



Inside MFC: Handle Maps and Temporary Objects

Mindfire Solutions

www.mindfiresolutions.com

March 5, 2002

Abstract:

This paper discusses the temporary objects and the handle maps mechanism, which MFC uses to provide a uniform method for getting a C++ object from the corresponding handles, allocated in C. It also tries to explain the basic relationship between a Windows' object handle allocated by an API call and the C++ MFC object wrapping it.

INSIDE MFC: HANDLE MAPS AND TEMPORARY OBJECTS..... 1

INTRODUCTION..... 2

REFRESHING BASICS 2

- WHAT IS A HANDLE? 2
- HANDLE VS. MFC OBJECTS 3
- RELATIONSHIP BETWEEN A C++ WINDOW OBJECT AND AN HWND..... 3
- NEW/DELETE VS CREATE/DESTROY 3
- ATTACH/DETACH 3

PREPARING GROUND..... 3

- CMAPPTRTOPTR 3
- CRUNTIMECLASS 3
- CHANDLEMAP 3
- CHANDLEMAP::FROMHANDLE..... 3

TAKING THE DIVE 3

- CWnd::GetFocus() 3
- CWnd::FromHandle()..... 3
- AfxMapHWND(BOOL bCreate) 3
- WHY THE RETURNED POINTERS ARE TEMPORARY?..... 3

CONCLUSION 3



Introduction

“The returned pointer may be temporary and should not be stored for later use.”

How often have you been through this line while scrolling the MSDN pages and wondering what this temporary thing is all about? Personally speaking, it is the above line that made me dig a little deep into the MFC code to find out what exactly happens behind the scene. Consequently here in this paper, I will try to explain what the above line means, talking about various other related topics along with.

You might already be familiar with some of the concepts discussed below, but to facilitate better understanding of the main discussion they are needed here. Anyway, these are the kind of things, you can never understand too well, so why not review it again now?

Refreshing Basics

- ***What is a Handle?***

Talking of the Windows **APIs** that interacts with any graphical window on the screen, they usually take a **HWND** as their first parameter. So, what this HWND stands for? As the name suggests, HWND is a handle to something called **WND**. Well, WND is a structure keeping all the information of an onscreen window (like its location, size, title, color etc. etc.) that operating system needs to have in order to uniquely identify and manage that window. This structure is internal and private to Windows and is opaque to the programmer. You can't access this structure directly but with a call to one of the accessor APIs passing HWND as a parameter. That means to change the visibility state of the window, instead of directly changing the visible flag in its WND structure, you must make a call to *ShowWindow* API like

```
::ShowWindow(hWnd, SW_SHOW); // show the window
```

The same concept is applicable for other Windows' objects like pen, brush, font, bitmap, palette, region, menu, imagelist, socket as well. Their handles HPEN, HBRUSH, HFONT etc. are actually handle to object-specific windows-internal structures holding information relevant for that object. These handles are then used to pass as arguments to different Win32 APIs for managing the corresponding object.

From programming perspective, these handles are just unique numbers identifying the different Windows' object. The fact that they are memory handles to some specific windows-internal data structure is irrelevant. *Ignorance is bliss.*



- **Handle vs. MFC Objects**

MFC by Microsoft is available as an alternate method for programming in Windows. It provides us a C++ framework to breathe into instead of old C-based Win32 APIs. MFC is in fact a collection of C++ foundation classes wrapped around the normal Windows objects handles (HWND, HPEN etc...). So, the MFC class wrapped around HWND representing a window is **CWnd**. Not surprisingly, the corresponding MFC classes for HPEN, HMENU are named as CPen, CMenu and so on.

These MFC classes have (barring some other utility classes like **CRect**, **CPoint** etc. that do not represent any of the Windows' object) a member variable declared inside them (usually the very first member) that stores the handle to the corresponding Windows object. For example, CWnd class has a member **m_hWnd**, that holds the HWND of the window it points to. MFC then uses this handle to translate MFC calls to corresponding Win32 API function. Most of the Win32 API functions have their MFC counterpart implemented as thin wrappers, which are small inline functions having a general format as follow:

```
inline pMFCObject->foo(param1, param2, .....) {  
    ::foo(this->m_hObj, param1, param2, .....);  
}
```

That means when you intend to show the window by writing

```
pWnd->ShowWindow(SW_SHOW);
```

