

# 3

## Class Design and Inheritance

### Introduction

Chapter 2 described the properties of classes and objects, introducing C++ support for the object-oriented programming (OOP) paradigms of abstraction, encapsulation and implementation hiding. This chapter introduces C++ provision of inheritance and both dynamic and static polymorphism.

Inheritance is a large and complex subject area, where innocuous and seemingly obvious facts often turn out to have far-reaching ramifications. Operator overloading, for example, has arguably led to more programming errors and inefficient code than the problems it solves. This is no reflection on the language itself but, more often than not, is caused by enthusiasm overtaking attention to detail.

This chapter does not give a complete specification of OOP inheritance but delivers the salient facts that an ASD candidate is expected to know and understand.

### 3.1 Class Relationships

C++ classes serve two primary purposes: to support user-defined types and to provide reuse through inheritance and composition relationships. This section describes the various OOP relationships.

#### Critical Information

##### *Inheritance, composition and aggregation*

There are fundamentally two kinds of relationship between classes in the C++ support of OOP: “is-a” or *inheritance*, and “has-a” or *composition*. Composition can be further divided into strong and weak or *aggregation*.

Relation	OO Term
S60 is a Symbian OS UI platform	Inheritance (is-a)
UIQ is a kind of UI platform	Inheritance (a-kind-of)
W950i is a mobile phone and is a music player	Multiple inheritance (is-a)
An N91 is part of the Nseries range	Aggregation (weak has-a)
A Smartphone device has an operating system	Composition (strong has-a)

(\*N91 and W950i are both Smartphones, Nseries is a product line)

The table above contains some examples of these relationships: S60 is specifically a Symbian UI platform, and along with UIQ is more broadly referred to as a kind of UI – as would be, say, a PC or a car dashboard. The Sony Ericsson W950i is both phone and a music player. All three of these relationships are **inheritance** relationships as they either extend or specialize (and in the last case also combine).

If the N91 was removed from the Nseries product line, both the product line and N91 would still exist, the N91 would continue to be a phone (or multimedia computer) and the Nseries would still be a collection of multimedia devices. This is weak composition or **aggregation**.

Whereas a Smartphone device has to have an operating system to be a Smartphone, the operating system is an integral part of the Smartphone but the Smartphone is not an operating system, in other words there is no inheritance. This is strong **composition**: the operating system requires device hardware to run on and the device requires an operating system.

In the examples above, it is possible to see additional relationships, for example a UI is part of Symbian OS and UIQ is a Symbian OS UI platform. Good OO analysis and design require the careful selection and use of these relationships.

### ***Reuse advantages and disadvantages***

Aggregation, composition and inheritance are primarily used to remove redundancies, in particular the repetition of the same code (often implemented differently) in different classes or programs. In the analysis of a problem, if a group of concepts share common properties and behavior they are good candidates for abstraction, in this case, the separation of the common functionality. The amount of commonality dictates the degree

of code *reuse* and the purpose of the relationships, that is, whether a class should inherit or contain the common entity.

The advantages of C++ class reuse:

- Reduced development time
- Fewer defects – this is a self-stoking cycle as the more a class is used, the more likely errors are to be uncovered and fixed
- Enforced interface consistency – a known interface at design time provides positive constraints on the design
- Known behavior
- Rapid prototyping with live code
- Faster implementation modifications – only a single class needs to be modified.

There are also disadvantages:

- Increased coupling between classes – modify a common class interface and there is a cascade effect or compatibility break (see Chapter 16)
- Design tradeoff problems – a large class can introduce unnecessary code bloat to client classes requiring only a specific aspect of that class
- Complexity – too many small base classes can introduce complex or multiple inheritance relationships
- Difficulties in understanding inheritance relationships – dynamic behavior is not always clear.

## Exceptions and Notes

In OOD, types and classes are seen as different concepts: a type has its roots in set theory and formal methods, whereas classes are seen as structure concepts. Some languages, such as Smalltalk, maintain this separation; in C++, types and classes are handled the same. This can lead to some confusion when talking about relationships in a more formal context.

