Chapter 2

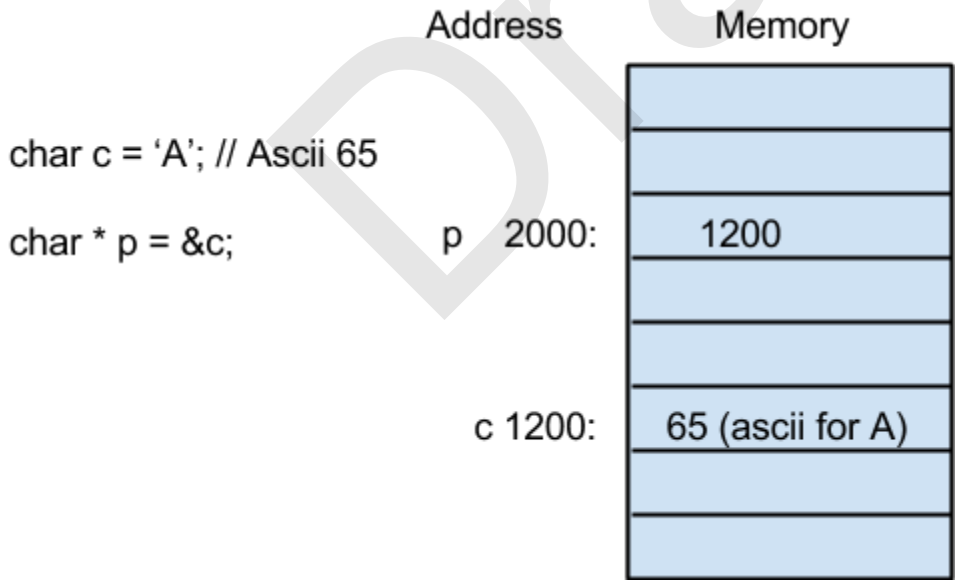
Review of Pointers and Memory Allocation

1.1. Introduction

In this chapter we will talk about the pointers in C. Pointers are fundamental in understanding how variables are passed by reference, how to access arrays more efficiently, and also memory allocation. We will start by reviewing how pointers are defined in “C”, and then we will continue with explaining some basic data structures that use pointers, and we will finish with pointer to functions.

1.2 Memory and Pointers

A pointer is a variable that contains an address in memory. In a 64 bit architectures, the size of a pointer is 8 bytes independent on the type of the pointer. Below you can see a graphical representation of memory along with a pointer. Notice the pointer p contains the address of the variable c, and not the value contained there. You would create a pointer like this in the manner shown to the left of the image.



1.3 Ways to get a pointer value

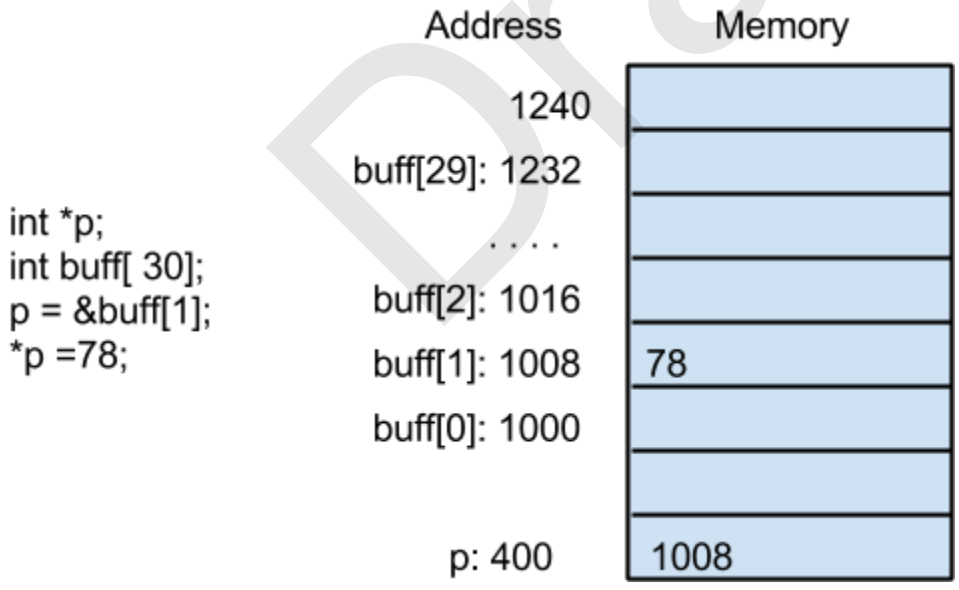
➔ Assign a numerical value into a pointer

```
char * p = (char *) 0x1800;
*p = 5; // Store a 5 in location 0x1800;
```

