Name: KEY

Washington University
Department of Computer Science
CS342: Object-Oriented Software Development Laboratory
Fall 2000

Final Exam: Fall 2000

Please answer all questions clearly, concisely, and legibly. You may not use any books, notes, calculators, neighbors, e-mail, microwave relays, newsgroups, or other external sources of information. Please make sure that your name is on the exam, and that you answer the questions on both sides of all pages.

Total points: 200 (plus 15 available Extra Credit points)

1. (8 points) Name two ways references and pointers are similar in C++.
   1) both are used to *refer to* objects 2) both can be used to give functions access to objects (without passing a *copy* of the object into the function) 3) both can be used to return objects from functions without returning a *copy* of the object 4) one object may have multiple references and/or pointers referring to it 5) a pointer or reference may only refer to a single object (at a time) 6) classes can have pointers and references as data members, to refer to instances of other classes (has-a or uses-a relationships) 7) both give access to the members of the object to which they refer (although pointers use `->` and references use `.`).
2. (8 points) Name two ways references and pointers are different in C++.
   Similarities: 1) A C++ pointer can refer to different objects during its lifetime, while a C++ reference can only refer to a single object during its lifetime 2) a C++ reference must always refer to an object during its lifetime, while a pointer can be 0 (which means it points to nothing) 3) a C++ reference must be initialized but a C++ pointer does not have to be 4) a C++ pointer must be explicitly dereferenced (operator *) to access the object to which it refers, while a reference does not 5) a pointer is a first-class type: *e.g.*, a pointer can be created on the heap, while a reference cannot 6) different operators (`->` and `.` respectively) are used to access the members of the object referred to by a pointer vs. a reference.
3. (5 points) What does it mean when you declare a C++ class member private?
   It can only be accessed by methods of that class (or by functions and/or classes that are declared friends).
4. (5 points) Write a single C++ statement that 1) creates a pointer to `int` on the *stack* and 2) initializes the pointer to 0.

```
int * ptr = 0;
```

5. (5 points) Using the pointer from the previous question, write a single C++ statement that 1) creates an `int` on the heap and 2) assigns its address to that pointer.
   `ptr = new int;`

6. (5 points) Why should the destructor of a **base class** almost always be virtual?
   So that the destructors are called starting at the most derived class, even if the object is accessed through a base class pointer or reference.

7. (4 points) What is the output of the following program?

```
class X {
public:
  X (unsigned int n) : n_ (n) {cout << "X: " << n_ << endl;}
  ~X () {cout << "~X: " << n_ << endl;}
private:
  unsigned int n_;
};

int main (int, char *[])
{
  X x(3);
  X *xp = new X (10);
  delete xp;
  return 0;
}
```

```
X: 3
X: 10
~X: 10
~X: 3
```

8. (9 points) **Briefly** define each of these broad design pattern categories:

   - Creational
     Deal with initializing and configuring classes and objects.
   - Structural
     Deal with decoupling interface and implementation of classes and objects. *or* Deal with how classes and objects are composed to form larger structures.
   - Behavioral
     Deal with dynamic interactions among societies of classes and objects.

10. (5 points) How does a *pure virtual function* differ from a *virtual function*?
A pure virtual function is a virtual function that *must* be overridden by a derived class.

11. (6 points) Identify two problems with the following simple string class. (Hint: There are at least four errors to be found.)

```
#include <string.h>
class String
{
public:
  String (const char *data)
  {
    data_ = new char [::strlen (data)];
    ::strcpy (data_, data);
  }

  ~String (void) { delete data_; }
  String &operator= (const String &s) {if (&s != this)
                                  ::strcpy (data_, s.data_);
                                  return *this;}

private:
  char *data_;
};
```

1) Constructor doesn't allocate space for the trailing null in the character array. 2) Constructor doesn't check for 0 returned from `new` on failed dynamic allocation. 3) Destructor uses `delete` instead of `delete []` 4) Assignment operator doesn't check whether there is enough room in its buffer to copy the other buffer safely.

12. (6 points) List three (of the possible five) kinds of class members that *must* be explicitly initialized in a constructor's base/member initialization section.
1) Data members of user-defined classes whose constructors require arguments. 2) Reference instance variables. 3) const instance variables. 4) Base classes whose constructors require arguments. 5) Virtual base classes, in most derived class (when no default constructor).

