COMP2012

COMP2012 Object-Oriented Programming and Data Structures
2018 Spring, HKUST

wliuax@connect.ust.hk

Basic Knowledge

- const to the left of the $*$ $\Longrightarrow$ pointer to const const to the right of the $*$ $\Longrightarrow$ const pointer

  (read from right to left)

Constructor and Destructor

- Brace initializer {} : if data members are **all public**

```
class Word { public: int frequency; const char* str; };
Word movie = {1, "Titanic"};
```

- A default constructor is automatically supplied only when **no user-defined constructors** are found!

```
//Word;         // Compilation Error: declaration does not declare anything
Word();         // initialization with default constructor, POOR GUY
Word w;         // initialization with default constructor
Word w0();      // function declaration with name "w0" and return type Word
Word w1{};      // initialization with default constructor
//w0.print();   // Compilation Error: request for member 'print' in 'w0', which is of non-
class type 'Word ()'
```

- A conversion constructor is a constructor accepting a **single argument**,

  or all but one argument have default values.

```
explicit Word(const char* s) { ... } // to disallow implicit conversion
Word movie0 {'A'}; // Explicit conversion
Word movie1 = 'B'; // Implicit conversion: Error!
```

- A copy constructor X::X(const X&) is called when:
  - parameter passed by value

- object returned by value
- initialization using the assignment syntax (e.g. Word y = x;)

default: memberwise copy by calling the copy constructor of each data member

C.f. memberwise assignment: calling the assignment operator= of each data member

```
void Bug(Word& x) { Word bug("bug", 4); x = bug; } // memory leak of x (movie)
int main() {
    Word movie {"Titanic"};
    Bug(movie); // memory leak of movie; and the Word{"bug", 4} is deleted right afterwards
    return 0;   // ~Word() cannot delete dangling pointer
}
```

- A parameter can have its default argument specified only once (usually in header, not in definition)!
- **const or reference members** must be initialized using member initialization list!

  It cannot be done using default arguments!
- The default destructor simply releases the memory in stack!
- To explicitly generate or not generate a constructor/destructor: = default; = delete;
- Order of construction/destruction:
  1. base class
  2. data members (in the order of that they're declared)

     ```
     class Word_Pair {
     private: Word w1; Word w2;
     public: Word_Pair(const char* s1, const char* s2) : w2(s2), w1(s1,5) {} };
     // w1 is created first, and then w2
     ```

  3. itself

  Destruction: in the reverse order (use delete properly in the destructors!)

Inheritance

- Constructors and destructors are never inherited
- Access:
  - public: everything
  - protected: member functions and friends of **the class and derived classes**
  - private: member functions and friends of **the class**
- Slicing: only for **public inheritance**

```
Base base;
base = derived;     // Slicing
cout << typeid(base).name() << endl; // the derived becomes a base

Base b = derived;   // Initializing with copy constructor
Base* b = &derived; // Can't use derived-class specific members
Base& b = derived;  // Can't use derived-class specific members

// The following assignments give compilation errors:
//derived = base;       // Unless it's user-defined
//Derived* d = &base;   // No such conversion!
//Derived& d = base;    // No such conversion!
```

- virtual
  - member functions: virtual in the base class, virtual in all derived classes
  - destructor: virtual only in the base class
  - pure virtual functions: only in an abstract base class
  - dynamic binding: only works on a **pointer or reference**

    ```
    class A { public: virtual void f() { cout << "func in A" << endl; } };
    class B : public A { private: virtual void f() { cout << "func in B" << endl; } };

    int main() {
        B bObj;
        //bObj.f();     // Error. Since func() in B is private
        A& aRef = bObj; // Liscov substitution principle

        aRef.f();   // We can call func() in B via dynamic binding, even it is private
        // Type of aRef is A, so we should first reach func() in A
        // f() in A is virtual, so we check and find the object aRef refers to is B type
        // f() in B is called, despite that it's private
    }
    ```

  - dynamic_cast<Class_Name*>(ptr_or_ref) only works on pointers and references of polymorphic class (with virtual functions) types
- Abstract Base Class (ABC)
  - No objects of ABC can be created
  - Its derived classes must implement the pure virtual functions, otherwise they will also be ABCs
  - cannot be used as:
    - argument type that is passed by value
    - return type that is returned by value
    - type of an explicit conversion
- final

```
class Student final : public UPerson { // No sub-classes can be derived from a final class
    public: /* Other data and functions */
    // a final virtual function can't be override any more
    virtual void print() const override final {}
};
```