LotusScriptor's

Plain Simple

Guide To The

NOTES C++ API

By Lee Powell

A Step by Step Guide with over 100 illustrations

www.notesapi.com

LotusScriptor's

Plain Simple

Guide To The

Lotus Notes C++ API

Published by Lee A. Powell
www.notesapi.com
September 2002

ISBN: **0-9543159-0-1**

Contents

Introduction

Have you ever worked on a solution to a problem whereby you found yourself cornered with no visible way out. You've tried solving the problem using Formula language and LotusScript, and a whole host of other strange work a rounds that become part of life when you program in Lotus Notes Domino. You try and try but to no avail. You've maybe heard other folks saying, "only the API could do that", but that in itself creates an even bigger problem and scares the life out of you. C and C++ yuk! That gets ugly! You simply do not have time to spend beating your head against a wall, learning the API and still meet next week's delivery deadline for your application.

Well there's good news. The C++ Notes API is really quite simple to use, although I admit myself, the C API is still not for the feint hearted. The C++ API is very similar to LotusScript in its object-orientated structure. And that's a massive advantage to developers already familiar with LotusScript. What has frustrated me over the past eight years as a Notes developer, is the distinct lack of introductory level material for the API's in general. I have often downloaded the API toolkits from Lotus and tried to understand them, and have after a few hours, sat back in my chair with a heavy sigh and said to myself, "this is all too difficult, I'll get around to it later".

Fortunately I did get back to the C++ API a few years back. I literally slogged my way through the toolkit documentation. Finally, I managed to get my head around it all. Realizing that it was all pretty simple stuff. If only someone could have shown me the very basics first I could have been using the API in a day and not two weeks like it took me. Once you get this basic understanding, the Lotus Toolkit documentation becomes your friend not your enemy, and before you know it you're writing applications that leave your work mates wondering how the hell did he do that? You boss will think your some kind of guru and could even be inclined to bump up your pay and extend your contract. Nice thought!

In light of this fact I created this very simple book that explains very graphically how to create C++ API programs for Notes. Whether it be an application or dll, I cover them here. I assume you are a Notes developer, and can program in LotusScript, I assume no C++ experience. I try to draw comparisons between LotusScript and the C++ API all the time, so that you can understand the API quicker. I also cover some basic C++ syntax and once again try to show you the LotusScript equivalent if there is one. I have found this method really helps Notes developers pick up and understand the API basics very quickly, as it explains new material in context to material they are already familiar with.

This book is not for the experienced C++ API programmer, it's only for Notes or VB developers who have very little or no experience in using the C++ API. It will get you to a place quickly whereby you can create some very useful API programs, and give you enough understanding to comfortably develop your skills further without feeling

out of your depth when exploring C++ API documentation. Although the book is presented in a simple fashion, don't be fooled by it's apparent simplicity. You will quickly become equipped to write some pretty smart and efficient programs that you just could not write before using LotusScript or Formula language. The API is very powerful and I think it's important that more developers realize it's potential.

Here's to your next killer app using the C++ API. May it bring you prosperity as you turn Black Belt in the art of Lotus Notes programming.

Dedication & Thanks

This book is dedicated to two very special people:

Firstly, to my father who has always been a constant source of encouragement to me. My career in the IT arena was born out of frustration with my life the way it was. Thanks Dad for all those encouraging conversations strolling down at Manly beach where you encouraged me to go for it. Thanks for teaching me early in life that people become what they believe they can be.

Secondly to Warren Hurt. An experienced and seasoned Australian businessman. Thanks Warren for giving me a shot into the IT contracting game all those years ago. I turned up at your office in Sydney with nothing but desire, the new suit and tie I'd purchased on my sister in law's credit card the day before. I had no IT experience at the time, but you saw my enthusiasm and gave me a break. Thanks for believing in me, my life has never been the same since thanks to you.

Special thanks also goes to five great Notes Domino Developers:

Tim McNamara

Dean Speer

Fergal Burnell

Victor Kaffa

Sol Whyte

Thanks for all your comments and constructive critique in putting this thing together. I guess I owe you guys a few pints then?

Finally, thanks to Peter Kemp for interpreting my thoughts into pictures with the book cover design, you did a great job!

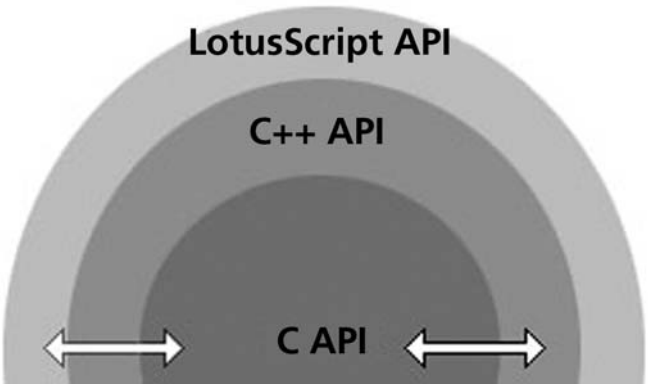
CHAPTER 1

Overview

The Notes C++ API is a C++ library that contains a set of C++ classes. These classes make it very easy to write simple programs that allow the developer to access, create and manage nearly all facets of Notes databases. The API is basically an interface built on the Notes C API. While the C API acts directly on the Notes program files, the C++ API has an additional degree of separation because the C++ API is really a wrapper for the C API.

What does this mean? Well the C++ API basically calls the C API, which in turn acts on the Notes program files. The advantage of this separation is that the C++ API is easier to use because it has an object orientated interface, similar to LotusScript, only much more granular and comprehensive. A NotesSession object in LotusScript is similar to the C++ LNNotesSession object, same too for NotesDatabase, NotesView and so on. This makes the C++ API much more intuitive for the LotusScriptor. The primary difference is that the C++ API accesses parts of the Notes object model that LotusScript can not currently access, it runs faster too because it does not need to run within the overhead of the Notes client.

The following diagram shows how the APIs interact with each other:



Here's a brief list of the types of things you can do using the C++ API from the toolkit documentation:

- Create, copy, modify, and delete folders, views, forms, subforms, shared fields, agents, actions, formulas, LotusScript and simple actions.
- Work on rich text like you always wished you could in LotusScript.
- Create, copy replicate and delete databases.
- Get and Set MANY database design properties.
- Create, copy and delete database help documents.

- Copy and delete database icons.
- Manipulate folder contents as well as move them.
- Read and modify design properties for views, folders, agents, actions, forms and subforms.
- Transfer information between Notes and non Notes databases such as an object orientated database and vice versa.
- Perform server side periodic maintenance tasks on Notes databases.

CHAPTER 2

Types of Applications You Can Write Using the C++ API

There are three basic types of programs you can write using the C++ API. These are:

Stand alone applications (exe files)

Stand-alone applications are programs that make API calls into the native Notes software. In order to facilitate this you must have the Notes software installed on the machine but the Notes client software does not necessarily need to be running when the API program is run.

Dynamic link libraries (dlls)

Dynamic link libraries are triggered by calls to their export functions. These dlls, can then in turn be called directly from LotusScript or any other application, such as Visual Basic.

Notes server add-in tasks

Server add-in tasks allow you to extend the Domino server software using the API. These custom server tasks run along side the standard Notes server tasks.

In all these program formats, the core API code is the same. In effect, all the same API functions can be called identically regardless of your program format. The only difference lies in the need to add additional API calls when creating server add-in tasks. Obviously stand-alone and dll files will be created in a slightly different manner, but the API part of the program remains the same. So learning to create different program types is very simple once the basics of the API are understood.

CHAPTER 3

What You Need to Get Started

Before you can start coding using the C++ API you will need to download and install the C++ API Toolkit from Lotus and configure the environment. In this book we will be using Microsoft's Visual C++ 6.0.

Note: *I have just heard that Borland have their own C++ development environment available for free trial download. Whilst, this book only focuses on Microsoft's Visual C++ environment, you may wish to try the Borland product as well. The C++ toolkit has information on how to set up the Borland environment. So if you really do not wish to fork out for Visual C++, then this could be another option.*

The last time I checked the Lotus site you could download the C++ API toolkit from the following URL:

http://www.lotus.com/developers/devbase.nsf/homedata/homecpp

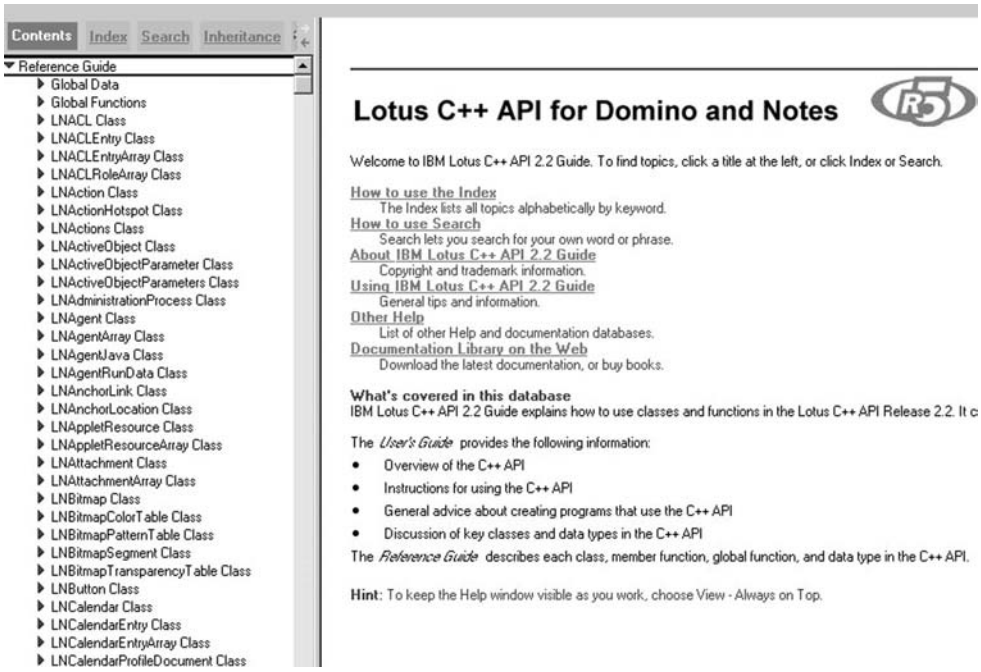
However, this download has tended to move around a bit since the merge of the IBM and Lotus sites, so you may need to search around both sites.

The 2.2 Toolkit is 4mb is size. If you cannot find the toolkit, try searching for "Notes C++ API Toolkit" on either the Lotus or IBM sites, it should appear in the search results.

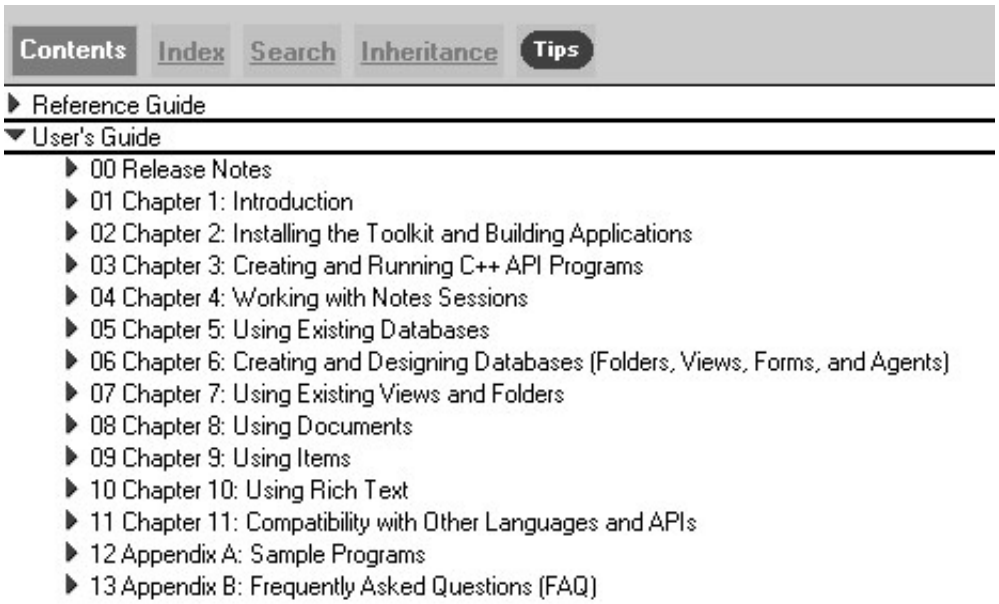
Download version 2.2 of C++ API toolkit for the Windows 32 bit operating system, it is free but requires registration.

In my examples I use Notes 5, which uses version 2.2 of the C++ API toolkit (Released first quarter 2002). I have also tested the examples with Notes 6 without any problems. If you are using Notes 4, you'll need to download 2.01 of the toolkit. I strongly advise downloading the toolkit documentation at the same time as this will prove an invaluable resource as you become familiar with the API. The documentation is titled: 'C++ API Reference Guide' and is contained in a Notes database which looks like the following:

In this comprehensive reference guide you will find details of all the classes, functions and data types as shown in the screenshot snippet below:



Also in the reference guide you will find details on setting up all different kinds environments such as UNIX and IBM's OS/2 - not just 32bit Windows. Included also, are brief bits of sample code and descriptions of how the API sees all the class objects and how they interact with each other. A screen shot is shown below of the types of information available.



We will cover how to setup the environment for a Windows 32bit platform, with a clean install of Microsoft's Visual C++ 6.0 and Notes Release 5.

So let's get started. After downloading the API toolkit, unzip it and extract all the files into the root of your hard drive, which in turn will create a NOTESCPP directory.

CHAPTER 4

Visual Studio Settings

Environment Variable Setup

If you let Visual C++ 6.0 set up your environment variables when you installed you're all set, other wise you may need to refer to the "About Environment Settings" for you operating system in the C++ API Reference Guide.

Compiler Setup Test

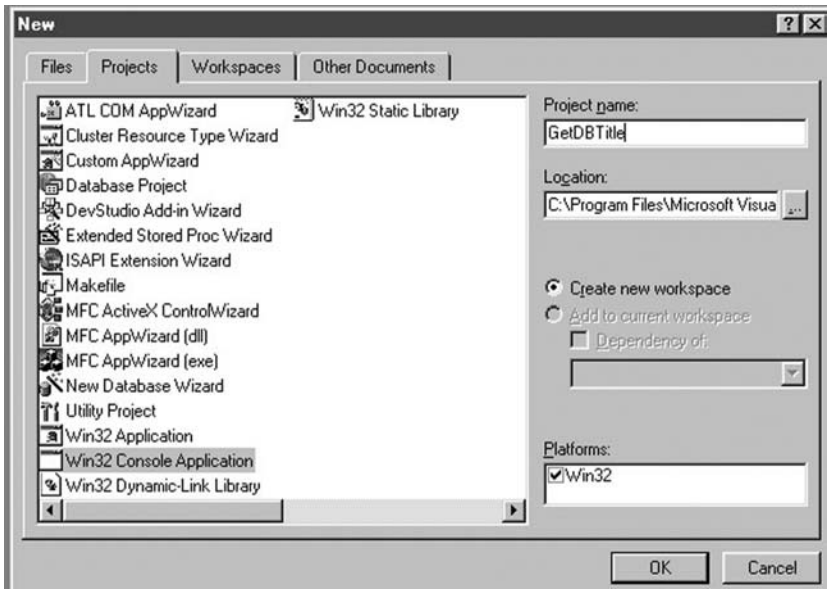
What we are going to do here is test one of the sample programs downloaded with the toolkit to ensure that your environment is setup correctly. Whilst ultimately you will create other types of programs yourself like dll's or server programs, the settings you define in the C++ environment will remain consistent even though the types of programs you create may change. It is critical that you set up these settings exactly as I have outlined here. This will save you much heart ache, so learn them well. If you can't get a future program to compile, chances are your settings are screwed somewhere. You should always check these first if you can not see any errors in your code.

So let's get started with the test program: GetDbTitle.exe

The first thing you will need to do is launch Visual Studio's Visual C++

Create a new project by selecting File > New

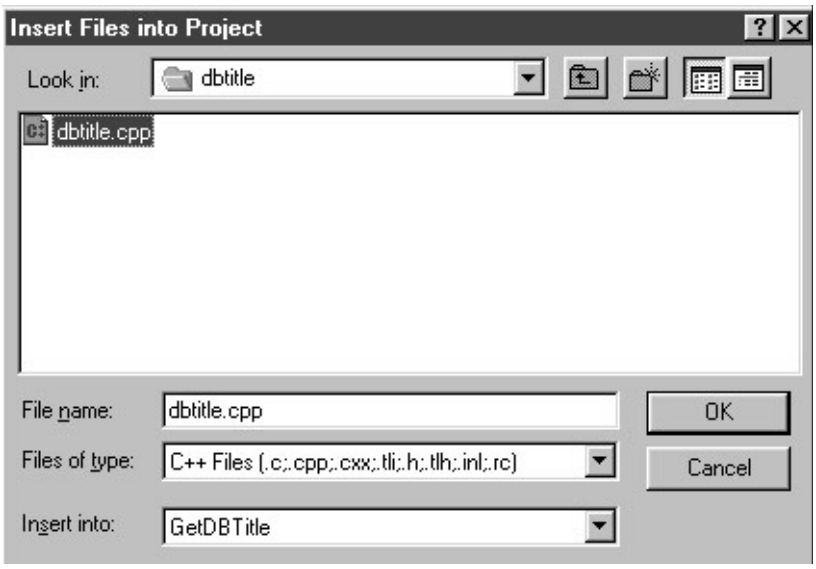
Select Win32 Console Application and enter 'GetDBTitle' for the project name and click OK.



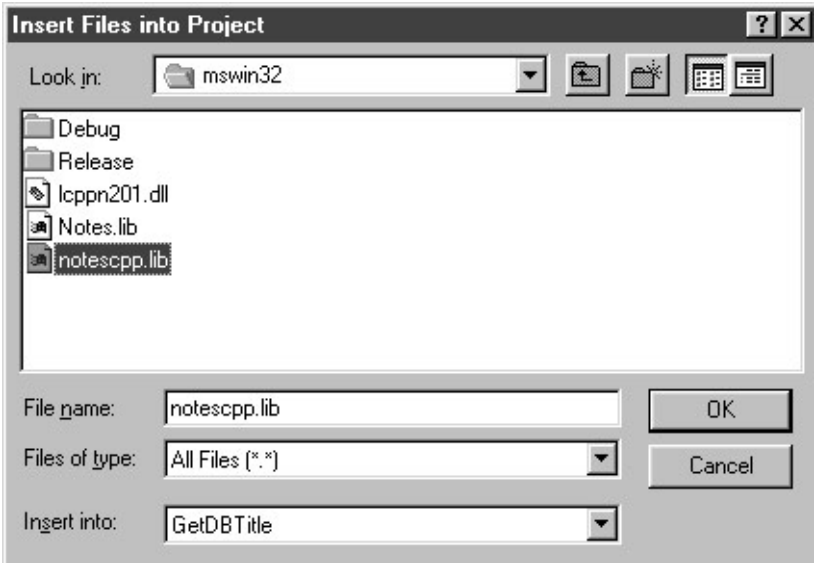
Click Finish, then OK.

Next, you need to add some source code to the project, so select Project > Add to Project > Files

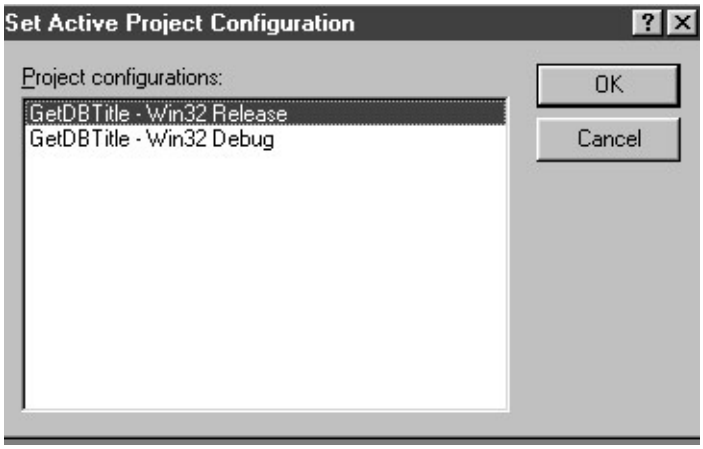
Select: C:\NOTESCPP\SAMPLES\DBTITLE\DBTITLE.CPP and click OK



Next, add the API library to the project as you will need to in all other projects you create. Select Project > Add to Project > Files and add the lib file C:\NOTESCPP\LIB\MSWIN32\NOTESCPP.LIB and click OK. If you can't see the file, make sure that *files of type* is set to *.*.

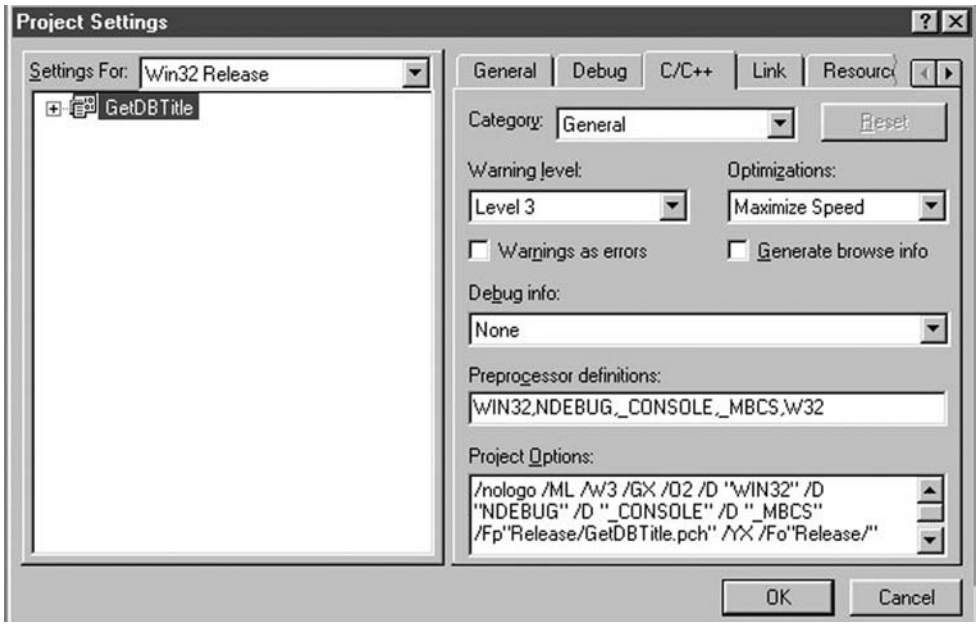


Next, you need to amend the project settings, once again you will need to do the following steps for each and every program you create using the API. Select Build > Set Active Configuration > Win32 Release > OK

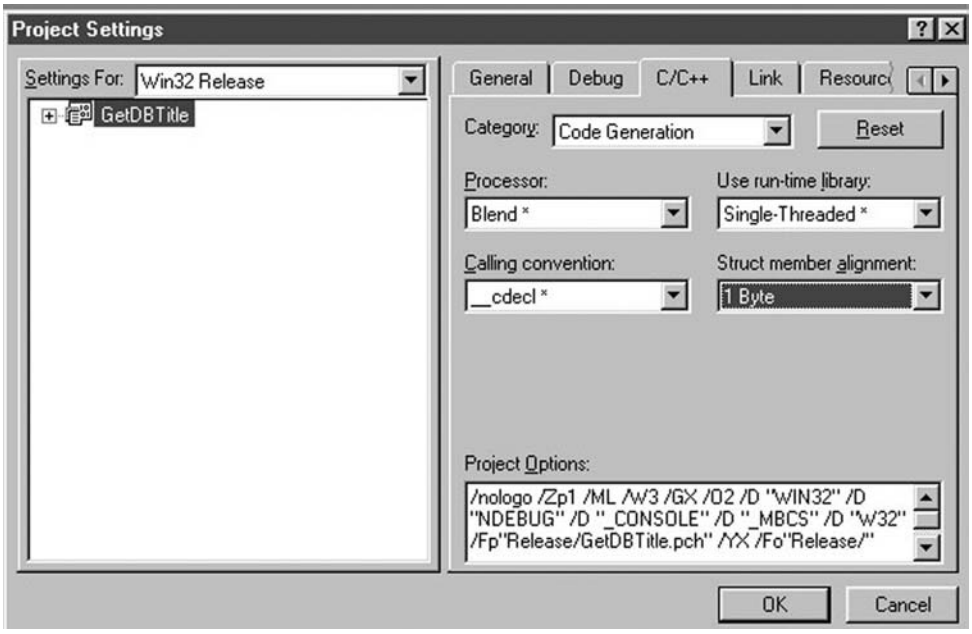


Select Project > Settings, choose the C/C++ tab and make the following changes:

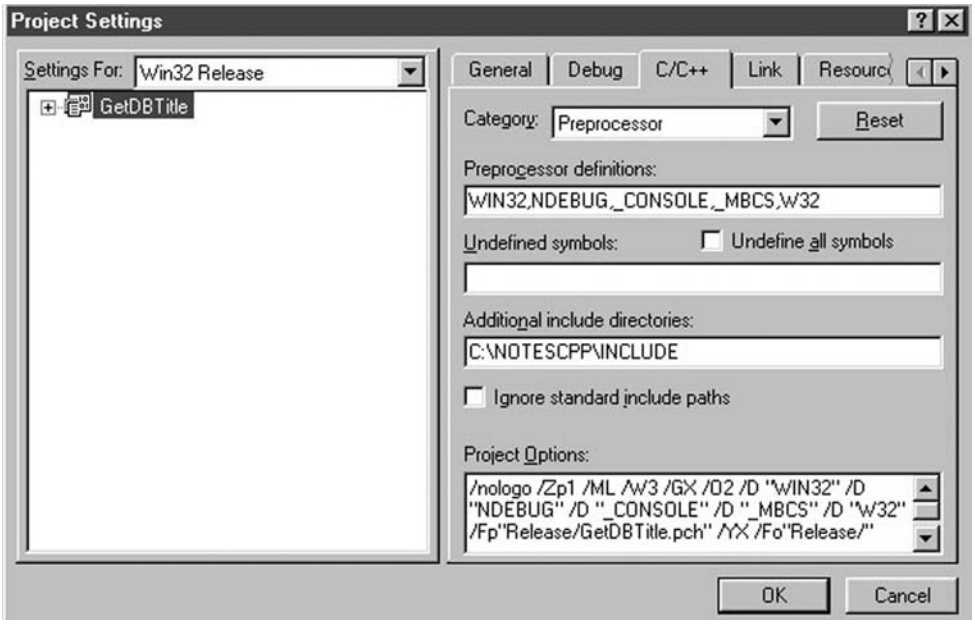
In the General Category, add W32 to the list of preprocessor definitions exactly as shown in the following diagram:



In the Code Generation category, set Struct Member Alignment to 1 Byte, as shown in the following diagram:



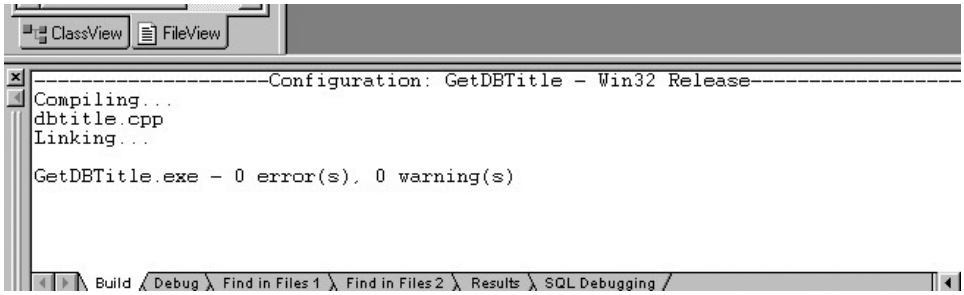
In the Preprocessor category, add 'C:\NOTESCPP\INCLUDE' to the Additional Include Directories as shown in the following diagram:



Click OK

Next, you need to build the program, select Build > Build GetDBTitle.exe

Wait to see if you get any errors or warnings. If the program did not compile and you did not see a report screen like the one below:

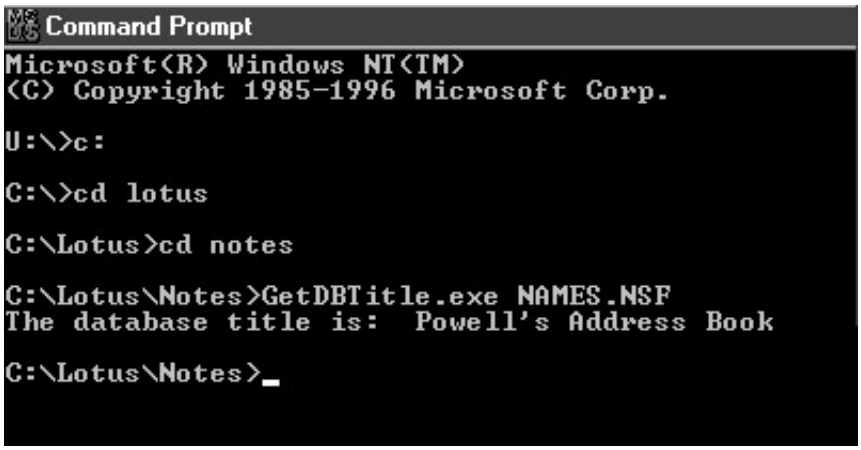


Then you will need to recheck your settings again. Any small changes will give you problems. If you still can not get a successful compile and link then you should check the toolkit documentation. As I said before, with a clean install of Visual C++ 6.0 and the API toolkit 2.2, this will work straight off.

Next, you need to test the program. What you need to do now is to copy the GetDBTitle.exe which you'll find in the C:\Program Files\Microsoft Visual Studio\My Projects\GetDBTitle\Release directory, and paste a copy into your Notes program directory i.e. C:\Notes (Do not drag and drop or windows will think you're trying to create a shortcut to the file).

The next thing you'll need to do is copy the C:\NOTESCPP\LIB\MSWIN32\ LCPPN22.DLL into your Notes program as well. You will need this file in the Notes directory for all C++ API programs you create.

To run the program open up a DOS Command prompt and move to the Notes directory, type in GetDBTitle.exe NAMES.NSF and hit return as in the following diagram:



Okay good, now you have a working program you can be assured that your environment is working and setup correctly.

