Malloc'd objects, garbage collection, and memory leaks

The Java language provides for "automagic" garbage collection in which storage for an object is magically reclaimed when all references to an object have gone out of existence.

C provides no such mechanism.

A *memory leak* is said to have occurred when:

1. the last pointer to a malloc'd object is reset or
2. the last pointer to a malloc'd object is a local variable in a function from which a return is made,

In these cases the *malloc'd* memory is no longer accessible. Excessive leaking can lead to poor performance and, in the extreme, *program failure*.

Therfore C programmers must recognize when last pointer to *malloc'd* storage is about to be lost and use the *free()* function call to release the storage before it becomes impossible to do so.

Several examples of incorrect pointer use and memory leaking have been observed in student programs

Problem 1: The instant leak.

This is an example of an *instant leak*. The memory is allocated at the time *temp* is declared and leaked when *temp* is reassigned the address of the first object in the list.

```
obj_t *temp = malloc(sizeof(obj_t));

temp = list->head;
while (temp != NULL)
  -- process list of objects --
```

Two possible solutions:

Insert *free(temp)* before *temp = list->head;*

This eliminates the leak, but what benefit is there to allocating storage and instantly freeing it???

Change the declaration to *obj_t *temp = NULL;*

This is the correct solution.

A rational rule of thumb is *never malloc memory unless you are going to write into it!*

Another good rule of thumb is to *never declare a pointer without initializing it.*

Problem 2: The traditional leak

Here storage is also allocated for *univec* at the time it is declared.

The call to *vl_unitvec3()* writes into that storage.

If the storage is not *malloc'd*, then *univec* will not point to anything useful and the call to *vl_unitvec3()* will produce a segfault or will overwrite some other part of the program's data. So this *malloc()* is necessary.

```
{
    double *univec = malloc(3 * sizeof(double));

    vl_unitvec3(dir, univec);
           :
    more stuff involving univec
           :
    return;
 }
```

However, the instant the return statement is executed, the value of *univec* becomes no longer accessible and the memory has been leaked.

Here the correct solution is to add

```
  free(univec);
```

just before the return;

A rational rule of thumb is: *malloc'd storage must be freed before the last pointer to it is lost.*

Problem 3: Overcompensation

The concern about leakage might lead to an overcompensation. For example, an object loader might do the following:

```
{
      obj_t *new_obj;
            :
      new_obj = malloc(sizeof(obj_t));
           :
      if (list->head == NULL)
      {
           list->head = list->tail = new_obj;
      }
      else
      {
           list->tail->next = new_obj;
           list->tail = new_obj;
      }
      free(new_obj);

      return(0);
}
```

This problem is the reverse of a memory leak. A live pointer to the object exists through the *list* structure, but the storage has been *freed.*

The results of this are:

1. Usually attempts to reference the freed storage will succeed.
2. The storage will eventually be assigned to another object in a later call to *()*.
3. Then "both" objects will occupy the same storage.

Rational rule of thumb: *Never free an object while live pointers to the object exist.* Any pointers to the freed storage that exist after the return from *free()* should be set to NULL.

To fix this problem the *free(new_obj) must be deleted from the code*. If the objects in the object list are to be freed, *it is safe to do so only at the end of the raytrace.*

It is not imperative to do so at that point because the Operating System will *reclaim all memory*used by the process when the program exits.

Problem 3b: Overcompensation revisited

The *free()* function must be used *only* to free memory previously allocated by *malloc()*

```
unsigned char buf[256];
:
:
free(buf);
```

is a fatal error.

The *free()* function must be not be used to free the same area twice.

```
buf = (unsigned char *)malloc(256);

:
free(buf);

:
free(buf);
```

is also fatal**.**

The general solution: Reference counting

For programs even as complicated as the raytracer it is usually easy for an experienced programmer to know when to *free()* dynamically allocated storage.

In programs as complicated as the Linux kernel it is not.

A technique known as *reference counting* is used.

```
typedef struct obj_type
{
   int refcount;
   :
  :
} obj_t;
```

At object creation time:

```
new_obj = malloc(sizeof(obj_t));
new_obj->refcount = 1;
```

When a new reference to the object is created

```
my_new_ptr = new_obj;
my_new_ptr->refcount += 1;
```

When a reference is about to be reused or lost

```
my_new_ptr->refcount -= 1;
if (my_new_ptr->refcount == 0)
   free(my_new_ptr);
my_new_ptr = NULL;
```

In a multithreaded environment such as in an OS kernel it is *mandatory* that the testing and update of the reference counter be done *atomically*. This issue will be addressed in CPSC 322.