CPP301-D

Module 1

Class Design & Implementation

1 Encapsulation

1.1 Managing Complexity

Implementation of object-oriented techniques in software engineering have proven to be successful in building and managing large and complex applications. There are key principles why using an object-oriented programming language like C++ is preferred over a functional programming language like C for building such applications. While C is still a powerful language, there is a limitation on the complexity of the applications that can be developed using it. As computer hardware increases in processing power and speed, requirements for better and more powerful software would also increase. As applications get larger, they would also get more complex to develop, manage and update. Even though there are many reasons why an object-oriented language should be used to build modern software applications the following are key features that such languages provide.

Key features of an object-oriented programming language

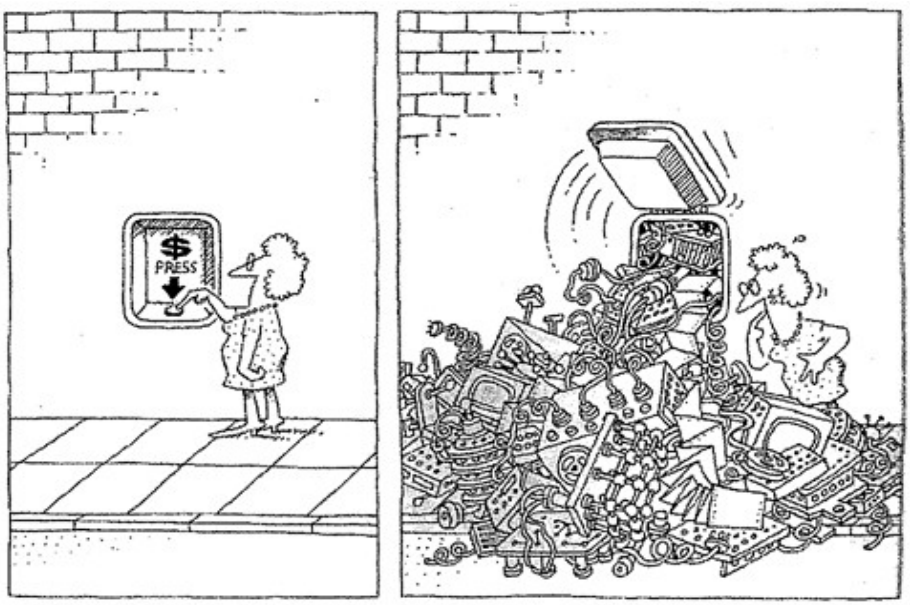
1. Manage software complexity
2. Promote code reusability
3. Allow Extensibility

We cannot reduce complexity but we can simulate simplicity by managing it. We can use the divide and rule concept by decomposing a complex system into smaller and more manageable sub-systems by classification. In object-oriented languages, a class keyword is used to define classifications. We can hide complex operations and data structures in classes and expose simple interfaces, a technique called encapsulation. Encapsulation also allows us to implement internal control to ensure stability and integrity in each sub-system. A complex system may be dangerous to change since a small fault anywhere may result in total breakdown. Smaller and simpler sub-systems would be easier to maintain and update safely. Changes to the hidden internals of one sub-system would rarely have impact on other sub-systems as long as the existing public interface is maintained.

Managing complexity

1. Decompose complex system into smaller and simpler sub-systems
2. Hide and protect complex data structures and operations
3. Expose public interfaces that are simple to use

Encapsulation: Internally complex, externally simple



The task of a software development team
is to engineer the illusion of simplicity

1.2 Component-Based Architecture

One way to implement an application is to think of it as a machine that is constructed from a set of components. Each component can be developed and tested individually. The components are then integrated together to form the internals of the machine. A major principle that has been proven to be successful is to make use of a component-based architecture in software designs. One common component that is usually found in applications is a message logging component. In this chapter you will implement a basic logging component following the principle of encapsulation. Create a new empty project in Visual Studio using the following information.

Visual C++ Project Information

```
Project Name: symbion
Project Type: C++ | Empty Project
Location    : <repository_directory>\src
Solution    : module1
```

Add a header file named **symbion.h** to the project. Use **#pragma once** in all header files to ensure that the contents of the file will be processed once regardless of how many times it has been included since it is considered an error to re-declare the class once or more during compilation of the source files. Since header files include other header files it is impossible to guarantee the same header file will not be directly or indirectly included twice or more. We will also use **symbion** as the namespace since we will turn this project into a shared library to avoid naming conflicts when used with other libraries in another project.

Basic header content: symbion.h

```
#pragma once

namespace symbion {

}
```

You can use either **struct** instead of **class**. The only difference between these two in C++ is that by default, all members in a struct is public and all members in a class is private. However this is not important since the access level can be changed anytime and any number of times within the scope by using access modifier keywords **public**, **protected** and **private**. Since encapsulation is based on the hiding of details private would be the more appropriate default so you should always use a class.

Changing access level between members

```
struct|class class_name {
        <-- Put public|private members here (not recommended)
public:
        <-- Put public members here
protected:
        <-- Put protected members here
private:
        <-- Put private members here
};
```

In the namespace we will declare a class named **CConsoleLogger**. Class names don't have to prefixed with an extra **C** but is a common naming convention to easily identify whether a symbol is referencing a class. The class has a **Write** member function that accepts a constant char pointer argument named **message** and an integer argument named **logType**. A constant pointer means that it can only be used to access memory it points to but cannot be used to modify that memory.

Class declaration

```
class CConsoleLogger {
public:
    void Write(const char * message, int logType);
};
```

Add a source file named **consolelogger.cpp** to implement the members of the class. We need to include the header file so that the implementation can be matched against the declaration. Use the scope resolution operator (**::**) to associate the function to the class and namespace. Alternatively you can place the member implementations within the namespace itself so that you do not need to attach it to every member function name.

Class implementation file: consolelogger.cpp

```
#include "symbion.h"

void symbion::CConsoleLogger::Write(const char * message, int logType) {
}
```

Implementation in namespace

```
namespace symbion {
    void CConsoleLogger::Write(const char * message, int logType) {
    }
}
```

The above logging component is supposed to output messages to the console window. We will use the **printf** function from the Standard C Library to do this. Include **cstdio** to use the library from a C++ project. The **logType** argument determines the type of **message** to show.