CHAPTER 5

A Very Quick C++ Overview

The idea of this chapter is to familiarize you with the common aspects of C++ syntax such as variable declaration, looping, if then else statements etc. This will provide you with the basic tools and a quick reference to start coding in the C++ API straight away. It is not a definitive list by any means. There are plenty of free web sites offering C++ tutorials and many good books if you want to delve deeper into the depths of C++ but this is not the objective of this book.

In order to speed up your familiarity and understanding of the common C++ syntax I have created tables, in the first row you will see some C++ syntax and in the row directly underneath, you will find the corresponding ***similar*** LotusScript syntax. Notice I used the word 'similar' not identical.

Note: *C++ is case sensitive and LotusScript is not. So in C++ a variable declared as 'Total' can not be referred to as 'total' or 'TOTAL' as it can be in LotusScript.*

Note:*At the end of a line in C++ you append a semi colon. While not every line of code needs one, most do. A semi colon is appended to the end of a statement in C++.*

C++ Comments

```
// this is a comments on a line

/*
comments line 1
comments line 2
*/
```

Similar LotusScript

```
' this is a comment on a line

%rem
comments line 1
comments line 2
%end rem
```

C++ Operators

```
== Equal to
!= Not equal to
< Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to
|| Logical OR operator
```

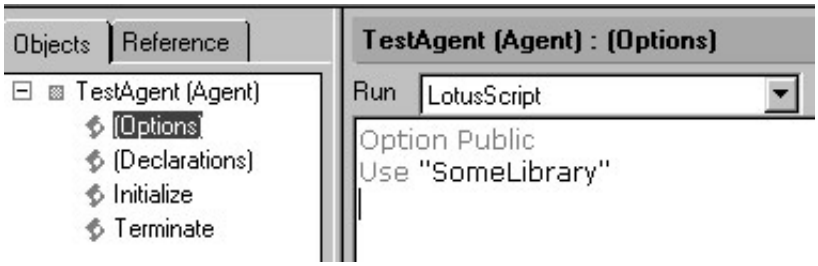
Similar LotusScript

= Equal to
<> Not equal to
< Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to
Or Logical OR operator

C++ Libraries

```
#include <someLibrary.h>
```

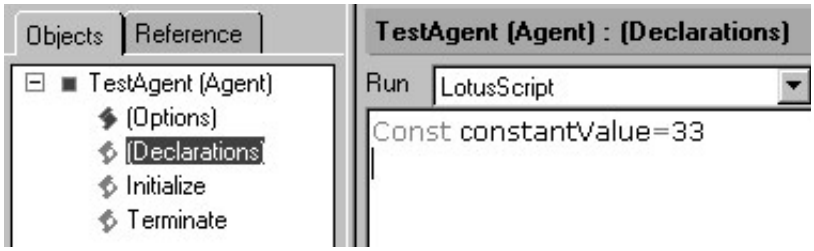
Similar LotusScript



C++ Constants

```
#define constantValue 33
```

Similar LotusScript



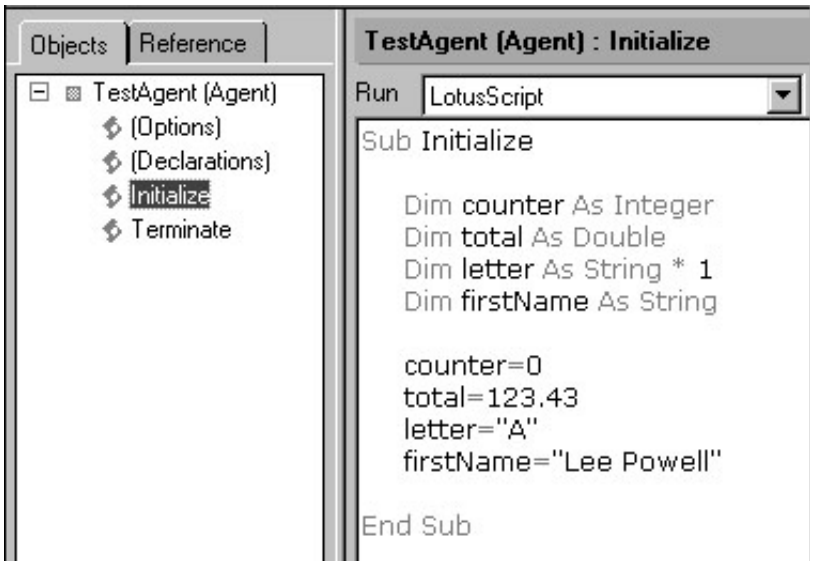
C++ Basic Data Types

Here are a few of the C++ basic data types - double forward slash // represents a comment in C++

```
int counter; //an integer
float total; //a floating point number i.e. a number with
             //a decimal point
char letter; //a character - any letter or symbol that can
             //be typed in at the keyboard
char * name; //a string(or array) of characters - unlike
//LotusScript, C++ has no concept of a String data type,
//strings are arrays of characters in C++. This will
//appear strange to begin with.

counter=0;
total=123.43;
letter='A';
name="Lee Powell";
```

Similar LotusScript

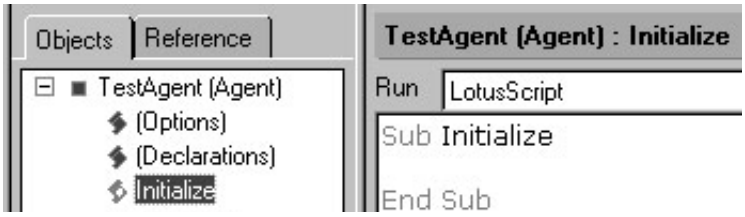
-i.e. vaguely similar data types in LotusScript:



C++ Main Function

```
main()
{

}
```

Similar LotusScript

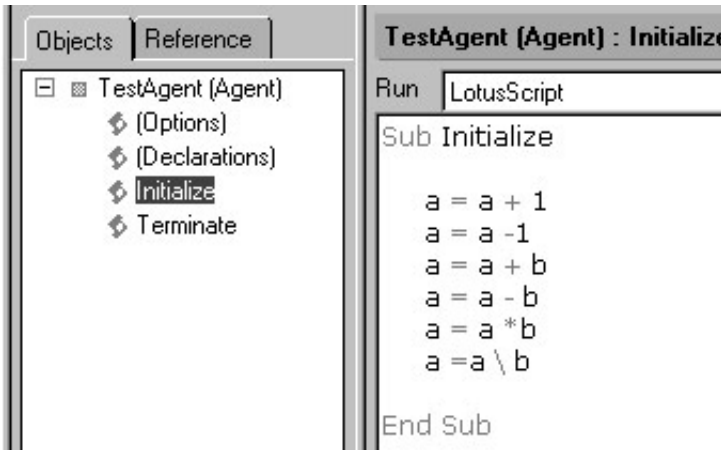


The initialize sub routine is similar to the main() routine in C++. In C++ the body of the program generally takes place within this function similar to the way in which the body of your LotusScript program takes place within the initialize routine. That is where the similarities end though. The main() routine can also be passed and return a variety of parameters whereas the Initialize routine can not mimic this.

C++ Shorthand

```
a++;
a—;
a += b
a -= b
a *= b
a /=b
```

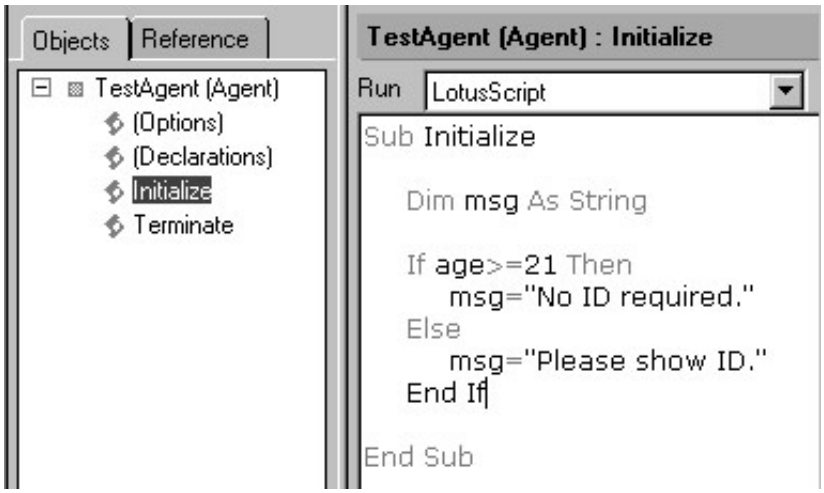
Similar LotusScript



The C++ IF ELSE

```
char * msg;

if(age>=21)
{
            msg="No ID required.";
}
else
{
            msg="Please show ID.";
}
```

Similar LotusScript



Looping in C++ the FOR Loop

```
int x;

for(x=0;x<10;x++)
{
            //do something in while loop
            //statement1;
            //statement2;
}
```

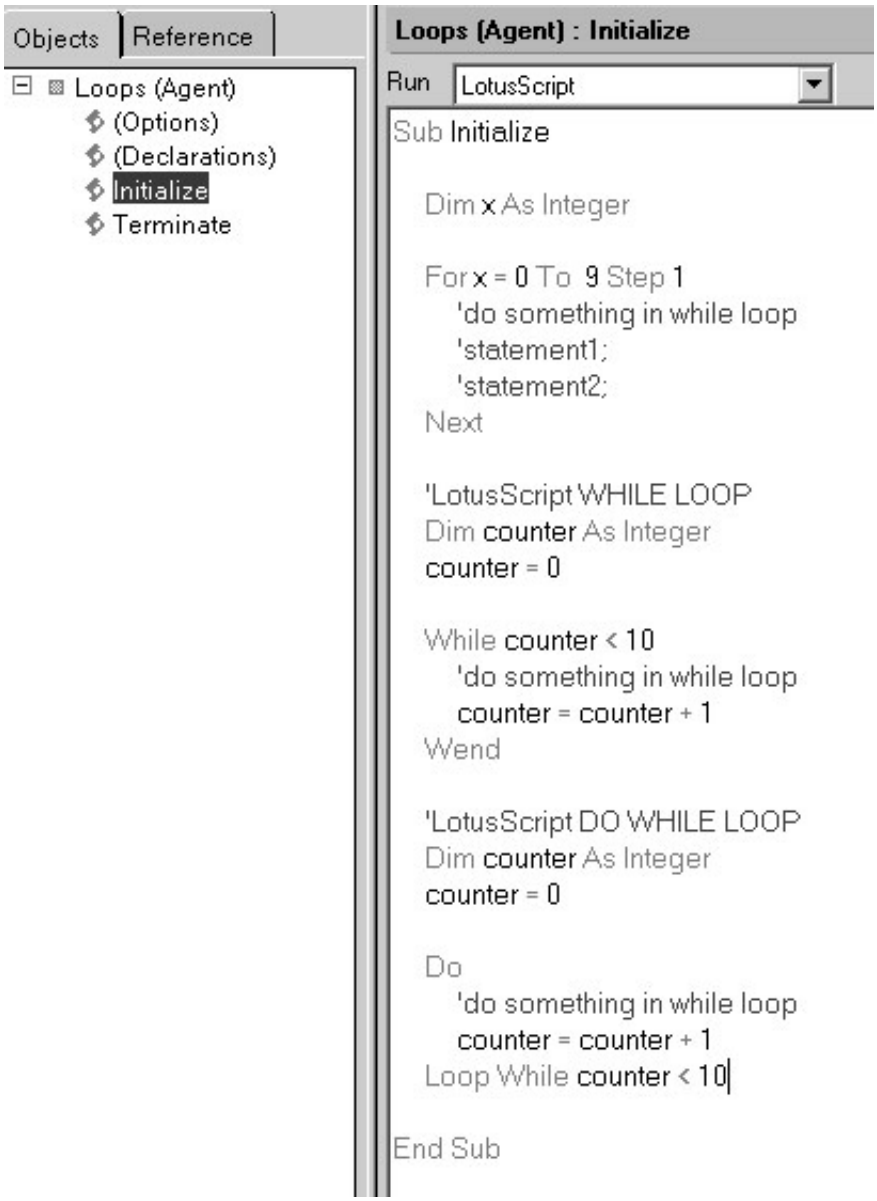
C++ WHILE Loop

```
int counter=0;

while (counter < 10)
{
            //do something in while loop
            counter ++;
}
```

C++ Do WHILE Loop

```
int counter=0;

do
{
            //do something in do while loop
            counter++;
}while (counter < 10);
```

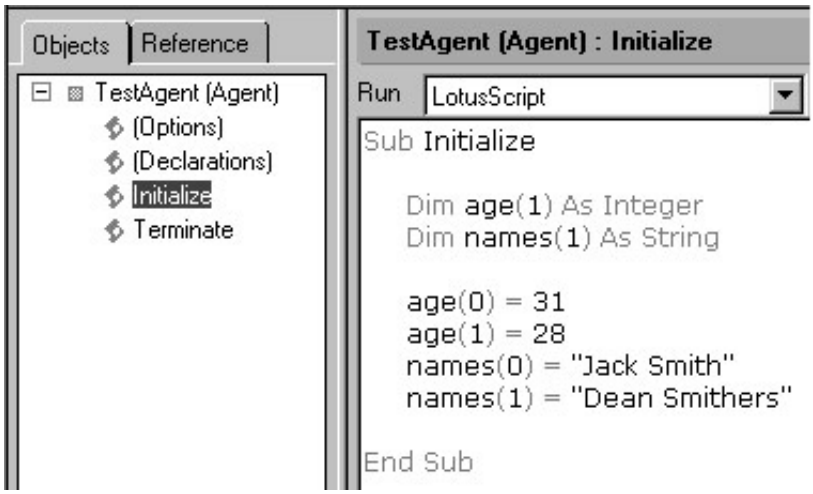
Similar LotusScript



C++ Basic Arrays

```
int age[2];
char * names[2];

age[1]=31;
age[2]=28;
names[1]="Jack Smith";
names[2]="Dean Smithers";
```

Similar LotusScript

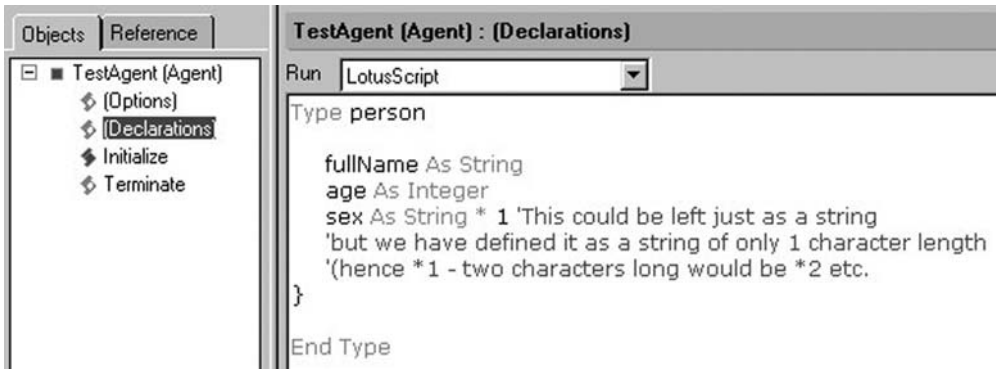


C++ Structures

```
struct person {
            char * fullName;
            int age;
            char sex;
}
```

Similar LotusScript

Structures are Types in LotusScript



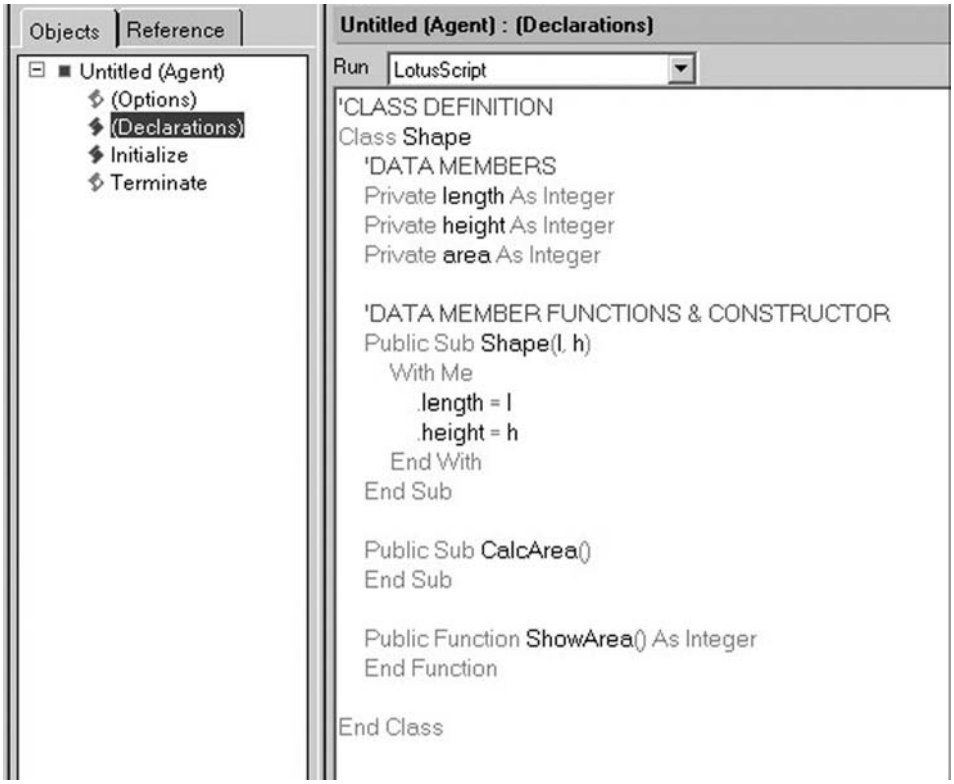
It should be noted here that there really is no equivalent for a LotusScript string data type in C++. When you dim something like: *dim something as string * 1* in LotusScript, even though you have stated the string is 1 character in length, LotusScript has allocated 4 bytes of memory for it, however if you defined a similar thing in C++ i.e. *char something* it would occupy 1 byte of memory for the same 1 character. The Notes C++ API does have an object called LNString, which we will look at later. For now do not worry too much about them, all will become clearer in the practical examples.

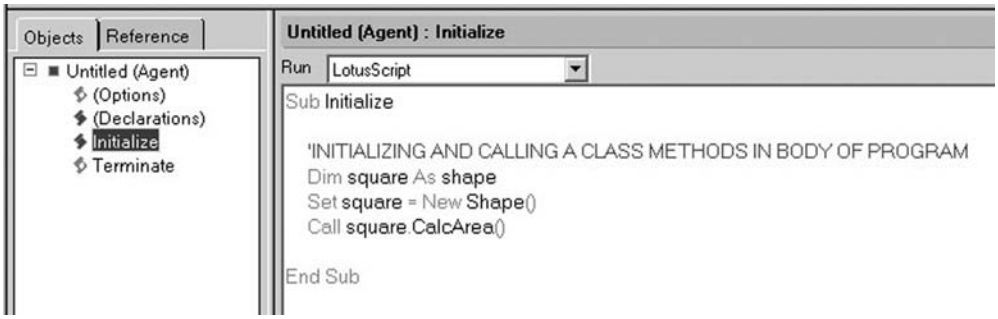
C++ Classes -

Do not worry about this if you do not ever use classes in LotusScript

```
//CLASS DEFINITION
class Shape {
            //DATA MEMBERS
            private:
               int length;
               int height;
               int area;

public:
            //DATA MEMBER FUNCTIONS & CONSTRUCTOR
               shape(int l=1, int h=1);
               void CalcArea(void);
               int ShowArea(void);
}

//INITIALIZING AND CALLING A CLASS METHODS IN BODY OF
PROGRAM
Shape square();
square.CalcArea();
```

Similar LotusScript





C++ CIN >> & COUT <<

The cin and cout commands in C++ are input and output commands and are contained in the iostream.h library, to use them you must include <iostream.h> library at the start of your program. These commands basically read data input from the keyboard and write data back to the command line respectively.

```
cout << "Enter your choice : " << endl;
//The previous line writes 'Enter your choice:' and a line
//return to the command line.

cin >> choice;
//The previous line reads characters from the keyboard
//into the variable 'choice' until the return key is
//pressed
```

Similar LotusScript

There is really no LotusScript comparison for these commands, I guess you could use inputbox and messagebox keywords, but these do not operate at the command line level, but perform a vaguely similar function.

Do not worry if you do not understand some bits in this chapter, things will become clearer as we progress.

CHAPTER 6

Creating Applications - The Bare Bones

The C++ API lets you create three kinds of programs:

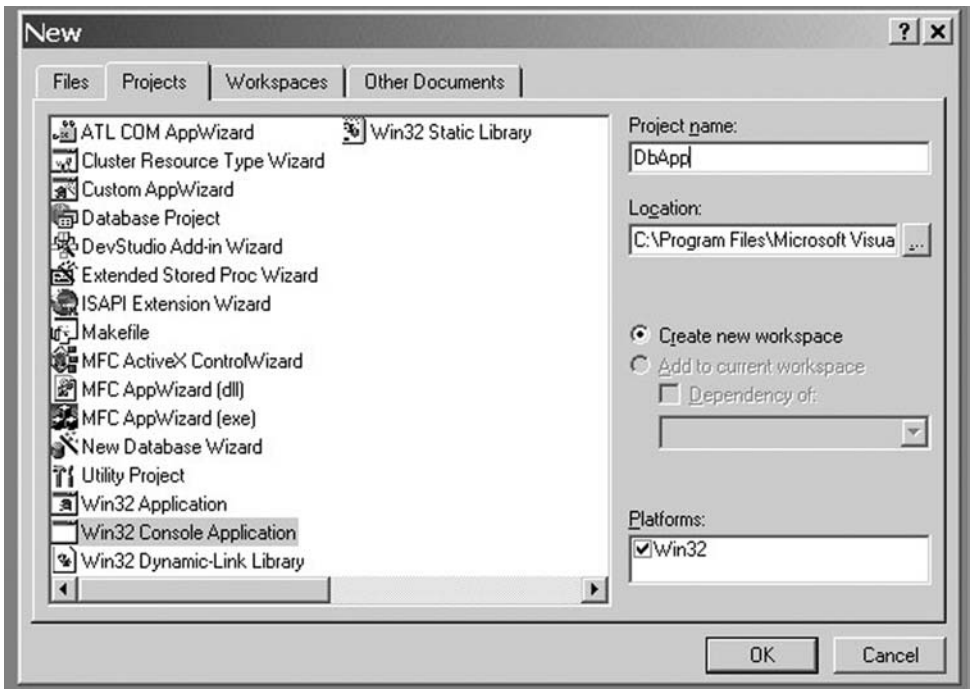
- Stand-alone applications.
- DLLS
- Server add-ins, custom Domino server tasks that extend and run alongside the standard server tasks.

In all these program formats the core API code is essentially the same. You use the same C++ API functions to operate on a Notes database. We will create programs in all three areas in the following pages, but first we must get right down to the basics of the API without all the confusing stuff. Once you get a handle on this, you will find that it is really not as complicated as many people believe.

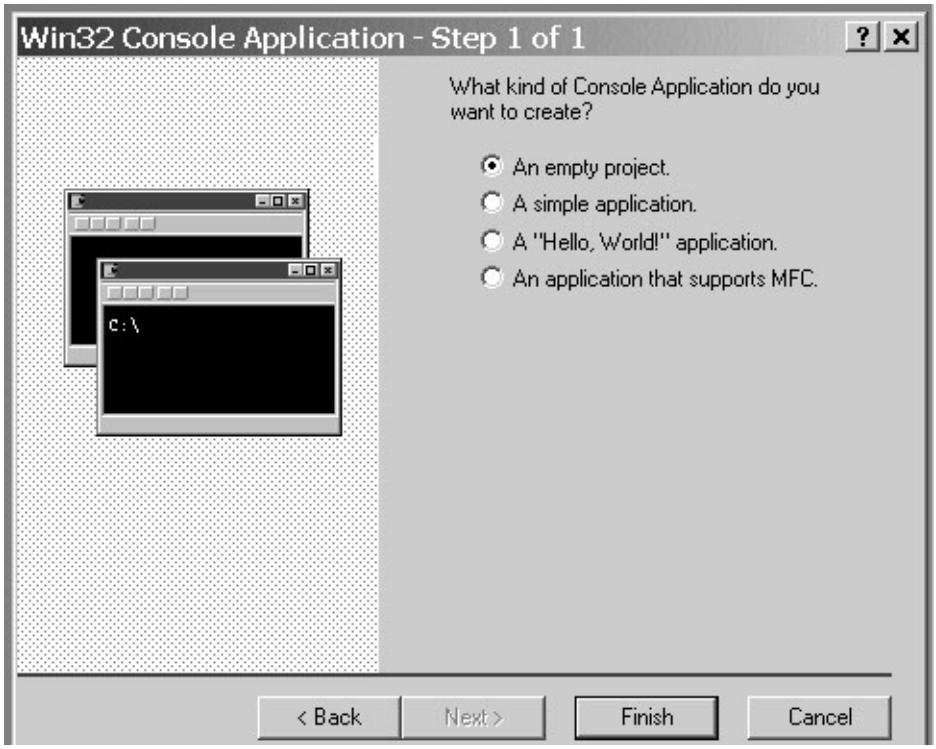
Let's get started. Start Microsoft's Visual C++. Select: File > New

We will first create a really simple console application (exe file).

Select Win32 Console Application and enter DbApp as the project name as in the following dialog:



Next click on the 'Finish' button then 'Ok' button



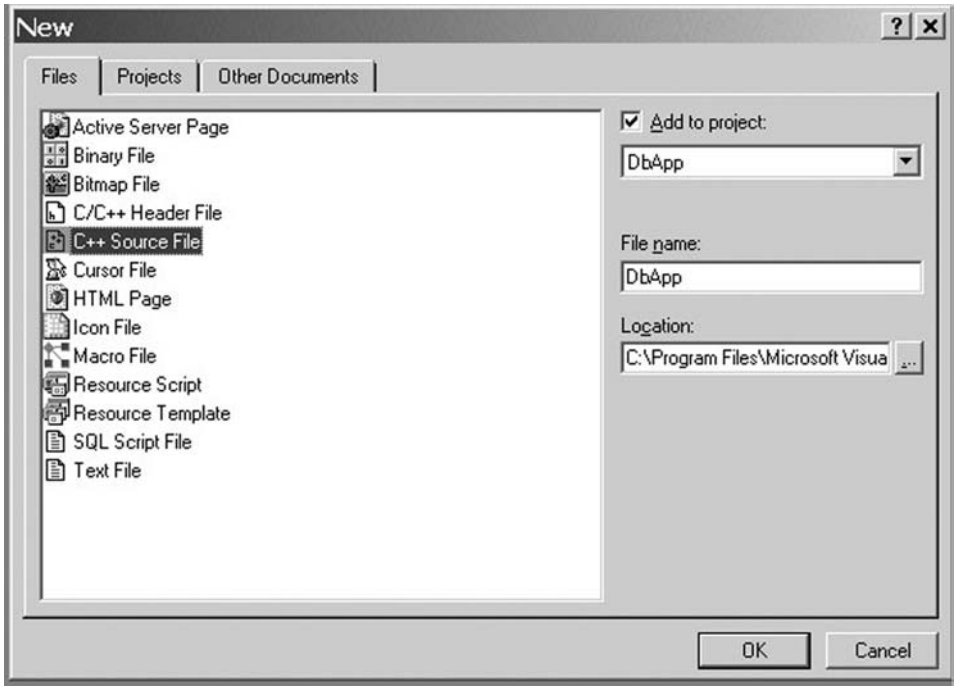
Next you will need to add the following file as we did in the section on 'Setting Up The C++ Environment':

Add the NOTESCPP.LIB file and ammend the following settings as we did before:

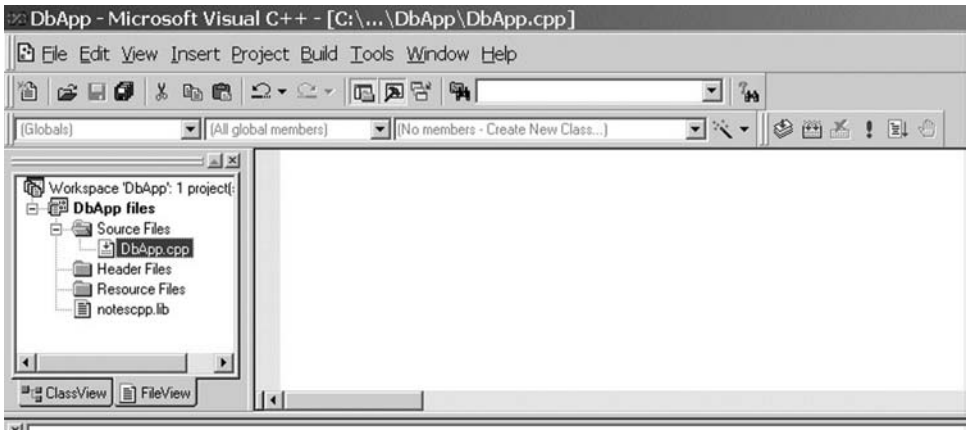
Build > Set Active Configuration > Win32 Release
Project > Settings - C/C++ tab - in General category - add W32 to list of preprocessor definitions.
In Code Generation category - set Struct Member Alignment to 1 byte.
In Preprocessor category - add C:\NOTESCPP\INCLUDE to additional directories

Next you need to add your very own blank C++ source file, so we can write our first bit of code.

To add our new blank source file to our project we need to select Project > Add to Project > New then select C++ Source File and enter a File name as 'DbApp' - if you want call it something else , then click OK



Your screen should now look like the following:

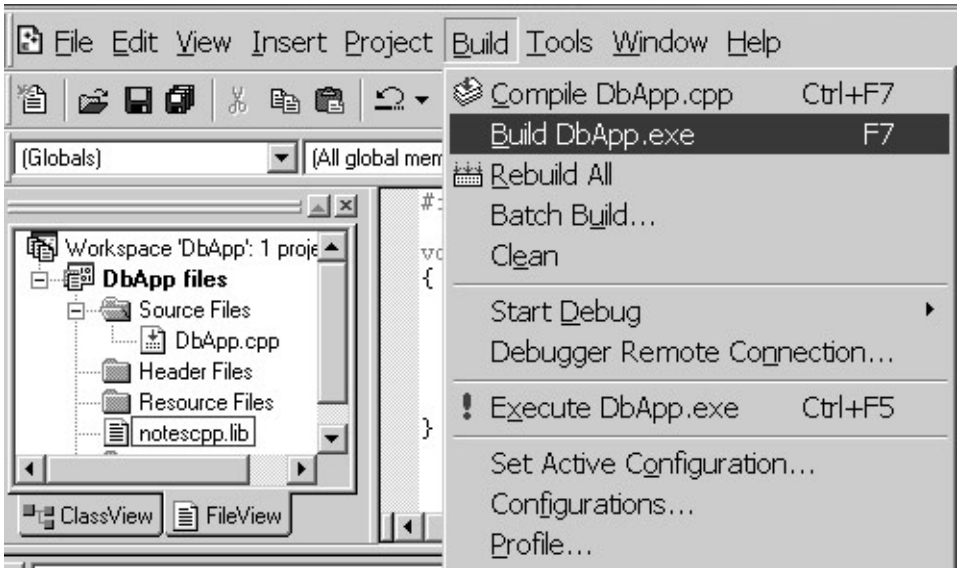


Notice I have the FileView Tab selected and I have expanded the Source Files folder to expose our new source file we added called DbApp.cpp. Double click on this file so we can edit it and begin to enter some code.

Enter the following code:

```
#include <lncppapi.h>

void main(int argc, char *argv[])
{
            LNNotesSession session;
            session.Init();
            session.Term();
}
```

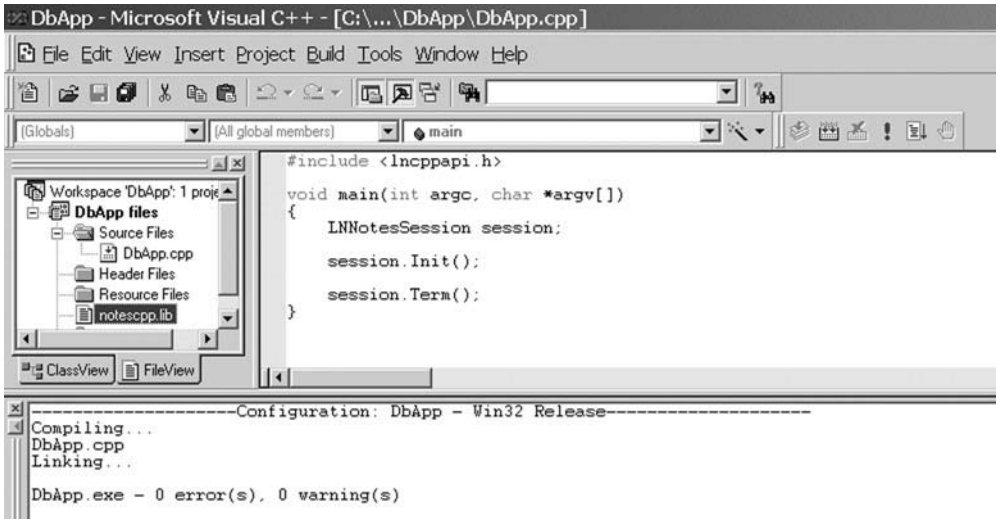
Once you have entered the code, select Build DbApp.exe from the Build menu to produce an exe file that you can then run from a DOS prompt



You should see a screen similar to the one below. If you have any compilation or linking errors, first check the code has been entered exactly as shown, remember C++ is case sensitive. If you still can't get it to compile, then one of the settings is not right. So go back and check the following again:

Make sure NOTESCPP.LIB file is present in you project, then check all the following settings, make sure you do number one before anything else as this will effect all the following settings:

1. Build > Set Active Configuration > Win32 Release
2. Project > Settings - C/C++ tab - in General category - add W32 to list of pre-processor definitions.
3. In Code Generation category - set Struct Member Alignment to 1 byte.
4. In Preprocessor category - add C:\NOTESCPP\INCLUDE to additional directories



Once you get the code to compile, congratulations you have written you first C++ API program. Whilst it does not do much other than start and end a single threaded session with Notes. It does serve as a great starting point, as you can see it's not all that hard if you are shown the basics. Before we spice up the code a little to make it do something usefull, let me just explain this simpe piece of code which you can use over and over again as a basic shell for your other work.

`#include <lncppapi.h>` - this line of code simply includes the Lotus Notes C++ API header file provided by Lotus into your code. This file lives in the C:\NOTESCPP\INCLUDE directory and basically links to other libraries that contain definitions for all the API's objects and data types such as the LNNotesSession class you referenced in you code.

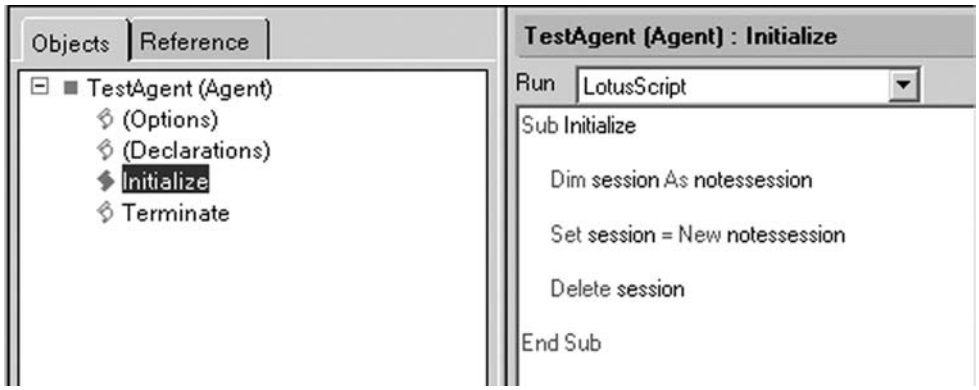
```
void main(int argc, char *argv[])
{
}
```

The previous bit of code represents the body of the C++ program. All C++ programs have a main function similar to an Initialize sub routine in LotusScript. I am not going to confuse you at this stage by explaining all the details, for now just except that the main function returns nothing because of the void keyword, and is passed two values an integer and a character array. I will explain this a little later on so do not worry about it.

Okay, more importantly, next we find the following three lines of code:

```
LNNotesSession session;

session.Init();

session.Term();
```

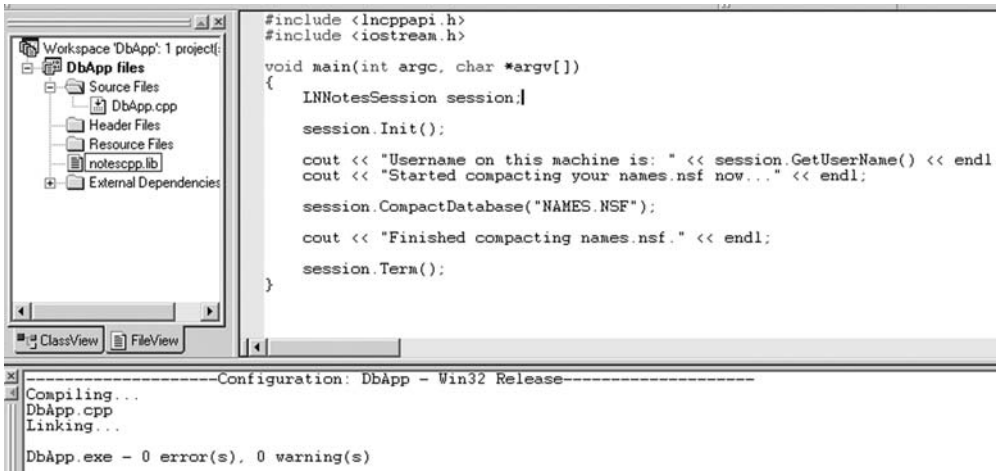
A similar example in LotusScript would look like the following:



The LNNotesSession class basically provides an application program with a connection to Notes.The **`Init()`** function establishes a Notes session for the API and initializes the C++ API. This function locates the Notes Data directory, reads the NOTES.INI file, and finds the user ID file. *All other Notes C++ API functions assume a Notes session is established successfully*. After you start a Notes session, you can make any API calls you like.

The **`Term()`** function frees any associated Notes resources for the session. It closes any open notes and databases, and deallocates memory allocated by the API. Any unsaved changes to notes or databases are lost.

That said let's add a few more lines of code between the Init() and Term() functions so that our program is a little bit more usefull. Type in the following code:



Notice I have added another library called **`<iostream.h>`**, this is the standard C++ input/output stream library that contains usefull functions like **`cout`** and **`cin`**, which output and input text respectively to the command line (DOS prompt in our case). It's not part of the Notes API it's simply one of many standard C/C++ library files.

I then make the following statements:

```
cout << "Username on this machine is: " <<
session.GetUserName() << endl;
```

This line basically outputs "Username on this machine is:" to the command prompt, then a session method called **`GetUserName()`** outputs the current Notes user name to the command line as well - even if Notes is not running at the time, then an end of line statement is written to the command line.

```
cout << "Started compacting your names.nsf now..." << endl;
```

The previous line simply writes the text to the command line and then writes a new line.

```
session.CompactDatabase("NAMES.NSF");
```

Next, another session method is called in the form of a compact database command on our personal name and address book. This performs a compact operation and then finally the last line of text is written to the command line in the form of:

```
cout << "Finished compacting names.nsf." << endl;
```

Then we terminate our session and our program closes.

Okay lets run the exe file now. To do this you will need to copy the exe file from your Visual Studio Release directory into the Notes program directory. On a standard install the exe file will be sitting in the following directory: C:\Program Files\Microsoft Visual Studio\MyProjects\DbApp\ Release\DbApp.exe.

Once you have copied this file into your Notes program directory, startup a command prompt (DOS) and run the exe

You will see that the API has been able find details about your current user id and compact the names.nsf database.

Well done our first application is complete.

CHAPTER 7

Error Handling

It's always a good idea to add some error handling in any of your code especially when using the API, failure to handle errors successfully could and often does result in your system crashing to some extent, if Notes is active on your machine you might even see the 'Red Screen of Death' before Notes crashes and you are forced to reboot your machine. Learning error handling now will save you time and prevent many headaches later. I am going to dive into some technical details about error handling in the next few paragraphs, do not worry if you do not understand it first off, it will become clearer as we proceed in our practical examples. It is really not difficult and worth reading several times to get a basic understanding.

Most C++ API functions return an LNSTATUS status code to indicate success or failure. LNSTATUS is a API data type, in fact it is a 32-bit status code. A non-zero value indicates an error or warning condition; zero (or LNNOERROR) indicates success. LNNOERROR is the LNSTATUS value that indicates success. LNNOERROR is guaranteed to be equal to zero.

API functions that don't return an LNSTATUS throw an LNSTATUS exception. To handle C++ exceptions, your program should use the C++ **`try`** and **`catch`** keywords, as illustrated below. If the API throws an LNSTATUS exception during execution of the try block, control transfers to the catch block.

```
try
{
// program statements
}

catch(LNSTATUS error)
{
//error-handling statements
}
```

Side Note: *Here's a code snippet of LotusScript doing in effect the same type of thing in a general sense as to what I'm trying to explain in the C++ API code snippet previously. In the LotusScript example we can see that we have set up an error handler to throw all errors to the errorHandler when an error of any kind occurs.*

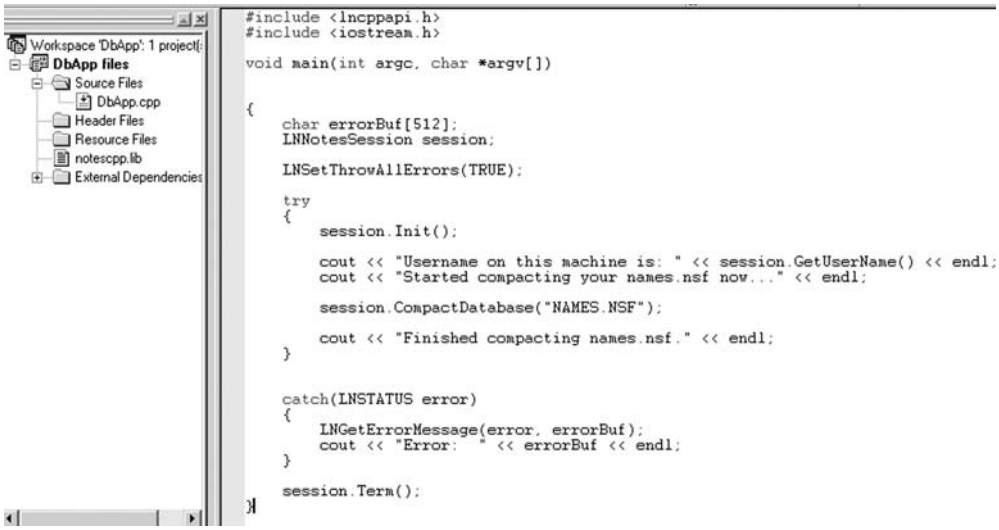
If an exception is thrown and your program doesn't provide a catch block, the program terminates immediately. It's good practice to provide at least one pair of try/catch blocks in your main function, so that if an exception is thrown, your program can clean up as needed before terminating.

Now don't freak out, this is all really rather simple, and once you get you head around it, you'll discover that the API provides some very granular error checking. However, because most API functions return an LNSTATUS value, your program can test the return value of each function to detect errors. As I've stated previously this provides your program with a lot of control over error handling, but it also makes the program more cumbersome to write. In many applications, you might prefer to handle API errors centrally, in one or more places in your program, rather than testing the return value of each function.

You can do this in the API by using the global LNSetThrowAllErrors function. The statement **`LNSetThrowAllErrors(TRUE)`** instructs the API to throw an LNSTATUS exception whenever an API error occurs anywhere in your program, even in API functions that normally return an LNSTATUS value. This means your program can use C++ try/catch blocks to handle all API errors, rather than testing return codes for individual functions. The ThrowAllErrors feature remains in effect until you use **`LNSetThrowAllErrors(FALSE)`** to disable it.

Each LNSTATUS error code has an associated error message as well. To display the appropriate message for a given error, you call the global function **`LNGetErrorMessage`**. To use this function you must pass it the error code. The function retrieves the error message in the buffer specified. If the message is longer than the buffer, LNGetErrorMessage truncates the string to fit into the buffer. The return value is the length of the error message.

Here is what our previous program would now look like if we incorporated this error checking functionality:



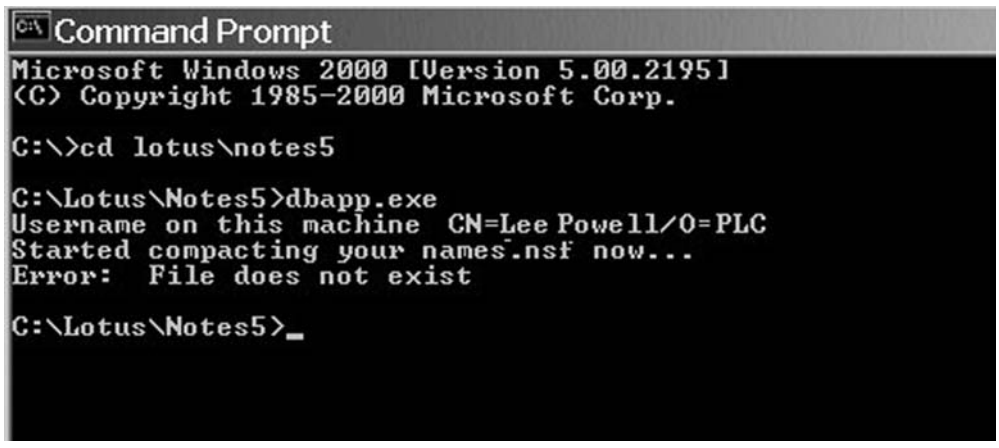
Ignore the warning when you rebuild the executable regarding the 'error' : unreferenced local variable, this is a safe warning. What you can do now to test your error handling works correctly is edit the C++ code, change the line:

```
session.CompactDatabase("NAMES.NSF");
```

to

```
session.CompactDatabase("NAMES.N45");
```

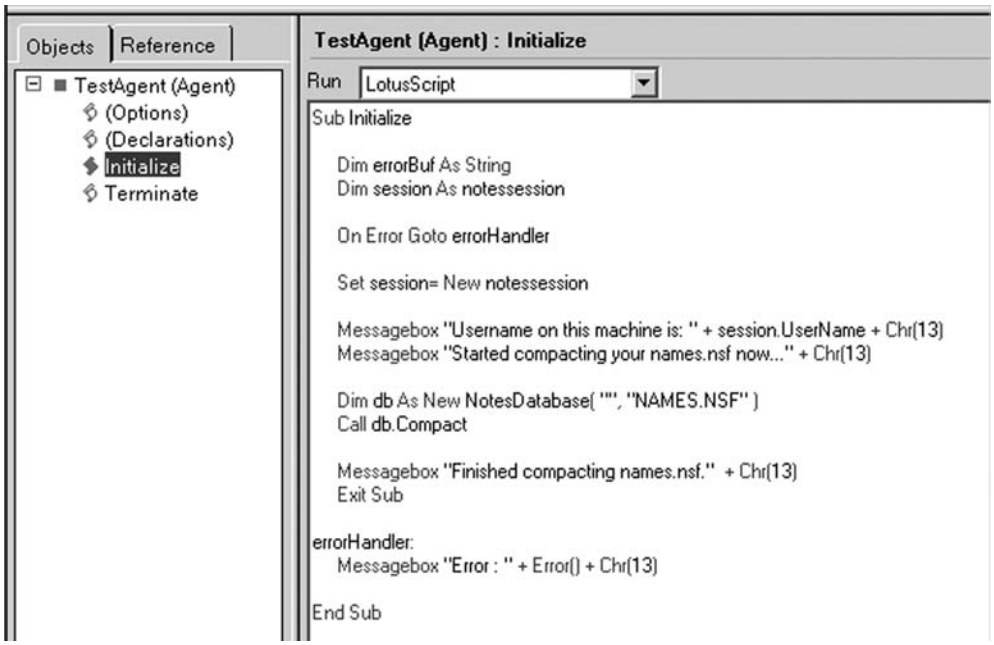
If you rebuild this code and copy the new exe file into your Notes program directory, open a Dos prompt and run the new exe you should see an error similar to the one below because the database NAMES.N45 does not exist forcing your code to error and be handled by your new catch block.



An error string of "Error: File does not exist" was output to the command line.

We have covered some basic error handling techniques here, what I'd like to do now is write pretty much the same sort of code in LotusScript just to help bring these new concepts back to something you are more familiar with. That way you can see for yourself that none of what we have covered is really all that different.

So here's the same kind of code in LotusScript:



In the next chapter we will write some API code that reaches parts of Notes that LotusScript can not, this is where the power and fun of the API really starts, it's also where your work colleges start looking at you as if your are some sort of programming genius.

CHAPTER 8

The DbWizard Application

We are going to create a new Win32 Console Application similar to what we did in the previous chapter when we created the DbApp. I'm not going to keep going over how to create a new C++ project and all the relevant Notes C++ API specific settings at this stage as I have already explained these in detail.

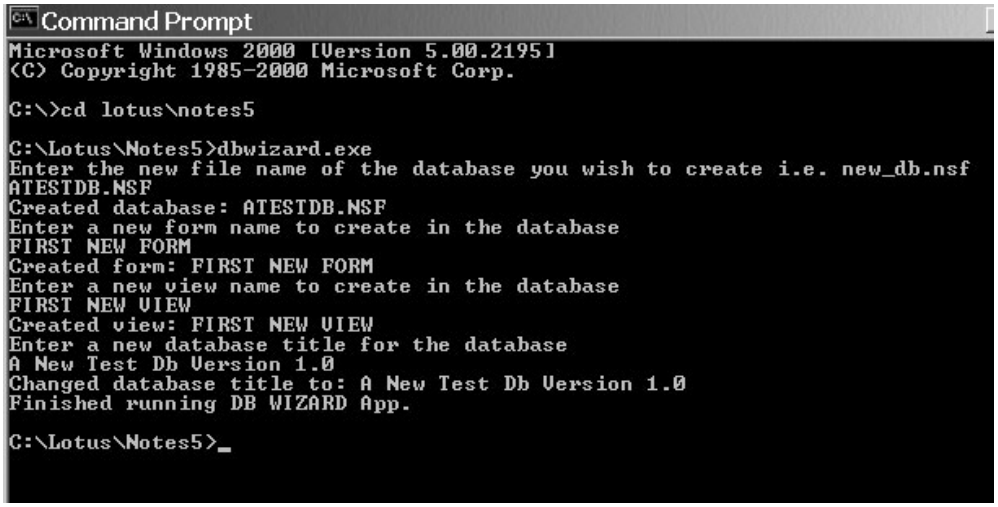
So let's get started!

Create a new project in Visual C++ called DbWizard, add all the relevant C++ API files and make all the respective setting changes. Now add to the project a new C++ source file and enter the following code shown in the screen shot below. I'll explain it in detail in just a moment. Once you have entered the code try and compile it, then copy the newly created DbWizard.exe file from C:\Program Files\Microsoft Visual Studio\MyProjects\DbWizard\Release into your Notes program directory.

```
#include <lncppapi.h>
#include <iostream.h>
#define MAX 100

int main( int argc, char *argv[] )
{
    char errorBuf[LNERROR_MESSAGE_LENGTH];
    LNNotesSession   session;
    LNSetThrowAllErrors(TRUE);

    try
    {
        session.Init();
        LNDatabase db;
        LNForm form;
        LNViewFolder view;
        char tempData[MAX];

        //Get database file name input
        cout << "Enter the new file name of the database you wish to create i.e. new_db.nsf" << endl;
        cin.getline(tempData,MAX);
        //Create the new database & return pointer to new database object(&db)
        session.CreateDatabase( tempData,"", &db);
        cout << "Created database: " << tempData << endl;

        db.Open(); //Open the actual database so we can work on it

        //Get form name input
        cout << "Enter a new form name to create in the database" << endl;
        cin.getline(tempData,MAX);
        //Create the new form in the database & return pointer to form object(&form)
        db.CreateForm(tempData, &form);
        form.Save();
        cout << "Created form: " << tempData << endl;

        //Get view name input
        cout << "Enter a new view name to create in the database" << endl;
        cin.getline(tempData,MAX);
        //Create the new view in the database & return pointer to view object(&view)
        db.CreateView(tempData, &view);
        view.Save();
        cout << "Created view: " << tempData << endl;

        //Get db title
        cout << "Enter a new database title for the database" << endl;
        cin.getline(tempData,MAX);
        db.SetTitle(tempData);
        cout << "Changed database title to: " << tempData << endl;

    }

    catch(LNSTATUS error)
    {
        LNGetErrorMessage(error, errorBuf);
        cout << "Error:   " << errorBuf << endl;
    }

    cout << "Finished running DB WIZARD App." << endl;
    session.Term();
    return(0);
}
```

Next, open a DOS prompt, and try running the DbWizard.exe file. If you are prompted to enter the password for the Notes ID on the machine - do so.

Here's what my output looked like:



What you will find in your Notes data directory is a new database with a file name, view, form and title with the details you entered at the DOS command line. If you had Notes up and running at the time, this time try closing Notes and execute the dbwizard.exe file again. What happens?

It runs right! Even though Notes was not opened.

Let's now take a look at the code in detail.

```
#include <lncppapi.h>
#include <iostream.h>
#define MAX 100
```

The first two lines of code simply incorporate two libraries into our program just as we would use the 'Use' statement in LotusScript to do a similar thing, in C++ we use '#include'. The first library we include is the Lotus Notes C++ API library file 'lncppapi.h', without this none of our Notes specific code will work. The second file as I mentioned in a previous chapter the 'iostream.h' library file is responsible for getting and sending information to the command line. In our code example we use two functions from this library: **`cin`** - which gets information from the command line and **`cout`** which sends information to the command line, both these functions make use of the extraction >> and insertion << operators, which basically do as they say insert and extract information to a stream.

Note: *There is plenty of information available regarding many of the common C++ libraries on the web, if you are really interested in finding out about these then simply go to a search engine and type in something like 'C++ AND <iostream.h>'.*

Okay, so we have worked out what the libraries are, next we see a constant definition for 'MAX' with a value of 100, basically we have defined an integer constant to the

value of 100 and we have done it globally to our application as it's appears outside our 'main() function. It's very important to remember where you declare your variables and objects in C++, similar to LotusScript if you declare something in the Declarations (comparable to defining something before the main() function in C++) section then everything else can see it by default in your program. If you declare a variable in a new sub routine called from your Initialize sub in LotusScript, do not expect your Initialize routine to be able to see the variables you declared in your new sub routine, like wise with C++, any variable declared within braces {} is available within the braces, not outside them - unless of course you declare your variables as public. A common error for most new time Notes API programmers is they define many variables in a try{} block, then try to use these same variables in an error catching catch{} block, but can't get at them because they have not been define globally.

Next, we have the main() function definition: **`int main( int argc, char *argv[] ),`** don't worry I will get around to explaining what argc and argv[], but I want to keep things really simple for now.

The next line of code is:

```
char errorBuf[LNERROR_MESSAGE_LENGTH];
```

This code defines a character array of 512 elements in length. I use the constant Notes API value of LNERROR_MESSAGE_LENGTH instead of 512 because you will see a lot of it in the example code and API documentation. So don't be scared it's harmless.

Next comes:

```
LNNotesSession    session;
```

We have seen this before so there's no surprises here, we are simply defining a notessession object similar to the way we would need to in LotusScript.

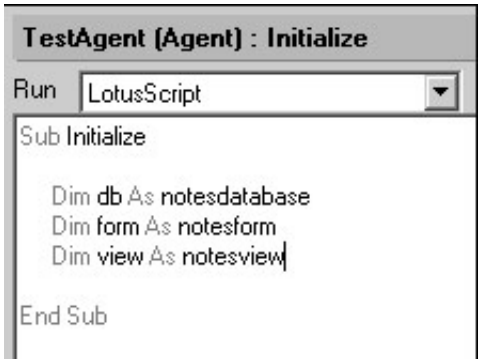
```
LNSetThrowAllErrors(TRUE);
```

Instructs the API to throw an LNSTATUS exception whenever an API error occurs.

Next, we enter our try { } block and find **`session.Init();`** - which initializes the Notes session object we previously defined.

```
LNDatabase db;
LNForm form;
LNViewFolder view;
```

The previous three lines of code define three Notes API objects: a Notes database, a Notes form and a Notes view, whilst not identical to LotusScript, it does not take much to see that they are in fact very similar and hence learning to harness the power of the Notes API is not entirely unfamiliar. Here's how I would generally define the same type of objects in LotusScript:



Next we find:

```
char tempData[MAX];
```

Now unlike LotusScript which has no comparable 'char' data type, we find a definition for a character array called tempData of 100 elements in size. This way of holding characters in a string is familiar territory to C++ folk, but to LotusScript folk it seems weird I know, but do not worry, the Notes API has a nice way of dealing with character arrays (strings) in a nice little object called LNString, we'll get to that in the next chapter.

```
cout << "Enter the new file name of the database you wish
to create i.e. new_db.nsf" << endl;
```

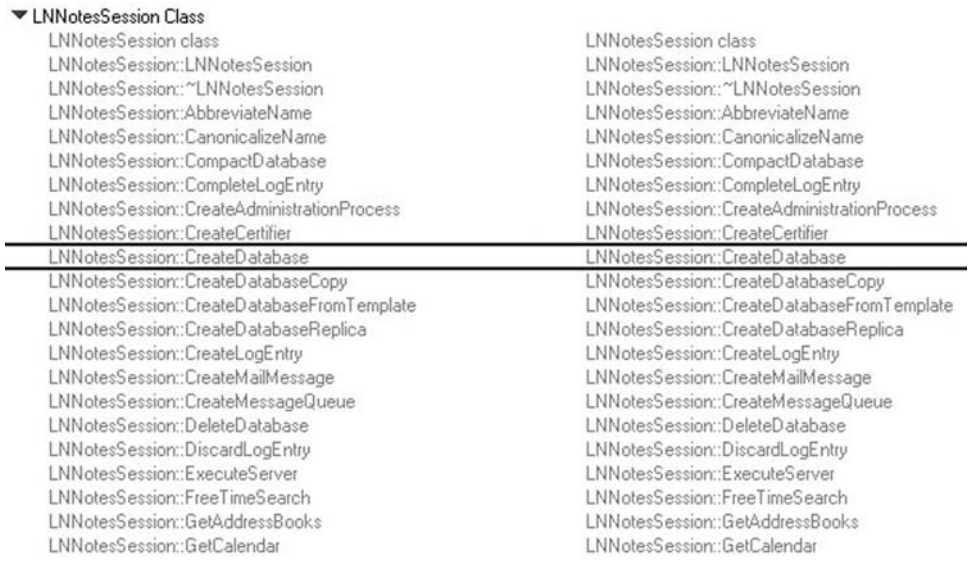
No prizes for this one, we simply send some text to the command prompt and move the prompt to the next line using 'endl' in effect a carriage return.

```
cin.getline(tempData,MAX);
```

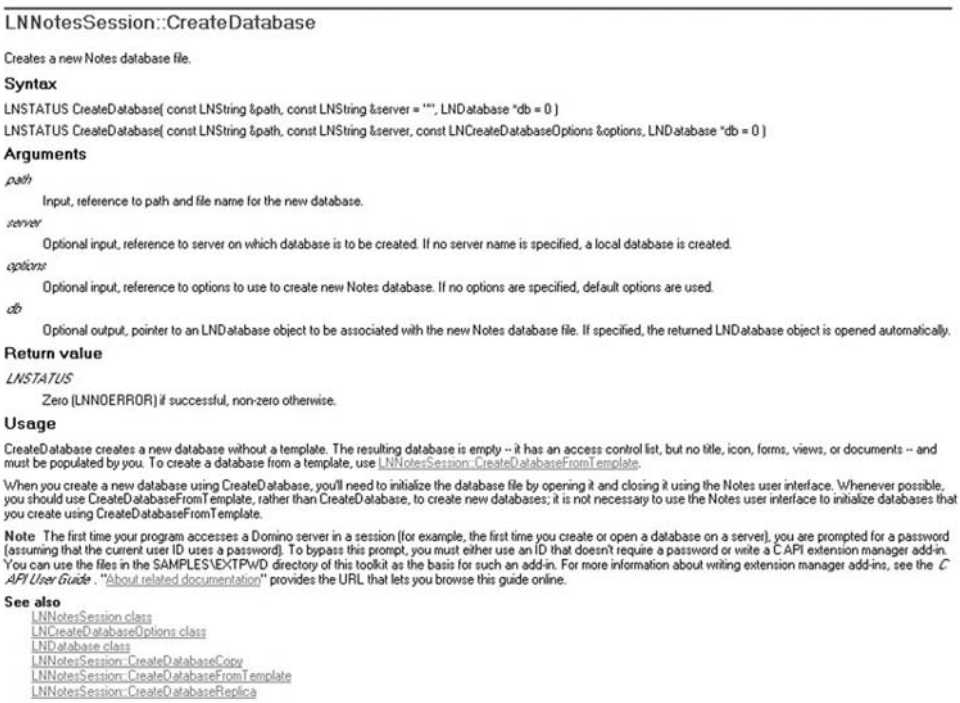
This next line retrieves up to 100 characters from the keyboard or until the return(enter) key is press signifying the end of a line. It reads these characters into the character array tempData. Both are defined in the iostream.h library which comes as part of Visual C++, this is not anything to do with the Notes API.

```
session.CreateDatabase( tempData,"", &db);
```

Having initialized the session object previously we can now use it to access it's methods and properties in exactly the same dot notation fashion we use in LotusScript. If you open up the 'Lotus C++ API Release 2.1 Guide' and expand the Reference Guide and expand the LNNotesSession Class you will see a list of all the methods you can use in the API for the session object - like wise for all other objects. Lotus have done a good job of listing out all the different objects in the Reference Guide. Once you get used to navigating your way through the Reference Guide, you will really take off in your ability to leverage from the API. You will quickly be able to discover if the C++ API is going to be able to do a specific task someone has requested that Notes formula and LotusScript can't do. My advice is get used to looking through it to see how all the objects fit together and interact with each other - you'll find it's not too different to LotusScript, just far more comprehensive and granular.



