Lecture Notes on Memory Layout

15-122: Principles of Imperative Computation
Frank Pfenning   André Platzer

Lecture 11

1 Introduction

In order to understand how programs work, we can consider the functions, their preconditions and postconditions, check whether loop bodies preserve the loop invariants, and reason about why the implementation achieves the postconditions without violating array bounds and without attempting invalid pointer accesses. This is a very useful and powerful style of understanding how a program works. It emphasizes the logical and computational thinking aspects of programming.

Another way of understanding how a program works is more operational where we follow the dynamic behavior of the program and understand what exactly each of its steps does. This is what today's lecture is about.

Both approaches of reasoning about programs, the logical and the operational are equally important and often go together hand in hand very well. In fact, in order to check whether a loop body preserves its loop invariant, we already need to understand the operational effect that the loop body has on the data.

So how, exactly, does a program operate, and what exactly happens to our code when we run it?

2 Data in Memory

Ultimately, "all" data resides in memory. In fact, part of the data may also be kept in fast registers directly on the CPU. You will learn about registers in detail in 15-213 and can learn about their use in programming languages in 15-411. For the purposes of today's lecture, it is sufficient to pretend all data would sit in memory and ignore registers for the time being. This simplifies the principles without losing too much precision.

The data in memory is addressed by memory addresses that fit to the addressing of the CPU in your computer. We will just pretend 32bit addresses, because those are shorter to write down. All addresses are positive, so the lowest address is 0x00000000 and the highest address $2^{32} - 1 =$ `0xFFFFFFFF`. All data (with the caveat about registers) sits in memory at some address. One important question about all data in memory is how big it is, so that the compiler can make sure program data is stored without accidental overlapping regions.

The basic memory layout looks as follows:

```
OS AREA
============
System stack   (local variables and function calls)
============
unused
============
System heap    (data allocated here... alloc or alloc_array)
============
.text (read only)  (program instructions sit in memory)
============
OS AREA
```

One consequence of this memory layout is that the stack grows towards the heap, and the heap usually grows towards the stack. The reason that the stack is called a stack is because it operates somewhat like the principle of the stack data structure that from Lecture 10. Your program can put new data on the top of the stack. It can also pop elements of the stack if this data is no longer necessary. Unlike the stack abstraction from Lecture 10 it may appear as if your program internally also modifies data that is on the stack, even if it is not quite at the top of it. However, the only data on the stack that the program modifies is in the top range of the stack (perhaps the top 512 bytes or so, depending on the function that runs), even if it is not just the top word of the stack.

Programs cannot access memory cells that belong to the operating system. If they try, programs get an "exception" like a segmentation fault. Where can that happen in C0? C0 takes great care to ensure that it never gives you any pointers to uninitialized or random or garbage data in memory, *except*, of course, the `NULL` pointer. `NULL` is a special pointer to the memory address 0, which belongs to the operating system. Any access by a user-land program by dereferencing a NULL pointer causes a segfault.

If, however, you are writing a program that will be running as part of the operating system, your program has no protection against `NULL` pointer dereferencing anymore, because memory address 0 is a valid address for the operating system, even if not for regular programs. When writing code for operating systems, you, thus, need to have mastered the art of protecting against illegal `NULL` dereferences. This is exactly one of the things that contracts, loop invariants, and assertions prepare you for.

3 References, Pointers, and Structs

In order to explain how members of a struct are accessed, we digress briefly to introduce *pointers*, although we will use them in only a very limited form in this lecture. C0 distinguishes between *small types* and *large types*. Small types have values that can be stored in variables and passed to and from functions. Large types can only be stored in memory. In order to access them we pass *references* or *pointers* to them. Alternatively, we can think of passing around their *address* in memory, rather than the values themselves. Once we pass around an address of data in memory, the program that receives that address can easily read the memory contents at that address and possibly even change it.