Including the C Standard I/O Library

```
#include <cstdio>
```

Function implementation

```
void CConsoleLogger::Write(const char * message, int logType) {
    switch (logType) {
        case 0: printf("Information(\"%s\")\n", message); break;
        case 1: printf("Warning(\"%s\")\n", message); break;
        case 2: printf("Error(\"%s\")\n", message); break;
    }
}
```

Add in a **main.cpp** source file containing the following code to test the class. Create a local object named **log** and then call the **Write** method repeatedly to pass in different **message** and **logType**.

Testing the class

```
#include "symbion.h"

int main() {
    symbion::CConsoleLogger log;
    log.Write("This is the 1st message.", 0);
    log.Write("This is the 2nd message.", 1);
    log.Write("This is the 3rd message.", 2);
    return 0;
```

While it works, the above implementation is far from perfect and can be improved to provide more features. There are many different ways to implement a class and each method has their pros and cons and potential issues to be aware of. Following is a list of topics where we we use the recommended and best practices to write safe working code in C++.

1.3 Enumerations

Sometimes you may use code-numbers in your source code. In C, we normally define macros or constants to replace the code-numbers. This makes the code easier to read and understand especially for those who are not the original developers. For example you may need to differentiate the different types of messages that will be written to a log by using a code-number from 0 to 2. You can still use macros or constants or you can replace them with an enumeration in C++.

Using macros to replace code numbers in C

```
#define LOGTYPE_INFORMATION 0
#define LOGTYPE_WARNING 1
#define LOGTYPE_ERROR 2
```

Using an enumeration in C++

```
enum LogType { Information = 0, Warning = 1, Error = 2 };
```

An enumeration is a collection of named integer constants where each is called as an enumerator. By default the first enumerator has a value of 0 and for each subsequent enumerator the value will be incremented by 1. If these values are the ones that you wish to assign to each member anyway, the values can the be ommitted.

Using automatically assigned values

```
enum LogType {
    Information,
    Warning,
    Error
};
```

One reason for using constants instead of macros is that they can be typed. Default type for enumerations is always **int**. In the following example, the type of constants is set to **unsigned short int** instead. If you are using C++/11 and above enumerations can now be typed as well. If type size is important to you then you should not use the standard C++ types as they are not guaranteed to be the same between compilers. Use type definitions from the standard C++ library instead.

Constants can be strongly typed

```
const unsigned char LogTypeInformation = 0;
const unsigned char LogTypeWarning = 1;
const unsigned char LogTypeError = 2;
```

Enumeration members can be typed in C++/11

```
enum LogType : unsigned char {
    Information,
    Warning,
    Error
};
```

Predefined integer types from C++ library

```
#include <cstdint>

void main() {
    int8_t d1 = 127;                        // signed char
    int16_t d2 = 32767;                     // short
    int32_t d3 = 2147483647;                // long
    int64_t d4 = 9223372036854775807;       // long long
    uint8_t d5 = 255;                       // unsigned char
    uint16_t d6 = 65535;                    // unsigned short
    uint32_t d7 = 4294967295;               // unsigned long
    uint64_t d8 = 18446744073709551615;     // unsigned long long
}
```

Using C++ type definitions instead

```
enum LogType : uint8_t { Information, Warning, Error };
```

Another reason why we use constants is that they can be scoped within a **struct**. This helps resolve conflicts between macros, constants or enumeration members that have the same name. This feature is now available to enumerations since C++/11. You can use either **enum class** or **enum struct**.

Defining scoped constants

```
struct LogType {
    static const uint8_t Information = 0;
    static const uint8_t Warning = 1;
    static const uint8_t Error = 2;
};
```

Referring to non-scoped and scoped-constants

```
auto t = Information;           // non-scoped
auto t = LogType::Information;  // scoped
```

Defining a scoped enumeration: symbion.h

```
#include <cstdint>

namespace symbion {
    enum struct LogType : uint8_t { Information, Warning, Error  };
}
```

Update the declaration and the implementation of the **Write** method to use **LogType** as the type for **logType** argument. Using the **enum** makes the code more readable and less chance of passing in the wrong value.

Update **Write** function declaration to use LogType

```
void Write(const char * message, LogType logType);
```

Update function implementation to use LogType: consolelogger.cpp

```
void CConsoleLogger::Write(const char * message, LogType logType) {
    switch (logType) {
        case LogType::Information: printf("Information(\"%s\")\n", message); break;
        case LogType::Warning: printf("Warning(\"%s\")\n", message); break;
        case LogType::Error: printf("Error(\"%s\")\n", message); break;
    }
}
```

Updated main method: main.cpp

```
int main() {
    symbion::CConsoleLogger log;
    log.Write("This is the 1st message.", symbion::LogType::Information);
    log.Write("This is the 2nd message.", symbion::LogType::Warning);
    log.Write("This is the 3rd message.", symbion::LogType::Error);
    return 0;
}
```

To reduce the usage of the namespace, you can import members of the namespace or import the entire namespace with **using** and **using namespace**. You can then access **LogType** and **CConsoleLogger** directly without the **symbion** scope. Global functions and variables are considered in the global namespace, accessible by using unnamed scope like **::main** for the main function if there is a naming conflict between members in global namespace and the local namespace.

Import enum from namespace

```
using symbion::LogType;

int main() {
    symbion::CConsoleLogger log;
    log.Write("This is the 1st message.", LogType::Information);
    log.Write("This is the 2nd message.", LogType::Warning);
    log.Write("This is the 3rd message.", LogType::Error);
    return 0;
}
```

Import entire namespace

```
using namespace symbion;

int main() {
    CConsoleLogger log;
    log.Write("This is the 1st message.", LogType::Information);
    log.Write("This is the 2nd message.", LogType::Warning);
    log.Write("This is the 3rd message.", LogType::Error);
    return 0;
}
```

1.4 Member Variables

Currently our class only has a member function but no member variable. We will now add a member variable named **m_source** to assign name of the application or the module that is currently using the component to do logging. Member variables should always be **private** so you have control on the values that can be assigned to them. You can implement accessor functions to allow external access to private member variables.

