PH502: Scientific Programming Concepts

Irish Centre for High End Computing (ICHEC)

September 23, 2020

Overview

- We have a final look at using pointers and dynamic arrays.
- There are some issues of which to be aware.
- We will show some of these.

Pointer and Arrays (again)

- Allocating a multiple of the memory block required to store a particular pointer type effectively allocates... an array of that type!

```c
int *a,i;

a = (int *) malloc(6 * sizeof(int));
for(i=0; i<=5; i++){
    a[i] = 2 * i;
}
for(i=0; i<=5; i++){
    printf("a[%d] == %d\n", i, a[i]);
}
```

- Such a dynamic array doesn't exactly behave like a static array though...

Pointer and Arrays (final)

- The `sizeof()` function can be used to compute the number of elements of a static array.
- It doesn't work on dynamic arrays allocated through `malloc()`.

```c
int *a, b[6];
a = (int *) malloc(6 * sizeof(int));
printf("Size of array 'a' is %d\n", sizeof(a)/sizeof(int) );
printf("Size of array 'b' is %d\n", sizeof(b)/sizeof(int) );
```

- Using dynamic arrays requires some bookkeeping to avoid access to out-of-bounds memory.

Pointers and Functions

- Pointers being variables, they can be passed to functions as arguments.
- A function can also return a pointer as its result.
- In fact the FORTRAN reallocate function did just that.

Pointers as function arguments

Works

- Arguments passed by value to a function are *copied*. The function is then free to modify the value of the argument without impacting the original variable. This is a *call by value*.
- Pointers are no exception, C only passes a copy of the pointer's value to the function.
- This value being a memory address, the function is then free to use it to access or modify that memory content.

```c
void addonetome (int *me){
  *me = *me + 1;
}
void main(){
  int i = 1;
  int *p = &i;
  addonetome(p); printf("i = %d\n",i);
}
```

- The function above modifies the *state* of the program. It is said to have a *side effect*.

Pointers as function arguments

Still Works

- As mentioned earlier, pointers are passed by value, so modifying a pointer argument does not impact the original variable.

```c
void addonetome (int *me){
  *me = *me + 1;
  me = NULL;
}

void main(){
  int i = 1;
  int *p = &i;
  addonetome(p);
  printf("i = %d\n",*p);
}
```

Dynamic Arrays as function arguments

Works if careful

- As opposed to other simple types, a C array is *not* passed by value.
- When an array is passed as an argument, the function receives a pointer to the first element of the array.
- An array passed as a function argument therefore behaves like a dynamic array: no way to get size automatically.

```c
void printarray(int a[]){
    int i;
    for (i=0; i< sizeof(a)/sizeof(int); i++){
        printf("%d\n", a[i]);
    }
}// fails. only prints 1st element

void printarray_n(int a[], int n){
    int i;
    for (i=0; i< n; i++){
        printf("%d\n", a[i]);
    }
}// works, given a valid n
```

Dynamic Arrays as function arguments

Works

- As an array passed as an argument generates a pointer, you can write your function to accept pointers.
- Notice the difference between the two `int a[]` and `int *a`.

```
void printarray_n(int a[], int n){
     int i;
     for (i=0; i< n; i++){
          printf("%d\n", a[i]);
     }
}
// function above same as:
void printarray_n(int *a, int n){
     int i;
     for (i=0; i< n; i++){
          printf("%d\n", a[i]);
     }
}
```

Function returns pointer

Works

- A pointer is a valid function return value if the function is thus defined.

```c
int *pointertothatint(int n){
// this useless function returns a pointer to an
// int of a chosen value. Useless, but works as
// intended.
    int *fp;
    fp = (int *) malloc(sizeof(int));
    *fp = n;
    return fp;
}

void main(){
    int *mp;
    mp = pointertothatint(14);
    printf("*mp = %d\n",*p);
}
```

Function returns pointer incorrectly

Doesn't Work!

- Warning this will not work because as the argument is passed by value, it is copied to another place in memory.

```
int *anotherpointertothatint(int n){
// same as before, but with a big scope issue
    int *fp;
    fp = &n;
    return fp;
}

void main(){
    int *mp;
    mp = pointertothatint(14);
    printf("*mp = %d\n",*p);
}
```

- When the function exits, the memory for the variable *n* is deallocated, so it may be filled up by something else at any time.

Function returns pointer to array

Works

- Using the pointer/array equivalence, we can return a dynamic array from a function.

```c
int *nintarray(int n){
    int *p;
    p = (int *) malloc(n * sizeof(int));
    return p;
}

void main(){
    int *p, i, nelements=100;
    p = nintarray(nelements);
    for(i=0; i<nelements; i++){
        p[i] = i+1;
    }
}
```

Summary

- This week we discussed:
  1. pointers,
  2. arrays and pointers,
  3. dynamically declared arrays,
  4. issues with dynamically declared arrays, memory leaks.