Topic 3 (Part III: Overloading and Polymorphism)
C++ advanced features review: *when can/should I use them?*

資料結構與程式設計

Data Structure and Programming

Sep, 2012

Sharing in the code…

◆Remember:

Many constructs (in C++) are to promote **sharing** in the code.

1. Pointer: share the same data location (as different variables)
2. Reference: an alias to an existing variable
3. Function: share the common codes
4. Class: data with the same attributes and definition (as data type)

Sharing in the code…

- And we will learn…
  5. Inherited class: different but similar classes sharing the common data members or member functions
  6. Function overloading: same function name, diff arguments
  7. Operator overloading: redefine the C++ operators for user-defined data type (class)
  8. Template class: same storage method, diff data types
  9. Template function: same algorithm flow, diff data types
  10. Functional object: same algorithm flow, diff argument types

Key Concept #1: "Has a" vs. "Is a"

- class Car {

```
    Engine   _eng;
};
```

➔ Class Car "has a" data member of type "Engine"

- class Dog : public Animal {

```
    …
};
```

➔ Class Dog "is a" inherited type of "Animal"

Key Concept #2: Inheritance to share common data and methods

◆
```
class Base {
 public:
   <public data or methods>
 protected:  // public to Derived classes
             // private to others
   <shared data or methods>
 private:    // Base's private only
   <private data or methods>
};
class Derived : public Base {
 public:
   <specific data or methods>
 private:
   <specific data or methods>
};
```


"protected" vs. "private" access specifiers

- protected:
  - To allow member functions of the derived classes to directly access the base class' data members and member functions
  - To shield other classes from directly accessing
- private:
  - Member functions of the derived classes cannot directly access the base class' private components
  - However, derived classes still inherit the private data members (Remember: "is a")
    - To access them, create protected or public functions in base class
- Note: "friend" specification is NOT inherited

Key Concept #3: Inheritance to specialize distinct methods with the same function

```
class Shape {
 public:       virtual void draw() = 0;
 protected:    double  _centerCoord;
};
class Square : public Shape {
 public:       void draw();
 private:      double  _edgeLength;
};
class Circle: public Shape {
 public:       void draw();
 private:      double  _radiusLength;
};
```

➔ **In C style, people use "switch" → NOT GOOD**

Is this virtual function useful?

```
class Base {
 public:
   virtual void f();
   void g();
};
class Derived: public Base
{
 public:
   void f();
   void g();
};
int main()
{
   Base b; b.f(); b.g();
   Derived d; d.f(); d.g();
}
```

➔ Which f() and g() are called?

Base::f()

Base::g()

Derived::f()

Derived::g()

➔ What does "virtual" keyword do in this case? What if we DO NOT declare "virtual" for f()?

➔ What's the difference if we DO NOT declare Derived as a derived class of Base?

Key Concept #4: Virtual function is useful ONLY with polymorphism

- Polymorphism occurs when a derived object invokes a virtual function through a base-class pointer or reference
  - C++ dynamically chooses the correct function for the class from which the object was instantiated
- Common usage:
  - Base *p = new Derived;
    p->virtualFunction();
  - Derived d;
    Base &r = d;
    r.virtualFunction();

Key Concept #5: Polymorphism for dynamic type specification

- Analogy:
  - The size of a dynamic array is undefined.
    It is determined during execution.

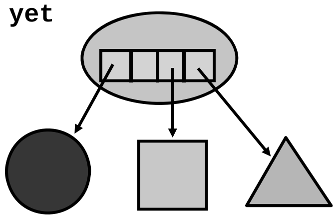
  ➔
```
int  *arr = 0;
…  // size is determined
arr = new int[size];
```
- When the type of a variable is not determined before execution, but its category is clearly defined…

  ➔ Category: base class;   type: inherited class

  ➔
```
Category *p;
…
p = new MyType;
```

[NOTE] We can use "pointer" when the type of the derived class is not determined in the beginning

```
class Node {
   Array<Node*>  _children;
public:
  void addChild(Node* c){_children.push_back(c); }
};
class BinaryNode : public Node {...};
class NNaryNode : public Node {...};
-------------------------------------------
Node*   o1, o2; // type is not yet
...             // determined
o1 = new Circle;
o2 = new Square;
n->addChild(o1);
n->addChild(o2);
```



