Data structures in C

Algorithms and data structures ID1021

Johan Montelius

Fall term 2024

Introduction

This is a quick introduction to data structures in C. The syntax, control flow, how to compile and how to run programs are things that I assume basic knowledge in. This tutorial is about how data structures are used and what you need to think about. I will assume that you know how to program in Java and will thus point out the similarities and differences.

Primitive data type

As in any programming language we have a set of primitive data types in C. In Java you have used for example: `int`, `double`, `boolean` and `char`. You have very similar but even more primitive types in C. If you want to hold an integer you can in Java choose from: `byte`, `short`, `int` or `long` depending on how large integers you will use. All these types are signed types so a `byte` can hold values from -128 to 127.

In C you can in a similar way choose from: `char`, `short`, `int` and `long` but you also have the types: `unsigned char`, `unsigned short` etc. This allows us to be more precise in our program and might allow the compiler to produce more efficient code.

In Java you have probably also used the wrapper classes: `Integer`, `Double` etc but since C is not an object oriented program we have nothing equivalent.

call by value

C uses, as Java, a strategy called *call-by-value* when passing argument to a procedure. This means that a value is copied (on the stack) and passed to the procedure. The procedure can of course make use of this value but it can not change the original value.

In the code below there is no discussion about what `x` is, it's obvious a variable of type `int` with the value `42`. When we pass `x` to the procedure `foo()` we pass a copy of the value `42` to the procedure. The procedure can change the value of the local variable `a` but this has no effect on the variable `x` in `main()`.

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

void foo(int a) {
  printf(" a = %d\n", a);
  a = 37;
  printf(" a = %d\n", a);
}

int main() {
  int x = 42;
  foo(x);
  printf(" x = %d\n", x);
}
```

Sometimes we do want to change the original value and of course use this all the time in Java. If we have created an object the variable holds a reference to the object. When this value is copied and given to a method, the code of the method has access to the object and can of course (if allowed) change the values of the object attributes.

This is all done by default in Java and you probably have not thought about it too much. In C however, you are in control over how we should pass a value and you have to make up your mind.

pointers

This is the most complicated thing with C and you must understand what you're doing. In the code below we have a procedure `bar()` that takes *a pointer* to an `int` (`int *`) as an argument. We call the procedure and pass a pointer to x (`&x`) as an argument.

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

void bar(int *a) {
  printf(" *a = %d\n", *a);
  *a = 37;
  printf(" *a = %d\n", *a);
}

int main() {
  int x = 42;
  bar(&x);
  printf(" x = %d\n", x);
}
```

When we pass the pointer to a value we can change the value it self. Note that you decide if the value or a pointer to the value should be passed as an argument and the procedure is defined to accept either or. In Java this is done without you noticing. If you declare a variable as an `Integer` and pass it to a method, a reference to the object is passed. You will however not be able to change the properties of the Integer object (I know how to do this and I'll show you) since this would be very confusing.

Structures

A compound data structure is called `struct` in C and consist of a set of named elements. When you define a struct you will almost always also declare a short hand name for it and this might look a bit confusing. The reason is that we have two declarations that are written in one statement.

declarations

The first declaration is a `typedef` declaration that introduces a short-name for a data structure. If we are going to use a color in our program and a color is represented by an `int` (just as an example) we could have the following declaration:

```
  typdef int color;
```

We could then use the label `color` as a type in our program instead of `int`. The advantage is that we later realize that we only need a `char` to represent a color we can do this at one location.

A compound data structure is declared as follows:

```
  struct rgb {
    char red;
    char green;
    char blue;
  }
```

This would declare a structure `rgb` with three components but every time we would like to use it we would have to write `struct rgb`. To make life a bit easier we could of course define a short-name for this with the following declaration:

```
typedef struct rgb color;
```

Another way to write this is doing it in one statement:

```
typedef struct rgb {
  char red;
  char green;
  char blue:
} color;
```

What is a bit confusing is that you would most often use the same name for the struct as the short-name so the declaration will probably look like this:

```
typedef struct color {
    char red;
    char green;
    char blue:
  } color;
```

It looks like we are redundantly use the label twice and in one way we are but I hope it is less confusing now when you know what we are doing.