Which types are small and large so far? It seems pretty clear that `int`, `bool`, and `char` are explicitly designed to be small. The natural size of a value of type `int` on recent architectures is 32 or 64, although in previous generations of processors it was 8 or 16. The C standard does not tell us the size of an `int`. They could be 16bits or 32bits or 64bits. In each case, we get modular arithmetic overflow at different numbers. In C0, `int` are fixed as 32bit (signed) integers. In C0 we fix the size of `int` to be 32. There are only two booleans, so we might expect them to be 1 bit. The processor architecture, however, has a natural word size which is handled most efficiently, so they may actually be implemented to take more space. Similarly, ASCII character values of type `char` should take 7 or 8 bits, although in reality an implementation might allocate more space for them.

The type of arrays is an interesting case. We might expect `t[]` to be a large type, since each array takes a fixed, but unbounded amount of space. But we have been passing them as arguments to functions and assigning them to variables without any problems. The reason is that a value of type `t[]` is actually a *reference* to an array that is stored in memory. Such a reference fits within the word size of the machine, since addresses are a basic type that the machine can manipulate. On most recent architectures, an address will take either 32 or 64 bits, depending on the configuration of the compiler and machine. The current environment in use for this class uses 64 bits, although there is no effective way to tell the difference from within a C0 program.

In summary, arrays with elements of type t and length n are allocated in memory, with `alloc_array(t,n)`. Such an allocation returns a *reference* to the array. In memory, this array will need space (at least) $|t| * n$, where $|t|$ is the size of the element type t. An array `alloc_array(int,10)` would need at least 40 bytes in memory, for example. An array `alloc_array(char,10)` could be stored with 10 bytes. C0 uses a slightly larger memory portion to store the length of the array in order to be enable dynamic checking of contracts and assertions that refer to `\length(A)`.

When two variables of type `t[]` are the *same* reference we say that they *alias*. As a programmer, it is important to be aware of this because assignment to an array variable does *not* copy the contents of the array, but only assigns references. For example, after the operations

```
int[] A = alloc_array(int,5);
int[] B = alloc_array(int,7);
A[2] = 37;
B = A;
B[2] = 11;
```

A and B alias, and we have `A[2] == 11` and also `\length(B) == 5`. In fact, the allocation of B is redundant, and we could just as well have written

```
int[] A = alloc_array(int,5);
int[] B;
A[2] = 37;
B = A;
B[2] = 11;
```

Besides arrays, strings are also manipulated by reference and therefore `string` is a small type.

In contrast, structs are *large* types, because they (usually) do not fit into a single memory cell (or register). This means that they are allocated in the heap and can *not* be stored in variables or passed to functions. Instead of references, as for arrays, we use explicit *pointers* when passing them to or from functions or assigning them to variables. In fact, because `struct s` is a large type it is an error to try to pass it to a function or directly declare a variable of such a type.

In general, we write `t*` for a pointer to a value of type t in memory. In this lecture we only use it in the form `struct s*`, that is, a pointer to a struct in memory. Structs (and other values) are allocated in memory using the form `alloc(t)` for a type t. This allocation returns a pointer to the new memory location, and therefore has type `t*`.

If we have a pointer p of type `struct s*` we refer to the field f of the struct using the notation `p->f`. To write to memory we use it on the left-hand side of an assignment, to read from memory with use it as an expression.

For example, consider

```
struct list {
       int data;
       struct list* next;
};
```

Then a variable `struct list* l;` would need the size of 1 pointer (e.g., 32bit or 64bit) to store. An operation

```
struct list* l = alloc(struct list);
```

would allocate a piece of memory on the heap where the struct list can be stored. That piece of memory at least has the size 8 bytes (on a 32bit architecture) or 12 bytes (on a 64bit architecture), because we need 32bit to store the `int data` information and another 32bit or 64bit to store the address that `next` points to. In 15-411, you will learn that the actual size of `struct l` may actually be slightly larger to make sure all information is aligned properly in memory, in a way that makes the access for the CPU maximally efficient, which is called padding.

