Dynamic Memory Allocation

The Free Store (Heap)

The Memory is divided into 5 main areas (for now **J**)

- Global name space - Global variables
- Registers - Used for internal housekeeping functions, such as keeping track of the top of the stack and the instruction pointer
- Code space - Where the Code is run from
- The stack - Local variables, along with function parameters.
- The free store - Just about all remaining memory is given to the heap

local variables are valid as long as you are in the defining function, when the function returns, the local variables are thrown away.

Global variables solve that problem at the cost of unrestricted access throughout the program, which leads to a difficult code to understand and maintain.

Putting data in the free store solves both of these problems.

Why Do we need the Heap?

Suppose you want to create a String Array representing someone's last name. The simplest way is to use a character array member to hold the name. But this has some drawbacks. You might use a 14-character array and then run into Bartholomew Smeadsbury-Crafthovingham. Or, to be safer, you may use a 40-character array.

But, if you then create an array of 2000 such Strings, you'll waste a lot of memory with character arrays that are only partly filled. (At this point, we're adding to the computer's memory load.)

There is an alternative. Often it is much better to decide many matters, such as how much storage to use, in Real Time when a program is running

The usual approach is to make a dynamic allocation for the correct amount of memory while the program is running.

The allocation\de-allocation could be quite tricky at times (why?)

The Free Store (Heap)

The stack is cleaned automatically when a function returns. All the local variables go out of scope, and they are removed from the stack.

The free store is not cleaned until your program ends, and it is your responsibility to free any memory that you've reserved when you are done with it.

The advantage to the free store is that the memory you reserve remains available until you explicitly free it.

If you reserve memory on the free store while in a function, the memory is still available when the function returns.

The advantage of accessing memory in this way, rather than using global variables, is that only functions with access to the pointer have access to the data.

This provides a tightly controlled interface to that data, and it eliminates the problem of one function changing that data in unexpected and unanticipated ways.

For this to work, you must be able to create a pointer to an area on the free store and to pass that pointer among functions.

malloc

Allocate memory block on the free store, returning a pointer to the beginning of the block. If malloc cannot create memory on the free store (memory is, after all, a limited resource) it returns the null pointer.

#define <stdlib>

*void * malloc (size_t size);*

size - Size of the memory block, in bytes.

The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.

To create an unsigned short variable on the free store:

*Char *ptr = (char *) malloc(sizeof(char));*

ptr - return value from `malloc` is the memory address and It must be cased to the pointers type.

**ptr = 72;* This means, "Put 72 to the address on the free store to which *ptr* points.“

malloc

malloc is a subroutine for performing dynamic memory allocation ,It is part of the standard library.

malloc is followed by the size of the variable that you want to allocate so that the compiler knows how much memory is required.

```
#include <stdlib.h>
void *malloc(size_t size);
```

The return value from malloc is a memory address and It must be assigned to a pointer.

To create an int variable on the free store, you might write

```
int *pPointer;
pPointer = malloc(sizeof(int));
*pPointer = 72;
```

This means, "Put 72 to the address on the free store to which pPointer points.“
If new cannot create memory on the free store (memory is, after all, a limited resource) it returns the null pointer. You must check your pointer for null.

delete (free)

When you are finished with your area of memory, you must call free on the pointer. free deallocates (returns) the memory to the free store.

```
#include <stdlib.h>
void    free(void *ptr);
```

Remember that the pointer itself is a local variable. When the function in which it is declared returns, that pointer goes out of scope and is lost.

The memory allocated with malloc is not freed automatically, if That memory becomes unavailable a situation called a memory leak occurs.

It's called a memory leak because that memory can't be recovered until the program ends.

To restore the memory to the free store, you use the keyword free system call.

```
free pPointer;
pPointer=NULL; //always zero it
```

** 8.5 allocating variables on the heap

Accessing Data Members

For an instance of an object - You accessed data members and functions by using the dot (`.`) operator

For a ptr to an instance of an object (access the object on the free store),

C++ provides a shorthand operator for indirect access:

the `points-to` operator (`->`),

pCat->GetAge();

You can also use

*(*pCat).GetAge();*

Parentheses are used to assure that `pCat` is dereferenced before `GetAge()` is accessed.

**8.6 accessing member variables on the free store.

Pointer calculations

What happens if we add 1 to a pointer that points to a variable of some type ?

*int *pHeap = NULL;*

*pHeap = (int *)malloc(sizeof(Array));*

**(pHeap+1) // increased by 1 byte?? NO it by 4=sizeof(int)*

The pointer is incremented by the size of the variable type it points to.

malloc \ free – Warnings

Each time you allocate memory you must check to make sure the pointer is not null (there was enough memory in the pool to allocate).

Stray /Dangling pointers :

1. When you delete a pointer, the memory it points to is freed. Calling delete on that pointer again will crash your program! When you delete a pointer, set it to zero (null).
2. Using a pointer after delete without nullizing it can crash the program. Calling delete on a null pointer is guaranteed to be safe.

Memory leaks :

1. For every time in your program that you allocate memory, there should be a call to deallocate it.
2. Allocating memory inside a function should NOT return the pointer out of that function

```
char  *pChar = malloc(sizeof(char));
free pChar ;     //frees the memory
pChar = 0;      //sets pointer to null
```

Memory Leaks

Another way you might inadvertently create a memory leak is by reassigning your pointer before deleting the memory to which it points.

Consider this code fragment:

```
int *ptr = (int *) malloc(sizeof(int));
 *ptr = 72;
 *ptr = (int *) malloc(sizeof(int)); //new address!!
* ptr = 84; //WRONG
```

The code should have been written like this:

```
int *ptr = (int *) malloc(sizeof(int));
 *ptr = 72;
 free ptr;
int *ptr = (int *) malloc(sizeof(int)); // alocate new Memory space from O.S.
 *ptr = 84;
```

Arrays on the Free Store

Nothing new here use malloc, sizeof and **casting** as usual

pHeap = (int *)malloc(sizeof(Array)); //pHeap points to 1st Element

declares pHeap to be a pointer to an array on the heap.

In order to access Array's elements on the heap we increment the pointer the (the Compiler know how to calculate the offset)

```
*pHeap = 10;            // set 1st Element to 10
 pHeap ++;              // advance to 2nd Element
 *(pHeap +2) = 20;      // set 4th Element to 20
```

** 8.6 allocating an array on the heap

Deleting Arrays on the Free Store

In order to delete the array on the heap we use free as usual:

```
free (pHeap );
pHeap = NULL;
```

this returns all the memory set aside for the array back to O.S., and forbids from Stray /Dangling pointers

Dynamic Memory-Sum it up (Heap)

Dynamic memory is useful.

Part of C strategy is letting the program decide about memory during runtime rather than during compile time. That way, memory use can depend on the needs of a program instead of upon a rigid set of storage-class rules. Dynamic memory has several caveats:

- Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()'d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a "memory leak".
- Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to it.
- Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic