Chapter 8 Memory and the Memory Manager

ROM Unlocks handles
(hlock and movehi slows programs down w/code) 118
Completion routine questions and anwsers..........118
Segmentation (tips on using unloadseg)...............119
Tips on unloadseg (and a story)...........................120
Handles and Virtual Memory (rewriting the Mem Mgr) 121
Setting up multiple heaps......................................121
NewPtrclear,NewhandleClear...............................122
NewPtrclear,NewhandleClear...............................123
StripAddress and pointer arithmetic.....................124
StripAddress and pointer arithmetic.....................124
IIci Trap dispatch Table...125
Identifying real handles (with Code).....................125
Identifying real handles...126
Need large array on Think C.................................126
Need large array on Think C.................................127
Need large array on Think C. (explanation on how to do it)
....127
Need large array on Think C.................................128
Identifying real handles...129

From: oster@dewey.soe.berkeley.edu (David Phillip Oster)
Subject: Re: ROM Unlocks handles (hlock and movehi slows programs down w/code)

In article <3842@atr-la.atr.co.jp> alain@atr-la.atr.co.jp (Alain de Cheveigne) writes:

>But hassle it is. Is there a better way ?

Yes. What complexity, and slowness! (all those unnecessary MovHHi()s can really slow a program down.) I too use very similar data structrues: handles in the RefCons of my windows that themselves reference other handles.

If a field referenced by an O.S. call is small, copy it. If it is large, lock the owning handle during the call. Do not write your own routines to take pointers into handles, instead, if you must do such a thing, do it by passing a handle an an offset. (Often you can just pass the handle, since the routine knows its offset.)

Here is an example, from a program that uses a window handle which references two pixmaps. (I keep the current window handle in a global variable called "theDoc", by analogy with "thePort".) Notice, no need for any HLocks or MoveHHis.

```
/* DocDispose - discard our window
*/
DocDispose(){
    if(NIL != (**theDoc).newPix){
        if(NIL != (**(**theDoc).newPix).baseAddr){
            DisposPtr((**(**theDoc).newPix).baseAddr);
            (**(**theDoc).newPix).baseAddr = NIL;
        }
        DisposPixMap((**theDoc).newPix);
    }
    if(NIL != (**theDoc).oldPix){
        if(NIL != (**(**theDoc).oldPix).baseAddr){
            DisposPtr((**(**theDoc).oldPix).baseAddr);
            (**(**theDoc).oldPix).baseAddr = NIL;
        }
        DisposPixMap((**theDoc).oldPix);
    }
    DisposHandle((Handle) theDoc);
    DisposeWindow(thePort);
}
```

Actually, this does bring up a question. Inside Mac Vol 5 doesn't say precisely which fields are disposed when you call DisposPixMap().

--- David Phillip Oster

●●●

From:Rick Holzgrafe
Subject: **Completion routine questions and anwsers**

In article <6732@hubcap.clemson.edu> mikeoro@hubcap.clemson.edu (Michael K O'Rourke) writes:

> If i am in the background and enter the iocompletion routine, is it
> possible to get interrupted or swapped out halfway thru the completion
> routine? I am having some funky errors when running in the background
> under multifinder and can't seem to figure out what is going on.

Your completion routine may be called from a higher-than-normal interrupt level, but not at the highest possible level. A higher-level interrupt can, um, interrupt you. But this shouldn't bother you any more than *your* interrupt bothers whoever you interrupted. MultiFinder in particular won't swap processes without being deliberately called: don't make any "Event" calls (e.g. WaitNextEvent, GetNextEvent) and you won't be yielding control to MultiFinder.

I don't know what your specific problem is, but here's some general hints:

- Keep completion routines SHORT. Best is to quickly store results someplace where your main-line code can find them, and get out.

- Minimize your stack usage in completion routines. You don't know whose stack you're running in, or whose heap you'll trash if you overflow.