call by value

Remember that C is (as Java) using call-by-value i.e. the value of an argument is copied and passed to the procedure we call. This happens even if the value is a struct. In the examples below, determine the final value of `clr.red`:

```
void foo(color c) {
  c.red = 41;
}

int main() {
  color clr = {31,32,33};
  foo(clr);
  printf("clr.red = %d\n", clr.red);
}
```

You can also, as with primitive values, return a struct from a procedure. The structure copied back to the calling procedure and things behave as you would expect.

```
color bar() {
  color clr = {51,52,53}
  return clr;
}

int main() {
  color clr = bar();
  printf("clr.red = %d\n", clr.red);
}
```

In order to change the properties of the structure we need to pass a pointer to the structure. This is very similar to how we work with primitive values. Try the following:

```
void grk(color *c) {
  c->red = 61;
}

int main() {
  color clr = bar();
  grk(&clr);
  printf("clr.red = %d\n", clr.red);
}
```

Note how we access the properties of the structure (`->`) now that we are given a pointer to the structure. What we here do explicitly is what is done in Java implicitly when we pass an object as argument to a method.

The heap

To understand why we have a *heap* you need to understand the limitations of the stack. The stack is a, as the name suggests, a memory area where we add new data structures on the top; we can also remove data structures but only from the top. When you call a procedure we add a *stack frame* on the stack were the called procedure can allocate its local data structures that it needs.

When the procedure returns, any data structure could be copied and returned as a result. The structure needs to be copied since the stack frame of the procedure will be removed from the stack and all its data structures invalidated.

If a data structure should survive after we have returned from a procedure, it can not be allocated on the stack. This is where *the heap* comes into play.

malloc and free

In C allocation on the heap is done explicitly using a call to the procedure `malloc(int size)`. The procedure will return a *void pointer* i.e. a pointer to an area where the data structure is allocated. To show how this is done we can continue with the color example. In the code below the procedure `zot()` allocate space large enough to hold a color structure. Note how it *cast* the return value from `malloc()` to become a pointer to a color structure.

```
color *zot() {
  color *clr = (color*)malloc(sizeof(color));
  clr->red = 71;
  clr->green = 72;
  clr->blue = 73;
  return clr;
}

int main() {
  color *clr = zot();
  printf("clr->red = %d\n", clr.red);
  free(clr);
}
```

In `main()` we call `zot()` to obtain the pointer to the color structure. We print the value of the red field and then *free* the structure. The call to `free()` will deallocate the memory area that was previous allocated. In this small example it does not matter but in a real program you have to make sure that data structures that are not longer needed are freed. If you just allocate new structures without freeing the ones not needed you will run out of memory.

Arrays

So now that you have learned how data structures are passed to and returned from procedures you need to change this understanding since it is not valid for array. For historical reasons arrays are not treated as regular data structures - C will never copy an array when passing it as and argument to or returning it from a procedure. Let's implement the equivalent procedures but now using an array.

In the first example we called a procedure with a copy of the color structure. The same thing using an array will not work as expected (if you expect the array to be copied).

```
void foo(int c[]) {
  c[0] = 41;
}

int main() {
  int clr[] = {31,32,33};
  foo(clr)
  printf("clr[0] = %d\n", clr[0]);
}
```

As you see a pointer is passed as the argument and the procedure `foo()` will change the content of the array. The procedure `foo()` could be described like this with exactly the same meaning:

```
void foo(int *c) {
  c[0] = 41;
}
```

The second procedure, `bar()`, has no equivalent when using arrays. C will not copy an array when it returns a value. You can try the following but it will not even compile:

```
int []bar() {
  int clr[] = {51,52,53};
  return clr;
}
```

You could try the following but this will first of all generate a warning and even if you ignore this it turns out that the compiler returns a null pointer rather than returning a pointer to an array that is located on the stack.

```
int *bar() {
  int clr[] = {51,52,53};
  return clr;
}
```

Passing a pointer instead of a copy as an argument is the way C works so the `grk()` procedure will work as expected. The difference is that we should not pass the address of the variable but the variable it self, exactly as we did when we called `foo()`.

The procedure `zot()` will also work exactly as the data structure version. An array is allocated on the heap and a pointer to the array is returned.

```
int *zot() {
  int clr[] = (int*)malloc(3*sizeof(int));
  clr[0] = 71;
```

```
    clr[1] = 72;
    clr[2] = 73;
    return clr;
  }

  int main() {
    int *clr = zot();
    printf("clr[0] = %d\n", clr[0]);
    free(clr);
  }
```

I highly recommend that you copy the above procedures and try them out. Remember: variables are normally passed as value (a copy) but not if it's an array, then it is passed as a pointer.