it is exactly same as writing:

```
::ShowWindow(pWnd->m_hWnd, SW_SHOW);
```

- **Relationship Between a C++ Window Object and an HWND**

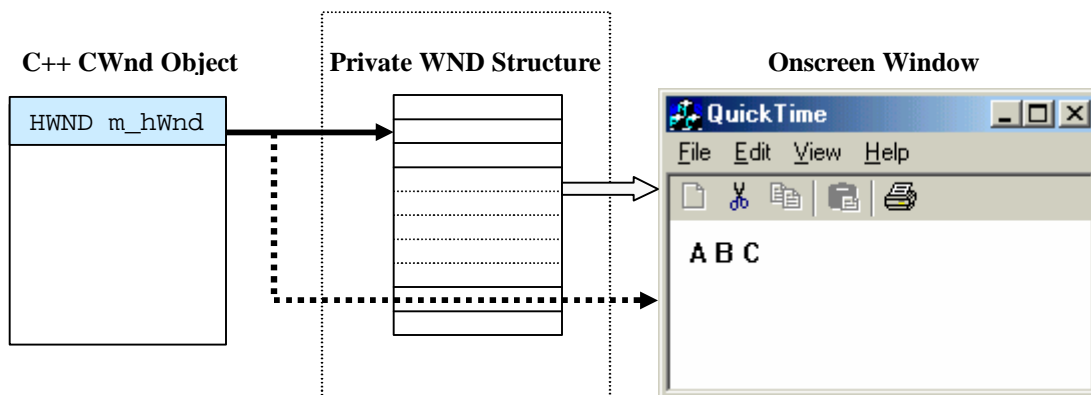




Figure 1 illustrates the relationship between various things. At leftmost we have a C++ object of class **CWnd** whose first member variable **m_hWnd** is of type **HWND**. **m_hWnd** is a handle to the Windows private **WND** structure, which further describes the onscreen window. The fact that the **WND** structure is opaque to programmer, is shown by enclosing the **WND** structure in a dotted rectangle and putting a dotted arrow directly, bypassing the private structure.

Although, the above figure specifically takes **CWnd** as an example, the same relationship exists between other MFC classes and their corresponding Windows object. Hence, a **CPen** class has **m_hPen** as its member variable of type **HPEN** pointing to the Windows pen object.

- **New/Delete vs Create/Destroy**

The above discussion should have been made the difference clear, between the window that lives on the screen and the C++ **CWnd** object that your program creates. Normally you create a C++ object either locally at stack or by explicitly allocating it on memory heap using **new** operator.

```
{
...
// CWnd object on stack which has life and scope
// with in the enclosing braces only
CWnd wnd;
...
}
or
{
...
// Create the CWnd object on heap that stays there till
// it is explicitly deleted using delete operator
CWnd* pWnd = new CWnd;
...
}
```

Whatever way from the above, you adopt to create an object of **CWnd** class (or a derived class), the fact remains same that it is just a plain C++ object in the memory and the Windows window doesn't come free with the above lines of code. So, what do you have to do to actually get the Windows window on the screen? Fortunately, you don't have to write too many lines of code for this. **CWnd** class provides you a member function called **Create** (or **CreateEx**), which actually creates the Windows window object by making a call to the **::CreateWindow** (or **::CreateWindowEx**) Win32 API and then stores the handle of the created window to its member variable **m_hWnd**.



The code will look something like following:

```
{
...
// Create the CWnd object on heap that stays there till
// it is explicitly deleted using delete operator
CWnd* pWnd = new CWnd;

// oops! Following line will fail, because pWnd doesn't have any
// Windows window object associated yet (its m_hWnd is NULL)
// pWnd->ShowWindow(SW_HIDE);

// We have got an C++ CWnd object, now try getting Windows window
// Call to Create will actually call the underlying Win32
// ::CreateWindow API and then stores the handle of the created
// window into its member variable m_hWnd
pWnd->Create(/*parameters specifying windows properties*/);
....
....
// It's ok to call the following line as now pWnd also represents
// the Windows window created above. The following line will
// use CWnd::m_hWnd to call the ::ShowWindow(m_hWnd, SW_HIDE)
// Win32 API
pWnd->ShowWindow(SW_HIDE);
....
....
}
```

Now what happens if the C++ CWnd object goes from the memory by either going out of scope when created on stack or by explicitly being deleted from the heap by a call to **delete** operator. Theoretically nothing should happen to the Windows object associated with the CWnd object with the deletion of the object. After all, the window was not created automatically when created the C++ object so why should it go automatically when the object goes from existence. So, even if the associated MFC CWnd object is no more there, the Windows window will still be there consuming system resources. To get rid of this window object we need to call **CWnd::DestroyWindow()** which ultimately calls the **::DestroyWindow(m_hWnd)**. DestroyWindow actually destroys the window object reclaiming the system resources occupied by it. So the normal flow the program goes like following:

```
{
// Create the MFC object
CWnd* pWnd = new CWnd;

// Create the Windows window and associate it to this CWnd object
pWnd->Create(.....);

// Interact with the windows
.....
}
```



```
.....  
.....  
  
// Get rid of the Windows  
pWnd->DestroyWindow()  
  
// Finally delete the MFC CWnd object  
delete pWnd;  
}
```

The above-created windows can also be implicitly destroyed in response to some user actions like clicking at top-right cross-box or pressing Alt+F4 (which anyway results into a call to DestroyWindow()). As a note, CWnd destructor does check for the m_hWnd and if it finds a Windows window still attached to that CWnd object then it makes a call to DestroyWindow(), so that the onscreen window will be automatically destroyed if the MFC object is deleted. But it is always good to destroy the window much before the destructor of the CWnd object.

