Lecture Notes on Types in C

15-122: Principles of Imperative Computation
Frank Pfenning, Rob Simmons

Lecture 19
June 17, 2014

1 Introduction

In lecture 18, we emphasized the things we *lost* by going to C:

- Many operations that would safely cause an error in C0, like dereferencing `NULL` or reading outside the bounds of an array, are *undefined* in C – we cannot predict or reason about what happens when we have undefined behaviors.
- It is not possible to capture or check the length of C arrays.
- In C, pointers and arrays are the same – and we declare them like pointers, writing `int *i`.
- The C0 types `string`, `char*` and `char[]` are all represented as pointers to `char` in C.
- C is not garbage collected, so we have to explicitly say when we expect memory to be freed, which can easily lead to memory leaks.

In this lecture, we will endeavor to look on the bright side and look at the new things that C gives us. But remember: with great power comes great responsibility.

This lecture has three parts. First, we will continue our discussion of memory management in C: everything has an address and we can use the address-of operation `&e` to obtain this address. Second, we will look at the different ways that C represents numbers and the general, though mostly

implementation-defined, properties of these numbers that we frequently count on. Third, we will look at the type `void*` and how it can be used to implement generic data structures.

2 Address-of

In C0, we can only obtain new pointers and arrays with the built-in `alloc` and `alloc_array` operations. As we discussed last time, `alloc(ty)` in C0 roughly translates to `malloc(sizeof(ty))` in C, with the exception that C does not initialize allocated memory to default values. Similarly, the C0 code `alloc_array(ty, n)` roughly translates to `calloc(n, sizeof(ty))`, and `calloc` does initialize allocated memory to default values. Because both of these operations can return `NULL`, we also introduced `xmalloc` and `xcalloc` that allow us to safely assume a non-`NULL` result.

C also gives us a new way to create pointers. If `e` is an expression (like `x`, `A[12]`, or `*x`) that describes a memory location which we can read from and potentially write to, then the expression `&e` 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.

3 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 `malloc` or `calloc` which allocate spaces on the system heap (again, this is different from the heap data structure we used for priority queues).

Note that we can only free memory allocated with `malloc` or `calloc`, 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;
}
```

In general, stack allocation is more efficient than heap allocation, because it is freed automatically when the function in which it is defined returns. That removes the overhead of managing the 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.

4 Pointer Arithmetic in C

We have already discussed that C does not distinguish between pointers and arrays; essentially a pointer holds a memory address which may be the beginning of an array. In C we can actually calculate with memory addresses. Before we explain how, please heed our recommendation:

Do not perform arithmetic on pointers!

Code with explicit pointer arithmetic will generally be harder to read and is more error-prone than using the usual array access notation $A[i]$.

Now that you have been warned, here is how it works. We can add an integer to a pointer in order to obtain a new address. In our running example, we can allocate an array and then push pointers to the first, second, and third elements in the array onto a stack.

```
int *A = xcalloc(3, sizeof(int));
A[0] = 0; A[1] = 1; A[2] = 2;
increment(A);   /* A[0] now equals 1 */
increment(A+1); /* A[1] now equals 2 */
increment(A+2); /* A[2] now equals 3 */
```

The actual address denoted by $A + 1$ depends on the size of the elements stored at $*A$, in this case, the size of an int. A much better way to achieve the same effect is

```
int *A = xcalloc(3, sizeof(int));
A[0] = 0; A[1] = 1; A[2] = 2;
increment(&A[0]); /* A[0] now equals 1 */
increment(&A[1]); /* A[1] now equals 2 */
increment(&A[2]); /* A[2] now equals 3 */
```

We cannot free array elements individually, even though they are located on the heap. The rule is that we can apply `free` only to pointers returned from `malloc` or `calloc`. So in the example code we can only free A.

```
int* A = xcalloc(3, sizeof(int));
A[0] = 0; A[1] = 1; A[2] = 2;
free(&A[2]); /* bug: cannot free A[1] or A[2] separately */
```

The correct way to free this is as follows.

```
int* A = xcalloc(3, sizeof(int));
A[0] = 0; A[1] = 1; A[2] = 2;
free(A);
```

5 Numbers in C

In addition to the undefined behavior resulting from bad memory access (dereferencing a `NULL` pointer or reading outside of an array), there is undefined behavior in C. In particular:

- Division by zero is undefined. (In C0, this always causes an exception.)
- Shifting left or right by negative numbers or by a too-large number is undefined. (In C0, this always causes an exception.)
- Arithmetic overflow for *signed* types like `int` is undefined. (In C0, this is defined as modular arithmetic.)

This has some strange effects. If `x` and `y` are signed integers, then the expressions `x < x+1` and `x/y == x/y` are either `true` or undefined (due to signed arithmetic or overflow, respectively). So the compiler is allowed to pretend that these expressions are just `true` all the time. The compiler is also allowed to behave the same way C0 does, returning `false` in the first case when `x` is the maximum integer and raising an exception in the second case when `y` is 0. The compiler is also free to check for signed integer overflow and division by zero and start playing Rick Astley's "Never Gonna Give You Up" if either occurs, though this is last option is unlikely in practice. Undefined behavior is unpredictable – it can and does change dramatically between different computers, different compilers, and even different versions of the same compiler.

The fact that signed integer arithmetic is undefined is particularly annoying. In situations where we expect integer overflow to occur, we need to use *unsigned* types: `unsigned int` instead of `int`. As an example, consider a simple function to compute Fibonacci numbers. There are even faster ways of doing this, but what we do here is to allocate an array on the stack, fill it with successive Fibonacci numbers, and finally return the desired value at the end.

```
unsigned int fib(int n) {
  REQUIRES(n >= 0);
  unsigned int A[n+2]; /* stack-allocated array A */
  A[0] = 0;
  A[1] = 1;
  for (int i = 0; i <= n-2; i++)
    A[i+2] = A[i] + A[i+1];
  return A[n];    /* deallocates A just before actual return */
}
```

In addition to `int`, which is a signed type, there are the signed types `short` and `long`, and `unsigned` versions of each of these types – `short` is smaller than `int` and `long` is bigger. The numeric type `char` is smaller than `short` and always takes up one byte. The maximum and minimum values of these numeric types can be found in the standard header file `limits.h`.

C, annoyingly, does not define whether `char` is signed or unsigned. A `signed char` is definitely signed, a `unsigned char` is unsigned. The type `char` can be either signed or unsigned – this is *implementation defined*.

(C also gives us floating point numbers, `float` and `double`, but we will not cover these in 122.)

6 Implementation-defined Behavior

It is often very difficult to say useful and precise things about the C programming language, because many of the features of C that we have to rely on in practice are not part of the C standard. Instead, they are things that the C standard leaves up to the implementation – implementation defined behaviors. Implementation defined behaviors make it quite difficult to write code on one computer that will compile and run on another computer, because on the other compiler may make completely different choices about implementation defined behavior.

The first example we have seen is that, while a `char` is always exactly one byte, we don't know whether it is signed or unsigned – whether it can represent integer values in the range $[128, 128)$ or integer values in the range $[0, 256)$. And it is even worse, because a byte can be more than 8 bits! If you really want to mean "8 bits," you should say *octet*.

In this class we going to rely on a number of implementation-defined behaviors. For example, you can always assume that bytes are 8 bits. When it is important to *not* rely on integer sizes being implementation-defined, it is possible to use the types defined in `stdint.h`, which defines signed and unsigned types of specific sizes. In the systems that you are going to use for programming, you can reasonably expect a common set of implementation-defined behaviors: `char` will be a 8-bit integer (maybe signed, maybe unsigned) and so on. This chart describes how these types line up:

C (signed)	`stdint.h` (signed)	`stdint.h` (unsigned)	C (unsigned)
`signed char`	`int8_t`	`uint8_t`	`unsigned char`
`short`	`int16_t`	`uint16_t`	`unsigned short`
`int`	`int32_t`	`uint32_t`	`unsigned int`
`long`	`int64_t`	`uint64_t`	`unsigned long`

However, please remember that we cannot count on this correspondence behavior in all C compilers!

There are two other crucial numerical types. The first, `size_t`, is the type used represent memory sizes and array indices. The `sizeof(ty)` operation in C actually returns just the size of a type in bytes, so `malloc` and `xmalloc` actually take one argument of type `size_t` and `calloc` and `xcalloc` take two arguments of type `size_t`. In the early decades of the 21st century, we're still used to finding both 32-bit and 64-bit machines and programs; `size_t` will usually be the same as `uint32_t` in a 32-bit system and the same as `uint64_t` in a 64-bit system. The same goes for `uintptr_t`, an integer type used to represent pointers. Everything in C has an address, every address can be turned into a pointer with the address-of operation, and every address is ultimately representable as a number. To make sense of what it means to store a pointer in a integer type, we're going to need to introduce a new topic, *casting*.

7 Casting Between Numeric Types

If we have the hexadecimal value `0xF0` – the series of bits 11110000 – stored in an unsigned `char`, and we want to turn that value into an `int`. (This is

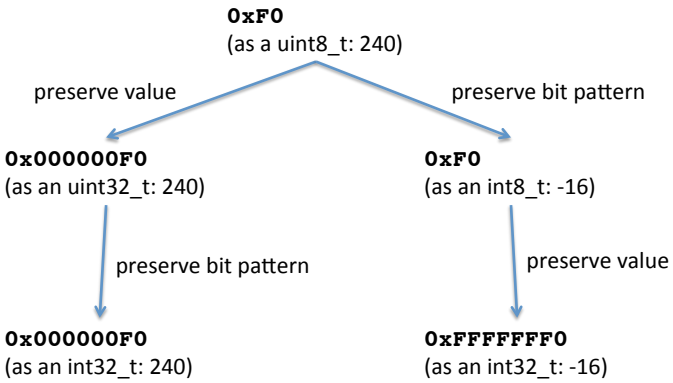
a problem you will actually encounter later in this semester.) We can cast this character value to an integer value buy writing `(int)e`.

```
unsigned char c = 0xF0;
int i = (int)c;
```

However, what will the value of this integer be? You can run this code and find out on your own, but the important thing to realize is that it's not clear, because there are two different stories we can tell.

In the first story, we start by transforming the unsigned `char` into an unsigned `int`. When we cast from a small unsigned quantity to a large unsigned quantity, we can be sure that the *value* will be preserved. Because the bits 11110000 are understood as the unsigned integer 240, the unsigned `int` will also be 240, written in hexadecimal as `0x000000F0`. Then, when we cast from an unsigned `int` to a signed `int`, we can expect the bits to remain the same (though this is really implementation defined), and because the interpretation of signed integers is two's-complement (also implementation defined) the final value will be 240.

In the second story, we transform the unsigned `char` into the signed-char. Again, the implementation-defined behavior we expect is that we will interpret the result as a 8-bit signed two's-complement quantity, meaning that `0xF0` is understood as `-16`. Then, when we cast from the small signed quantity (`char`) to a large signed quantity (`int`), we know the quantity `-16` will be preserved, meaning that we will end up with a signed integer written in hexadecimal as `0xFFFFFFF0`.



The order in which we do these two steps matters! Therefore, if we want to be clear about what result we want, we should cast in smaller steps to be explicit about how we want our casts to work:

```
unsigned char c = 0xF0;
int i1 = (int)(unsigned int) c;
int i2 = (int)(char) c;
assert(i1 == 240);
assert(i2 == -16);
```

8 Void Pointers

In C, a special type `void*` denotes a pointer to a value of unknown type. For most pointers, the type of a pointer tells C how big it is. When you have a `char*`, it represents an address that points to one byte (or, equivalently, an array of one-byte objects). When you have a `int*`, it represents an address that points to four bytes (assuming the implementation defines 4-byte integers), so when C dereferences this pointer it will read or write to four bytes at a time. A `void*` is just an address; C does not know how to read or write from it. We can cast back and forth between void pointers to other pointers.