- You can't call the Memory Manager from a completion routine. This means you can't call ANY routine which may move or purge memory: see the appendix in Inside Mac for a list of these routines. (I bet you knew that one already. :-)

- You can make new asynchronous calls from a completion routine. (But beware of the pitfalls mentioned in these tips.) You cannot make synchronous calls from completion routines.

- Be aware that your completion routine can (in some cases) be called BEFORE your async PBControl call returns! Don't rely on having any time after the PBControl to get ready for the completion. And if you're making new async calls in your completion routines, beware of "recursion" and stack overflow.

- Remember that your completion routine can be interrupted by a higher-priority interrupt, and that it will itself be interruptingyour main-line code. You may need to protect critical code or data. If you do this (typically by temporarily forcing the interrupt level high), do it as rarely and as quickly as you can.

- Be sure your code segment containing the completion routine is loaded, locked, and unpurgeable. Don't unload it!

- Be careful about A5 (access to global variables, and calls to routines outside the segment). A5 is not guaranteed correct in a completion routine.

- Don't allocate your parameter blocks as local variables of routines which may return before the completion.

Hope this helps.

-
Rick Holzgrafe

●●●

From: lippin@spam.berkeley.edu (The Apathist)
Subject: Re: Segmentation (tips on using unloadseg)

Recently tim@hoptoad.UUCP (Tim Maroney) wrote:
>And sure, you can move things into smaller partitions this way, as long
>as you don't care about certain minor facts like keeping to the
>interface guidelines. User operations are supposed to begin taking
>effect instantaneously. If they require fetching in a large resource
>from the application file, then they are anything but instantaneous.

Remember, those segments aren't getting swapped out unless you needed the memory for something else. When your user is pushing the program to where this would matter, the difference between unloading and not unloading is the difference between running slowly and not running at all. I think it's clear which one the user wants.

>You also ignored the function pointer issue. I received a letter from
>a person at Apple that I respect and who is usually right, but not this
>time. He says that you should put all function-pointer fetching
>routines into the main code segment: that way, they will be jump table
>pointers. This may work fine for code that uses one or two function
>pointers and calls them all itself. However, just imagine passing a
>function pointer into the jump table to the vertical retrace manager or
>as a completion routine to the file manager. Can you say "intermittent
>crash when A5 is not CurrentA5"? I knew you could.

Both of you got this one wrong. Any development system worth its salt, and certainly any I've used, will create function pointers as pointers into the jump table, precisely so that they're always valid. They're truly absolute pointers, so they don't care one whit about CurrentA5 (although your code might; that's the well-known gotcha of interrupt tasks, and has established solutions.)

[user-confusion caused by bugs that don't happen deleted]

>I prefer instant response and program reliability, as well as the
>ability to use function pointers freely in my code. There are really
>very few people who need to use more than two or three programs in
>conjunction with the Finder at once, and those people can get SIMMs
>pretty cheap these days. Real RAM hogs like MPW Shell and FullWrite
>may need to use it, but my programs rarely get over 250K.

Wow! You make it sound like having teeth pulled! It's not so difficult as all that.

Tom's three easy rules to a long and happy life using UnloadSeg:

1. Put the trap glue in your main segment.
2. Put your low memory emergency code in your main segment.
3. Put your main event loop in your main segment, and have it call UnloadSeg on all other segments, often.

Clever programmers can improve somewhat on this, depending on the structure of their particular programs. But this will work well, and your users would appreciate it, if somebody explained the difference to them.

After all, this is the computer for the rest of them.

--Tom Lippincott

●●●

From: oster@dewey.soe.berkeley.edu (David Phillip Oster)
Subject: Tips on unloadseg (and a story)

I recently HAD to use UnloadSeg(). The marketing manager called me on one of the products I was writing and said, "Help, the program is now so big it will only allow 2400 records on a 1Meg MacPlus. The box says 2600. It will take six weeks to make new boxes and we need to ship NOW!"