Member variables and accessor functions: symbion.h

```
class CConsoleLogger {
private:
    const char * m_source;
public:
    const char * GetSource();
    void PutSource(const char * source);
};
```

Implementation for accessor functions: consolelogger.cpp

```
const char * CConsoleLogger::GetSource() { return m_source; }
void CConsoleLogger::PutSource(const char * source) { m_source = source; }
```

Update format strings to include logging source

```
printf("[%s] Information(\"%s\")\n", m_source, message);
printf("[%s] Warning(\"%s\")\n", m_source, message);
printf("[%s] Error(\"%s\")\n", m_source, message);
```

Setting the source: main.cpp

```
int main() {
    CConsoleLogger log;
    log.PutSource("symbion");
    log.Write("This is the 1st message.", LogType::Information);
    log.Write("This is the 2nd message.", LogType::Warning);
    log.Write("This is the 3rd message.", LogType::Error);
    return 0;
}
```

It is common naming convention to prefix member variable names with an underscore (_) or **m_** to potentially differentiate it from local variables and function arguments or you will need to use the **this** pointer to identify an object member since it returns the pointer to the current object. The following would be the code that we need to write if the member variable and function argument name is the same.

Using the this pointer to identify object members

```
void CConsoleLogger::PutSource(const char * source) {
    this->source = source;
}
```

1.5 Constructors & Destructor

Depending on the memory segment the object is allocated in, the initial value of the **m_source** member variable may be null or an invalid value. Static memory is always initialized to zero, but stack and dynamic memory are not. Since we cannot trust the initial value and **m_source** pointer should not be null anyway, we need to implement our own constructor to set the initial value since the default constructor generated by the compiler will not. We can also add another custom constructor to allow the caller to pass in the initial value as well.

Adding constructors: symbion.h

```
class CConsoleLogger {
    private:
        const char * m_source;
    public:
        CConsoleLogger();
        CConsoleLogger(const char * source);
                :
```

Implementation of constructors: consolelogger.cpp

```
CConsoleLogger::CConsoleLogger() : m_source("CConsoleLogger") { }
CConsoleLogger::CConsoleLogger(const char * source) : m_source(source) { }
```

Constructors are always extended from other constructors where calling it would also call parent and child constructors if they are available using the extend operator (**:**) to determine which constructor to call. For child objects, the default constructor which is the constructor with no arguments are called by default. Constructing an object would also construct all child objects.

We cannot trust any pointer passed to functions or constructors as they can be null or it can point to dynamic memory that may be released later. Instead of just copying a pointer, you need to copy the memory pointed to into memory allocated by the object so the object has full control over the memory. Update **PutSource** method to allocate memory to store a copy of the **source** string. Release previously allocated memory if **m_source** is not null. We need to update the constructors to initialize **m_source** to null using the **nullptr** constant and call **PutSource** to set **m_source**. Change type of **m_source** to non-constant pointer since **PutSource** needs to copy a string into the memory pointed to by the member variable.

Using dynamic memory to store a copy of a string

```
void CConsoleLogger::PutSource(const char * source) {
    if (m_source != nullptr) delete m_source;
    m_source = new char[strlen(source) + 1];
    strcpy(m_source, source);
}
```

Source must be a non-constant pointer : symbion.h

```
class CConsoleLogger {
    private:
        char * m_source;
               :
```

Update constructors to call PutSource

```
CConsoleLogger::CConsoleLogger() : m_source(nullptr) {
    PutSource("CConsoleLogger");
}

CConsoleLogger::CConsoleLogger(const char * source) : m_source(nullptr) {
    PutSource(source);
}
```

There are two ways to handle null pointers being passed to **PutSource**. Consider it as a fault and throw an exception to generate a runtime error. Any value or object can be thrown as an exception; an integer value to represent an error code or constant char pointer to an error message or a object to contain both the error code and message. You can implement your own custom exception class or use **exception** class from the Standard C++ Library. Note C++ Library uses the **std** namespace. If you do not wish to generate a runtime error use a default value. The **strcpy** function can cause buffer overrun as it does not check string length against buffer size so it is not safe to use. Use the safe version **strcpy_s** instead.

Include C++ exception header: consolelogger.cpp

```
#include <exception>
```

Throwing exception on null source argument

```
void CConsoleLogger::PutSource(const char * source) {
    if (source == nullptr) throw std::exception("Source cannot be null");
    if (m_source != nullptr) delete m_source;
    auto length = new char[strlen(source) + 1];
    strcpy(m_source, source);
}
```

Using a default value and safe version of strcpy instead

```
void CConsoleLogger::PutSource(const char * source) {
    if (source == nullptr) source = "CConsoleLogger";
    if (m_source != nullptr) delete m_source;
    auto length = strlen(source) + 1;
    m_source = new char[length];
    strcpy_s(m_source, length, source);
}
```

If you support a default value, change **PutSource** to pass in **nullptr** from the default constructor or remove the constructor to force the caller to provide a source.

Setting the default value from constructor

```
CConsoleLogger::CConsoleLogger() : m_source(nullptr) {
    PutSource(nullptr);
}
```

We have to make sure any dynamically allocated memory or resource will be released when an object is destroyed otherwise memory leaks will occur. This can be done by implementing a destructor function. The destructor is called just before an object is destroyed. We can check if **m_source** has memory allocation and release it. Normally it is customary to invalidate the pointer once memory is released by setting it to null but it does not matter in this case since **m_source** already ceased to exist once the object is destroyed.

Declaring the destructor: symbion.h

```
class CConsoleLogger {
                :
        ~CConsoleLogger();
    }
}
```

Implementing the destructor: consolelogger.cpp

```
CConsoleLogger::~CConsoleLogger() {
    if (m_source != nullptr) delete m_source;
}
```

Objects can be passed and returned by pointer, reference or value. Using pointers and references are always efficient because you are accessing the original object but this can be a problem if you are returning a stack or inline object that is destroyed at the function, thus the pointer and reference becomes invalid. When you pass and return an object by value, a copy is made and used instead. So even if the original no longer exists, you have the copy instead. To be able to make a copy, the compiler will always generate a copy constructor. The copy constructor is a constructor that will accept a reference to an object of the same type. The generated copy constructor just copies all the member variables including pointers. This means that the original and the copy is pointing to the same memory allocation or resource. If the original is destroyed the copy no longer works and vice versa, since the memory or resource used by both will be released. Each object requires their own memory allocation or resource handle so changes to one object does not affect the other object. To resolve this issue we need to have a custom implementation of the copy constructor.

