Method Call Interception

Method call interception means executing some code before and after the body of a function or a method is executed.

April 01, 2005
URL:http://www.drdobbs.com/cpp/method-call-interception/184401947

Method call interception means executing some code before and after the body of a function or a method is executed. During development, there are many situations where method call interception can come in handy. For instance, you may want to:

- Trace every method call to follow the flow of your program in a certain scenario in real time.
- Log every method invocation for later study or audit call patterns in an application.
- Profile how much time is spent in which method.
- Ensure that there are no resource leaks on method exit.

Method call interception is in the realm of Aspect-Oriented Programming (AOP), which deals with issues that can't be encapsulated in an object-oriented class or set of classes because these issues cut across the entire software (cross-cutting concerns). Programming these aspects of the program is difficult because they are scattered all over the code. AOP deals with this problem by identifying that you can't properly manage aspects using conventional object-oriented techniques, and then goes on to define language extensions that help in programming aspects (see http://www.aosd.org/ for more information).

In this article, I discuss several ways to perform method call interception, introduce the Method Call Interceptor (MCI) mechanism that enables source-level interception, discuss MCI automation and the overhead incurred by MCI, and finally present the Poor Man's Profiler (PMP), a simple profiler that demonstrates the use of MCI in a real-world scenario.

Intercept Method Calls

There are various ways to intercept method calls, depending on your execution environment, source code access, and expertise. COM+, for example, provides a hooking mechanism that can be used for this purpose. DLL calls can be easily intercepted because the address of all functions is kept in an Import Address Table (IAT), which can easily be modified to call your interception code instead. Another technique is to inject interception code into object files before linking. However, in this article I concentrate on source-level interception. This means modifying the source of a program you typically develop to do the interception.

The naive approach (which works fine in many cases) is just to write some code at the beginning and end of each method body; see Example 1. The problem with this approach is that it is tedious, labor intensive, nothing can enforce it, and across a multideveloper project, inconsistent formats are likely to appear. You are also likely to forget the `finish` statement, which leads to functions that appear to never exit (at least in the trace output). Yet another problem with this approach is that you often need different types of interception at different times. For example, when trying to locate a difficult resource leak, you may want to track resources on entering/leaving a method; while hunting a stubborn logical bug, you may want a call tree tracing; and while tuning performance, you may want to know how long each method takes and how often it is called. Putting all this code in every method or changing the interception code every now and then is inhuman, if not inhumane.

Example 1: Naive approach.

```
void Foo(int x, int y)
{
    cout << endl << "Foo(int x, int y) - start";
    ...
    cout << endl << "Foo(int x, int y) - finish";
}
```

A somewhat better solution is defining an automatic object that does the "start" thing in its constructor and the "finish" thing in its destructor; see Example 2. The automatic object makes sure you won't forget that closing `finish` line and keeps all the code in one place so you don't have to edit the code in every method separately. Still, you must pass the name of each method specifically, and if you want to add logging code or profiling, you must put AutoLog and AutoProfiler automatic objects in every method. Again, you have different interception needs in different methods and in different phases of the development process. Enter the Method Call Interceptor mechanism.

Example 2: Class AutoTrace.

```
class AutoTrace
{
public:
   AutoTrace(const string & s) : m_line(s)
    {
        cout << endl << m_line << " - start";
    }
    ~AutoTrace()
    {
        cout << endl << m_line << " - finish";
    }
private:
    string m_line;
};

void Foo(int x, int y)
{
    AutoTrace("Foo(int x, int y)");
    ...
}
```

Method Call Interceptor: The Mechanism

Method Call Interceptor (MCI) is a mechanism that addresses the aforementioned problems. It consists of class `Mci`, an abstract class (`IMciEvents`), and a utility class (`MethodAnalyzer`). The basic idea is this: An instance of the `Mci` class is placed automatically at the beginning of each method. The constructor of `Mci`, which is called upon entering each method, collects some information regarding the current method using the `MethodAnalyzer` and notifies a preregistered events sink (an object that implements the `IMciEvents` interface).