Above, you can see how you would go about creating a pointer to a location, and then assigning a value to the memory location the pointer is pointing to. Note: Assigning a numerical value to a pointer isn't recommended and only left to programmers of OS, kernels, or device drivers

➔ Get memory address from another variable:

Below you can see a graphical representation of how you would retrieve the address of a specific index in an array and then assign a value to that index.



➔ Allocate memory from the heap

Here we can see an example of assigning memory from the heap to pointers using both the new operator from C++ and the malloc function call used in C. While using malloc in C++ is entirely valid, you will often tend to use new when allocating data structures to ensure their constructor is called to properly set up the variable.

```
int *p
p = new int;
int *q;
q = (int*)malloc(sizeof(int))
```

➔ **Pass a pointer as a parameter to a function (Pass by Reference)**

```
void swap (int *a, int *b){
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}

In main:
        swap(&x, &y)
```

1.4 Common Problems with Pointers

➔ When using pointers make sure the pointer is pointing to valid memory before assigning or getting any value from the location.

➔ String functions do not allocate memory for you:

char *s;
strcpy(s, "hello"); --> SEGV(uninitialized pointer)

➔ The only string function that allocates memory is strdup (it calls malloc of the length of the string and copies it)

1.5 Printing Pointers

It is useful to print pointers for debugging

```
char*i;
char buff[10];
printf("ptr=%d\n", &buff[5])
```

Or In hexadecimal

```
printf("ptr=0x%x\n", &buff[5])
```

Instead of using ***printf***, We recommend to use ***fprintf(stderr, …)*** since stderr is unbuffered and it is guaranteed to be printed on the screen.

1.6 sizeof() operator in Pointers

The size of a pointer is always 8 bytes in a 64bit architecture independent of the type of the pointer:

```
sizeof(int)==8 bytes
sizeof(char)==1 byte
sizeof(int*)==8 bytes
sizeof(char*)==8 bytes
```

1.7 Using Pointers to Optimize Execution

Assume the following function that adds the sum of integers in an array using array indexing.

```
int sum(int * array, int n)
{
      int s=0;
      for(int i=0; i<n; i++)
      {
            s+=array[i]; // Equivalent to
                  //*(int*)((char*)array+i*sizeof(int))
      }
      return s;
}
```

Now the equivalent code using pointers

```
int sum(int* array, int n)
{
      int s=0;
      int *p=&array[0];
      int *pend=&array[n];
      while (p < pend)
      {
            s+=*p;
            p++;
      }
```

```
        return s;
    }
```

When you increment a pointer to integer it will be incremented by 8 units because sizeof(int)==8. Using pointers is more efficient because no indexing is required and indexing require multiplication. An optimizer may substitute the multiplication by a "<<" operator if the size is a power of two. However, the size of the array entries may not be a power of 2 and integer multiplication may be needed.

1.8 Array Operator Equivalence

When C was designed by Kernighan, Ritchie, and Thompson, they wanted to create a high level language that would not sacrifice the same optimization opportunities that you have by programming in an assembly language. They had the great idea to give arrays and pointers the same equivalence and to be able to use arrays or pointers interchangeably.