When you open the document LNNotesSession.CreateDatabase, here's what you'll see:



LNNotesSession::CreateDatabase

Creates a new Notes database file.

Syntax

LNSTATUS CreateDatabase(const LNString &path, const LNString &server = "", LNDatabase *db = 0)

LNSTATUS CreateDatabase(const LNString &path, const LNString &server, const LNCreateDatabaseOptions &options, LNDatabase *db = 0)

Arguments

path

Input, reference to path and file name for the new database.

server

Optional input, reference to server on which database is to be created. If no server name is specified, a local database is created.

options

Optional input, reference to options to use to create new Notes database. If no options are specified, default options are used.

db

Optional output, pointer to an LNDatabase object to be associated with the new Notes database file. If specified, the returned LNDatabase object is opened automatically.

Return value

LNSTATUS

Zero (LNNOERROR) if successful, non-zero otherwise.

Usage

CreateDatabase creates a new database without a template. The resulting database is empty -- it has an access control list, but no title, icon, forms, views, or documents -- and must be populated by you. To create a database from a template, use LNNotesSession::CreateDatabaseFromTemplate.

When you create a new database using CreateDatabase, you'll need to initialize the database file by opening it and closing it using the Notes user interface. Whenever possible, you should use CreateDatabaseFromTemplate, rather than CreateDatabase, to create new databases; it is not necessary to use the Notes user interface to initialize databases that you create using CreateDatabaseFromTemplate.

Note The first time your program accesses a Domino server in a session (for example, the first time you create or open a database on a server), you are prompted for a password (assuming that the current user ID uses a password). To bypass this prompt, you must either use an ID that doesn't require a password or write a C API extension manager add-in. You can use the files in the SAMPLES\EXTPWD directory of this toolkit as the basis for such an add-in. For more information about writing extension manager add-ins, see the *C API User Guide*. "About related documentation" provides the URL that lets you browse this guide online.

See also

LNNotesSession class
LNCreateDatabaseOptions class
LNDatabase class
LNNotesSession::CreateDatabaseCopy
LNNotesSession::CreateDatabaseFromTemplate
LNNotesSession::CreateDatabaseReplica

This is another thing you must get used to looking at, although at first it can appear a little strange. You can see several different definitions of the same CreateDatabase method depending on what parameters you pass it. We have used the first version. We know from looking at this definition that CreateDatabase returns a LNSTATUS, which we know from our error checking chapter is a 32 bit value and that anything

other than a zero (LNNOERROR) represents an error. The CreateDatabase method expects a constant path and server LNString value to be passed in to create the database and returns an optional pointer to the created database.

The first thing you will notice is that we did not pass in an LNString value, we simply passed in a literal string in the form of a character array for the database file name. The LNString class represents a text string. You can think of an LNString as an array of characters. Many functions in the C++ API take LNStrings as arguments or return them. Wherever an LNString argument is expected, you can also specify a literal string. So why use LNString you say?

Well for a start most API objects that return a string value will always return it as an LNString, so you'll be forced to use it anyway. However it does have some rather nice in built functions for manipulating the strings. So lookup LNString in the reference guide for a list of all the functions.

You can see in our code:

```
session.CreateDatabase( tempData,"", &db);
```

I passed in literal strings for the filename and server parameters of the CreateDatabase constructor method. Then there this funny looking **'&db'**, all this means is that we are passing the address in memory for the existing **db** object we created earlier with the line: **LNDatabase db;**. The function **CreateDatabase** will simply use the address of the **db** object in memory and initialize it for you.

We now have a session object and a db object and can call their respective methods. If I am confusing you by saying method one sentence and function in another then I mean the same thing. The reality is that methods are simply functions within a class, and when a class in instantiated(initialized) it becomes an object. Functions tend to be functions that exist in normal code outside of classes and objects - but they really are the same thing.

```
cout << "Created database: " << tempData << endl;
```

Outputs text and the value stored in the tempData to the command line.

```
db.Open();
```

Because we returned a pointer to the database object 'db' when we created our new database we now can call upon the database classes methods. Once again go into the reference guide and take a look at the long list of available methods for the LNDatabase class. We do not really need to open the database here as it is already opened when we created it, but because when you work with databases generally you'll need to open them first, I put it in here so you become familiar with it.

```
cout << "Enter a new form name to create in the database"
<< endl;
cin.getline(tempData,MAX);
db.CreateForm(tempData, &form);
form.Save();
```

Next we prompt the user for a new form name and read in their input from the command line, then we call another LNDatabase class method for 'db' called CreateForm, which creates a new Notes form in the database and returns a pointer to the new '&form' object so we can work with it further i.e. to add a title or text, tables, fields, hotspots, sections, formulas etc. We next call the form.Save() method, to save the form in the database, if we don't call this then our design changes will be lost in exactly the same way if we create a new document in LotusScript and fail to save the new document, our document is lost.

Note: *If you are looking for the Save() method in the LNForm class and are scratching your head because you can't see it, then take a look at the C++ Object Architecture Appendix and you'll see that this LNForm class inherits like many other objects from the LNNote class, which has a Save() method, which is inherited by LNForm class. If you struggle with this then think about the NotesRichTextItem class and the NotesItem class it inherits from in LotusScript. When you create a NotesRichTextItem you can still use the methods of NotesItem because NotesRichTextItem inherits from NotesItem and thus has access to all it's methods and properties. This works in the same way.*

```
cout << "Created form: " << tempData << endl;
cout << "Enter a new view name to create in the database"
<< endl;
cin.getline(tempData,MAX);
```

The next lines of code outputs the name of the created form stored in the tempData character array to the command line, followed by a prompt to enter a view name to create in the database, the view name entered at the command line is stored in the tempData character array. No suprises here.

```
db.CreateView(tempData, &view);
view.Save();
cout << "Created view: " << tempData << endl;
cout << "Enter a new database title for the database" <<
endl;
cin.getline(tempData,MAX);
```

Next, we use another LNDatabase class method of our 'db' object to create a new view in the database with a title set to the value previously stored in the tempData character array that we prompted the user for from the command line. You'll once again notice we have returned a pointer to the new view object using '&view'. Now we have access to a new view object. You'll also notice that the view object was defined as part of the LNViewFolder class, this essentially is very similar to the NotesView class we are accustomed to in LotusScript. If you have a look at the LNViewFolder class in the toolkit reference you'll be surprised at the amount of methods that are available to you like creating view columns, selection formulas, backgrounds etc. Just think of the dynamic reporting possibilities where you could potentially create a view on the fly based on some other data. After the view has been created, we need to save it. Once again you'll notice the Save() method is part of the LNNote class which the LNViewFolder class inherits from. Hopefully now you'll see where Notes gets it's name Notes - as most objects are considered Notes as they inherit from the LNNote class.

```
db.SetTitle(tempData);
cout << "Changed database title to: " << tempData << endl;
}
catch(LNSTATUS error)
{
            LNGetErrorMessage(error, errorBuf);
            cout << "Error:  " << errorBuf << endl;
}

cout << "Finished running DB WIZARD App." << endl;
session.Term();
return(0);
```

The next call is once again using a method of the LNDatabase class object 'db' SetTitle, which get's passed the character array containing the new database title. This sets the database title for the database.

Then we simply write this title back to the command line, and that ends our try{} block. We then move onto the catch{} block for handling errors which I'm not going to go over again. Finally we terminate the session object which closes down all other objects and return(0) back from the main() routine, which terminates the application.

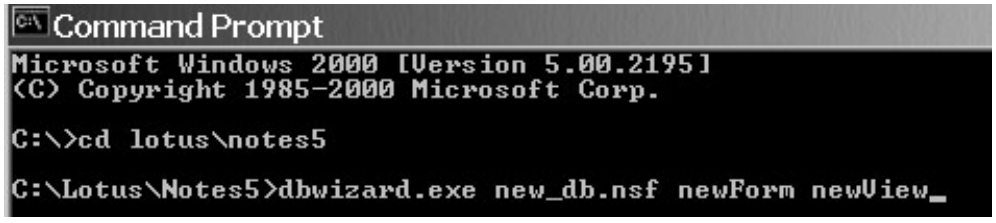
And that's it.

CHAPTER 9

Main Program Parameters Explained

Now I will explained the parameters that are passed into the *main(int argc, char * argv[])* function for the curious. In the good old days of command line programming using DOS and UNIX, this method was common place. The parameters passed through to the main program function are of a certain format that must be strictly adhered to. Following on from our previous example where we created a command line application that created a new database, a newform, a new view and new title. We could take advantage of the parameters in the main function and code the application in a slightly different way.

In this example we will pass on the command line the file name for the database, the new form and view name respectively in one swift blow. To do this at the Dos prompt we will probably enter something like the following:



Now what happens in the **`main(int argc, char * argv[])`** function when you press the return key is that the integer value for argc is set automatically to 4 because there are four parameters being passed i.e

1=dbwizard.exe
2=new_db.nsf
3=newForm
4=newView

As you can see where ever there was a space in the command line, argc expect a new parameter to follow.

What happens in argv[] is that it automatically gets set as an array with four elements of string(character arrays). Thus if we were to look at the argv array it would now look like this:

argv[0]=”dbwizard.exe”
argv[1]=”new_db.nsf”
argv[2]=”newForm”
argv[3]=”newView”

Because these parameters are passed into the main() function for us automatically we can hence reference them anywhere in our main function. It's really that simple.

Take a look at the following code, which essentially is the dbwizard application, with most of the comments ripped out and instead of continually prompting the user for values using cout and cin, we simply look at the command line initial input for everything.

```
#include <lncppapi.h>
#include <iostream.h>

int main( int argc, char *argv[] )
{
    char errorBuf[LNERROR_MESSAGE_LENGTH];
    LNNotesSession   session;
    LNSetThrowAllErrors(TRUE);

    try
    {
        session.Init();
        LNDatabase db;
        LNForm form;
        LNViewFolder view;

        // Check the correct values were entered at the command line, no more no less
        if (argc < 4 || argc > 4)
        {
            cout << "Invalid parrameters." << endl;
            cout << "Enter as follows: dbwizard <filename> <formname> <viewname>" << endl;
            return (1);
        }

        session.CreateDatabase( argv[1],"", &db);
        db.CreateForm(argv[2], &form);
        form.Save();
        db.CreateView(argv[3], &view);
        view.Save();
    }

    catch(LNSTATUS error)
    {
        LNGetErrorMessage(error, errorBuf);
        cout << "Error:  " << errorBuf << endl;
    }

    cout << "DB WIZARD created the following" << endl;
    cout << "Database: " << argv[1] << endl;
    cout << "Form: " << argv[2] << endl;
    cout << "View:" << argv[3] << endl;
    session.Term();
    return(0);
}
```

This code should all make sense to you now as we have gone over it before.

One thing I will point out is the following:

```
if (argc < 4 || argc > 4)
{
    cout << "Invalid parrameters." << endl;
    cout << "Enter as follows: dbwizard <filename>
    <formname> <viewname>" << endl;
    return (1);
}
```

What you see here is a check that no less than and no more than 4 parameters are being passed. If less than or greater than four parameters are passed we send some text to the command line explaining how to enter a valid command line sequence. We then return a 1 to the calling application and exit the main() function.

Try and run this application, enter your password if prompted at the command line. Be careful about your spaces, try not to put spaces in your form and view names, then again try it anyway and see what happens.

Note: *A more realistic command line entry that allowed for spaces, would be to use a single cin.getline command and retrieve an entire line delimited by a character and then split the string up in C++ and put the chuncks into variables.*

CHAPTER 10

A Glimpse At What is Possible with the DbWizard App

We have looked at some pretty basic code thus far that really does not do a whole lot. More importantly however, you have been able to get familiar with the API without sinking out of your depth - this has been my goal thus far. Hopefully you can see now that the basic structure of an API program is pretty straightforward.

I now want to show you a piece of code that will give you a better glimpse at some of the powerful features of the API. I will comment on the code briefly afterwards, so you can see what is going on. In this example for simplicity sake, I do not pass in any command line arguments but have hard coded many values. It is not important that you understand every line, this will come soon enough. I do however, want to excite you a little, before we move away from creating applications and move onto creating dlls. Which in my opinion are a whole lot more fun as we can call dlls directly from LotusScript and receive values back from them for further processing.

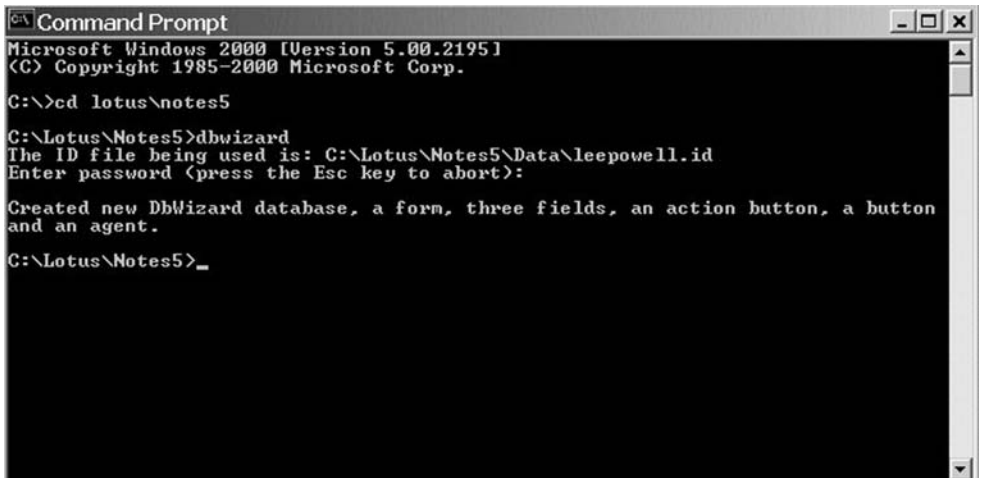
Here's our final DbWizard application code, just read through it first slowly, I have purposely named variables and objects in a way that should make it easier for you:

```
#include <lncppapi.h>
#include <iostream.h>

int main( int argc, char *argv[] )
{
       char errorBuf[LNERROR_MESSAGE_LENGTH];
       LNNotesSession   session;
       LNSetThrowAllErrors(TRUE);

       try
       {
              session.Init();
              LNDatabase db;
              LNForm form;
              LNViewFolder view;
              LNFormField newfield;
              LNRichText rt;
              LNRTCursor cursor;
              LNString buttonFormulaString("@Prompt([OK];\"Your user
name\";@Name([CN];@UserName))");
              LNString textFieldFormulaString("\"www.notesapi.com\"");
              LNFormula buttonFormula(buttonFormulaString);
              LNFormula textFieldFormula(textFieldFormulaString);

              session.CreateDatabase( "dbwizard.nsf","", &db);
              db.CreateForm("TestFormOne", &form);
              form.SetBackgroundColor(LNCOLOR_RED);
              form.GetFormBody(&rt);
              rt.GetCursor(&cursor);
              rt.Insert("Text Field One:  ", &cursor);
              rt.CreateFormField("TextFieldOne", LNRTTYPE_TEXT_FIELD,
&cursor, &newfield);
              newfield.SetValueFormula(textFieldFormula);
              rt.StartNewParagraph( &cursor);
              rt.Insert("Number Field One:  " , &cursor);
              rt.CreateFormField("NumberFieldOne", LNRTTYPE_NUMBER_FIELD,
&cursor);
              rt.StartNewParagraph( &cursor);
              rt.Insert("Rich Text Field:  ",  &cursor);
```

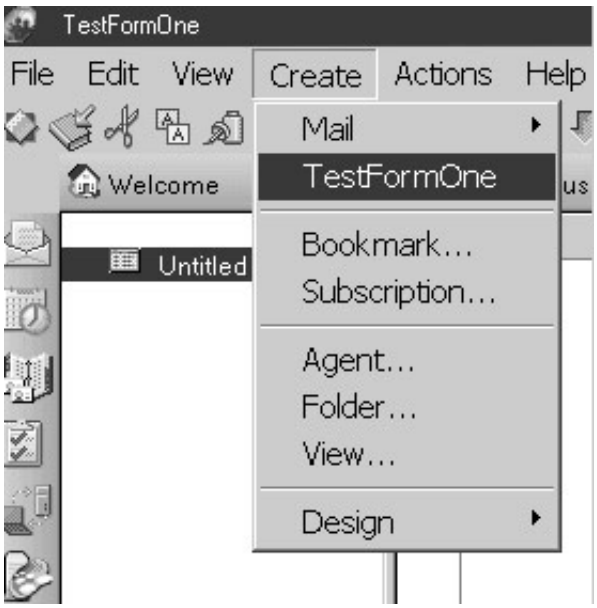
```
            rt.CreateFormField("BodyField", LNRTTYPE_RICH_TEXT_FIELD,
&cursor);
            rt.StartNewParagraph( &cursor);
            rt.CreateButton("Display User Name", buttonFormula,
&cursor);

            //before we save the forms lets create an action button to
display at the top of the form
            LNActions actionButtonsArray;
            LNAction actionButton ("Show User Name Action Button",
buttonFormula);
            form.GetActions(&actionButtonsArray);
            actionButton.SetIconIndex (1);
            actionButton.SetShowInButtonBar (TRUE);
            actionButton.SetShowInMenu (TRUE);
            actionButtonsArray.Append(actionButton);
            form.Save();

            //we still need to create a single view or the db will not
open in the Notes client without one.
            db.CreateView("Untitled", &view);
            view.Save();

            //let's create a basic agent in the database as well
            LNAgent newAgent;
            db.CreateAgent("Display User Name", TRUE, &newAgent);
            newAgent.SetFormula(buttonFormula);
            newAgent.Save();
            cout << "Created new DbWizard database, a form, three
fields, an action button, a button and an agent." << endl;
      }

      catch(LNSTATUS error)
      {
            LNGetErrorMessage(error, errorBuf);
            cout << "Error:  " << errorBuf << endl;
      }


      session.Term();
```

```
        return(0);
}
```

Create a new project in Visual C++ or amend your previous version of the DbWizard application and enter and compile the new DbWizard code. Copy the executable into your Notes program directory and delete any previously created versions of the 'DbWizard.nsf' database if there is one in your Notes data directory. Then, open a DOS prompt and run the new DbWizard.exe. Enter a password if needed. Your output should look something like this in DOS:



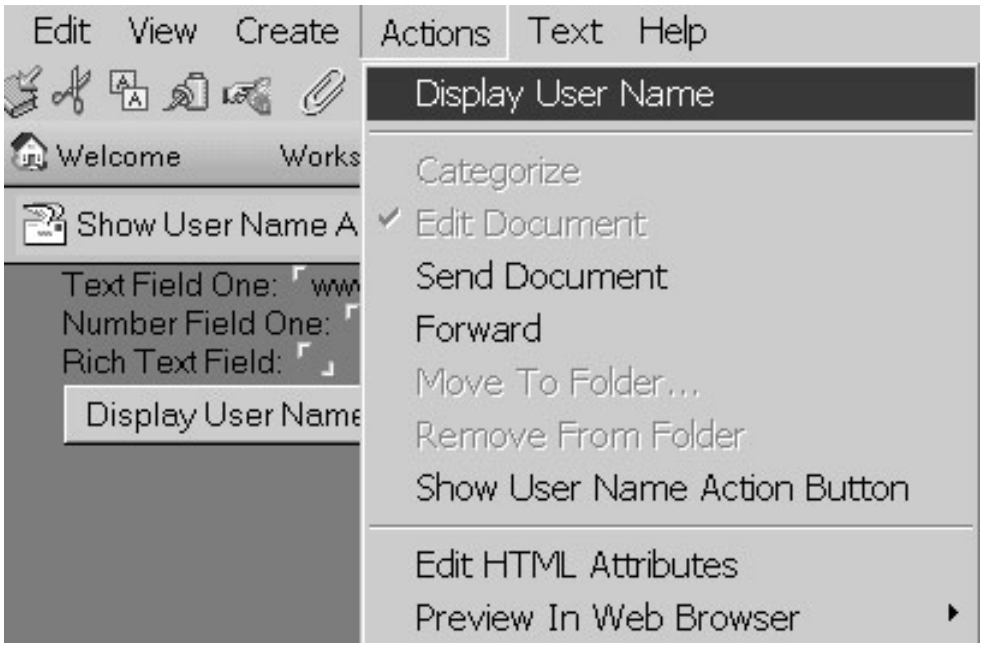
You will notice that a new DbWizard database was created in your Notes data directory and if you open it you will be able to compose a form called 'TestFormOne'. From the Notes 'Create' menu compose this form.



You will see a disgusting red colored form with an action button titled 'Show User Name Action Button', three fields one text, one number and one rich text in value with the corresponding labels. You will also see a button titled 'Display User Name'. Try

clicking on this button to see your user name displayed. Notice also our text field has a default value of 'www.notesapi.com'.

You will also notice that from the Notes 'Action' menu we have a new agent available titled 'Display User Name', try running this.



Powerful huh! We have created a brand new database from scratch and created some useful design elements in a few lines of API code. Imagine the possibilities of this for a moment, being able to potentially create databases designs on the fly. Imagine you created a Standard Notes database with a whole bunch of different documents in it that define various generic forms, views, fields etc. You could then check which forms, views and fields you wanted in the database and click a build button and voila! A newly created database has been made courtesy of you API code based on what you or a non technical user had selected in the original database. Takes a little imagination I know, but is definitely possible. Maybe you'll write an app that does this and make a mint.

Okay, let's go through the code now. The first thirteen lines of code need no explanation as we have covered them several times before. The following three lines of code we have not seen before:

```
LNFormField newfield;
LNRichText rt;
LNRTCursor cursor;
```

The LNFormField, LNRichText and LNRTCursor classes and all their methods and properties are all well defined in the Toolkit Reference guide so take a brief look if you have time. Basically the 'newField' object represents a field record in a Notes database, it is the object you would use for all form fields. You could draw a vague similarity to a NotesItem object in Lotusscript.

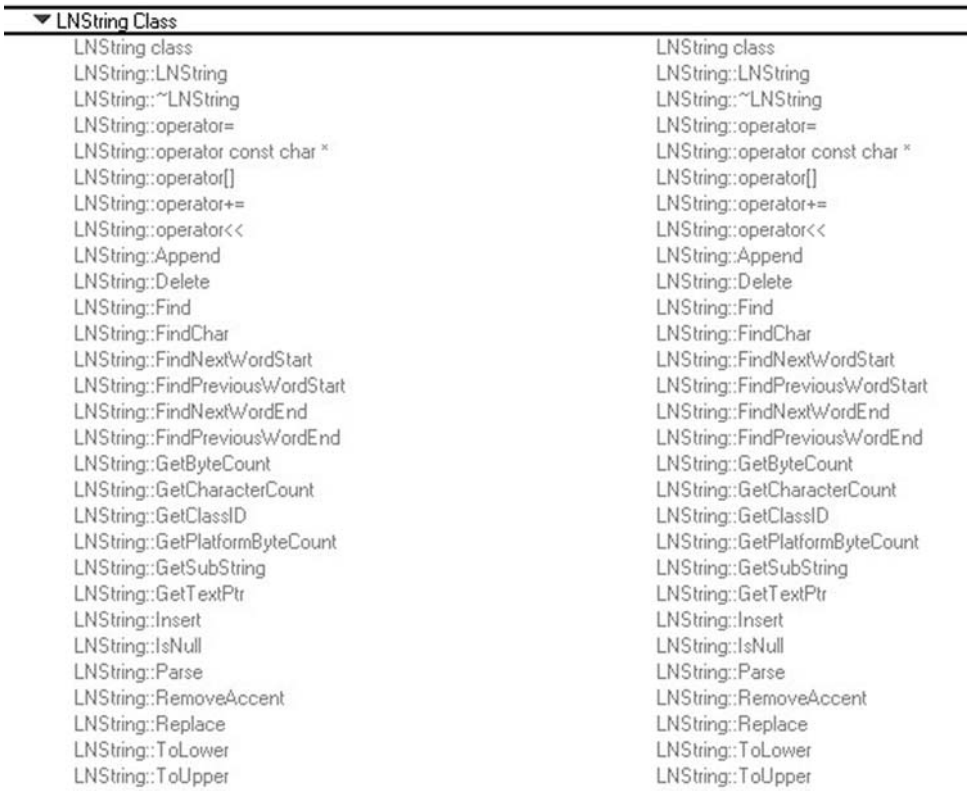
LNRichText represents a rich text item in a note. A rich text item can contain stylized text and a variety of different objects, including file attachments, OLE objects, bitmaps, tables, form fields and subforms, doclinks and more. There is a good object hierarchy chart in the 'More Info on the Rich Text Object' chapter in this book, this will show you exactly what the LNRichText Object encompasses.

To navigate rich text, we use the LNRTCursor object. An LNRTCursor loosely resembles the cursor in the Notes editor. In order to position elements such as fields and buttons on our newly created Notes form, we use the LNRTCursor object to move between elements in the rich text, so that we can insert our new elements exactly where we want them.

Although you could draw a similarity between the API's LNRichText object and Lotusscript NotesRichTextItem, be aware the API's LNRichText object is a lot more powerful and far more granular. For one thing it is not limited to a rich text field in the same way that NotesRichTextItem is, as you can see in the code the LNRichText object can contain one or many rich text fields as well as many other things.

```
LNString buttonFormulaString("@Prompt([OK];\"Your user
name\";@Name([CN];@UserName))");
LNString textFieldFormulaString("\"www.notesapi.com\"");
LNFormula buttonFormula(buttonFormulaString);
LNFormula textFieldFormula(textFieldFormulaString);
```

The next four lines basically define two formulas we will use in our form design. The LNString class represents a LMBCS (Lotus Multi Byte Character Set) text string. Which is essentially just an array of characters. In most cases you can simply just pass in your standard literal string (i.e "a string like this"). Take a look at LNString in the in the Toolkit reference guide:



Here you'll see many useful functions for manipulating and parsing strings, and this will save you from having to use other C++ libraries to manipulate literal strings. So my advice is to use it.

In the code you will notice that we created two LNstring objects and passed them literal strings that will represent formulas for our new form. You will also notice that the first one is simply an @Prompt written in standard formula language with backslashes before any internal double quotes, so that the API knows that these are double quotes within double quotes and not the end of the literal string.

The LNFormula object we have not seen before. This object represents a Notes formula, as used in views, forms, agents/macros, and so on. To get or set a formula in a database you will need to use this object. You may be interested to know that there are similar objects for LotusScript, JavaScript and some classes to work with Java in the toolkit.

All we are doing in the last two lines of code is passing the two LNFormula objects two LNStrings, which represent two formulas we want to insert into our form, one @Prompt and the other just a literal string that we'll use for a default field value formula.

```
session.CreateDatabase( "dbwizard.nsf","", &db);
db.CreateForm("TestFormOne", &form);
form.SetBackgroundColor(LNCOLOR_RED);
form.GetFormBody(&rt);
rt.GetCursor(&cursor);
rt.Insert("Text Field One:  ", &cursor);
```

The first two lines of code need no further explanation, as we already know how to create a database and form and return a pointer to the newly created form. The **`form.SetBackgroundColor(LNCOLOR_RED);`** function speaks for itself and is simply a method of the LNForm class. Here's a list of other colors you could use as they are defined in the Toolkit:

LNCOLOR_BLACK
LNCOLOR_WHITE
LNCOLOR_RED
LNCOLOR_GREEN
LNCOLOR_BLUE
LNCOLOR_MAGENTA
LNCOLOR_YELLOW
LNCOLOR_CYAN
LNCOLOR_DARK_RED
LNCOLOR_DARK_GREEN
LNCOLOR_DARK_BLUE
LNCOLOR_DARK_MAGENTA
LNCOLOR_DARK_YELLOW
LNCOLOR_DARK_CYAN
LNCOLOR_GRAY
LNCOLOR_LIGHT_GRAY

LNCOLOR represents a color value, which is an integer from 0-239. Colors 0-15 have symbolic names, so if you want other colors you can specify an integer between 0-239.

The next line of code **`form.GetFormBody(&rt);`** This returns a pointer to the form's $BODY item as an LNRichText object. We need to do this because basically the static data and objects in the form are positioned and formatted in the $Body item to control how the documents created with the form will look when they are first created. We use this rich text item to access and manipulate all facets of the form design, similar to how you would use the NotesRichTextItem in Lotusscript to format content in a rich text field, we do the same thing here, but for the entire form.

