Eine sichere Methode Speicherlöcher zu vermeiden

Smart-Pointer

Smart-Pointer sind ein sehr gutes Hilfsmittel zur Verwaltung von Zeigern, die in der Regel bei der Nutzung eines C-Interfaces auftauchen. Ein Beispiel für ein C-Interface ist das Win32-API (C-Funktionen). Dieses wird unter der MFC-Bibliothek (C++-Klassen) gekapselt und kann natürlich auch direkt vom Programmierer genutzt werden.

Peter Thömmes, Oktober 2004

Um sich von der manuellen Freigabe von Resourcen zu entbinden, benutzt man in C++ sinnvollerweise Smart-Pointer. Smart-Pointer verhalten sich auf den ersten Blick wie Zeiger, sind aber in Wirklichkeit Objekte, die verschiedene Dinge automatisieren bzw. verwalten, wie z.B. die Freigabe von Resourcen. Hier ein paar Beispiele für Dinge, die Smart-Pointer tun können:

- **Heap-Speicher-Verwaltung**
  Automatisches `delete` bzw. `delete[]` nach vorangegangenem `new` bzw. `new[]`.
  Beispiel: `auto ptr` der STL.
- **GDI-Objekt-Verwaltung**
  Automatischer Aufruf von `DeleteObject()` nach dem Aufruf von Funktionen, wie `CreatePen()`, `CreateSolidBrush()`, `CreateFontIndirect()` und nicht zu vergessen `CreateRectRgn()`.
- **COM-Interface-Verwaltung**
  Automatische Freigabe des COM-Servers durch Aufruf von `Release()` nach vorhergehendem Aufruf von `CoCreateInstance()`
  Beispiel:`_com_ptr_t` bzw. das vordefinierte MACRO `_COM_SMARTPTR_TYPEDEF()`
- **BSTR-Verwaltung**
  Automatische Freigabe eines BSTR (OLE-String) durch `SysFreeString()`, nachdem eine Allokierung durch `SysAllocString()` erfolgte.
  Beispiel: `bstr_t`
- **DB-Verbindungs-Verwaltung**
  Bei **ADO** werden ODBC-Verbindungen über einen Smart-Pointer vom Typ `_ConnectionPtr` und SQL-Query-Results über einen Smart-Pointer vom Typ `_RecordsetPtr` zugänglich gemacht.

Grundlage der Programmierung eines Smart-Pointers bilden die sogenannten **Templates**, deren Konzept Bjarne Stroustrup mit der Einführung seiner Sprache C++ vorstellte (Templates gibt es nicht in C, Java oder C#). Im Gegensatz zum Weg, der von Java oder C# gegangen wird, nämlich Algorithmen generisch zu halten, indem man jedes Objekt von einer zentralen Basisklasse ableitet, werden Templates für jeden Typ gesondert instanziiert und somit auf jeden Typ zurechtgeschnitten. Ein Integer, wie `unsigned long` bleibt dabei ein schlanker System-Datentyp und wird nicht in ein Objekt gekapselt (Boxing).
Die wohl mächtigste Implementierung, die auf Templates beruht ist die **STL** (Standard-Template-Library), eine generische Algorithmen-Sammlung, deren Schnelligkeit und Effizienz bisher wohl einzigartig ist.
Anhand von `auto_ptr` der STL möchte ich die Arbeitsweise eines Smart-Pointer kurz skizzieren:

auto_ptr der STL

Die STL bietet neben ihren Algorithmen einen Smart-Pointer für die Heap-Speicher-Verwaltung. Am folgenden Beipiel wird der Unterschied zur manuellen Heap-Speicher-Verwaltung deutlich:

```
#include <stdio.h>
//wegen auto_ptr:
#include <memory>
using namespace std;


template<class T>
void Func(auto_ptr<T>& sp)
{
  printf("*sp = %d\r\n",*sp);
  *sp = 7;
  printf("*sp = %d\r\n",*sp);
}
```

```
int main()
{
  auto_ptr<unsigned long>
    spdwTest(new unsigned long);
  Func(spdwTest);
  return 0;
}
```