So, I had to use unloadseg to get space when the user's files got large.

Here is the big problem: There MUST be space to load a required segment. If the o.s. can't find space, it will crash. You are making an innocent, arbitrary procedure call, and _kaboom_. Hard to debug and hard to fix.

Some work arounds:

1.) You can do a setjump in your main loop, and a longjump in your growzone routine. This should sort of work. Note though, your growzone function is called whenever anything needs memory including the o.s. doing a longjump from inside the o.s. is a good way to leave it in an inconstant state. For your own code, you can use Signal (see the tech notes,) and declare a cleanup routine that will get executed during the longjump, but it isn't easy to retrofit this kind of thing into an almost done 40,000 line program.

2.) Your development system will let you find out what segment (by number)each procedure is in. You can add a few:

```
if(NIL == GetResource('CODE', MAINDOCUPDATESEGNUM)){
    Error(IAMSORRYDAVE);
}
```

that is, the program makes a probe to see if it will be able to get the segment it needs to do a command. If it can't the user sees an error message. (This assumes the error message code is in the main segment.) This is technique I used. My programs are object oriented, so my message dispatcher can return the error code MESSAGEUNDELIVERABLENOTATHOME, if I send a message to an object whose code can not be read.

This has some problems: I have to manage my own symbols for my segments, which might get out of date as I rearrange my procedures. I wish there were a system call: CanLoadSeg(procedureName) which returns NIL or a handle to the segment. It always bothers me when I see a function with an obvious inverse that is not implemented. (Yes, I know I could write it, but would it work in System 7?)

My next program will use technique 1, unless you good folk can tell me a better way to do things.

I usually only unload segments in my main loop, but occasionally, when I need extra room (for example during a Save or a spool Print.) I call a procedure that unlocks all purgable segments except 1: the save segment for save or the print segment for Print.

This issue is only a small part of the larger issue of memory management in Mac programs: for example: should your grow zone function throw away undo information as a last ditch attempt to make extra room? How about if you need the room while in the middle of executing a ReDo command?

The simple answer is: don't print "max sizes handled" on your boxes. Have your program enforce conservative limits on the user. Maybe I'm just taking "the power to be your best" too much to heart.

You may have noticed that I answer more questions than I ask. Am I trying too hard on this issue?

> drs@bnlux0.bnl.gov (David R. Stampf)
--- David Phillip Oster -master of the ad hoc odd hack.

Summary/Conclusion: if your mac program calls a procedure in a segment that isn't loaded, the segment loader will call LoadSeg() to fetch it. If LoadSeg() fails, the mac crashes. This can bite you even if you never call UnloadSeg() since a segment only gets loaded when you call it for the first time (which might be AFTER you read a giant file.)

●●●

From: oster@dewey.soe.berkeley.edu (David Phillip Oster)
Subject: Re: **Handles and Virtual Memory (rewriting the Mem Mgr)**

In article <1669@intercon.com> amanda@mermaid.intercon.com (Amanda Walker) writes:

>Indeed, I've often been tempted to write a rudimentary version of the Mac
>memory manager for things like UNIX machines or PCs, simply so that I
>can have relocatable (and more importantly, *resizable*) blocks of memory
>without having to do my own memory management & reallocation.

Go for it! It will take you about 1 minute: consider:

```
typedef char *Ptr, **Handle;

Handle UnixNewHandle(size)long size;{
    Handle h;

    h = (Handle) malloc(sizeof(Handle));
    *h = (Ptr) malloc(size);
    return h;
}

void SetHandleSize(h, newSize)Handle h;long newSize;{
    realloc(*h, newsize);
}
```

does almost all the work for you. Error handling is left as an exercise. (The above assumes you have a virtual memory system, and a decent implementation of malloc(), so you don't have to worry about fragmentation of the heap.)