Once we have a handle to the rich text item of the form we can call upon the LNRichText functions to manipulate the form. In order to locate our exact position in the Rich Text we use a very handy LNCursor object, now this may seem a little weird at first, but once you get used to it, it really does make life very easy. This cursor is like a cursor in a word processor, you can't see it flashing on the screen, but it works in the same way, it basically stores your position in the rich text. If you insert some text

or a button or whatever else, the cursor will automatically move to the end of the object you inserted, allowing you to position it at the next insert point. When you first call the **`rt.GetCursor(&cursor);`** method, the cursor is positioned before the first element in the rich text, in a similar way to a word processor, when first opened, would position the cursor in the top left hand corner of the document. If you are interested in finding out how to navigate rich text and the composition of rich text into CD records then the Toolkit user guide has some good articles addressing this topic.

We know that the cursor object is positioned before the first element in the rich text area, so we call our LNRichText method **`rt.Insert("Text Field One:", &cursor);`**, which inserts the text 'Text Field One' to the top left hand corner of the new form. We pass in the cursor reference, so that the API knows exactly at what cursor position to insert the text. After inserting the text into the form's Rich text object the *cursor* is adjusted to point to the cursor location after the inserted text.

```
rt.CreateFormField("TextFieldOne", LNRTTYPE_TEXT_FIELD,
&cursor, &newfield);
newfield.SetValueFormula(textFieldFormulaString);
rt.StartNewParagraph( &cursor);
rt.Insert("Number Field One:  " , &cursor);
rt.CreateFormField("NumberFieldOne", LNRTTYPE_NUMBER_FIELD,
&cursor);
rt.StartNewParagraph( &cursor);
rt.Insert("Rich Text Field:  ",  &cursor);
rt.CreateFormField("BodyField", LNRTTYPE_RICH_TEXT_FIELD,
&cursor);
rt.StartNewParagraph( &cursor);
rt.CreateButton("Display User Name", buttonFormula,
&cursor);
```

Next, we call another rich text function **`rt.CreateFormField("TextFieldOne", LNRTTYPE_TEXT_FIELD, &cursor, &newfield);`**. This function creates a new field in the rich text on the new form. The name of the field is passed in as the first value 'TextFieldOne', we then define the field type as a text field at the cursor position we passed in and the final argument to the function returns a pointer to the newly created field, which we will access in a moment to insert a default value formula. The API has the following other available field types which you can specify instead of just text, these will be very familiar to you:

LNRTTYPE_FORM_FIELD
LNRTTYPE_TEXT_FIELD
LNRTTYPE_TIME_FIELD
LNRTTYPE_NUMBER_FIELD
LNRTTYPE_KEYWORDS_FIELD
LNRTTYPE_RICH_TEXT_FIELD
LNRTTYPE_NAMES_FIELD
LNRTTYPE_AUTHORS_FIELD
LNRTTYPE_READERS_FIELD

Now that we have a pointer to the newly created field we can now call upon the newField objects functions to set attributes for the field. The next line **`newfield.SetValueFormula(textFieldFormula);`** sets the default value formula for the newly created text field. Here you can see we pass in the LNFormula object we created earlier, this LNFormula object contains the text string "www.notesapi.com", which will be the default value for this new field.

The next line **`rt.StartNewParagraph( &cursor);`**, starts a new paragraph in the rich text at the specified cursor position, which now after creating the new text field will appear immediately after the created field. By calling this function we now move the cursor to the beginning of a new paragraph.

Next, we insert another text label at the beginning of the new paragraph in the rich text and then create a number field. We again call the new paragraph function and insert another text label and rich text field, we again call the new paragraph function.

The next line of code **`rt.CreateButton("Display User Name", button Formula, &cursor);`** creates a button with a title of 'Display User Name', we pass it another LNFormula object, this time one that contains the @Prompt formula we specified earlier, and once again we create the button at the location of the cursor we specified.

```
LNActions actionButtonsArray;
LNAction actionButton ("Show User Name Action Button",
buttonFormula);
form.GetActions(&actionButtonsArray);
actionButton.SetIconIndex (1);
actionButton.SetShowInButtonBar (TRUE);
actionButton.SetShowInMenu (TRUE);
actionButtonsArray.Append(actionButton);
form.Save();
```

Next, I wanted to add something a little more interesting by adding an action button to the form. The LNActions class represents a collection of action objects stored in a design note (a form, view, subform, page, or shared actions note). It provides access to the actions themselves, and also provides functions that let you control the appearance of the action bar. It's basically an array of action objects that you reference by specifying a subscript i.e. action[1], action[2]...action[n].

Firstly, we must set up an object to represent this array of actions and we do this by declaring: **`LNActions actionButtonsArray;`**. Now we have an object capable of holding all the actions in our form.

In the line **`form.GetActions(&actionButtonsArray);`**, you can see clearly that we are calling the form object's function to retrieve whatever actions the form contains currently, which is none, and store them in the LNActions object we just created. In the previous line to this

```
LNAction actionButton ("Show User Name Action Button",
buttonFormula);
```

we created our own new action titled 'Show User Name Action Button' and we passed the action our infamous @Prompt formula we used previously in our button. However, because all individual action buttons are stored in an array in the form, we need to define which index position the action button should appear in, and if the action button should be visible on the actions bar and actions menu. We do this in the following lines of code:

```
actionButton.SetIconIndex (1);
actionButton.SetShowInButtonBar (TRUE);
actionButton.SetShowInMenu (TRUE);
```

If we were to do the same thing in Notes Designer it would look like the following:

The last thing we need to do is append the action button we just created to the **actionButtonsArray** object that represents all the actions on our new form. The way we do this is by calling a member function of the LNActions class **actionButtonsArray. Append(actionButton);**

Now we have created our form with some text labels, fields, a button and an action button. To avoid loosing all our hard work we must save the form by calling form.Save();.

```
db.CreateView("Untitled", &view);
view.Save();
```

These two lines of code need no explanation as we have covered them before. It is worth noting though that if no view exists in your database, then you will not be able to open it with the Notes client, an error will appear saying the database is not fully

initialized. So make sure you create at least one view when creating databases from scratch using the API.

```
LNAgent newAgent;
db.CreateAgent("Display User Name", TRUE, &newAgent);
newAgent.SetFormula(buttonFormula);
newAgent.Save();
```

The last piece of code in the application before we more into the standard error 'catch' block creates a new agent in the database titled 'Display User Name'. I pass this our @Prompt formula we used in our button and action button objects. To create an agent in the database we must create a LNAgent object, we do this by declaring:

`LNAgent newAgent;` Next, we call one of the LNDatabase functions called **`CreateAgent("Display User Name", TRUE, &newAgent);`**, in this case we pass in the title for the agent as the first parameter, TRUE for the second to allow the agent to be shared and then return a pointer to the newly created agent object called newAgent. Now we can call on functions of the LNAgent object, such as SetFormula, which we use in the next line to set the formula for the agent. We then must save the agent by calling the **`Save();`** function.

For your interest here is a partial list of other objects you can create using the methods of LNDatabase as defined in the Toolkit reference guide:

LNDatabase::CreateAboutDatabaseDocument
LNDatabase::CreateAgent
LNDatabase::CreateApplet
LNDatabase::CreateDocument
LNDatabase::CreateEncryptedNote
LNDatabase::CreateFolder
LNDatabase::CreateForm
LNDatabase::CreateFrameset
LNDatabase::CreateFTIndex
LNDatabase::CreateIcon
LNDatabase::CreateImage
LNDatabase::CreateNavigator
LNDatabase::CreateNote
LNDatabase::CreateOutline
LNDatabase::CreatePage
LNDatabase::CreateScriptLibrary
LNDatabase::CreateSharedActions
LNDatabase::CreateSharedField
LNDatabase::CreateSubform
LNDatabase::CreateUsingDatabaseDocument
LNDatabase::CreateView
LNDatabase::CreateViewFolder

Now that was not too bad was it? If you have stayed with me thus far, give yourself a pat on the back. You should by now be realizing the potential power of the API. We are now going to move onto writing some dll's.

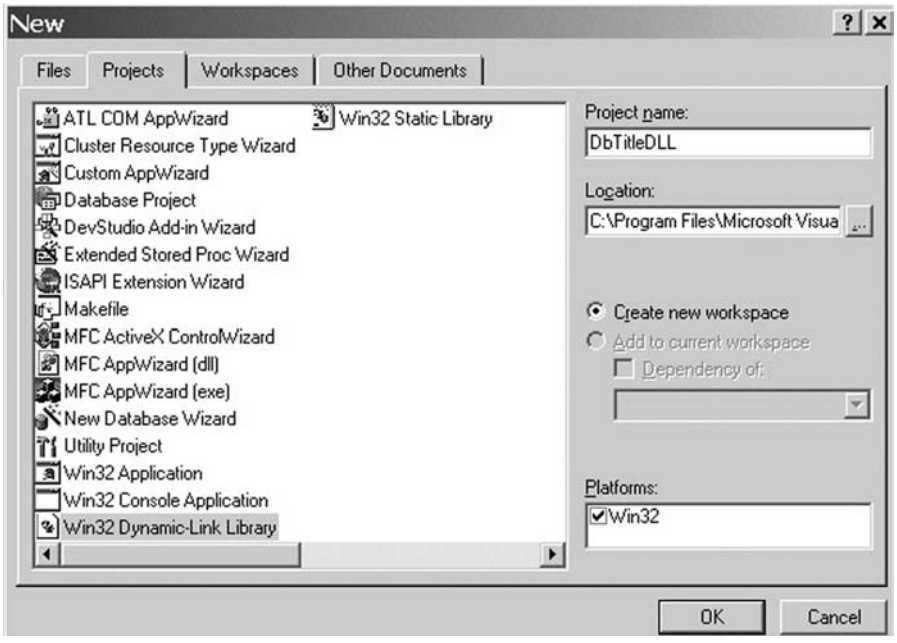
CHAPTER 11

Creating DLLs

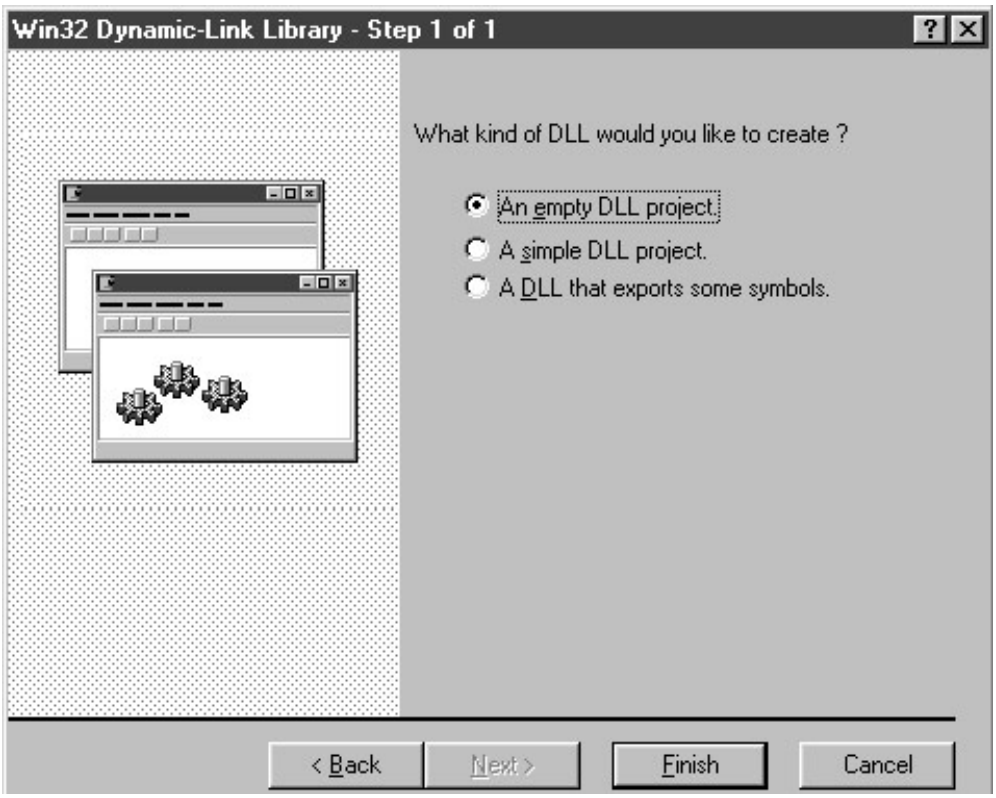
DLL stands for Dynamic Link Library. A dll is a binary file which contains some functions that can be called from most languages including LotusScript at run time, these calls to the dll can be made one after the other or simultaneously. Dlls are great for providing modularity when writing full blown applications as you can break your program into smaller modules, write some common functions and compile them in a single dll file. Hence an application might call three dlls: input.dll, output.dll and processing.dll. In this way all function calls for inputting data into the application can reside in the common input dll and so on. Whilst there are several types of dlls, we are just going to create regular ones, which export 'C functions', which can be called from LotusScript directly.

In order to show you how dlls are created we are going to create a dll called DbTitle which will retrieve the title of the database file name we pass it. We will call the dll directly from an action button in a Notes database using LotusScript and we will pass the dll a database file name and server name, in turn it will pass back the database title. One thing that will no doubt please you, is that regardless of whether you are writing an application, dll or server add in task with the C++ API, the code is essentially the same, the only difference being the way the calls are made. So pretty much all the following code will be familiar to you already, so let's get started.

In order to create dlls open Microsoft Visual C++ and create a new project, Select Win32 Dynamic-Link Library and enter a project name of DbTitleDLL then click OK.



Click Finish to create an empty DLL project and then OK



Next, you will need to add the following file and settings exactly as we have been doing all along when we first learnt about ‘Setting Up The C++ Environment’:

Add the NOTESCPP.LIB file and amend the following settings as we did before:

Build > Set Active Configuration > Win32 Release

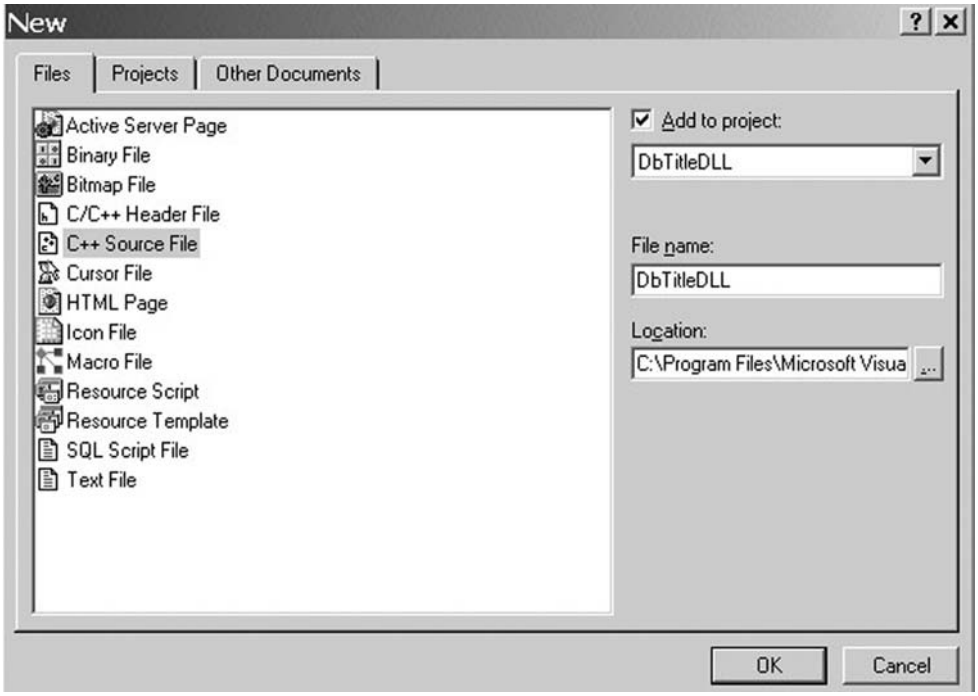
Project > Settings - C/C++ tab - in General category - add W32 to list of preprocessor definitions.

In Code Generation category - set Struct Member Alignment to 1 byte.

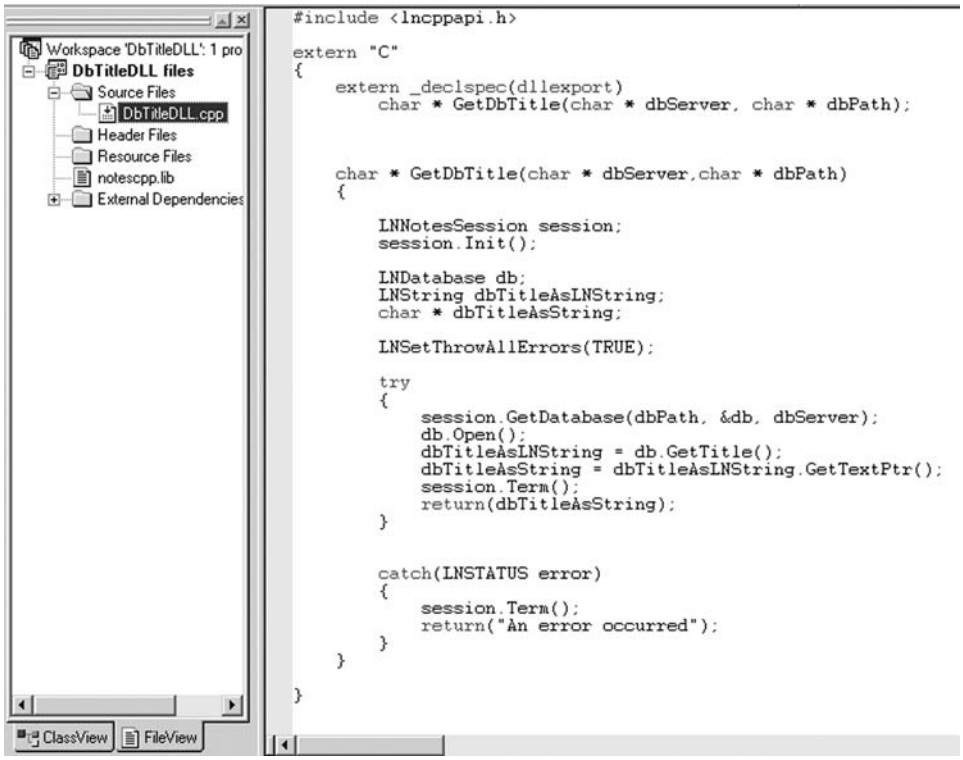
In Preprocessor category - add C:\NOTESCPP\INCLUDE to additional directories

Next you need to add your very own source file

Select Project > Add to Project > New then select C++ Source File and enter the File name as ‘DbTitleDLL’, then click OK



Here's the C++ source code I have entered to return a database title:



The **extern "C"** syntax will be new to you, the **extern "C"** syntax makes it possible for a C++ module to share data and routines with other languages.

The **extern _declspec(dllexport)** attribute is new also and enables you to export data, functions, classes, or class member functions from a DLL by using this **__declspec(dllexport)** keyword.

To export functions, the **extern_declspec(dllexport)** keyword must appear to the left of the calling-convention keyword, if a keyword is specified. For example:

```
extern _declspec(dllexport) returnValueOfFunctionDataType
__functionNameThatYouWishToMakeAvailableToOtherPrograms(fun
ctionParameter1, functionParameter...n);
```

Do not get caught up on these, an expanded explanation of these would require a more in depth understand of C & C++, which is not what this book is about. The thing to do is simply except them as a thing you need to incorporate into your dll C++ code and get on with writing the Notes API specific things. If you wish to find out more about these subjects, then you can search the web, buy C & C++ programming book, or if you have the Microsoft MSDN CD's then there's some good explanations of these. I would not get too hung up on them, get to grips with the basics of dlls as a whole first.

```
extern _declspec(dllexport)
char * GetDbTitle(char * dbServer, char * dbPath);
```

Here we define a function that we want to make available to LotusScript to call. The function name we wish to export in this case is called **GetDbTitle** which returns an array of characters, which LotusScript will happily treat as a string data type. The parameters the function expects to receive from LotusScript is a dbServer and dbPath value, both in the form of a character array, which we will pass from our LotusScript as string values. You can if fact list more than one function definition for export here, so you could potential list many different functions in this dll that you could call. For this example however for simplicity we list only one function.

```
char * GetDbTitle(char * dbServer,char * dbPath)
    {

        LNNotesSession session;
        session.Init();

        LNDatabase db;
        LNString dbTitleAsLNString;
        char * dbTitleAsString;

        LNSetThrowAllErrors(TRUE);
```

Next we see what appears to be the same definition of the function **GetDbTitle**, although the definition is identical we are now actually writing the function, whereas in the **extern _declspec(dllexport)** definition we were simply saying that we have a function in this code module called **GetDbTitle** and we would like to make it available to other languages to call. In a way we are exposing this function to the outside world so to speak.

All the rest of the code should make perfect sense to you now, we initialize a session, define a database object, a LNString and character string which will eventually contain the database title. We next make sure all errors are thrown to our catch block with the statement **LNSetThrowAllErrors(TRUE);**

```
try
    {
        session.GetDatabase(dbPath, &db, dbServer);
        db.Open();
        dbTitleAsLNString = db.GetTitle();
        dbTitleAsString = dbTitleAsLNString.GetTextPtr();
        session.Term();
        return(dbTitleAsString);
    }
```

Next, we try to get a handle to a database and return an initialize database based on what path and server names we pass into the function as shown here: **session.GetDatabase(dbPath, &db, dbServer);**

You will notice straight away that the dbPath and dbServer values we pass the **GetDatabase** function of the session object are the same ones that are passed to our exported **GetDbTitle** function which we will call from our LotusScript code in a moment. In this way we are able to pass values directly into a C++ API program from LotusScript. All will become clear when we start writing our LotusScript code in a moment, hang in there whilst I finish explaining the rest of the API code.

Next, we need to open the initialized database object by calling **db.Open();**

Next, we get the title of the database by calling the database object function **db.GetTitle();** This returns the database title as an LNString object. Remember what I said about being able to pass literal strings to objects that expect LNStrings throughout the API, but when it comes to API objects returning string values, invariably you will always be passed an LNString object back. If you try to capture this LNString object in a character array that you have defined, then you will get an error. So like it or lump it, you'll have to become familiar with the LNString object. So we have a LNString object that contains the database title, the next thing we do is simply capture the database title as a literal string so that our **GetDbTitle** function can return a character array to the calling LotusScript. This line

dbTitleAsString = dbTitleAsLNString.GetTextPtr();

gets the literal text string from the LNString object **dbTitleAsLNString** and assigns it to our character array

dbTitleAs String.

Next, we terminate the session with **session.Term();** and return of database title string back to the calling process with the following statement:

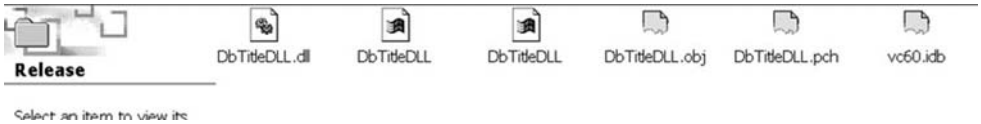
return(dbTitleAs String);.

In this way our calling LotusScript will receive the database title in tact from the API.

```
catch(LNSTATUS error)
    {
        session.Term();
        return("An error occurred");
    }
```

Our catch block simply terminates the session and returns a string titled "An Error Occurred" to the calling process. When we write our LotusScript next, try putting a valid and then invalid database path to see if the error is handled correctly by the API.

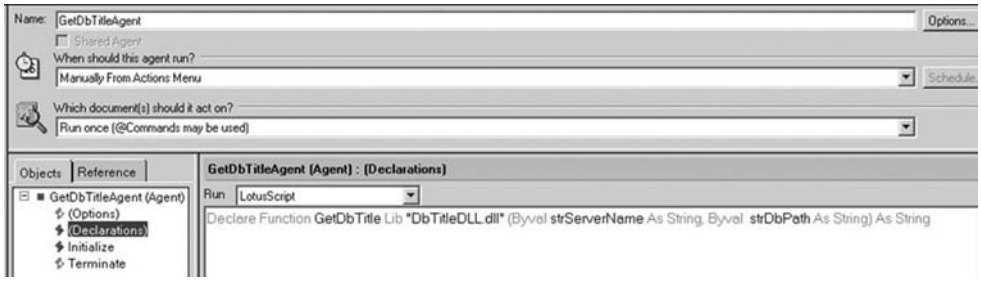
Okay, well done! Compile the C++ code.



You will notice in the Release directory, in my case: C:\Program Files\Microsoft Visual Studio\MyProjects\DbTitleDLL\DbTitleDLL.dll.

A new dll file has been created for you. Copy this file into your Notes program directory along with the LCPPN22.DLL file if you have not already copied it there in previous exercises. If you have been following the exercises then the LCPPN22.DLL file will already have been copied to your Notes program directory. In my case with the stock standard C++ API Toolkit install, this file resides in: C:\NOTESCPP\lib\mswin32\lncppn22.dll.

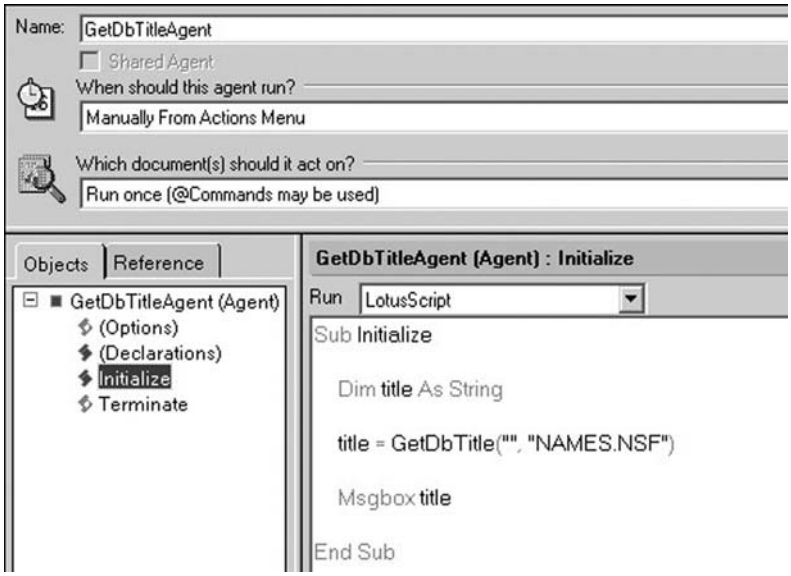
Next, for the LotusScript, create an agent in any database as shown below:



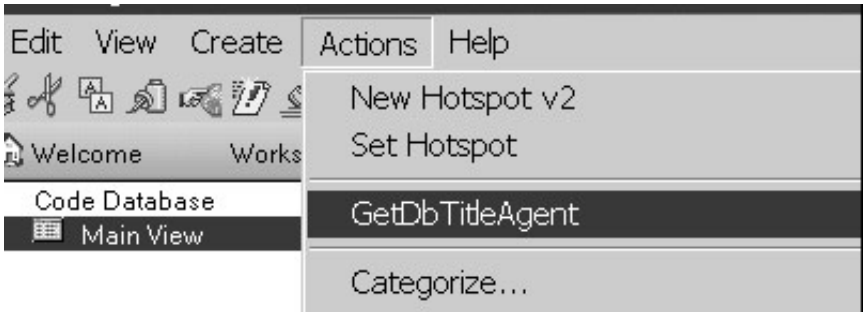
You will notice here in the Declaration section we define the function in our newly created dll. You will also notice we specify the dll file name after the Lib value as DbTitleDLL.dll, so that our LotusScript knows where to find the dll file in the Notes program directory. Here we also define the parameters of the function, you can see here that we pass in a strServerName and strDbPath name as string values, feel free to lable these parameters as you wish, the only important thing is that the data types match up to those in our C++ dll definition. LotusScript string data types will pass into our API program as character arrays (char *). The other thing you will notice is the ByVal keyword, we use this because by default, LotusScript passes arguments to external functions by reference. Arguments can be passed by value using the ByVal keyword. There is some very good documentation on this 'ByVal' keyword in the

Domino Designer Help database, if you are unsure of the difference between passing by value and by reference.

In the initialize sub write something similar to the following to actually call the function we have defined. You'll see in my example below that I pass the GetDbTitle an empty server string parameter to signify my local machine and then pass the "names.nsf" as the database file path. I then display a message box of the result. Simple!



I then save the agent and call it from the Actions menu.



Here's the result!

Now you might like to try passing some invalid database file names to the function in your LotusScript and see the result. You should see the text “An error occurred” as we defined in our C++ catch block.

Well done, you have created a dll and called it from LotusScript.

CHAPTER 12

The Export DLL Example

Here is another dll example that exports a document's rich text content to a Word file. The API provides some nice exporting and importing functionality. You can input from and export to a wide variety of different file types. In this example we are going to create a dll that will accept a server and database file path, a view name and a word file name string. The dll will then go and open a database, open the first document in the specified view, navigate to the rich text field and export the entire contents to a Word file.

Here is the C++ source code:

```
#include <lncppapi.h>

extern "C"
{
    extern _declspec(dllexport)
        char * ExportRT(char * strDbServer, char * strDbPath, char * strViewName, char * strWordFileNameAndPath);
}

        char * ExportRT(char * strDbServer, char * strDbPath, char * strViewName, char * strWordFileNameAndPath)

{


    LNNotesSession session;

    session.Init();

    LNDatabase db;
    LNDocument doc;
    LNRichText rt;
    LNViewFolder view;
    LNVFEntry viewRow;
    LNRTCursor cursorStart, cursorEnd;
    LNSetThrowAllErrors(TRUE);

    try
    {
        session.GetDatabase(strDbPath, &db, strDbServer);
        db.Open();
        db.GetViewFolder(strViewName, &view);
        view.Open();
        view.GetEntry(&viewRow);
        viewRow.GetDocument(&doc);
        doc.Open();
        doc.GetItem("Body", &rt);
        rt.GetCursor(&cursorStart);
        rt.GetCursor(&cursorEnd);
        cursorEnd.GotoLast(LNRTTYPE_STYLIZED_TEXT);
        rt.Export(strWordFileNameAndPath, cursorStart, cursorEnd);
    }

    catch (LNSTATUS lnError)
    {

        session.Term();
        return("OPPS! An Error occurred");
    }

    session.Term();
    return("Exported first view document to Word okay");
}
```

First, we get a handle to the database using the database path and server parameters that were passed to the dll with

`session.GetDatabase(strDbPath, &db, strDbServer)`

Next, we open the database and then return a handle to the view name that was passed using **`db.GetViewFolder(strViewName, &view)`**.

We then have to open the view so we can work with it. Next, we get a handle to the first entry(row) in that view with the statement **`view.GetEntry(&viewRow)`**, this returns the view Entry object **`viewRow`**.

