An Introduction to T Classes

This article provides a comprehensive introduction to Symbian T (type) classes

The Symbian C++ standard defines several types of classes which are easily identifiable by the naming conventions used. 'T classes' are one of these types of classes defined in the standards. The 'T' in their name stands for 'Type' because these classes behave similarly to the C++ built-in types.

Anatomy of a T Class

T precedes the name of a T class, according to the naming convention. This ensures that the users of the class will know from its name how it should be used. T classes are supposed to define types that can be used without any need for cleaning up.

T classes behave similarly to C++ built-in types, but underneath they are defined in one of several ways: some are just a `typedef` of a C standard type, some are enumerations and others hold a more extensive structure of data, similar to the use of `struct` in standard C, but with a well defined API.

T classes as typedefs

In Symbian C++, all the simple types are redefined as T classes. For example, int is `typedef`'ed as `TInt`, unsigned char is `TText8` and double is `TReal`. This is so that all the types, both simple and complex, are named in a standard way and are portable across different machine architectures. Even `void` is defined as `TAny`, but this is only used to mean that a pointer can point to any type of object. So,{{|TAny*}} is used instead of using `void*` and `void` is used when it means 'nothing,' for example, as a return type when you want to specify that a function does not return anything.

For example, the signature of a function that can send a command together with some data could be:

```
void SendCommand(TCommandCode aCommand, TAny* aData);
```

This function doesn't return anything, so the return type is `void`. The first argument, `aCommand`, has a value taken from an enumeration representing the list of commands. The data sent with the command depends on what command is sent and so the interface takes some generic data (the pointer to any type, `TAny*`) that will be converted accordingly by the receiver of the command.

T classes as enumerations

Enumerations are also T types; when defining an `enum`, the type name should be prefixed with a T and the enumeration members prefixed with an 'E'. For example, if you want to define the states for a state machine which connects to a server and sends messages, you could define it as follows:

```
enum TStates
    {
    EInitial,
    EConnected,
    EMessageSent,
    EMessageReceived,
    EDisconnected
    };
```

T classes as structures

T classes may be used as data structures, from simple structures with only a few member variables and a small API (just getters and setters), to complex structures with an extensive API that provides a lot of processing methods (for example, `TLex`). Sometimes, when T classes are used for simple structures and do not need to provide an API, the member variables are made public. For example, a structure for holding information about an event may be defined in the following way:

```
class TEvent
    {
    public:
    TDate iDate;
    TEventType iType;
    TReal iCost;
    };
```

You will find T classes that have everything public, but they are mostly classes internal to components, or classes that are not expected to be changed or extended after they are published (for instance, `TPoint`). This approach is not recommended for object-oriented programming when you have a class that is likely to be modified, especially if it is a published class, because it can restrict the ability to change and extend the structure. For example, if we later want to remove `iCost` and add in its place `iVenueCost`, `iStaffCost` and `iMaterialsCost`, we cannot do so without breaking all the existing users of `TEvent`. To avoid this problem, it is better to make the member variables private and provide access to them through an interface. A better-defined `TEvent` class would be:

```
class TEvent
    {
public:
    void SetDate(TDate aDate);
    TDate Date() const;
    void SetEventType(TEventType aType);
    TEventType EventType() const;
    void SetCost(TReal aCost);
    TReal Cost() const;
private:
    TDate iDate;
    TEventType iType;
    TReal iCost;
    TReal iRealReserved; // reserved for future use
    TInt iIntReserved; // reserved for future use
    };
```


(C) Copyright Nokia 2013. All rights reserved.

You can see that in this example there are two member variables which are reserved for future use. These are defined in case new variables need to be added at a later point; the size of the class will not change so the binary compatibility will be maintained. This is common practice for writing classes that can be extended later.

In this example, if we wish to split `iCost` into `iVenueCost` and `iStaffCost`, we can easily do so, by modifying `iCost` to be `iVenueCost` and `iRealReserved` to be `iStaffCost`, and then by modifying the `Cost()` method to return the sum of the two costs. We can also extend the API to provide getters and setters for the new member variables. The only tricky part is what to do with `SetCost()` method, since there will no longer be an `iCost` member variable. This method has to be kept in order to avoid breaking the users of the class, and you, as the writer of the class, should decide how to change this so that the users see the same behavior. (A possible solution is to make this method set `iVenueCost` to the argument and `iStaffCost` to zero. In this way, the new version of the class will behave like the initial version.)

