Debug those mysterious problems with your application's memory

Debugging aids for IBM i heap memory

Scott Hanson December 14, 2012

Debugging heap memory problems within an application can be difficult on any platform. Fortunately, IBM i 6.1 and later releases provide support to help debug heap memory problems within the ILE environment. This article explains what heap memory is and illustrates the correct usage of heap memory from several ILE languages, including C, C++, RPG, COBOL, and CL. It also describes some of the common problems that occur when using the heap memory and how to use the support provided on IBM i to debug those problems.

What is heap memory?

Heap memory is a common pool of free memory used for dynamic memory allocations within an application. The term heap memory is used because most of the implementations for dynamic memory allocation make use of a binary tree data structure called a *heap*. Dynamic memory allocation is the explicit allocation and deallocation of memory (or storage) during the run time of an application. In this article, the term *heap manager* is used to describe the software that handles the dynamic memory allocations for an application.

There are two primary operations performed by the heap manager. The first operation is *allocation*. This operation reserves a block of storage of a given size and returns a pointer to the reserved block. Exclusive ownership of the block of storage is given to the application; the application can use the block of storage for whatever purpose it requires. The second operation is *deallocation*. This operation returns a (previously allocated) block of storage to the heap manager. When a block of storage is deallocated, the ownership of the block is returned to the heap manager. This block of storage can subsequently be reused by the heap manager for future allocations.

A third operation that is commonly provided by heap managers is *reallocation*. This operation resizes a given block of storage to a new size and returns a (possibly updated) pointer to the block of storage. Although not strictly required (as it can be implemented with the basic allocation and deallocation operations), the reallocation operation is frequently provided by heap managers.

On IBM i® , there are two different types of heap storage: single-level storage and teraspace storage. The type of storage used by an application is determined by the program attributes

specified at the program creation time. By default, single-level storage is used. In order to use teraspace storage, special compile options and program creation options must be used. In addition, application programming interfaces (APIs) can also be used to allocate and deallocate teraspace storage within a single-level storage application. Additional information about these two types of storages can be found within the "Teraspace and Single-Level Storage" section of the *ILE Concepts* manual.

There are some important differences between the two types of heap storage. A single-level storage allocation from the heap can be a maximum of 16 MB in size. A teraspace storage allocation from the heap can be many terabytes in size. Sixteen-byte pointers must be used to address the single-level storage. Eight-byte pointers can be used to address the teraspace storage for languages (such as C and C++) that support eight-byte pointers. The total size of a single-level storage heap is limited to 4 GB per job. There is no such limit for a teraspace heap; the total size of a teraspace storage heap is limited only to the amount of the available system storage.

How to allocate or deallocate heap memory

Although each language has different methods for allocation and deallocation, heap memory can be used in all the Integrate Language Environment (ILE) languages. The `malloc()` and `free()` functions are used in C, the `new` and `delete` operators are used in C++, and the `%ALLOC` built-in function and the `DEALLOC` operation are used in RPG. Although COBOL and CL do not have built-in functions to manage heap memory, the ILE model allows functions to be called from any ILE language, and therefore, those languages can call the `malloc()` and `free()` functions to manage heap memory. In fact, all the ILE languages can call the `malloc()` and `free()` functions to manage heap memory. Several examples follow that illustrate the allocation, usage, and deallocation of heap memory in all of these languages.

Example 1 is written in C.

Example 1 – Heap memory in C

/* To compile: CRTBNDC PGM(EXAMPLE1) */

```
#include <stdlib.h>
#include <string.h>

int main (int argc, char *argv[])
{
  /* allocate 16 bytes of storage from the heap */
  char *ptr = (char*)malloc(16);

  /* set the allocated storage */
  memcpy(ptr, "abcdefghijklmnop", 16);

  /* deallocate storage */
  free(ptr);

  return 0;
}
```

Example 2 is written in C++.