We can get a handle to the actual document from the view entry object, by calling the function **`viewRow.GetDocument(&doc)`**, this then returns a document object which we open and get the rich text 'Body' item from the document with the following calll **`doc.GetItem ("Body", &rt)`**.

Once we have this rich text item, we initiate two cursor objects, which by default are both set to the start position of the rich text object i.e. top left hand corner. We do this with the following calls:

```
rt.GetCursor(&cursorStart), rt.GetCursor (&cursorEnd).
```

We then move the cursorEnd to the end of the last stylized text object in the rich text field, which in our example is at the end of the rich text field, we do this with the call

```
cursorEnd. GotoLast(LNRTTYPE_STYLIZED_TEXT).
```

We now have a cursor at the beginning and end of the rich text, next we simply export everything in between these cursor positions to our word document with the call

```
rt.Export (strWordFileNameAndPath, cursorStart, cursorEnd).
```

You will notice we passed this export function the name of the Word file we also passed into the dll.

That is all there is to it. Hopefully, you can see how easy it is to export data from rich text, and how you could literally pin point any position in the rich text by positioning your cursors around some selected elements.

Here is a list of other rich text element keywords that you could use the

```
cursor. GotoFirst, cursor.GotoLast, cursor.GotoNext
```

functions to navigate around, the same way we called

```
cursorEnd.GotoLast(LNRTTYPE_STYLIZED _TEXT)
```

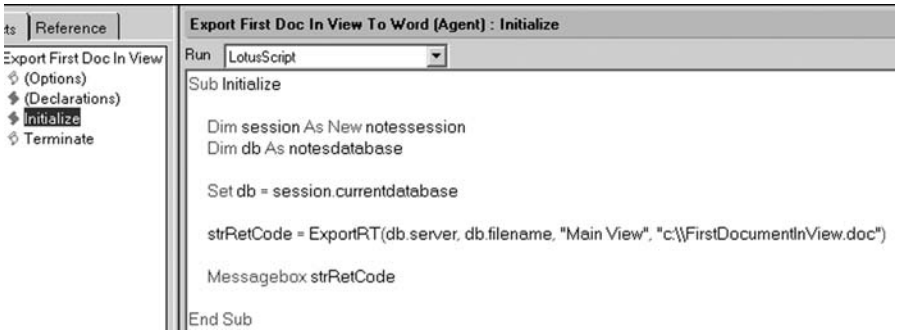
in our export code.

Here they are:

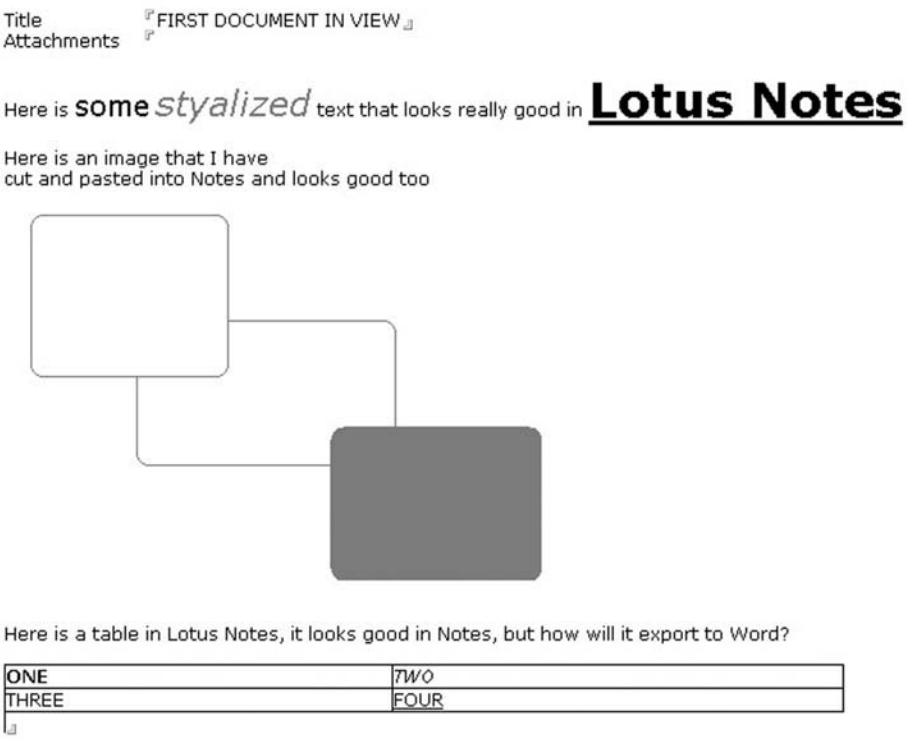
NRTTYPE_OBJECT
LNRTTYPE_BITMAP
LNRTTYPE_CONTAINER
LNRTTYPE_COMPOSITE_CONTAINER
LNRTTYPE_HOTSPOT
LNRTTYPE_ACTION_HOTSPOT
LNRTTYPE_COMPUTED_TEXT
LNRTTYPE_FORMULA_POPUP
LNRTTYPE_LINK_HOTSPOT
LNRTTYPE_TEXT_POPUP
LNRTTYPE_URL_LINK
LNRTTYPE_ITEM
LNRTTYPE_SECTION
LNRTTYPE_STYLIZED_TEXT
LNRTTYPE_TABLE

LNRTTYPE_TABLE_CELL
LNRTTYPE_ELEMENT
LNRTTYPE_ACTIVE_OBJECT
LNRTTYPE_JAVA_APPLET
LNRTTYPE_ANCHOR_LOCATION
LNRTTYPE_SUBFORM
LNRTTYPE_COMPUTED_SUBFORM
LNRTTYPE_BUTTON
LNRTTYPE_COMPOSITE_DATA
LNRTTYPE_EMBEDDED_OBJECT
LNRTTYPE_ATTACHMENT
LNRTTYPE_FORM_FIELD
LNRTTYPE_FORMULA_FIELD
LNRTTYPE_KEYWORDS_FIELD
LNRTTYPE_NAMES_FIELD
LNRTTYPE_AUTHORS_FIELD
LNRTTYPE_READERS_FIELD
LNRTTYPE_NUMBER_FIELD
LNRTTYPE_PASSWORD_FIELD
LNRTTYPE_RICH_TEXT_FIELD
LNRTTYPE_TEXT_FIELD
LNRTTYPE_TIME_FIELD
LNRTTYPE_GRAPHIC
LNRTTYPE_LINK
LNRTTYPE_ANCHOR_LINK
LNRTTYPE_DATABASE_LINK
LNRTTYPE_DOCUMENT_LINK
LNRTTYPE_VIEW_LINK
LNRTTYPE_OLE_OBJECT
LNRTTYPE_METAFILE
LNRTTYPE_CGM_METAFILE
LNRTTYPE_MAC_METAFILE
LNRTTYPE_PM_METAFILE
LNRTTYPE_WIN_METAFILE
LNRTTYPE_BITMAP_COLOR_TABLE
LNRTTYPE_BITMAP_PATTERN_TABLE
LNRTTYPE_BITMAP_SEGMENT
LNRTTYPE_BITMAP_TRANSPARENCY_TABLE
LNRTTYPE_MAC_METAFILE_SEGMENT
LNRTTYPE_NEW_PARAGRAPH
LNRTTYPE_PM_METAFILE_SEGMENT
LNRTTYPE_WIN_METAFILE_SEGMENT

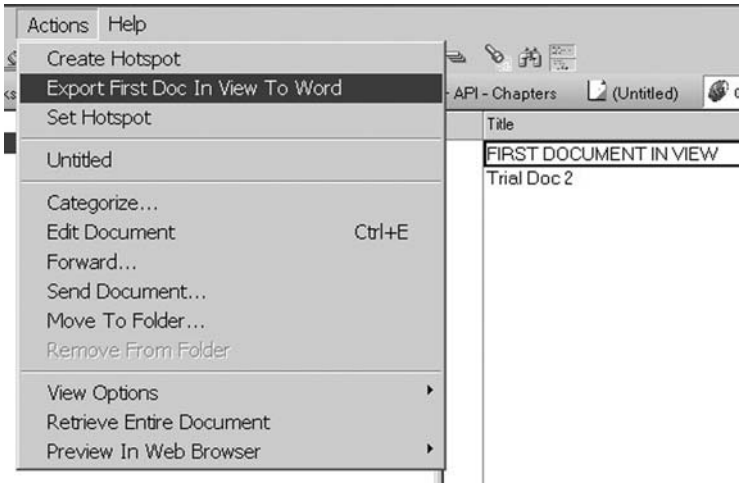
Here's what the calling LotusScript code looks like:



Here is what my Notes document rich text body field looks like in the first document in the view I'm going to pass as a parameter. As you can see it contains stylized text and colored text with spelling mistakes, an image and a table with stylized text in each cell.

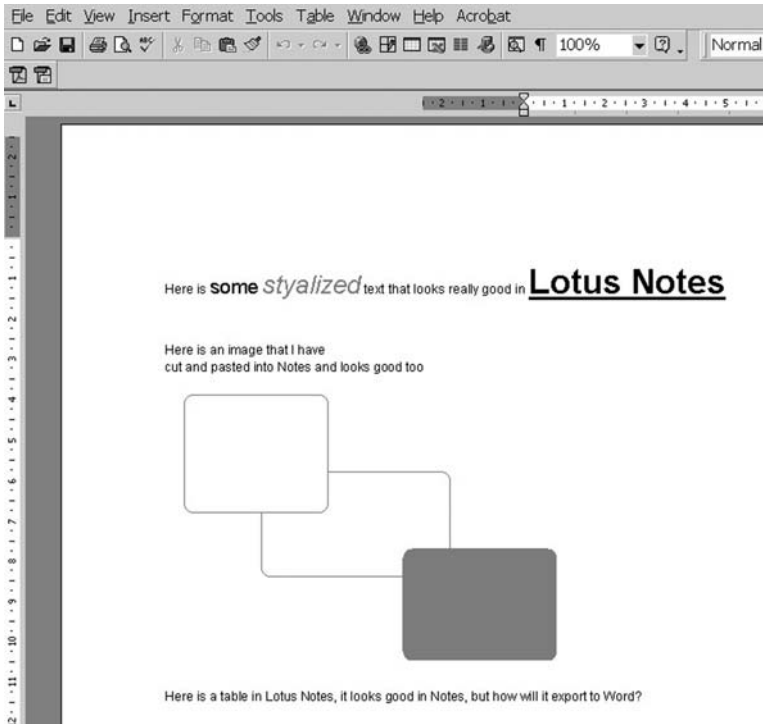


I close the document and call the 'Export First Doc In View To Word' agent I have written, which in turn will call the dll function and export the contents of the first document's rich text field contents to a Word file.



The following message is displayed when the agent has finished:

I go and open the Word file on my system and voila! The rich text has been preserved exactly as it was in Lotus Notes.



CHAPTER 13

The Create Table DLL Example

A common question I often hear is: “can the C++ API create and navigate tables in rich text.” The answer is yes it can, and it can quite easily.

Here’s another dll example to illustrate this fact. This dll will create a new document and in that document, create a new table in the rich text with two rows and two columns, it will color each cell a different color, insert some text into the first two cells in the first row and increase the border size between the two columns.

Here’s the source code:

```
#include <lncppapi.h>

extern "C"
{
	extern _declspec(dllexport)
		char * CreateTable(char * strDbServer, char * strDbPath, char *
firstCellText, char * secondCellText);
}

		char * CreateTable(char * strDbServer, char * strDbPath, char *
firstCellText, char * secondCellText)

{


	LNNotesSession session;

	session.Init();

	LNDatabase db;
	LNDocument doc;
	LNRichText rt;
	LNINT rows = 2;
	LNINT cols =2;
	LNTable table;
	LNTableCell cell1, cell2, cell3, cell4;
	LNRTCursor cursor;
	LNSetThrowAllErrors(TRUE);

	try
	{
		session.GetDatabase(strDbPath, &db, strDbServer);
		db.Open();
		// Create a new document in the database and open it.
		db.CreateDocument(&doc, "Main Doc");
		// Create the rich text "Body" field in the note.
		doc.CreateItem("Body", &rt);
		//Get cursor position at start of rich text
		rt.GetCursor(&cursor);
		rt.StartNewParagraph(&cursor);
		rt.Insert("Here we insert our table on next line:-", &cursor);
		rt.StartNewParagraph(&cursor);
		// Add the new table to the current doc.
		rt.CreateTable(rows, cols,&cursor, &table );
		cell1 = table(0,0);
		cell2 = table(0,1);
		cell3 = table(1,0);
		cell4 = table(1,1);
		cell1.SetBackgroundColor(LNCOLOR_RED);
		cell2.SetBackgroundColor(LNCOLOR_GREEN);
		cell3.SetBackgroundColor(LNCOLOR_BLUE);
		cell4.SetBackgroundColor(LNCOLOR_YELLOW);
		cell2.SetLeftBorderWidth(9);
		cell4.SetLeftBorderWidth(9);
		cursor.GotoFirst(LNRTTYPE_TABLE);
		cursor.GotoFirst(LNRTTYPE_TABLE_CELL);
		rt.Insert(firstCellText, &cursor);
		cursor.GotoNext(LNRTTYPE_TABLE_CELL);
		rt.Insert(secondCellText, &cursor);
		// Save and close the document & database
		doc.Save();
		doc.Close();
		db.Close();

	}

	catch (LNSTATUS lnError)
```

```
	{

		session.Term();
		return("OPPS! An Error occurred");
	}

	session.Term();
	return("Table created okay");
}
```

As you can see the code is pretty straight forward, so I'll pick up at the line:

```
rt.CreateTable(rows, cols,&cursor, &table )
```

This line of code creates our new table of two rows and two columns at the cursor position and returns a pointer to the new table object. Next we see the following:

```
cell1 = table(0,0);
cell2 = table(0,1);
cell3 = table(1,0);
cell4 = table(1,1);
```

What I'm doing here is initializing the LNTableCell objects cell1, cell2 etc. by specifying the exact positions in the table i.e. for cell1 we can see that it exists in the table at row 0, column 0, whereas cell2 exists within the table at row 0 column 1. Now that we have initialized LNTableCell objects we can call their functions and set their properties, which is exactly what we see in the next portion of code:

```
cell1.SetBackgroundColor(LNCOLOR_RED);
cell2.SetBackgroundColor(LNCOLOR_GREEN);
cell3.SetBackgroundColor(LNCOLOR_BLUE);
cell4.SetBackgroundColor(LNCOLOR_YELLOW);
cell2.SetLeftBorderWidth(9);
cell4.SetLeftBorderWidth(9);
```

Currently, the cursor position is residing just after the table we created and inserted into the rich text. If we want to insert text into the first two cells of the table, which we do, then we need to move the cursor back to the start of the table object and then move it into the first cell, we do this as follows:

```
cursor.GotoFirst(LNRTTYPE_TABLE);
cursor.GotoFirst(LNRTTYPE_TABLE_CELL);
```

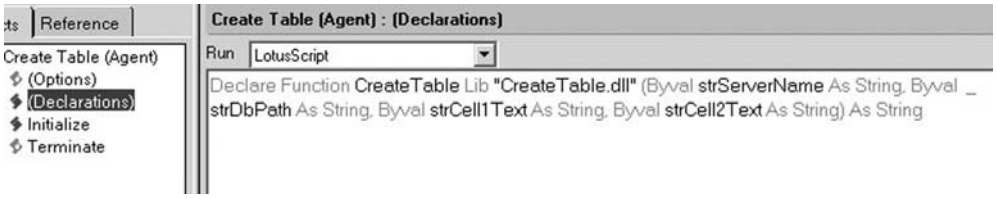
Once inside the first cell we can call the Insert method of the rich text object and insert text as follows:

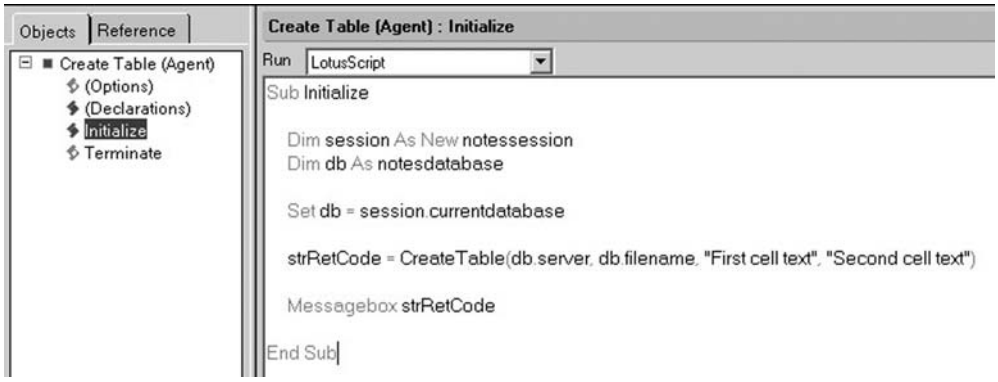
```
rt.Insert(firstCellText, &cursor);
```

We then move to the next cell and do the same

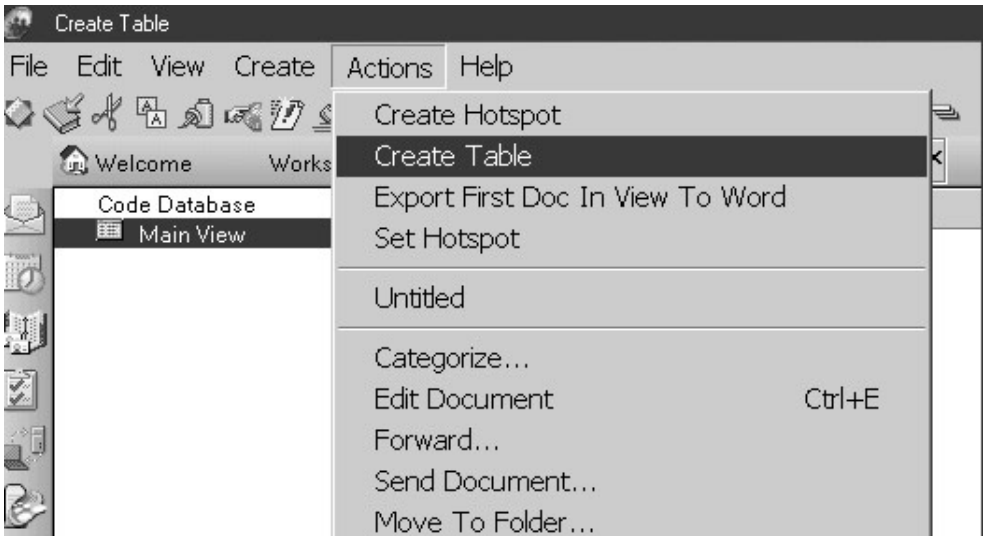
```
cursor.GotoNext(LNRTTYPE_TABLE_CELL);
rt.Insert(secondCellText, &cursor);
```

All that's left to do now is save the document and we're done. Here's the calling LotusScript code:



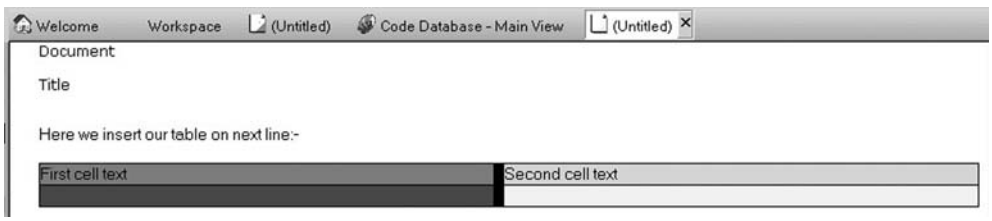


I call the agent from Notes as follows:



The following message appears once the agent has finished running:

I open the newly created document and view what the API program created:



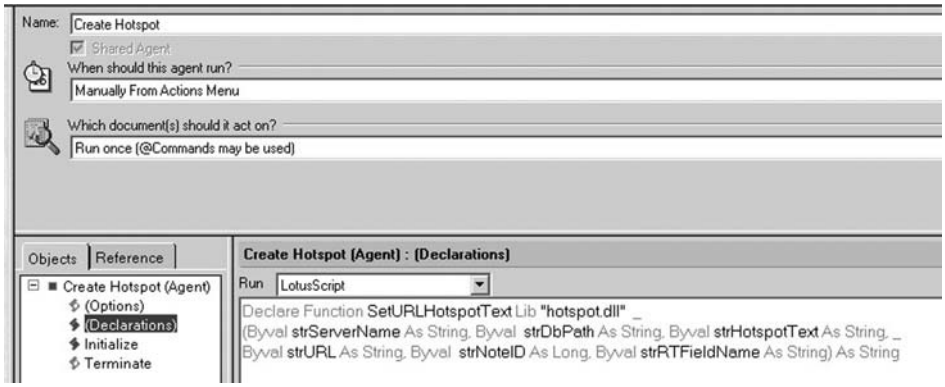
CHAPTER 14

The Hotspot DLL Example

We are now going to create an agent. We will open a document in edit mode, highlight any piece of text in a given rich text field and run the agent to turn the selected text into a URL hotspot, which in our case will link back to www.lotus.com.

Take a look at the following LotusScript agent:

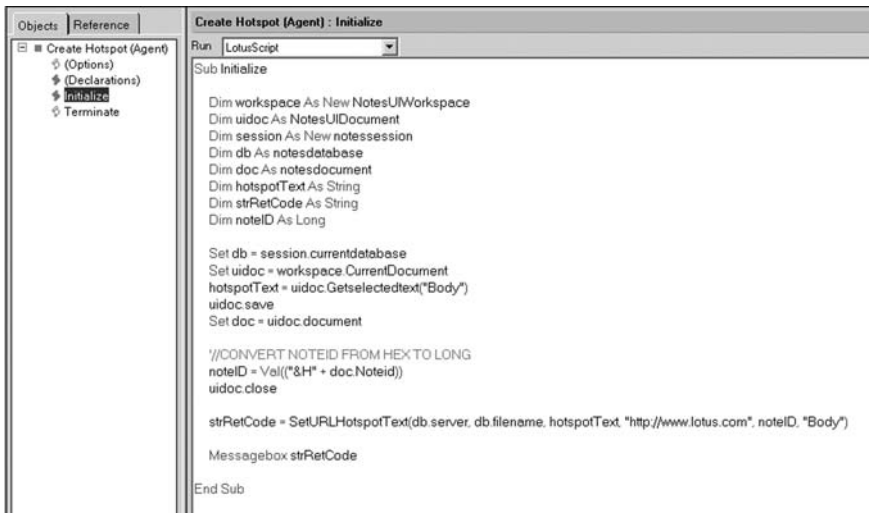
Here's the declarations section:-



You can clearly see we have defined a function called **`SetURLHotspotText`**, which exists in a hotspot.dll file we have not created yet. It passes the following parameters to the dll function:

server name, database path, the text string to be turned into a URL hotspot, the URL we want the hotspot to use, the NoteID of the document containing the highlighted text and the name of the rich text field containing the highlighted text.

Here's the Initialize sub:-



We set up some standard variables for our current workspace, we get a handle to the current database and current document we are working on in the user interface (UI). We then set the text string variable 'hotspotText' to the value of the currently highlighted text, using a nice little Notes document function called 'GetSelectedText', we then save the doc and get a handle to the same document using the backend NotesDocument class.

We then retrieve the NotesID of the current document using a **`doc.NoteID`** method of the backend NotesDocument class. We need to turn this hex value into a long, so we can pass it to the dll as a long value, we use the LotusScript 'Val()' function to do this.

We then close the UI document and call the dll function SetURLHotspotText, passing all the values for the current server, database filename, the highlighted text, and the URL.

Here's the C++ dll source code:

```
#include <lncppapi.h>

extern "C"
{
    extern _declspec(dllexport)
        char * SetURLHotspotText(char * strDbServer,
        char * strDbPath, char * strHighlightedText, char * strURL, long lngNoteID, char * strRTFieldName);
}

char * SetURLHotspotText(char * strDbServer,char * strDbPath, char * strHighlightedText,
        char * strURL, long lngNoteID, char * strRTFieldName)

{


    LNNotesSession session;

    session.Init();

    LNDatabase db;
    LNDocument doc;
    LNRichText rt;
    LNRTObject temp;
    LNRTCursor cursor;
    LNINT lnintHotspotCharLength = 0;
    unsigned int x=0;

    LNSetThrowAllErrors(TRUE);

    try
    {
        session.GetDatabase(strDbPath, &db, strDbServer);
        db.Open();
        db.GetDocument((NOTEID)lngNoteID, &doc);
        doc.Open();
        doc.GetItem((LNString)strRTFieldName, &rt);
        rt.GetCursor(&cursor);
        cursor.SetSearchOptions(LNRTSEARCHFLAGS_MATCH_CASE);
        LNURLLink lnurllinkHotspot((LNString)strURL, (LNString)strHighlightedText);

        if(cursor.GotoFirst((LNString)strHighlightedText)==LNNOERROR)
        {
            cursor.GetObject(&temp);

            if(temp.IsType(LNRTTYPE_STYLIZED_TEXT))
            {
                rt.Insert(lnurllinkHotspot,&cursor);

                for(x=1; x <= strlen(strHighlightedText); x++)
                        lnintHotspotCharLength++;

                rt.Delete(&cursor,lnintHotspotCharLength);

                doc.Save(LNNOTESAVEFLAGS_FORCE);
                doc.Close();
                db.Close();
                session.Term();
                return("DONE");
            }
        }
    }

    catch (LNSTATUS lnError)
    {
        session.Term();
        return("An error occurred");
    }

    session.Term();
    return("Okay");
}
```

In the C++ code you can see we have declared the exported function as **`SetURLHotspotText`** - exactly as we referenced it in our LotusScript code. Starting in the 'try' block we have already created a session for the API at the start of the function, next we get a handle to the database where we highlighted our text. We do this by passing the **`GetDatabase`** method of the session object, the server and database path parameters, which were passed into our SetURLHotspotText function as the **`strDbPath`** and **`strDbServer`** arguments.

We then open the database and get a handle to the document that contains the text we want to turn into a URL hotspot, notice how we convert the lngNoteID long variable we passed in into a NOTEID type, this is called casting in C++ and is similar to converting say an integer value in LotusScript into a String value by a statement like: **`Cstr(integerValue)`**

Here's a screen shot from the Notes API reference which shows you three different ways of getting a handle directly to a document in a Notes database, in the example here, we have used the first option.

LNDatabase::GetDocument

Gets a pointer to the specified document in the database.

Syntax

LNSTATUS GetDocument(const NOTEID idnote, LNDocument *doc)
LNSTATUS GetDocument(const UNID *uniidnote, LNDocument *doc)
LNSTATUS GetDocument(NOTEHANDLE handle, LNDocument *doc)

Arguments

idnote
Input, note ID of a document in the database.

uniidnote
Input, pointer to universal note ID of document.

handle
Input, C API open note handle.

doc
Output, pointer to document.

Return value

LNSTATUS
Zero (LNNOERROR) if successful, non-zero otherwise.

Usage

If you call the NOTEHANDLE version of this function successfully, you must not subsequently use the handle to modify the document. Also, you must not use the handle in another call to GetDocument, to instantiate another LNDocument object. Using a handle to instantiate a C++ API object gives ownership of that handle to the C++ object.

See also

LNNote::GetNoteHandle
LNDatabase class

Now that we have got a handle to the document using the NoteID long, we now open the document and get a handle to the rich text field item by using the **`GetItem`** method of the newly returned document object. You will notice that we pass this function the name of the rich text field we want to get a hold of, I also cast the field name into a LNString object. You do not really need to do this, the literal string we passed will still work, but it gives you another example of casting.

The next thing you see, is that we return a rich text cursor object and set the search options for the cursor to make sure that any searches for text in the rich text object are case sensitive. This means that when we search for the highlighted text string passed to the dll in the rich text object, we want the case to match exactly.

```
rt.GetCursor(&cursor);
cursor.SetSearchOptions(LNRTSEARCHFLAGS_MATCH_CASE);
```

Next, we initialize a LNURLLink object, which basically contains the text and the link for a Notes URL link. As you can see I pass it both parameters;

```
LNURLLink lnurllinkHotspot((LNString)strURL,
(LNString)strHighlightedText);
```

Next, we call a very handy cursor function **cursor.GotoFirst**, however instead of passing it a rich text object type as we have done in previous chapters, this time we pass it a text string. It will now try and position the cursor at the beginning of the string I passed it if it exists in the the rich text.

```
if(cursor.GotoFirst((LNString)strHighlightedText)==LNNOERRO
R)
```

You will notice that we also check for an error, just in case the string can't be found. If a string can't be found using this cursor function an error is raised, although we do not really need it as we know the string exists, it's good practice to include it. Next, we call the **GetObject** function, which basically returns the object to the immediate right of the cursor position, this should now be at the beginning of the stylized text block containing our highlight string. We return the object as a **&temp** pointer

```
cursor.GetObject(&temp);
```

Now we have the object **&temp**, we can test to see if it's stylized text. If it is we can assume it's safe to proceed. Once again this is not really required, now before you hit me, I merely wanted to show you how you can navigate through rich text objects using the cursor and test to see what type of objects are there. This may sound relatively useless, but the rich text object is often the most misunderstood and complicated object. I want you to see that it's really quite simple. It's likely to be an object you are forced to master as many people want to use the API so they can get more granular access to Notes Rich Text items. Anyway I digress.