Key Concept #6: Virtual function makes polymorphism meaningful

- Use base class pointer or reference as the interface.
  Pass inherited class pointer or object for different application scenarios.
- [Example] HW #3's command registration

```
class CmdExec {
public:
   virtual CmdExecStatus exec(const string&) = 0;
   virtual void usage(ostream&) const = 0;
   virtual void help() const = 0;
};
class HelpCmd : public CmdExec {
public:
   CmdExecStatus exec(const string& option);
   void usage(ostream& os) const;
   void help() const;
};
class QuitCmd : public CmdExec { ... };
```

More on HW#3: CmdExec as common interfaces for command-related operations

◆ Command registration

```
class CmdParser {
   map<const string, CmdExec*>  _cmdMap;
};
int main() {
   if (!initCommonCmd() || !initCalcCmd())...
}
bool initCommonCmd() {
   if (!(cmdMgr->regCmd("Quit", 1, new QuitCmd) &&
         cmdMgr->regCmd("HELp", 3, new HelpCmd)...
}
bool CmdParser::regCmd(..., CmdExec* e) {
   return (_cmdMap.insert
          (CmdRegPair(mandCmd, e))).second;
}
```


More on HW#3: Command Execution

```
int main() {
   while (status != CMD_EXEC_QUIT) {
      status = cmdMgr->execOneCmd();
   }
}
CmdExecStatus
CmdParser::execOneCmd()
{
   readCmd(*dofile);
   // read cmd string from _history.back()
   // retrieve cmd from map<string, CmdExec*>
   CmdExec* e = parseCmd(option);
   return e->exec();
}
```

More on HW #3: CmdClass MACRO

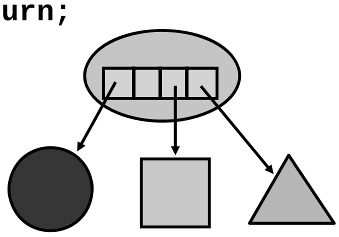
- For each inherited class:

```
#define CmdClass(T)                             \
class T: public CmdExec {                       \
public:                                         \
   T() {}                                       \
   ~T() {}                                      \
   CmdExecStatus exec(const string& option);    \
   void usage(ostream& os) const;               \
   void help() const;                           \
}
```

- Implement “exec()”, “usage()” and “help()” functions independently in each package/directory
  - ➔ Easy to extend the set of commands

In the previous “Node” example...

- ```
  class Node { virtual void draw() const=0; }
  class Circle: public Node { void draw() const; }
  class Square: public Node { void draw() const; }
  ```
- ```
  void Graph::dfsTraverse() {
     Graph::setGlobalRef();
     dfsTraverse(_root);
  }
  void Graph::dfsTraverse(Node *n) {
     if (n->isGlobalRef()) return;
     n->setGlobalRef();
     for_each_child(c, n)
        dfsTraverse(c);
     n->draw();
  }
  ```



Key Concept #7: Function prototype of virtual function

- Be sure to make the function prototype of the inherited class exactly the same as that of the base class, including “const”, etc.
- Once a function is declared `virtual`, it remains `virtual` all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared `virtual` when a class overrides it.
  - But explicitly declare virtual will make the program more readable

Virtual Functions

```
class Animal {
   // no “bark” is defined
};
class Dog: public Animal {
 public:
   virtual void bark();
};
class KDog: public Dog {
 public:
   void bark();
};
class GDog: public KDog {
 public:
   void bark();
};
```

```
int main() {
   Animal *a = new KDog;
   a->bark();

   Dog *b = new KDog;
   b->bark();

   Dog *c = new GDog;
   c->bark();

   Kdog *d = new Gdog;
   d->bark();
}
```

➔ **Any compilation error?**
➔ **Which bark() is called?**

Virtual Functions

```
class Base {
 public:
   virtual void f();
   void g();
   virtual void h();
};
class Derived: public Base
{
 public:
   void f();
   void g();
};
int main()
{
   Base* p = new Derived;
   p->f(); p->g(); p->h();
```

```
   Base* q = new Base;
   q->f(); q->g(); q->h();

   Derived* r = new Derived;
   r->f(); r->g(); r->h();
}
```

➔ Any compilation error?
➔ Which f(), g(), h() are called?