Example 2 – Heap memory in C++

```
/* To compile: CRTBNDCPP PGM(EXAMPLE2) */
#include <string.h>

int main (int argc, char *argv[])
{
  /* allocate 16 bytes of storage from the heap */
  char *ptr = new char[16];

  /* set the allocated storage */
  memcpy(ptr, "abcdefghijklmnop", 16);

  /* deallocate storage */
  delete [] ptr;

  return 0;
}
```

Example 3 is written in RPG.

Example 3 – Heap memory in RPG

```
     * To compile: CRTBNDRPG PGM(EXAMPLE3)
    H dftactgrp(*no)
    D ptr@            S               *
    D based_data      S             16A   BASED(ptr@)
    D sz              S              5  0 INZ(16)
     /free
       // allocate 16 bytes of storage from the heap
       ptr@ = %ALLOC(sz);

       // set the allocated storage
       based_data = 'abcdefghijklmnop';

       // deallocate storage
       DEALLOC ptr@;

       *inlr = '1';
```

Example 4 is written in COBOL.

Example 4 – Heap memory in COBOL

```
       PROCESS NOMONOPRC.
       IDENTIFICATION DIVISION.
      * To compile: CRTBNDCBL PGM(EXAMPLE4) BNDDIR(QC2LE)
       PROGRAM-ID. EXAMPLE4.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  ptr                POINTER.
       01  sz                 PIC S9(9) BINARY.
       LINKAGE SECTION.
       77  based-data PIC X(16).
       PROCEDURE DIVISION.
       MAIN-LINE SECTION.
       001-MAIN-FLOW.
           MOVE 16 TO sz.
      *    allocate 16 bytes of storage from the heap
           CALL LINKAGE PRC "malloc" USING BY VALUE sz
                                     RETURNING ptr.
```

```
      *    set the allocated storage
           SET ADDRESS OF based-data TO ptr.
           MOVE "abcdefghijklmnop" TO based-data.
      *    deallocate storage
           CALL LINKAGE PRC "free" USING BY VALUE ptr.
           STOP RUN.
```

Example 5 is written in CL.

Example 5 – Heap memory in CL

```
/* To compile: CRTBNDCL PGM(EXAMPLE5) */
PGM
  DCL VAR(&PTR) TYPE(*PTR)
  DCL VAR(&SZ) TYPE(*INT) VALUE(16)
DCL VAR(&BASED_DATA) TYPE(*CHAR) LEN(16) STG(*BASED) +
    BASPTR(&PTR)

  /* allocate 16 bytes of storage from the heap */
  CALLPRC PRC('malloc') PARM( (&SZ *BYVAL) ) RTNVAL(&PTR)

  /* set the allocated storage */
  CHGVAR VAR(&BASED_DATA) VALUE('abcdefghijklmnop')

  /* deallocate storage */
  CALLPRC PRC('free') PARM( (&PTR *BYVAL) )
ENDPGM
```

These examples allocate a very small amount (16 bytes) of heap storage. Typically, the blocks of storage allocated would be much larger. As an example, consider the fact that the maximum size of a CL variable is 32767 bytes. Much larger blocks of storage can be managed within CL by using heap storage and the *PTR variables.

Common problems with heap memory

As heap allocation and deallocation must be explicitly performed by an application, the potential exists for incorrect usage of these operations. Common scenarios for the incorrect usage of heap memory include: Writing an amount of data larger than the size of the allocated storage (memory overwrite), reading an amount of data larger than the size of the allocated storage (memory over read), writing to storage or reading from storage which was previously deallocated (reuse of deallocated memory), deallocating storage more than one time (duplicate deallocation), and failing to deallocate memory when it is no longer used (memory leak).

The following example programs illustrate these heap memory problems.

The first two scenarios involve the size of the heap allocations. At allocation time, the size of the desired storage is given to the heap manager and it returns a block of storage which is large enough to satisfy the size requested. If the application incorrectly calculates the heap size, it can sometimes unintentionally read or write a portion of the heap storage that does not belong to the current allocation. This can and often does cause problems for the application.

