



Contents

- [1 Overview](#)
- [2 MyApp.cpp file](#)
 - ◆ [2.1 MyApp.cpp in S60 3rd Edition](#)
 - ◆ [2.2 MyApp.cpp in S60 1st & 2nd Editions](#)
 - ◆ [2.3 Modifying MyApp.cpp](#)
- [3 CMyAppApplication class](#)
- [4 CMyAppDocument class](#)
- [5 CMyAppUi class](#)
- [6 CMyAppContainer class](#)

Overview

When a new project is created, 4 classes are automatically created with it. If the application name is, for example, MyApp, these classes are typically called CMyAppApplication, CMyAppAppUi, CMyApp and CMyAppDocument. Each of them has two corresponding files: header .h and implementation .cpp. There is also a file called MyApp.cpp which does not contain any classes. So, all in all there are 9 files even without any classes explicitly created by the developer. While it may seem too much for simple HelloWorld application, this "overhead" does make sense in a larger application. This article explains the roles served by these classes and additional file and their typical use. Both old (pre 9.0) and new (9.0 onwards) [Symbian OS](#) versions are considered.

MyApp.cpp file

This file provides an entry point for the application. If you have experience with Borland Delphi, you may think it is somewhat similar with .dpr file. Execution starts here, but very soon leaves the file.

NOTE: This file differs between [S60 3rd Edition](#) and older [S60](#) versions. This is caused by the change of the targettype from [APP](#) to [EXE](#) (see [Symbian Developer Library](#) for more information about different target types). However, the functionality of this file remains the same.

MyApp.cpp in S60 3rd Edition

```
#include <eikstart.h>
#include "MyAppApplication.h"

LOCAL_C CAppApplication* NewApplication()
{
    return new CMyAppApplication;
}

GLDEF_C TInt E32Main()
{
```

Description_of_the_classes_automatically_created_with_the_project

```
return EikStart::RunApplication( NewApplication );  
}
```

E32Main() serves as an entry point for the whole process. You can think of it as of an equivalent of main() or WinMain() functions in DOS and Microsoft Windows C++ respectively. This function is exported and called when the application is launched. Its task is only to tell the framework how to construct the CApaApplication subclass corresponding to our application. NewApplication() does exactly that: creates our application's CMyAppApplication and returns pointer to it.

NOTE: NewApplication() cannot leave. So it just returns NULL if out of memory.

MyApp.cpp in S60 1st & 2nd Editions

```
#include "MyAppApplication.h"  
#include <stdio.h>  
  
GLDEF_C TInt E32Dll(TInt TDllReason)  
{  
    return KErrNone;  
}  
  
EXPORT_C CApaApplication* NewApplication()  
{  
    return new CMyAppApplication;  
}
```

This file exports two functions, however only one is really useful. Remember, prior to Symbian OS 9.0 all the applications used to be not EXEs, but APPS, and APPS are just some special kind of DLLs. E32Dll() is exported by any DLL, and it is quite similar to DllMain() in Microsoft Windows. It is called whenever dll is loaded/unloaded to/from the process address space and whenever a new thread is started or terminated. Usually it just returns KErrNone indicating no error.

NewApplication() is called by the S60 framework immediately after the application's APP file is loaded. This function creates the CApaApplication subclass and returns pointer to it.

NOTE: NewApplication() cannot leave. So it just returns NULL if out of memory.

Modifying MyApp.cpp

While technically possible, it is practically of no use. Most of the framework is not initialized at this stage, so it cannot not be used. Only very specific, low-level actions should be put into the MyApp.cpp file.

CMyAppApplication class

CMyAppApplication.h file

```
#ifndef MYAPPAPPLICATION_H  
#define MYAPPAPPLICATION_H  
  
#include <aknapp.h>
```

Description_of_the_classes_automatically_created_with_the_project

```
class MyAppApplication : public CAknApplication
{
public:
    TUid AppDllUid() const;

protected:
    CApaDocument* CreateDocumentL();
};

#endif
```

CMyAppApplication represents application as a whole. This class is not often used directly, but this class, in essence, made the difference between a generic DLL and concrete APP in Symbian OS prior 9.0 and it makes the difference between a generic EXE, which can be a server, for example, and the application in Symbian OS 9.0 and later.

The most important piece of information when dealing with an application is its UID and CMyAppApplication::AppDllUid() (name retained from the past) serves just to let anyone interested to get that information. This function lets the framework distinguish one application from another.

NOTE: UID returned by CMyAppApplication::AppDllUid() must correspond to the UID defined in the MMP file. Otherwise application will fail to launch. The code example presented here uses ILLEGAL application UID. See Symbian Developer Library for more information concerning UIDs

CMyAppApplication::AppDllUid() creates document class CMyAppDocument, associated with our application. The role of a document class will be dwelled upon a bit later.