We have the following equivalences in C between arrays and pointers:

Assume an array such as

```
int a[20];
```

Then we have:

```
a[i]       - is equivalent to
*(a+i)     - is equivalent to
*(&a[0]+i) – is equivalent to
*((int*)((char*)&a[0]+i*sizeof(int)))
```

This means that you may substitute any array indexing such as ***a[i]*** in your program by ****((int*)((char*)&a[0]+i*sizeof(int)))*** and it will work! C was designed to be a machine independent assembler.

1.9 2D Array Representation and Jagged Arrays

We can represent 2D arrays in C in at least these three different ways. Surprisingly, the array operator in C can be used in any of these three ways without change.

- ❏ Consecutive Rows in Memory
- ❏ Array of pointers to Rows
- ❏ Pointer to an Array of Pointers to Rows

2D Array as Consecutive Rows in Memory

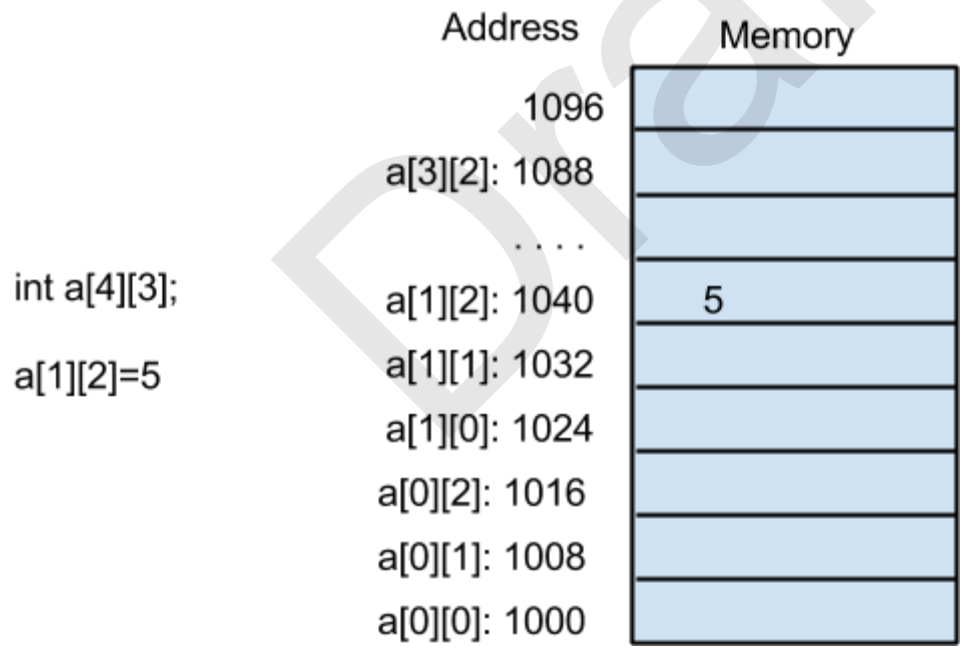
Assume the normal array

```
int a[4][3];
```

You will see that to access an element `a[i][j]` in this array C will generate the following pointer operations:

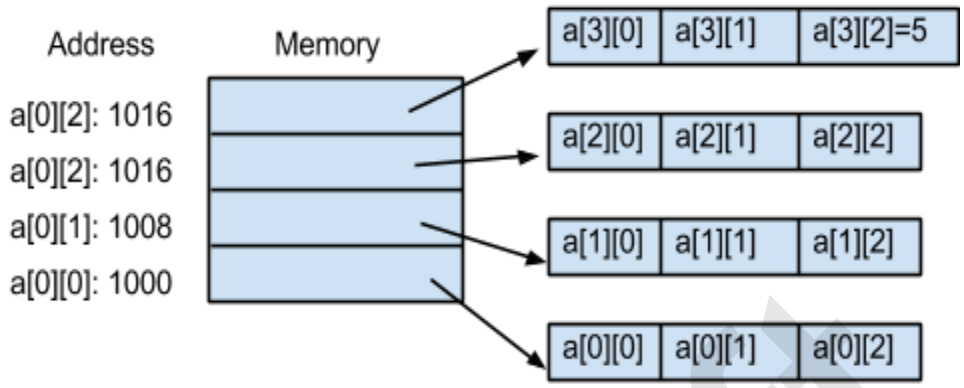
```
a[i][j] == *(int*)((char*)a + i*3*sizeof(int) + j*sizeof(int))
```