--- David Phillip Oster

●●●

From:siegel@endor.UUCP (Rich Siegel)
Subject: Re: Setting up multiple heaps

In article <21673@cup.portal.com> Andrew_James_Peterson@cup.portal.com writes:

>I'm going to try to set up an application that uses multiple heaps.

OK, but why?

Setting up a new heap zone is pretty simple:

```
       var
               hcZone: THz;

       procedure Heap;
               var
                       oldzone: THz;

       begin
               OldZone := GetZone;
               hcZone := THz(NewPtr(MaxZoneSize));
               InitZone(nil, 16, Ptr(Ord4(hcZone) + MaxZoneSize), Ptr(hcZone));
               SetZone(OldZone);
       end;
```

To make your new zone the current zone, do a

```
       SetZone(hcZone);
```

Multiple heaps generally aren't required, unless you're THINK Pascal or you want to perform some action on your own heap which requires blocks to be allocated, and you don't want to affect your own heap.

Rich Siegel

•••

From:Earle R. Horton
Subject: Re: NewPtrclear,NewhandleClear

Matthew,

In article <3444@unix.SRI.COM> you write:
>I'm trying to get Offscreen sample source code from Apple to compile
>in LSP and its asking for Newptrclear and NewHandle clear. I think these are
>defined in MPW 3.0 but I can't afford to upgrade. Does someone have these
>definitions that they can send to me?

These are object dumps of part of the Interface.o library that comes with MPW 3.0, along with the C prototypes. They differ from the standard functions in that the trap word has an extra bit set so the system knows you want cleared space.

Earle R. Horton

```
dumpobj -h -m NEWPTRCLEAR {libraries}interface.o

Dump of file Boot:MPW:Libraries:Libraries:interface.o

Module:              Flags $08 Module="NEWPTRCLEAR"(309) Segment="Main"(273)
Content:             Flags $08
Contents offset $0000 size $000C
         MOVEA.L    (A7)+,A1
         MOVE.L     (A7)+,D0
         _NewPtr    ,Immed              ; A31E
         MOVE.L     A0,(A7)
         JMP        SAVERETA1           ; id: 287

dumpobj  -m SAVERETA1 {libraries}interface.o
```

```
Dump of file Boot:MPW:Libraries:Libraries:interface.o

; Note: SAVERETA1 starts here.  It stuffs the result code from D0
; into MemErr, then returns to the address stashed in A1.
0000000C: 2F09          '/.'            MOVE.L     A1,-(A7)
0000000E: 31C0 0220     '1.. '          MOVE.W     D0,$0220
00000012: 4E75          'Nu'            RTS

dumpobj -h -m NEWHANDLECLEAR {libraries}interface.o
Dump of file Boot:MPW:Libraries:Libraries:interface.o

Module:              Flags $08 Module="NEWHANDLECLEAR"(326) Segment="Main"(273)
Content:             Flags $08
Contents offset $0000 size $000C
         MOVEA.L    (A7)+,A1
         MOVE.L     (A7)+,D0
         _NewHandle  ,Immed               ; A322
         MOVE.L     A0,(A7)
         JMP        SAVERETA1            ; id: 287

pascal Handle NewHandleClear(Size byteCount);
pascal Ptr NewPtrClear(Size byteCount);
```

Note: These are declared as Pascal, and so exist in the object file, {libraries}interface.o, as uppercase symbols.

