Lecture 08
Dynamic Memory Allocation

In this lecture

- **Dynamic allocation of memory**
  malloc, calloc and realloc
- **Memory Leaks and Valgrind**
- **Heap variables versus stack variables**
- **Revisiting * and ****
- **Memcpy and memmove**
- **Case for Dynamic Variables**
- **Examples**
- **Further Readings**
- **Exercises**

Dynamic memory allocation is necessary to manage available memory. For example, during compile time, we may not know the exact memory needs to run the program. So for the most part, memory allocation decisions are made during the run time. C also does not have automatic garbage collection like Java does. Therefore a C programmer must manage all dynamic memory used during the program execution. The <stdlib.h> provides four functions that can be used to manage dynamic memory.

```
NAME
       calloc, malloc, free, realloc - Allocate and free
       dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *calloc(size_t nmemb, size_t size);
       void *malloc(size_t size);
       void free(void *ptr);
       void *realloc(void *ptr, size_t size);

DESCRIPTION
       calloc() allocates memory for an array of nmemb elements
       of  size  bytes  each and returns a pointer to the allo-
       cated memory.  The memory is set to zero.

       malloc() allocates size bytes and returns a  pointer  to
       the allocated memory.  The memory is not cleared.

       free()  frees  the memory space pointed to by ptr, which
       must have been returned by a previous call to  malloc(),
       calloc()  or  realloc().  Otherwise, or if free(ptr) has
       already been called before, undefined behaviour  occurs.
       If ptr is NULL, no operation is performed.

       realloc()  changes  the size of the memory block pointed
       to by ptr to size bytes.  The contents will be unchanged
       to the minimum of the old and new sizes; newly allocated
```

```
memory will be uninitialized.  If ptr is NULL, the  call
is equivalent to malloc(size); if size is equal to zero,
the call is equivalent  to  free(ptr).  Unless  ptr  is
NULL,  it  must have been returned by an earlier call to
malloc(), calloc() or realloc().
```

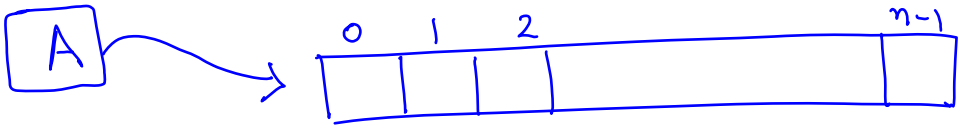
The malloc function

The malloc function allocates a memory block of size n bytes (size_t is equivalent to an unsigned integer) The malloc function returns a pointer (void*) to the block of memory. That void* pointer can be used for any pointer type. malloc allocates a *contiguous block* of memory. If enough contiguous memory is not available, then malloc returns NULL. Therefore always check to make sure memory allocation was successful by using

```
void* p;
if ((p=malloc(n)) == NULL)
    return 1;
else
    { /* memory is allocated */}
```

Example: if we need an array of n ints, then we can do

```
int* A = malloc(n*sizeof(int));
```



A holds the address of the first element of this block of 4n bytes, and A can be used as an array. For example,

```
if (A != NULL)
 for (i=0;i<n;i++)
     A[i] = 0;
```

will initialize all elements in the array to 0. We note that A[i] is the content at address (A+i). Therefore we can also write

```
for (i=0;i<n;i++)
   *(A+i) = 0;
```

Recall that A points to the first byte in the block and A+i points to the address of the ith element in the list. That is &A[i].
We can also see the operator [] is equivalent to doing pointer arithmetic to obtain the content of the address.

A dynamically allocated memory can be freed using free function. For example

free(A);

will cause the program to give back the block to the heap (or free memory). The argument to free is any address that was returned by a prior call to malloc. If free is applied to a location that has been freed before, a **double free memory error** may occur. We note that malloc returns a block of void* and therefore can be assigned to any type.

```
double* A = (double*)malloc(n);
int* B    = (int*)malloc(n);
char* C   = (char*)malloc(n);
```

In each case however, the addresses A+i, B+i, C+i are calculated differently.

- A + i is calculated by adding 8i bytes to the address of A (assuming sizeof(double) = 8)
- B + i is calculated by adding 4i bytes to the address of B
- C + i is calculated by adding i bytes to the address of C

calloc and realloc

calloc and realloc are two functions that can be useful in dynamic memory management

void *calloc(size_t nmemb, size_t size);

allocates memory for an array of **nmemb** elements each of **size** and returns a pointer to the allocated memory. Unlike malloc the memory is automatically set to zero.

calloc(n, sizeof(int))

is equivalent to

malloc(n*sizeof(int))

except for the fact that calloc block is already initialized. Calloc is appropriate when allocating a dynamic array of ints.