4 Variables on the Stack, Allocated Data on the Heap

As an example to understand what really happens operationally, we consider a program that builds a linked list with the values from 1 to n from start to end. So the goal is to build a list in which we will find the values from 1 to n when we traverse it from its start to its end (NULL). The convenient way to do that is to let the function build a list from the end to the start. It obviously needs to fill in the numbers in reverse order, which is what the reverse order `for` loop accomplishes, in which i decreases after each iteration.

```
struct list* upto(int n)
//@requires n >= 0;
//@ensures list_length(\result) == n;
{
       struct list* r = NULL;
       for (int i = n; i >= 1; i--)
       //@loop_invariant 0 <= i && i <= n;
       {
               struct list* l = alloc(struct list);
               l->next = r;
               l->data = i;
               r = l;
       }
       return r;
}
```

First note that we initialize `r` to be NULL, because every variable must be initialized. If you allocate an array using `alloc_array(t,n)`, its elements however are going to be initialized, but single variables are not. Now one may wonder why the `alloc(struct list)` operation returns a pointer to a `struct list`, instead of the `struct itself`? From what we have learned about the memory layout, we can see that the reason is that structs are too big to fit into single memory cells (and also too big to fit into a register). Structs are large types, but functions can only return small types, so `alloc` returns a pointer to `struct list` instead, which is a small type and fits conveniently into a memory cell or a register.

When running the `upto` function, its local variables need to be put in memory at some place (remember that we ignore registers to simplify matters). The arguments to the function also need to be passed to it and will, thus, be kept on the stack (ignoring registers).

For example, for a call of `upto(2)` the stack layout for the (essential part of the) stack frame of `upto` will look as follows:

```
stack:
n  2
r  0x00 (represents NULL)    (set to 0xDE after node is initialized)
```

```
i  2
l  0xDE (example of an address on the heap)
```

Upon allocation via `l = alloc(struct list)`, the allocation will allocate a region on the heap that is large enough to fit data of type `struct list` and will return a pointer to it, i.e., the address of that newly allocated region in memory. In this example, we assume the address is `0xDE`. The information contained in the struct that l points to is stored on the heap, relative to this base address:

```
heap:
   l->data is stored at 0xDE  (value set to 2)
   l->next is stored at 0xE2  (value set to 0x00)
```

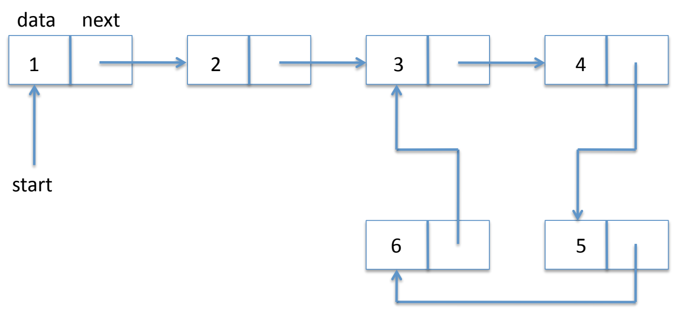
The `alloc` function already initializes all elements of the struct to default values. The default value for such `int` values in a struct or array are 0. For pointers they are `NULL`, which coincides with the numerical representation of `NULL`, which is the address 0.

When the loop repeats, the same record on the stack stores the values of the variables. In particular, the old values get overwritten by the new ones.

```
stack:
n   2
r   0xDE                     (value set to 0x13 after node initialized)
i   2
l   0x13 (example)

heap:
   l->data is stored at 0x13  (value set to 1)
   l->next is stored at 0x17  (value set to 0xDE)
```

In our pictures of pointer structures like linked lists, when we draw an arrow, it means that cell has an address that references the thing the arrow is pointing to. That is, for example, what we meant by the pointy arrows in the following diagram of a list structure from Lecture 10:



5 Stack

When you call a function, the system stack has a *stack frame* that holds space for function arguments and local variables. Every time a function is called a new frame is created and *pushed* on the system stack. Now a good question is why it is not sufficient to just work with one single stack frame per function? Before you read on, try to figure out this answer for yourself.