• **Attach/Detach**

We just saw how we can create a window object by calling the Create member function of the CWnd class. But what if we have somehow magically acquired HWND of already existing window object created by someone else and we want our CWnd object to represent to that window? The answer is surprisingly simple. Since, we know that the MFC CWnd object is linked with the Windows window object by its member variable m_hWnd, if we just store the magically acquired handle to our CWnd object's m_hWnd variable then we are done with it. That means that we can have our CWnd object wrapped around the handle of window not created by us. **CWnd::Attach()** function does exactly that for us.

```
{  
// Create the MFC object  
CWnd* pWnd = new CWnd;  
  
// Acquire handle to an already existing window  
HWND hWnd = AcquireHandleToSomeOtherWindow();  
  
// Attach the handle to our MFC object  
pWnd->Attach(hWnd);  
  
// Its ok to call the following line as now pWnd represents  
// the window whose handle we acquired somehow.  
pWnd->ShowWindow(SW_HIDE);  
  
...  
}
```



The corresponding function for detaching the window's handle from an existing CWnd object is **CWnd::Detach()**

```
{
// Create the MFC object
CWnd* pWnd = new CWnd;

// Create the Windows window and associate it to this CWnd object
pWnd->Create(....);

// Do something with the windows
....

// Detach the handle from this CWnd object
// Store the handle for whatever you want to do with it
HWND hWnd = pWnd->Detach();

// Delete the MFC object
// Its Ok to delete the MFC object and it won't destroy the
// Windows' window, since it has already been detached above and
// the member variable pWnd->m_hWnd is set to NULL
delete pWnd;

// Hide the above window using the stored handle
// Its OK to call like this even if the pWnd is gone now.
// Remember CWnd object and HWND are different?
::ShowWindow(hWnd, SW_HIDE);
}
```

Needless to say that all the above concepts of Create/Destroy or Attach/Detach are applicable to other Windows objects like CPen, CBrush & CMenu etc. as well. Like, CPen class has **CreatePen** and **DeleteObject** member functions for creating and destroying the pen object respectively and so on.

Preparing Ground

What follows next is a brief discussion of some of the keywords/terms, which will be used in dealing with our main topic. Understanding these topics is important, as they will largely act as building blocks for our main discussion.

Instead of explaining these concepts in place, as and when they are introduced, I am telling you about them here well in advance, so that you are not caught off guard with so many new terms thrown at you later on.



Knowing actual implementation and functioning of the following concepts are not necessary/relevant to our discussion. You just need to be familiar with them. In fact, I could have made you life easier by using some pseudo names/implementations to explain the things, but I decided otherwise. Largely because, I don't want you to feel lost in the unfamiliar territory of the MFC's source code (I will be listing the corresponding file names along with), if you ever decided to dig into them yourself. *Good Luck.*

- **CMapPtrToPtr**

CMapPtrToPtr is one of the standard MFC collection classes. MFC provides three basic collection shapes namely, lists, arrays & maps. The lists class provides an ordered, non-indexed list of elements (implemented as doubly linked list). The array class provides a dynamically sized, ordered and integer-indexed array of objects. Maps, also known as a dictionary, is a collection that maps a key object with a value object. These three are conceptually same as C++ STL's **list**, **vector** and **map**. In fact, MFC has two types of collection classes: collection classes created from C++ templates and collection classes not created from C++ templates.

CMapPtrToPtr is a non-template based dictionary class that maps a unique void pointer to another void pointer. It provides the basic functionality of setting, iterating, deleting the key-value pairs in the map. Because CMapPtrToPtr uses a hash table for internally storing the data, lookup is very fast (for the same reason, if you iterate through the map, it is not guaranteed to be sequential by key value, rather the sequence of retrieved elements is indeterminate).

The two main member functions for managing the map are:

```
// Looks up a void pointer based on the void pointer key.  
BOOL CMapPtrToPtr::Lookup(void* key, void*& rValue) const;  
  
// Inserts an element into the map; replaces an existing element  
// if a matching key is found.  
void CMapPtrToPtr::SetAt(void* key, void* newValue);
```

MFC File(s): MFC\Include\AfxColl.h, MFC\Include\AfxColl.inl & MFC\Src\Map_pp.cpp

- **CRunTimeClass**

Each **CObject** class (or any other class derived from CObject) is associated with a **CRunTimeClass** structure (yes, names can be confusing, it's actually a structure not derived from any other class or structure). CRunTimeClass structure can be used in obtaining information about an object or its base class at run time.