## Exam Essentials

- Understand the key benefits and purpose of inheritance
- Specify the differences between composition, aggregation and inheritance
- Cite the object-oriented relationship for inheritance

## References

- [Ambler 2001 Chapters 5–7]
- [Gamma *et al.* 1995 Chapter 4]
- [Meyers 2005 Chapter 6]
- [Morris 2006 Chapter 5]
- [Stroustrup 2000 Section 24.3]
- [Sutter and Alexandrescu 2004 Item 34]

## 3.2 Inheritance

The terminology used here is *base* and *derived* class where the base class is the parent or superclass and the derived class is the child or sub-class. *Class user* refers to code, or another class, that has no inheritance or friend relationship with the class and has only public access.

### Critical Information

#### *Public inheritance*

Public inheritance is a relationship between derived and base classes where the derived class “is-a” specialization of the more generic base class.

The following example typifies a public base-derived inheritance relationship, showing the use of constructors to ensure the correct initialization of values from a derived class. It introduces the `protected` keyword as an inheritance access scope and the `virtual` member function.

```
// A read-only base class
class Base
{
public:
    Base():iVal(0){}           // Default constructor
    Base (int n):iVal(n){};   // Overload for value

    int Get() const {return iVal;} // Access method (read only)

protected:
    virtual void Set(int n){iVal=n;} // write method
    void Reset() {Set(0);} // will call the correct Set()

private:
    int iVal; // private data member
};

// A read-write class:
// Derived overrides the protected Set() method in Base extending the
// functionality of the base class by allowing write objects.
class Derived : public Base // Inherits the public and protected
                          // interface
```

```

{
public:
    // Default constructor
    Derived(bool write = false): Base(42), AllowWrite(write)
    {
    }

    // Override
    void Set(int n) // Overrides the protected Set() method of Base
    {
        if (AllowWrite)
            Base::Set(n);
    }

private:
    const bool AllowWrite; // const state i.e. only one state defined at
                           // initialization
};

// In use
Derived foo(true); // allow write
foo.Set(24); // use Derived::Set() to set a value
int n = foo.Get(); // use Base::Get() to get the value

```

The derived class publicly inherits the base-class `public` interface but has no special access permission to the `private` area of the base class. To support a finer degree of access control for inheritance, C++ provides the `protected` keyword. This allows publicly inheriting derived classes access to member functions and data declared inside the `protected` sections of the base class, while still hiding them from class users. Use of the `protected` keyword tells the developer this is where the base class is expected to be extended (see Section 3.3 for the use of substitution and virtual member functions).

Base-class default constructors are implicitly invoked in the order of inheritance, before their derived-class constructors, that is before the body of the derived constructor executes. Base-class destructors are called in the reverse order after the body of the derived-class destructor has finished executing. (See Section 3.3 for the use of virtual destructors.)

### Virtual methods

Virtual methods are covered in more detail in Section 3.3, but for better understanding of the role of inheritance, here is an overview.

A virtual method is a method declared in a base class which, provided the prototypes are identical, allows any derived class to redefine it or, more formally, *override* it. The `virtual` keyword tells the compiler to ensure the correct version of the function is invoked for the relevant derived class.

The purpose of *virtual overriding* is to allow member functions to be *dynamically bound* to their defining object class. In the previous

example, the `Reset()` function in the `Base` class will call the redefined `Derived::Set()` and not `Base::Set()` because `foo` is created as a `Derived` class. A non-virtual override of `Set()` would result in `Reset()` calling `Base::Set(0)` regardless of the derived implementation, as it would be *statically bound* to `Base`.

Virtual overrides allow base-class run-time behavior to be *modified* by a derived class without the base-class implementation *seeing* the derived class. In fact the derived class can be added at a later date, such as a plug-in to a framework. This is sometimes referred to, somewhat hyperbolically, as “code written today to call code written tomorrow”.

Note the use of the scope resolution operator `::` where the derived `Set()` function calls `Base::Set(n)`. Without the `Base::` qualifier, `Derived::Set()` would recursively call itself.

### ***What is not inherited (who needs friends?)***

Certain methods and properties are not inherited in a base-derived relationship:

- The base class constructors and destructors
- The assignment operator, `=`
- Friends – a friend of a base class has no special access permissions to a derived class (see information on friends in Section 2.3).

### ***Private inheritance***

Public inheritance is intuitive. The derived class has access to the `public` and `protected` members of the base class; the class user automatically has access to the base public interface while the `protected` members stay hidden. Public inheritance is the C++ representation of an “is-a” relationship.