The motivation behind this observer-style design is separation of concerns. The code that performs the interception (`Mci`) is totally independent and is actually unaware of the code that performs the actual tracing, logging, profiling, or what have you. This partitioning allows sophisticated implementations such as filtering and performing different actions for different methods without changing the generic method interception code or the method's code.

The `Mci` class is similar to the `AutoTrace` class previously mentioned. Listing One contains the `Mci` class. `Mci` just notifies its sink that some method has entered or left (by the program counter). It also provides a lot of information about this method to the sink. This information consists of the filename and line where this specific instance of `Mci` is located and also a method info struct, which contains the class name, method name, and type of each argument. The sink is registered by calling the static `Register` method. The `GetSink()` method is an interesting hack; it allows using a static variable without declaring it in a .cpp file. Although the C++ Standard allows declaring and initializing a static variable in the class definition, VC++ 6.0 (which I use) doesn't. The `GetSink()` method returns a reference to an internal static object and thus circumvents the problem. This means that all the instances of `Mci` in each and every method in your code notifies this single sink. On the surface, it looks like the same code is executed for every method, but in practice, the registered sink may employ a filtering and classification system based on the method information passed to it and dispatch the events accordingly to different handlers. For example, the initial sink may dispatch events according to groups of filenames (dispatch all events from files a.cpp, b.cpp, and c.cpp to `Handler_1` and all other events to `Handler_2`). The important point here is that `Mci`—the event's source—is completely unaware of the entire procedure. It doesn't even know the true type of the original sink. All `Mci` knows is that someone registered an `IMciEvents` pointer to which it sends all the events.

Listing One

(a)

```
#ifndef MCI_H
#define MCI_H

#include <string>
#include "MethodAnalyzer.h"

struct IMciEvents;

class Mci
{
public:
    Mci(const std::string & filename, int lineNumber, std::string line);
    ~Mci();
    static IMciEvents * & GetSink();
    static void Register(IMciEvents * pSink);
private:
    std::string GetLine(const std::string & filename, int lineNumber);
private:
    std::string     m_filename;
    int             m_lineNumber;
    MethodInfo      m_methodInfo;
};
#endif // !defined(__MCI_H__)
```

(b)

```
#include "Mci.h"
#include "IMciEvents.h"
#include "MethodAnalyzer.h"
#include <fstream>

using std::string;
using std::ifstream;

Mci::Mci(const string & filename, int lineNumber, string line) :
        m_filename(filename),
        m_lineNumber(lineNumber)
{
    if (!GetSink())
        return;
    if (line.empty())
        line = GetLine(filename, lineNumber-2);
    m_methodInfo = MethodAnalyzer::Analyze(line);
    // verify corectness of class name using typeinfo
    GetSink()->OnEnter(m_filename, m_lineNumber, m_methodInfo);
}
Mci::~Mci()
{
    if (!GetSink())
        return;

    GetSink()->OnLeave(m_filename, m_lineNumber, m_methodInfo);
}
IMciEvents * & Mci::GetSink()
{
    static IMciEvents * pSink = 0;
    return pSink;
}
void Mci::Register(IMciEvents * pSink)
{
    GetSink() = pSink;
}
string Mci::GetLine(const string & filename, int lineNumber)
{
    ifstream f;
    f.open(filename.c_str());
    const int BUFF_SIZE = 1024;
    char buff[BUFF_SIZE];
    for (int i = 0; i < lineNumber; i++)
        f.getline(buff, BUFF_SIZE);
    return string(buff);
}
```

`IMciEvents`. This interface (abstract class) should be implemented by some object and registered with the `Mci` class by calling the static `Mci::Register()` method; see Listing Two. There is nothing much to say about this interface, except that it provides an empty implementation for the events in case some sink doesn't care about one of the events. If the events were declared pure virtual, the implementing sink is compelled to implement all the events, even if it is only interested in the `OnLeave` event. This is not a big deal for an interface with two methods, but I call it consideration and putting the client first. You may also notice that the return type is `void` since `Mci` doesn't care what the sink does with the information it sends. The `<string>` header is included, although a forward declaration would have been good enough. Unfortunately, it is forbidden by the Standard to add declarations or definitions to namespace `std` (to let vendors add their own extensions without collisions with user's code).