To use `CRuntimeClass` structure in your derived class whose runtime information you want to have, you generally include the **IMPLEMENT_DYNAMIC**, **IMPLEMENT_DYNACREATE** or **IMPLEMENT_SERIAL** macro provided by MFC in the implementation of the class. Although, actual implementation detail of `CRuntimeClass` and the above macros and the concept of dynamic object creation is a complete topic in itself, I will just try to explain a few members of `CRuntimeClass` structure that are relevant to our discussion.

```
// A function pointer to the default constructor that creates an
// object of the class if it supports dynamic creation, otherwise
// NULL
CObject* PASCAL* m_pfnCreateObject( )

// If class derived from CObject supports dynamic creation, i.e.
// the ability to create an object of a specified type at run
// time, then the CreateObject member function can be used to
// implement the creation of objects of that class.
CObject* CreateObject( );
```

MFC File(s): MFC\Include\Afx.h

This function merely calls the function pointed by the **m_pfnCreateObject** member variable and is defined as:

```
CObject* CRuntimeClass::CreateObject() {
    if (m_pfnCreateObject == NULL) {
        TRACE(_T("Error:..."));
        return NULL;
    }

    CObject* pObject = NULL;
    TRY {
        pObject = (*m_pfnCreateObject)();
    }
    END_TRY
    return pObject;
}
```

MFC File(s): MFC\Src\Objcore.cpp

Although this function returns a pointer to `CObject` class, the created object is actually of your derived class type, since, `m_pfnCreateObject` points to **YourClass::CreateObject** function of your class which create an object of actual class type.



- **CHandleMap**

This is the key class on which most of our discussion will rest upon. It is basically used to implement the mapping mechanism of Windows object handles to its corresponding MFC wrapper class pointers. It manages two dictionaries internally (implemented as CMapPtrToPtr) to keep track of handle-pointer pair mapping. The two maps are purposefully named as **m_permanentMap** and **m_temporaryMap** (detailed discussion in next section). It also has a pointer **m_pClass** of type CRuntimeClass pointing to the run time class associated with that MFC wrapper class.

The **m_nOffset** member keeps the byte offset of the handles in the MFC object (remember, each MFC wrapper class has a handle to the corresponding Windows object as its first member). The **m_nHandles** keeps the number of handles that MFC class has. For most of the MFC wrapper classes it is equal to 1, but for some as **CDC** it is equal 2 (CDC has two types of **HDC** associated, **m_hDC** and **m_hAttribeDC**).

Looking at the class definition of CHandleMap:

```
class CHandleMap
{
private:    // implementation
    CMapPtrToPtr m_permanentMap;
    CMapPtrToPtr m_temporaryMap;
    CRuntimeClass* m_pClass;
    size_t m_nOffset;    // offset of handles in the object
    int m_nHandles;    // 1 or 2 (for CDC)
    ...
    ...
    CHandleMap(CRuntimeClass* pClass, size_t nOffset,
               int nHandles = 1);

    CObject* LookupPermanent(HANDLE h);
    CObject* LookupTemporary(HANDLE h);

    void SetPermanent(HANDLE h, CObject* permOb);
    void RemoveHandle(HANDLE h);

    CObject* FromHandle(HANDLE h);
    void DeleteTemp();
}
```

MFC File(s): MFC\Src\Winhand_.h

Out of the member functions listed here, **LookupPermanent** and **LookupTemporary** looks up the corresponding dictionaries (m_permanentMap & m_temporaryMap) for the specified handle and returns the corresponding MFC object pointer paired to that handle.



If they do not find any entry for the specified handle in the respective map then they return NULL.

The Constructor takes three parameters for initializing the three corresponding member variables. These three member variables `m_pClass`, `m_nOffset` & `m_nHandles` can be set at the creation time only by the constructor and you can't change them explicitly afterwards.

SetPermanent member, as the name suggests, makes an entry for the specified handle-object pair into the permanent dictionary (`m_permanentMap`). The next function **RemoveHandle** removes the handle-key pair from the permanent dictionary. The point to notice is that the class does not have any corresponding member functions for setting or removing the key-value pair in the temporary map. *Come next and you will know why.*

• ***CHandleMap::FromHandle***

This member function is the main interface to the outside world provided by the `CHandleMap` class. It actually sums up the functionality of `CHandleMap` class and is important enough to deserve a separate section.

The reason for maintaining two separate dictionaries (temporary and permanent) will be clear here. You will also come to know about the meaning of other member variables, as they will be used in this routine.

The source code of the above function after stripping out some memory and error handling code and adding some helpful comments is following:

```
CObject* CHandleMap::FromHandle(HANDLE h)
{
    if (h == NULL) return NULL; // Invalid handle, return

    // Lookup in the permanent map first
    CObject* pObject = LookupPermanent(h);
    if (pObject != NULL)
        return pObject; // return permanent one, and be glad
    else if ((pObject = LookupTemporary(h)) != NULL) // in temp map
    {
        // Aah! The pair is already there in the temporary map
        // So just set the handle(s) in the existing object
        // the handle is at m_nOffset bytes offset after CObject
        HANDLE* ph = (HANDLE*)((BYTE*)pTemp + m_nOffset);
        ph[0] = h;
        if (m_nHandles == 2) // Some (CDC) may have 2 handles
            ph[1] = h;

        return pObject; // return current temporary one
    }
}
```



```
// Couldn't find the handle either in permanent or temporary map
// probably, this handle wasn't created by us, so we must create
// a temporary C++ object to wrap it.

CObject* pTemp = NULL;

// Use the runtime class associated to this type of handle to
// create the object of the Wrapper class
pTemp = m_pClass->CreateObject();

// Object is temporary, so make an entry in temp map
m_temporaryMap.SetAt((LPVOID)h, pTemp);

// now set the handle in the object
HANDLE* ph = (HANDLE*)((BYTE*)pTemp + m_nOffset); // after CObject
ph[0] = h;
if (m_nHandles == 2) // Some class (CDC) may have 2 handles
    ph[1] = h;

return pTemp; // return the created temporary object
}
```