The purpose of private and protected inheritance may be a little less obvious.

```
class Derived : public Base;    // public inheritance - as above
class Derived : private Base;  // private inheritance
class Derived : protected Base; // protected inheritance
```

In a private inheritance relationship, everything in the `Base` class is imported into the private area of the `Derived` class and is hidden from the users of the `Derived` class. The `Base` class in effect becomes a private class in the `Derived` class; thus the relationship is no longer “is-a” (inheritance) but “has-a” (composition).

Private inheritance is an implementation feature; there is no design-time or *conceptual* relationship between a privately inherited base class

and a derived class. It is more accurate to represent private inheritance as the derived class “is-implemented-in-terms-of” the base class.

```
// Base class
class Radio // GSM or CDMA Radio, not FM or bluetooth
{
public:
    // many interface functions ...
    // ...
    virtual TState OnModemRequest() // Virtual Method

    void ModemReq()
    {
        TState state = OnModemRequest(); // Override call
        switch (state) // ... etc
        }
};

// Derived class
class CellPhone : private Radio
{
public:
    void MakeCall();

    // ... many interface functions ...
private:
    TState OnModemRequest(); // note the virtual keyword is not required
};
```

In the example, the Radio class is a generic utility class, its methods are public and there is a virtual event method `OnModemRequest()` which requires some action on the part of a derived class.

A cell phone “has-a” radio, so at design time it makes perfect sense to use composition – the CellPhone class would have a Radio member object. But the CellPhone class requires notification of a ModemReq to tell the Radio its state, delivered via the virtual `OnModemRequest()` method; to override a method, the class has to be inherited.

If the CellPhone class publicly inherits Radio, its interface would be made available to users of the CellPhone class. This would make very little sense, as the CellPhone is not a Radio type (not “is-a”) and it is no longer a “has-a” relationship, either. The CellPhone class “is-implemented-in-terms-of” the Radio class.

The advantage of privately inheriting is that it hides the Radio interface while maintaining the CellPhone logical interface; in other words, it maintains encapsulation while providing the implementation functionality that composition cannot support.

### ***Protected inheritance***

Protected inheritance, like private inheritance, is a property of implementation and hides the public interface of the base class. Protected

inheritance applies `protected` access in the derived class to the `public` and `protected` members of the base class. This allows any classes (*grandchildren*) that further derive from the derived class (*child*) to have access to the base (*parent*) methods, unlike private inheritance, which stops any further access to the base class in the inheritance hierarchy.

There is rarely a valid reason for using protected inheritance.

Note: neither private nor protected inheritance prevents virtual functions from being overridden.

### ***Inheritance summary***

- Private inheritance prevents both public and any further inherited access to base members.
- Protected inheritance prevents public access but allows further derived classes access.
- Friends of base classes are not friends of derived classes.
- Neither private nor protected inheritance prevents virtual functions from being overridden.

### **Exam Essentials**

- Be able to define public inheritance
- Understand the scope resolution operator syntax for accessing the base-derived class hierarchy
- Given a base-derived hierarchy, specify the access rules (including those of friend classes)
- Describe the scope access rules and purpose of public, protected and private inheritance
- Specify the implicit invocation order of constructors and destructors in a base-derived hierarchy

### **References**

[Meyers 2005 Chapter 6 (Items 38 and 39)]

[Stroustrup 2000 Section 12.2 and Section 15.3]

[Sutter 1999 Items 22 and 24]

[Sutter and Alexandrescu 2004 Items 34–7]

## **3.3 Dynamic Polymorphism – Virtual Methods**

This section looks at C++ support for *polymorphism*, specifically *dynamic* or *run-time* polymorphism, and related implementation issues. C++ also

supports *parameterized* or *compile-time/static* polymorphism, which is covered in Section 3.4.

Unless otherwise stated, all inheritance relationships are public.

## Critical Information

### ***Polymorphism defined***

The definition of polymorphism, along with its ubiquitous shape example, is possibly one of the most well-trodden in C++. A simplification would be: objects of different (but related) types can respond to the same method without the caller of the method needing to know the type of object that defines the method.

Thus a square, a circle and a triangle are all types of shape. A developer wishing to write a program to render them to a screen makes the shape type polymorphic by declaring a shape base class with a virtual draw method and providing all the derived shape classes (squares, etc.) with a redefined draw method which is called from within a single rendering routine.

