

OBJECT ORIENTED DESIGN PRINCIPLES CHEAT SHEET





What is OOD?

Design of software using classes and class relationships: inheritance, composition, aggregation, and using, to provide a software structure.

Main elements of OOD S.W Design:

- a. Inheritance** supports sharing of code between base and derived classes, and most important, Liskov Substitution “functions that accept pointers and references to base classes must accept pointers and references bound to derived class objects without using any knowledge specialized to those derived classes.
- b. Composition** allows a composing class to contain instances of other types that provide help in implementing the classes’ responsibilities. This decomposes classes into parts each of which should support the Single Responsibility Principle.
- c. Aggregation** allows an aggregating class to create instances of other classes, as needed, and bind them to member pointers. It serves a purpose similar to composition, but supporting dynamic creation and lazy loading.
- d. Using** allows a using class to access and employ instances of classes created by some other

Relationships between C++ Classes

Relationship	Diagram	Code	Explanation
Inheritance D "is-a" B		<pre>class D : public B { ... };</pre>	Derived class D is a specialization of the Base class B. D inherits all the members of B except constructors
Composition Ownership, P is "part-of" C		<pre>class C { ... Private: P p; };</pre>	Composite class C owns, or contains, a part class P. P is created and destroyed with C. The interface of P is visible only to C, not its clients.
Aggregation Ownership, P is "part-of" A		<pre>class A { ... Void fun() { P* ptrP = new P(); ... } };</pre>	The Aggregator class A owns a part class P. P is created by a member function of A, and so its lifetime is strictly less than that of A. A is expected to destroy P.
Using Referral: U uses R through a reference		<pre>public class U { ... public void register(R& r) { // use r } };</pre>	A class U uses instance of class R, to which it holds a reference. R is created by some other entity and a reference to it is passed to some member function of class U.

Constructors with syntax:

```
class X
{
    public: X(); // default constructor

    x(const std::string& msg); // another constructor

    X(const X& x);           // copy constructor

    X& operator=(const X& x); // copy assign operator

    ~X(); // destructor

    std::string getMessage(); //mt
```

```
private:

std::string* pMsg;    // private data

};
```

Invoking definition:

```
X::X() : pMsg(new std::string()) {}
X::X(const std::string& s) : pMsg(new std::string(s)) {}
X::X(const X& x) : pMsg(new std::string(*x.pMsg)) {}
X& X::operator=(const X& x)
{ if(this != &x { delete pMsg; pMsg = new std::string(*x.pMsg);} return *this;}
X::~~X(){delete msg}
std::string getMessage(){return *pMsg;}
```

Describe the syntax of a C++ package. Why do we build projects using packages?

A C++ package consist of two files, a header and an implementation file. The header contains a prologue, manual page, maintenance history, and declarations of classes and global functions. For templates and inline functions, the implementations also appear in the header file. The header is enclosed in preprocessor selection controls ensuring it is included only once in the composite code file passed to the compiler for translation. The implementation file contains all the implementations of functions declared in the header if not template or inline. It also contains a test stub, enclosed with preprocessor selection controls. This allows the test stub, used for construction testing, to be excluded without changing any of the package code. We use packages to divide large programs into relatively small parts that are relatively easy to design, implement, and test. Other benefits of using packages are that dependencies are easier to control and communication becomes easier to manage. For almost every other class, the compiler generated operations will not be correct, and you must either provide them or disable them. Classes that contain pointers are examples of where you need to do this.

Describe what happens when a C++ package is compiled. The preprocessor reads the package implementation (.cpp) file, building a composite temporary source file for the compiler. Each of the #includes causes the text of the included file to be placed inline at the site of the include. Each #ifdef - #endif causes code to be included or not included in the temporary source file. Each #ifndef #define - #endif causes an included header to be included only once. The compiler processes one implementation file at a time and, if there are no translation errors, generates an .obj file for each one. If there are errors, error messages are emitted and compilation of that .cpp ends. Subsequently the linker will bind all the obj files and any cited libraries into an executable file, static library, or dynamic link library. Should any executable build include a DLL lib file, the loader will link that to the executable image at load time, or optionally at run-time due to an explicit load library command.