== p ==
Derived::f()
Base::g()
Base::h()
== q ==
Base::f()
Base::g()
Base::h()
== r ==
Derived::f()
Derived::g()
Base::h()

Key Concept #8: Abstract class and pure virtual function

- A class is said “abstract” if we have no intention to create any object out of it.
  - e.g. “Node”, “CmdExec” in the previous examples
- A “pure virtual function” is a function defined as “= 0”.
  - We cannot omit the function definition of any pure virtual function in the derived class.
- If a class has a pure virtual function, this class becomes “abstract”.
  - We cannot create any object for an abstract class (e.g. Node n; Node *p = new Node;)
  - But polymorphism is OK (e.g. Node *n = new Circle)

Summary #1: Keyword "virtual"

- Explicitly add the keyword "virtual" whenever applicable
  - Only if this function will NOT be made virtual in the future
- The function definition in the inherited class can be omitted if the intention is to call the base-class function
  - But NOT applicable if the function in the base class is pure virtual.

Key Concept #9: Constructors

- As its name suggests, the constructor of the "base" class will be called before that of the inherited class.
  - Both will/must be called.
- Constructor cannot be virtual
  - Doesn't make sense to be virtual.
- What about destructor? Which one will be called first?

Key Concept #10: Virtual Destructor

```
class Base
{
   A  _a;
 public:
   Base(){}
   ~Base(){}
};

class Derived:public Base
{
   B  _b;
 public:
   Derived(){}
   ~Derived(){}
};
```

```
int
main()
{
   Base* p = new Derived;
   Base* q = new Base;
   Derived* r = new Derived;
   ...
   delete p; delete q;
   delete r
}
```

➔ Which constrcutors / destructors are called?

(~Base(); ~Base(); ~Derived())

➔ What happens if the derived class' destructor is not called?

Declaring Virtual Destructor

```
class Base
{
   A  _a;
 public:
   Base(){}
   virtual ~Base(){}
};

class Derived:public Base
{
   B  _b;
 public:
   Derived(){}
   ~Derived(){}
};
```

```
int
main()
{
   Base* p = new Derived;
   Base* q = new Base;
   Derived* r = new Derived;
   ...
   delete p; delete q;
   delete r
}
```

➔ Which constrcutors / destructors are called?

What's the difference?

```
class Base
{
 public:
   Base(int){}
   virtual ~Base(){}
};

class Derived:public Base
{
 public:
   Derived(int){}
   ~Derived(){}
};
```

```
int
main()
{
   Base *p
     = new Derived(10);
   Base *q = new Base(20);
   ...
   delete p;
   delete q;
}
```

➔Compilation error. Why?

Why compilation error?

◆By default, "Base()" will be called by any "Derived(...)"

```
[Sol #1]
class Base {
 public:
   Base() {}
   Base(int){}
   virtual ~Base(){}
};

class Derived: public
   Base {
 public:
   Derived(int){}
   ~Derived(){}
};
```

➔ But "Base(int)" won't be called

```
[Sol #2]
class Base {
 public:
   Base(int){}
   virtual ~Base(){}
};

class Derived: public Base {
 public:
   // Explicitly call Base(i)
   Derived(int i):Base(i){}
   ~Derived(){}
};
```

➔ Recommended

Summary #2: Constructor & Destructor

In short, when calling constructor / destructor of the derived class,

make sure the data members in the base class are well taken care of

➔

1. Explicitly calling Base constructor
2. Define “virtual” Base destructor

Key Concept #11: Casting a base class pointer to the derived class

- class Base { };
  class Derived: public Base {
  public: void f() {}
  };
  ======
  Base *p = new Derived();
  p->f();
  ➔ Any problem?
  ➔ Compile error if “f()” is not defined in Base
- When we declare a member function in a derived class, and we use polymorphism to define the variable as a base class pointer
  - How can we call the derived class’ member function?
  - Create a (pure) virtual function that does nothing?
    - If so, what about the other derived classes?