MFC File(s): MFC\Src\Winhand.cpp

The function basically returns a pointer to MFC object, which wraps around the passed handle. This function is guaranteed to return an object pointer for the given handle if it is valid. If there is any MFC object already wrapped around the said handle, then it simply returns it, otherwise it goes on creating an object of the class type represented by the `m_pClass` which was specified while constructing the `CHandleMap` object. It then keeps an entry of the created object in its temporary map and if next time it encounters the same handle and able to find its entry in the temporary map then it simply returns the object from the temporary map which it had created last time

The flow goes like this:

1. If the handle entry is already there in permanent map, then just return it.
2. Otherwise try locating it in the temporary map
3. If found, just return it after setting the passed handle in the found object. This is done by using the information that `m_nOffset` is the byte offset of the member variable representing the handle of the underlying Windows object within the MFC wrapper class. Set the other handle also if the number of handle is 2 (`m_nHandles`), as in CDC.
4. If not able to find the entry either in permanent or temporary map then create one by calling the **CreateObject** function of the run time class pointed by the `m_pClass` member variable. `CreateObject` returns the pointer of the created MFC wrapper object.
5. Make an entry for given handle and the newly created object pair into the temporary map.



6. Return the newly created MFC object after setting the handle(s) as mentioned in step 3.

As you can see that you can set the handle-object pair into the permanent map or remove them from it from outside (SetPermanent and RemoveHandle). Whereas, the temporary map, has the entries for only those handle-object pairs, whose object is created implicitly by the class in the above function. To delete these pairs from the temporary map, the class has **DeleteTemp** function that is different in the sense that it doesn't have any option for deleting individual elements from the dictionary as in case of permanent map (RemoveHandle).

DeleteTemp function, in fact, goes on deleting all the objects stored in the temporary map, but not before zeroing out all the handles in those objects. This is necessary; otherwise the corresponding MFC class object's destructor might try to destroy the Windows' object represented by that handle (*Refreshing Basics: New/Delete vs. Create/Destroy*). This shouldn't happen as we did not create those handles, but just the temporary wrapper objects around them. It also resets the temporary dictionary by removing all the entries from m_temporaryMap.

The class destructor calls DeleteTemp() function when the object goes, but you will see some other instances of DeleteTemp being called also, when we come to see the actual use of this class in our context.

Taking The Dive

Armed with the above details, let us go back to our original discussion. But do you really remember what we were discussing? In case, you don't, I started this article with the following line.

“The returned pointer may be temporary and should not be stored for later use.”

So, what I will do is to take an example function whose documentation includes the above line, and then walk inside that function to find out what magic it actually does to return us a pointer, which is temporary. And in the process you will find references to the terms/concepts what we have already discussed in the above section (so in case you are still confused about the above stuffs, then go on revising them once again before proceeding)



- **CWnd::GetFocus()**

Suppose you want to know which window is currently in focus and then do whatever stunts you want to do with that window. To get the currently focused windows you made a call to **CWnd::GetFocus()** like:

```
CWnd* pFocusWnd = CWnd::GetFocus();
```

The above function will returns a pointer to the window that has the current focus, or NULL if there is no focus window. Now, if at the time when you called this function your main app window was in focus, then this will return you a pointer to your main window, which you might have created at your application startup. But wait, what if you are not so lucky and at the time of you calling this function, some other window created by *god knows whom*, was in focus? Will you still get a valid CWnd pointer or the function will just return NULL. Answer, as you already know, is that it will still return a valid CWnd pointer that correctly represents the focused window. So how come you get a pointer to object you never created, or for that matter, the window pointed by the returned pointer may not be created using MFC at all?

To make the matter more complicated, even if the currently focused windows is the one which you explicitly created using CWnd's create function, the question is, who does all the book keeping to make sure me that you will get your window pointer back on a call to CWnd::GetFocus() if that window is currently in focus. Somebody got to keep track of the MFC object created by you as well. In fact, MFC does keep track of all the windows (or any other Windows object as discussed above) created by you.

The code for CWnd::GetFocus() is surprisingly simple.

```
_AFXWIN_INLINE CWnd* PASCAL CWnd::GetFocus()  
{ return CWnd::FromHandle(::GetFocus()); }
```

MFC File(s): MFC\Include\afxwin2.inl

Just a single line inline function calling the corresponding API **HWND ::GetFocus()**; Then it calls **CWnd::FromHandle(HWND)** passing it the handle of the focused window, which somehow returns an MFC object wrapping the passed handle. How CWnd::FromHandle manages to return an MFC object from a Windows handle is what we are interested in now.