### ***Substitution***

The key property of any programming language that supports dynamic or run-time polymorphism is that it should be possible to replace a base-class object with a derived-class object without affecting the behavior of the code which calls the interface. This is known as *substitution*, specifically referred to as the “Liskov substitution principle” after its inventor.

To support substitution, and good class design, a class is *specialized* by extending its interface and should not put any additional constraints, or stronger *preconditions*, on the use of the base-class interface functions. For example, if in the earlier example of read-only and read/write classes (see “Public Inheritance” in Section 3.2) `Base` was writable and `Derived` was read-only, the behavior of the calling code would be affected, as the writable `Set` method would no longer be available in the derived (read-only) class.

On the other hand, *post-conditions* may be strengthened. For example, if only modified values were returned from the `get()` method, the behavior of the calling program would not be affected as it would only act on a narrower band, or specialization, of values.

### ***Dynamic binding and static strong typing***

C++ is said to be a “statically strongly typed” language, that is, declarations and expressions are checked for correct usage at compile time. But dynamic polymorphism requires dynamic binding, where the exact object type is not known at compile time.

The solution to this dilemma is that for an object to be polymorphic, it has to be handled via either a pointer or a reference.

```
class Base
{
public:
    virtual void Foo();
};

// Derived class ...
class Derived : public Base
{
public:
    void Foo();
};

//
void SomeNonMember (Base* p)    // no binding to Derived
{
    p->Foo();                    // calls the correct virtual override
}

// calling
Base* p = new Derived();        // p is bound to Derived
SomeNonMember (p);
```

Section 3.2 stated that virtual methods ensure the correct version of a derived method is called within the base class, thus allowing a derived class to extend the base class with any modification.

In the same way, in the above example the `SomeNonMember()` function needs to deal only with `Base` pointers and the virtual override mechanism ensures the correct `Foo()` method is called – `Derived::Foo()`. If at a later stage a `DerivedTwo` class was also created in the same manner, `DerivedTwo::Foo()` would be invoked without any modification to `SomeNonMember()`.

It is possible for `delete` to be called on the `Base` pointer; this causes only the `Base` destructor to be called, leaking or orphaning any memory owned by the `Derived` class. Thus, any class that has at least one virtual method should declare its destructor `virtual`, ensuring that the derived destructor is called (the base destructor will be called after the derived destructor as normal).

### ***Overrides and overloads***

These terms can be a source of confusion so they have a short section of their own. Virtual overriding is a property of C++ support for dynamic binding and polymorphism, and operates *vertically* in a class hierarchy: `D::Foo()` overrides `B::Foo()`.

Overloading operates *horizontally* in a single class scope. Functions that share the same name but have different parameter lists are said to

be overloaded. `A::Set(int)` and `A::Set(char)` are overloads of a `Set()` function.

Overriding and overloading are completely different and share no relationship with each other.

## Overloading

Overloading allows a developer to provide more than one version of a function or operator within the same scope. Operator overloading allows user-defined types to behave in the same manner as integral types. Arguably, overloading provides a loose ad-hoc form of polymorphism via the same method having different types, but it has no claim to true OO polymorphism and is seen as a notational convenience.

```
class Foo
{
public:
    // Constructor
    Foo(int n = 0):iVal(n){}

    // Overloaded functions
    int Get() const {return iVal;} // Differentiate on parameter
    void Get(int* val) const {*val = iVal;} // lists, not on return type

    // Overloaded operators
    Foo& operator+=(int n) // Add an int
    {
        iVal+=n;
        return *this; // return (see this pointer section)
    }

    Foo& operator+=(const Foo& foo) // Add a Foo
    {
        iVal+=foo.iVal;
        return *this;
    }

    // conversion (or cast) operator:
    operator int&() // converts to an int(ref)
    {
        return iVal;
    }

private:
    int iVal;
};

// a binary overload outside class scope (value return)
Foo operator+(const Foo& a, const Foo& b)
{
    Foo res = a; // default initialization
    return res+=b; // Uses operator+=(const Foo&) overload
}
```

```
// In use ...
Foo foo(42);
foo+=100;                // foo.iVal == 142
Foo bar(1000);
foo+=1000;              // foo.iVal == 1142

int n = foo.Get();      // n == 1142
int m = 0;
bar.Get(&m);            // m == 1000
Foo foobar;
foobar = foo+bar;      // foobar.iVal == 2142

// Conversion using the operator int&()
int val = static_cast<int>(foo); // explicit compile-time
val = foo;                    // automatically call
```

Note that the conversion operator `int&()` may be used in conjunction with the `static_cast` operator or called automatically as part of the integral `int` assignment operator.

