

Computational Physics

Allocating space

Comments and questions to [John Rowe](#).

So far we have used **pointers** to point to existing structures, arrays and, when using `scanf()`, variables. But this requires us to know the sizes of our arrays and the number of structures we will need when we write the program. If we write our word processor with enough structures to represent four open documents, what happens when our user wants to edit five?

For this reason, it is also possible to **dynamically allocate** memory as we need it. This can then be used like any other memory such as that obtained by declaring structures arrays.

Dynamically allocated memory acts like anonymous (nameless) structures and arrays which we can create and destroy at will.

Remember, as far as the computer is concerned a variable, structure or array is just a name for a location in memory.

malloc

The `malloc` function allocates some memory for us to use. Let's first consider allocating memory to be used in the same way as a one-dimensional array:

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

#define N 100
int
main() {
    float *p = NULL;

    p = malloc(N * sizeof *p);
    if (p == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }
    p[0] = p[1] = 3.14159;
    /* Use p for something */
    return 0;
}
```

`p` above is an ordinary pointer, just like any other. As always, trying to use `p` without making sure it points to something would be a disaster. **Once we have allocated some memory for it** we can use a pointer just like an array.

Notice how we have initialised `p` to `NULL` so that if we try to use it then, on most systems, the program will crash immediately.

The sizeof operator

For any pointer `p` the expression `sizeof *p` returns the number of bytes needed for whatever `p` points to. It's the ideal argument to `malloc`! There's also a version `sizeof(float)` which is less useful.

NULL

AS with `fopen`, `NULL` is used to indicate that `malloc` ran out of memory. We must always check that `malloc` did not return `NULL`. We have met the `exit()` function before, does pretty much what it says and by convention, successful programs exit with zero.

You are welcome to check the result of `malloc` every time you call it. You are equally welcome to define a function like:

```
void *xmalloc(size_t n) {
    void *p = malloc(n);
    if (p == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }
}
```

Reminder: `NULL` and `exit()` are declared in `<stdio.h>` and `<stdlib.h>` respectively.

```

return p;
}

```

and to call `xmalloc` instead of `malloc`. I know which I prefer to do!

Pointers to pointers

Whilst conceptually simple, pointers to pointers can cause confusion. If you are a little less confident you may wish to treat this indented section as a bit of a recipe.

Arrays of pointers

These work in the obvious way:

```

#define N 100
#define M 4
int
main() {
    float *p[M];
    int i;

    for (i = 0; i < M; ++i)
        p[i] = xmalloc(N * sizeof *p[i]);

    p[0][1] = 3.14159;
    /* Use p for something */
    return 0;
}

```

Dynamically allocated '2-D arrays'

Of course, often we won't know how many pointers we will need so we can dynamically allocate them too. The following function allocates memory that can then be used like a two dimensional array (but see [note below](#)).

```

int **
new2dint(int m, int n) {
    int **p;
    int i;

    p = xmalloc(m * sizeof *p);
    for (i = 0; i < m; ++i)
        p[i] = xmalloc(n * sizeof *p[i]);
    return p;
}

int
main() {
    int **p = NULL;
    int n;

    do {
        printf("Size of array (>0)?\n");
        scanf("%i", &n);
    } while (n <= 0);

    p = new2dint(n, n);
    p[0][0] = 17;
    p[n-1][0] = 13;
    p[n-1][n-1] = 10;

    return(0);
}

```

Differences from true multi-dimensional arrays

Executive summary: the things above may look like ordinary two-dimensional arrays but they're not

At first sight `p` above looks just like a "real" array, in that their elements are accessed using the same source-level notation:

```
int array[M][N];
int **p = NULL;

p = new2dint(M, N);

p[1][7] = 17;
array[1][7] = 17;
```

However, `p` and `array` are different things. `p` is a pointer to `M` pointers to `N` ints. By contrast, `array` points to a single block of `M*N` ints.

They are dereferenced as follows:

```
p[i][j] => *( *(p+i) + j)
```

Double dereference: dereference `(p+i)`, add `j` and dereference the result.

```
array[i][j] => *(array + (N*i + j))
```

Single dereference: add `N*i+j` to `array` and dereference that.

This is why for functions which have arguments which are multidimensional arrays we must tell the function the size of the array (which the left-most size being optional):

```
int oldstylefun(float array[][N]);
int newstylefun(int n, float array[][n]);
int three_d_fun(int m, int n, float array[][m][n]);
```

Allocating structures

A major use of `malloc()` is to dynamically create **structures**. The mechanics work just as we would expect:

```
struct thingy *p = NULL;
p = xmalloc(sizeof *p);
```

In practice however it's a very good idea to define a function whose job is to allocate the space for a structure and do any initialisation:

```
#include <stdlib.h>

typedef struct wotsit {
    float val;
    char *str;
    int length;
} Wotsit;

Wotsit *newwotsit(int len) {
    Wotsit *w = xmalloc(sizeof *w);

    w->length = len;
    w->val = 0;
    w->str = xmalloc(w->length);
    return w;
}

int main() {
```

Notice how `str` is a pointer not an array. In order to use it we must make it point at something.