Implementation of CMyAppApplication class is quite straightforward.

```
#include "MyAppDocument.h"
#include "MyAppApplication.h"
const TUid KUidMyApp = { 0x12345678 };
    //NOTE: This UID must correspond to the UID defined in the MMP file

CApaDocument* MyAppApplication::CreateDocumentL()
{
    return (static_cast<CApaDocument*>
        ( MyAppDocument::NewL( *this ) ) );
}

TUid MyAppApplication::AppDllUid() const
{
    return KUidMyApp;
}
```

CMyAppDocument class

CMyAppDocument.h file

```
#ifndef MYAPPDOCUMENT_H
#define MYAPPDOCUMENT_H

#include <akndoc.h>

class CEikAppUi;
```

CMyAppApplication class

Description_of_the_classes_automatically_created_with_the_project

```
class CMyAppDocument : public CAknDocument
{
public:
    static CMyAppDocument* NewL(CEikApplication& aApp);
    virtual ~MyAppDocument();

private:
    CMyAppDocument(CEikApplication& aApp);
    void ConstructL();

private:
    CEikAppUi* CreateAppUiL();
};

#endif
```

CMyAppDocument represents the data application is working with. This idea is quite familiar to those acquainted with this MVC (Model-View-Controller) application architecture. In this architecture, the term CMyAppDocument is the model of the application.

In real life, however, some applications defently have an entity represented by CMyAppDocument (for example word-processing or spreadsheet programs), but others mostly lack it. For example, most (if not all) kinds of game applications do not need to create documents. Calculators also do not use file-saving much.

But in all the applications a subclass of CAknDocument is used to create an application UI, and it is the UI which is used to interact with user and in that way drive all the application features.

CMyAppDocument::CreateAppUiL() virtual function does exactly that.

The rest of the methods in CMyAppDocument deal with the two-phase construction. If you are new to this idiom, it is highly recommended to study it as soon as possible for it is one of the key idioms of Symbian OS. You can check Symbian Developer Library for more information about two-phase construction and related topics.

```
#include "S60ResourceLabDocument.h"
#include "S60ResourceLabAppUi.h"
CMyAppDocument::CMyAppDocument(CEikApplication& aApp)
: CAknDocument(aApp)
{
}

CMyAppDocument::~~CMyAppDocument()
{
}

void CMyAppDocument::ConstructL()
{
}

CMyAppDocument* CMyAppDocument::NewL(
    CEikApplication& aApp)
{
    CMyAppDocument* self = new (ELeave) CMyAppDocument( aApp );
    CleanupStack::PushL( self );
    self->ConstructL();
    CleanupStack::Pop();
    return self;
}
```

Description_of_the_classes_automatically_created_with_the_project

```
CEikAppUi* CMyAppDocument::CreateAppUiL()
{
    return new (ELeave) CMyAppUi;
}
```

CMyAppUi class

CMyAppUi.h file

```
#ifndef MYAPPUI_H
#define MYAPPUI_H

#include <eikapp.h>
#include <eikdoc.h>
#include <e32std.h>
#include <coeccntx.h>
#include <aknappui.h>

class MyAppContainer;

class MyAppUi : public CAknAppUi
{
public:
    void ConstructL();
    ~MyAppUi();

private:
    void DynInitMenuPaneL(TInt aResourceId, CEikMenuPane* aMenuPane);

private:
    void HandleCommandL(TInt aCommand);
    virtual TKeyResponse HandleKeyEventL(
        const TKeyEvent& aKeyEvent, TEventCode aType);

private:
    MyAppContainer* iAppContainer;
};

#endif
```

CMyAppUi class represents the Controller part of the MVC application architecture. This class receives user input, in form of keypresses (CMyAppUi::HandleKeyEventL()), mouse movements and clicks, menu commands (CMyAppUi::HandleCommandL) et cetera. Its responsibility is to process these events, make necessary changes to the Model of the application and issue order for changing the application View (for example, it can change application menu before it is shown; CMyAppUi::DynInitMenuPaneL must be overloaded for this functionality). Quite often the Model part of the application is platform-independent, that is it can run on any Symbian OS platform with no or minimal changes; UI part is practically always platform-dependent. Ideally it should encapsulate all the platform-dependent code.

Of course the application can process user input in any form it wants to, so this file is practically always changed and extended by the developer. Methods presented here are just some basic function that should be overloaded, they are provided as a starting example.

In this example (and nearly always in real life) CMyAppUI uses CMyAppContainer. However, it is not necessary the case. Symbian OS and application framework do not dictate any particular way of presenting

Description_of_the_classes_automatically_created_with_the_project

information to the user.