Function overloading occurs when a function name is used more than once with different arguments in the same scope. These are treated as distinct functions by the compiler. Overloaded functions are resolved at compile-time using name mangling. Overloading allows the same conceptual operation to be invoked with different arguments. Class constructors are a good example.

Function overriding occurs when a base class virtual function is redefined, in a class derived from the base, with the same argument types and the same or covariant return type. When invoked using a base class pointer, the overridden function belonging to the type of the referenced object is called. Overrides are resolved at run-time using the Virtual Function Pointer Table (VFPT). Overriding allows a derived class to modify the behavior of its base in specified ways, without modifying the base interface.

You are preparing to design a class that may need to be extended in the future, in ways you can't anticipate now. How might you do that? Here are four common ways

- Use templates with template policies. We can modify the way a template class behaves by substituting appropriate policy classes for different application needs.
- Provide template method(s) that accept callable objects.
- Use inheritance and polymorphic classes. We can extend an inheritance hierarchy simply by adding new derived classes that override base virtual functions. The parser is an outstanding example of the flexibility afforded by the use of inheritance.
- Use interfaces and object factories. This is an elaboration of the item, c., above. If we implement an interface with code that compiles to a dynamic link library, then we can extend the existing functionality by adding a new library that implements the interface and modifying the small amount of code in the object factory to create instances needed in the new library. See XmlElement
- Add additional interfaces and support querying from one interface for another supported interface.

C++ Dark Corners:

1. Compiler generated functions
2. Constructor initialization
3. Overriding errors
4. Overloading out of scope
5. Using default parameters with virtual functions
6. Virtual destructors
7. Multiple Inheritance

```
Str* sptr = new Str; // initialize object on heap
Str s2 = s1; // copy Str s2(s1);
Str s[2] = { s1, s2 }; // copy state
inArr Str *sptr = new Str(s1); // cs heap
```

Use of an initialization sequence is the *only* way to chose which constructor will be called for base classes and set member reference values. We will see in the demonstration code that this is

```
class D : public B { // public interface private: ostream& _out; int _size; double *darray; };
D::D(ostream& _out, int _size, std::string& s) : B(s), // initialize base class _out(_out), _size(_size), darray(new double[_size]) { }
```

Constructor:

```
Copy: D(const D& d) : B(d) { std::cout << "\n D copy constructed"; }
```

Note this uses the B copy constructor to take care of C. U is simply used by D, so should not be copied. D's copy constructor will be called when an instance of D is passed or returned by value to a function or when we explicitly invoke it as here: **D d1; D d2 = d1; // copy construction**

```
Move: D(D&& d) : B(std::move(d)) { std::cout << "\n D moved constructed"; }
```

Note this uses the B move constructor to take care of C. U is simply used by D, so should not be moved. D's move constructor will be called when an instance of D is return by value from a function that created the local instance of returned D or when we explicitly invoke it as here: **D d1; D d2 = std::move(d1); // move construction may invalidate d1**

2) The code below assumes that D contains a std::string member msg_.

```
D::D(D&& d) : B(std::move(d)), msg_(std::move(d.msg_)), pA(d.pA)
{ std::cout << "\n move construction of D"; d.pA = nullptr; }
```

3) D& D::operator=(D&& d)

```
{ std::cout << "\n move assignment of D"; if (this == &d) return *this; B::operator=(std::move(d));
msg_ = std::move(d.msg_); if (pA) delete pA; pA = d.pA; d.pA = nullptr; return *this; }
```

//----< copy construction of D >-----

```
D::D(const D& d) : B(d), msg_(d.msg_) {
std::cout << "\n copy construction of D"; if (d.pA)
pA = new A(*d.pA); else pA = nullptr; }
```

What const implies is determined by where you find it:

–Str(const Str& s); is a contract that the argument s will not be changed. The compiler attempts to enforce the contract.

```
char operator[](int n) const;
```

implies the state of the object on which the operator is applied will not change. Again, the compiler attempts to enforce the contract. Thus:

`constStrcs= "a constant string";cs[3] = 'a';` will fail to compile because the compiler will call the constversion of `operator[]` on the `const str` disallow

When you define a class, under what conditions will you chose to implement copy, assignment, and destruction operations? When would you decide not to provide those? If you don't, what happens if code using your class attempts to copy or assign instances?