  ➔ Leave the member function in derived class only; use “type casting” to cast the pointer from base class to derived class

dynamic_cast<Type>(variable)

- [Note] Use "dynamic_cast" to cast between "base" and "derived" classes

```
class Child : public Parent {
  void childOnlyMethod();
  // no virtual childOnlyMethod() in Parent
  ...
};
// Better make sure, "Parent* p = new Child;"
void f(Parent *p) {
   dynamic_cast<Child *>(p)
          ->childOnlyMethod();
};
```

- [Note] If the underlying object is NOT of the derived type, 0 is assigned; ➔ Used with caution!!

static_cast<Type>(variable)

- [Note] Use "static_cast" to cast between "base" and "derived" classes

```
class Child : public Parent {
  void childOnlyMethod();
  // no virtual childOnlyMethod() in Parent
  ...
};
// Better make sure, "Parent* p = new Child;"
void f(Parent *p) {
   static_cast<Child *>(p)
         ->childOnlyMethod();
};
```

- [Note] No checking between sizes of objects; also use with caution

Key Concept #12: Access specifier in derived classes

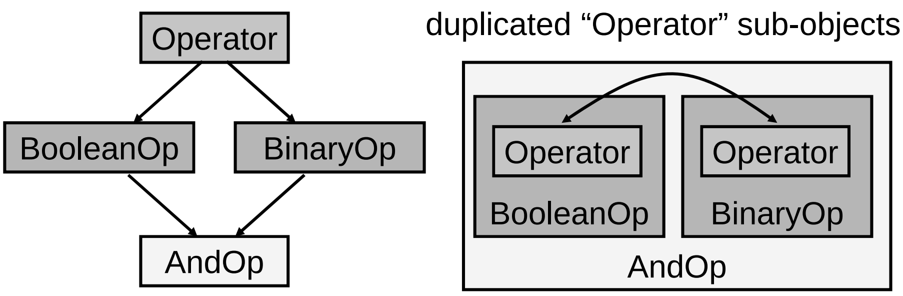
- class Derived : [accessSpecifier] Base { ... };
  - private/protected/public
- Data accessibility in derived classes

access specifier \ data in Base	private	protected	public
private	N/A	private	private
protected	N/A	protected	protected
public	N/A	protected	public

- Note: “accessSpecifier” is optional
  - class Derived: Base; ➔ class Derived: private Base;
  - struct Derived: Base; ➔ struct Derived: public Base;

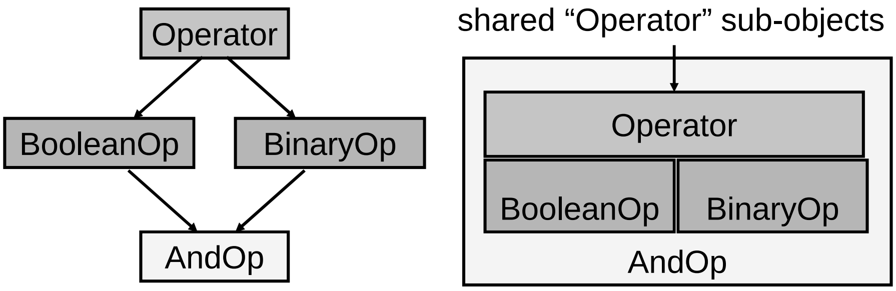
Key Concept #13: Multiple Inheritance

- class Operator {};
  class BooleanOp : public Operator {};
  class BinaryOp : public Operator {};
  class AndOp : public BooleanOp, public BinaryOp {};



Multiple Inheritance

◆ class Operator {};
  class BooleanOp : virtual public Operator {};
  class BinaryOp : virtual public Operator {};
  class AndOp : public BooleanOp, public BinaryOp {};



Sharing in the code…

◆ And we will learn…

5. Inherited class: different but similar classes sharing the common data members or member functions
6. Function overloading: same function name, diff arguments
7. Operator overloading: redefine the C++ operators for user-defined data type (class)
8. Template class: same storage method, diff data types
9. Template function: same algorithm flow, diff data types
10. Functional object: same algorithm flow, diff argument types

Key Concept #14: Function Overloading

- Sometimes we want to call the same function with different types/number of parameters, and we don't want to create different function names for them...
  - e.g. // kind of awkward...
    void computeScoreByInt(int);
    void computeScoreByStudent(const Student&);
- Function overloading
  - Same function name, different function arguments

Key Concept #15: Can't overload a function with different return types

- "Return type" is NOT part of the function signature.
  - e.g.
    bool f() { ... }
    int f() { ... }
    int main() { int i = f(); }
    ➔ Which one is called?