●●●

From:siegel@endor.UUCP (Rich Siegel)
Subject:NewPtrclear,NewhandleClear

I wrote these inlines in 1987. (!!) Just change the names of the inlines, and you're in business.

```
FUNTION NewClearPtr (logicalSize : LongInt) : Ptr;
INLINE
        $201F, {move.l (a7)+, d0}
        $A31E, {_NewPtr, CLEAR}
        $2E88; {move.l a0, (a7)}
{Allocates a zeroed block of memory using _NewPtr, CLEAR}

FUNCTION NewSysPtr (logicalSize : LongInt) : Ptr;
INLINE
        $201F, {move.l (a7)+, d0}
        $A51E, {_NewPtr, SYS}
        $2E88; {move.l a0, (a7)}
{Allocates a block of memory in the system heap}

FUNCTION NewSysClearPtr (logicalSize : LongInt) : Ptr;
INLINE
        $201F, {move.l (a7)+, d0}
        $A71E, {_NewPtr, CLEAR+SYS}
        $2E88; {move.l a0, (a7)}

{Allocates a zeroed block in the system heap}

FUNCTION NewClearHandle (logicalSize : LongInt) : Handle;
INLINE
        $201F, {move.l (a7)+, d0}
        $A322, {_NewHandle, CLEAR}
        $2E88; {move.l a0, (a7)}
{Allocates a zeroed relocatable block}

FUNCTION NewSysHandle (logicalSize : LongInt) : Handle;
INLINE
        $201F, {move.l (a7)+, d0}
```

```
          $A522, {_NewHandle, SYS}
          $2E88; {move.l a0, (a7)}
     {Allocates a relocatable block in the system heap}
```

```
FUNCTION NewSysClearHandle (logicalSize : LongInt) : Handle;
INLINE
        $201F, {move.l (a7)+, d0}
        $A722, {_NewHandle, SYS+CLEAR}
        $2E88; {move.l a0, (a7)}
{Allocates a zeroed relocatable block in the system heap}
```

•••

From: beard@ux1.lbl.gov (Patrick C Beard)
Subject: Re: StripAddress and pointer arithmetic
Summary: Only addresses which are master pointers.

In article <1990May25.194025.9751@csrd.uiuc.edu> bruner@sp15.csrd.uiuc.edu (John Bruner) writes:

```
#I've just read the technical note on StripAddress (#213, April 1990).
#On the second page under the (sub)heading "Ordered Address Comparison"
#it says
#
#       If you need to sort by address or do any other kind of
#       ordered address comparison, you need to call _StripAddress
#       on each address before doing any ordered comparisons
#       (>, <, >=, <=).  Remember, even though the CPU only uses
#       the lower 24 bits in 24-bit mode, it still uses all 32 bits
#       when performing arithmetic operations.
#
#Taken literally, this means that it is unsafe to write
#
#       struct something *p, things[64];
#
#       for (p = things; p < things + 64; p++)
#               ...
#
#because one can't rely upon the comparison "p < things+64" working.
```

Ok, time for a clarification. Note that your example implies variables that are allocated on the stack. This code will always work. Address arithmetic only has problems when you are dealing with addresses that might be dereferenced handles, or "master pointers". This is because the *current* memory manager keeps state information in the high byte (actually high nibble) of master pointers (i.e. if a handle is locked, purgeable, or a resource). These bits will throw off address comparisons obviously, and so StripAddress is in order. In general, addresses are clean. Examples of where they might not be include: dereferenced handles, code resources dereferenced then called will put garbage in the pc, and so should be stripped before being called.

A 32-bit operating system will make all this stuff go away. Let's hope we get it soon.
-
Patrick Beard, Macintosh Programmer (beard@lbl.gov) -

•••

From: shebanow@Apple.COM (Andrew Shebanow)
Subject: Re: StripAddress and pointer arithmetic

In article <1990May25.194025.9751@csrd.uiuc.edu> bruner@sp15.csrd.uiuc.edu (John Bruner) writes:

```
>Taken literally, this means that it is unsafe to write
>
>       struct something *p, things[64];
>
>       for (p = things; p < things + 64; p++)
>               ...
```

> >
> >because one can't rely upon the comparison "p < things+64" working.

I wrote that Tech Note, and I agree that it could be clearer (sigh).

The type of loop you show above will work fine without calling StripAddress. The only time you have to worry is if you are comparing two arbitrary (and possibly unrelated) addresses (for instance, if you compare a pointer parameter to a pointer stored in a list when doing a sort by address).