```
Wotsit *william = newwotsit(100);
/* Blah, blah, blah */
}
```

In practice structures are nearly always dynamically allocated, seldom statically.

Freeing memory

When we've finished with some memory we should get rid of it so it can be reused. If we fail to do this our program will gradually use more and more memory until the computer runs out. (When our program finishes it will automatically free any memory it was still using.)

The absolute rule is that we can't use memory after we've freed it and we can free it only once. To reuse the above example we might define a function `freewotsit()`:

```
void freewotsit(Wotsit *w) {
    free(w->str);
    free(w);
}
```

The above version is correct. Had we written:

```
free(w);
free(w->str);
```

It would have been very, very wrong.

Lists, etc.

Allocating memory is the easy bit and in reality is just the first of three tasks:

1. **Getting more memory.**
2. **Keeping track of the memory we have obtained.**
3. **Releasing it when we have finished with it.**

We have already covered 1 (`malloc`) and 3 (`free`); this section is about 2.

Keeping track of memory

We can call `malloc` as often as we like to give ourselves more and more structures but there's a problem: once we have lost the value of a pointer to memory allocated by `malloc` there is no way to find it out again so how do we remember where all our structures are?

The following code illustrates the problem: it allocates memory for `n` structures but promptly forgets where the first `n-1` are stored:

```
for (i = 0; i < n; ++i)
    p = malloc(sizeof *p);
```

The first `n-1` structures are thus inaccessible: we have allocated the memory but forgotten where it was. It's like writing down someone's phone number and losing the piece of paper we wrote it on - they have a perfectly good phone but we can't ring it because we've lost the number.

We could try having an array of pointers:

```
typedef struct particle {
    float mass;
    float x;
    float y;
} Particle;

struct particle *p[NMAX] = { NULL };

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

Can you think of a better way for us to have defined the coordinates in this structure?

```
p[i] = malloc(sizeof *p[i]);
```

but of course we are back to the same problem: how big do we make *NMAX*? Clearly we need a method that can handle an arbitrary number of pointers to structures.

[Class exercise.]

Linked lists

One answer to this problem is to make a **list** of pointers. To make a list we need to add a new element to our structure: a pointer to the next one. The pointer is `NULL` if there is no next one, ie this element is the last one in the list:

```
typedef struct particle {
    float mass;
    float x[NDIM];
    struct particle *next;
} Particle;
```

(Notice we have also tidied up how we store the co-ordinates.)

Creating a linked list

In the following code the function `newparticle` allocates and initialises a new structure, and adds it on to the beginning of an existing linked list:

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

typedef struct particle {
    float x[NDIM];
    struct particle *next;
} Particle;

void *xmalloc(size_t n);
Particle *newparticle(Particle *first);

Particle *newparticle(Particle *first) {
    Particle *p;

    p = xmalloc(sizeof *p);
    printf("Please enter the x and y values.\n");
    scanf("%f %f", &p->x[0], &p->x[1]);
    printf("Please enter the mass.\n");
    scanf("%f", &p->mass);
    p->next = first;
    return p;
}

main() {
    int i, n;
    Particle *first = NULL;

    printf("How many structures would you like?\n");
    scanf("%d", &n);

    for (i = 0; i < n; ++i)
        first = newparticle(first);
}
```

Notice:

- We initialised the first element to `NULL` to show that the list started off empty: **this is vital**.
- We put all the checking that `malloc` didn't fail into a utility function called `xmalloc` that we can now use just like `malloc` but without the tedious checking.

Accessing every member of a list

The following `for` loop loops over every element of the list in turn printing out their mass and co-ordinates. The pointer `p` is initially equal to `first` and then becomes `first->next` and so on.

```
for (p = first; p != NULL; p = p->next)
    printf("%f (%f, %f)\n", p->mass, p->x[0], p->x[1]);
```

Removing and freeing a member of a list

This introduces an extra complication: as well as freeing the structure itself we also have to remove it from the list.

To remove the first member of a list we simply set the new first member to `old_first->next` and free `old_first`. For every member except the first we find its predecessor and change its `next` pointer to the `next` of the one we want to free:

Initial list

```
A -> B -> C -> D -> NULL
```

After removing 'B'

```
A -> C -> D -> NULL
```

```
free(B)
```

After removing 'A'

```
C -> D -> NULL
```

```
free(A)
```

A complete example

```
/* The structure definition and newparticle are the same as above */
Particle *freeparticle(Particle *this, Particle *first) {
    Particle *newfirst = first;

    if (this == NULL)
        return newfirst;

    if (this == first)
        newfirst = first->next;
    else { /* Find its predecessor */
        while (first != NULL && first->next != this)
            first = first->next;
        if (first == NULL) {
            fprintf(stderr, "%p isn't in the list!\n", this);
            return newfirst;
        }
        first->next = this->next;
    }

    free(this);
    return newfirst;
}

void printallparticles(Particle *p) {
    for (; p != NULL; p = p->next)
        printf("%f (%f, %f)\n", p->mass, p->x[0], p->x[1]);
    printf("\n");
}

int main() {
    int i, n;
    Particle *first = NULL;

    printf("How many structures would you like?\n");
```

```
scanf("%d", &n);

for (i = 0; i < n; ++i)
    first = newparticle(first);

printallparticles(first);
if (i > 0) {
    first = freeparticle(first->next, first);
    printallparticles(first);
}
first = freeparticle(first, first);
printallparticles(first);

return 0;
}
```

Summary

- Dynamic allocation allows us to change the number of "things" we have as the program runs.
 - But these things have no name so we must have a way of remembering where they are.
 - We must free memory when we are finished and not try to use it after we have freed it.
 - Pointers to pointers can be confusing and are not the same as two-dimensional arrays.
-