Key Concept #16: Default argument

- You cannot overload a function with and without default argument
  - e.g.
    void f(int i = 0);
    void f(int i);
    ➔ Compile error!! "f(int)" is redefined...
    But this is OK:
    void f(); // co-exist with "void f(int i = 0)"
- Default argument can ONLY appear once in the entire program. And it should be declared in the first encounter.
  - Usually the function prototype or inside the class definition
  - Compile error if multiply declared, even with the same value!!

Key Concept #17: Why operator overloading?

- Operator overloads are very useful in making the code more concise (c.f. Function overload)
- Basic concept:

```
MyNumber n1, n2;
n1 = "32hf908abc0";
n2 = f(...);
...
MyNumber n3 = n1 + n2;
```

1. n1 calls "MyNumber::operator +" with parameter n2
   ➔ return a temporary object, say n4
2. n3 calls "MyNumber::operator =" with parameter n4
   ➔ returned result is stored in n3 itself

Key Concept #18: Pay attention to the function prototypes for operator overloading

```
1.  T& operator = (const T& v);
2.  T& operator [] (size_type i);
3.  const T& operator [] (size_type i) const;
4.  T operator ~ () const;   // also for -, &, |, etc
5.  T& operator ++();        // ++v
6.  T operator ++(int);      // v++
7.  T operator + (const T& v) const;
    // also for -,*,&,etc
8.  T& operator += (const T& v);
    // also for -=,*=,&=,etc
9.  bool operator == (const T& v) const;
    //also for !=, etc
10. friend ostream& operator << (ostream&, const T&);
```

- The operator '()' can also be overloaded and used as "generator"

Key Concept #19: Member or global function?

- e.g. "a + b" can be treated as
  1. Member function : "a.operator +(b)"
  2. Global function : "::operator +(a, b)"

  ➔ Either one is fine, but...

  ➔ Compile error will arise if both are defined.
- Explicit calling overloaded operator functions
  - e.g. "a.operator +(b)" is equivalent to "a + b"
  - Or: "::operator +(a, b)"

Key Concept #20: Why "friend"?

- It's common to see "friend" in "operator <<"

```
class A {
   friend ostream& operator <<
   (ostream& os, const A& a);
};
int main() {
   cout << a1 << a2 << endl;
}
```

- "operator <<" here is NOT a member function
  - Can it be a member function?
  - Who calls "cout << a1 << a2"?
  - Is there a "operator << (const A&)" member function for class ostream?
    - Can we overload "ostream::operator <<"?

Global Function: "ostream& operator << (ostream&, const A&)"

- Since "operator << (const A&)" cannot be a member function for class ostream
  - "ostream& operator << (ostream&, const A&)" must be a global function
- "cout << a1"
  - "cout" is an object of class ostream
    - Tied to standard output (screen)
  - How is it called? ::operator << (cout, a1)
- ostream& operator << (ostream& os, const A& a) {
     return (os << a._data);
  }
  - cout << a1 << a2 ➔ cout << a2
- Declaring class A as friend of "operator << (ostream& os, const A& a)" is just for easy data access
  - Can we NOT declare it friend?

Key Concept #21: Syntax and Semantics for Operator Overloading

- There is no restriction on the semantics of the overloaded operators.
  - For example, you can overload an addition operator "+" and define it as performing "subtraction".
  - No compile error/warning.
  - But since it is counter-intuitive, you may introduce some runtime error.
- The syntax of the operators is defined in language parser (compiler). You cannot change it.
  - For example, you cannot do "a ++ b".
- The return type of operators can be arbitrary.
  - However, please make it intuitive.
- The arguments for "()" operator can be arbitrary.

Key Concept #22: Return-by-Object or Reference?

- To share the codes in operator overloading implementations, the "return-by-object" version of the operator overloading function usually reuses the "return-by-reference" one.
- e.g.

```
T operator ++(int) {   // i++
   T ret = *this; ++(*this); return ret;
}
T operator + (const T& v) const {
   T ans = *this; ans += v; return ans;
}
```

Example: Random Number Generator

```
class RandomNumGen {
 public:
   RandomNumGen() { srandom(getpid()); }
   RandomNumGen(unsigned seed) { srandom(seed); }
   int operator() (int range) const {
      return int(range * (double(random()) / INT_MAX));
   }
   int operator() (int min, int max) const { ... }
};

main()
{
   RandomNumGen rn;
   ...
   int a = rn(10);  // random number in [0, 9]
   int b = rn(100); // random number in [0, 99]
   int c = rn(10, 100);
}
```


Key Concept #23: Template Class

- When the methods of a class can be applied to various data types
  - Specify once, apply to all
  - Container classes

e.g.

```
template <class T>
class vector {
     ....
};
--------------------
vector<int>           arr;
vector<vector<int> >  arr2D;
```

➔ [note] make sure a space between “> >”

➔ [note] “template <class T> is a modifier, not a variable definition, to the class/function in concern. It can be repeated in the same file.