Memory management

Knowing how to work with the heap: how to allocate data structures and when to deallocate them, is something you need to master when programming in C. If you only have been programming in Java you might never have thought about how data structures are allocated and probably never have thought about when they are deallocated. Those days are gone but when you master C and return to Java you will have a better understanding of what takes place under the hood.

Your first problem is to determine if a data structure should be allocated on the stack or on the heap. If it is an array where the size is only known at run-time then you have no choice, it has to be allocated on the heap. If it's a structure with known size you have a choice and you need to determine for how long time the structure should live.

The simple answer is that if a data structure should survive after the return of the procedure then the structure must be allocated on the heap. However, sometimes you have the choice of allocating a structure on the stack and pass a reference to it instead of allocating a the structure on the heap. The following program shows the two options:

```
  color *zot() {
    color *clr = (color*)malloc(sizeof(color));
    clr->red = 71;
    clr->green = 72;
    clr->blue = 73;
    return clr;
  }
```

```
void zit(color *clr) {
  clr->red = 71;
  clr->green = 72;
  clr->blue = 73;
}

int main() {
  color *ptr = zot();
  printf("ptr->red = %d\n", prt->red);
  free(ptr);
  color clr;
  zit(&clr);
  printf("clr.red = %d\n", clr.red);
}
```

In this case it would be more efficient to allocate the structure on the stack and pass a reference to it as an argument (i.e. `zit(&clr)`). This is however a special case and it is more common that we use the first alternative `zot()` since the number of structures to allocate and their life time is often a run-time decisions.

always free but never twice and ...

Allocating things on the stack or heap is one decision that you need to master. If the decision is to allocate on the heap you need to determine when it is time to free the structure. When we free a structure we return it to the underlying memory manager so that the memory could be used for other purposes. If you forget to free a structure and the code is executed over and over again then you will run out of memory after a while - this is called a *memory leak*.

If you do a mistake and accidentally free a structure twice then anything could happen. Your program will most likely crash with a segmentation fault and will be very hard to understand why. The code that crashes the system could be miles away from the place where you do the erroneous free operation.

A similar problem occurs if you free a structure but forget that you have and accesses the structure again. What will happen is not defined and you might not even detect the error... until everything crashes with a segmentation fault and you have no idea of where to start looking for the error.

In the operating systems course you might have as an assignment to implement `malloc()` and `free()`. Then you will understand why things will break if you don't do the right thing. Until then the rule is: free exactly once and never touch a freed structure.

Java ....

If you have programmed in Java you might wounder why C can not do the same. Why do you have to decide if a structure should be on the stack or on the heap why do you have to keep track of when a data structure is no longer needed? The answer is efficiency; a program written i C is typically two to three times faster than the same program written in Java. Sometimes this is important and then C is the preferred language. Another answer is more control of resources; in C you know how much memory your program will use, in Java this is hidden. If you're developing embedded systems you might also have to interact directly with different memory areas, something that is possible in C but this is not part of this course.

You might wounder how Java can determine if a data structure is not any longer needed and the answer is that it can not. What it does is that it periodically stop the execution and determine which data structures that are still accessible from the execution state. These structures are copied to a new memory are and the old memory are is reclaimed. This is called `garbage collection` and is of course not inexpensive. Java is a programming language where this is an accepted cost since the ease of programming has priority. Doing the same in C would not be acceptable since you want to be in control of the execution.

There are languages that fall in between Java and C. Rust is one example that is on the same level as C but some restrictions in the language allows the compiler to insert free operations exactly where they should be. Rust then becomes a robust alternative to C, the efficiency is almost the same but you do not have to think about memory management.

Even if the memory management is very explicit in C you do not have any control over what happens inside `malloc()` and `free()`. In most cases this is fine since the operations are very efficient and in most cases are constant time operations. However, if you use a library you do not know if and how much this library will use the heap. Most of the time this is not an issue but sometimes you need to be in control all the way. This is were a programming language like Zig comes into play. In Zig you will have more control over how the memory allocation is done and a library routine can not allocate memory without you knowing about it.