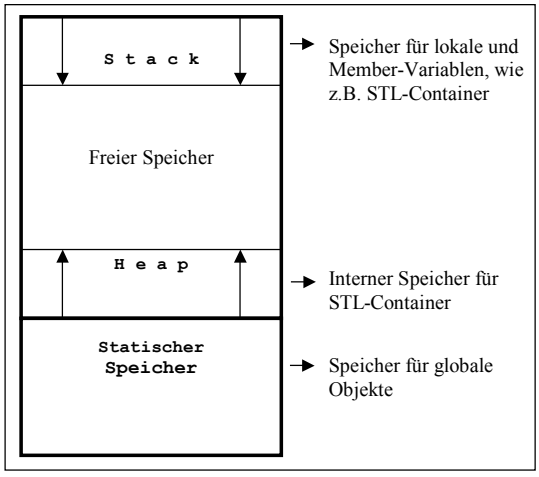
Zu Beginn von `main()` wird eine Speicherzelle vom Typ `unsigned long` auf dem Heap allokiert (`new`) und der Zeiger darauf sofort dem Smart-Pointer-Objekt `spdwTest` übergeben. Sobald der Gültigkeitsbereich verlassen wird, also sozusagen die Klammer `}` von `main()` überschritten wird, ruft `spdwTest` automatisch

```
delete p;
```

auf, wobei `p` der Zeiger auf die Speicherstelle ist.

Heap und Stack

Um zu verstehen, was da eigentlich passiert, muss man ein bisschen Speicherkunde betreiben. Ein C++-



Compiler generiert für seinen Programm-Code 3 Arten von Speicher:

Der Statische Speicher wird zu Beginn der Laufzeit eingeteilt und alle globalen und statischen Variablen verbleiben dort an der

gleichen Stelle, bis das Programm beendet ist. Lediglich hinsichtlich der Sichtbarkeit kann sich während des Programmlaufs dort was ändern, denn lokale statische Variablen sind immer nur dann sichtbar, wenn der lokale Code (Block zwischen { und }), in dem sie definert sind, durchlaufen wird. Genau dann, wenn dies zum ersten mal geschieht, werden sie auch initialisiert.
Stack und Heap teilen sich den restlichen freien Speicher (dynamischen Speicher) und wachsen aufeinander zu. Der Stack wird immer dann beansprucht, wenn in lokalem Code eine Variable definiert wird. Sie wird vom Compiler durch PUSH-Befehle auf den Stack gelegt und beim Austritt aus dem Block, also bei der nächsten schliessenden geschweiften Klammer }, wieder entfernt (POP). Der einzig kritische Speicher ist der **Heap**, denn seine Benutzung muss der Programmierer manuell steuern. Diese ist auch nicht automatisch steuerbar, da der Heap dazu da ist, Daten über die Laufzeit eines lokalen Blocks (Funktion) hinaus am Leben zu erhalten und dabei die Datengröße dynamisch zu ändern. Wenn einer Allokierung (`new`, `new[]` oder direkt durch `malloc()`) nicht irgendwann eine Freigabe (`delete`, `delete[]` oder direkt durch `free()`) folgt, solange man einen gültigen Zeiger auf den Speicher hat, dann entsteht ein **Speicherloch** (**Memory-Leak**). Hierbei ist zu beachten, dass man nie `new`/`delete`, `new[]`/`delete[]` und `malloc()`/`free()` miteinander vermischen darf. Ich habe den Heap-Speicher ein wenig provokativ "Speicher für STL-Container" genannt, da ich damit deutlich machen will, dass es immer besser ist, sich eines STL-Containers (`list`, `set`, `map`, `hash_map`, ...) zu bedienen, als selbst das Heap-Management zu übernehmen. Ist es dann doch mal nicht möglich, wie z.B. beim Aufruf einer C-Funktion, die einen Zeiger zurückliefert (C++ Funktionen sollten nur mit Referenzen und const-Referenzen arbeiten - siehe: www.notizen-zu-cpp.de), dann ist man gut beraten, sich des `auto_ptr` der STL zu bedienen oder einen eigenen Smart-Pointer zu programmieren.