Key Concept #24: Template's Arguments

- Can also contain expression
  - However, the 1st argument must be class name

  e.g.

```
template<class T, int SIZE>
class Buffer
{
   T   _data[SIZE];
};
------------------------------
Buffer<unsigned, 100>  uBuf;
Buffer<MyClass, 1024>  myBuf;
```

➔ **Why not use "#define" or declare it as a data member?**

Key Concept #25: Template Function

- A common method/algorithm that can be applied to various data types

  e.g.

```
template<class T>
void sort(vector<T>&)
{
   ...
}
--------------------
vector<int> arr;
...
sort<int>(arr);
```

Notes about template function

- Template arguments
  - Any of the template arguments can be class type or expression
  ➔ `template <int S> void f() { … while (i < S)… }`
  - The template type symbol(s) can be used in function prototype and/or function body
- When calling template functions, template type symbols can be omitted
  - ```
    template <class T> void f (T a) { ... }
    int main() { f(3); f(3.0); }
    ```
- However, if there is(are) "non-type" symbol(s), or ambiguity arises, you need to explicitly specify the template symbol(s)
  - ```
    template <class T> void f() { ... }
    int main() { f(3); f(3.0); }  // compile error
    ```


Key Concept #26: Functional Object

- Remember:
  - You can overload the "()" operator for a class
  - e.g.
    ```
    class A {
       bool operator() (int i) const {
          return (_data > i); }
    };
    ```
    ➔ Note: returned type and input parameters may vary
  - What if you pass in such kind of an object to a function?
  - e.g.
    ```
    void f(const Compare& cmp,
           const T& a, const T& b) {
       if (cmp(a, b)) ...
    }
    ```
    ➔ Look like a function pointer?

Functional Object in Polymorphism

- A class/object whose main purpose is to perform a specific function
  - "()" is overloaded
  - Usually passed as reference or pointer to other functions
- Work with class inheritance (HW #1.2)
  - ```
    class Compare {
       virtual bool operator() (int) const = 0; };
    class Less : public Compare {
       bool operator() (int) const; };
    class Greater : public Compare {
       bool operator() (int) const; };
    ====================================
    void f(const Compare& p);
    int main() {
       Less a;  f(a);
       Greater b;  f(b);
    }
    ```


Example of Functional Object Applications

- Graph traveral
  - In a graph data structure, provide a generic traversal function (DFS or BFS).
  - Take a base class functional object as the parameter
    - ```
      class DoVertex {
         virtual void operator() (Vertex *) = 0;
      };
      ```
  - Define derived classes for intended actions
    - e.g. PrintVertex, Simulate, SetMark, etc
  - ➔ Same graph traversal code, different functionalities

Key Concept #27: A() vs. a()

- Let ‘A’ be a class name ➔
  A() is explicitly calling constructor
  - void f() { ... return A(10); }
  - void g() { ... A *p = new A(); }
  - void h() { ... A *q = new A; }
- Let ‘a’ be an object of class A ➔
  a() is equivalent to a.operator() ()
  - void f() { A a; ... if (a(3)) ... }
- [cf] Data member initializer
  - class A {
      A(int i): _b(i) { ... }
    };

(FYI) Functional Object and Algorithm Classes in STL

- Many algorithm and functional object classes in STL
  - for_each, find, copy, sort, swap, search, random_shuffle, power, ...etc
  - unary function, binary function, predicate
  - arithmetic, logic, comparison operations
  - ➔ For more information, please refer to: http://www.sgi.com/tech/stl/
    (See: Table of contents)

Summary #3: Template Class/Function vs. Function Overload vs. Functional Object

To maximize code reuse (less duplicated code)

- Template
  - Class template
    - Same storage method, different data types
  - Function template
    - Same algorithm flow, different data types
- Function overloading
  - Same function name, different function arguments
- Functional object
  - Same algorithm flow, different functional methods as "arguments"