The examples also include an example of the binary operator `+` which is defined outside the class but implemented using the member operator `+=`. The binary operator could be defined within the class but with one fewer argument:

```
Foo Foo::operator+(const Foo& foo)
{
    Foo res = *this; // default initialization
    return res+=foo; // Uses operator+=(const Foo&) overload
}
```

The rules of overloading can become a little tricky. Here are the more common ones:

- Unary operator overload members take no arguments
- Binary operator overload members take one argument
- Non-member unary operators take one argument and non-member binary operators two arguments
- Ternary member operators are not supported
- The parameter list determines which overload is called and the return type is ignored
- Precedence and associativity of operators cannot be changed
- All overloaded operators can be inherited except the assignment operator
- It is not possible to overload operators using only built-in or integral types
- It is not possible to create completely new operators (for example `operator#`)

- It is not possible to change the arity (number of arguments) of any operator, for example a unary operator must remain unary.

Operators that may be overloaded:

<b>Unary operators</b>	+ - * & ~ ! ++ -- -> ->*
<b>Binary operators</b>	+ - */% ^ &   << >> += -= *= /= %= ^= &=  = <<= >>= < <= > >= == != &&    , [] () new new[] delete delete[]

Operators that may not be overloaded: . .\* ?: ::

Overload function-parameter-matching strategies in order of precedence:

Exact Match	Including simple conversions (e.g. array to pointer, non-const to const)
Match using promotion	Integral type-safe (smaller to larger) promotions (char to int, short to int etc.)
Match using standard conversions	Includes derived* to base*, int to unsigned, p* to void*
Match using user-defined conversions	Another already user-defined operator e.g. operator int& Foo to int. See above example.
Match using ellipses	Matching using ... is a last-ditch attempt i.e. any number of arguments

Any ambiguous match at the same level will be marked as an error, for example `Bar(T& ref)` and `Bar(T val)`. Calling with a value of type `T` is seen as an exact match for `Bar(T val)`, but because C++ treats references identically to values in this context, it is also an exact match with `Bar(T& ref)` and the compiler will flag it as an error.

### Virtual table

To support dynamic binding, and thus polymorphism, C++ uses a virtual table (*vtable*) to resolve the correct derived virtual method from a base-class pointer. The vtable is created in classes that declare at least one virtual function.

The vtable of a base-class object contains an array of pointers (offsets or addresses) to each of the derived-class overridden function implementations. Thus a pointer to a base-class object calls the correct derived method by dereferencing the pointer to the method.

Only the base-class object contains the vtable, with each derived-class object containing a pointer (*vptr*) to the table. When a method is called in a derived class, the *vptr* is used to get the correct method address from the vtable. This is an overhead compared to a non-virtual member method, which is statically linked by the linker and may be called directly.

### ***Multiple inheritance and abstract base classes***

The use of multiple inheritance is seen as controversial in general C++ and is definitely not encouraged in Symbian OS C++ except under specific circumstances. When a derived class derives from two bases, member name ambiguities can arise; in the case of diamond inheritance, where the base classes both derive from a common base, there is a complete duplication of data.

```
class B
{
protected:
    int iVal
};

class D1 : public B{};
class D2 : public B{};
class MI : public D1, public D2
{...} // which iVal ?
```

The class now has two versions of `B::iVal`.

Symbian OS C++ uses multiple inheritance purely as an interface inheritance mechanism. These interface classes are called *mixins* (see Chapter 4) and use the C++ language's support of *abstract base classes*.

An abstract base class is a class that cannot be instantiated, that is no object of that class may be directly created. Its primary purpose is to allow an interface to be declared without providing any implementation. Although it is possible for abstract classes to contain data members, this is not recommended.

For a class to be abstract it must contain at least one *pure virtual* member method:

```
class Interface    // Abstract Base Class
{
public:
    virtual void Foo()=0;           // Pure virtual method declaration
    // ...
};
```

```
class Impl : public Interface    // Derived
{
public:
    void Foo(){...}             // Definition
};
```

All pure virtual methods must be defined in the derived class, otherwise the derived class itself is abstract and cannot be instantiated.