T classes are also used for structures that provide a more extensive API but do not own any resources that need to be freed at cleanup time.

T classes have member variables which do not need any cleaning up, such as other T types. For this reason, T classes usually do not have destructors, since they do not need to release any resources. If the T class needs to use a more complex object that requires cleanup, it cannot own it, but can only hold a reference to it. It must be owned and cleaned up elsewhere.

It is possible for a T class to have a destructor defined. The destructor gets called when the object goes out of scope, in the case of a stack object, or if delete is called in the case of a heap object. It is important to note that when the T object is on the heap and has been added to the cleanup stack, the cleanup stack will not call the destructor in the case of a leave.1 If you have a T object on the heap and you need to call its destructor in case of a leave, then you need to use a `TCleanupItem`. The way this is done will be described later.

In general, T classes do not have a constructor, but when the class has exported virtual functions it needs to have a constructor which is exported. This is because exported T classes are constructed from outside the library in which they are defined, by the user of the class.

T classes do not automatically initialize their member variables; sometimes the author of a T class might wish to initialize the member variables on construction. In this case, the author will provide a default constructor and, usually, a constructor for initializing the variables with input from the user. For example, the constructors for the TEvent class could be defined in the following way:

```
TEvent::TEvent()
    : iDate(1,1,2006), iType(EUndefinedType), iCost(0)
    {}

TEvent::TEvent(TDate aDate, TEventType aType, TReal aCost)
    : iDate(aDate), iType(aType), iCost(aCost)
    {}
```

Because T classes hold data directly, copying from one object to another is straightforward: all the information gets copied directly and so there is no need to define a copy constructor or an assignment operator. The only situation where you might wish to define them is to prevent copying by defining them as private (or protected).

An example of a T class with a more extensive API might be one that holds some data, processes it and reports the results to a parent:

```
class TDateProcessor
    {
public:
    TDateProcessor(CSomeClass& aParent);
    void SetDate(TDate aDate);
    void AddWeeksToDate(TUint aNumberOfWeeks);
    void AddDaysToDate(TUint aNumberOfDays);
    void SubstractWeeksFromDate(TUint aNumberOfWeeks);
    void SubstractDaysFromDate(TUint aNumberOfDays);
    void CalculateDateOfFirstMondayInNextMonth();
...

private:
    TDate iDate;
    CSomeClass& iParent;
    };
```

This class holds a simple type, and a reference to a parent object which is more complex and requires cleanup. However, since it holds only a reference to the parent, it is clear that it does not need to clean the parent object up and so the class does not need to do any cleanup when it is not needed anymore.

Usually T classes are small, so they can be allocated on the stack. However, if a larger structure with lots of data is needed, the object can also be allocated on the heap.

Basic Usage Pattern

T classes are generally allocated on the stack because of their small size. If a class has member variables that are of a T class type, they can just use the object without any need for cleanup. For example, if `CMyClass` has a member variable of type `TEvent` and one integer for counting, it would define it in the following way:

```
class CMyClass : public CBase
    {
Public:
    ...
Private:
    TEvent iEvent;
    TInt iCount;
    ...
    };
```

The author of `CMyClass` does not need to worry about initializing `iEvent` and `iCount\`, since `iEvent` is initialized by the default constructor of `TEvent` (defined above) and `iCount` is initialized to 0,2 these variables can be used immediately; they do not need to be cleaned up and no memory needs to be released for them. The stack will automatically be deleted when the `CMyClass` object is destructed.

(C) Copyright Nokia 2013. All rights reserved.

When using a T Class object as a local variable, it is instantiated on the stack. The cleanup will happen automatically when the object comes out of scope. For example, a method to create a new `TEvent` and pass it on to another method would look like this:

```
void CreateAndUseEvent()
    TDate date(12, 8, 2008);
    TEvent event(date, EEventTypeExhibition, 15000);
    UseEvent(event);
    }
```

A method that returns an event would be written in the following way:

```
TEvent CreateAndReturnEvent()
    {
    TDate date(27, 11, 2008);
    TEvent event(date, EEventTypeConcert, 53700);
    return event;
    }
```

Note: Note that the return type is `TEvent`, not `TEvent&`. This is because the event object goes out of scope when the method returns, so the reference to it would be erroneous. In general, with T classes you need to watch out for them going out of scope since they are stack-based.