Next, we test to see if our temp object is in fact a stylized text object i.e. containing stylized text and not something else like a table. If it is, then I simply insert the LNURLLink object we created earlier just before it at the cursor position.

```
if(temp.IsType(LNRTTYPE_STYLIZED_TEXT))
{
rt.Insert(lnurllinkHotspot,&cursor);
```

Now we have inserted the url hotspot link and text, but still have the original stylized text in the rich text as well, so we now need to remove the old stylized text. To do this we need to find the length of the stylized text, this we know because we passed it into the dll function in the first place, so we write a simple for loop that loops through each character in the stylized text and increments a counter called **lnintHotspotCharLength++**. You will notice the **strlen** function, which is a function of a C++ library that returns the length of a string.

```
for(x=1; x <= strlen(strHighlightedText); x++)
lnintHotspotCharLength++;
```

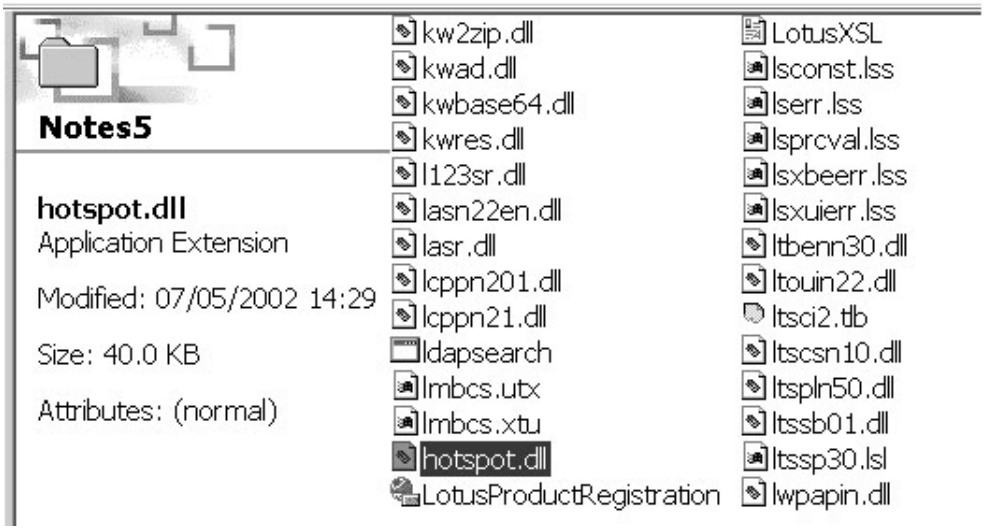
The cursor position now is immediately after the URL hotspot object we inserted, which is just before the stylized text, now what we do is call the **`rt.Delete`** function and pass it a cursor position and integer value which happens to be the length of the stylized text. What the delete function will now do is delete the specified number of characters from the cursor position, the number of characters in our case is equal to the stylized text length. Hence we delete the entire old stylized text string and leave only the URL Hotspot link.

```
rt.Delete(&cursor,lnintHotspotCharLength);
```

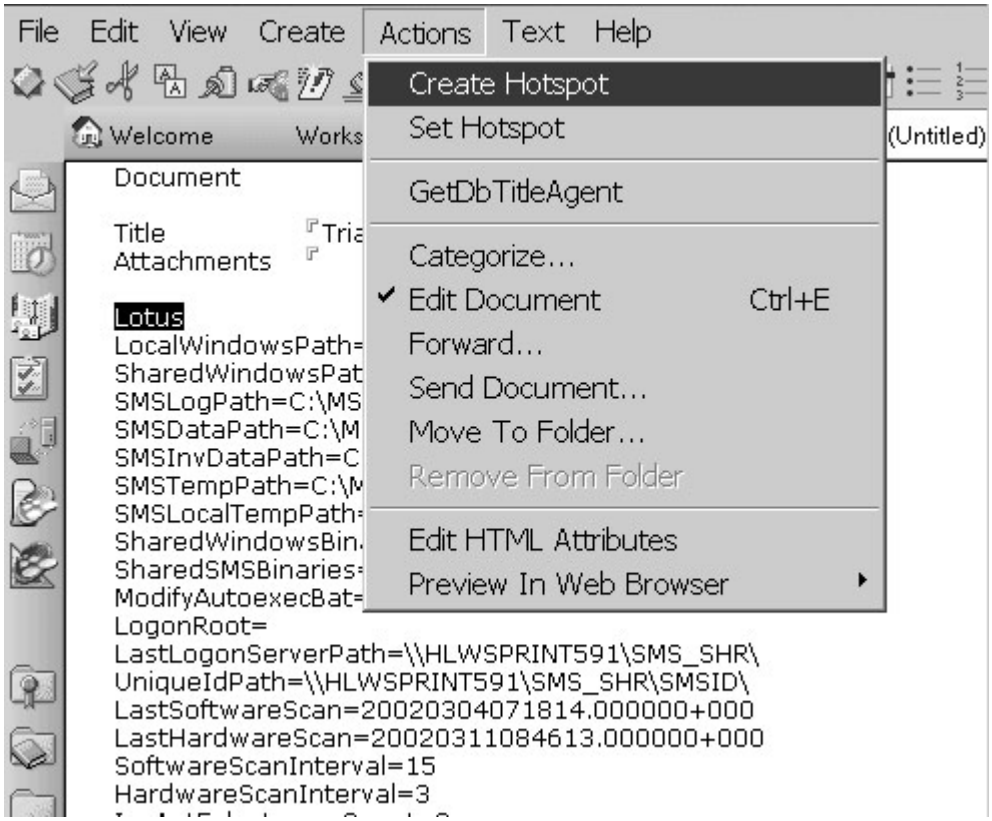
Next we simply save and close the document and we are done.

```
doc.Save(LNNOTESAVEFLAGS_FORCE);
doc.Close();
db.Close();
session.Term();
```

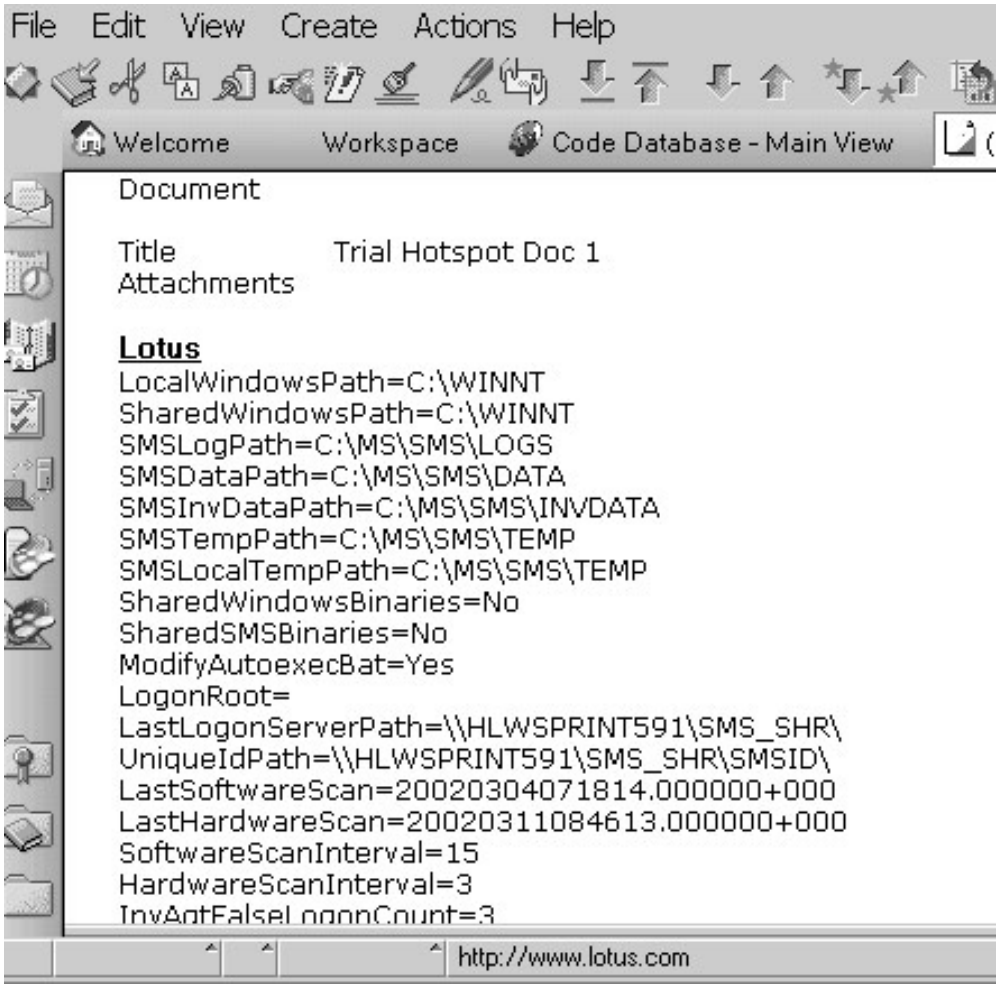
We copy our dll file into our Notes program directory.



We can then open a document in Notes that contains rich text in edit mode, highlight some text and then trigger our agent from the actions menu:


File Edit View Create Actions Text Help
Create Hotspot
Set Hotspot
GetDbTitleAgent
Categorize...
Edit Document Ctrl+E
Forward...
Send Document...
Move To Folder...
Remove From Folder
Edit HTML Attributes
Preview In Web Browser
Welcome
(Untitled)
Document
Title
Attachments
Lotus
LocalWindowsPath=
SharedWindowsPat
SMSLogPath=C:\MS
SMSDataPath=C:\M
SMSInvDataPath=C
SMSTempPath=C:\M
SMSLocalTempPath=
SharedWindowsBin
SharedSMSBinaries=
ModifyAutoexecBat=
LogonRoot=
LastLogonServerPath=\\HLWSPRINT591\SMS_SHR\
UniqueIdPath=\\HLWSPRINT591\SMS_SHR\SMSID\
LastSoftwareScan=20020304071814.000000+000
LastHardwareScan=20020311084613.000000+000
SoftwareScanInterval=15
HardwareScanInterval=3

The agent runs and closes the document, if we re-open the document, you will see that our highlighted 'Lotus' stylized text has now turned into a URL hotspot, that links to www.lotus.com.



CHAPTER 15

A Brief Overview of Notes Server Add-In Tasks

This section describes very briefly how to use the Notes C++ API to create Notes server add-in tasks.

A server add-in task is an executable program that runs alongside other tasks that make up the Notes server software. It allows you to perform any operation on Notes databases accessible to the Notes server. You can use server add-in tasks for a single operation, but its main purpose is to perform some operations periodically i.e. on a scheduled basis.

Note: *If you want to create a server add-in to perform a single operation, you may want to consider a stand-alone or a dll as an alternative. Apart from the fact that a server add-in automatically logs in with the server account, there is no other difference.*

For example, suppose that you want to retrieve the contents of a relational database periodically to update corresponding documents in a Notes database. You can do this with an add-in task as follows:

1. Develop a stand-alone program that contains all of the C++ API code needed to read the relational database and update the Notes database.
2. Add scheduling code to the program to extend it to a Notes server add-in. For example, you can specify the task to execute once per hour.
3. Start the add-in task with the Notes server software.

The add-in will run until the Notes server is shut down. Every hour, the add-in task will check the relational database for changes, and write these changes to the Notes database.

If some other time interval between executions of the task is required, the timing control code may be modified to specify any schedule, such as once each night at 3:00 AM, or once every 15 seconds. Add-ins may also be written to execute more than one job, each with its own schedule.

Program Structure of an Add-In Task

While the structure of an add-in task that executes only once is identical to a stand-alone API program, for periodic scheduling it is distinguished from other API applications by the presence of an LNServerAddin object, representing the add-in task.

In addition to the other API code you write to work with Notes, you will need to insert calls to control the execution of the server add-in. In order to do this, declare an LNServerAddin object, and include one call to

LNNotesSession::GetServer Addin,

which actually retrieves the LNServerAddin object for your Notes session. The function GetServerAddin is declared as follows:

```
LNSTATUS GetServerAddin (const LNString &task, const
LNString &text, LNServerAddin *addin)
```

The first and second arguments are the task name and description used for the default status line. If you issue a SHOW TASKS command at the server console, these arguments will be displayed together on a single line. The third argument is the address of an LNServerAddin object which will be initialized on return.

Once you have called GetServerAddin, you can call LNServerAddin functions as needed to control the execution of your server add-in task.

Controlling the Task Execution

The functions provided by the class LNServerAddin allow you to set up an appropriate schedule interval for your server add-in:

Task scheduling.

```
LNBOOL IsNewDay ()
LNBOOL HaveMinutesElapsed (LNINT n)
LNBOOL HaveSecondsElapsed (LNINT n)
```

These functions return TRUE once every day, n minutes or seconds since the add-in started.

For example, to schedule a task to execute every 2 hours, you would call HaveMinutesElapsed(120), and when this call returned a non-zero value, execute the task.

Yielding processor control.

```
LNBOOL Idle ()
LNBOOL Idle (LNINT msecs)
```

These functions yield control of the processor to Notes so that other tasks can execute, and receive control back when the server decides the add-in may proceed. If no argument is specified, Notes decides when the add-in should resume. If an argument is specified, Notes suspends the add-in for the specified number of milliseconds; if zero is specified, the function returns immediately. A TRUE value is returned from the function call when the Notes server wants the add-in to shut down.

Checking for task termination conditions.

```
LNBOOL ShouldAddinTerminate ()
```

The function ShouldAddInTerminate will return TRUE when the Notes server wants the add-in to shut down. If the task execution of your add-in requires a long time to complete, you may want to call this function periodically to enable the add-in to respond to a termination request sent by the Notes server.

Putting all these functions together, the main loop of a server add-in looks like this:

```
LNNotesSession session; // Create a session object
LNServerAddin addin;
// Initialize the session
session.init ();
// Get the associated add-in object
session.GetServerAddin("Add In's Name", "<Your description here>",  &addin);
while (TRUE)
{
    // check the schedule condition (day, n minutes, n seconds)
    if (addin.HaveSecondsElapsed (120))
    {
        // perform the operation
    }
    else
    if (addin.Idle (120) == TRUE)
    {
        // Notes server requests you to shut down
        // So, clean up and return
        return;
    }
}
```

Note: *In general, you may prefer a call to Idle with an explicitly specified wait time, because it may affect the performance of the Notes server.*

Status and Log Information of an Add-In Task

When you create a server add-in task, you can provide users with information about the status of the task by specifying messages to be displayed on the server console and stored in the server LOG.NSF log file.

Every add-in task you create has one default status line. If you have a complex task that performs several sub-operations, you could define separate status lines for each of them.

The following functions in the LNServerAddin class enable you to work with the default add-in task status line and the server log:

Change the task name.

```
void SetDefaultStatusLineTaskName (const LNString &name)
```

Change the task description.

```
void SetDefaultStatusLineText (const LNString &text)
```

Add an entry in the server log LOG.NSF.

```
void AppendLogMessage (const LNString &message)
```

This function is also extremely useful when you need to debug your add-in.

The following C++ program implements a Notes server add-in that runs once every day, and compacts a particular database in the local NOTES directory. You can extend it easily by directory search functions to compact all databases stored on the server once a day.

```
#include <iostream.h>
#include <lncppapi.h>

// Function to compact a database.

void compact_db(LNNotesSession &session)
{
    LNDatabase db;
    session.GetDatabase ("DISCUSAPI.NSF", &db);
    db.Compact ();
}

void main (int argc, char *argv[])
{
    LNNotesSession session; // Create a session object
    LNServerAddin addin;
    LNSetThrowAllErrors(TRUE); // Enable exceptions
    try
    {
        // Open a session and get the add-in object
        session.Init();
        session.GetServerAddin("Database Compactor", "Idle", &addin);

        while (TRUE)
        {
            // Start it once a day: returns TRUE
            // at or after 2am in the morning
            if (addin.IsNewDay ())
            {
                // Update the status line
                addin.SetDefaultStatusLineText ("Busy");
                // Call the task execution function
                compact_database(session);
                addin.SetDefaultStatusLineText ("Idle");
            }
            // Here, we will wait always one hour:
            // it doesn't matter whether the task is
            // executed at 2am or at 3am
            if (addin.Idle(3600000) == TRUE)
            {
                addin.AppendLogMessage ("Accepting request to shut down");
                return;
            }
        }
    }

    catch(LNSTATUS error)
    {
        // Handle errors:
        // We're on the server, so write it into the log.
        char errorBuf[LNERROR_MESSAGE_LENGTH];
        LNGetErrorMessage(error, errorBuf);
        addin.AppendLogMessage ("An error occurred: ");
        addin.AppendLogMessage (errorBuf);
    }
}
```

CHAPTER 16

More info on the LNNotesSession Object

In these next chapters titled “More info On...” I am going to go over some basic functionality of the respective API components. Full working code samples will not be provided, instead I assume that you have read the previous chapters, and that you are familiar with constructing the basic API framework around some of these objects. The explanations are still quite basic and basic in nature, I simply want to expose some of the very basic objects and functions that you may or may not be fully aware of to compliment your burgeoning skill set. These reference chapters will complete your introduction to the C++ API, and hopefully give you the confidence to progress and explore further yourselves in the Lotus API Toolkit documentation. From this very documentation I have taken most of the following information and simply added more detailed and easier to understand explanations.

Let us start with the core LNNotesSession class which provides an application program with a connection to Notes.

Here are some useful LNNotesSession class functions:

Retrieving the local Notes data directory.

```
LNString GetDataDirectory ()
```

Retrieving the user name from the ID file on the local machine.

```
LNString GetUserName ()
```

Environment variable handling.

```
LNString GetEnvironmentString (const LNString &variable)
```

and

```
LNSTATUS SetEnvironmentString (const LNString
&variable,const LNString &string)
```

These two functions allow you to return and set the values of environment variables as defined in the local NOTES.INI file.

Current date and time.

```
LNDatetime GetCurrentDatetime ()
```

Returns the current date/time.

The following functions allow you to create, delete and access databases

Open an existing database.

```
LNSTATUS GetDatabase (const LNString &path, LNDatabase
*db, const LNString &server = "" )
```

Given the path and the server of the database to be opened, this function initializes a database object whose address is the second parameter.

Note: *Whenever a function returns an object as a result of the operation, it is a pointer type argument to the function. In that case, you have to pass the address of an existing object of that type that will be initialized on return.*

Create and open a new database copy.

As for many other class member functions, this function exists in several versions allowing you to use the simplest one that fits your needs. The version with the most parameters is the following:

```
LNSTATUS CreateDatabaseCopy (const LNString &srcdb_path,
const LNString &srcdb_server, const LNString &newdb_path,
const LNString &newdb_server, constLNCreateDatabaseOptions
&options, LNDatabase *newdb = 0 )
```

You specify the Notes server and the path of the source database, the server and the path of the database to be created, and a set of options for the new database. In the last argument you pass the address of an LNDatabase object that is initialized on successful return.

Delete a database.

```
LNSTATUS DeleteDatabase (const LNString &path, const
LNString &server = "")
```

This version of the DeleteDatabase function gets the server and the path of the database in question.

Access an existing database

```
LNSTATUS GetDatabase (const LNString &path, LNDatabase
*db, const LNString &server = "" )
```

Note: *As you may have noticed, a Notes database is not created by instantiating a new object of class LNDatabase, and is not deleted by a corresponding delete operator, but rather by function calls to a session object. This is due to the general API concept to use these classes as easy-to-use interfaces to the real Notes objects. As such, an LNDatabase object is not a database itself; think of it as a handle to a database. The same applies to all other objects stored in databases.*

Example: Creating a New Database

Now, we will use the member functions of the LNNotesSession class to create a discussion database that will be used throughout the rest of the chapter. The example also shows how to initiate a session, and to set up an error handler.

```
#include <iostream.h>
#include <lncppapi.h>

void main(int argc, char *argv[])
{
    LNNotesSession session; // Create a session
    LNDatabase newDiscussDB; // Create a database object
    LNCreateDatabaseOptions options; // Create an options object
    LNSetThrowAllErrors(TRUE); // Enable the API exception handling

    try
    {
        session.Init();
        // Set options: the new database will inherit design updates
        options.SetInheritDesign (TRUE);
        // Create a database from the discussion template on the
        // local machine. It will be located at the Notes data directory.
        session.CreateDatabaseFromTemplate ("DISCUSS4.NTF","","DISCUSAPI.NSF","",options, &newDiscussDB);
        newDiscussDB.Close(); // close the database
    } // end try

    catch(LNSTATUS error)
    {
        char errorBuf[LNERROR_MESSAGE_LENGTH];
        LNGetErrorMessage(error, errorBuf);
        cout << "Error: " << errorBuf << endl;
    }
}
```

CHAPTER 17

More info on the LNDatabase Object

The LNDatabase class provides access to all objects contained in a database: agents, views, folders, and documents. Furthermore, you can access and modify the database properties, such as the name of the database, and the ACL.

Opening and Closing a Database

Before you can perform any operations on a database, you must open it. Creating a database with **`LNNotesSession::CreateDatabase`** automatically opens the database, whereas a call to **`LNNotesSession::GetDatabase`** does not.

Opening a database.

```
LNSTATUS Open (LNDBOPENFLAGS flags)
```

The passed optional argument specifies how operations on the database should be performed. The only flag you can specify is **`LNDBOPENFLAGS_DELAY_COMMIT`**. Setting this flag means that all updates will take place in cache memory, when you close the database, its contents on disk are synchronized.


Note: *Using this flag will speed up your API operations, but if the machine crashes, the changes are lost.*

Closing a database.

```
LNSTATUS Close ()
```

Closing a database also closes any notes it contains.


Note: *Any unsaved changes you made to any notes in the database will be lost when you close the database, so be sure to save any changes first. Also, if you don't close a database object before it goes out of scope, it will close automatically.*

Accessing Database Properties

Retrieving the Notes server and the path of a database.

```
LNString GetServer ()
LNString GetFilepath ()
```

If the database is stored on the local machine, GetServer returns an empty string.

Accessing the database title.

```
LNString GetTitle ()
LNSTATUS SetTitle (const LNString &title)
```

Accessing the design template attributes.

```
LNString GetTemplateName ()
```

It returns an empty string if the database is not a template.

```
LNSTATUS SetTemplateName (const LNString &name)
```

This function defines a new name for this database template.

```
LNString GetInheritsFromTemplateName ()
LNSTATUS SetInheritsFromTemplateName (const LNString
&name)
```

You can use these functions to determine whether the database is based on a template, and you can assign a new database template to it.

Accessing the database ACL.

```
LNSTATUS GetACL (LNACL *acl)
```

This function initializes the ACL object whose address you pass as an argument.

Continuing with our sample database, we will now assign a meaningful title to it, and define a different default access right.

```
LNNotesSession session; // Create a session object
LNDatabase discussDB; // Create a database object
LNACL acl;
LNSetThrowAllErrors(TRUE); // Enable exceptions
    try
    {
        session.Init();
        // Get the database and open it
        session.GetDatabase ("DISCUSAPI.NSF", &discussDB, "");
        discussDB.Open ();
        // Set the title
        discussDB.SetTitle ("notesapi.com discussion");
        // Get the ACL
        discussDB.GetACL (&acl);
        // Set the default access to "no access"
        // Save the change
        acl.Save ();
        // Close the database
        discussDB.Close()
    }
```

Accessing Views and Folders

In order to access database views and folders, an LNDatabase object provides the following functions:

Accessing a view or folder by name.

```
LNSTATUS GetViewFolder (const LNString &viewname,
LNViewFolder *view)
```

This function initializes the view object whose address is passed as argument, to the view or folder with the specified name.

Accessing all views and folders.

```
LNSTATUS GetViewFolders (LNViewFolderArray *views)
```

Test whether a particular view or folder exists.

```
LNBOOL ViewFolderExists (const LNString &viewname)
```

Accessing Documents

The LNDatebase class provides functions to create, delete, and access documents. Most of them either require an LNDocument object as parameter, or they return such an object. Again, whenever a function returns a database object, you pass the address of the object to be initialized.

Create new documents.

```
LNSTATUS CreateDocument (LNDocument *newdoc)
LNSTATUS CreateDocument (LNDocument *newdoc, const
LNString &formname)
```

The first function creates a new blank document with the default form. The new document is returned in newDoc.

Create copies of existing documents.

```
LNSTATUS CreateDocument (LNDocument &document, LNDocument
*newdoc)
```

This function takes a document and creates a new copy of it that is returned in newDoc.

Delete documents.

```
LNSTATUS DeleteDocument (LNDocument *document)
```

Get all documents in the database.

```
LNSTATUS GetDocuments (LNDocumentArray *docs)
```

This function initializes the document array whose address is passed as argument, with all documents in the database.

Searching for Documents

When you have to search for the documents you want to access, you have basically two choices. Either you use one of the following search functions as provided by a database object, or you navigate through a view or folder.

Selective search for documents.

```
LNSTATUS Search (const LNString &formula, LNNoteArray
*results)
LNSTATUS Search (const LNString &formula, LNNoteArray
*results, LNSearchOptions *options)
```

These are general search functions that allow you to search for any notes in the database, including agents, views, folders, and documents. The selection criteria are expressed by a formula string that can include any of the Notes @ functions, field names, and logical operators. With the search options, you can restrict the search to notes modified during a given period of time. Moreover, you can specify only to retrieve a certain note type such as documents, for example.

Full-Text search for documents is also available. Refer to the API User Guide for details of setting up a full-text index and performing a full-text search.

Here's an example of how we could delete all documents that were created by a particular person in a discussion database. Here's the code:

```
LNNotesSession session; // Create a session object
LNDatabase discussDB; // Create a database object
LNSearchOptions options; // Create a search options object
LNDocumentArray resultDocs; // Create a search result container
LNINT i;
LNSetThrowAllErrors(TRUE); // Enable exceptions

try
{
    session.Init();
    // Get the database and open it
    session.GetDatabase ("DISCUSS.NSF", &discussDB, "");
    discussDB.Open ();
    // Specify in the options that we are only interested
    // in notes of type documents within the specified period of time.
    options.SetNoteType (LNNOTETYPE_DOCUMENT);
    options.SetBeginDate ("01/01/2000");
    options.SetEndDate (session.GetCurrentDatetime());
    // The query is: Find all documents whose author Peter Smith.
    discussDB.Search ("@Name( [CN]; @Author) = \"Peter Miller\"", &resultDocs, &options);

    // Delete all found documents (arrays start at index 0)
    for (i = 0; i < resultDocs.GetCount(); i++)
        discussDB.DeleteDocument (&resultDocs[i]);
    // Close the database
    discussDB.Close();
}
```

CHAPTER 18

More Info on Document Objects

Once you have got an LNDocument object, you can can retrieve and modify all its properties and items.

Opening, Saving, and Closing Documents

You get a document of a Notes database either by creating it, or as a result of a database search. Creating a document implicitly opens it. In all other cases, you need to open it before you can access its properties and items. Likewise, after you have performed all changes, you need to call the Save method before closing the document. Otherwise all changes are lost.

The functions are the following:

Opening a document.

```
LNSTATUS Open (LNNOTEOPENFLAGS
flags=LNNOTEOPENFLAGS_DEFAULT )
```

In the flags, you can specify not to mark the document as read, for example.

Saving a document.

```
LNSTATUS Save (LNNOTESAVEFLAGS
flags=LNNOTESAVEFLAGS_DEFAULT )
```

In the flags, you can specify options such as delayed writing, or forced saving.

Closing a document.

```
LNSTATUS Close ()
```

Creating, Deleting, and Accessing Items and Their Values

These are the functions to retrieve items of a document, and to create new ones.

Creating new items.

```
LNSTATUS CreateItem (const LNString &name, LNItem
*newitem, LNITEMFLAGS flags = 0, LNITEMOPTIONS options
=LNITEMOPTIONS_DELETE_APPEND)
```

This function creates the new item with the specified name in the document. In the flags, you can specify whether the item shall be signed, encrypted, or protected. With the options argument, you can influence the operation behavior by specifying what action should take place when an item with this name already exists: append the item, delete the existing item first, or treat this case as an error.

```
LNSTATUS CreateItem (const LNItem &item, LNITEMOPTIONS
options =LNITEMOPTIONS_DELETE_APPEND, LNItem *newitem = 0)
LNSTATUS CreateItem (const LNString &name, const LNItem
&item, LNITEMFLAGS flags = 0, LNITEMOPTIONS options
=LNITEMOPTIONS_DELETE_APPEND, LNItem *newitem = 0 )
```

These functions copy an existing item into the document. The second version allows you to specify a name different from the source item.

Deleting items.

```
LNSTATUS DeleteItem( const LNString &name )
LNSTATUS DeleteItem( LNItem &item )
```

Retrieving existing items.

```
LNSTATUS GetItem (const LNString &name, LNItem *item)
```

This function gets the specified item within the note. The input is a string that contains the name of the item.

```
LNINT GetItemCount ()
```

Returns the number of items within the note.

```
LNSTATUS GetItems (LNItemArray *items, LNITEMTYPE type =
LNITEMTYPE_ANY)
```

This function gets all items of a document.

Note: *Although all these declarations expect LNItem objects as input and output argument types, you will never pass such an object but rather an object of a class derived from it, such as a LNText, LNNumbers, or LNRichtext.*

Here's some code to automatically create some documents in a discussion database. You should notice how the common class types such as LNText are used to work with the actual item values. In this example, we simply copy all documents of the database, and modify their subjects.

```
LNNotesSession session; // Create a session object
LNDatabase discussDB; // Create a database object
LNDocumentArray documents; // Create a document container
LNINT i;
LNSetThrowAllErrors(TRUE); // Enable exceptions
try
{
    session.Init();
    // Get the database and open it
    session.GetDatabase ("DISCUSS.NSF", &discussDB, "");
    discussDB.Open ();
    // Get all documents in the database
    discussDB.GetDocuments (&documents);
    // For all these documents:
    // first, create copies of them in the database,
    // and then modify their subject item
    for (i = 0; i < documents.GetCount(); i++)
    {
        LNDocument srcDoc = documents[i];
        LNDocument newDoc;
        LNText subjectItem;
        LNString newSubject;
        // Create a new copy of the i-th document in the database
        // *** DO NOT USE THE COPY CONSTRUCTOR! ***
        discussDB.CreateDocument (srcDoc, &newDoc);
        // Get the title item of the new document
        // and change it (the index 0 of the string array!)
        newDoc.GetItem ("Subject", &subjectItem);
        newSubject = "This is a copy of: ";
        newSubject += subjectItem [0];
        subjectItem[0] = newSubject;
        // Save the change
        newDoc.Save ();
    }
    // Close the database
    discussDB.Close();
}
```

Working With Response Documents

Many databases make use of response documents to create relationships between documents. For example, in the sample database, you can create responses to any of the discussed topics. The C++ API provides the following functions to work with response documents:

Creating a response.

```
LNSTATUS MakeResponse (const LNDocument &parent)
```

This function makes the current document a response of the one specified in the argument.

Retrieving responses.

```
LNINT GetResponseCount ()
LNSTATUS GetResponses (LNDocumentArray *responses)
```

These functions return the number of response documents, and the response documents itself, respectively. They include only the immediate responses of the documents, not the responses to responses.

Retrieving the parent documents.

```
LNSTATUS GetParentDocument(LNDocument *parent)
```

This function gets the parent of the document if it is a response document.