Listing Two

```
#ifndef MCI_EVENTS_H
#define MCI_EVENTS_H

#include <string>

struct MethodInfo;
struct IMciEvents
{
    virtual void OnEnter(const std::string & filename,
                                    int line, const MethodInfo & mi) {}
    virtual void OnLeave(const std::string & filename,
                                    int line, const MethodInfo & mi) {}
};
#endif
```

`MethodAnalyzer`. This class is responsible for analyzing the current method and populating a `MethodInfo` struct. Example 3 contains a censored definition of the class. `MethodAnalyzer` exposes a single static method `Analyze()`. This method accepts as input a string that contains the text line from the source where the method was declared. The important thing about the `Analyze` method is that it is called dynamically every time a method is entered by the code, even if the same method is called lots of times. It could be wasteful if the analysis results were always the same for each method. In this case, some sort of caching per method would be helpful. However, it is likely that the analysis may also include the values of input/output arguments, and the return value of the method in the future. Clearly, this is a classic time/space trade-off.

Example 3: Class MethodAnalyzer.

```
class MethodAnalyzer
{
public:
    static MethodInfo Analyze(std::string line);
private:
    ...
};
```

Automating MCI

Putting an `Mci` object at the beginning of each method in your code is a tedious task, and if you want to use it on a large existing project, it becomes daunting. To remedy this and cater to the natural programmer's laziness, I present some automation options. The objective is to have a project where all source files `#include <Mci.h>` and all methods contain as their first statement the line:

```
Mci m(__FILE__, __LINE__, "Method(ArgType1 arg1, ArgType2 arg2...");
```

To achieve this objective automatically, I came up with the following algorithm:

1. Identify all the project source files.
2. `#include <Mci.h>` in every source file.
3. Scan each source file.
4. Identify every method (or function).
5. Extract the string that the `Mci` constructor requires as a third parameter.
6. Place a proper `Mci` line at the beginning of the method.

This automation procedure can be done offline in any language. I use Python, which is great in general and superb for such text-processing tasks. The script I wrote is naive and you are encouraged to modify it, or write a completely new automation script. Listing Three contains the InjectMci.py script. I will not delve into all the gory details. The basic idea is to detect lines that contain a method definition (using regular expressions), generate an `Mci` line, and inject it in the proper place. I put in a moderate amount of flexibility, such as working with several brace styles and whitespace filtering.

Listing Three

```
#!/usr/local/bin/python
import os, sys, glob, re

index = 0
text  = ''

def InjectMci(text, selective):
   if selective and text.find('INJECT_MCI') == -1:
      return text
   index = 0
   text = InjectMciHeader(text)
   text = InjectMciObjects(text, selective)
   return text

def InjectMciHeader(text):
   if text.find('#include "Mci.h"') != -1:
      return text

   lines = text.split('\n')

   index = 0
   # find index of last line that contains #include (0 if no #include
   is found)
   for i in range(len(lines)-1, 0, -1):
      if lines[i].find('#include') != -1:
         index = i+1
         break

   text = '\n'.join(lines[:index])
   text += '\n#include "Mci.h"\n'
   text += '\n'.join(lines[index:])

   return text

def GetMciLine(line):
   mci_mask = '\tMci m(__FILE__, __LINE__, "%s");'
   line = line.replace('{', ' ').strip()
   return mci_mask % line

def InjectMciObject(base, index, lines, new_lines, selective):
   line = lines[index]
   if selective:
      if lines[index+base+1].find('INJECT_MCI') != -1:
         if base == 1:
            new_lines.append(lines[index+1])
         new_lines.append(GetMciLine(line))
         index += base+2; # skip the INJECT_MCI line
      else:
         index += 1
   else:
      if base == 1:
         new_lines.append(lines[index+1])
      new_lines.append(GetMciLine(line))

      index += base+1;
   return index

def InjectMciObjects(text, selective):
   method_re = r'[ \t]*.+[ \t]+.+::.+\(.*\)[ \t]*'
   open_par_re = r'[ \t]*{[ \t]*'
   p1 = re.compile('%s$' % method_re)
   p2 = re.compile('%s$' % open_par_re)
   p3 = re.compile('%s%s$' % (method_re, open_par_re))
   lines = text.split('\n')

   new_lines = []

   index = 0
   while index < len(lines)-2:
      line = lines[index]
      new_lines.append(line)
      if p1.match(line) and p2.match(lines[index+1]):
         index = InjectMciObject(1, index, lines, new_lines, selective)
      elif p3.match(line):
            index = InjectMciObject(0, index, lines, new_lines, selective)
        else:
            index += 1

    return '\n'.join(new_lines)

if __name__ == "__main__":
    selective = len(sys.argv) > 1 and sys.argv[1] == 'selective'
    cpp_fi les = glob.glob('*.cpp')
    for f in cpp_files:
        print '-'*20
        text = open(f).read()
        text = InjectMci(text, selective)
        print text
        open(f, 'w').write(text)
```