Another useful function is realloc. Typically in order to resize an existing memory block, one must reallocate a new block, copy the old values to the new block and then free the old memory block.

void *realloc(void *ptr, size_t size);

realloc() changes the size of the memory block pointed to by ptr to **size** bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if size is equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().

Exercise:
Write realloc function using malloc and free

Memory Leaks

Memory leaks refer to memory that has been allocated by an application, but not properly released back once that memory is no longer needed. Many systems have multiple applications running on their systems and programming errors usually result in “memory leaks”. Memory leaks may not affect the functionality of the application, but left unattended in a system, memory leaks can cause the machine to crash. This is why, servers restart often to avoid them to from going down. Programmers typically allocate memory and then somehow may lose the reference to that memory block. For example, consider the following code.

```
int A[n],i=0;
for (i=0;i<n;i++)
   A[i] = random();

int* B = malloc(2*n);
B = A;
```

The above code initializes a static array A to n random numbers and then requests a memory block (with reference B) that is twice the size of the array A. Then A is assigned to B (note that B = A is a legal assignment. But A = B; is illegal. Why?)

This causes the program to lose a reference to the dynamic block of memory and hence that becomes garbage. We call this a “memory leak”. Therefore once you allocate memory and obtain a reference, DO NOT modify the original reference. You can always define other pointers and copy the address but the original pointer is necessary to free the memory.

Quiz: Consider the following code.

```
int* A = malloc(4*n);
int *B = A;
free(B);
```

does this free the original memory?

It is a good idea to assign NULL to a pointer that has been freed. Otherwise the pointer still contains the original address and a programmer could accidentally assign values to the block that has been freed.

Detecting Memory leaks

There are tools that detects and reports memory leaks. The most widely used tool is called “valgrind”. The Valgrind home page is at http://www.valgrind.org and you can find many resources on Valgrind there. To learn more about valgrind on unix, type

% man valgrind

```
NAME
       valgrind - a suite of tools for debugging and profiling programs

SYNOPSIS
       valgrind [valgrind options] your-program [your-program options]

DESCRIPTION
       valgrind  is  a flexible program for debugging and profiling Linux exe-
       cutables. It consists of a core, which  provides  a  synthetic  CPU  in
       software, and a series of "tools", each of which is a debugging or pro-
       filing tool. The architecture is modular, so that new tools can be cre-
       ated easily and without disturbing the existing structure.

       This  manual  page covers only basic usage and options.  Please see the
       HTML documentation for more comprehensive information.

INVOCATION
       valgrind is typically invoked as follows:

           valgrind program args

       This runs program (with arguments args) under valgrind using  the  mem-
       check  tool.   memcheck  performs a range of memory-checking functions,
       including detecting accesses to uninitialized memory, misuse  of  allo-
       cated memory (double frees, access after free, etc.) and detecting mem-
       ory leaks.

       To use a different tool, use the --tool option:

           valgrind --tool=toolname program args

and more…….
```

To use Valgrind on Andrew unix, compile your code under

% gcc -g -ansi -pedantic -W -Wall main.c

Then run the code with Valgrind as

% valgrind --tool=memcheck --leak-check=full ./a.out

In addition to memcheck, valgrind has many other tools to check the use of functions, cache events etc. For now, we are only interested in making sure our programs don’t leak memory. The report provided by valgrind after running your code may look like

```
==18768== Memcheck, a memory error detector.
```

```
==18768== Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et al.
==18768== Using LibVEX rev 1471, a library for dynamic binary translation.
==18768== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==18768== Using valgrind-3.1.0, a dynamic binary instrumentation framework.
==18768== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==18768== For more details, rerun with: -v
==18768==
==18768==
==18768== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 1)
==18768== malloc/free: in use at exit: 30 bytes in 1 blocks.
==18768== malloc/free: 1 allocs, 0 frees, 30 bytes allocated.
==18768== For counts of detected errors, rerun with: -v
==18768== searching for pointers to 1 not-freed blocks.
==18768== checked 63,760 bytes.
==18768==
==18768== 30 bytes in 1 blocks are definitely lost in loss record 1 of 1
==18768==    at 0x4905599: malloc (vg_replace_malloc.c:149)
==18768==    by 0x400565: main (valgrind1.c:12)
==18768==
==18768== LEAK SUMMARY:
==18768==    definitely lost: 30 bytes in 1 blocks.
==18768==      possibly lost: 0 bytes in 0 blocks.
==18768==    still reachable: 0 bytes in 0 blocks.
==18768==         suppressed: 0 bytes in 0 blocks.
```

For now, we are interested in bytes that are definitely lost. Possibly lost bytes may be funny things that you may do with pointers such pointing to the middle of a heaped block etc. In all programs you write, you should look for memory leaks to make sure program is

Dynamic Variables vs Automatic Variables

Each application may use stack (low memory) and heap (high memory) for managing static and dynamic variables it assigns. A variable defined as

int x;

is given memory (4 bytes) from the stack space and is called an automatic variable. This memory is managed the compiler and remains in the stack as long as variable is within scope.

We call variables that are assigned space through malloc or calloc dynamic variables. For example, we may assign

int* A = malloc(100);

and A is considered a dynamic variable. The dynamic variables are provided space in the "heap". Heap is an area of processes virtual memory managed by an allocator. In most unix systems heap grows upwards (towards higher memory addresses). Unix kernel maintains a variable (called brk) that points to the top of the heap. An allocator maintains heap as a collection of variable size blocks. Each block is contiguous and either allocated or free.

We will discuss more about runtime stack when we study the assembly language later in the course.

Revisiting * and **

We have seen that any generic pointer variable can be defined as

void* ptr or can be specific like int*, char* etc..

The type (int*, char*..) determines how many bytes are dereferenced when the actual pointer is dereferenced. For example, if we define the following

```
char name[30]="guna\0";/* does not copy guna to name*/
int num = 23;
int* intptr = &num;
char* charptr = name;
```

Then *intptr would result in 23 and *charptr will result in ‘g’. Explain why.

intptr in the above example is a “pointer to an integer” or int*. So we can classify intptr as a variable of type int* What about then the address of intptr or &intptr?

We note that intptr is a variable, and hence has an address in the memory. So we are really talking about the address of a pointer to an int (or the address of an address). This is of type int**. To understand this concept, let us look at the following function.

```
void assignint(int** ptr){
     *ptr = malloc(sizeof(int));
}
```

The purpose of the function is to take the address of an int* (that is an int**) and assign enough memory to hold an int. So we can make a call to the function from any calling program as follows.

```
int* ptr;
assignint(&ptr);
*ptr = 10;
```

Case for Dynamic Memory Allocation

Managing memory is an important part of C programming. How much memory is available for static allocation depends on the system. A typical system may have from 1MB–5MB of free memory for stack space. Often we are asked to program devices with limited memory. In these cases C is the language of choice as we have many ways to manage limited memory. Case for dynamic memory is clear. Assume that you wanted to allocate a table of ints of size 100x100 in a device that only has 512K working memory. The table would require a contiguous block of 40K bytes. Now assume that you do not have 40K bytes of free memory in one block available. Your program has already used 392K bytes of memory and only 120K bytes of memory is currently available, but in 6 blocks of size 20K each. Therefore we cannot request a block of size more than 20K memory. Now as a programmer you are to decide how best to use this memory. We need to store 10,000 integers that require 40,000 bytes. Each row with 100 integers would require 400 bytes each.

Therefore each block of 20K bytes can store up to 50 integer rows. Now we can devise a strategy to use a flexible structure to store the integers. That would require allocating an array of 100 int*'s and each location in the array pointing to a block of size 400 bytes. Although we are using 40K bytes (400 ints) + 400 bytes (100 int*'s) all memory can be allocated using available memory.

Memcpy and memmove

Memcpy and memmove are two functions that come in handy in dealing with memory copying.

The prototype of the memcpy function is
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
The memcpy() function copies n bytes from memory area src to memory area dest. The memory areas should not overlap.

RETURN VALUE
The memcpy() function returns a pointer to dest.

Here is an example of copying n values from one array to another.

```
void arraycpy(int A[], int B[], int n) {
    memcpy(A,B,n*sizeof(int));
}
```

What assumptions did we make in this memory copy?

Exercise: Write a function a betterArrayCopy that copies n integers from array B starting at startB to array A, starting at startA.

```
void betterArrayCopy(int A[], int B[], int startA, int
startB, int num) {
```

```
}
```

When using memcpy, the source and destination CANNOT overlap.

When source and destination overlaps, then we can use the function memmove. The prototype of the function memmove is

NAME
memmove – copy memory area

SYNOPSIS
```
#include <string.h>
void *memmove(void *dest, const void *src, size_t n);
```

DESCRIPTION
The memmove() function copies n bytes from memory area src to memory area dest. The memory areas may overlap.

RETURN VALUE
The memmove() function returns a pointer to dest.

Exercise: Write a function that shifts n integers in array A to the right by one starting from ith location.

```
void shift(int A[], int n, int i){




}
```

Example Programs

This code can be found on course democode folders.

