3. Memory

Ján Dugáček

September 11, 2017

Table of Contents

Pointers

- Every variable is stored at a certain address
- This address can be obtained and is often very useful
- The variable type that contains an address is called *pointer*, pointer to a different variable type is itself a different type
- A pointer to an integer is written as `int*`, a pointer to a character as `char*`, pointer to a pointer to an integer as `int**`
- The size of a pointer is usually 8 bytes (64-bit architecture) or 4 bytes (32-bit architecture)
- This can be explained in several slides, but learning to use it properly takes a lot of time

Pointers #2

```
int a = 2;
int* b = &a;
int c = *b;
*b = 3;
```

- Here, we first define and set a variable a
- Then, we get its address and save it into variable b
- After, access the contents of whatever b is pointing that (it is variable a) and save into variable c
- Last, we save 3 into the contents of what b is pointing at, setting variable a to 3

Pointers #3

```
void swap(int* a, int* b) {
        int buf = *a;
        *a = *b;
        *b = buf;
}

int main(int argc, char** argv) {
        int a = 3;
        int b = 8;
        swap(&a, &b);
```

- Function `swap` takes addresses of two integers and swaps their contents (the addresses are copied as arguments, their contents are the same, so it changes the values outside of the function)
- `int* a` can be written as `int *a`, `int * a` or `int*a`

Const pointers

```
const char* a;
char* const b;
const char* const c;
```

- a is a pointer to a constant, so the pointer may be changed, but the data it points to cannot
- b is a constant pointer, so the pointer cannot be changed, but the data it points to can
- c is a constant pointer to a constant, so neither the pointer nor the data can be changed

Pointers and arrays

```
int sum(int* arr, unsigned int size) {
        int total = 0;
        for (unsigned int i = 0; i < size; i++)
                total += arr[i];
        return total;
}

int main(int argc, char** argv) {
        int array[] = {2, 3, 8, 0, 21, 3983989, 42};
        int inTotal = sum(array, 7);
```

- An array is used identically as a pointer to the first element of the array and a pointer is used as an array that begins at address contained in the pointer

Pointers and arrays #2

```
long int pack(short int a, short int b, short int c, short in
        short int array[] = {a, b, c, d};
        long int* together = (long int*)array;
        return *together;
}
```

- This is a function that packs four small variables of type `short int` (2 bytes) into a single variable of type `long int` (8 bytes)
- First, we copy these variables into an array
- The array can be used as a pointer of type `short int*`, but we change its type to pointer of type `long int*` (the contents of the variables is not changed)
- Then we can read the array as a single variable of type `long int`
- We return its copy as a result

Pointers summary

- Star(s) between variable type and variable name in declaration (`int* a`) - declaration of a pointer variable
- Star(s) before variable name outside declaration (`*a`) - access to address the pointer contains
- Ampersand before variable name outside declaration (`&a`) - obtain pointer to the variable
- *Stars before variable remove stars from its type, ampersands before variable add stars to its type*
- Pointer multiplication is useless and thus undefined, so these operations are not easily confused with multiplication

Pointer arithmetic

```
int sum(int* arr, unsigned int size) {
        int total = 0;
        for (unsigned int step = 0; step < size; step++) {
                total += *arr;
                arr++;
        }
        return total;
}
```

- This function does the same as the earlier one
- Incrementing works on pointers, it does not increase the address by 1, but by the size of the object whose address they hold (if the pointer is an array, then it's the next element)
- Adding to pointer increases its address by a multiple of the size of the object
- Subtraction and decrementing works too
- You can set 0, or `NULL` to a pointer as a reliably wrong value to set an empty pointer (and use it in a condition)

Exercises

1. Write a function that computes both sine and cosine of a number, saving the results into two variables whose addresses are given to the function as arguments
2. Write a function that swaps two pointers
3. Write a function that uses pointer arithmetic to return the sum of elements in an array, assuming that the last element is 0
4. Write a function that sums elements of two arrays, saving the sum of each pair into third array (three arrays and sizes given as arguments)

Advanced:

1. Write a function that splits a string (`char*`) into two strings according a separator given as the second argument, the two strings' addresses will be written to variables with addresses given by the user (you can split a string by setting a terminating 0 at the end of one and saving the pointer to the beginning of the second one)
2. Write a program that reads input (using `getchar`) from a line of any size and then writes it back (without using dynamic allocation)