Sample CMyAppUi implementation

```
#include "MyAppUi.h"
#include "MyAppContainer.h"
#include <MyApp.rsg>
#include "MyApp.hrh"

#include <avkon.hrh>

void MyAppUi::ConstructL()
{
    BaseConstructL();
    iAppContainer = new (ELeave) MyAppContainer;
    iAppContainer->SetMopParent(this);
    iAppContainer->ConstructL( ClientRect() );
    AddToStackL( iAppContainer );
}

MyAppUi::~MyAppUi()
{
    if (iAppContainer)
    {
        RemoveFromStack( iAppContainer );
        delete iAppContainer;
    }
}

void MyAppUi::DynInitMenuPaneL(
    TInt /*aResourceId*/, CEikMenuPane* /*aMenuPane*/)
{
}

TKeyResponse MyAppUi::HandleKeyEventL(
    const TKeyEvent& /*aKeyEvent*/, TEventCode /*aType*/)
{
    return EKeyWasNotConsumed;
}

void MyAppUi::HandleCommandL(TInt aCommand)
{
    switch ( aCommand )
    {
        {
            case EAknSoftkeyBack:
            case EEikCmdExit:
                {
                    Exit();
                    break;
                }

            default:
                break;
        }
    }
}
```

CMyAppContainer class

CMyAppContainer.h file

```
#ifndef MYAPPCONTAINER_H
#define MYAPPCONTAINER_H

#include <coecntrol.h>

class CEikLabel;

class MyAppContainer : public CCoeControl, MCoeControlObserver
{
public:
    void ConstructL(const TRect& aRect);
    MyAppContainer();
    void SizeChanged();
    TInt CountComponentControls() const;
    CCoeControl* ComponentControl(TInt aIndex) const;
    void Draw(const TRect& aRect) const;
    void HandleControlEventL(CCoeControl* aControl, TCoeEvent aEventType);

private:
    CEikLabel* iLabel;
};

#endif
```

NOTE: Unlike previous classes and `MyApp.cpp` that are required by the application framework, the `MyAppContainer` object can be absent. However, in real life it is nearly always present, in some form.

`MyAppContainer` is basically the main windows of the application, it contains (hence the name) all other controls. Quite often there are many containers in a single application, and `CMyAppUi` switches them as necessary. Again this class is heavily changed and extended by the developer to suit his or her needs. In this particular example `CMyAppContainer` has only one label control.

The key functions of `CMyAppContainer` shown in this example are

`MyAppContainer::SizeChanged()`, `MyAppContainer::CountComponentControls()`, `MyAppContainer::ComponentControl()`, `MyAppContainer::Draw()` and `MyAppContainer::HandleControlEventL()`. They are not complicated but crucial for interaction with the framework.

- `MyAppContainer::SizeChanged()` is called during initialization of the container and when its size is changed. It must rearrange all the contained controls according to the new size.
- `MyAppContainer::CountComponentControls()` returns the number of controls currently in container.
- `MyAppContainer::ComponentControls(TInt aIndex)` returns control whose number is `aIndex`. Components are numbered starting from zero.
- `MyAppContainer::Draw()` draw only container specific graphics. All the controls, contained within the container are drawn by the framework without further coding. Obviously two previous functions are used for this functionality.
- `MyAppContainer::HandleControlEventL()` must handle all the events that contained controls can generate. For example, using of lists requires overloading of this function.

Description_of_the_classes_automatically_created_with_the_project

CMyAppContainer::Draw() sample implementation

```
#include "MyAppContainer.h"
#include <MyApp.rsg>

#include <eiklabel.h>
#include <coemain.h>

_LIT(KTextHelloWorld, "Hello World!");

void MyAppContainer::ConstructL(const TRect& aRect)
{
    CreateWindowL();

    iLabel = new (ELeave) CEikLabel;
    iLabel->SetContainerWindowL( *this );
    iLabel->SetTextL(KTextHelloWorld);
    CleanupStack::PopAndDestroy();

    SetRect(aRect);
    ActivateL();
}

MyAppContainer::~MyAppContainer()
{
    delete iLabel;
}

void MyAppContainer::SizeChanged()
{
    iLabel->SetExtent( TPoint(0,0), TSize(Rect().Width()/2, Rect().Height()/2));
}

TInt MyAppContainer::CountComponentControls() const
{
    return 1;
}

CCoeControl* MyAppContainer::ComponentControl(TInt aIndex) const
{
    switch ( aIndex )
    {
        case 0:
            return iLabel;
        default:
            return NULL;
    }
}

void MyAppContainer::Draw(const TRect& aRect) const
{
    CWindowGc& gc = SystemGc();
    gc.SetPenStyle(CGraphicsContext::ENullPen);
    gc.SetBrushColor(KRgbGray);
    gc.SetBrushStyle(CGraphicsContext::ESolidBrush);
    gc.DrawRect(aRect);
}

void MyAppContainer::HandleControlEventL(
    CCoeControl* /*aControl*/, TCoeEvent /*aEventType*/)
{
}
```