Example 6 illustrates a memory overwrite in C++. A total of 17 bytes are written to a heap allocation which is only 16 bytes in size. The `strcpy()` function copies not only the 16 bytes for the character string, but also the trailing zero byte (NULL character).

Example 6 – Memory overwrite in C++

```
/* To compile: CRTBNDCPP PGM(EXAMPLE6) */
#include <string.h>

int main (int argc, char *argv[])
{
  char *ptr = new char[16];

  /* strcpy() copies the trailing NULL character */
  strcpy(ptr, "abcdefghijklmnop");  /* memory overwrite */
  delete [] ptr;
  return 0;
}
```

Example 7 illustrates a memory overwrite in C. A total of 14 bytes are written to a heap allocation which is only 13 bytes in size. The `strcpy()` function copies not only the 13 bytes for the character string, but also the trailing zero byte (NULL character).

Example 7 – Memory overwrite in C

```
/* To compile: CRTBNDC PGM(EXAMPLE7) */
#include <stdlib.h>
#include <string.h>

int main (int argc, char *argv[])
{
  char *ptr = (char*)malloc(13);

  /* strcpy() copies the trailing NULL character */
  strcpy(ptr, "abcdefghijklm");  /* memory overwrite */
  free(ptr);
  return 0;
}
```

Example 8 illustrates a memory overwrite in RPG. A total of 14 bytes are written to a heap allocation which is only 13 bytes in size. The based variable is 14 bytes in size.

Example 8 – Memory overwrite in RPG

```
     * To compile: CRTBNDRPG PGM(EXAMPLE8)
    H dftactgrp(*no)
    D ptr@            S               *
    D message         S             14A   BASED(ptr@)
     /free
       ptr@ = %ALLOC(13);

       // the entire 14-byte message variable is
       // updated, including trailing blanks
       message = 'hello';   // memory overwrite
       DEALLOC ptr@;

       *inlr = '1';
```

The next scenario involves deallocation of heap storage too early, while it is still being used by the application. Logic errors can allow an application to reference the heap storage after it has been deallocated and returned to the heap manager for reuse. The data referenced in such an allocation might or might not be the expected data and this can lead to intermittent failures in an application.

Example 9 is written in C++ and illustrates a write to heap storage which is no longer allocated.

Example 9 – Deallocated memory reuse

```
/* To compile: CRTBNDCPP PGM(EXAMPLE9) */
#include <string.h>

int main (int argc, char *argv[])
{
  char *ptr = new char[16];

  delete [] ptr;
  strcpy(ptr, "abc");  /* reuse of deallocated memory */
  return 0;
}
```

The next scenario involves deallocation of the same storage multiple times.

Example 10 is written in C and illustrates a duplicate call to deallocate memory.

Example 10 – A duplicate call to deallocate memory

```
/* To compile: CRTBNDC PGM(EXAMPLE10) */
#include <stdlib.h>

int main (int argc, char *argv[])
{
  char *ptr = (char*)malloc(16);

  free(ptr);
  free(ptr);  /* duplicate deallocation */
  return 0;
}
```

The last scenario is to fail in doing a deallocation for some allocated heap memory. This is called a *memory leak* because the memory *leaks* out of the heap and is no longer available for application use. Even though the memory is no longer referenced, the heap manager does not know that and so cannot reuse the memory. Memory leaks can be problematic. A significant amount of memory leak can cause performance issues or can cause an application to run out of memory. This tends to be particularly problematic in long-running applications. At some point, after the application ends (at activation group termination), the operating system reclaims all of the heap storage for the application, and therefore, the memory leak is no longer a problem.

Example 11 is written in RPG and illustrates a memory leak. Even though the application is attempting to deallocate the storage, the pointer value passed to the deallocation function is not the original pointer returned by the allocation function and so the memory is not deallocated.

