# Lecture 21 Notes
# Types in C

15-122: Principles of Imperative Computation (Spring 2016)
Frank Pfenning, Rob Simmons

## 1 Introduction

Previous lectures have 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 explore some of the new things that C gives us. But remember: with great power comes great responsibility. Today 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.

## 2 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 are other undefined behaviors 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 too-large a 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 overflow is undefined is particularly annoying. A check like `(x + 1 > x)`, which was a perfectly acceptable way to check that `x` was not `INT_MAX` in C, is now a check that the compiler is allowed to optimize to just `true`, because the result of this expression, in C, is either `true` or undefined.

There are two ways of coping with signed integer overflow being undefined. One option is to use *unsigned* types, which are required to obey the laws of modular arithmetic: **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.

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

There's another solution, particular to the compiler, `gcc`, that we usually use to compile C programs. This compiler (as well as other modern C compilers like `clang`), has a flag `-fwrapv`. When we compile with `-fwrapv`, then the compiler promises it will treat overflow from addition and multiplication as signed two's complement modular arithmetic, exactly like C0 does.

## 3 Implementation-defined Behavior

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*.

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 the other compiler may make completely different choices about implementation-defined behaviors.

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 are going to rely on a number of implementation-defined behaviors. For example, you can always assume that bytes are 8 bits on the computers we're using for this class in this decade. 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 the `<stdint.h>` types match up to the standard C types in most modern C compilers:

| **Signed** | | **Unsigned** | |
|---|---|---|---|
| C | `stdint.h` | `stdint.h` | C |
| **`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`. As we approach the third decade of the 21st century, we're increasingly using 64-bit systems and not dealing with 32-bit systems anymore. On 32-bit systems, `size_t` is usually 4-byte, 32-bit unsigned integer. We now usually expect `size_t` to be a 64-bit, 8-byte unsigned integer.

## 4 Casting Between Numeric Types

Now that we've introduced a bunch of different integer types, we need to see how to work with multiple integer types in the same program.

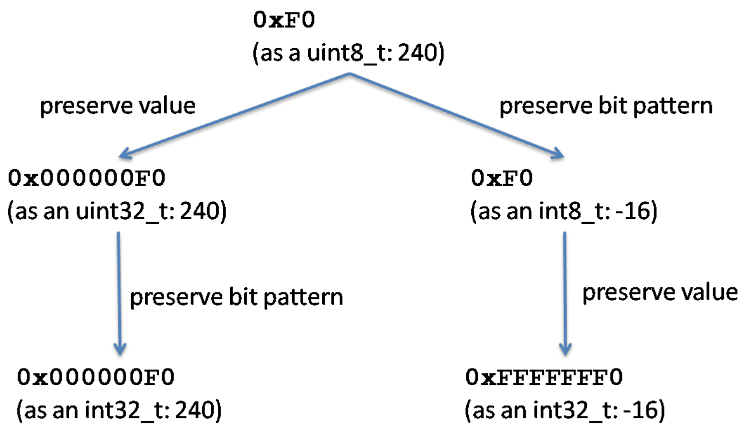
Imagine we have the hexadecimal value `0xF0` — represented as a sequence of bits as `11110000` — stored in an unsigned **`char`**, and we want to turn that value into an **`int`**. (This is a problem you will actually encounter later in this semester.) We can *cast* this character value to an integer value by 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 first transform the unsigned **`char`** into a 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`. In order to preserve the value as we go from a small to a large signed quantity, all we have to do is use *sign extension* — copy the high-order bit into all the new spaces.



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);
```

The C standard does define which of these two things will happen when you cast directly from **`unsigned char`** to **`int`**. However, for the purposes of this class, where we're trying to teach you enough C to write clear and correct programs, it's worth obeying the following rules:

- Never cast between signed and unsigned types of *different sizes*. Only cast between signed and unsigned types with the same size (implementation defined to preserve bits) and between small and large types that are either both signed or both unsigned.
- When you cast from a large signed (or unsigned) type to a small signed (or unsigned) type, make sure that the type you're casting to can represent the number. (So, for instance, you can cast the **`int`** 17 to an signed **`char`**, but don't cast the **`int`** 1000 to a signed **`char`**, because a signed **`char`** can only represent numbers between -128 and 127, inclusive.
- When you add, subtract, multiply, divide, compare, or do bitwise operations involving multiple variables, it's best to make sure that all the numbers you're working with have the same size and same signedness. One important "gotcha" here: if you just write the number `4`, it's treated as an **`int`** by default, so writing

  ```
  int64_t i = 1 << 40;
  ```

  will actually be undefined behavior, because `1` is (implementation-defined to be) a 32-bit quantity that can only be shifted by numbers between 0 and 31, inclusive. The fix, in this situation, is to write:

  ```
  int64_t i = 1;
  i = i << 40;
  ```

## 5 Other Types In C

C introduces a number of other types as well that we didn't have in C0. In particular, many C programs use **`enum`** types, **`union`** types, and the floating point types, **`float`** and **`double`**, which are used to represent fractional numbers like `0.25`. You'll learn more about these other C types in later courses, like 15-213.