13. (6 points) Explain how **`cin`** and **`cout`** are alike, and how they differ. **Both are standard C++ I/O streams, but cin is an input stream (for reading from the keyboard, while cout is an output stream (for printing to the terminal/screen/window).**

14. (5 points) Explain briefly what is meant by a virtual base class in C++.

    A virtual base class means that a single copy of it is shared by all of its derived classes (even under multiple inheritance, *i.e.*, in the "diamond of death" inheritance graph).

15. (6 points) Given the following class declarations, what is the access level (public, protected, or private) of data member **`Base::i_`** in an instance (object) of class **`Derived`**? Of data member **`Base::j_`**?

```
class Base
{
public:
  int i_;
protected:
  int j_;
private:
  int k_;
};

class Derived : private Base { /* ... */ };
```

    i_ and j_ are both private.

16. (8 points) Briefly explain how *cohesion* and *coupling* impact software design. **Modules should be highly cohesive, meaning everything within a module should contribute to its purpose, and be necessary for its completeness. There should be minimal coupling between modules, to maximize reuse of individual modules.**

17. (9 points) How are frameworks related to design patterns? Which would show up during design? How about during implementation/coding? **A framework contains *implementations* of design patterns. Patterns are more abstract than frameworks. Patterns can be abstract descriptions of frameworks, and can be used to document frameworks. Patterns show up in design; frameworks are used during implementation.**

18. (5 points) Why is the implementation of **`Foo::func`** below illegal in C++?

```
class Bar; // forward declaration

class Foo {
public:
  void func (const int, const Bar &) const;
private:
  const int object_state_;
};

void
Foo::func (const int i, const Bar &b) const {
  object_state_ = i;
}
```

func is a const method, but attempts to modify the object on which it is called.

19. (5 points) Using the **declaration** of **`Foo`** from the previous question, is the following call to member function **`func`** legal?

```
const Foo foo;
Bar bar;

foo.func (75, bar);
```

Yes.

20. (5 points) You are trying to link an executable, but the linker complains about the same symbol being multiply defined (i.e., in two different .o files that you are trying to link). What is one thing that could be causing this?
1) the same symbol could actually be defined in two different .cc files 2) a .cc file containing non-template definitions could be `#included` by a .h file 3) a definition could have been placed in a .h file that is included by two .cc files.

21. (5 points) What's the "big problem" with the following method definition (assuming **`buffer_`** and **`size_`** are class members of type **`char *`** and **`size_t`**, respectively)?

```
Foo& Foo::operator= (const Foo & f)
{
  if (this != &f)
    {
      if (f.size_ > this->size_)
        {
          this->size_ = f.size_;
          this->buffer_ = new char [this->size_];
        }
      ::strcpy (this->buffer_, f.buffer_);
    }
  return *this;
}
```

If `f.size_` is greater than `size_` there is a memory leak because `buffer_` is never deleted.

22. (8 points) In the following code fragment, are baz_ref and baz_ptr of the same type? What about baz_ref and baz?

```
void some_function (int i)
{
  Baz baz (i);
  Baz &baz_ref = baz;
  Baz *baz_ptr = &baz;
}
```

No and yes (baz_ptr is of type Baz *, while baz and baz_ref are of type Baz).

23. (8 points) Describe what is meant by *aliasing*, and how that can lead to problems in C++ programs.

Aliasing is when two references or pointers refer to the same object (*i.e.*, the same place in memory). If one of the references or pointers is used to modify or destroy the object without the other being updated to account for that

fact, then access violations, program corruption, etc. can result.

24. (4 points) What is the output of the following program?

```
class Base
{
  virtual void a () {cout << "Base::a" << endl;}
  void b () {cout << "Base::b" << endl;}
};

class Derived : public Base
{
  virtual int a () {cout << "Derived::a" << endl;}
  int b () {cout << "Derived::b" << endl;}
}

int main (int argc, char * argv [])
{
  Derived *derived_ptr = new Derived;
  Base *base_ptr = derived_ptr;
  base_ptr->a ();
  base_ptr->b ();
  derived_ptr->a ();
  derived_ptr->b ();
}
```

Derived::a
Base::b
Derived::a
Derived::b

25. (4 points) In the previous question, does method Derived::a hide Base::a, or override it? Does method Derived::b hide or override Base::b?