This return by copy works well if the T class is small in size as a copy would be quick. If this is not the case, you might want to do it in the following way:

```
void SetUpEvent(TEvent& aEvent)
    {...}

void SomeMethod()
    {
    ...
    TEvent event;
    SetUpEvent(event);
    ...
    }
```

In this case, `event` is created and owned by the parent method. This avoids the need to copy the class, which can take time if it is a large class.

One other way to ensure that the object does not go out of scope, and to avoid copying, is to instantiate it on the heap. In this case, you have to make sure that delete gets called on it at some point. If you are using it as a local variable and something can leave, you should add it to the cleanup stack. Here is an example of using a T class on the heap:

```
void SomeMethodL()
    {
    TDate date(7, 9, 2008);
    TEvent* event = new(ELeave) TEvent(date, EEventTypeParty, 3400);
    CleanupStack::PushL(event);
    DoSomethingL(*event);
    CleanupStack::PopAndDestroy(event); // equivalent to Pop(event)
                                        // followed by delete event
    }
```

In this example, `event` is added to the cleanup stack in order to avoid leaking the heap memory occupied by event. If event requires delete to be called on the object, you might consider using an automatic pointer, as in the following example:

```
class TEventAutoPtr
    {
    TEventAutoPtr(TEvent* aEvent);
    ~TEventAutoPtr();
    TEvent& operator*();
private:
    TEvent* iEvent;
    };

TEventAutoPtr::TEventAutoPtr(TEvent* aEvent)
    : iEvent(aEvent)

TEventAutoPtr::~TEventAutoPtr()
    {
    delete iEvent;
    }

TEvent& TEventAutoPtr::operator*()
    {
    return *iEvent;
    }
void SomeMethodL()
    {
    TDate date(7, 9, 2008);
    TEvent* event = new(ELeave) TEvent(date, EEventTypeParty, 3400);
    TEventAutoPtr eventPtr(event);
    DoSomethingL(*eventPtr);
    // there will be no deletion of event because eventPtr
    // owns it and will delete it when this method, SomeMethodL(),
    }
```

In this example, the automatic pointer takes ownership of `event` and deletes it when `eventPtr` goes out of scope. This is one way to deal with T classes on the heap that have destructors, but the problem is that you need to define your own auto pointer class. Another way is to use `TCleanupItem`, where you only need to define one method; this is shown in the next section.

As T classes are similar to structures, it is straightforward to copy them. You can use the assignment operator and this will ensure the information gets copied from the source object to the target one, so copying T classes is always safe. For this reason, when the size of a T class is small, it is passed as argument to a method by copying; for example:

```
void DoSomething(TEvent aEvent)
```


(C) Copyright Nokia 2013. All rights reserved.

If the event will be modified, pass the argument by reference so that the signature becomes:

```
void ModifyEvent(TEvent& aEvent)
```

Exceptions to the Common Usage Pattern

Two special T classes are `TAutoClose` and `TCleanupItem`. They are both provided for cleaning up objects that are added on the cleanup stack, even though they are not instantiated on the heap. As a result of this, and also because they own objects which need cleaning up, these classes are used differently to other T classes.

`TAutoClose` provides functionality for automatically calling `Close()` on a class when it goes out of scope. It is used for adding to the cleanup stack an object which should have `Close()` called for releasing resources, typically an R class object. `TAutoClose` is a templated class and it holds an object of the template type; the encapsulated object gets constructed through the `TAutoClose` class. The destructor of `TAutoClose` calls `Close()` on the object it owns so that when the variable goes out of scope, `Close()` is called automatically. If code may leave, you need to add the object to the cleanup stack, otherwise the `TAutoClose` destructor will not be called. This is done through the `PushL()` method of `TAutoClose`, as follows:

```
void AutoCloseExampleL()
    {
    TAutoClose<RMyClass> obj;
    obj.PushL();
    User::LeaveIfError(obj.iObj.Open());
    DoSomethingL(obj.iObj);
    obj.Pop();
    }
```

When the method `AutoCloseExampleL()` returns, the `RMyClass` object's `Close()` method is called. This is done automatically from the destructor of `TAutoClose`. Because `DoSomething()` can leave, `obj` is added to the cleanup stack by calling `PushL()` and `Pop()` gets called when it is not needed on the stack any more. `TAutoClose` is one of the few T classes that owns a more complex structure and has a destructor defined.