Sorry for the confusion,

Andrew Shebanow

●●●

From: tecot@Apple.COM (Ed Tecot)
Subject: Re: IIci Trap dispatch Table

In article <4925@daffy.cs.wisc.edu> upl@gumby.cs.wisc.edu (Undergrad Projects Lab) writes:

>I am working on a program which depends on the location of the IIci Trap
>Dispatch Tables. The problem is, they don't seem to be in the place described
>in IM IV (page 13). I have been able to find the OS Traps, but I need to know
>where the Toolbox Traps are. Is there any one out there (APPLE?) who can give
>me that info?

"I can tell you, but I'll then I'll have to kill you."

Bad news. The trap dispatch tables are not guaranteed to be in a certain location. They've been forced to move with each new ROM and system as the size and format of the table grows. You could probably reverse-engineer the information for your particular ROM and system, but I can guarantee that the location will change on other machines and in the future.

"There is another..."

Use NGetTrapAddress to access the dispatch table. It always knows where to find it, even if it's split into several pieces.

"Live long and prosper."

I hope I've helped you.

_emt

●●●

From: lim@iris.ucdavis.edu (Lloyd Lim)
Subject: Identifying real handles (with Code)

This is perhaps not the best thing to do on a Mac but I need to be able to tell if some arbitrary long is a real handle in the System heap or the current app heap. I want to keep it relatively clean so I don't want to go looking through the heap's internal structures. Currently, I use the following code:

```
Boolean ValidHandle(address)

register long address;

{
  register Boolean  valid;
  register THz      heapZone;

  valid = FALSE;
  if (address && !(address & 1)) {
     heapZone = HandleZone(address);
     if (!MemError() && (heapZone == SystemZone() ||
                         heapZone == ApplicZone())) {
```

```
        valid = TRUE;
    }
```

```
  }
  return(valid);
}
```

The problem is that this routine can get passed any arbitrary long and that some values seem to cause a bus error with HandleZone. I say "seem" because I haven't been able to find a value which causes a bus error in a test program but bus errors do occur in the real situation if enough values are examined. The real situation is practically unobservable.

Any ideas on a better way? If you try doing extra checking before calling HandleZone, you'll get bus errors right away when you do (*(Ptr) address). If there is a solution that isn't right 100% of the time but doesn't cause bus errors, that would also be ok.

+++
Lloyd Lim   Internet: lim@iris.ucdavis.edu (128.120.57.20)

●●●

From: russotto@eng.umd.edu (Matthew T. Russotto)
Subject: Re: Identifying real handles

In article <7426@ucdavis.ucdavis.edu> lim@iris.ucdavis.edu (Lloyd Lim) writes:
>This is perhaps not the best thing to do on a Mac but I need to be able to
>tell if some arbitrary long is a real handle in the System heap or the current
>app heap. I want to keep it relatively clean so I don't want to go looking
>through the heap's internal structures. Currently, I use the following code:

Check to see if the handle itself is below BufPtr or MemTop or whatever variable you feel is appropriate. If it's below that, there should never be a bus error. Then do validity checks on *address, and check to see if IT is below BufPtr or MemTop. Then call HandleZone.

--
Matthew T. Russotto   russotto@eng.umd.edu   russotto@wam.umd.edu

●●●

From: ted@cs.utexas.edu (Ted Woodward)
Subject: Re: Need large array on Think C.

In article <265986A9.26645@paris.ics.uci.edu> jchoi@paris.ics.uci.edu (John Choi) writes:

> Sorry for this trival question, but I really need to get this done.

no prob; that's what this group is here for...

>When I declare an array ('char arr[100][300]'), the complier
>gives me an 'illegal array bounds' error. Is there any way I can
>increase this limit?

You are limited to 32K of global and static variables because of the compiler. This is because the 68000 can only have a 16 bit offset to an address reg, the technique used for global vars. But there is a way. Read on...

