Appending additional payload to a PROPSHEETPAGE structure

devblogs.microsoft.com/oldnewthing/20211124-00

November 24, 2021

Raymond Chen

A not-well-known feature of the common controls property sheet is that you can append your own custom data to the end of the `PROPSHEETPAGE` structure, and the system will carry it around for you.

The traditional way of setting up a `PROPSHEETPAGE` is to use the `lParam` member to point to a structure containing additional data that is used by the property sheet:

```
struct WidgetNameData
{
    HWIDGET widget;
    bool uppercaseOnly;
    int renameCount;
};

void ShowWidgetProperties(HWIDGET widget, HWND hwndOwner)
{
    WidgetNameData nameData;
    nameData.widget = widget;
    nameData.uppercaseOnly = IsPolicyEnabled(Policy::UppercaseNames);
    nameData.renameCount = 0;

    PROPSHEETPAGE pages[1] = {};

    pages[0].dwSize = sizeof(pages[0]);
    pages[0].hInstance = g_hinstThisDll;
    pages[0].pszTemplate = MAKEINTRESOURCE(IDD_WIDGETNAMEPROP);
    pages[0].pfnDlgProc = WidgetNameDlgProc;
    pages[0].lParam = (LPARAM)&nameData;

    PROPSHEETHEADER psh = { sizeof(psh) };
    psh.dwFlags = PSH_WIZARD | PSH_PROPSHEETPAGE;
    psh.hInstance = g_hinstThisDll;
    psh.hwndParent = hwndOwner;
    psh.pszCaption = MAKEINTRESOURCE(IDS_WIDGETPROPTITLE);
    psh.nPages = ARRAYSIZE(pages);
    psh.ppsp = pages;
    PropertySheet(&psh);
}
```

For simplicity, this property sheet has only one page. This page needs a `WidgetData` worth of extra state, so we allocate that state (on the stack, in this case) and put a pointer to int in the `PROPSHEETPAGE`'s `lParam` for the dialog procedure to fish out:

```
INT_PTR CALLBACK WidgetNameDlgProc(
    HWND hdlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    auto pageData = (WidgetNameData*)GetWindowLongPtr(
        hdlg, DWLP_USER);
    switch (uMsg)
    {
    case WM_INITDIALOG:
        {
            auto page = (PROPSHEETPAGE*)lParam;
            pageData = (WidgetNameData*)page->lParam;
            SetWindowLongPtr(hdlg, DWLP_USER, (LONG_PTR)pageData);

            ... initialize the page ...
            return TRUE;
        }

    ... other message handlers ...
    }
    return FALSE;
}
```

For a property sheet dialog procedure, the `lParam` of the `WM_INITDIALOG` points to a `PROPSHEETPAGE` structure, and you can pull out the `lParam` to access your private data.

Now, this gets kind of complicated if the property sheet page was created via `CreatePropSheetPage`, say, because it is a plug-in that is added dynamically into an existing widget property sheet.

```
HPROPSHEETPAGE CreateWidgetNamePage(HWIDGET widget)
{
    PROPSHEETPAGE page = {};

    page.dwSize = sizeof(page);
    page.hInstance = g_hinstThisDll;
    page.pszTemplate = MAKEINTRESOURCE(IDD_WIDGETNAMEPROP);
    page.pfnDlgProc = WidgetNameDlgProc;
    page.lParam = (LPARAM)&(what goes here?);

    return CreatePropertySheetPage(&page);
}
```

You can't use a stack-allocated `WidgetNameData` because that will disappear once the `CreateWidgetNamePage` function returns. You probably have to create a separate heap allocation for it, and pass a pointer to the heap allocation as the `lParam`, but now you also need to add a property sheet callback so you can remember to free the data when you get a `PSPCB_RELEASE` callback.

Exercise: Why is `WM_DESTROY` the wrong place to free the data? (Answer below.)

To avoid this extra hassle, you can use this one weird trick: Append your private data to the `PROPSHEETPAGE`, and the system will carry it around for you.

