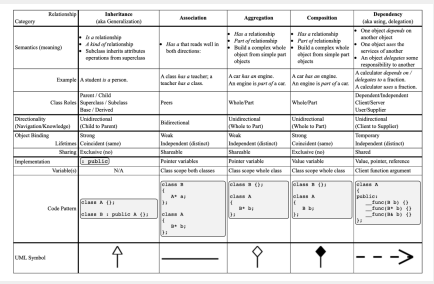
Class Relationships



Relationship / Category	Inheritance (aka Generalization)	Association	Aggregation	Composition	Dependency (aka using, delegation)
Semantics (meaning)	▸ *Is a* relationship ▸ *A kind of* relationship ▸ Subclass inherits attributes operations from superclass	▸ *Has a* that reads well in both directions.	▸ *Has a* relationship ▸ *Part of* relationship ▸ Build a complex whole object from simple part objects	▸ *Has a* relationship ▸ *Part of* relationship ▸ Build a complex whole object from simple part objects	▸ One object *depends on* another object ▸ One object *uses* the services of another ▸ An object *delegates* some responsibility to another
Example	A student *is a* person.	A class *has a* teacher; a teacher *has a* class.	A car *has an* engine. An engine is *part of* a car.	A car *has an* engine. An engine is *part of* a car.	A calculator *depends on* / delegates to a fraction. A calculator *uses* a fraction.
Class Roles	Parent / Child Superclass / Subclass Base / Derived	Peers	Whole/Part	Whole/Part	Dependent/Independent Client/Server User/Supplier
Directionality (Navigation/Knowledge)	Unidirectional (Child to Parent)	Bidirectional	Unidirectional (Whole to Part)	Unidirectional (Whole to Part)	Unidirectional (Client to Supplier)
Object Binding Lifetimes	Strong Coincident (same)	Weak Independent (distinct)	Weak Independent (distinct)	Strong Coincident (same)	Temporary Independent (distinct)
Sharing	Exclusive (no)	Shareable	Shareable	Exclusive (no)	Shared
Implementation	(- public)	Pointer variables	Pointer variable	Value variable	Value, pointer, reference
Variable(s)	N/A	Class scope both classes	Class scope whole class	Class scope whole class	Client function argument
Code Pattern	`class A {};` `class B : public A {};`	`class B { A* a; };` `class A { B* b; };`	`class B {};` `class A { B* b; };`	`class B {};` `class A { B b; };`	`class A { public: __func(B b) {} __func(B* b) {} __func(B& b) {} };`
UML Symbol	△	———	◇	◆	- - ->

Class Relationships

Unidirectional:

- all but association

Bidirectional:

- association

can change after instantiation

- aggregation, dependency, association

cannot change after instantiation

- inheritance, composition

which relationships must be created when objects instantiated: (have shared lifetime)

- inheritance and composition

which relationships can be created at any convenient time (independent lifetimes)

- aggregation, dependency, association

allow sharing some related objects

- aggregation, dependency, association

exclusive (no sharing)

- inheritance, composition

"is a" "is a kind of"

- inheritance

"has a"

- aggregation, composition, association

Class Relationships (cont)

whole-part

- aggregation, composition

implemented with dedicated computer syntax or keyword

- inheritance

Inheritance:

- add features to an existing (general) class without having to rewrite it.
  `class Foo: public Bar`

Streams

A C++ stream is a flow of data from one place to another.

three stream classes commonly used: `ifstream, ofstream, fstream`

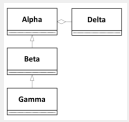
`ofstream sales("SALES.JUN");` output file associated with SALES.JUN

`fileOut.put(ch);` OR `fileOut<<ch` put what's in variable 'ch' in fileOut (an ofstream thing)

Cstrings `char s[100]; cin.getline(s, 100);`

Strings `string s; getline(cin, s);`

Q27 ch12



delta, alpha, beta, gamma - this is the order they have to be in.

Polymorphism

Five requirements: 1. inheritance 2. function overriding 3. up casting 4. a virtual function 5. a pointer (usually) or reference variable

virtual functions allow you to use the same function call to execute member functions of objects from different classes

deciding what function call executes after a program starts is polymorphism

pure virtual function `virtual void dang() = 0;` it causes it's class to be abstract and it is in the super class

an abstract class is useful when no objects should be instantiated from it.

An abstract class can: be a base (parent or super) class have concrete features (both variables and functions) that can be inherited by derived (child or sub) classes participate in (i.e., be the target of) an upcast participate in polymorphism

Object oriented model: inheritance, encapsulation, & polymorphism

overloaded function: Are defined in the same class Must have unique argument lists May have different return types

overridden functions: Are defined in two classes that are related by inheritance Must have the same name Must have exactly the same argument list Must have the same return type

Constructors

Default: foo() no arguments `foo f1;` or `foo* f2 = new foo;`

Conversion: foo(int i) one argument to be turned into the class object `foo f1(3);` or `foo* f2 = new foo(3);`

General: foo(int x, int y) anything with more than one

Copy: foo(foo& f) pointer argument

Move: foo(foo&& f) double pointer

Chap 12 stuff

a)order is not significant, follow the pattern: member-name(argument-name)

b) must initialize inheritance first, this is a function call so the number, type, and order of the parameters must match the number, type, and order of the arguments in the function definition: class-name(parameters), member(argument)

c) order is not significant; must use the second argument to access the members

d) the general pattern is class-name::function-name()

first part

```
class Bar
{
   private:
      int    count;
      int    balance;

   public:
             Bar(int a_count, int a_balance): (a)_________ {}
      friend ostream& operator<<(ostream& out, Bar& b)
      {
         . . . . .
         return out;
      }
      void display()
      {
         cout << count << " " << balance << endl;
      }
};
```

count(a_count), balance(a_balance)

second part

```
class Foo : public Bar
{
    private:
        string    name;
    public:
              Foo(string a_name, int count, int balance) :
                  (b)_____________________________________{}
        friend ostream& operator<<(ostream& out, Foo& f)
        {
            out << (c)_____________________ << endl;
            return out;
        }
        void display()
        {
            (d)_____________________;
            out << name << endl;
        }
};
```

Bar(count, balance), name(a_name)

(Bar &)f << " " << f.name

Bar::display()

Templates

When creating a template function, the template argument or variable is preceded by the keyword `typename` or `class`.

When creating a template class, it's the same ^

The template class works with different datatypes.

Template source code is placed in a header file so that it can be included with "normal" source code where it is compiled following the type expansion or substitution. (There can be more than one template argument)

When a class is "templatized" all member functions are placed in the header file: the functions can't be compiled until the templitized variable is expanded. So it can't be in a regular library

```
template <class T>
class FooBar
```

correct beginning of an operation called Foo:

```
template <class T>
FooBar<T>::Foo()
```

`while (true) CList<person>people; ...` wrong because it creates a new list every time it loops.