The matrix a will be represented in memory in the following way:



2D Array as an Array of Pointers to Rows (Fixed Rows Jagged Array)

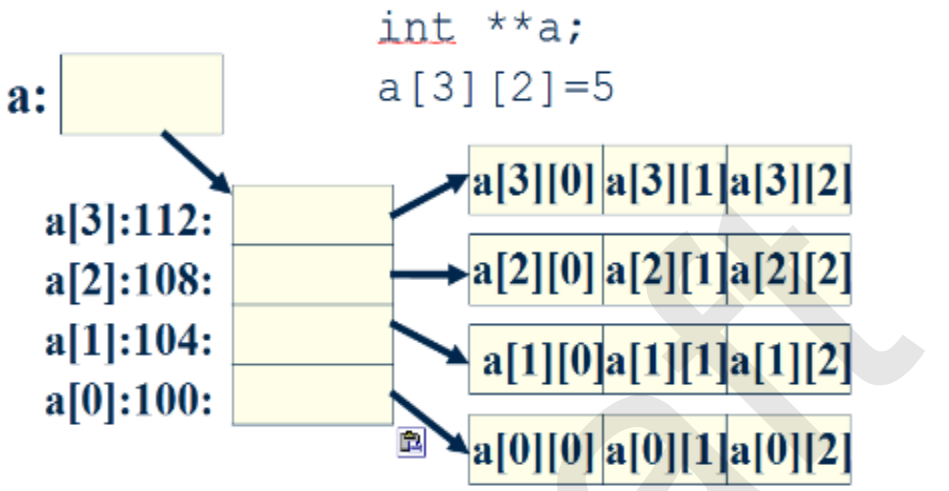
You may also represent a 2D array as an array of pointer to rows. The array may be declared as a global or local variable and the number of rows is defined at compilation time.




```
// Implementation of Jagged Array
int*(a[4]);
for(int i=0; i<4; i++){
   a[i]=(int*)malloc(sizeof(int)*3);
   assert(a[i]!=NULL);
}
a[3][2] = 5;
```

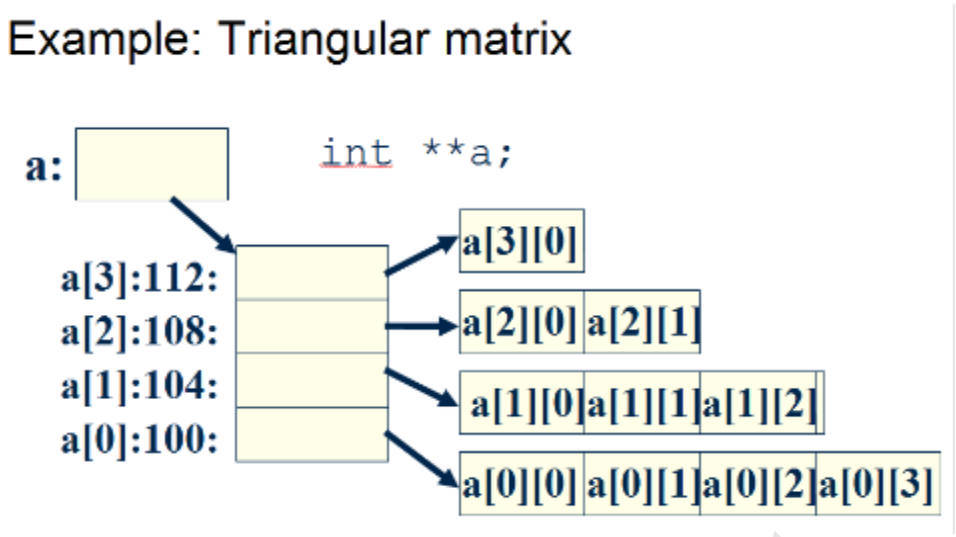
2D Array as a pointer to an Array of Pointers to Rows (Open Jagged Array)

