Lecture 20 Notes
C's Memory Model

15-122: Principles of Imperative Computation (Summer 1 2015)
Frank Pfenning, Rob Simmons

1 The C0 and C Memory Model

When we talk about memory in C0, C1, and C, that memory was always in one of three places:

- *Local variables* (including the arguments to functions) are stored in memory. In both C0 and C, this memory is reserved automatically when we declare a new local variable, though the contents of that local memory aren't initialized. The local memory gets reclaimed as soon as the local variable goes out of scope.

- *Allocated memory*, which we reserved with `alloc` and `alloc_array`. We always accessed this memory by referring to its address (the address stored in a non-`NULL` pointer or the address of an array). When we reserved allocated memory in C0, this memory was initialized to default values for us. In C, `xmalloc` does not initialize memory.

- In C1, we said that the compiled code for functions was stored in *read-only memory*, and we could get pointers into that read-only memory by taking the address of a function: `&f`. Many more things in C are stored in read-only memory, like string literals.

In this class, we think about *all* of this data residing in memory.[1] In this picture, memory is comprised of an enormous array of bytes, where each index in the array of bytes is an address. The lowest address is 0, and the highest address on most modern, 64-bit processors is $2^{64} - 1 =$ `0xFFFFFFFFFFFFFFFF`. This is an almost inconceivably large array of bytes,

---

[1]Later classes will complicate this picture by talking about things like *registers* that you don't need to know about in 122.

but the operating system plays tricks to allow processors to pretend like they have access to this enormous array. One way this is done is by only giving programs access to individual *segments* of this array. Modern hardware prevents individual running programs from accessing memory outside their allocated segments. (This is a good thing: it means that, no matter what how much you mess up while programming in C, it's going to be difficult for you to interfere with other running programs like your text editor or virus scanner.)

The memory used for local variables of a function is allocated and deallocated according to a stack discipline, so we call this portion of memory the *stack*. Memory that we explicitly allocate is reserved in what we call the *heap*, though it has no relationship to the heap data structure. Read-only memory is also called the *text* region. Therefore, the big picture of memory looks like this:

```
0xFFFFFFFFFFFFFFFF |
                   | OS AREA
                   | ============
                   | System stack   (local variables)
                   | ============
                   | unused
                   | ============
                   | System heap    (allocated memory)
                   | ============
                   | .text          (read only memory)
                   | ============
                   | OS AREA
0x0000000000000000 |
```
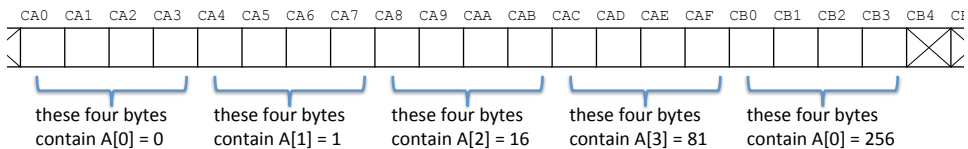
One consequence of this memory layout is that the stack grows towards the heap, and the heap usually grows towards the stack. Programs cannot access addresses (indices in this enormous array) that belong to the operating system. If they try, programs get an "exception" like a segmentation fault. C0 takes 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. The address 0 will usually not be a part of one of the memory segments you are allowed to read from or write to, so accessing the NULL pointer causes you to read or write outside of your segment: a *segmentation fault*, or segfault.