Copy constructor declaration: symbion.h

```
class CConsoleLogger {
                :
        CConsoleLogger();
        CConsoleLogger(CConsoleLogger& obj);
        CConsoleLogger(const char * source);
                :
```

Copy constructor implementation: consolelogger.cpp

```
CConsoleLogger::CConsoleLogger(CConsoleLogger& obj) : m_source(nullptr) {
    PutSource(obj.m_source);
}
```

This ensures that the copy constructor does not simply copy **m_source** pointer from the original to the copy but makes a separate copy of the string instead. Since both original and the copy has their own string, destroying the original would not affect the copy and vice versa. Alternatively, you can stop the copy constructor from working by throwing an exception.

Unique objects cannot be copied

```
CConsoleLogger::CConsoleLogger(CConsoleLogger& obj) {
    throw std::exception("Object is unique and cannot be copied.");
}
```

The following shows the **explicit** syntax of calling constructors. If the constructor has one argument, it is an assignment constructor so you can use assignment operator to call it. If it has one or more arguments, you can use an initializer list instead.

Calling constructors: consolelogger.cpp

```
int main() {
    CConsoleLogger log_0;               // CConsoleLogger()
    CConsoleLogger log_1("symbion");    // CConsoleLogger(const char * source)
    CConsoleLogger log_2(log_1);        // CConsoleLogger(CConsoleLogger& obj)
    log_0.Write("Hello 0!", LogType::Information);
    log_1.Write("Hello 1!", LogType::Information);
    log_2.Write("Hello 2!", LogType::Information);
    return 0;
}
```

Using assignment operator to call constructor

```
CConsoleLogger log_3 = "symbion";   // CConsoleLogger(const char * source)
CConsoleLogger log_4 = log_3;       // CConsoleLogger(CConsoleLogger& obj)
log_3.Write("Hello 3!", LogType::Information);
log_4.Write("Hello 4!", LogType::Information);
```

Using initializer list to call constructor

```
CConsoleLogger log_5 { "symbion" }; // CConsoleLogger(const char * source)
log_5.Write("Hello 5!", LogType::Information);
```

To disable the use of assignment operator for construction as it may be confused with an assignment operation for classes that also have an assignment operator function, mark the constructor declaration with **explicit**.

Explicit constructor declaration

```
explicit CConsoleLogger(CConsoleLogger & obj);
```

1.6 Constant Functions & Mutable Variables

An object can be passed or returned as a constant. This means that the receiver can only access the variables in the object but cannot modify the object's variables. To be able to control this, only constant functions can be called on a constant object. Since none of **CConsoleLogger** functions are constant, they cannot be called. The following code cannot be compiled.

Using a const object

```
#include <cstdio>   // for puts function
int main() {
    const CConsoleLogger log; puts(log.GetSource());
    log.Write("Hello const log!", LogType::Information);
    return 0;
}
```

Mark functions that does not change object state with the **const** keyword in both the declaration and implementation.

Declaring constant member functions: symbion.h

```
const char * GetSource() const;
void PutSource(const char * source);
void Write(const char * message, LogType logType) const;
```

Implementing constant functions: consolelogger.cpp

```
const char * CConsoleLogger::GetSource() const {...}
void CConsoleLogger::Write(const char * message, LogType logType) const {...}
```

PutSource does change **m_source** variable so it cannot be a constant function. The compiler will not allow it. However there is still a way to change a member variable in a constant function by marking it as **mutable**. Mutable variables can be changed in a constant function. However this defeats the purpose of having a constant object.

Mutable variable

```
mutable char * m_source;
```

We can now use a constant version of the object in any function that does not require to change the object state including the copy constructor. Update the copy constructor to accept a **const CConsoleLogger** reference instead.

Constant object reference in copy constructor declaration: symbion.h

```
CConsoleLogger(const CConsoleLogger & obj);
```

Update copy constructor implementation

```
CConsoleLogger::CConsoleLogger(const CConsoleLogger& obj) : m_source(nullptr) {
    PutSource(nullptr);
}
```

1.7 Object Lifetime

Except for dynamic objects, the C++ compiler generates code to call the constructor and the destructor for objects. For global and static objects, the objects are created at the start of the program before **main** function is even called and destroyed at the end of the program. This guarantees that the object will always be there while your code is running. Global objects can be accessed from anywhere directly while static objects can only be accessed directly from within its scope. A static object within a function is only directly accessible from code in the function but it can still be passed to another function or return back to a caller function. These objects reside in static memory that is automatically initialized to 0.

Global and static objects

```
CConsoleLogger log_0;   // created before main, destroyed after main

int main() {
    static CConsoleLogger log_1; // created before main, destroyed after main
}
```

Local objects are created at the beginning of its scope and destroyed at the end of its scope. A local object within a function is created at the beginning of the function and destroyed at the end of the function. Always return a copy of a local object and never return a pointer or reference to the original object since it is already destroyed at the end of the function so the pointer or reference is no longer valid. Local objects reside in stack memory which is not initialized to 0.

Creating and returning a local object

```
const CConsoleLogger get_log() {
    CConsoleLogger log;     // created before get_log, destroyed after get_log
    putSource("symbion");
    return log;
}

int main() {
    auto log = get_log();   // copy of local object from get_log
    log.Write("Hello copy!", LogType::Information);
}
```

Inline objects are single-use objects that are created before a statement and released after the statement. The following shows a **CConsoleLogger** object created just for a **Write** function call and destroyed immediately after the function call completes.

Inline object

```
CConsoleLogger("symbion").Write("Hello inline!", LogType::Information);
```

Dynamic objects are created only when you use the **new** keyword and destroyed with the **delete** keyword. You have full control over its lifetime. Even though pointers are used for dynamic objects both references and pointers remain valid until the objects are destroyed with **delete**.

Using dynamic objects

```
CConsoleLogger *pLog = nullptr; // pointer can exist without object
pLog = new CconsoleLogger("symbion");   // pLog now points to a new object
pLog->Write("Hello dynamic!", LogType::Information);
delete pLog;    // object destroyed, pointer is no longer valid
pLog = nullptr; // an invalid pointer should always be nullified
```

C++ does not ensure pointers are always valid. Using a pointer that does not point to an existing object will always result in a runtime error. To allow code to easily check if a pointer is valid, initialize a pointer to **null** or a valid object and assign **null** once the object no longer exists. You can use a macro to simplify this operation. We can then easily check the pointer before it to access the object.