You choose to implement copy, assignment, and destruction operations only if the class bases and members do not have correct copy, assignment, and destruction semantics. You may also choose not to provide them for the case cited above, but must then qualify the copy and assignment operations with `= delete`; If you don't provide those operations:

a. If copy and assignment are not marked delete, then the compiler will generate them if needed by doing base-wise and member-wise copy and assignment. **b.** If copy and assignment are marked delete, implied copy and/or assignment operations will fail to compile. Examples of these rules.

c. If the class has no bases and its data members are all primitive types and/or STL containers, then their copy, assignment, and destruction semantics are correct. So you should choose to let the compiler generate these operations.

d. If the class has a member pointer to an instance allocated on the heap, then you must provide them or qualify copy and assignment as `= delete`. In this case you have to define the destructor to avoid a resource leak.

What class methods may be generated by a standard conforming C++ compiler? When would those operations be generated? The following operations are, if needed and not declared, provided by the compiler. Each delegates its operation to each of the class's bases and data members

a. Default constructor: Provided only if no constructors are declared for the class.

b. Copy constructor: Always provided if not declared by the class and used in application code.

c. Move constructor: Only provided if not declared by the class and copy and destructor operations are not declared by the class.

d. Copy assignment: Always provided if not declared by the class and used in application code.

e. Move assignment: Only provided if not declared by the class and copy and destructor operations are not declared by the class.

f. Destruction: Always provided if not declared by the class.

Given the compound object described in the diagram, below, describe all the operations that occur when the object is created and when it is destroyed. Do you have any control over the specific operations invoked?

The composite object `d` constructed from `D` contains, within its memory footprint, an instance of the base class `B` which in turn contains an instance of the composed class `C`. Construction of `D` will always cause `B` to be constructed (whether we explicitly invoke the `B` constructor or not) and construction of `B` will always cause `C` to be constructed (whether we explicitly invoke the `C` constructor or not). When `D`'s constructor starts, it immediately calls `B`'s constructor and doesn't complete until `B` construction finishes. When `B` construction starts it immediately calls `C`'s constructor and doesn't complete until `C`'s construction finishes. The same sequence occurs on destruction. Designers can always control how construction proceeds by explicitly invoking constructions on bases and member instances in constructor initialization sequences. They have no control over destruction other than the code that is supplied by each class since there is only one destructor defined by each class. For default construction there is no need to do that as the compiler will always invoke default constructors on bases and members. For copy and move constructions the designer must explicitly invoke the base and member constructors. Assuming a default construction of `D` we get default construction of `B` and `C`. When destroyed, the destructor of `D` calls the destructor of `B` which, in turn, calls the destructor of `C`. Since `U` is used but not owned by `D`, none of the `D` operations directly causes `U` to be created or destroyed.

Describe the syntax and semantics for modern C++ casts.

Answer: There are four modern casts:

a. `T t = static_cast<T>(s)`, where `S s`; This creates a new instance of `t`, casting from an instance of `S`. The cast operator is either generated by a built in C++ conversion rule for native types, or by a cast operator defined by

the S class, e.g., `S::operator T()`; If no cast operator is defined by the class, but T has a promotion constructor taking an instance of S, that will be called. Otherwise, compilation fails.

b. `T* pT = const_cast<T*>(pConstT)`, where `const T* pConstT`; pT is typed as a pointer to non-const T, and may be passed to non-const functions. You should only do that if you are certain that the function will not change the state of the referenced instance of T.

c. `T* pT = reinterpret_cast<T*>(pS)`; where `S* pS`; The pointer pT references an instance s, referenced by pS, but treats it like an instance of T, e.g., uses T's type rules on the S instance. We normally only use `reinterpret_cast` to pack data into byte arrays for serialization or some similar process.