CHAPTER 19

More Info on View & Folder Objects

So far, you know that views and folders are represented by the class LNViewFolder, which is a special kind of a note. Folders and views are both collections of Notes documents. They can be treated in the same way, because they share the same design properties. So, your starting point is the class LNViewFolder which represents a view of folder. The entries (rows) of such objects are represented by LNVFEntry objects.

Views can contain main documents and response documents, and they can be organized by categories and sub-categories. Using the C++ API, you can access all or some entries in a collection. Objects of the class LNViewFolder provide you with many functions to navigate within the collection. However, if you want to access the responses (or more general, the children) or even several levels of descendants of an entry in the collection, you may want to instruct the LNViewFolder object to create a navigator object of the class LNVFNavigator. Although you could also use the LNViewFolder object for that purpose, such a navigator object has the advantage that you don't lose the current position of the LNViewFolder object, and allows you to access entries at arbitrary levels. Furthermore, you can create multiple navigators to navigate through different collection areas simultaneously.

Regardless of whether you navigate through a collection with an LNViewFolder or LNVFNavigator object, the current position is represented by an LNVFEntry object. Of course, there is an exception: if the collection is empty, this entry doesn't exist. Each entry can provide you with its position in the collection, which is represented by an LNVFPosition object. You can use such objects to compare the positions of different entries in the collection. The order is defined by the order in which they appear in the view or folder.

Opening and Closing View and Folders

You create an LNViewFolder object by calling the function LNDatabase:: GetViewFolder. In order to access the collection, you then need to open it. When you don't need the collection any more, call the Close function for that object to release the associated resources.

Opening a Collection

```
LNSTATUS Open (LNVFOPENFLAGS flags =LNVFOPENFLAGS_DEFAULT)
LNSTATUS Open (LNVFOPENFLAGS flags, const LNDatabase &db )
```

```
LNSTATUS Open (LNVFOPENFLAGS flags, const LNString
&pathname, const LNString &server = "" )
```

When you specify a database as an argument, the view or folder is assumed to reside in that database, rather than the database containing this view/folder note.

Closing a Collection

```
LNSTATUS Close()
```

Navigational Functions

You can use an opened LNViewFolder object to navigate a view or folder in the following ways:

Go to the first or last entry in the collection.

```
LNSTATUS GotoFirst (LNVFEntry *entry = 0)
LNSTATUS GotoLast (LNVFEntry *entry = 0)
```

Go to the next or previous entry in the collection.

```
LNSTATUS GotoNext (LNVFEntry *entry = 0)
LNSTATUS GotoPrevious (LNVFEntry *entry = 0)
```

Go to a main, parent, or child document.

```
LNSTATUS GotoMain (LNVFEntry *entry = 0)
LNSTATUS GotoParent (LNVFEntry *entry = 0)
LNSTATUS GotoChild (LNVFEntry *entry = 0)
```

Go to the first or last sibling having a common parent.

```
LNSTATUS GotoFirstSibling (LNVFEntry *entry = 0)
LNSTATUS GotoLastSibling (LNVFEntry *entry = 0)
```

Note: *A sibling entry is one that has the same parent as the current entry. There are even more functions to go to the next or previous category, non-category, main-topic, parent document, or sibling document. Furthermore, there is an extensive set of selective and full-text search functions for collection objects available. Please refer to the API User Guide for details. All these functions affect the current position of the LNViewFolder object. If you pass the address of a collection entry object as argument, it is initialized with the current entry.*

Caution

Not all navigation functions are valid in all positions of the LNViewFolder object. Whenever a next or previous entry cannot be found, the warning status

LNWARN_NOT_FOUND is returned. So, you should test the return value of these functions.

The current position of an LNViewFolder object is represented by an LNVFPosition object. Using the following functions, you can retrieve it, and you can use it to set the collection object to a particular position:

```
LNSTATUS GetPosition (LNVFPosition *position)
LNSTATUS SetPosition (const LNVFPosition &position)
```

As mentioned previously, you can also create a new independent navigator object. An LNViewFolder object provides the functions to initialize a navigator object with all collection entries, the immediate children, or all descendants of the current entry:

```
LNSTATUS GetEntries (LNVFNavigator *navigator)
LNSTATUS GetChildren (LNVFNavigator *navigator)
LNSTATUS GetDescendants (LNVFNavigator *navigator)
```

Caution If the collection is a view, its contents are recomputed during object construction. This means that the original LNViewFolder object might contain different entries than the new navigator.

Accessing Collection Entries

Once you have positioned the LNViewFolder or LNVFNavigator object at an appropriate entry, you can retrieve its type, specific column values of that entry, and the complete document associated with it.

Accessing a column either by position number or by name.

```
LNItem LNVFEntry::operator [] (LNINT n)
LNItem LNVFEntry::operator [] (LNString name)
```

Some restrictions apply: you cannot access hidden columns; they are not counted for determining the position number. Some more restrictions apply to categorized columns; refer to the API Reference Guide for details.

Accessing the entire document.

```
LNSTATUS GetDocument (LNDocument *document)
```

Retrieving the entry type.

```
LNBOOL IsCategory ()
LNBOOL IsMainTopic ()
LNBOOL IsResponse ()
LNBOOL IsTotal ()
```

The following example uses LNDatabase::GetViewFolder to select a view, opens the view, and uses the navigator function LNViewFolder::GotoNextMain to move through

the main-topic entries. LNVFEntry::GetDescendantCount returns the number of responses for each main entry. The example counts the total number of responses and displays the result:

```
try
{
   LNNUMBER          avgresponses
   LNINT             maintopics = 0,
                     totalresponses = 0;
   LNDatabase        db;
   LNViewFolder      view;
   LNVFEntry         mainentry;

   session.Init();   // Initialize the API

   session.GetDatabase("discuss.nsf", &db, "");
    // Get "Discussion View" from the database and open the view
   db.GetViewFolder("Discussion View", &view);
   view.Open();

    // The view position is set to the first category or entry when
    // the view opens. If the view has categories, go to
    // a main entry; otherwise, we are already at a main entry.

   if (view.IsCategorized())
      view.GotoNextMain(&mainentry);
   else
      view.GetEntry(&mainentry);

   maintopics = mainentry.GetSiblingCount();   // The sibling count includes mainEntry

    // Loop through main topic entries. Count the number of
    // descendants (that is, responses, responses-to-response, etc.)
    // for each main topic. Continue while there are no warnings or errors.
   do
      totalresponses += mainentry.GetDescendantCount();

   while(view.GotoNextMain(&mainentry) == LNNOERROR)

    // Display number of main topics
   if (maintopics > 0)
      cout << "Number of Main Topics: " << maintopics << endl;

   view.Close();      // Close view
   db.Close();        // Close database
}
```

CHAPTER 20

More Info on the Rich Text Object

Richtext items are the most powerful items that a Notes document can contain. They form a single big container that can store stylized text, tables, document links, OLE objects, file attachments, bitmaps, and a combination of all of these. The richness of these items requires a finer grained structure for its representation than the class LNRichText provides. You will need to know about this representation, because it defines how you access individual elements in a richtext item.

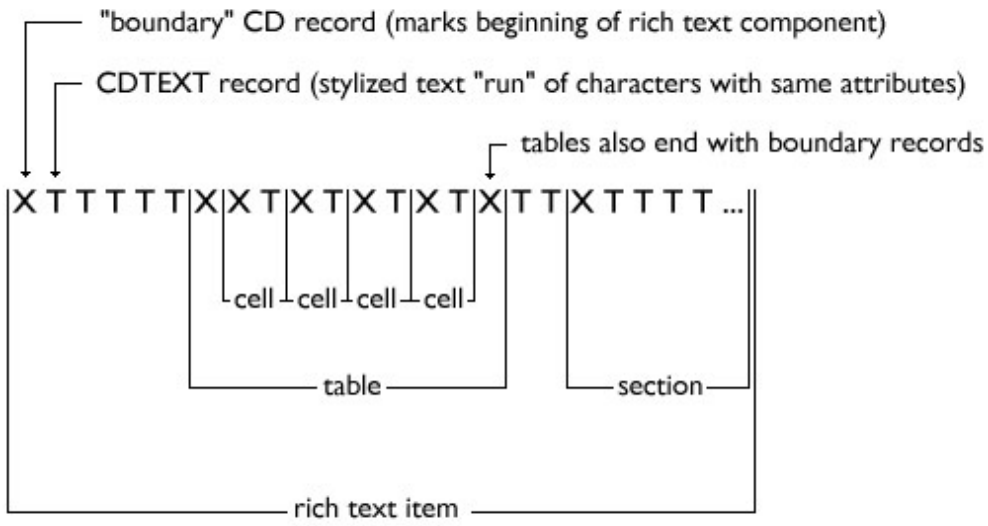
The C++ Representation of Richtext Items

Lotus Notes uses rich text items to store a variety of components, including text, tables, document links, bitmaps, and OLE links. The C++ API uses the LNRichText class to represent rich text items in a note.

A rich text item is simply a series of composite data records (CD records) that define the item's components-text, graphics, tables, links, and so on. There are many kinds of CD records; some contain data, such as text, while others contain meta-data(data that describes other data), such as the length of data in a following CD record.

It may take several CD records to define a single rich text component. For example, a table in a rich text item actually comprises a series of CD records, including begin and end records for the entire table, begin records for each cell in the table, individual CD records for each piece of stylized text in each table cell, and so on.

A rich text item might look something like this:



If you had to deal with all these CD records individually, it would take a lot of work to define or modify an object like a table. The C++ API hides much of this complexity by defining classes that represent rich text objects like tables and stylized text. When you use a class to represent a rich text object, you do not need to worry about the individual CD records that define it. The C++ API takes care of creating and placing meta-data CD records, while you concentrate on working with actual rich text data.

The C++ API visualizes some rich text components (such as sections and tables) as “containers,” because they contain data that you can navigate through, and other components as “elements,” because you cannot step through their data. For example, runs of stylized text characters (sequences of characters that share the same style attributes) are actually stored in a rich text item as a single CDTEXT composite data record. However, since you can step through the characters in a stylized text run, the C++ API views it as a container and provides a container class (LNStylizedText) to represent it.

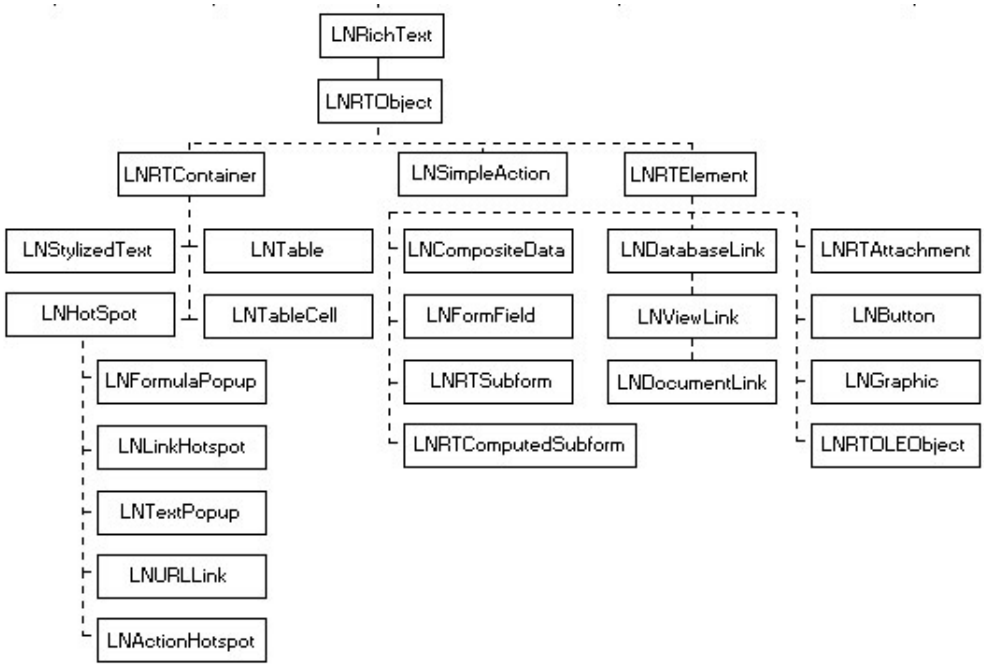
The C++ API regards some supported rich text components as elements (for example, database links) and provides element classes to represent them (LNDatabaseLink). You also use an element class (LNCompositeData) to represent any CD record that defines all or part of an unsupported rich text component.

Whether a component is represented as a container or an element, for navigational purposes it is still simply one or more CD records in the stream of records that make up a rich text item. To access different points in that stream, you use a cursor associated with the item. For convenience, the C++ API skips over CD records that you don’t need to worry about (such as paragraph breaks), but essentially you simply use Goto functions to move from one component to another and from point to point within containers. When you add a new component to a rich text item, the API inserts the necessary CD records into the stream at the appropriate location.

The LNRichText class maintains the current position in the container by an object of

the class LNRTCursor. You use this cursor object like a cursor of a text editor: it supplies functions to move to the next, previous, first, and to the last object in the richtext item.

The following figure shows the complete class hierarchy for richtext items:



Navigating Through a Richtext Item

Given a richtext item, you call one of the following functions to obtain a cursor for that item:

```
LNSTATUS GetCursor (LNRTCursor *cursor)
LNSTATUS GetEndCursor (LNRTCursor *cursor)
```

The first function returns a cursor that points to the first position in the item; the second form returns a cursor that points to the end of the last position.

Using this cursor object, you can now navigate through the container sequence:

```
LNSTATUS GotoFirst (LNRTTYPE type, LNRTObject *object)
LNSTATUS GotoLast (LNRTTYPE type, LNRTObject *object)
LNSTATUS GotoNext (LNRTTYPE type, LNRTObject *object)
LNSTATUS GotoPrevious (LNRTTYPE type, LNRTObject *object)
```

The first argument for all these functions is the type of container you want to go to. You can specify one of the following values:

LNRTTYPE_OBJECT	LNRTTYPE_NAMES_FIELD
LNRTTYPE_BITMAP	LNRTTYPE_AUTHORS_FIELD
LNRTTYPE_CONTAINER	LNRTTYPE_READERS_FIELD
LNRTTYPE_COMPOSITE_CONTAINER	LNRTTYPE_NUMBER_FIELD
LNRTTYPE_HOTSPOT	LNRTTYPE_PASSWORD_FIELD
LNRTTYPE_ACTION_HOTSPOT	LNRTTYPE_RICH_TEXT_FIELD
LNRTTYPE_FORMULA_POPUP	LNRTTYPE_TEXT_FIELD
LNRTTYPE_LINK_HOTSPOT	LNRTTYPE_TIME_FIELD
LNRTTYPE_TEXT_POPUP	LNRTTYPE_GRAPHIC
LNRTTYPE_URL_LINK	LNRTTYPE_LINK
LNRTTYPE_ITEM	LNRTTYPE_ANCHOR_LINK
LNRTTYPE_SECTION	LNRTTYPE_DATABASE_LINK
LNRTTYPE_STYLIZED_TEXT	LNRTTYPE_DOCUMENT_LINK
LNRTTYPE_TABLE	LNRTTYPE_VIEW_LINK
LNRTTYPE_TABLE_CELL	LNRTTYPE_OLE_OBJECT
LNRTTYPE_ELEMENT	LNRTTYPE_METAFILE
LNRTTYPE_ACTIVE_OBJECT	LNRTTYPE_CGM_METAFILE
LNRTTYPE_JAVA_APPLET	LNRTTYPE_MAC_METAFILE
LNRTTYPE_ANCHOR_LOCATION	LNRTTYPE_PM_METAFILE
LNRTTYPE_SUBFORM	LNRTTYPE_WIN_METAFILE
LNRTTYPE_COMPUTED_SUBFORM	LNRTTYPE_BITMAP_COLOR_TABLE
LNRTTYPE_BUTTON	LNRTTYPE_BITMAP_PATTERN_TABLE
LNRTTYPE_COMPOSITE_DATA	LNRTTYPE_BITMAP_SEGMENT
LNRTTYPE_EMBEDDED_OBJECT	LNRTTYPE_BITMAP_TRANSPARENCY_TABLE
LNRTTYPE_ATTACHMENT	LNRTTYPE_MAC_METAFILE_SEGMENT
LNRTTYPE_FORM_FIELD	LNRTTYPE_NEW_PARAGRAPH
LNRTTYPE_FORMULA_FIELD	LNRTTYPE_PM_METAFILE_SEGMENT
LNRTTYPE_KEYWORDS_FIELD	LNRTTYPE_WIN_METAFILE_SEGMENT

The second argument to these functions is an optional output argument by which you access the richtext object at the new position.

There are many more navigation functions. For example, you can position the cursor by searching for a text string.

Accessing and Modifying Richtext Items

In the simplest form, you simply append some text or another richtext item to the richtext item using the LNRichText::Append function.

When you want to insert or delete some text or a composite object at a particular position in the item, do the following:

1. Use the navigation functions to position the cursor of the LNRichText object.

2. Retrieve the cursor object.

3. Call either the LNRichText::Insert, or the LNRichText::Delete function.

To give you a simple example, here's a program that opens the sample discussion

database, retrieves the first document, and then inserts a stylized text at the second position:

```
LNNotesSession session; // a session object
LNDatabase discussDB; // a database object
LNDocumentArray documents; // a document container
LNDocument doc; // a document
LNRichText rtItem; // a richtext item
LNRTCursor rtCursor; // a richtext cursor

try
{
    session.Init();
    // Get the database and open it
    session.GetDatabase ("DISCUSS.NSF", &discussDB, "");
    discussDB.Open ();
    // Get all documents of the database
    discussDB.GetDocuments (&documents);
    // Retrieve the first document and open it
    doc = documents[0];
    doc.Open ();
    // Access the Body richtext item
    doc.GetItem ("Body", &rtItem);
    // Get a cursor for it
    rtItem.GetCursor (&rtCursor);
    // Skip the first two richtext objects:
    rtCursor += 2;
    // equivalent to:
    // rtCursor.GotoNext (LNRTTYPE_OBJECT);
    // rtCursor.GotoNext (LNRTTYPE_OBJECT);
    // Now insert a new text
    rtItem.Insert ("INSERTED TEXT", &rtCursor);
    // save the changes
    doc.Save ();
    // Close the database
    discussDB.Close();
}
```

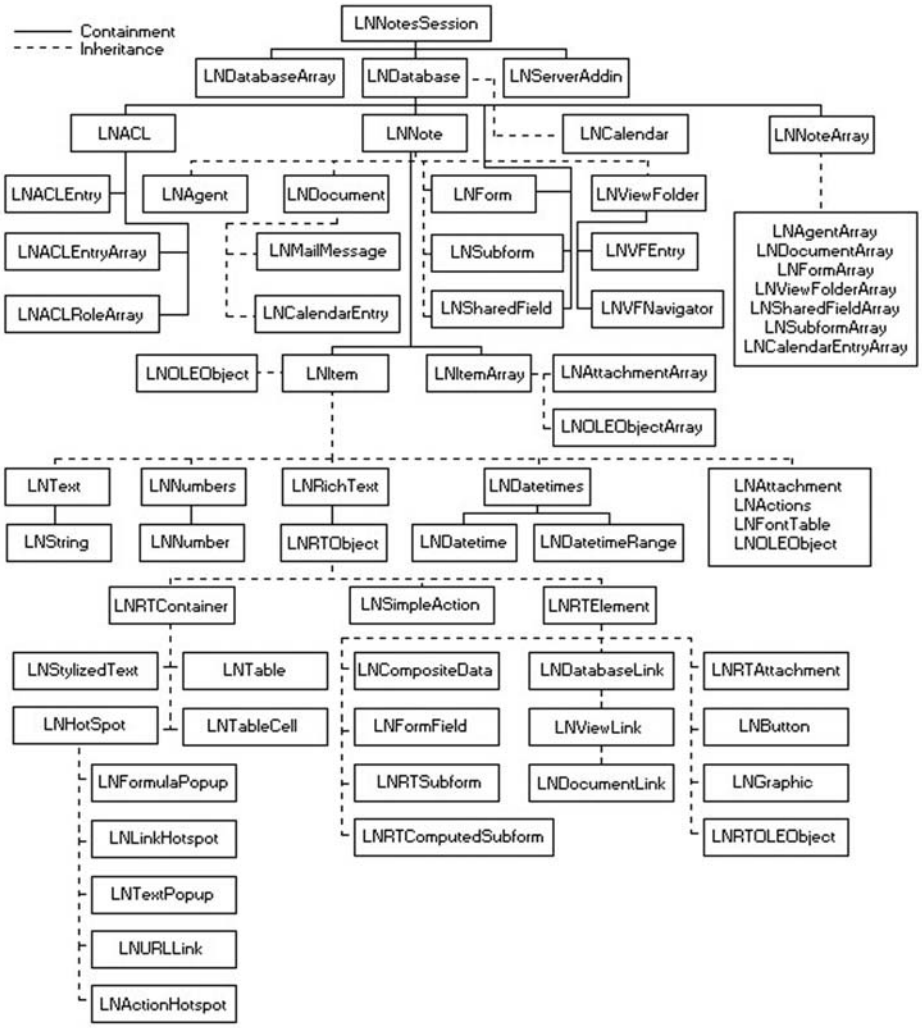
APPENDIX A

C++ API Object Model Architecture

Here is the C++ object model architecture taken directly from the C++ Toolkit documentation.

The C++ API architecture is based on a conceptual containment model, where the containment model defines an object's scope. A container always creates objects it contains. For example, an LNNotesSession object creates LNDatabase objects, and an LNDatabase object creates LNDocument objects.

The following figure shows a combined inheritance and containment hierarchy for commonly-used classes in the C++ API:



Closing a container object automatically closes all contained objects. For example, if you create a database object, use a member function to create a document, and then close the database, the document also closes. Consequently, you should be careful to save changes to contained objects before closing the object that contains them.

Note: *If you don't close a container object (for example, LNViewFolder, LNDatabase, LNNote, LNDocument, or LNAgent) before it goes out of scope, it closes automatically at some point. However, for efficiency, you should close resources as soon as you are finished with them. That way, you can be sure that memory associated with objects is freed immediately.*

APPENDIX B

C++ API Built In Data Types

You are encouraged to use these data types instead of the standard C++ data types as they do provide you with cross-platform portability where standard data types may differ in memory size from platform to platform. The following brief explanations are contained in a similar fashion in the API user guide.

LNBOOL

A C++ API boolean value that is true (non-zero) or false (0). Use the constants TRUE and FALSE to set a boolean value. When checking returned LNBOOL values, note that a false value is always equal to FALSE, or 0, but a returned true value is only guaranteed to be non-zero. It may not equal the constant value TRUE, so you should not assume that it does. The size of an LNBOOL is platform-dependent.

LNCHAR

A LMBCS character in a byte stream. LNCHAR * identifies a stream of LMBCS characters, while char * represents a stream of characters in the platform's native character set. LNCHAR is used mainly for internal functions. The C++ API typically uses LNString objects to represent LMBCS strings.

LNINT

A C++ API 32-bit integer. Unlike an int, LNINT is both fixed-size and unsigned.

LNNUMBER

On OS/390, LNNUMBER is defined as a double, which represents an OS/390 floating-point number. On all other Notes platforms, LNNUMBER is defined as a C++ API 64-bit IEEE real number (a double).

LNSINT

A C++ API 32-bit signed integer.

LNSTATUS

A 32-bit error status code returned by C++ API functions. A non-zero value indicates an error or warning condition; a value of 0 (for example, LNNOERROR) indicates success. You can get the error message for a non-zero LNSTATUS value using the global LNGetErrorMessage function.

The C++ API has several classes that represent and deal with certain data types.

LNNumber

A class that represents an LNNUMBER and provides functions to convert strings into numbers and vice versa.

LNNumbers

A class that represents a collection of number values, which might be stored in a note item. It provides you with functions to copy such arrays, and to insert, append, update and delete members.

LNString

A class that represents a null-terminated LMBCS string. It contains functions for character, substring and word handling as well as operators to compare strings and extract characters.

LNText

A class that represents a collection of string values, which might be stored in a note item. It provides you with the same functionality for array handling as the class LNNumbers does.

LNRichText

A class that represents rich text, which might be stored in a note. It provides you with a whole host of functions for manipulating rich text.

LNAttachment

A class that represents a file attachment appended to a note. It provides you with functions for manipulating attachments.

LNDatetime

A class that contains date and time information. It provides you with various functions to retrieve and set individual date and time components.

LNDatetimes

A class that represents a collection of date/time or date/time range values, which might be stored in a date/time item in a note.

LNDatetimeRange

A class that represents a pair of date/time values, indicating a period of time.

LNDatetime, LNDatetimeRange, LNNumber, and LNString represent “standalone” data, not items stored in a note. Depending on how you create them, LNDatetimes, LNNumbers, and LNText represent simple collections of standalone data or collections of items stored in a note. Similarly, LNRichText can represent either rich text data unconnected with a document or a rich text item in a note.

Since they can represent items in a note, LNDatetimes, LNNumbers, LNText, and LNRichText are particularly important classes. Whenever possible, use one of these classes, rather than the LNItem class from which they all inherit, to represent an item in a note. The derived classes, used in conjunction with LNDatetime, LNDatetimeRange, LNNumber, and LNString, provide access to functions that let you easily access and manipulate data of a given type. If you use the LNItem class, you can only use its generic functions to work with item data, forcing you to work harder.

In general, use LNItem only when you are dealing with an item of an unsupported data type, or when you are performing generic processing that doesn’t depend on the type of a particular item.

APPENDIX C

Three Most Common Reasons Why Your Program Will Not Run or Compile

Here's a list of things to check if you program will not run or compile properly:

1. The most common reason is because you have not copied the **`lcppn22.dll`** into your Notes program directory. This is the API's main dll needed by all API program, it's needed on every machine your code will run on. Different versions of this dll are available depending on which version of the C++ API toolkit you are using, in this book we have been using the most up to date version of the toolkit 2.2 - **`lcppn22.dll`**.

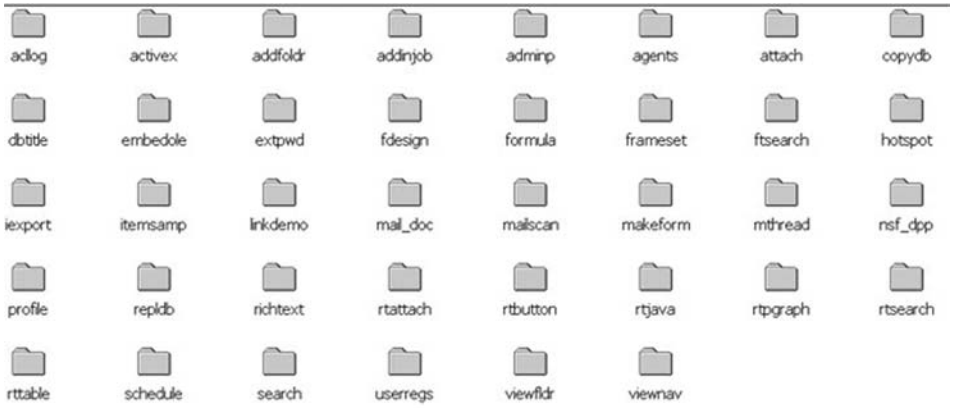
2. If you have written a dll yourself and are calling this from a Notes database, make sure you have copied this dll into the Notes program directory of the machine you want it to run on. Also be sure you have clearly entered the declarations for the dll's functions in the 'Declarations' section of your LotusScript and have entered the correct file name for the dll file and that the function name exactly matches that defined in your C++ source code for the dll function.

3. Compile errors can often be the result of not setting up your Visual Studio settings correctly. Go through the **VISUAL STUDIO SETTINGS** chapter again making sure all your settings are correct, if you do this and then still receive compilation errors, then you can be sure it's got to do with you C++ code and not your setup. A common thing I do, is forget to set the Active Configuration to Win32 release. By default Visual Studio sets this to debug mode, so if you make all the configuration changes necessary to run Notes C++ API and then change the active configuration to Win32 release, you'll notice that all your previous configuration settings have been reset as they only applied to the debug release, so you'll need to set them all over again for Win32 release. Best bet is to set the Active Configuration to Win32 release first up as outlined in the Visual Studio Settings chapter.

APPENDIX D

Invaluable Resources

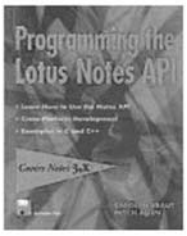
1. *C++ API Toolkit documentation is an excellent source of information. C++ Toolkit sample files stored in the NOTESCPP\SAMPLES directory, as you can see there are many examples in all facets of the API. Each directory below contains source C++ code and readme help files which you can use.*



2. *Further Notes C & C++ API teaching material - an old but solid book based on version 3x of Notes*

www.amazon.com

Programming the Lotus Notes API
by Carolyn Kraut, Mitch Allen



Seller usually ships in 1-2 business days
Paperback: 528 pages; Bk&Disk edition (May 1995) More product details

Product Details

- **Paperback:** 528 pages ; Dimensions (in inches): 1.23 x 9.20 x 7.52
- **Publisher:** John Wiley & Sons; ISBN: 0471117765; Bk&Disk edition (May 1995)
- **Average Customer Review:** ★★★☆☆

3. *C & C++ Programming Tutorials and Information*

4. *General all round Notes Domino information*

www.notes.net

5. *Excellent Domino web development resource for all who develop web based applications*

www.codestore.net

6. *Excellent, professional, well priced Notes Domino Hosting*

www.dominodeveloper.net

7. *Hard-core LotusScript snippets*

An excellent resource for direct calls to the Notes C API dll's. Many of these C function calls have been wrapped up in LotusScript classes so you can simply instantiate a new LotusScript object and call actual C API functions. This could prevent you doing any C/C++ API programming at all. There are libraries here to create Notes design elements, FTP, HTTP all sorts. Great resource and free!

http://www.greentechnologist.org/domino/LotusScript/#CAPI

8. *Notes.ini settings*

This is a really good resource. Everything you ever wanted to know about Notes.ini settings and parameters.

http://domino-8.prominic.com/A55711/ref/notesini.nsf/all!OpenView

9. *Notes Tips - another really good resource for Notes Domino developers.*

http://www.notestips.com/

10. *Domino Zone - is another professional site for Notes Domino developers, with loads of great info and links*

http://www.dominozone.net/