> The program uses array notation throughout and I don't want to
>recode using pointers and malloc(). I just need to increase the array
>size. Is there a complier option I need to set using Think C 4.0

No compiler option. You have to use pointers. But, because of the way C does arrays and pointers, you can address a pointer as an array. Because you have a multidimensional array, C needs to know how big each dimension is. Actually, C only needs to know how big the 1st dimension is, and doesn't care how big the second is (same as when you declare an array).

Try this:

```
#define MAXX 100
#define MAXY 300

char (*arr)[MAXX-1]  /* C arrays go from 0-n */
       /* this declares a pointer to an array of MAXX chars */

main()
{   arr = (char *) NewPtr(sizeof(char) * (long) MAXX * MAXY);
etc...
}
```

this will give you the array, addressable like a normal array. You need to declare it like above so it knows how many elements in the 1st parameter, and the NewPtr actually allocates the space on the heap. You could (and possibly should) use a handle, but this makes it hard to access in this case. Instead, do this FIRST so you don't frag the heap. You don't really need the sizeof(char) because that is 1 byte for char, but you do need the type coercion to long there because NewPtr wants a long, and because you might overflow an int.

> Thanks for the help.

No prob. Hope this solves the problem...

--
Ted Woodward (ted@cs.utexas.edu)

●●●

From: bruner@sp15.csrd.uiuc.edu (John Bruner)
Subject: Re: Need large array on Think C.

In article <736@grit.cs.utexas.edu>, ted@cs (Ted Woodward) writes:

```
]In article <265986A9.26645@paris.ics.uci.edu> jchoi@paris.ics.uci.edu (John Choi) writes:
]>When I declare an array ('char arr[100][300]'), the complier
]>gives me an 'illegal array bounds' error.  Is there any way I can
]>increase this limit?
]
]Try this:
]
]#define MAXX 100
]#define MAXY 300
]
]char (*arr)[MAXX-1]  /* C arrays go from 0-n */
]                      /* this declares a pointer to an array of MAXX chars */
]
]main()
]{      arr = (char *) NewPtr(sizeof(char) * (long) MAXX * MAXY);
]etc...
]}
```

Ted is close, but not quite correct. You need to declare the pointer to the array using

```
    char (*arr)[MAXY];
```

This technique works if the dimensions of the array (except the most-significant one) are constants at compile time; otherwise, you need to use arrays of pointers and multiple levels of indirection (using "arr[x][y]" notation) or explicit index calculation.

John Bruner Center for Supercomputing R&D, University of Illinois

●●●

From: stoms@castor.ncgia.ucsb.edu (David Stoms)
Subject: Re: Need large array on Think C. (explanation on how to do it)

In article <265986A9.26645@paris.ics.uci.edu> jchoi@paris.ics.uci.edu (John Choi) writes:

> Sorry for this trival question, but I really need to get this
>done. When I declare an array ('char arr[100][300]'), the complier
>gives me an 'illegal array bounds' error. Is there any way I can
>increase this limit?

I'm getting tired of this question so I think I'll try to give a generic answer to cover any further problems.

When you want lots of variable space >32K in Think C or any compiler that limits globals to 32K, you have to use the Memory Manager (gasp!). If you want a big array, such as "char arr[100][400]" then you need to sever your dependancy from the compiler and get your hands dirty.

To allocate this array:

```
char   *arr;

arr = NewPtr((Size)100*400);
```

Then to use the array you can index like this:

```
x = 20; y = 32;
*(arr+x+y*100) = 'H';
```

Thats it! If you want to use a handle the only big difference is:

```
*(*arr+x+y*100)  << note the extra * >>
```

If you do this, be sure to lock down the handle when your r.h.s. could move memory.

Josh.

●●●

From: dudek@ai.toronto.edu (Gregory Dudek)
Subject: Re: Need large array on Think C.