Ad Hoc MCI

The InjectMci.py script is a lifesaver when you need it, but requires integration with the build process or manually running it every time you add a new method. Also, logging each and every method in your code base might be a little excessive. Sometimes, all you want is a quick benchmark or to sift through the call graph of a few selected functions. In this case, adding a couple of `Mci` objects manually is completely reasonable. However, doing it for more than three or four methods gets old really fast. You have to paste the `Mci` line, then type the method definition as the third string argument (or copy-and-paste it and lose the `Mci` line in your clipboard). The solution is running InjectMci.py in selective mode. Instead of putting an `Mci` object in every method, it will put it only in methods that contain `INJECT_MCI`. This way, you can quickly annotate a few methods with `INJECT_MCI` and be done with it. To use InjectMci.py in selective mode, just pass `selective` as a command-line argument.

MCI Overhead

MCI puts a heavy load on your program if used precariously. The work of parsing the method definition line might slow your program to a crawl in tight loops, which call noninline methods (bad idea in general). Remember that MCI is primarily a development aid and should not be active even at development time while stress testing your code. There are several things you can do to reduce the performance hit incurred by MCI:

- Do not put MCI in inline functions. Inline functions are often called in tight loops, and generally should execute as fast as possible. The easiest thing to do is to put MCI objects only in functions defined in .cpp files, which may or may not work in all cases. For example, if you always define inline functions in .H files, it works perfectly, but if you define inline functions in .cpp files and then `#include` the .cpp file, it won't work; see Example 4.
- Employ smart filtering during the MCI automation, so that infrastructure classes such as custom containers or communication code are not burdened by MCI.
- If you don't care about dynamic method interception, cache the resulting `MethodInfo` struct. The question is where to keep the `MethodInfo`. You can't keep `Mci` a static object in the method because then the constructor is called only once. You can't make the `m_methodInfo` data member of `Mci` a static object in the constructor because it is shared by all `Mci` instances. The only solution I could think of is to keep a static map indexed by a pairing of a filename and line (which is guaranteed to be unique). Whenever the `Mci` constructor is called, it first queries the cache using the filename and line parameter. If this method's info is already cached, then it just uses it to notify the sink; otherwise, it calls `MethodAnalyzer::Analyze()` and stores the resulting `MethodInfo` in the cache. However, searching a huge cache can hurt even more (think about CPU cache misses). I didn't implement such a system.
- Use preprocessor `#define` to enable/disable MCI. This method puts the responsibility to enable/disable MCI in each method or file separately. It is simpler than a full-scale filtering system, but doesn't scale well and requires high maintenance. In addition, it smudges ugly preprocessor `#ifdef`s all over the code; see Example 5.

Example 4: (a) GoodInline.h; (b) BadInline.h; (c) BadInline.cpp.

(a)

```
// Good : no MCI for .h file
void inline Foo() { ... }
```

(b)

```
void inline Foo();

#include Inline.cpp
```

(c)

```
void inline Foo()
{
    // Bad : Mci injected for .cpp
    Mci mci(...);
    ...
}
```

Example 5: Overhead.

```
void inline Foo()
{
#ifdef ENABLE_MCI
    Mci mci(...);
#endif
    ...
}
```

Packaging

MCI is packaged as a static library that you link with your target project. I used pure Standard C++. I tested it using Visual C++ 6 and Visual C++.NET 2003. It should compile under any C++ compiler (no templates or other tricks—this time).