Macro to release a dynamic object and invalidate its pointer

```
#define SAFE_DELETE(ptr)    delete ptr; ptr = nullptr
```

Initialize a pointer to a new object

```
CConsoleLogger * pLog = new CConsoleLogger("symbion");
```

Checking pointer is valid

```
if (pLog) pLog->Write("Hello dynamic!", LogType::Information);
```

Dynamic objects reside in dynamic heap memory which is not initialized to 0. While dynamic memory will be released back to operating system once the program ends, memory leaks will occur during execution if you keep creating dynamic objects that are never destroyed. Sooner or later the program runs out of memory and it will then crash. A safer way to use dynamic objects is through smart pointers, a topic that will be covered later.

1.8 Default Arguments & Properties

Function arguments can be assigned default values during declaration. The compiler will then pass in the values if those arguments are ommitted during a function call. In the following declaration, we no longer need to have the default constructor since the custom constructor has a default value for the **source** argument. A default **LogType** is also assigned for the **Write** method.

Default value for source argument: symbion.h

```
CConsoleLogger(const char * source = "CConsoleLogger");
```

Default value for logType argument

```
void Write(const char * message, LogType logType = LogType::Information) const;
```

Using default arguments

```
CConsoleLogger().Write("Hello defaults!");
```

The problem with default arguments is that the values can be changed by anyone that has the class declaration header file. Alternatively you can implement overloading and have two **Write** functions instead. Overloading allows you to have multiple functions with the same name but a different number or type of arguments. The new **Write** will call back the original function but passes in the default value. Since the default value is hardcoded, it cannot be changed externally.

Overloading Write method

```
void Write(const char * message, LogType logType);
void Write(const char * message);
```

Overloaded Write calls original Write with default value

```
void CConsoleLogger:Write(const char * message) {
    Write(message, LogType::Information);
}
```

Finally, you can still have both but not by using overloading but implementing helper functions that will log specific type of messages. Add the following declaration into the **CConsoleLogger** class and implement the functions as shown. While **LogType** for **Write** can be changed, the following functions will each used a fixed value. Using the helper functions does not improve performance but can reduce code size. It is always the role of an object-oriented developer to encapsulate complexity within a class and protect the internals but also to make them as easy to use as possible.

Declaration of helper functions: symbion.h

```
void Message(const char * message) const;
void Warning(const char * message) const;
void Failure(const char * message) const;
```

Implementation of helper function: consolelogger.cpp

```
void CConsoleLogger::Message(const char * message) const {
    Write(message, LogType::Information);
}

void CConsoleLogger::Warning(const char * message) const {
    Write(message, LogType::Warning);
}

void CConsoleLogger::Failure(const char * message) const {
    Write(message, LogType::Error);
}
```

Using the helper functions

```
CConsoleLogger log;
log.Message("This is a message!");
log.Warning("This is a warning!");
log.Failure("This is an error!");
```

Properties are not a standard C++ feature. This is only applicable to Visual C++. This feature makes accessor functions look like member variables so you can write much simpler code for components that has a lot of accessor functions. Use the **_declspec** keyword to declare a **property**. For each property, specify the accessor functions for **get** and **put** followed by the property type and name. Properties can be both used in our outside the class, not only making the class easier to use but also simpler during implementation as well.

Declaring a property for GetSource and PutSource: symbion.h

```
_declspec(property(get=GetSource,put=PutSource)) const char * Source;
```

Setting and accessing the property

```
log.Source = "symbion"; puts(log.Source);
```

Property can be used internally as well

```
CConsoleLogger::CConsoleLogger(const char * source) { Source = source; }
```

1.9 Inline Functions

You can decide to put simple function implementations into the class declaration. This is called as inlining. The function will not be generated in the release version. Instead a call to an inline function is replaced by the function code. The benefit of using inline functions is that it does not trash the CPU cache thus your code will run considerably faster, making full use of the amount of cache memory available. The bad side is you completely fail encapsulation and programs will become extremely buggy.

Implementing inline functions

```
class CConsoleLogger {
        :
public:
    CConsoleLogger(const CConsoleLogger & obj): m_source(nullptr) {
        PutSource(obj.m_source); }
    CConsoleLogger(const char * source = "CConsoleLogger"): m_source(nullptr) {
        PutSource(source); }
    const char * GetSource() const { return m_source; }
    void PutSource(const char * source);
    void Write(const char * message,
        LogType logType = LogType::Information) const;
    void Message(const char * message) const {
        Write(message, LogType::Information); }
    void Warning(const char * message) const {
        Write(message, LogType::Warning); }
    void Failure(const char * message) const {
        Write(message, LogType::Error); }
    ~CConsoleLogger();
};
```

2 Generalization & Inheritance

2.1 Generalization

You may need to develop a set of compatible components that can be used to replace each other. This means not only will they share the same interface but may also share code. To avoid redundancy, you can implement generalization where general features and content for all components is placed into a base class. To support polymorphism where one component can be substituted by another component, you can implement **virtual** or **abstract** functions for code that is different between them. In the previous chapter you have implemented a **CConsoleLogger** class. However you also want to implement additional logging components to record to a file or write to the Windows event log. Since they will have a lot of code that is the same, you do not wish to re-invent the wheel for all classes. You will need to generalize features in into a separate **CBaseLogger** class. Copy **CConsoleLogger** class declaration to **CBaseLogger**. Add a **baselogger.cpp** source file and copy all the code from **consolelogger.cpp** source file and rename to **CBaseLogger**. Currently these classes are completely the same which means that **CConsoleLogger** is a redundant version of **CBaseLogger**. Replace the declaration of **CConsoleLogger** with the following so that it can inherit existing functionality from the base class instead.

Updated CConsoleLogger class declaration: symbion.h

```
class CConsoleLogger : public CBaseLogger {
public:
    CConsoleLogger();
    CConsoleLogger(const char * source);
};
```

Use the extend operator (**:**) to inherit all the features of **CBaseLogger**. A **public** type of inheritance means that no access level is changed for the inherited members. For a **protected** inheritance, all public members in **CBaseLogger** becomes protected when inherited into **CConsoleLogger**. For **private** inheritance, all **public** and **protected** members becomes **private**. Constructors are never inherited but always extended. Each class should have its own constructors extended from base constructors. Since the base class has no default constructor, you have to specify exactly the constructor to extend from.

Updated CConsoleLogger implementation: consolelogger.cpp

```
namespace symbion {
    CconsoleLogger::CConsoleLogger():CBaseLogger("CConsoleLogger") { }
    CConsoleLogger::CConsoleLogger(const char * source):CBaseLogger(source) { }
}
```

2.2 Virtual & Abstract Functions

It is actually pointless for **CConsoleLogger** to exist since **CBaseLogger** class is fully functional. Sometimes we implement a base class simply to allow derived classes to inherit data and code, and not be used directly. **CBaseLogger** should not determine the log target. Copy the declaration and implementation of **Write** method from the base class to **CConsoleLogger**.