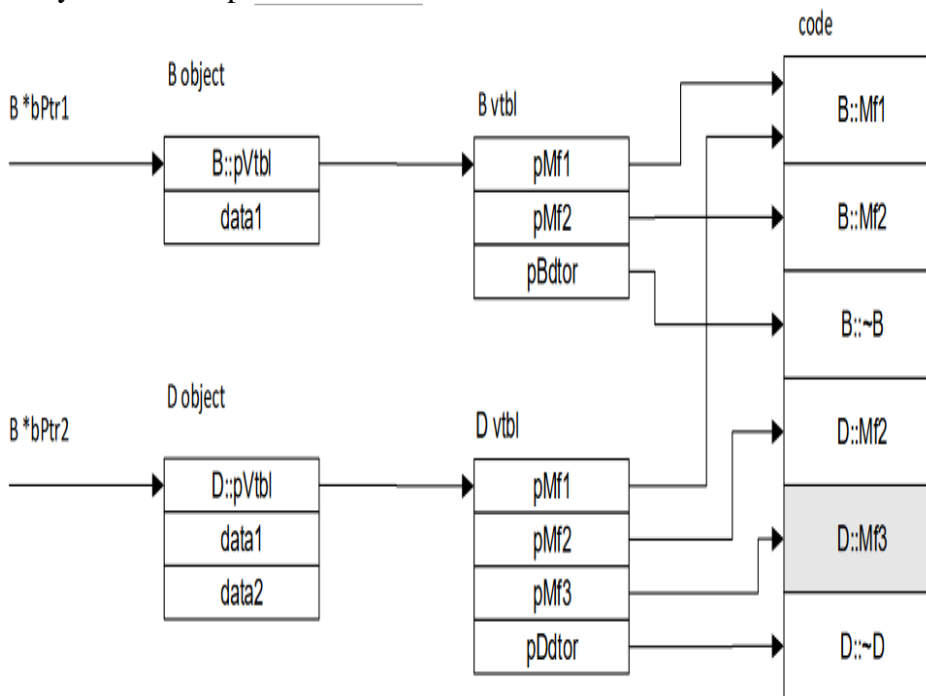
d. `D* pD = dynamic_cast<D*>(pB)`; where `B* pB = new D`; The pointer, pD, has the same address value as pB if D is derived from B. Otherwise its value is nullptr (0 in pre C++11 code). This supports accessing public members of D that are not inherited from B. Note: no cast is required to bind a pointer or reference of a derived type to a pointer or reference of its base type.

Describe the dynamic binding process used by C++ to implement polymorphism.

Answer: Each class with at least one virtual function has a Virtual Function Pointer Table (VFPT). That table directs calls, to virtual functions that have not been overridden, to the base function code (usually 1) at run-time, using dynamic dispatching using the VFPT.

Any call to a virtual function that has been overridden will be directed to the derived class code, by VFPT dispatching process. It is the VFPT that allows a derived class to specialize the behavior of its base. That specialization is the essence of polymorphism. All calls to non-virtual functions are bound at compile-time and are not part of the class's polymorphic behavior, although they do participate in the Liskov Substitution process. Bjarne Stroustrup, the author of the C++ language, describes the use of template parameterization as static polymorphism. However, that does not involve any dynamic binding, e.g., all calls to template functions (either global or member) are bound at compile-time. Grading note: `Dynamic_cast` checks the inheritance chain to determine if a derived reference to an object is valid. It does not do any dynamic binding.