Poor Man's Profiler

PMP is a Windows console application that uses MCI for simple method-profiling tasks. The profiler (available at http://www.cuj .com/code/) implements the `IMciEvents` interface, measures how long each method of the `A` and `B` classes takes, and writes the results to standard output. Beware that this is a simple implementation that breaks under nested calls (no stack for start times), not to mention multithreaded scenarios. The motivation is to show what an `Mci` sink looks like and how it interacts with MCI. You should not try to sell it to your boss as the next generation of profiling technology. Listing Four contains the `Profiler` class that derives from `IMciEvents` and `implements` its two methods: `OnEnter()` and `OnLeave()`. The implementation is just as simple—the start time of the method is stored in the `OnEnter` event and the duration is calculated by subtracting the current tick count from the stored start time.

Listing Four

(a)

```
#ifndef __PROFILER_H__
#define __PROFILER_H__

#include <windows.h>
#include "IMciEvents.h"

class Profiler : public IMciEvents
{
public:
    // IMciEvents methods
    virtual void OnEnter(const std::string & filename, int lineNumber,
                                               const MethodInfo & mi);
    virtual void OnLeave(const std::string & filename, int lineNumber,
                                               const MethodInfo & mi);
private:
    DWORD   m_start;
};
#endif
```

(b)

```
#include "Profiler.h"
#include "MethodInfo.h"
#include <iostream>

using std::cout;
using std::endl;
using std::string;

void Profiler::OnEnter(const string & filename, int lineNumber,
                                          const MethodInfo & mi)
{
    m_start = ::GetTickCount();
}

void Profiler::OnLeave(const string & filename, int lineNumber,
                                          const MethodInfo & mi)
{
    DWORD duration = ::GetTickCount() - m_start;
    cout << endl << "Method '" << mi.className << "::" << mi.name <<
        "' took " << duration << " milli-seconds";
}
```

This is the right point to yell: "Hey, the profiler didn't register itself as the `Mci` sink." This is true and intentional. Even in such a simple setup, the event source (`Mci`) and the event handler (`Profiler`) don't know each other (don't `#include` their respective .h files). The registration is done by the `main` function in this case (see Example 6). This is an idiom of component-based development called "Third Party Binding." `Mci` gets a pointer to the `Profiler`, but actually it sees an `IMciEvents` pointer thanks to the automatic upcasting. The `Profiler` class is totally oblivious to the existence of `Mci`. All it knows is that someone is supposed to call its `OnEnter()` and `OnLeave()` method. Period. The same goes for `Mci`. It knows nothing about the `Profiler`. All it knows is that in its constructor, it should call the registered sink's `OnEnter()` method and in its destructor it should call the sink's `OnLeave()` method. Period. This is another idiom called "Abstract Interactions."

Example 6: Registration.

```
int main(int argc, char* argv[])
{
    Profiler p;
    Mci::Register(&p);
    ...
}
```

Note on #defining {

When I first conjured the idea of MCI, I thought about making it look completely transparent. I wanted to `#define` the opening curly brace like this:

```
#define {  { Mci m(...);
```

The code would have looked totally normal (sans `Mci`), but under the covers, MCI would have done its stuff. There are many problems with this approach, such as the fact that *every* opening curly brace gets an MCI object including regular scope braces (`if`, `while`, `for`, and so on), and braces in namespace, enum, struct, and class definitions:

```
namespace { Mci m(...);
enum { Mci m(...); SUNDAY, MONDAY... }
```

Yet, what really threw me off this track is the simple fact that you can't `#define` curly braces. If I think objectively about it, it looks like the only advantage for this approach is the "Wows" I would have gotten from my colleagues (which clearly makes it a worthwhile endeavor).

Conclusion

MCI combines several C++ features, techniques, and idioms creatively to provide an easy-to-use solution for a limited domain of cross-cutting aspects—entering and leaving method calls during development time. However, using MCI and MCI automation (through Python) is not a no-brainer (in other words, it's a brainer) and you must consider the significant overhead MCI puts on your code and adapt it to your needs.

Gigi Sayfan is a software developer specializing in object-oriented and component-oriented programming using C++.