In the case of multiple inheritance, it is valid to publicly inherit an abstract base class (interface), and privately inherit an implementation class – privately because it is good design to hide the implementation class interface from the user (see information on private inheritance in Section 3.2). It is also valid to inherit a number of abstract base class interfaces.

### ***Interface and implementation inheritance***

From the above sections it is possible to see the difference between inheriting an *interface*, or abstract base class, and inheriting an *implementation*. To summarize:

- Public inheritance means the interface is always inherited
- Pure virtual methods indicate the class is abstract and is an interface which must implemented in a derived class
- Ordinary virtual methods are interface functions that maybe overridden, but the base class provides a default implementation
- Non-virtual methods are the interface and mandatory implementation of the base class
- Private inheritance means the interface is not exposed to class users but is visible to the derived class; this is known as implementation inheritance.

## **Exceptions**

Mixins may contain actual implementation code for common default behavior.

## **Exam Essentials**

- Specify the mechanisms of OO reuse available in C++
- Be to able to state C++ support for polymorphism

- Understand the purpose of and difference between overriding and overloading
- Understand the use of overriding to modify behavior in base-derived class inheritance
- Understand the rules and pattern-matching criteria for correct overloaded-function invocation
- Identify the typical uses and behavior for operator overloading
- Describe the purpose of the virtual table, citing constraints and overheads
- Specify the use of virtual functions and their implementation tradeoffs
- Understand how an abstract base class is implemented in C++
- State the differences between interface and implementation inheritance
- Understand and recognize the problems associated with multiple inheritance
- Cite the implementation requirements to support the `static_cast` operator in user-defined classes

## References

[Ambler 2001 Chapter 7]

[Dewhurst 2005 Items 2, 21 and 33]

[Meyers 2005 Items 34, 40]

[Morris 2006 Chapter 5]

[Stroustrup 2000 Chapter 11, Section 12.2.6 and Section 15.2]

[Sutter 1999 Items 21 and 24]

[Sutter and Alexandrescu 2004 Items 26–31]

## 3.4 Static Polymorphism and Templates

The previous section focused on the use of virtual overrides for dynamic polymorphism. This section looks at static or compile-time polymorphism using C++ parameterized types called templates. Although standard C++ templates are not widely used in Symbian OS development, an Accredited Symbian Developer is expected to understand the basic terminology and semantics of C++ templates. Symbian OS thin templates are covered in the final part of this section.

## Critical Information

### *Template functions*

Templates can be tricky things to talk about, primarily because they fall between C++ programming and programming the compiler itself. A template is a function or a class that is decoupled from the specific type, or types, it acts on. The template is provided with a parameter list containing type names that allow the compiler to create type-specific instances of the function or class.

```
// template function for a simple addition function
template<typename T>           // template parameter list
T Add(T a, T b)               // function template name
{
    return a+b;
}

// int example
int a = 2;
int b = 2;
int res = Add(a,b);           // i.e. int Add(int, int)

// float example
float c = 2.2;
float d = 2.2;
float res2 = Add(c,d);        // i.e. float Add(float, float)
```

The `template` keyword is followed by the template parameter list, in which `<typename T>` tells the compiler to substitute `T` with the type of the argument being passed when it is generating code using the template.

Thus when the compiler encounters the `int` in the first example the code for `int Add(int a, int b)` is generated, and for the second example `float Add(float a, float b)`. Note the template parameter list maybe also be declared using “`class`” instead of “`typename`” that is `template<class T>`. These names are interchangeable when declaring a template parameter.

The reuse of the same generic code for different types is polymorphism. As the code is generated at compile time, and statically bound to the type passed in the template parameter list, it is known as static or compile-time polymorphism. The generated code allows calls to the function to be type-safe, a great advantage over using preprocessor macros, which have no type checking. It is also generally easier to debug templates than macros.

### *Template classes*

As well as function templates, C++ supports class templates.

```

// template class.
template<typename T>           // The parameter list
class Foo                     // The template name id
{
public:
    // constructor
    Foo();                    // default
    Foo(T val):iVal(val){}

    // Modifiers & Accessors
    void Set(T val){iVal=val;}
    T Get()const {return iVal;}

private:
    T iVal;
};

// in use
Foo<int> foo(0);             // Builds an int iVal class
Foo<float> boo(0.0);        // Builds a float iVal class

```

The semantics for template classes are the same as for template functions. Here two classes are generated from the template definition, one for dealing with an `int` and one for dealing with a `float`. Template classes may inherit other template classes:

```

template <typename T, typename C>
class Bar : public Foo<T>      // this is using a template
{
public:
    // ...

private:
    C iDerivedVal;
};

// in use
Bar<int, float> bar;
bar.Set(42);

```

The first parameter (T) is passed to the base class `Foo` and the second (C) is used by the derived class `Bar`.