Virtual Function Pointer Table (vtbl) is a table of pointers to virtual methods created for every class that contains at least one virtual method. It is used for polymorphic dispatching, e.g., invoking a derived class method via a base class pointer, to which it is bound. Vtbls are created at compile time, one for each class with virtual methods. Vtables are used at run time to dispatch virtual function calls whenever we invoke a method of a derived class by means of a pointer to its base class.



SR (Single Responsibility Principle) Each software entity: function, class, package, and module should have one responsibility.

OCP(Open Closed Principle) Software entities (classes, packages, functions) should be open for extension, but closed for modification. It doesn't make much sense to apply OCP to Application-Side software. But for Solution-Side software we expend a lot of effort to make code reusable, and that is where we expect to apply OCP. We do that by building template classes, Inheritance hierarchies, and using Hooks and Mix-ins.

LSP(Liskov Substitution Principle) Any function accepting a pointer or reference to an instance of a base type will accept a pointer or reference to a class derived from the base. The function's use of the pointer or reference will not depend on any knowledge specific to the derived class. **C++ has two features essential to Liskov Substitution.** a. All inheritable functions² of the base class are inherited by every derived class. That means that any calls made by a function accepting a base class pointer or reference will succeed with a pointer or reference to an instance of a class derived from the base. b. Each class with at least one virtual function has a Virtual Function Pointer Table (VFPT). That table directs calls, to virtual functions that have not been overridden, to the base function code. Any call to a virtual function that has been overridden will be directed to the derived class code, by a dispatching process that uses the VFPT. It is the VFPT that allows a derived class to specialize the behavior of its base.

Liskov Substitution supports: flexibility afforded by substitution ability to create reusable components that interact with applications without any application specific code.

DIP:(Dependency Inversion Principle) High level components should not depend upon low level components. Instead, both should depend on abstractions. DIP essentially implies that we use interfaces and object factories of implementation of components. The advantages of using DIP are that client code is isolated from any implementation changes made in components it uses if it accesses them using object factories and interfaces. It also means that parts that implement the interface can be interchanged without affecting other parts of the system.

How do you support DIP in a C++ program?

High level components should not depend on low level components. Instead, both should depend on abstractions. To break dependencies between a calling high-level component, H, and a low-level component, L, the low level component provides an **interface**, IL, and an **object factory** LFactory that has a creational function, $IL^* pL = LFactory::create()$; Now, both H and L depend on the text of IL. The high level component also depends on the create() interface, but not on its implementation.

If we build L as a **Dynamic Link Library (DLL)**, then any change made to L to fix errors, performance issues, or add new functionality, will not affect H, as it only depends on the text of IL and the factory interface. When H is started, it will load L's DLL and does not have to be re-compiled or re-linked. Grading Note: Weak version of DIP uses interfaces and object factories to avoid recompilation when lower level changes. Strong version of DIP also implements lower-level components as dynamic link libraries to avoid relinking after change.

Code constructs that may cause substitution errors:

- Overloading base class methods in a class derived from the base – causes hiding
- Overloading base class virtual functions – causes hiding when those functions are overridden in the derived class.
- Redefining in a derived class a non-virtual method in the base – non-virtual method calls are based on the type of the pointer, not the type of the object bound to the pointer.
- Failure to provide a virtual destructor in a class that will serve as a base for inheritance – causes incomplete destruction of derived objects on the heap when deleted from a base class pointer bound to the derived instance.
- Use of default parameters in both base and derived with different values if accessed through a base pointer or reference – will use values determined by the static type of the pointer or reference, not the type of the bound instance.

Template: A template policy is a template parameter⁶ that is used to modify the operation of a template class, e.g., it does not replace the primary functioning of instances of the class, but only modifies how they carry out their operations. A trait is an alias for a template parameter type that provides an immutable name for that type, regardless of how it is instantiated by an application. Traits allow us to write code in a template function or class that depends on the specific type of a template parameter without knowing how the application will instantiate it.

