Initialization sequences

Use of an initialization sequence is the only way to choose which constructor will be called for base classes and set member reference values.

Polymorphism

Polymorphic dispatching

An invocation of a virtual function through a base class pointer or reference to an object will call the function definition provided by the class of the object referred to.

Abstract base class

A base class should probably never be instantiated. Define a pure virtual function in the class.

Virtual void draw() = 0; (the compiler that draw may not be called by a client of this class.)

A body may be defined for a pure virtual function

No instance of an abstract class can be created.

If a derived class does not redefine all pure virtual functions in its base class, it also is an abstract class.

Abstract base classes are called protocol classes

Inheritance of base members

Constructors, destructor, assignment operator must be defined for the derived class.

Default members

Shallow copy & deep copy

Default Moves

Multiple inheritance

Virtual base class (class B : virtual public A)

The most derived class to invoke a virtual base class's constructor.

C++ dark corners

The compiler will generate certain member functions

Each constructor may be initialized with list

Constant and reference members must be initialized with an initialization list since they cannot be reset.

Overriding:

Non-virtual member function do not have vtable entries and so the function called is the type of the pointer or reference, not the type of object attached to the pointer or reference, breaking polymorphism.

So it is possible for a base class function to be called on a derived class object, with possibly disastrous results.

Overloading:

Avoid overloading non-virtual base class functions in derived classes

Avoid overloading virtual functions

If a derived class redefines a base class virtual function which has overloads in the base. That will hide the base class overloads that are inherited.

Avoid using default parameters in virtual functions:

Parameters don't have vtable entries. This results in a derived class using base class defaults even though the derived class defined different values for the defaulted parameters.

Always provide a virtual destructor if your class may be used as the base class for a derivation:

Multiple virtual inheritance of implementation:

Value types

Operations normally expected from the class

Construction;;Destruction;;Array;;Passing to functions;;Observing and modifying object state;;Assignment of objects;;Coercion of objects Operator symbolism

Memory Model

Static memory

Public global functions and data

Defined outside any function (globals) and initialized before main is entered.

Private global functions and data

Global data and functions are made private by qualifying as static, otherwise they are public

Local static data

Memory allocations local to a function, but qualified as static

Stack memory

Main stack frame

Function called by main stack frame

More stack frames

Current function stack frame

Heap memory

Allocated heap memory

Free heap memory

Templates

Agenda

```
Template<class T>
T max( const T& t1, const T& t2){}
```

- Template specializations
- Containers

Partial specialization:

```
Template<class U> Widget<U,double>
{ … };
Widget<String,double> yourWidget;
```

Complete specialization:

```
template<> Widget { … };
Widget<string,double> ourWidget;
```

Thread Guard

```
void test(std::mutex&  mtx)
{for(int i=0; i<5; ++i){
   {std::lock_guard<std::mutex>  guard(mtx);
    std::cout << "\n  in " << __FUNCTION__;}
std::this_thread::sleep_for(std::chrono::milli
seconds(100));}}
```

Define Thread

```
std::thread t(test, ref(mtx));
```

Call Once

```
void calls()
{static std::once_flag beenCalled;
  std::call_once(beenCalled, one);
  two();}
```

Concurrency

```
HANDLE handle[3];
handle[0] = t1.native_handle();
handle[1] = t2.native_handle();
handle[2] = t3.native_handle();
::WaitForMultipleObjects(3,
handle, true, INFINITE);
```

Callable objects

```
template <class CallObj>
void Executor(CallObj& co)
{co();}
```

Functor

```
std::string testFunction(size_t lineNumber,
const std::string& msg)
{std::ostringstream out;
 out << "\n  testFunction invoked from line
number " << lineNumber << " - " << msg;
 return out.str();}
class Functor {
public:
 Functor(size_t lineNumber, const
std::string& msg) : ln_(lineNumber),
msg_(msg) {}
 std::string operator()() { return
testFunction(ln_, msg_); }
private:
 size_t ln_;
 std::string msg_;
};
```