Method a overrides, method b hides.

26. (5 points) A class is considered *abstract* if instances of that class cannot be created directly by the programmer. One way to make a class abstract is to declare pure virtual functions in the class. What's another way to make a class abstract? **Declare a protected (or private) default constructor and copy constructor (and**

make all other constructors protected or private).

27. (4 points) Name one of the casting operators provided by C++, and explain what it does. **const_cast - casts away constness**
static_cast - casts a type to another related type
dynamic_cast - casts a base class pointer to a derived class pointer
reinterpret_cast - reinterprets the bits of one type as another type.

28. (8 points) Name two patterns and describe how they could be used together. Feel free to draw on examples from lectures or labs.

    For example, Factory Method was used in lab 5 to allow each derived stack class to create an Iterator of the appropriate type for iterating over itself at run-time.

29. (4 points) What is a core dump?

    A core dump is a memory image from a (previously ;-) running program. (Really good answer: ...produced before program termination because of an access violation or bus error...)(Need to get a life answer ;-) : ...the term core comes from when memory was implemented using magnetic cores).

30. (10 points) Suppose you are tasked to design the GUI "front end" of a database application. You are provided a collection of source code modules, written in C, which you can use to interact with the database. Which of the design patterns you've learned in this class would you envision coming into play in this scenario? Describe how. Discuss any and all patterns you feel are applicable.

    Facade might prove useful for presenting an alternative, easier-to-understand interface to the C modules. Database transactions would likely be composable into hierarchies, bringing Composite into the picture. The Command pattern could be used to cleanly separate menus and buttons from the functionality they invoke. The Command pattern might also help with "undoing" operations, although the Memento could be used for that as well. Iterators are so versatile that there are probably numerous ways one might be used in this situation–perhaps for manipulating all rows of a table in some way. (All the other patterns we've covered are likely equally-defensible.)

31. (6 points) The Template Method pattern is related to both Strategy and Factory Method. Choose either of the two (Strategy or Factory Method) and give a one- or two-sentence characterization of its relationship or similarity to Template Method.
Factory methods are often called by template methods. (For example, one aspect of a template method might be to create an iterator. But if the specific kind of iterator is unknown until run-time, a factory method will be used to create that iterator.)

The Strategy pattern uses delegation to allow run-time selection of an algorithm. The Template Method pattern uses inheritance to vary part(s) of an algorithm.

Extra Credit

1. (5 points) Consider the following copy constructor declaration.

   ```
   Stack (Stack s);
   ```

   This is illegal in C++. Why? (Don't explain how to fix it. Say why it's illegal.)
   Because the argument is passed by value, its copy constructor is called, so using this function would result in unbounded calls to itself (non-halting recursion).

2. (10 points) The Adapter, Bridge and Facade are all structural patterns. How does an Adapter differ from a Facade? Of Adapter and Bridge, which would you expect to show up earlier in a software design effort, and why?
   An Adapter allows two *existing*, otherwise-incompatible interfaces to work together, whereas a Facade defines a *new* interface to an existing set of components.

   Bridge would show up earlier in a software design effort, because it is a mechanism for allowing abstractions and their implementations to vary independently, which is something we can anticipate from the beginning. Adapters generally come into play in a system that has already been designed, but where two components that weren't intended to interact must now do so.

9. (12 points) Match the pattern name, above, with the intent, below:

1. Adapter	7. Iterator
2. Bridge	8. Memento
3. Command	9. Observer
4. Composite	10. Strategy
5. Facade	11. Template Method
6. Factory Method	12. Singleton

Encapsulate an operation as an object.	3
Define an algorithm skeleton in an operation, but defer some steps to subclasses.	11
Aggregate objects, hierarchically, into tree structures.	4
Capture an objects' internal state, without violating encapsulation.	8
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.	9
Decouple an abstraction from its implementation so that the two can vary independently.	2
Convert the interface of a class into another interface that clients expect.	1
Define a family of algorithms, encapsulate each one, and make them interchangeable.	10
Provide a unified interface to a set of interfaces in a subsystem.	5
Ensure a class has only one instance, and provide a global point of access to it.	12
Define an interface for creating an object, but let subclasses decide which class to instantiate.	6
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	7