`TCleanupItem` allows even more flexibility with regard to what is called when an object from the cleanup stack gets destroyed. It holds a pointer to an object and another one to the method that should be called for cleaning up. After it is added on the cleanup stack, if there is a leave, or `CleanupStack::PopAndDestroy()` gets called, the method passed to the cleanup item will run. For example, if there is a large T class (so it needs to be instantiated on the heap) which also has a destructor, you can use `TCleanupItem` to ensure that the destructor gets called by the cleanup stack. This can be done in the following way:

```
void DoDelete(TAny* aPtr)
    {
    TSomeClass* ptr = static_cast<TSomeClass*>(aPtr);
    delete ptr;
    }
void DeleteTObjectExampleL()
    {
    TSomeClass* obj = new(ELeave) TSomeClass;
    CleanupStack::PushL(cleanupObj);
    DoSomethingL(obj);
    CleanupStack::PopAndDestroy(obj);
    }
```

When `PopAndDestroy()` gets called, the cleanup stack will pop the last item from the stack and check that its address matches the one passed as the parameter. If so, it calls the method given in the `TCleanupItem` for this object, `DoDelete()`. This method takes a `TAny*`as an input parameter. If we called delete on that pointer, the memory would be cleared, but the destructor of `TSomeClass` would not be invoked. This is because the compiler does not know the type of the pointer. To ensure that the destructor is called, the pointer has to be cast to the appropriate type, `TSomeClass*`.

So when a leave occurs, or `PopAndDestroy()` gets called, the cleanup stack will call `DoDelete()` for the pointer to `obj`, which will cause the destructor of obj to be invoked.

Some Common T Classes

This section provides information about some commonly used Symbian C++ T classes. The examples are divided into simple T classes, which are simple structures with straightforward APIs, and complex T classes, which provide a more elaborate interface.

Simple T classes

These are classes which are either just `typedefs` or simple structures.

- `TInt` is the Symbian C++ type used for integers. It is guaranteed to be at least 32 bits, but if an exact size is required the user should use other variants, such as `TInt32` and `TInt64`. If you need to use an unsigned integer then you should use `TUint`.
- `TBool` is the Symbian defined type for Booleans. It is a `typedef` for an integer, so it will actually occupy 32 bits. Booleans can be assigned to be either `ETrue` or`EFalse`, but you need to be careful about comparing them: just use logical operators, instead of comparing to `ETrue`. So, instead of if (`{{{1}}}`), use if (`someBool`). Also, don't compare two Booleans directly, for example, if (`{{{1}}}`), because even though `EFalse` is defined to be 0 and `ETrue` to be 1, any non-zero value is considered as true. Instead, use if (`{{{1}}}`). This works because the negation of `EFalse` will be `ETrue` (since negation of 0 is 1) and the negation of any non-zero value will be 0, i.e., `EFalse`.
- `TUid` is a class that holds a unique 32-bit number. It is used for the UIDs of processes or of plug-in interfaces. It has one member variable, a `TInt32` object, and it provides an appropriate interface for it: comparison operators and a static builder of the `TUid` object.
- `TPoint` is a class which holds the Cartesian co-ordinates of a two-dimensional point. All its members are public and can be manipulated directly. This is okay since the members of this class will not change (or at least are not expected to). The class also provides comparison and arithmetic operations on co-ordinates through the overriding of operators. For other similar classes you can also see `TSize` and `TRect`.
- `TPositionInfo` is used to pass information about the current position from the Location Acquisition API to the client application. It contains such information as the position (coordinates, speed, bearing, accuracy), the positioning mode, the update type etc. Its interface is comprised of setters and getters of the information fields.
- `THandleInfo` holds information on the usage of kernel objects. The information is retrieved from an `RHandleBase` object and it indicates the number of open handles to the kernel resource.

(C) Copyright Nokia 2013. All rights reserved.

Complex T classes

Complex T classes provide a more extensive interface, for example, classes that allow a lot of data processing.