The reason why a single function may need more than one frame on the stack is because that function could be called recursively. Then, if a function would simply overwrite the data in its stack frame, it would interfere with the values of the corresponding arguments and local variables in other instances of the same function, that still expect to find their data intact.

Recall the recursive factorial function:

```
int fact(int n) {
       if (n == 0) return 1;
       else return n * fact(n-1);
}
```

Suppose we call `fact(4)`, then we end up with the following stack (recall that the stack grows upside-down towards the heap):

```
stack:
n    4
------
n    3
------
n    2
------
n    1
------
n    0
```

When a function returns, the top frame is popped off the system stack and the result is used in the function call belonging to the previous frame.

6 Excursion to Concurrency

Similarly, programs need to be careful where they pass their pointers to. If your program is the only one who has pointers to its own stack and heap, then your program is in control of what gets changed in that region of the memory. If, however, other programs (or other threads) also have pointers to the same memory regions, they could write data into the same memory as your program, possibly overwriting the information. This concurrency leads to very difficult situations. Recall

```
struct list {
       int data;
       struct list* next;
};
```

Suppose your program P1 has a pointer `list p;` pointing to memory address `0xAB0000C0`. Further suppose this pointer got communicated or leaked to another program P2 that has a pointer `list q;` with the same address `0xAB0000C0`. Now your program P1 may write the total number of students in this class as an integer into `p->data` by executing the first line of the program P1

```
  p->data = 241;
  long_computation;
  if (p->data >= 0) {
    //@assert false;
    // this should not happen
  }
```

But now suppose that, while `long_computation` is still running and before our program P1 gets a chance to read this data again in the `if` statement, the other program P2 suddenly overwrites the same information by running

```
  q->data = -10;
```

Then our program P1 may get very confused when reading from `p->data` again:

```
  if (p->data >= 0) {
    // this assertion may fail if P2 concurrently modified data
    //@assert false;
  }
```

This is the reason why programs that run concurrently with multiple programs running at the same time are very difficult. The model of concurrency we use here is that two programs running at the same time can just have their instructions executed in an arbitrary interleaving. So for each program, we know that its next instruction will only run after the previous instruction has completed, but we do not know which of the instructions in two separate programs runs first. It is possible that one program even completes before the other starts, just unlikely for long running computations.

Similarly, if a different program had a pointer with the same memory address, program data could change in surprising ways. Fortunately, the

C0 compiler is taking great care to ensure that C0 programs only ever get to know memory addresses via `alloc` and `alloc_array` that are “owned” by that C0 program and, basically, no other program knows about them. Under those circumstances, the C0 program can rely on never reading different data from memory that what it had put in there in the first place. Aliasing issues of pointers may, of course, still cause similarly surprising effect if the programmer is not careful about reasoning whether two pointers could have the same address.

```
struct duplexlist {
       int data1;
       int data2;
       struct duplexlist* next;
};
void swap(struct duplexlist* p, struct duplexlist* q) {
  int d1 = p->data1;
  p->data1 = q->data2;
  int d2 = p->data2;
  p->data2 = q->data1;
  q->data1 = d2;
  q->data2 = d1;
}
```

Exercises

Exercise 1 *Give an example of implementations for two functions* `int f(struct list* p);` *and* `int g(struct list* q);` *that use pointers. Give a scenario where they* `f(p)` *will return different answers depending on whether or not* `g(q)` *runs* concurrently, *i.e., both execute at the same time.*

Exercise 2 *Give an example of an implementation of a function* `int f(t* p, t* q);` *that will return different answers depending on whether the pointers* `p` *and* `q` *have been properly allocated, or whether one pointer points into the middle of a data structure on the heap (which C0 does not allow but C would). Unlike in Exercise 1, do not use concurrency in your solution.*

Exercise 3 *Give an example of an implementation of a function* `int f(struct list* p, struct list* q);` *that will return different answers depending on whether or not there is an aliasing situation in the data reachable from* `p` *and* `q`*. Unlike in Exercise 1, do not use concurrency in your solution. Unlike in Exercise 2, only use properly allocated C0 data, and do not use pointers that point into the middle of a data structure on the heap.*