Template classes are the same as any other class in many respects:

- They can have friends – a friend will be a friend of every class created from the template
- They have statics – each class created from the template has its own unique static, that is `class<int>` and `class<float>` will have separate static globals
- They can have nested classes

- They can inherit from non-templates
- They can have overrides and overloads.

### ***Template specialization***

In the template function example of `Add()`, above, the heart of the template is the addition expression `a+b`. This means the template makes sense for any type where `a+b` is semantically correct, including user-defined classes that override the addition operator `+`. But there are exceptions where adding is a valid operation but there is no `operator+` defined, or a function is required: for example, the `char*` type uses the function `strcat()` to add two strings together.

```
// Generic
template<typename T>           // template parameter list
T Add(T a, T b)               // function template name
{
    return a+b;
}

// Specialization
// The empty parameter list tells the compiler it is dealing with
// a specialization:
template<>
char* Add<char*>(char* a, char* b) // Template argument list
{
    return strcat(a,b);
}

// In use
char* res = Add<char*>("foo", "bar"); // note different syntax
int n = Add(2,2); // calling the int version
```

The empty `template<>` parameter list indicates to the compiler to expect a template specialization. An important note about terminology and syntax: the angle brackets in the statement `char* Add<char*>(char* a, char* b)` indicate an argument list and not a parameter list as in the declaration. They tell the compiler to create this version of the template when the `<char*>` type is encountered. The argument list must also be contained in the function call itself (`Add<char*>`) to ensure the compiler creates the correct version.

### ***Non-type parameters***

It is also possible to pass non-type (that is, value) parameters to templates. These are typically `const` values.

```
template<typename T, int size>
```

```

class Foo
{
    T iElements[size];    // an array of Ts
};

// In use
Foo<int,10> foo;          // Creates an array of 10 integers

```

### ***Introducing Symbian OS thin templates***

Symbian OS provides its own variation of templates. This is partly because compilers at the time of EPOC's inception lacked sufficient template support. More importantly, templates may lead to code duplication if not used correctly.

The Symbian thin template idiom works by implementing the essential functions (for example `Pop()` and `Push()` for a stack template) in a protected base class and using `void*` (`TAny*`) pointers to provide *type-agnostic* behavior. The base class is privately inherited (see Section 3.2) with the derived class using a regular C++ template to specify the type.

```

class StackBase                // Implementation class
{
protected:                    // template interface *hidden*
    void Push(void* item);
    void* Pop();

private:
    void* iStack[100];        // or some other internal representation
};

template<typename T>
class Stack : private StackBase
{
public:
    void Push(T* item) {StackBase::Push((void*)item);} // must be inline
    T* Pop() {(T*)StackBase::Pop();} // must be inline
};

```

The `Stack` class implements its functions inline to reduce the class overheads, only writing the code where it is needed. In effect the `Stack` template is a type-safe wrapper for the underlying type-agnostic `StackBase` container class.

### **Exceptions and Notes**

Friend template classes may be specialized for specific classes; for example inside a class template declaration, `friend class Bar<T>;` binds that class to be only a friend of classes of type `T`.

## Exam Essentials

- Specify the syntax for a simple function template specialization
- Be able to cite the advantages of function templates (for example over macros)
- Understand the inheritance rules and syntax supported by class templates
- Understand the syntax and semantics of a template type/class declaration
- Recognize the prototype declaration and pattern-matching properties of a template declaration and its use
- Understand the purpose and implementation differences of the Symbian OS thin template and mainstream C++ templates

## References

- [Alexandrescu 2001 Forewords (Meyers, Vlissides), Chapter 1]
- [Dewhurst 2005 Items 45–9]
- [Meyers 2005 Item 41]
- [Stichbury 2004 Chapter 19]
- [Stroustrup 2000 Section 2.7 and Chapter 13]
- [Sutter 1999 Item 23 Part 2]
- [Sutter and Alexandrescu 2004 Items 64–7]