In the previous approach the number of rows has to be set at compilation time. You may also represent a 2D array as an array of pointer to rows. The pointer to the array of pointer to rows is defined as a double pointer. In this way the number of rows and columns can be prespecified at run time.




```
int **a;
a=(int**)malloc(4*sizeof(int*));
assert( a!= NULL)
for(int i=0; i<4; i++)
{
        a[i]=(int*)malloc(3*sizeof(int));
        assert(a[i] != NULL
}
```

The main advantage of a jagged array is that it is the memory allocator does not need to allocate a single large chunk of memory that often forces more memory usage, but instead smaller blocks may be used. In addition, you can define an array with different row sizes like the following triangular matrix:



Pointers to Functions

Pointers to functions have many uses in C programming. A pointer to a function is a variable that contains the address of a function in memory. A pointer to a function allows executing a function through a pointer.

There are many uses for pointer to functions:

1. You can write functions that take a pointer to a function as a parameter to be able to customize the same algorithm without modifying the implementation. For example, you could write a sorting function that takes a pointer to a comparison function that allows sorting an array in ascending, or descending order, or to be able to use custom comparison functions without having to modify the code that sorts the array.
2. You could write a function that iterates over the elements of a data structure such as a tree or hash table and applies a function passed as parameter to every element in the data structure.
3. You can implement object orientation and subclassing in "C" without having to use C++. For instance, UNIX is written in C and it has the concept of a FILE. A FILE has a table called the "vnode" that has pointers to functions to a read(), write(), open(), close() functions that can be used for that FILE. A FILE in disk will have different read/write functions than a FILE that represents a console. It is interesting to see that the implementation of FILEs came before C++ or the invention of Object Oriented Languages. In fact C++ implements inheritance by using a table of pointer to functions called the V-Table that points to the virtual functions of the class that may be overwritten by a subclass.

Here is an example of a function pointer:

```
typedef void (*FuncPtr)(int a);
```

FuncPtr is a type of a pointer to a function that takes an “int” as an argument and returns “void”.

Implementing an Array Mapper

Here is an example of an mapper for an array of integers. The function ***intArrayMapper***() takes as argument an ***array*** of type int, ***n*** the size of the array, and a function ***func*** that is applied to every member of the array.

```
typedef void (*FuncPtr)(int a);

void intArrayMapper( int *array, int n, FuncPtr func ) {
 for( int = 0; i < n; i++ ) {
 (*func)( array[ i ] );
 }
}

int s = 0;
void sumInt( int val ){
 s += val;
}

void printInt( int val ) {
 printf("val = %d \n", val);
}

int a[ ] = {3,4,7,8};
main( ){
  // Print the values in the array
      intArrayMapper(a, sizeof(a)/sizeof(int), printInt);

  // Print the sum of the elements in the array
  s = 0;
  intArrayMapper(a, sizeof(a)/sizeof(int), sumInt);
  printf(“total=%d\”, s);
}
```