Debugging practices

- The program does not work - it happens - we say that it's *bugged* and thus it needs *debugging*
- The most usual approach is to check the contents of variables, it can be done with `printf` for review when the program has ended
- Finding where the results are unexpected usually helps pinpoint the place where the error happens
- If the program writes into addresses that are nowhere near a place where it's allowed to write, the program crashes, usually discarding some recent stuff written into the command line, in that case, use `fprintf` to print into `stderr`

```
long int array[16];
int* pointer = (int*)malloc(sizeof(int));
array[16] = 241352523; // Ooops, overwrites pointer
fprintf(stderr, "pointer: %p", pointer); // See what's up
*pointer = 10; // Writing at address 241352523, crashes
```

- Use it like `printf`, but use `stderr` before the usual arguments

Debugging programs

- When debugging, it's very useful to compile with -g (makes programs larger) and without -O3 (without it, programs are slower)
- The -g argument allows debugging programs to find the line in source code that caused a problem
- *gdb* is a debugging program that runs programs in a special interface that can break at given line, view variables' contents and investigate where crashes happened, without slowing programs down; it is directly controlled by QtCreator
- *valgrind* is an interpreter of assembly code, it allows finding where crashes happen, it can detect invalid reads (reading from variables that are not allocated), invalid writes (writing into variables that are not allocated) and detect memory that has not been freed; it slows down the program considerably and modifies its reaction to invalid reads or writes (may cause it to stop crashing on it or to start crashing on it)

Memory sections

There are 5 places in memory used by the program

- *Stack* - the place where local variables are created, variables here are automatically destroyed when their scope ends, size is limited to several megabytes
- *Heap* - access granted on demand, exists until asked to free it, often called *dynamic allocation*
- *Constant data* - constants defined in code, may be read, but writing there causing the program to crash
- *Non-constant data* - global variables (defined outside any block) are stored here
- *Code* - the compiled source code, can't be neither changed nor read on most OS

Heap

- Why do we need it?

```
int* sumArrays(int* arr, int* arr2, unsigned int size) {
        int arrSum[size];
        for (unsigned int i = 0; i < size; i++) {
                arrSum[i] = arr[i] + arr2[i];
        }
        return arrSum;
}
```

- The code above does not work properly, because variable `arrSum` is destroyed when the function ends, meaning that the result will be overwritten very soon (and editing it then will change new variables created in its place)

Heap #2

```
#include <stdlib.h>
int* sumArrays(int* arr, int* arr2, unsigned int size) {
        int* arrSum = (int*)malloc(sizeof(int) * size);
        for (unsigned int i = 0; i < size; i++) {
                arrSum[i] = arr[i] + arr2[i];
        }
        return arrSum;
}
```

- We can allocate memory on heap that will stay allocated until explicitly freed or the program ends
- It is allocated by function `malloc` from library `stdlib.h`
- `malloc` does not care about the variable type we'll store there, so we must set the size in bytes (size of the array times size of an element, **not only the size of the array!**)
- `sizeof` is not a function, it is replaced by the size of the variable type in brackets by the compiler, **be careful not to get the size of a pointer instead of the size of the variable whose address it holds!**
- `malloc` returns a pointer to unspecified type, `void*`, you need to change its type

Heap #3

```
int a[] = {2, 8, 666};
int b[] = {7, 9, 777};
int* summed = sumArrays(a, b, 3);
// Do some stuff
free(summed);
```

- Sections between // and new line are ignored and can be used to disable parts of the code or to add a comment
- Here, we use the function `sumArrays` defined in the previous slide
- The memory allocated by `malloc` must be freed explicitly or the program will keep it until it ends (even if the variable holding its address is long forgotten)
- The memory is freed using function `free` from `stdlib` (you must use the pointer to exactly the same address as given by `malloc`)

Exercises

1. Write a function that returns an array of Fibonacci's numbers of size given by an argument
2. Write a function that resizes a dynamically allocated array to a new size, returning the new pointer and copying the contents of the old one to the new one
3. Change the function from the previous exercise so that it returns no value, but accepts a pointer to the array it has to resize
4. Write a program that reads the first argument and prints the section until the first occurrence of q, assuming that the first argument can be of any size and you cannot create an array that is too large

Advanced:

1. Write a function that splits a string into strings according a separator given as the second argument, returns a dynamically allocated array of strings (`char*`)
2. Write a function that sorts an array of numbers (any algorithm except *bogosort* or similar)

Homework

- Write a function that uses one number as argument, it finds the closest greater prime number and returns an array whose each element is an array of prime divisors of the numbers between the given number and the closest greater prime
- Advanced homework: Create a calculator program that reads a formula with numbers, single letter long variables, plus, minus, multiplication and division from command line arguments, assuming that all operations are brackets to avoid dealing with operator priority as first argument and finds values of these variables where the sum of its values in range given as second and third argument is the closest to zero