Dynamic Memory Deallocation

It's generally considered good programming practice to *deallocate* any dynamically allocated memory that you're done with.

Deallocating isn't especially crucial in programs that only use a small amount of memory, but if you've got a program that uses many GB and does a lot of dynamic memory allocation, you really do want to deallocate any memory that you're done with, because that'll free the memory up for other things.

In C, dynamic memory is deallocated using the `free` function:

```
...
dynamic_element =
  (float *)malloc(sizeof(float) * numelts);
...
free(dynamic_element);
...
```

What does `free` do?

C maintains a list of *memory blocks* that are available for dynamic allocation.

When you call `malloc`, you send it a length in bytes, and it grabs a block of the appropriate size and returns the address of the block's first byte (i.e., a pointer to the block).

Likewise, when you call `free`, you send it a pointer to a block, and it puts that block back into the list of free blocks. (This is a gross oversimplification.)

An Array Variable Is a Pointer

In C, when we declare an array statically

```
float static_element[100];
```

we are setting up a block in memory, but we're doing it at compile time instead of at runtime.

Otherwise, an array is identical to a pointer. Specifically, it's a pointer to the block of memory that holds the array.

```
% cat arrptr.c
#include <stdio.h>

main () { /* main */
  float array[5] = { 0.0, 1.5, 3.0, 4.5, 6.0 } ;
  float *arrayptr;
  int   i;

  arrayptr = array;
  for (i = 0; i < 5; i++)
    printf("%d: %f %f\n",
      i, array[i], arrayptr[i]);
} /* main */
% cc -o arrptr arrptr.c
% arrptr
0: 0.000000 0.000000
1: 1.500000 1.500000
2: 3.000000 3.000000
3: 4.500000 4.500000
4: 6.000000 6.000000
```

Pointing to an Array Is the Same As Pointing to the Array's First Element

```
% cat arrptrelt.c
#include <stdio.h>

main () { /* main */
  float array[5] = { 0.0, 1.5, 3.0, 4.5, 6.0 } ;
  float *arrayptr;
  float *array0ptr, *arrayptr0ptr;
  int   i;

  arrayptr = array;
  printf("array        = %d\n", array);
  printf("arrayptr     = %d\n", arrayptr);
  array0ptr    = &array[0];
  arrayptr0ptr = &arrayptr[0];
  printf("array0ptr    = %d\n", array0ptr);
  printf("arrayptr0ptr = %d\n", arrayptr0ptr);
  for (i = 0; i < 5; i++)
    printf("%d: %f %f %f %f\n",
      i, array[i], arrayptr[i],
      array0ptr[i], arrayptr0ptr[i]);
} /* main */
% cc -o arrptrelt arrptrelt.c
% arrptrelt
array        = 536869688
arrayptr     = 536869688
array0ptr    = 536869688
arrayptr0ptr = 536869688
0: 0.000000 0.000000 0.000000 0.000000
1: 1.500000 1.500000 1.500000 1.500000
2: 3.000000 3.000000 3.000000 3.000000
3: 4.500000 4.500000 4.500000 4.500000
4: 6.000000 6.000000 6.000000 6.000000
```

Difference in Addresses of Successive Array Elements

As we saw in Fortran 90, successive array elements have successive addresses; in fact, this is the definition of an array. The difference in addresses equal to the size of the base data type.

```
% cat arreltptr.c
#include <stdio.h>

main () { /* main */
  double darray[5] = { 0.0, 1.5, 3.0, 4.5, 6.0 } ;
  float  farray[5] = { 0.0, 1.5, 3.0, 4.5, 6.0 } ;
  int    i;

  printf("sizeof(double) = %d\n", sizeof(double));
  printf("darray =\n");
  for (i = 0; i < 5; i++)
    printf("%d: %f %d\n", i, darray[i], &darray[i]);
  printf("sizeof(float) = %d\n", sizeof(float));
  printf("farray =\n");
  for (i = 0; i < 5; i++)
    printf("%d: %f %d\n", i, farray[i], &farray[i]);
} /* main */
% cc -o arreltptr arreltptr.c
% arreltptr
sizeof(double) = 8
darray =
0: 0.000000 536869672
1: 1.500000 536869680
2: 3.000000 536869688
3: 4.500000 536869696
4: 6.000000 536869704
sizeof(float) = 4
farray =
0: 0.000000 536869648
1: 1.500000 536869652
2: 3.000000 536869656
3: 4.500000 536869660
4: 6.000000 536869664
```

Pointer Arithmetic

In C, a pointer is really just an integer that happens to refer to a memory address.

Since a pointer is just an integer, then we ought to be able to do arithmetic on it. For example, we ought to be able to increment a pointer:

```
% cat ptrinc.c
#include <stdio.h>

main () { /* main */
  char name[20] = "Henry Neeman";
  char *nptr;

  printf("name:  %s\n", name);
  printf("sizeof(char) = %d\n", sizeof(char));
  for (nptr = name; *nptr != '\0'; nptr++)
    printf("nptr = %d, *nptr = %c\n", nptr, *nptr);
  printf("nptr:  ");
  for (nptr = name; *nptr != '\0'; nptr++)
    printf("%c", *nptr);
  printf("\n");
} /* main */
% cc -o ptrinc ptrinc.c
% ptrinc
name:  Henry Neeman
sizeof(char) = 1
nptr = 536869680, *nptr = H
nptr = 536869681, *nptr = e
nptr = 536869682, *nptr = n
nptr = 536869683, *nptr = r
nptr = 536869684, *nptr = y
nptr = 536869685, *nptr =
nptr = 536869686, *nptr = N
nptr = 536869687, *nptr = e
nptr = 536869688, *nptr = e
nptr = 536869689, *nptr = m
nptr = 536869690, *nptr = a
nptr = 536869691, *nptr = n
nptr:  Henry Neeman
```

