you don’t understand

ANY QUESTIONS?