Lambdas

```
std::function<std::string()>
CreateLambda1(int i)
{std::function<std::string()> f = [=]()
 {std::ostringstream convert;
  convert << i;
  return convert.str();};
 return f;}
std::function<std::string(int)>
CreateLambda2()
{std::function<std::string(int)> f = [](int i)
 {std::ostringstream convert;
  convert << i;
  return convert.str();};
 return }
```

Software Design Principles

Liskov Substitution Principle(LSP)

Inheritance is a powerful mechanism, used by most Object Oriented Languages. Here's a statement:

"Functions that use pointers or references statically typed to some base class must be able to use objects of classes derived from the base through those pointers or references, without any knowledge specialized to the derived classes."

LSP supports great flexibility in the implementation of programs using OOD.

Open/Closed Principle (OCP)

OCP is a very broad principle. It says that we should strive to extend or wrap existing code rather than change the code to meet new requirements. Here's a statement:

"Software entities [classes, packages, functions] should be open for extension, but closed for modification.

This is a very important goal when we design software. In order to satisfy new or changing requirements, we try to extend, through inheritance or template parameterization, existing code, or wrap parts of the code with new classes, using composition or aggregation. That is not always possible, but that should be our first attempt to satisfy a need for change.

Dependency Inversion Principle(DIP)

Making references to concrete classes and functions binds the caller to a specific design and implementation, and makes that software "brittle", e.g., hard to change without causing a lot of breakage. To avoid these brittle designs we use DIP:

"High level components should not depend upon low level components. Instead, both should depend on abstractions."

"Abstractions should not depend upon details. Details should depend on the abstractions."

Here, abstractions refer to interfaces and object factories.

Interface Segregation Principle(ISP)

When a class has many clients with differing needs, it is tempting to extend the class's interface and implementation to satisfy the needs of each new client. However, the other clients may not need those new public methods, but must be recompiled every time a method changes, even if they don't use the method.

"Clients should not be forced to depend upon interfaces they do not use.

We usually satisfy the ISP by factoring a class into a core and specialized parts that can serve the specialized needs of each client. One common way to do that uses multiple inheritance, allowing selection of what's needed for each application.

Vector

```
std::cout << show(vecStr);
std::cout << "\n  vector size = "
<< vecStr.size();
std::cout << ", vector capacity =
" << vecStr.capacity();
std::cout << "\n  first element =
" << vecStr.front() << ", last
element = " << vecStr.back();
std::cout << "\n  second element =
" << vecStr[1];
```

Map

```
std::map<int, std::string>
MapIntStr mapIntStr { { 5,
"five" }, { 2, "two" }, { 7,
"seven" } };
std::cout << show(mapIntStr);
mapIntStr[key] = value;
showItem(*mapIntStr.find(key));
auto item = std::make_pair(key,
"fore");
auto result =
mapIntStr.insert(item);
int numRemoved =
mapIntStr.erase(key)
MapIntStr::iterator pos;
  for (pos = begin(mapIntStr);
pos != end(mapIntStr); ++pos)
  {
    if (pos->second == value)
      pos = mapIntStr.erase(pos);
  }
  std::cout << show(mapIntStr);
```

Deque

```
std::deque<double> deqDbl{ -1.0,
3.1415927, 1.5e-3 };
show(deqDbl);
deqDbl.push_front(d);
deqDbl.push_back(d);
deqDbl.pop_front();
deqDbl.pop_back();
```

Interface

First answer

Public members of a class

Second answer

public members of a class plus global functions packaged with the class. Packaged means in the same header file and in the same namespace.

Third answer

Abstract class with no data members, no constructor and at least one pure virtual function. • An abstract class has the same role as a C# or Java interface. • It provides a means to use an implementation class but only binds to the abstraction provided by the interface.