This int array mapper has the disadvantage that it can only be used by int arrays. We could rewrite this array mapper to be used on any type of array.

```
typedef void (*GenFuncPtr)(void * a);
void genericArrayMapper( void *array,
      int n, int entrySize, GenFuncPtr fun )
```

```
{
      for( int i = 0; i < n; i++; ){
          void *entry = (void*)(
                 (char*)array + i*entrySize );
                 (*fun)(entry);
      }
}

void sumIntGen( void *pVal ){
  //pVal is pointing to an int
  //Get the int val
  int *pInt = (int*)pVal;
  s += *pInt;
}

void printIntGen( void *pVal ){
  int *pInt = (int*)pVal;
  printf("Val = %d \n", *pInt);
}

int a[ ] = {3,4,7,8};
main( ) {
  // Print integer values
  s = 0;
  genericArrayMapper( a, sizeof(a)/sizeof(int),
             sizeof(int), printIntGen);

  // Compute sum the integer values
  genericArrayMapper( a, sizeof(a)/sizeof(int),
                sizeof(int), sumIntGen);
  printf("s=%d\n", s);
}
```


You may use the same approach for example to write a generic mysort function that can be used to sort an array regardless of the type of the array.

```
typedef int  (*ComparisonFunction)(void * a, void * b);
void mysort(void * array, int nentries, int entrySize,
                ComparisonFunction compFunc);
```

This function takes a pointer to the array, the number of entries, and the size of each entry to be able to compute where each entry in the array starts, and a pointer to a comparison function.

When swapping two entries of the array, you will have pointers to the elements void *a, *b and the size of the entry entrySize.

```
// Allocate temporary memory area dynamically
// used for swapping elements. Do it only once at
// the beginning of the sorting function
 void * tmp = (void *) malloc(entrySize);
assert(tmp != NULL);
  . . .

// Inside sorting loop
void * a = (void*) ((char*)array + i * entrySize);
void * b = (void*) ((char*)array + (i+1) * entrySize);
memcpy(tmp, a, entrySize);
memcpy(a,b , entrySize);
memcpy(b,tmp , entrySize);
…

// Outside sorting loop
free(tmp)
```

Note: You may allocate memory only once for tmp in the sort method and use it for all the sorting to save multiple calls to malloc. Free tmp at the end.

If you are sorting strings, the comparison function passed to mysort will be receiving a “pointer to char*” or a ”char**” as argument.

```
int StrComFun( void *pa, void *pb) {
      char** stra  =  (char**)pa;
      char ** strb  = (char**)pb;
      return strcmp( *stra, *strb);
}
```

Memory Allocation Errors

Explicit Memory Allocation (calling free) uses less memory and is faster than Implicit Memory Allocation (Garbage Collection) .However, Explicit Memory Allocation is Error Prone. Here is a list of the errors that a programmer may make when using malloc()/free() incorrectly.

1. Memory Leaks
2. Premature Free

3. Double Free
4. Wild Frees
5. Memory Smashing

Memory Leaks

Memory leaks are objects in memory that are no longer in use by the program but that are not freed. This causes the application to use excessive amounts of heap memory until it runs out of physical memory and the application starts to swap slowing down the system.

If the problem continues, the system may run out of swap space. Often server programs (24/7) need to be “rebounced” (shutdown and restarted) because they become so slow due to memory leaks.

Memory leaks are a problem for long lived applications (24/7). Short lived applications may suffer memory leaks but is often not a problem since memory is freed when the program goes away. Memory leaks are a “slow but persistent disease”. There are other more serious problems in memory allocation like premature frees.

Example of an (extreme) Memory Leak:

```
int * i;
while (1) {
  ptr = new int;
}
```

Premature Frees

A premature free is caused when an object that is still in use by the program is freed.
The freed object is added to the free list modifying the next/previous pointer. If the object is modified, the next and previous pointers may be overwritten, causing further calls to malloc/free to crash. Premature frees are difficult to debug because the crash may happen far away from the source of the error.