Templates

Es gibt grundsätzlich 2 verschiedene Arten von Templates:

- **Class-Template**

Vorlage für eine Klasse. Beispiel:

```
template<class T>
class SmartPtr
{
public:
 MemSmartPtr(T* pT) : m_pT(pT)
 {}
 ...
}
```

Das Class-Template wird vom Programmierer **explizit** im Code für einen bestimmten Typ `class T` instanziiert.
Beispiel für `T = String`:

```
SmartPtr<String>
        spString(new String);
```

- **Function-Template**

Vorlage für eine Funktion, wobei hier nochmal zwischen globaler Funktion und Member-Funktion unterschieden wird.

```
template<class T>
void Func(SmartPtr<T>& sp)
{
 ...
}
```

Ein Function-Template wird **implizit** durch einen Funktionsaufruf instanziiert. Beispiel:

```
Func(spString);
```

Der Compiler erkennt, dass `spString` ein `SmartPtr`-Objekt vom Typ `T = String` ist und instanziiert die Funktion für diesen Typ.

Beispiel: MemSmartPtr

Ein plattformunabhängiges Beispiel soll das Programmieren eines Templates skizzieren. Grundsätzlich ist es so, dass ein Template nur aus einer Header-Datei besteht und der ganze Code inline dort definiert wird. An den Smart-Pointer, der

```
MemSmartPtr
```

heissen und zur Heap-Speicher-Verwaltung eingesetzt werden soll, gibt es folgende Anforderungen:

- **Konstruktion**

Ein Zeiger auf ein zuvor mit `new` oder `new[]` allokiertes Objekt/Array muss entgegen genommen und gespeichert werden. Ein Flag muss dem Template mitteilen, ob es sich um ein einzelnes Objekt (`new`) oder ein Array von Objekten (`new[]`) handelt, denn diese Angabe entscheidet darüber, ob `delete` oder `delete[]` zum löschen angewendet werden soll.

- **Destruktion**

Hier kommt die ureigenste Aufgabe des Smart-Pointers zum Vorschein, nämlich das Automatisieren einer Folgehandlung (hier: `delete` / `delete[]`) zu einer vorhergehenden Handlung (hier: `new` / `new[]`).

- **Zuweisung**

Im Gegensatz zur deep-copy wird hier lediglich der Zeiger kopiert und die Eigentümerschaft (Verantwortung für das `delete` / `delete[]`) übertragen.

- **Dereferenzierung**

Damit der Smart-Pointer sich auch wie ein Zeiger verhält, ist es erforderlich einen `operator*()` zu definieren, der das Objekt zurückliefert, auf den der interne Zeiger zeigt.