CConsoleLogger declaration: symbion.h

```
class CConsoleLogger : public CBaseLogger {
public:
     CConsoleLogger();
     CConsoleLogger(const char * source);
     void Write(const char * message, LogType logType = LogType::Information) const;
};
```

Since **m_source** is private to **CBaseLogger**, so either use **GetSource** or the **Source** property to access it from the derived class. You can change **m_source** to **protected** to allow derived classes to access but this fails encapsulation because a derived class may accidentally set **m_source** to **null**.

CConsoleLogger implementation: consolelogger.cpp

```
void CConsoleLogger::Write(const char * message, LogType logType) const {
     switch (logType) {
          case LogType::Information:
               printf("[%s] Information(\"%s\")\n", Source, message); break;
          case LogType::Warning: printf("[%s] Warning(\"%s\")\n", Source, message); break;
          case LogType::Error: printf("[%s] Error(\"%s\")\n", Source, message); break;
     }
}
```

You can now remove the code in the **CBaseLogger::Write** function. Derived classes will determine the logging target not the base class.

A blank implementation of Write: baseLogger.cpp

```
void CBaseLogger::Write(const char * message, LogType logType) const {
//  to be implemented by derived classes
}
```

To support polymorphism where components can be replaceable, it will be required to mark functions that could be implemented, extended or replaced by derived classes with the **virtual** keyword. The derived class can optional used the **override** keyword to allow the C++ compiler to verify that it is implemented a virtual method.

Marking overridable functions in CBaseLogger: symbion.h

```
virtual const char * GetSource() const;
virtual void PutSource(const char * source);
virtual void Write(const char * message,
     LogType logType = LogType::Information) const;
```

Declaring an overriding function in CConsoleLogger

```
class CConsoleLogger : public CBaseLogger {
public:
    CConsoleLogger();
    CConsoleLogger(const char * source);
    void Write(const char * message,
        LogType logType = LogType::Information) const override;
};
```

There is still a problem with **CBaseLogger**. It is not suppose to be fully functional yet you can still create an instance of the class and use the object but **Write** doesn't do anything which other programmers may not know this fact. It would be better to stop the class being used for anything else except inheritance. Mark the **Write** method as **abstract** instead and remove the **Write** implementation code. This means that the **CBaseLogger::Write** method is not implemented at all. The C++ compiler will not allow an object to be instantiated from an incomplete implementation. If **abstract** is not supported by your C++ compiler, use the pure virtual syntax instead where you assign 0 to the virtual function.

Marking Write as abstract in CBaseLogger: symbion.h

```
virtual const char * GetSource() const;
virtual void PutSource(const char * source);
virtual void Write(const char * message,
    LogType logType = LogType::Information) const abstract;
```

Pure virtual function syntax for abstract

```
virtual void Write(const char * message,
    LogType logType = LogType::Information) const = 0;
```

For virtual functions, you can choose to inherit or override. For abstract functions you cannot inherit, so you must override. For other functions, you can only inherit but not override. Even though these are not important just for inheritance, this must be done in order to achieve polymorphism.

2.3 Helper Functions

A base class should not only provide inheritance but provide support to make it easier to implement the derived classes as well. One feature that is needed by loggers is to convert **LogType** into a string. We can add an additional function to do this that will also be inherited as well. If the function doesn't require access to any object members it can be marked **static** so it can be called directly without having to create an object at all. Basically static member variables and static member functions are like global variables and functions but scoped to the class so encapsulation rules can still apply. For example, only member functions can access a **private static** variable even if it is allocated in global static memory. The following are multiple implementations of the same functionality. You can either throw an exception or return a default string if the **LogType** value is not valid.

Declaration of CBaseLogger static member function: symbion.h

```
static const char * GetLogTypeText(LogType logType);
```

Using switch block to return LogType string: baselogger.cpp

```
const char * CBaseLogger::GetLogTypeText(LogType logType) {
    switch (logType) {
        case LogType::Information: return "Information"; break;
        case LogType::Warning: return "Warning"; break;
        case LogType::Error: return "Error"; break;
        default: return "Message";
    }
}
```

Using array and LogType as index

```
const char * CBaseLogger::GetLogTypeText(LogType logType) {
    static const char * strings[] = {
        "Information", "Warning", "Error", "Message"  };
    int type = (int)logType;
    if (type < 0 || type > 3) type = 3;
    return strings[type];
}
```

Simplify CConsoleLogger Write method: consolelogger.cpp

```
void CConsoleLogger::Write(const char * message, LogType logType) const {
    printf("[%s] %s(\"%s\")\n", GetLogTypeText(logType), Source, message);
}
```

3 Polymorphism

3.1 Compatibility

You can achieve compatibility between classes through inheritance. When you extend from a base class, the base class becomes a level of compability. All derived classes is compatible to the base class. To demonstrate this, we need to implement a few more **CBaseLogger** derived classes. However to achieve polymorphism, we need to follow a few more rules during this implementation.

CDebugLogger class declaration: symbion.h

```
class CDebugLogger : public CBaseLogger {
public:
    CDebugLogger();
    CDebugLogger(const char * source);
    void Write(const char * message,
        LogType logType = LogType::Information) const override;
};
```

The following is the implementation code for the component. The **LogType** is used to select a string that combined with the source and message into a single C string that is then send to Visual Studio debug output window by using the Windows API function **OutputDebugStringA**. Since Visual Studio 2012, it is considered as an error rather than a warning to use unsafe CRT (C-Runtime Library) functions so **sprintf_s** is used instead of **sprintf**. If for some reason you have to use the unsafe functions, define a **_CRT_SECURE_NO_WARNINGS** macro.