Example of a Premature Free:

```
int * p = new int;
* p = 8;
delete p; // delete adds object to free list updating header info
…
```

```
*p = 9;      // next ptr in the malloc free list may be modified.
int *q = new int;   // this call or other future malloc/free
                    // calls will crash because the free
                    // list is corrupted.
```

```
To reduce the occurrence of Premature Frees
it is a good practice to assign NULL (p=NULL)
after free.

In this way your program will get a SEGV
signal if the object is used after delete
so you can debug it.

int * p = new int;
* p = 8;
delete p; // delete adds object to free list updating header info
p = NULL; // Set Pointer to NULL so it cannot be used again.
…
*p = 9; // This causes <SEGV> trying to write to a NULL pointer
        // You may determine with a debugger where the
        // error happened.
```


Double Free

Double free is caused by freeing an object that is already free. This can cause the object to be added to the free list twice corrupting the free list. After a double free, future calls to malloc/free may crash. Here is an example of a double free:

```
Example of a Double Free

int * p = new int;

delete p; // delete adds object to free list
delete p; // deleting the object again
          // overwrites the next/prev ptr
          // corrupting the free list
          // future calls to free/malloc
          // will crash
```

Wild Frees

Wild frees happen when a program attempts to free a pointer in memory that was not returned by malloc. Since the memory was not returned by malloc, it does not have a header. When attempting to free this non-heap object, the free may crash. Also if it succeeds, the free list will be corrupted so future malloc/free calls may crash. In addition, memory allocated with malloc() should only be deallocated with free() and memory allocated with new should only be deallocated with delete. Wild frees are also called “free of non-heap objects”.

```
Example of a Wild Free

int q;
int * p = &q;

delete p;
 // p points to an object without
 // header. Free will crash or
 // it will corrupt the free list.
```

```
Another Example of a Wild Free

char * p = new char[100];
p=p+10;

delete [] p;
 // p points to an object without
```

```
// header. Free will crash or
// it will corrupt the free list.
```

Memory Smashing

Memory Smashing happens when less memory is allocated than the amount of memory that will be used. This causes the header of the object that immediately follows to be overwritten, corrupting the free list. Subsequent calls to malloc/free may crash. Sometimes the smashing happens in the unused portion of the object causing no damage.

```
  char * s = new char[8];
  strcpy(s, "hello world");

// We are allocating less memory for
// the string than the memory being
// used. Strcpy will overwrite the
// header and maybe next/prev of the
// object that comes after s causing
// future calls to malloc/free to crash.
// Special care should be taken to also
// allocate space for the null character
// at the end of strings.
```


Debugging Memory Allocation Errors

Memory allocation errors are difficult to debug since the effect may happen farther away from the cause. Memory leaks are the least important of the problems since the effects take longer to show. As a first step to debug premature free, double frees, wild frees, you may comment out free calls and see if the problem goes away. If the problem goes away, you may uncomment the free calls one by one until the bug shows up again and you will find the offending free.

There are tools that help you detect memory allocation errors. Here is a list of a few of them:

- Valgrind (Open Source: http://valgrind.org/)
- IBM Rational Purify
- Bounds Checker
- Insure++

Using gdb

While the focus of this book is not on learning fundamental programming topics like debugging, we will briefly review using gdb and some simple debugging paradigms you may or may not have heard before. This is in the hopes that if you are new to programming or it has been a while you will have a brief starting point from which to begin. Note that while we are focusing on the GNU debugger, the common commands we will cover are similar to and found in almost all other debuggers including IDEs like the Microsoft Visual Studio built-in debugger.

The GNU debugger is available on all platforms, but is most readily available on linux distributions from the command line. You will use gdb often, or a similar tool to debug your programs when you come upon troublesome issues you can't easily resolve. You can use gdb to step slowly through your program in order to look at it a line or a few at a time to specify where and what your problem is while it is running.