Why are the methods of a template class placed in the header file of its package? A C++ compiler can't generate code for a template class or function until the using application instantiates it with a known type. So templates are checked for syntax when compiled, deferring code generation until an application includes the template's header file and is compiled. The compiler does not have access to the template's implementation file (it only sees the included header files' text) so you must provide all the template implementation details in the header. This is the sequence of assertions that demonstrate the need:

- When templates are compiled their parameter types are not known and so no code can be generated. Only simple syntax checking takes place.
- Only when an application instantiates the template type or function does the compiler know the type. Then, if and only if, it has the complete definition of the template can it generate code for the instantiation.
- Since only header files are included by the application code, the entire definition of the template must be found in one of its included headers.

Lambda: What are the advantages of using a lambda in place of a function? Lambdas are a quick way to write the equivalent of a function: a. Lambdas can be defined locally, including inside a function, so that it is obvious what happens when invoked. b. They can be stored or enqueued for later execution – see the Enqueued WorkItems in CppThreadTechniques. c. They can be passed to and returned from functions. d. They can carry their arguments as captured variables.

Write a function that will create a child thread using the same function. Is there a problem with that? If so, can you write code to avoid the problem? Why would you write such a function? If you don't provide some way to limit the number of threads, the program will keep creating threads until it runs out of stack memory.

```
void threadProc(size_t count) {
    std::cout << "\n entered threadProc with count = " << count;
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    if (count++ < 10) {
        std::thread t(threadProc, count);
        t.detach(); }
    std::cout << "\n leaving threadProc with count = " << --count; } using namespace Utilities;
using Utils = StringHelper; void main() {
    Utils::Title("MT3Q7 - Recursive thread creation");
    std::thread t(threadProc, 1); t.detach();
    std::cout << "\n press any key to exit: \n"; }
```

You write a threadProc like this when you want to create threads in a specific sequence, probably doing some setup needed for each. Note that you are guaranteed that each thread starts running before the next thread is created.

ct function call). Sharing data between processes has the

Write all the code for a function that searches for a string of text in a text file, asynchronously2. Assume that you pass the file name as a function argument.

When you run the function asynchronously, the caller returns quickly, perhaps before the passed lambda finishes execution.

```
using FPtr = bool (*)(const std::string&, const std::string&);
bool findText(const std::string& text, const std::string& filePath) { std::ifstream in(filePath); if (!in.good()) {
    std::cout << "\n can't open \"" << filePath << "\""; return false; } std::string fileText;
while (in.good()) { char ch; in >> ch;
if(ch != '\n') // text may span multiple lines
    fileText += ch;
This while loop can be replaced by:
Std::ostringstream out;
Out << in.rdbuf();
Std::string fileText = out.str();
We will discuss this when we talk about iostreams.
}
```

```

size_t pos = fileText.find(text);
return pos < fileText.size();
} std::string text = "main";
std::string fPath1 = "../Test/Copy_MT2Q4.h";
std::string fPath2 = "../Test/Copy_MT2Q4.cpp";
std::future<bool> f1 = std::async(std::launch::async, findText, text, fPath1);
std::future<bool> f2 = std::async(std::launch::async, findText, text, fPath2);

```

Describe several ways to pass arguments to a thread you are creating and to retrieve results from the thread. You may use code in your discussion, but are not required to do so. Answer: You can pass arguments to a thread in the following ways:

- Add each argument to a thread constructor statement, following the thread's callable object, and separated by commas.
 - You can do exactly the same thing using the `async` template function.
 - You can pass the thread a lambda with the arguments it needs as captured variables.
 - You can pass a functor instance to the thread with the arguments it needs as member data, perhaps passed into the functor's construction statement.
 - Use shared `BlockingQueue` to pass a sequence of inputs.
- You can retrieve results from the thread in the following ways:
- Use the `async` template function that returns a `future<T>` where `T` is the type of the return value. You eventually call the future's `get()` function which will block until the answer is ready.
 - You can pass, using `std::ref()` a container for the result and pass by `std::ref()` a `std::mutex` that the thread function uses when it computes the result, and which the using code acquires before accessing the result. Note that the `std::ref()`s are required because arguments passed to a thread are passed by value.
 - Use a shared `BlockingQueue` to return a sequence of results.
 - Use a callback to deposit result in some thread-safe location.