CDebugLogger class implementation: debuglogger.cpp

```
#include "symbion.h"
#include <windows.h>
#include <cstdio>

namespace symbion {
    CDebugLogger::CDebugLogger() : CBaseLogger("CDebugLogger") {}
    CDebugLogger::CDebugLogger(const char * source): CBaseLogger(source) { }
    void CDebugLogger::Write(const char * message, LogType logType) const {
        char text[1024];
        sprintf_s(text, "[%s]%s(\"%s\")\n",
            Source, GetLogTypeText(logType), message);
        OutputDebugStringA(text);
    }
}
```

CFileLogger class declaration: symbion.h

```
class CFileLogger : public CBaseLogger {
private:
    const char * m_filename;
public:
    CFileLogger();
    CFileLogger(const char * source);
    CFileLogger(const CFileLogger & obj);
    void PutSource(const char * source) override;
    void Write(const std::string& message,
        LogType logType = LogType::Information) const override;
    ~CFileLogger();
private:
    void UpdateFilename();
};
```

CFileLogger class implementation: filelogger.cpp

```
#include "symbion.h"
#include <fstream>
#include <ctime>

namespace symbion {
    CFileLogger::CFileLogger()
        : m_filename(nullptr), CBaseLogger("CFileLogger") {
        UpdateFilename();
    }
    CFileLogger::CFileLogger(const char * source)
        : m_filename(nullptr), CBaseLogger(source) {
        UpdateFilename();
    }
    CFileLogger::CFileLogger(const CFileLogger & obj)
        : m_filename(nullptr), CBaseLogger(obj) {
        UpdateFilename();
    }
    void CFileLogger::PutSource(const char * source) {
        CBaseLogger::Source = source;   // CBaseLogger::PutSource(source);
        UpdateFilename();
    }
    void CFileLogger::UpdateFilename() {
        if (m_filename) delete m_filename;
        auto length = strlen(Source) + 5;
        m_filename = new char[length];
        strcpy_s(m_filename, length, Source);
        strcat_s(m_filename, length, ".log");
    }
    CFileLogger::~CFileLogger() {
        if (m_filename) delete m_filename;
    }
}
```

The Write method for CFileLogger

```
void CFileLogger::Write(const char * message, LogType logType) const {
    tm datetime;
    char text[2048];
    auto now = time(nullptr);
    localtime_s(&datetime, &now);
    sprintf_s(text, "[%04d-%02d-%02d %02d:%02d:%02d]%s(\"%s\")",
    //  tm_year is relative to 1900, tm_mon is 0 for January
        datetime.tm_year + 1900, datetime.tm_mon + 1, datetime.tm_mday,
        datetime.tm_hour, datetime.tm_min, datetime.tm_sec,
        GetLogTypeText(logType), message);
    std::fstream file(m_filename, std::ios::app);
    file << text << std::endl;
    file.close();
}
```

CFileLogger requires an extra member variable **m_filename** that is **m_source** plus a file extension (**.log**). This is done by **UpdateFilename** method and called when the overriding **PutSource** function is called. Since all the **CBaseLogger** constructors only initializes **m_source**, **CFileLogger** constructors have to initialize **m_filename**. There is a polymorphic issue that base constructors can only call their own virtual functions and not overriding functions in the derived classes. So **CBaseLogger** constructors call **CBaseLogger::PutSource** and not **CFileLogger::PutSource** which does not update the filename. That is why **UpdateFilename** is called on all **CFileLogger** constructors. The technical explaination is virtual functions are called through a virtual table. Each class has its own virtual table assigned to the object by the constructor. Since the **CBaseLogger** constructor runs before **CFileLogger** constructor, the object only has access to the base class virtual table in the base constructor. After total construction, the object will have the correct virtual table and will then call the correct **PutSource** function in **CFileLogger**. As long as the function is virtual, the correct function will be called including the destructor. However, we did not declare a virtual destructor in **CBaseLogger**. Any class that has at least one virtual function is called a polymorphic type where the destructor must be virtual or you may call the wrong destructor. Mark **CBaseLogger** destructor as virtual. If the object is a **CFileLogger**, then destructor for **CFileLogger** is called to release **m_filename**. Since destructors are extended as well, the **CBaseLogger** destructor is also called to release **m_source**.

Declare virtual destructor: symbion.h

```
public class CBaseLogger {
        :
    virtual ~CBaseLogger();
}
```

Now that we have implemented more classes, we can now test compatibility. Add the following method that can use any instance from **CBaseLogger** derived class. They are all compatible based on inheritance. Pointers and references must always be used to achieve polymorphism because they allow you to access the original object. If you pass by value, you are making a copy and the copy will always be **CBaseLogger** but this cannot work at all since this class is **abstract**.

Testing compatibility: main.cpp

```
void LogTo(const CBaseLogger & log) {
    log.Write("Hello!");
    log.Write("Goodbye!");
}

int main() {
    LogTo(CConsoleLogger());
    LogTo(CDebugLogger());
    LogTo(CFileLogger());
    return 0;
}
```

3.2 Polymorphism

Polymorphism is when you replace one type of object with a different type of object. It will still work as long as the replacement object type is compatible with the original object type. Compatibility is automatically achieved between the derived classes and all the base classes regardless of how many levels of inheritance. Code that has been written for a base class will support all the derived classes.

Compatibility through inheritance

```
class A {virtual void f(){}}    // a polymorphic type
class B:public A { }            // compatible to A
class C:public B { }            // compatible to B and A
class D:public C { }            // compatible to C, B and A

void foo1(A& obj) {     // function that can accept A, B, C and D objects
    std::cout << "Using a " << typeid(obj).name()
        << " object." << std::endl;
```

Rather than forcing the application to use a specific logging component, we can make the choice of the component to use as an external configuration. You can use a file, the Windows registry or environment variables to store configuration information. At runtime, you can fetch the information to create the correct type of component. Since all components are compatible to **CBaseLogger**, you can use this as the interface to access the component without having to know exactly what type it is. Since **Write** function is virtual, the correct function will be called. And if it is a dynamic object, we do not have to worry about destroying the object as the destructor is marked virtual, guaranteeing the right destructor is called. However rather than writing code to create the object everytime you can encapsulate it into a class following the factory method design pattern.