```
int x = 12;
int *y = xcalloc(1, sizeof(int));
int *z;
void *px = (void*)&x;
void *py = (void*)y;
z = (int*)px;
z = (int*)py;
```

Casting out of `void*` incorrectly is generally either undefined or implementation-defined. We can also cast between pointers and the `intptr_t` types that can contain them.

```
int x = 12;
int *y = xcalloc(1, sizeof(int));
int *z;
intptr_t ipx = (intptr_t)&x;
uintptr_t ipy = (uintptr_t)y;
z = (int*)ipx;
z = (int*)ipy;
```

Thus, we don't strictly need the `void*` type – we could always use `uintptr_t` – but it is helpful to use the C type system to help us avoid accidentally, say adding two pointers together.

The return type of `xmalloc` and company is actually a void pointer.

```
void *xcalloc(size_t nobj, size_t size);
void *xmalloc(size_t size);
```

We have not shown explicit casts when we allocate, because C is willing to insert some casts for us. This is convenient when allocating memory, but in other situations it is a source of buggy code and does more harm than good. If we wanted to be explicit about the cast from `void*` to `int*`, we would write this:

```
int *px = (int*)xmalloc(sizeof(int));
```

As one last example, while this is implementation defined behavior, we can also store integers directly inside of a void pointer:

```
int x = 12;
void *px = (void*)(intptr_t)12;
int y = (int)(intptr_t)px;
```

This is a bit of an abuse – `px` does *not* contain a memory address, it contains the number 12 pretending to be an address – but this is a fairly common practice.

9 Simple Libraries

We can use void pointers to make data structures more generic. For example, an interface to generic stacks might be specified as

```
typedef struct stack* stack;
bool stack_empty(stack S);      /* O(1) */
stack stack_new();              /* O(1) */
void push(stack S, void* e);    /* O(1) */
void* pop(stack S);             /* O(1) */
void* stack_free(stack S);      /* S must be empty! */
```

Notice the use of `void*` for the first argument to `push` and for the return type of `pop`.

```
stack S = stack_new();
struct wcount *wc = malloc(sizeof(struct wcount));
wc->name = "wherefore"
wc->count = 3;
push(S, wc);
```

```
wc = malloc(sizeof(struct wcount));
wc->name = "henceforth"
wc->count = 5;
push(S, wc);

while(!stack_empty(S)) {
  wc = (struct wcount*)pop(S);
  printf("Popped %s with count %d\n", wc->name, wc->count);
  free(wc);
}
```

Because we can squeeze integers into a `void*`, we can also use the generic stacks to store integers:

```
stack S = stack_new();
push(S, (void*)(intptr_t)6);
push(S, (void*)(intptr_t)12);

while(!stack_empty(S)) {
  printf("Popped: %d\n", (int)(intptr_t)pop(S));
}
```

Translating stacks from C0 to C and making them generic is no different than translating BSTs. In fact, we no longer need stacks to know about the client interface, because rather than having one specific element, we have a generic element. The trade-off is that we no longer know how we are supposed to free a generic element when we free a stack. As the previous example shows, the elements stored as void pointers might not even be pointers!

The easy way out is to require that `stack_free` only be called on empty stacks, which means there are no elements that we have to consider freeing. This makes the implementation of `stack_free` simple:

```
void stack_free(stack S) {
  REQUIRES(is_stack(S) && stack_empty(S));
  ASSERT(S->top == S->bottom);
  free(S->top);
  free(S);
}
```

In the next C lecture, we will learn how to extend the stack implementation so that we can free non-empty stacks without leaks. This strategy

will also be necessary to make generic versions of more interesting data structures like BSTs, hash tables, and priority queues.