- `TDesC` is the base class for descriptors (Symbian C++ strings) and provides the interface for handling the data: comparing, finding, matching, extracting portions of the descriptor or reading individual characters. The classes inheriting from `TDesC` provide the actual implementation of the descriptor and handle the transition from how the data is actually stored (on the heap or on the stack) to how it is presented to the user of the `TDesC` class.

There are two variants of descriptors, `TDesC8` and `TDesC16`, for 8-bit and 16-bit character representation.

Depending on whether the system supports Unicode or not, `TDesC` can be defined to be one or the other variant. In general you should use `TDesC`, but if it is important to have a fixed size for the characters then use the variant directly. For more information on the descriptor classes, see `TBufC`, `TPtrC`, `TDes`, `TPtr` and `TBuf`.

> **Note:** Note that not all descriptors are T classes: for example, `RBuf`.

`TDesC` provides the interface for copying descriptors as well, which makes the copying independent of how the descriptor is represented in the memory.

- `TLex` provides parsing functionality for a descriptor, or in general for a string. It stores the pointer to a string and two markers indicating the current item being analyzed and the next one. As for descriptors, there are 8-bit and 16-bit variants of `TLex`, `TLex8` and `TLex16`.
- `TTime` stores and manipulates the date and time. It stores the time internally as a `TInt64` representing the number of microseconds since midnight, 1 January, 0AD3 nominal Gregorian. The object can be built from an integer as well as from a string or a `TDateTime`, which is a structure that contains all the fields for recording time: year, month, day, hour, minute, second and microsecond. The TTime API allows the processing of the stored date, such as adding or subtracting intervals of time, finding out the interval between two given times, comparing two given times and parsing text to obtain a time value. The class also provides an interface for obtaining information in more readable formats, either by obtaining the year, month, day of the week and minute, or by formatting the time into a descriptor.

Best Practices

Using a T class is very easy and straightforward, but there are a few things you need to be aware of:

- Make sure that the T class object is not used once it goes out of scope. If you need to use the object, instantiate it on the heap and make sure that the memory gets freed when the object is no longer needed. If the class has a destructor which needs to be called, use a `TCleanupItem` and a method which invokes the destructor to add it to the cleanup stack.
- You should usually pass T class objects by value, but if it is a more complex class, like a descriptor, it is better to pass it as a reference. The decision should depend on the size of the class.
- When returning a T class, you should return it by value, rather than reference, if it is small. Otherwise, pass the reference to the object that will be changed into the function, or instantiate the object on the heap.

There are also some points that should be considered when creating a T class:

- The member variables of the T class should be other T classes or references to other classes. The T class should not own any resources that need to be cleaned up. If you want to use a pointer, consider defining the member variable as reference for clarity. The T class should not have ownership of any R class.
- If a destructor is necessary, although you can define it, consider whether you should define your class as a different type (most probably a C class). You should consider this especially if the size of the class is quite large and the object should be instantiated on the heap. Although you can have T classes with destructors on the heap and ensure they get cleaned up properly, the mechanism to do this is a lot more complicated than the one in place for C classes (which derive from CBase).
- Consider whether you need to define constructors or not. Since the member variables of a T class don't get initialized, usually it is good to provide a default constructor.
- Since the member variables of T classes are T classes themselves, or references to other objects, the `{{{1}}}` operator copies all the information from the source object to the target object, so there is no need to override it. If for any reason you want to prevent copying, make sure you make the copy constructor and `{{{1}}}` operator private.

In reality the first year is 1AD (which is same as 1BC), but the class, which is linked with `TDateTime`, starts counting from 0 and this is why it remembers the number of microseconds from the fictive year 0AD (which is same as 1BC).

- Consider if you should be defining other operators; for example, if you are writing a T class that holds the first name and surname of a person, it is good practice to provide comparison operators.

Conclusion

T classes are used for simple classes and types, like structures. They are generally small in size and do not require any cleanup and so this makes their use straightforward, as it is the case with any C++-defined type.

Further Reading

Fundamentals of Symbian C++/Class Types & Declarations#T Classes

© 2010 Symbian Foundation Limited. This document is licensed under the Creative Commons Attribution-Share Alike 2.0 license. See http://creativecommons.org/licenses/by-sa/2.0/legalcode for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.