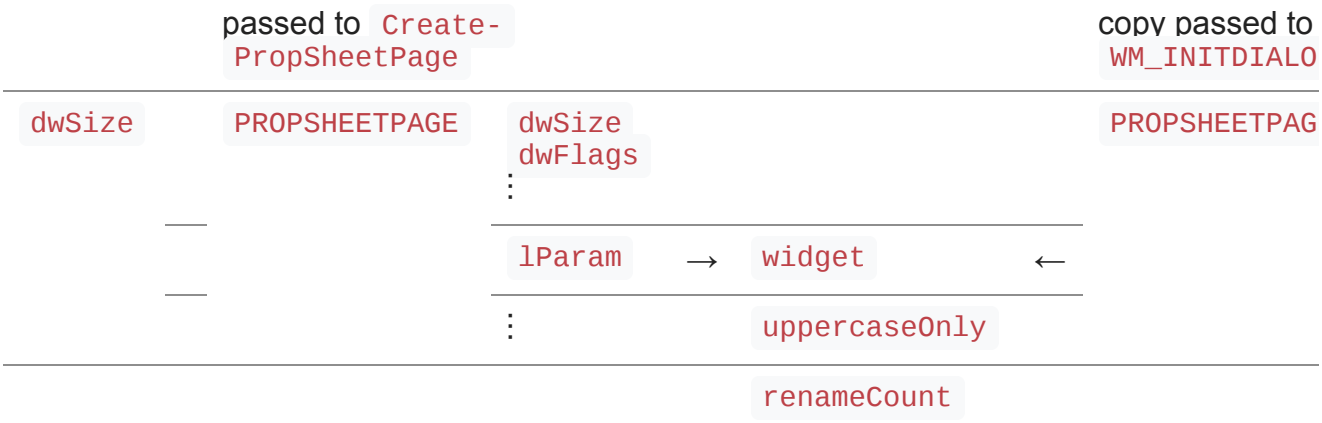
```
struct WidgetNameData : PROPSHEETPAGE
{
    HWIDGET widget;
    bool uppercaseOnly;
    int renameCount;
};

HPROPSHEETPAGE CreateWidgetNamePage(HWIDGET widget)
{
    WidgetNameData page = {};

    page.dwSize = sizeof(page);
    page.hInstance = g_hinstThisDll;
    page.pszTemplate = MAKEINTRESOURCE(IDD_WIDGETNAMEPROP);
    page.pfnDlgProc = WidgetNameDlgProc;

    // store the extra data in the extended page
    page.widget = widget;
    page.uppercaseOnly = IsWidgetPolicyEnabled(WidgetPolicy::UppercaseNames);
    page.renameCount = 0;

    return CreatePropertySheetPage(&page);
}
```

We append the data to the `PROPSHEETPAGE` by deriving from `PROPSHEETPAGE` and adding our extra data to it. The trick is that by setting the `page.dwSize` to the size of the entire larger structure, we tell the property sheet manager that our private data is part of the page. When the property sheet manager creates a page, it copies all of the bytes described by the `dwSize` member into its private storage (referenced by the returned `HPROPSHEETPAGE`), and by increasing the value of `page.dwSize`, we get our data copied there too.