Class to help create objects: symbion.h

```
class CLoggerFactory {
    public:
        CBaseLogger *CreateInstance();
};
```

Implementation code: LoggerFactory.cpp

```
#include "symbion.h"

CBaseLogger *CLoggerFactory::CreateInstance() {
    char buffer[256];
    if (GetEnvironmentVariableA("LOGGER", buffer, sizeof(buffer)) > 0) {
        _strupr_s(buffer);
        if (strcmp(buffer, "CONSOLE") == 0) return new CConsoleLogger();
        if (strcmp(buffer, "FILE" ) == 0) return new CFileLogger();
        if (strcmp(buffer, "DEBUG") == 0) return new CDebugLogger();
    }
    return new CConsoleLogger(); // default
}
```

Using factory to instantiate correct logger component

```
void main() {
    CLoggerFactory factory;
    CBaseLogger *pLog = factory.CreateInstance();
    pLog->Message("Hello!");
    pLog->Message("Goodbye!");
    delete pLog;
}
```

3.3 Singleton

Every time you call **CreateInstance** you will get a new object. Sometimes we many only need a single instance throughout the entire application. We will now update our factory to support a singleton. A static member variable will be used to keep track of the singleton. Since static variables are allocated in static memory, is it guaranteed to be initialized to a null pointer at the beginning. The **GetInstance** function will create the object if the singleton has not been created yet but will return the same object if the method is called again.

Declaration to support singleton: symbion.h

```
class CLoggerFactory {
private:
    static CBaseLogger *m_pInstance;
public:
    static CBaseLogger *CreateInstance();
    static CBaseLogger *GetInstance();
}
```

Implementing singleton pattern: LoggerFactory.cpp

```
static CBaseLogger CloggerFactory::*m_pInstance;

CBaseLogger *CLoggerFactory::GetInstance() {
    return m_pInstance != nullptr ? m_pInstance :
        (m_pInstance = CreateInstance());
}
```

Using the singleton

```
CBaseLogger *pLog = CLoggerFactory::GetInstance();
pLog->Message("Hello!");
pLog->Message("Goodbye!");
```

Single-use of the singleton

```
CLoggerFactory::GetInstance()->Write("Sayonara!");
```

Note that because the singleton is a dynamic object, it will never be destroyed. This is not a huge problem since we will only forever have one instance of the object so even if there is a memory leak, it is fixed, consistent and small. Eventually when a program ends, all memory will go back to the operating system anyway. However, if it is very important for you to have to destroy the singleton at the end of the program usually when you need to the destructor to be called, you can use a smart pointer to manage the object.

3.4 Smart Pointer

A smart pointer is an object helps you to manage a dynamic object. The smart pointer itself should never be dynamic. This ensures that the smart pointer is destroyed when it goes out of scope. For example a global smart pointer is destroyed at the end of the program while a local smart pointer is destroyed at the end of function. Destructor of the smart pointer ensures that the dynamic object will be destroyed. You can choose to implement a unique or shared pointer. A unique pointer cannot be copied and must always be passed by reference or by pointer. You can do this by throwing an exception in the copy constructor.

Declaration of the smart pointer: symbion.h

```
class CLoggerPtr {
private:
    CBaseLogger *m_ptr;
public:
    CLoggerPtr();
    CLoggerPtr(CBaseLogger *ptr);
    CloggerPtr(const CLoggerPtr& obj);
    ~CLoggerPtr();
    void PutPtr(CBaseLogger *ptr);
    CBaseLogger *GetPtr() const;
};
```

Smart pointer class implementation: LoggerPtr.cpp

```
CLoggerPtr::CLoggerPtr() : m_ptr(nullptr) { }
CLoggerPtr::CLoggerPtr(CBaseLogger *ptr) : CLoggerPtr() { PutPtr(ptr); }
CloggerPtr::CLoggerPtr(const CLoggerPtr& obj) {
    throw "Attempt to copy unique pointer";
}
CLoggerPtr::~CLoggerPtr() { if (m_ptr != nullptr) delete m_ptr; }
void CLoggerPtr::PutPtr(CBaseLogger *ptr) {
    if (ptr != m_ptr && m_ptr != nullptr) delete m_ptr;
    m_ptr = ptr;
}
CBaseLogger *CLoggerPtr::GetPtr() const { return m_ptr; }
```

It is not enough for a smart pointer to manage the dynamic object but it should make itself behave as though it is actually the dynamic object. It can be done through the overloading of operators that you normally use with pointers such as assignment, the pointer member access operator (->). For automatic conversion between pointers and smart pointers, you can create an assignment constructor, and overload pointer type operator.

Operators to overload

```
operator CBaseLogger *() const; // type conversion operator
CLoggerPtr& operator =(CBaseLogger *ptr);  // assignment operator
CBaseLogger* operator ->() const;   // pointer member access operator
```

Operator implementation: LoggerPtr.cpp

```
CLoggerPtr::operator CBaseLogger *() const { return GetPtr(); }
CLoggerPtr& CLoggerPtr::operator =(CBaseLogger *ptr) {
    PutPtr(ptr); return *this;
}
CBaseLogger *CLoggerPtr::operator ->() const { return GetPtr(); }
```

Testing the smart pointer

```
void main() {
    CLoggerPtr log = CLoggerFactory::CreateInstance();
    log->Write("Hello!");
    log->Write("Goodbye!");
}
```

You can now easily update the **CLoggerFactory** class to use smart pointer to manage the singleton. This guarantees that the singleton will be destroyed when the program ends if the singleton has been created.

Using smart pointer to manage the singleton

```
class CLoggerFactory {
private:
    static CLoggerPtr m_pInstance;
public:
    static CBaseLogger *CreateInstance();
    static CBaseLogger *GetInstance();
};
```

Smart pointer is not dynamic and uses static memory: LoggerFactory.cpp

```
CLoggerPtr CloggerFactory::m_pInstance;
```

Rather than implementing your own smart pointer types, the standard C++ library do come with templates that helps you to generate smart pointer classes. Both unique and shared pointer templates are available.

Using smart pointer templates from standard C++ library

```
#include <memory>

void main() {
    std::unique_ptr<CBaseLogger> log(CLoggerFactory::CreateInstance());
    log->Write("Hello!");
    log->Write("Goodbye!");
}
```

Shared pointers allow multiple smart pointers to share the same dynamic object that is not allowed for unique pointers. The shared pointer has internal reference counting to keep track of how many shared pointers per dynamic object. The dynamic object is only destroyed when the last shared pointer is destroyed. The dynamic object will still be there as long as there is still one shared pointer referencing it. This guarantee that the object will never be destroyed until no code is using it anymore.

Using shared pointers

```
void main() {
    std::shared_ptr<CBaseLogger> log1(CLoggerFactory::CreateInstance());
    std::shared_ptr<CBaseLogger> log2(log1);
    log1->Write("Hello!");   // different smart pointer - same dynamic object
    log2->Write("Goodbye!"); // different smart pointer - same dynamic object
}
```