Suppose we have a FileMgr class that, given a path and a set of patterns, navigates the directory subtree rooted at the path, and finds all the files matching any of the patterns. How do you design it so that applications can use FileMgr without putting any application specific code in the FileMgr package?

Method #1 – use events and put application code in event handlers FileMgr provides two hooks, `IFileEventHandler` and `IDirEventHandler`. Applications derive from these and register the derived instances using `FileMgr::regForFiles(IFileEventHandler*)` and `FileMgr::regForDirs(IDirEventHandler*)`. When a file is discovered by FileMgr it invokes its collection of `IFileEventHandlers` like this: `pEvtHandler->execute(fileSpec)`; When a directory is discovered by FileMgr it invokes its collection of `IDirEventHandlers` like this: `pEvtHandler->execute(dirSpec)`; See FileMgr project in `MTCodeSupplement-S16` for all the details.

Method #2 – put application code in derived class virtual function overrides An alternate method is to make virtual functions, say `virtual file(const std::string& fileSpec)` and `virtual dir(const std::string& dirSpec)` in FileMgr that pass the `fileSpec` or `directorySpec` when a new file matching one of the patterns is discovered or a new directory is entered. FileMgr should also declare its destructor as virtual. An application can simply derive from FileMgr and override the the virtual methods to do application specific stuff. We don't need to make the base FileMgr application specific at all. This way you get to reuse the directory navigation code with no change. Note that you cannot simply have a FileMgr function return a collection of all the results. That is certainly independent for applications where that will work. However, for applications like Project #3, where you have to enqueue `fileSpecs` as you find them, it will not work

Write a class declaration for a TestDataFormatter class. The formatter accepts a test driver name, author, date-time, test result, and a test specific instance of an unspecified type. It provides a method to return a formatted string1 for display purposes. Answer:


```

template<typename AppType>
class TestFormatter {
public:
    void name(const std::string& name);
    void author(const std::string& author);
    void date(const std::string& date);
    void result(bool rslt);
    void appType(const AppType& appType);
    std::string format();
    std::string format(ITestResult<AppType>* pTestResult);
private:
    std::string name_;
    std::string author_;
    std::string date_;
    bool result_;
    AppType appType_; // will use to hold log items
};

```

Describe several ways to pass arguments to a thread you are creating and to retrieve results from the thread. You may use code in your discussion, but are not required to do so. Answer: You can pass arguments to a thread in the following ways:

- a. Add each argument to a thread constructor statement, following the thread's callable object, and separated by commas.
- b. You can do exactly the same thing using the async template function.
- c. You can pass the thread a lambda with the arguments it needs as captured variables.
- d. You can pass a functor instance to the thread with the arguments it needs as member data, perhaps passed into the functor's construction statement.
- e. Use shared BlockingQueue to pass a sequence of inputs.

You can retrieve results from the thread in the following ways:

- f. Use the async template function that returns a future<T> where T is the type of the return value. You eventually call the future's get() function which will block until the answer is ready.
- g. You can pass, using std::ref() a container for the result and pass by std::ref() a std::mutex that the thread function uses when it computes the result, and which the using code acquires before accessing the result. Note that the std::ref()s are required because arguments passed to a thread are passed by value.
- h. Use a shared BlockingQueue to return a sequence of results.
- i. Use a callback to deposit result in some thread-safe location.

NAMESPACE: A namespace can be declared at global scope and within the scope of any other namespace. It may not be declared in any other scope. Note that multiple namespaces can be declared in the same scope, e.g., a namespace A may contain more than one lower level namespace, B, C. You should never write a using namespace declaration in a header file. That forces every user of your package to have that declaration. You may write a using namespace declaration in an implementation file in order to use types and variables from that namespace without constantly qualifying them with the namespace.