Example 11 – Memory leak in RPG

```
     * To compile: CRTBNDRPG PGM(EXAMPLE11)
    H dftactgrp(*no)
    D ptr@            S               *
    D name            S             10A   BASED(ptr@)
    D i               S              5  0
     /free
       // allocate enough storage for 100 names
       ptr@ = %ALLOC(%SIZE(name) * 100);

       // move along the storage one name at a time
       for i = 1 to 100;
         name = 'something';
         ptr@ += %SIZE(name);
       endfor;

       // deallocate storage
       DEALLOC ptr@;        // memory leak

       *inlr = '1';
```

Incorrect heap usage can cause intermittent application failures, incorrect application behavior, or even corrupted data. One characteristic that makes heap problems so difficult to debug is the fact that there is often no immediate consequence for a heap error. As an example, consider a buffer overwrite in which data is written past the end of an allocated memory buffer. Symptoms of the problem typically do not arise until much later in the application when the memory that was overwritten is referenced and no longer contains the expected data.

IBM i heap memory managers

In IBM i 6.1 and later releases, three heap memory managers are provided: the default memory manager, the Quick Pool memory manager, and the debug memory manager. The memory manager in effect for a given application is controlled by the setting of the `QIBM_MALLOC_TYPE` environment variable at the time the application is run. Environment variable access to the alternate heap memory managers was provided in 6.1 with PTF 5761SS1-SI33945 and is contained in the subsequent IBM i releases. The debug memory manager was added at the same time the environment variable access was added. The Quick Pool memory manager can also be used in versions 5.4 and 6.1 by calling APIs to set up the support. Complete documentation of the heap memory manager support is available in the *ILE C/C++ Runtime Library Functions* manual.

Default memory manager

The default memory manager is a general-purpose memory manager that attempts to balance the performance and memory requirements. It provides adequate performance for most of the applications while attempting to minimize the amount of additional memory needed for overhead. The default memory manager is the preferred choice for most of the applications and the memory manager is enabled by default. If the `IBM_MALLOC_TYPE` environment variable is not set or is set to an unrecognized value, the default memory manager is used.

Quick Pool memory manager

The Quick Pool memory manager breaks up the memory into a series of pools. It is intended to improve heap performance for applications that issue large numbers of small allocation requests.

When the Quick Pool memory manager is enabled, allocation requests for sizes within a given range of allocation sizes are allocated a fixed size cell within a pool. These requests can be handled more quickly than requests for sizes outside this range. Allocation requests outside this range are handled in the same manner as the default memory manager.

The Quick Pool memory manager is not enabled by default, but can be enabled by setting the following environment variable:

```
QIBM_MALLOC_TYPE=QUICKPOOL
```

The Quick Pool memory manager can also be enabled using API calls within an application. Refer to the *ILE C/C++ Runtime Library Functions* manual for more information.

Debug memory manager

The debug memory manager is used primarily to find the incorrect heap usage by an application. It is not optimized for performance and might negatively affect the performance of the application. However, it is valuable for the determination of incorrect heap usage.

Memory problems detected by the debug memory manager result in one of the two behaviors:

- If the problem is detected at the time when the incorrect usage occurs, a machine check handler (MCH) exception message (typically an MCH0601, MCH3402, or MCH6801) is generated. In this case, the error message typically stops the application.
- If the problem is not detected until later, after the incorrect usage has already occurred, a C2M1212 message is generated. In this case, the message does not typically stop the application.

The debug memory manager detects memory problems in two ways:

- First, it uses restricted access memory pages. A memory page with restricted access is placed before and after each allocation. Each memory block is aligned on a 16 byte boundary and placed as close to the end of a page as possible. As memory protection is allowed only on a page boundary, this alignment allows the best detection of memory overwrites and memory over reads. Any read or write from one of the restricted access memory pages immediately results in an MCH exception.
- Second, it uses padding bytes before and after each allocation. A few bytes immediately before each allocation are initialized at the allocation time to a preset byte pattern. Any padding bytes, following the allocation required to round the allocation size to a multiple of 16 bytes, are also initialized at the allocation time to a preset byte pattern. When the allocation is deallocated, the padding bytes are verified to ensure that they still contain the expected preset byte pattern. If any of the padding bytes have been modified, the debug memory manager generates a C2M1212 message with the reason code X'80000000'.