Common Commands

run / r

This command runs your program inside the debugger, often this will print information relevant to where the crashed occured if you have not set a breakpoint since the program will simply run until it finishes or crashes. If it crashes you will find te next command quite useful.

list / l

This will list the lines of code near where the program crashed. This will often give you a good idea of where to start looking for errors but it is important to note that this will certainly not always show you the lines where the root of the problem lies, and more often than not it will not show you the line where the problem actually lies.

where

This is equivalent to printing a backtrace from within the program, allowing you to see the call stack that led to the program crash. This is helpful by allowing you to see which functions were called in what order to lead to the crash.

up

This command simply moves you one step back up the call stack

p / print VARIABLENAME

This command has two main uses. You can use it to print out the whatever the current value of a variable is at the time of execution, or you can use it to change the value of a variable that is currently in scope while the program is running. This can be useful in some cases to determine if a variable is accidentally uninitialized, set to the incorrect value, etc.

b / breakpoint LINE# or FUNCTION_NAME

Setting a breakpoint will allow you to force a program to “take a break” when it reaches that point of execution. This allows you to watch your program step by step as it continues from that point by using step or next. This way you can determine exactly at which point things start to go awry during your program.

n / next

The next command will run a single line after the previous line allowing you to step through your program without stepping inside of each function, it will just keep executing calls from the current function.

s / step

Step works similarly to next, except instead of stepping over each function call you reach, it allows you to step inside the function. This lets you see every command executed instead of treating function calls you reach as “black boxes” that simply take their input produce their output and move on.

Debugging Example

In the below debugging example, we are running a sample program which has a memory allocation error. One of the arrays is not initialized and a value is expected to be placed into it. We fix this by setting a breakpoint at where the problem is, discovering the problem, and then rerunning the program and using print to initialize the array while still in gdb to check the solution.

```
bash-2.05$ gdb debug
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (sparc-sun-solaris2.5), Copyright 1996 Free Software Foundation, Inc...
(gdb) break main
Breakpoint 1 at 0x10bb8: file public.c, line 19.
(gdb) run
Starting program: /home/champion/e/cs252/test-grr/lab1-src/debug
warning: Unable to find dynamic linker breakpoint function.
warning: GDB will be unable to debug shared library initializers
warning: and track explicitly loaded dynamic code.

Breakpoint 1, main (argc=1, argv=0xffbffae4) at public.c:19
19        printf ("Starting tests.\n");
(gdb) n
Starting tests.
20        fflush (stdout);
(gdb) n
22        initialize_array ();
37          int *numbers = NULL, i = 0;
(gdb) n
```

```
39          for (i = 0; i < 4; i++) {
(gdb) n
40          numbers[i] = i + 1;
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
0x10c3c in initialize_array () at public.c:40
40          numbers[i] = i + 1;
(gdb) print numbers
$1 = (int *) 0x0  ← Here we see that the array is uninitalized.
(gdb) run
Breakpoint 1, main (argc=1, argv=0xffbffae4) at public.c:19
19        printf ("Starting tests.\n");
(gdb) n
Starting tests.
20        fflush (stdout);
(gdb) n
22        initialize_array ();
(gdb) s
initialize_array () at public.c:39
37          int *numbers = NULL, i = 0;
(gdb) n
39          for (i = 0; i < 4; i++) {
(gdb) print numbers = (int*) malloc(sizeof(int) * 5)
$4 = (int *) 0x603010
(gdb) n
40            numbers[i] = i + 1;  ← Program doesn't crash here, so we seem to be good to go.
(gdb) n
```


General Debugging Strategies and Tips

- Write short unit tests after finishing a small section of code
- When a problem is found, start from the last module you didn't unit tests
- Keep tested modules self-contained to prevent having to retest code or risk introducing new bugs to “correct” code.
- Explain the problem to a "rubber duck". Explain to a friend, your monitor, the cat, anything nearby to make sure your logic sounds reasonable out loud.
- Don't think "that's impossible", because it obviously just happened. Don't assume anything when debugging, that section of code that "can't be wrong" is more likely to be the problem than not.

Chapter Summary Questions

1. In your own words, attempt to explain what a pointer really is, and what it is used for. If you struggle to come up with a clean explanation that you wouldn't want to show to someone, try to reread some select sections of this chapter.
2. What is a memory allocation error? What are the different types of memory allocation errors?
3. What is gdb? Why would you use it?