Hier nun das Code-Beispiel:

```
#ifndef MEMSMARTPTR_H_
#define MEMSMARTPTR_H_

#ifdef _MSC_VER //MSVC++
	#pragma warning(disable:4284) //no warning 'return...not a UDT or reference to a UDT'
#endif

template<class T>
class MemSmartPtr
{
	public:
		MemSmartPtr(bool bIsArray,T* pT = NULL) : m_bIsArray(bIsArray),m_pT(pT)
		{
		}
		MemSmartPtr(MemSmartPtr<T>& Obj) : m_pT(Obj.GetPtr()) //Become owner
		{
			Obj.Release(); //Release Obj from ownership
		}
		~MemSmartPtr()
		{
			if(m_pT)
			{
				if(m_bIsArray)
					delete[] m_pT;
				else
					delete m_pT;
			}
		}
		void Release()
		{
			m_pT = NULL; //give up ownership
		}
		T* GetPtr()
		{
			return m_pT;
		}
		bool IsArray()
		{
			return m_bIsArray;
		}
		bool IsValid()
		{
			if(m_pT != NULL)
				return true;
			return false;
		}
		operator bool()
		{
			return IsValid();
		}
		MemSmartPtr<T>& operator=(MemSmartPtr<T>& Obj)
		{
			if(this == &Obj)
				return *this;
			if(m_pT) //Release old memory
			{
				if(m_bIsArray)
					delete[] m_pT;
				else
					delete m_pT;
			}
			m_bIsArray = Obj.IsArray();
			m_pT = Obj.GetPtr(); //Become owner
			Obj.Release(); //Release Obj from ownership
			return *this;
		}
		T& operator[](unsigned long i)
		{
			return m_pT[i];
		}
		T* operator->()
		{
			return m_pT;
		}
		T& operator*() const
		{
			return reinterpret_cast<T&>(*m_pT);
		}
	private:
		bool m_bIsArray;
		T* m_pT;
};

#endif
```

Wie man sieht, ist natürlich nicht nur der Zuweisungs-Operator, sondern auch der Copy-Konstruktor mit einem `Release()` auf das Original zu versehen. Die Anwendung könnte dann wie folgt ausshen:

```
#include "MemSmartPtr.h"

template<class T>
void Func(MemSmartPtr<T>& sp)
{
	printf("*sp = %d\r\n",*sp);
	*sp = 9;
	printf("*sp = %d\r\n",*sp);
}

class String
{
	public:
		String()
		{
			strcpy(m_szStr,"Hello");
		}
		String(const char* const szStr)
		{
			strcpy(m_szStr,szStr);
		}
		virtual ~String() {}
		virtual void PrintIt();
		char& operator[](int pos)
		{
			return m_szStr[pos];
		}
	protected:
		char m_szStr[256];
};

void String::PrintIt()
{
	printf("%s\r\n",m_szStr);
}

int main()
{
	MemSmartPtr<unsigned long> spdw2(true,new unsigned long[15]);
	Func(spdw2);
	spdw2[2] = 7;

	MemSmartPtr<String> spString(false,new String);
	spString->PrintIt();
	(*spString)[0] = 'h';
	spString->PrintIt();
	printf("spString.GetPtr() = %08X\r\n",spString.GetPtr());

	printf("\r\nPlease press <ENTER> to quit...");
	getchar();

	return 0;
}
```

Beispiel: GDISmartPtr

Ein weiteres Beispiel zeigt die automatische Freigabe von GDI-Resourcen des Betriebssystems Windows. Unter GDI (Graphics Device Interface) versteht man diejenigen Windows-SDK-Funktionen, die zur graphischen Ausgabe dienen. Bei der GDI-Programmierung werden systeminterne Resourcen allokiert (Linienstil, Füllstil, Schriftart, etc.), die hinter dem Code für die graphische Ausgabe wieder freigegeben werden müssen. Wird diese Freigabe vergessen, dann entstehen Resource-Leaks, die füher oder später einen Crash verursachen.

An den GDI-Smart-Pointer, der

```
MemSmartPtr
```

heissen soll, stellen sich folgende Anforderungen:

- **Konstruktion**

Erwartet wird ein sogenanntes Handle, was in diesem Fall ein Zeiger auf ein GDI-Objekt ist, welches durch einen Betriebssystemaufruf allokiert wurde. Ein solcher Aufruf kann unter anderem sein:

```
CreatePen()
CreateSolidBrush()
CreateFontIndirect()
CreateRectRgn()
```

- **Destruktion**

Der wichtigste Punkt ist natürlich die automatische Resource-Freigabe, die hier durch den Aufruf von `DeleteObject()` erledigt wird.