The debug memory manager is not enabled by default, but can be enabled by setting the following environment variable:

```
QIBM_MALLOC_TYPE=DEBUG
```

Debugging common problems with heap memory

Earlier, several common problems with the usage of heap memory were listed and several example programs were given to illustrate various heap problems. The debug memory manager allows detection of many common problems with heap memory including: Memory overwrites, memory over reads, reuse of deallocated memory, and duplicate deallocations. The debug memory manager does not detect memory leaks. Detecting memory leaks within the ILE applications is a topic to be covered in a future article.

The behavior of each of the example programs, which illustrate the heap problems, is described when the programs are run using the debug memory manager.

Example 6 illustrates a memory overwrite. It illustrates an attempt to write beyond the end of a data item that is a multiple of 16 bytes in size. Compiling the example as a single-level storage program and running it with the debug memory manager yields an MCH0601 message at the point where the memory overwrite occurs. Compiling the example as a teraspace storage program and running it with the debug memory manager yields an MCH6801 message at the point where the memory overwrite occurs. In each case, the message details will point to the statement that is doing the memory overwrite. The example illustrates a memory overwrite, but a memory over read would get the same result.

Example 7 illustrates a memory overwrite. It illustrates a memory write which does not cross a 16 byte boundary. Compiling the example as a single-level storage program or a teraspace program and running it with the debug memory manager yields a C2M1212 message with a reason code of X'80000000'. The message details will point to the statement, which is calling `free()`. A memory over read which does not cross a 16 byte boundary is not detected by the debug memory manager.

Example 8 illustrates a memory overwrite. Because the debug memory manager is not enabled by default for RPG applications, the memory overwrite is not detected. An IBM i 7.1 enhancement added the ALLOC(*TERASPACE) control specification keyword to RPG. This keyword instructs the RPG runtime to use the C runtime teraspace heap functions for memory allocation and deallocation and allows the debug memory manager to be used for RPG applications. When this is done, this example will have the same behavior as Example 7. For additional details on the ALLOC keyword, refer to the *ILE RPG Language Reference* manual.

Example 9 illustrates a write to the heap storage which is no longer allocated. Compiling the example as a single-level storage program and running it with the debug memory manager yields an MCH3402 message at the point where the memory write occurs. Compiling the example as a teraspace storage program and running it with the debug memory manager yields an MCH6801 message at the point where the memory write occurs. In each case, the message details will point to the statement which is doing the memory write. The example illustrates a memory write, but a memory read would get the same result.

Example 10 illustrates a duplicate call to deallocate memory. Compiling the example as a single-level storage program or a teraspace program and running it with the debug memory manager yields a C2M1212 message. The message details will point to the statement, which is calling `free()`.

Example 11 illustrates a memory leak. As indicated earlier, the debug memory manager does not detect memory leaks.

When the debug memory manager is not used, these memory problems go undetected and have the potential to cause intermittent application failures, incorrect application behavior, or corrupted data.

Heap mystery solved

The ability to understand what heap memory is and how to correctly use heap memory is critical to writing and maintaining ILE applications. Knowledge and awareness of the common heap problems can be combined with the debug memory manager to easily detect heap problems within an application and quickly put an end to heap memory problems in IBM i ILE applications.

Resources

- Visit the IBM i 7.1 Information Center to get more information about the following topics:
  - ILE Concepts
  - ILE RPG Language Reference
  - ILE C/C++ Runtime Library Functions
  - IBM i forum
  - RPG Cafe

This content was originally published in the August 2010 issue of *iProDeveloper*.