Program 8.1: Write a program that allocates dynamic memory required for n integers.

```
#include <stdlib.h>

int* A = NULL;
if ((A = malloc(sizeof(int)*n)) != NULL)
  {
```

```
    for (i=0;i<n;i++)
      *(A+i) = 0; // you can replace *(A+i) by A[i] if you wish
  }
else
  { printf("malloc failed: Exiting Program!\n\n");
    exit( EXIT_FAILURE );
   }
```

Program 8.2: Write a program that allocates dynamic memory required for n strings each with m length.

```
#include <stdlib.h>

char** A = NULL; // char* is a string.
                 // char** is an array of strings

if ((A = malloc(n)) != NULL)
  {
    for (i=0;i<n;i++)
      A[i] = (char*)malloc(m);
  }
else
  { printf("malloc failed: Exiting Program!\n\n");
    exit( EXIT_FAILURE );
   }
```

Program 8.3: Write a function that gets a word from stdin. Assume the max word length is 50. Return the address of the word.

```
#define MAX_WORD_LENGTH 50

char* getword() {
   char* s = malloc(MAX_WORD_LENGTH*sizeof(char));
   if (s == NULL)
   {
      printf("malloc failed: Exiting Program!\n\n");
      exit( EXIT_FAILURE );
    }
   printf("Enter a word (<%d chars): ",MAX_WORD_LENGTH);
   scanf("%s",s);
   return s;
}
```

We can write another version of getword where it takes an address of a string as an argument and allocates memory and reads a string into it.

```
void getWord(char **word)
{
       *word =  malloc(MAX_WORD_LENGTH*sizeof(char));
       if (*word==NULL)
       {
           printf("malloc failed: Exiting Program!\n\n");
            exit( EXIT_FAILURE );
       }

       printf("Enter a word (<%d chars): ",MAX_WORD_LENGTH);
       fflush(stdout);
       scanf( "%s", *word );  /* DANGER: vulnerable to buffer overflow*/
}
```

Further Readings

[1] K & R – Chapter 5 – Pointers and Arrays

Exercises

1. Consider the following code. What is wrong with this code?

   ```
   char* answer;
   printf("Please type something: ");
   gets(answer);
   printf("you typed %s \n",answer);
   ```

2. Why isn't a pointer NULL after calling free?

3. I wanted to allocate space to hold a string s. So I did malloc(strlen(s)). It did not work. Why?

4. I am allocating a large array for some numeric work. So I wrote: **double *array = malloc(256*256*sizeof(double);**
   Malloc is not returning NULL. But the program is acting strangely, as if it's overwriting memory, or malloc isn't allocating as much as I asked for, or something like that. What is wrong with me?

5. Rewrite program_6_2 to allocate a random block of size m (between 1 and 10) that can hold m integers for each A[i].
6. What is the purpose of this program.

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

void intptr(int** ptr){
  *ptr = malloc(4);
}

int main(int argc, char* argv[]){
  int* ptr;
  intptr(&ptr);
  *ptr = 25;
  int num = ptr;

  printf("%d \n", sizeof("gunawar") + 1);
  char* ptr1 = malloc(10);
  char* ptr2 = ptr1;
  strcpy(ptr1,"guna");
  free(ptr1);
  printf("%s\n", ptr2);
  return EXIT_SUCCESS;
}
```

Answers

1. Consider the following code. What is wrong with this code?

```
char* answer;
printf("Please type something: ");
gets(answer);
printf("you typed %s \n",answer);
```

ANSWER: gets assumes that you have already allocated memory to hold a string. But answer is just a pointer variable. No memory was allocated using malloc. Here is a follow up question. Rewrite the code so that it may not segfault.

2. Why isn't a pointer NULL after calling free?

ANSWER: free(ptr) only deallocates the memory pointed to by the ptr. However, the pointer still contains a value that can be misused if the programmer is not careful. Therefore, it is always a good idea to assign ptr = NULL; after freeing the memory.

3. I wanted to allocate space to hold a string s. So I did malloc(strlen(s)). It did not work. Why?

ANSWER: strlen(s) returns the number of characters required to represent the string. For example, "guna" requires 4 characters. However, a string also must end with a '\0' character and therefore it is necessary to allocate one more the strlen(word)

4. I am allocating a large array for some numeric work. So I wrote: **double *array = malloc(256*256*sizeof(double);**
   Malloc is not returning NULL. But the program is acting strangely, as if it's overwriting memory, or malloc isn't allocating as much as I asked for, or something like that. What is wrong with me?

ANSWER: nothing is wrong with you ☺ . here is what happens. When you multiply 256*256*sizeof(double), the multiplication can happen in short int mode. Therefore it is possible that the answer can overflow the max size allocated for an int variable. Hence enough memory may not have been allocated. To fix the problem, make sure you multiply in the unsigned int mode, that has a much larger range of values.

5. Rewrite program_6_2 to allocate a random block of size m (between 1 and 10) that can hold m integers for each A[i].