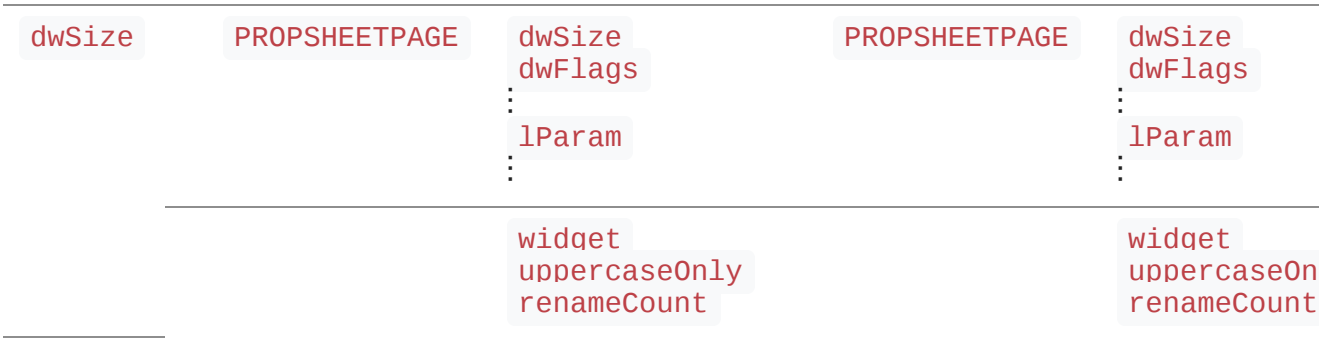
Recall that the `lParam` parameter to the `WM_INITDIALOG` message is a pointer to a `PROPSHEETPAGE`. In our case, it's a pointer to our custom `PROPSHEETPAGE` structure, and we can downcast to the specific type to access the extra data.

```
INT_PTR CALLBACK WidgetNameDlgProc(
    HWND hdlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    auto pageData = (WidgetNameData*)GetWindowLongPtr(
        hdlg, DWLP_USER);
    switch (uMsg)
    {
    case WM_INITDIALOG:
        {
            // the lParam points to our extended page
            pageData = (WidgetNameData*)lParam;
            SetWindowLongPtr(hdlg, DWLP_USER, (LONG_PTR)pageData);

            ... initialize the page ...
            return TRUE;
        }

    ... other message handlers ...
    }
    return FALSE;
}
```

In pictures: The traditional way creates a separate allocation that the `PROPSHEETPAGE`'s `lParam` points to. When the system copies the `PROPSHEETPAGE` into private storage, the `lParam` is copied with it.



The new way makes the whole thing a giant `WidgetNameData` with a `PROPSHEETPAGE` at the top and bonus data at the bottom.



The catch here is that the bonus data is copied to the internal storage via `memcpy`, so it must be something like a POD type which can be safely copied byte-by-byte.

If you didn't know about this trick, you would wonder why the `lParam` of the `WM_INITDIALOG` message points to the full `PROPSHEETPAGE`. After all, without this trick, the only thing you could do with the `PROPSHEETPAGE` pointer was access the `lParam`; all the other fields are explicitly documented as off-limits[1] during the handling of the `WM_INITDIALOG` message. Why bother giving you a pointer to a structure where you're allowed to access only one member? Why not just pass that one member?

And now you know why: Because you actually can access more than just the `lParam`. If you hung extra data off the end of the `PROPSHEETPAGE`, then that data is there for you too.

Extending the `PROPSHEETPAGE` structure means that each `PROPSHEETPAGE` can be a different size, which makes it tricky to pass an array of them, which is something you need to do if you're using the `PSH_PROPSHEETPAGE` flag. We'll look at that problem next time.

Bonus chatter: This is an expansion of a previous discussion of the same topic. In that earlier topic, I said that the array technique requires all of the elements to be the same size. But next time, I'll show how to create an array of heterogeneous types.

Bonus bonus chatter: This is similar to, but not the same as, the trick of adding extra information to the end of the `OVERLAPPED` structure. In the case of overlapped I/O, the same `OVERLAPPED` pointer is used; there is no copying. You can therefore put arbitrary complex data after the `OVERLAPPED`; they don't have to be POD types. Of course, you have to be careful to destruct them properly when the I/O is complete.

In the case of a `PROPSHEETPAGE`, the memory is copied, so the data needs to be `memcpy`-safe. You could still use it to hold non-POD types, though, by treating it as uninitialized memory that the system conveniently preallocates for you. You'll have to placement-construct the objects in their copied location, and manually destruct them when the property sheet page is destroyed.

Answer to exercise: If the property sheet page is never created (because the user never clicks on the tab for the page), then the dialog is never created, and therefore is never destroyed either. In that case, you don't get any `WM_DESTROY` message, and the memory ends up leaked.

[1] The documentation cheats a bit and says that you cannot modify anything except for the `lParam`, when what it's really saying is that you cannot modify any *system-defined* things except for the `lParam`. Your private things are yours, and you can modify them at will.

Raymond Chen

Follow