- **CWnd::FromHandle()**

Once again, I'm simplifying the code by omitting some line from and adding more comments to the code, but essentially it looks something like this:



```
CWnd* PASCAL CWnd::FromHandle(HWND hWnd)
{
    // Get the global handle map, create if not exist
    CHandleMap* pMap = afxMapHWND(TRUE);

    // call the FromHandle function of the returned
    // CHandleMap to get the corresponding MFC object pointer
    // Remember? CHandleMap::FromHandle creates a new
    // object if it doesn't exist already
    CWnd* pWnd = (CWnd*)pMap->FromHandle(hWnd);

    return pWnd;
}
```

MFC File(s): MFC\Src\WinCore.cpp

As I had promised, you are beginning to see the use of all the stuffs we discussed in previous section (*Preparing Ground*). Here, in this function it first acquires a pointer to a CHandleMap object by calling **afxMapHWND**. Then it calls the FromHandle function of the returned CHandleMap object to get an MFC object pointer. This MFC object pointer is returned either from the permanent map or from the temporary map or a new object is created on the fly in that order. So, now our aim is to find out who manages this handle map and what afxMapHWND does.

- ***afxMapHWND(Bool bCreate)***

The global function afxMapHWND returns a pointer to the handle map, creating it if it does not exist already and bCreate parameter is true.

```
CHandleMap* PASCAL afxMapHWND(BOOL bCreate)
{
    AFX_MODULE_THREAD_STATE* pState =
        AfxGetModuleThreadState();

    if (pState->m_pmapHWND == NULL && bCreate)
        pState->m_pmapHWND = new
            CHandleMap(RUNTIME_CLASS(CTempWnd),
                offsetof(CWnd, m_hWnd));

    return pState->m_pmapHWND;
}
```

MFC File(s): MFC\Src\WinCore.cpp

The returned handle map is a member of the **AFX_MODULE_THREAD_STATE** object that is obtained by a call to **AfxGetModuleThreadState()**.

AFX_MODULE_THREAD_STATE is basically a class keeping information about the current thread state. MFC keeps a global object of this type on per thread basis. This



object is local to thread and is a member variable of AFX_MODULE_STATE, which in turns keeps track of the current module state.

```
class AFX_MODULE_THREAD_STATE : public CNoTrackObject
{
public:
    ...
    ...
    // current CWinThread pointer
    CWinThread* m_pCurrentWinThread;

    // temporary/permanent map state
    DWORD m_nTempMapLock;           // if not 0, temp maps locked
    CHandleMap* m_pmapHWND;
    CHandleMap* m_pmapHMENU;
    CHandleMap* m_pmapHDC;
    CHandleMap* m_pmapHGDIOBJ;
    CHandleMap* m_pmapHIMAGELIST;
    ...
    ...
}
```

MFC File(s): MFC\Include\Afxstat.h

So there you are, with all handle maps of the concerned Windows objects like Window, Menu, DC, GdiObject and ImageList. You can rightly guess that global functions similar to `afxMapHWND` should exist for other type of handle map also, which simply returns the corresponding member variable of the above global thread state object. They are named as **`afxMapHMENU`**, **`afxMapHDC`**, **`afxMapHGDIOBJECT`** and **`afxMapHIMAGELIST`**.

If the concerned handle map doesn't exist and the **`bCreate`** parameter is true then the function creates one before returning it. If you remember then, the `CHandleMap` constructor takes a run time class as its first parameter and the byte offset of the handle within that object as its second parameter. The third parameter is number of handles for that object which is 2 for CDC and 1 for rest (default). Now here

So the first parameter `RUNTIME_CLASS(CTempWnd)` is actually a pointer to the run time class associated with the class `CTempWnd`. This run time class information is used by `CHandleMap::FromHandle()` to create the temporary object on fly. `CTempWnd` is actually a `CWnd` with just two macros to the `CWnd` class:

```
class CTempWnd : public CWnd
{
    DECLARE_DYNCREATE(CTempWnd)
    DECLARE_FIXED_ALLOC(CTempWnd);
};
```



MFC File(s): MFC\Src\Winhand.cpp

These macros enable the CWndTemp class to support dynamic object creation at run time. For us CTempWnd is same as CWnd in all meaning. Similar wrapper classes, aptly named CTempDC, CTempMenu etc. exist for CDC, CMenu and other classes. (This should solve the mystery of VC++ debugger showing CTempWnd as the class type for the window pointer returned by functions like CWnd::GetFocus())

The second parameter is the offset of the handle within the runtime class object. **offsetof** is just a macro defined as

```
#define offsetof(s,m) (size_t)&(((s *)0)->m)
```

The third parameter, number of handles is implicitly set to 1 (default parameter) in this case. For CDC, it will be 2.