2   Arrays and Pointer Arithmetic

When compared to C0, the most shocking difference is that C does not distinguish arrays from pointers. We allocate enough space for a single integer by writing sizeof(int), and we allocate enough space for an array of 5 integers by just multiplying the size of a single integer by 5.

```
int *A = xmalloc(sizeof(int) * 5);
for (size_t i = 0; i < 5; i++) A[i] = i*i*i*i;
```

Assuming 4-byte integers, this 5 element array is treated by C as no more and no less than a single 20 byte memory segment that we are allowed to use. If the call to malloc returned the memory address 0xCA0, then after the for loop is done, the four bytes of memory addressed by 0xCA8 to 0xCAB will together represent the integer 16, the contents of the third index of the array, A[2], and so on:

| CA0 | CA1 | CA2 | CA3 | CA4 | CA5 | CA6 | CA7 | CA8 | CA9 | CAA | CAB | CAC | CAD | CAE | CAF | CB0 | CB1 | CB2 | CB3 | CB4 | CB |

these four bytes contain A[0] = 0

these four bytes contain A[1] = 1

these four bytes contain A[2] = 16

these four bytes contain A[3] = 81

these four bytes contain A[0] = 256

One consequence of this conflation of pointers and integers is that writing *A and writing A[0] necessarily means the same thing. We can also add an integer to a pointer, but this modifies the pointer not in terms of bytes but in terms of array elements. This allows us to get pointers into arrays!

```
int *x = A + 3;
*x + 5;
```

In the example above, then after running these two lines, the local x will contain the pointer 0xCAC, and the assignment will cause both *x and A[3] to evaluate to 86 instead of 81. This is a form of aliasing that was impossible in C0, but it is relatively common in C. But because only the pointer 0xCA0 in the example above was returned from xmalloc, only that pointer should be freed: it would be a memory error to free the pointer 0xCAC stored in x.

When we allocate very large arrays, we may want them allocated with default values, the way we did in C. The C standard library provides a function, calloc, to do just that, and our xalloc.h library has a non-NULL-returning xcalloc version of calloc that you should use. With xcalloc, we can allocate an array of seven elements, all of which are initialized to 0, like this:

```
int *B = xmalloc(7, sizeof(int));
for (size_t i = 1; i < 7; i++) A[i] = A[i-1]*2 + 3;
```

The only differences between xmalloc and xcalloc is that the latter initializes the memory to be all zeroes and takes two arguments. The xcalloc function takes two sizes n and m and allocates $n \times m$ bytes. We think of xmalloc as being like C0's alloc and we think of xcalloc as being like C0's alloc_array, but in C you can allocate arrays and pointers with either xmalloc or xcalloc.

3 Undefined Behavior

We have described the following as memory errors:

- Reading uninitialized memory (on the stack or on the heap)

- Dereferencing memory outside of an valid allocated segment (which includes NULL pointer dereference, array out of bounds errors, or trying to read or write to an int when you've only allocated enough size for a char.

- Writing to memory that is in a read-only segment like .text.

- Using a memory allocation that has been freed, double-freeing a pointer, or freeing any pointer that wasn't returned from xmalloc or xcalloc.

In C0, memory errors would always predictably and consistently cause the program to stop executing. In C, this is *definitely not the case*.

Array accesses are not checked at all, and out-of-bounds memory references (whose result is formally undefined) may lead to unpredictable results. The program might stop with an error, or keep going, but after undefined behavior occurs it is difficult, if not impossible, to predict a program's behavior. For example, the code fragment

```
int main() {
  int* A = xmalloc(sizeof(int));
  A[0] = 0;    /* ok - A[0] is like *A */
  A[1] = 1;    /* error - not allocated */
  A[317] = 29; /* error - not allocated */
  A[-1] = 32;  /* error - not allocated(!) */
  printf("A[-1] = %d\n", A[-1]);
  return 0;
}
```

will not raise any compile time error or even warnings, even under the strictest settings. Here, the call to `xmalloc` allocates enough space for a single integer in memory. In this class, we are using `gcc` with all our standard flags:

```
gcc -Wall -Wextra -Werror -Wshadow -std=c99 -pedantic -g
```

The code above executes ok, and in fact prints 32, despite four blatant errors in the code.

To discover whether such errors may have occurred at runtime, we can use the `valgrind` tool.

```
% valgrind ./a.out
...
==nnnn== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
```

which produces useful error messages (elided above) and indeed, flags errors in code that didn't appear to have any errors when we ran it without valgrind.

`valgrind` slows down execution, but if at all feasible you should test all your C code in the manner to uncover memory problems. For best error messages, you should pass the `-g` flag to `gcc` which preserves some correlation between binary and source code.

You can also guard memory accesses with appropriate assert statements that abort the program when attempting out-of-bounds accesses.

There's an old joke that whenever you encounter undefined behavior, your computer could decide to play *Happy Birthday* or it could catch on fire. This is less of a joke considering recent events:

- In 2010, Alex Halderman's team at the University of Michigan successfully hacked into Washington D.C.'s prototype online voting system, and caused its web page to play the University of Michigan fight song, "The Victors."[2]

- The Stuxnet worm caused centrifuges, such as those used for uranium enrichment in Iran, to malfunction, physically damaging the devices.[3]

Not quite playing *Happy Birthday* and catching on fire, but close enough.

---

[2]Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. *Attacking the Washington, D.C. Internet Voting System.* Proceedings of the 16th Conference on Financial Cryptography and Data Security, February 2012.

[3]Holger Stark. *Stuxnet Virus Opens New Era of Cyber War.* Spiegel Online, August 8, 2011.

4   Address-of

In our C0 and C memory model, almost *everything* has an address. If `e` is an expression (like `x`, `A[12]`, `*x`, `A.fld`, or `A->fld`) that describes a memory location which we can read from and potentially write to, then that memory location exists in memory as some number of bytes with an address. Writing `&e` then gives us a *pointer* to that memory location. In C0, if we have a struct containing a string and an integer, it's not possible to get a pointer to *just* the integer. This is possible in C:

```
struct wcount {
  char *word;
  int count;
};

void increment(int *p) {
  REQUIRES(p != NULL);
  *p = *p + 1;
}

void increment_count(struct wcount *wc) {
  REQUIRES(wc != NULL);
  increment(&(wc->count));
}
```

Because the type of `wc->count` is `int`, the expression `&(wc->count)` is a pointer to an `int`. Calling `increment_count(B)` on a non-null struct will cause the `count` field of the struct to be incremented by the `increment` function, which is passed a pointer to the second field of the struct.

Because of the address-of operation, we never have to actually use pointer arithmetic if we don't want to. `A + 3` is always equivalent to `&A[3]`.

5   Stack Allocation

In C, we can also allocate data on the *system stack* (which is different from the explicit stack data structure used in the running example). As discussed in the lecture on memory layout, each function allocates memory in its so-called *stack frame* for local variables. We can obtain a pointer to this memory using the address-of operator. For example:

```
int main() {
```

```
    int a1 = 1;
    int a2 = 2;
    increment(&a1);
    increment(&a2);
    ...
}
```

Note that there is no call to `xmalloc` or `xcalloc` which allocate spaces on the system heap (again, this is different from the heap data structure we used for priority queues).

We can only free memory allocated with `xmalloc` or `xcalloc`, but not memory that is on the system stack. Such memory will automatically be freed when the function whose frame it belongs to returns. This has two important consequences. The first is that the following is a bug, because `free` will try to free the memory holding a_1, which is not on the heap:

```
int main() {
  int a1 = 1;
  int a2 = 2;
  free(a1);
  ...
}
```

The second consequence is pointers to data stored on the system stack do not survive the function's return. For example, the following is a bug:

```
int *f_ohno() {
  int a = 1; /* bug: a is deallocated when f_ohno() returns */
  return &a;
}
```

A correct implementation requires us to allocate on the system heap, using a call to `malloc` or `calloc` (or one of the library functions which calls them in turn).

```
int *f() {
  int* x = xmalloc(sizeof(int));
  *x = 1;
  return x;
}
```

When stack allocation is possible, it can be a real benefit, because it saves you from having to remember to free memory explicitly. However, if

the data structure we allocate needs to survive past the end of the current function you *must* allocate it on the heap.