Pointer Addition

We also ought to be able to add an integer to a pointer:

```
% cat ptradd.c
#include <stdio.h>

main () { /* main */
  int array[5] = { 7, 14, 21, 28, 35 } ;
  int *ap;
  int i;

  ap = array;
  printf(" array = %d,  ap = %d\n",  array,  ap);
  printf("*array = %d, *ap = %d\n", *array, *ap);
  printf("&array[3] = %d, array[3]  = %d\n",
    &array[3], array[3]);
  printf("ap + 3    = %d, *(ap + 3) = %d\n",
    ap + 3, *(ap + 3));
  printf("*ap + 3  = %d\n", *ap + 3);
} /* main */
% cc -o ptradd ptradd.c
% ptradd
 array = 536869680,  ap = 536869680
*array = 7, *ap = 7
&array[3] = 536869692, array[3]  = 28
ap + 3    = 536869692, *(ap + 3) = 28
*ap + 3  = 10
```

Notice that

`ap + 3`

is not 3 plus the address that `ap` points to, but rather the address plus **the size of 3 elements of the base type**; in this case, the address plus 12, since each `int` element has 4 bytes.

So, what we discover is that

`array[i]`

is really just a shorthand notation for

`*(array + i)`

The Null Pointer

C has a special pointer called the *null pointer*, which indicates that the pointer doesn't point to anywhere. It's indicated as `NULL`, which is a macro, defined in `stdio.h`, whose value is zero.

Good programming style requires initializing all pointers to NULL (except statically allocated arrays):

```
% cat ptrnull.c
#include <stdio.h>

main () { /* main */
  char name[20] = "Henry Neeman";
  char *nptr = NULL;

  printf("name:  %s\n", name);
  printf("sizeof(char) = %d\n", sizeof(char));
  printf("nptr = %d\n", nptr);
  if (nptr == NULL)
    printf("No value for nptr yet.\n");
  for (nptr = name; *nptr != '\0'; nptr++)
    printf("%c", *nptr);
  printf("\n");
} /* main */
% cc -o ptrnull ptrnull.c
% ptrnull
name:  Henry Neeman
sizeof(char) = 1
nptr = 0
No value for nptr yet.
Henry Neeman
```

Notice that we can compare a pointer to `NULL` to determine whether the pointer has been assigned a value yet.

So, `NULL` is a *sentinel*: a value that we've selected to mean something special (in this case, the fact that the pointer doesn't point to anything).

Aside: Type Casting

In C, we can convert from one data type to another using *type casting*.

This has nothing to do with actors getting stuck playing the same kind of role over and over.

```
% cat typecast.c
#include <stdio.h>
#include <math.h>

main () { /* main */
  double d = 75.0;
  float  f;
  int    i;
  char   c;
  void   bitprint(void *ptr, int numbytes);

  f = (float)d;
  i = (int)f;
  c = (char)i;
  printf("d = %f, f = %f, i = %d, c = %d = %c\n",
    d, f, i, c, c);
  printf("pow(i,2) = %g\n", pow(i,2));
  printf("pow((double)i,(double)2) = %d\n",
    (int)pow((double)i,(double)2));
} /* main */
% cc -o typecast typecast.c
ld:
Unresolved:
pow
% cc -o typecast typecast.c -lm
% typecast
d = 75.000000, f = 75.000000, i = 75, c = 75 = K
pow(i,2) = 4.26182e+140
pow((double)i,(double)2) = 5625
```

Pointer to `void`

In C, we use the type `void` to indicate a function that doesn't return anything; that is, a `void`-valued function in C is analogous to a subroutine in Fortran 90.

So what would a pointer to type `void` be?

Oddly enough, it's not a pointer to nothing (i.e., to null).

Instead, it's a generic pointer; that is, a pointer that could point to any data type. When we want to use what the pointer points to, we simply cast the pointer.

For example, if we have a `void` pointer and we want to use its *referent* as an `int`, we cast the `void` pointer to an `int` pointer.

Example: Pointer to `void`

```
% cat voidptr.c
#include <stdio.h>

#define LENGTH  7

main () { /* main */
  int  *array = NULL;
  int  i;
  void *mymalloc(int);
  void myfree(void *, int, int);

  array = (int *)mymalloc(sizeof(int) * LENGTH);
  for (i = 0; i < LENGTH; i++)
    array[i] = i * 5;
  printf("array =\n");
  for (i = 0; i < LENGTH; i++)
    printf("  %2d", array[i]);
  printf("\n");
  myfree((void *)array, LENGTH, sizeof(int));
} /* main */

void *mymalloc (int numbytes)
{ /* mymalloc */
  printf("Allocating %d bytes.\n", numbytes);
  return (void *)malloc(numbytes);
} /* mymalloc */

void myfree (void *ptr, int numelts, int eltsize)
{ /* myfree */
  printf("Freeing %d bytes.\n", numelts * eltsize);
  free(ptr);
} /* myfree */
% cc -o voidptr voidptr.c
% voidptr
Allocating 28 bytes.
array =
   0   5  10  15  20  25  30
Freeing 28 bytes.
```