This all explains why and how you get a valid MFC's CWnd object for a window every time, no matter if you created it or not. But there is a small thing still missing. Fine, if you need to have a window pointer for a corresponding window's handle, which is not created by you, CWnd::FromHandle will call FromHandle() function of the corresponding CHandleMap to create one for you. But how do you get the pointer of the permanent object that your program has explicitly created. I mean, how its entry is made into the permanent map of the corresponding handle map. The answer is straight rather, you did it by calling CWnd::Attach explicitly or by it being called implicitly due to some other function call like CWnd::Create() (actually Create doesn't call CWnd::Attach directly, rather it is called from the hook function which is setup from within the Create)

```
BOOL CWnd::Attach(HWND hWndNew)
{
    // create map if not exist
    CHandleMap* pMap = afxMapHWND(TRUE);

    pMap->SetPermanent(m_hWnd = hWndNew, this);

    return TRUE;
}
```

MFC File(s): MFC\Src\WinCore.cpp

So CWnd::Attach along with setting the m_hWnd to the passed handle also makes an entry in the permanent handle map by calling CHandleMap::SetPermanent. Consequently, CWnd::Detach removes the entry from the permanent map and sets m_hWnd to NULL. That completes the complete cycle.



The whole process of getting an `CWnd` object by calling `CWnd::FromHandle` can be summed as:

1. Get the global thread state object by calling `AfxGetModuleThreadState()`
2. Get a pointer to the `HWND-CWnd` handle map managed by the `AFX_MODULE_THREAD_STATE` (`CHandleMap* m_pmapHWND`).
3. If the returned pointer is `NULL` (map not created yet), then go on creating one by passing the run time class pointer of the corresponding `CTempWnd` class. This run time class will be used to create the object of the class type at run time by `CHandleMap::FromHandle()`;
4. Lookup the permanent map for the specified handle entry (which might have been made due to a call to `SetPermanent`, in case this object is explicitly created). If entry found just return it. This object will be of the type you actually created. (Like if you had created a `MyWindow` object derived from `CWnd`, then the returned object will actually be a pointer to `MyWindow` object).
5. If the entry not found in permanent map, then lookup the temporary map. If entry found then set the handle(s) within the found object to link it with the specified handle(s).
6. If no entry found in temporary map also, then create an object of the type as specified by the `m_pClass` member variable at run time (which is `CTempWnd` in this case).
7. Make the entry for this newly created object in temporary map and return it after setting the handle(s) within it.

Steps 5-7 translate into an interesting fact. That, no matter what is the type of the window's handle (button, edit, list etc.) you will always get a object of type `CTempWnd` from `CWnd::FromHandle()`. This is because run time class associated with `CTempWnd` class is what you had passed in the constructor of the `CHandleMap` while creating the global handle map for windows object (you don't have different maps for edit, button etc., just one for windows, one for `GDIObjects` and so on). It means that if have a button control with an `IDC_MYBUTTON` and you wrote something like

```
CWnd* pWnd = GetDlgItem(IDC_MYBUTTON); // OK
CButton* pButton = (CButton*) pWnd; // WRONG, its not a CButton
pButton->SetCheck(0); // Should fail
```

then it should fail. The first line is still OK, because you are typecasting the returned `CTempWnd` pointer (`GetDlgItem` too uses `CWnd::FromHandle()` to return the temporary pointer) to its base class `CWnd`. But the second line you are trying to typecast an object of type `CTempWnd` `CWnd` into `CButton`, which is nowhere in picture. So, technically the above code is incorrect, even if you swear that you have used code like above all over your life and it always worked. The code worked because, **SetCheck** is a thin wrapper function which sends a **BM_SETCHECK** message using **SendMessage** API to the window pointed by `pButton->m_hWnd`. Being a true button control, it responds to the message by doing what it was expected to do. Had the `SetCheck` function been a virtual function instead of inline, calling it would result in a call to `CWnd`'s `SetCheck` because



the object is actually of type CWnd (CTempWnd to be precise). Function SetCheck is missing from CWnd, so the code is likely to crash. And if the function SetCheck uses some data member that was added to CButton, then again the code will fail even if the function is inline.

So, what's the correct way to get it done if you want to make sure that you are not left with using the brute casting as above. The answer lies in Subclassing the control. Subclassing a control mainly does two things. First, it somehow associates the control to the desired class so that the windows messages are routed to that class (Message routing is a separate topic, and deserves more space than I can give here). Secondly it attaches the control's handle to the class object by calling CWnd::Attach(), which in turns make an entry of the handle-actual object pair into the permanent map.

So you declare a member variable of type CButton in your class and then subclass it to link with the desired control

```
class CMyDialog : public CDialog {
    CButton m_Btn;
    // ...
    virtual BOOL OnInitDialog( );
}
and
BOOL CMyDialog::OnInitDialog( )
{
    // ...
    m_Btn.SubclassDlgItem(IDC_MYBUTTON, this);
    // ...
}
```

The other way is to create a CButton object temporary and then attach the returned handle to it whenever you need, like

```
{
    // Create a CButton object
    CButton* pButton = new CButton;

    // Get the handle of the control by calling ::GetDlgItem
    // API, and not CWnd::GetDlgItem
    HWND hWnd = ::GetDlgItem(this->m_hWnd, IDC_MYBUTTON);

    // Attach the handle to this object
    pButton ->Attach(hWnd);

    // its safe to use call this as pButton is an actual
    // CButton object
    pButton->SetCheck(0);

    // Detach the handle, otherwise CButton destructor will
```



```
// try to destroy the control
pButton->Detach();

// Get rid of the temporary CButton object
delete pButton;
}
```

Finally, if you want to know that whether these is any permanent object already created for a particular handle, then you can make use of **CWnd::FromHandlePermanent(HWND)**, which unlike **CWnd::FromHandle(HWND)** doesn't create any temporary object and just looks up in the permanent handle map. If it finds the entry then it returns that, otherwise it returns NULL. You can use this fact to modify the above code to create a button object only if there is no CButton object already attached to that handle.

- **Why the returned pointers are temporary?**

Although, the process of getting a MFC object (permanent and temporary) corresponding to the handle is clear now, but our question is still unanswered, that why the returned pointers should not be stored for the later use. Fine, if we didn't create that object ourselves and that object is created implicitly within **ChanldeMap::FromHandle**, but that fact alone shouldn't make the created object temporary. We should still be able to refer to the desired window object by using the pointer stored by any previous call to **CWnd::FromHandle**. But the documentation clearly discourages that and warns that the returned object may not be available next time you try to use them.

The reason that the temporary handle map is called temporary and the objects within it are temporarily available is that the temporary dictionary is periodically emptied by MFC deleting all the entries as part of its normal idle time processing. In fact, this is done from within **CWinApp::OnIdle()** which is called during every message processing loop. **CWinApp::OnIdle()** calls **AfxUnlockTempMaps** with **bDeleteTemp** parameter set to true where the temporary maps are actually deleted. The **m_nTempMapLock** member of the local thread state object does the reference counting to keep track of if temp map are still locked (Temp maps are deleted only after lock count goes down to zero (unlocked)).

```
BOOL AFXAPI AfxUnlockTempMaps(BOOL bDeleteTemps)
{
    AFX_MODULE_THREAD_STATE* pState =
        AfxGetModuleThreadState();
    if (pState->m_nTempMapLock != 0 &&
        --pState->m_nTempMapLock == 0)
    {
        if (bDeleteTemps)
        {
```



```
        //...
        //...

        // clean up temp objects
        pState->m_pmapHGDI OBJ->DeleteTemp();
        pState->m_pmapHDC->DeleteTemp();
        pState->m_pmapHMENU->DeleteTemp();
        pState->m_pmapHWND->DeleteTemp();
        pState->m_pmapHIMAGE LIST->DeleteTemp();
    }
    //...
    //...
}
//...
//...
}
```

MFC File(s): MFC\Src\Winhand.cpp

As you can see that not only the window objects, but other temporary maps are also deleted. The fact that this function is called during idle time processing (in between message processing loops), explains that the temporary pointers are typically valid during message process loop and should be used accordingly. So, if you obtained a pointer to the focused window (CWnd::GetFocus) in CWnd::OnLButtonDown, then you can use it within that message handler at that time only. Trying to use the same pointer in, say, CWnd::OnRButtonDown is likely to fail (the only exception might be in case of a modal loop (like CDialog::DoModal()) when CWinApp::OnIdle is not called periodically, but it is always safe to always avoid storing the pointer for any later use).

The temporary map can also be deleted by an explicit call to **CWnd::DeleteTemp()** (or CMenu::DeleteTemp for Menu and so on) which actually calls the corresponding **CHandleMap::DeleteTemp** function.

```
void PASCAL CWnd::DeleteTempMap()
{
    CHandleMap* pMap = AfxGetModuleThreadState()->m_pmapHWND;
    pMap->DeleteTemp();
}
```

MFC File(s): MFC\Src\Winhand.cpp

Why the temporary map is deleted? Partly to reclaim the memory, but mainly due to the fact that you might want to create a permanent object for that handle later on in your program. MFC permits only one C++ attached to a handle at one time, so MFC needs to free the temporarily created object to make sure that you can always associate any



permanent object to that handle (again once only). Using the same temporary object as the permanent one won't work because then you will be limited to use only the CWnd object (temporary objects can be of type of the run time class only, stored in the corresponding CHandlMap) in place of CDialog, CEdit, or CButton objects, which you may don't want to.



Conclusion

It is important to keep in mind that whatever things we discussed above taking CWnd/HWND as an example applies to other Windows objects like HDC, HPEN, HFONT HMENU etc. as well. You can actually browse through the MFC code to find the corresponding functions for other classes like CMenu, CImageList, CDC etc. and don't be surprised to find exactly same functions as we had discussed for CWnd.

Last but not least, instead of being clearer you might actually be in a confused state after going through all the above details (I was, for a long time), but that only proves the saying that “*more study more confusion!*” What I will suggest is to try proving it wrong by actually doing more study. This topic anyway is complicated enough to deserve that extra.

Mindfire Solutions is an IT and software services company in India, providing expert capabilities to the global market. Mindfire possesses experience in multiple platforms, and has built a strong track record of delivery. Our continued focus on fundamental strengths in computer science has led to a unique technology-led position in software services.

To explore the potential of working with Mindfire, please drop us an email at info@mindfiresolutions.com. We will be glad to talk with you.

To know more about Mindfire Solutions, please visit us on www.mindfiresolutions.com