In article <5506@hub.ucsb.edu> stoms@castor.ncgia.ucsb.edu () writes:

>In article <265986A9.26645@paris.ics.uci.edu> jchoi@paris.ics.uci.edu (John Choi) writes:
>> Sorry for this trival question, but I really need to get this
>>done. When I declare an array ('char arr[100][300]'), the complier
>>gives me an 'illegal array bounds' error. Is there any way I can
>>increase this limit?
>
>
>When you want lots of variable space >32K in Think C or any compiler
>that limits globals to 32K, you have to use the Memory Manager (gasp!).
>If you want a big array, such as "char arr[100][400]" then you need to
>sever your dependancy from the compiler and get your hands dirty.
>
>To allocate this array:
>
>char   *arr;
>
>arr = NewPtr((Size)100*400);
>
>Then to use the array you can index like this:
>x = 20; y = 32;
>*(arr+x+y*100) = 'H';
>
>Thats it! If you want to use a handle the only big difference is:
>*(*arr+x+y*100) << note the extra * >>

```
>
>If you do this, be sure to lock down the handle when your r.h.s. could
```

Although the method above works fine, I find it a bit ugly. Not only does sticking in calls to NewPtr explicitly lose portability, but the indexing scheme is much less readable than regular arrays. Finally, you may want to have the arrays initialized to zero, like ``real'' ones.

I prefer the following scheme which is much more isomorphic to the commonly used sytax. Note, also, that since the arrays are allocated using pointers not handles, no locking down of memory is required at any time.

1) Add this header code:
======================

```
#define DCLARRAY(type,name,y,x)   type *name[y]
#define INITARRAY(type,name,y,x)  allocate(name,(int)sizeof(type),y,x)


allocate(array,elementsize,y,x)
char *array[];
int elementsize;
int y,x;
{
   register int ix, iy;
   for (iy=0;iy<y;iy++) {
    array[iy] = NewPtr((long)(x*(long)elementsize));
    if (!array[iy]) exit(1);
   }
   /* init array to zero here, if desired */
}
```

2) Replace array declarations of the form:

```
 float foo[999][566]       with
 DCLARRAY(float,foo,999,566)
```

3) Insert this before the array would be used:

```
    INITARRAY(float,foo,999,566)
```

4) Use the array like normal, i.e. foo[23][21]

Greg Dudek

●●●

From: odawa@well.sf.ca.us (Michael Odawa)
Subject: Re: Identifying real handles

In article <7426@ucdavis.ucdavis.edu> lim@iris.ucdavis.edu (Lloyd Lim) writes:

> \> ...I need to be able to tell if some arbitrary long is a real handle in
> \> the System heap or the current app heap. I want to keep it relatively clean
> \> so I don't want to go looking through the heap's internal structures...The
> \> problem is that this routine can get passed any arbitrary long and that some
> \> values seem to cause a bus error with HandleZone....Any ideas on a better
> \> way?

You have to make a range check against the high end of your memory. Here's something you might add to your code:

```
{
  register Boolean  valid;
  register THz      heapZone;
```

```
valid = FALSE;
if (address && !(address & 1)) {
    if (address < long(**ApplicZone.BkLim)) { /***** Add this line *****/
```

```
         heapZone = HandleZone(address);
     if (!MemError() && (heapZone == SystemZone() ||
                           heapZone == ApplicZone())) {
           valid = TRUE;
     }
  }
  return(valid);
}
```

A different check (which I do in my version of ValidHandle) might be to determine whether the tag byte on the block header (the first of the eight bytes which preceed the address) contained 0x8x, as documented in IM II-24:

```
   if (address < long(**ApplicZone.BkLim))
      if ((*(ptr)(address - 8) & 0x80) != 0)
         valid = TRUE;
```

Of course, we have not even begun to discuss problems with 24-bit vs 32-bit addressing schemes. So perhaps we ought to preceed our code by

```
   address = (long)StripAddress((ptr)address);
```

Michael Odawa

●●●