- **Zuweisung**

Wieder mal wird der Zeiger kopiert und die Eigentümerschaft, also die Verantwortung für den Aufruf von `DeleteObject()` übertragen. Zuvor müssen alte Resourcen ggf. freigegeben werden.

- **Dereferenzierung**

Es wäre kein Smart-Pointer, wenn nicht das Verhalten eines Zeigers da wäre, was natürlich über den `operator*()` zu definieren ist, der das Objekt zurückliefert, auf den der interne Zeiger (das Handle) zeigt.

Das Code-Beispiel:

```
#ifndef GDISMARTPTR_H_
#define GDISMARTPTR_H_

template<class PT>
class GDISmartPtr
{
    public:
        GDISmartPtr(PT pT = NULL) : m_pT(pT) //default c'tor + c'tor with implicit cast
        {
        }
        GDISmartPtr(GDISmartPtr<PT>& Obj) : m_pT(Obj.GetPtr()) //Become owner
        {
            Obj.Release(); //Obj is no longer owner
        }
        ~GDISmartPtr()
        {
            if(m_pT)
                ::DeleteObject(m_pT);
        }
        void Release() //be no longer owner
        {
            m_pT = NULL;
        }
        PT GetPtr() //return the real pointer
        {
            return m_pT;
        }
        bool IsValid()
        {
            if(m_pT != NULL)
                return true;
            return false;
        }
        operator bool()
        {
            return IsValid();
        }
        GDISmartPtr<PT>& operator=(GDISmartPtr<PT>& Obj)
        {
            if(this == &Obj)
                return *this;
            if(m_pT)
                ::DeleteObject(m_pT); //Free old resources
            m_pT = Obj.GetPtr();      //Become owner
            Obj.Release();            //Obj is no longer owner
            return *this;
        }
        PT operator->() { return m_pT; }
    private:
        PT m_pT;
};

#endif
```

Und auch hierzu ein kleines Anwendungsbeispiel. Es handlet sich um die überschriebene Methode `OnPaint()` einer von `CDialog` abgeleiteten Klasse (`CDialog` ist eine Klasse der MFC-Bibliothek):

```
void MyDialog::OnPaint()
{
	//Get the client DC:
	CPaintDC dc(this);
	HDC hDC = dc.m_hDC;

	//Get the client rect:
	CRect rect;
	GetClientRect(&rect);

	//Define the PEN (linestyle):
	GDISmartPtr<HPEN> spLineStyle =  ::CreatePen(PS_SOLID,1,RGB(0,0,0)); //black
	HPEN hpenOldLineStyle = (HPEN) ::SelectObject(hDC,spLineStyle.GetPtr());

	//Define the BRUSH (fillstyle):
	GDISmartPtr<HBRUSH> spFillStyle = ::CreateSolidBrush(RGB(0,0,255)); //blue
	HBRUSH hbrOldFillStyle = (HBRUSH) ::SelectObject(hDC,spFillStyle.GetPtr());

	//Draw:
	::Ellipse(hDC,rect.left,rect.top,rect.right,rect.bottom);

	//Restore the DC:
	::SelectObject(hDC,hbrOldFillStyle);
	::SelectObject(hDC,hpenOldLineStyle);

	CDialog::OnPaint();
}
```

Fazit

Immer dann, wenn man C++ programmiert, sollte man von vorn herein vermeiden mit Zeigern zu arbeiten und statt dessen auf Referenzen zurückzugreifen. Stößt man dennoch auf ein C-Interface mit Zeigern (wie z.B. ein Betriebssystem-SDK), dann ist es sehr ratsam die Zeiger in Smart-Pointer einzukapseln um Speicher- oder Resource-Leaks zu vermeiden. Anhand der gezeigten Beispiele kann man sich abschauen, wie Smart-Pointer-Templates für andere Resourcen programmiert werden können.