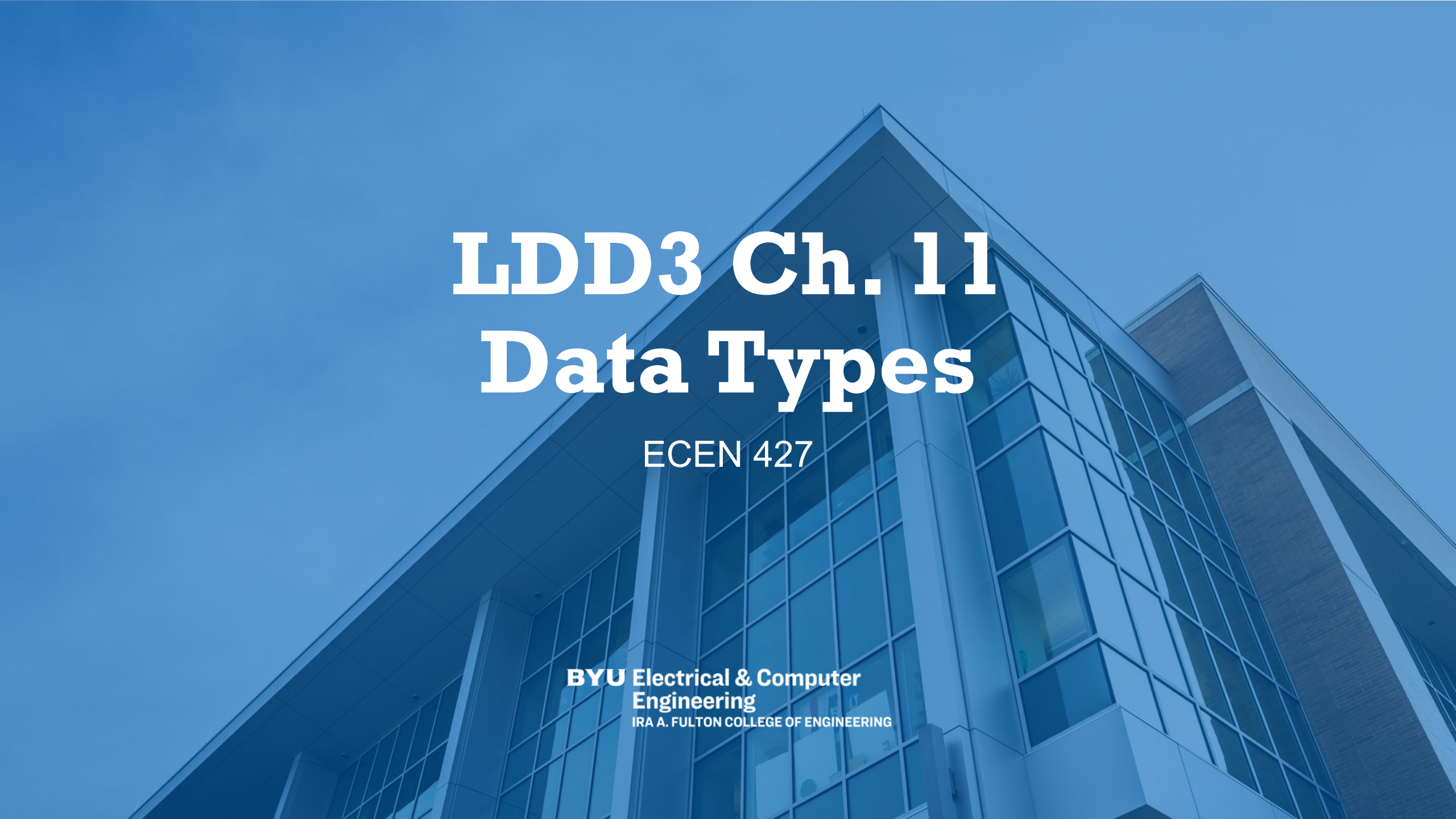




LDD3 Ch. 11

Data Types

ECEN 427

BYU Electrical & Computer
Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

- Data types used by kernel data are divided into three main classes:
 1. standard C types such as int,
 2. explicitly sized types such as u32, and
 3. types used for specific kernel objects, such as pid_t.

Explicit Size Types

- Sometimes kernel code requires data items of a specific size, perhaps to match predefined binary structures,* to communicate with user space, or to align data within structures by inserting “padding” fields
- The kernel offers the following data types to use whenever you need to know the size of your data. All the types are declared in `<asm/types.h>`, which, in turn, is included by **`<linux/types.h>`**:

```
u8;    /* unsigned byte (8 bits) */
u16;   /* unsigned word (16 bits) */
u32;   /* unsigned 32-bit value */
u64;   /* unsigned 64-bit value */
```

Physical Memory Addresses

Although, conceptually, addresses are pointers, memory administration is often better accomplished by using an unsigned integer type; the kernel treats physical memory like a huge array, and a memory address is just an index into the array.

Furthermore, a pointer is easily dereferenced; when dealing directly with physical memory addresses, you almost never want to dereference them in this manner. Using an integer type prevents this dereferencing, thus avoiding bugs.

`phys_addr_t`

<https://elixir.bootlin.com/linux/v4.5/source/include/linux/types.h#L162>

Interface Specific types

- Some of the commonly used data types in the kernel have their own typedef statements, thus preventing any portability problems. For example, a process identifier (pid) is usually pid_t instead of int.
- Note that, in recent times, relatively few new interface-specific types have been defined. Use of the typedef statement has gone out of favor among many kernel developers, who would rather see the real type information used directly in the code, rather than hidden behind a user-defined type. Many older interface-specific types remain in the kernel, however, and they will not be going away anytime soon.

Printing Interface Specific Types

- The main problem with `_t` data items is that when you need to print them, it's not always easy to choose the right `printf` or `printk` format, and warnings you resolve on one architecture reappear on another.
- For example, how would you print a `size_t`, that is unsigned long on some platforms and unsigned int on some others?
- Whenever you need to print some interface-specific data, the best way to do it is by **casting the value to the biggest possible type** (usually long or unsigned long) and then printing it through the corresponding format.

Pointer Return Types

- Many internal kernel functions return a pointer value to the caller. Many of those functions can also fail.
- In **most** cases, failure is indicated by returning a NULL pointer value.
 - This technique works, but it is unable to communicate the exact nature of the problem.
 - Some interfaces really need to return an actual error code so that the caller can make the right decision based on what actually went wrong.
- A number of kernel interfaces **(but not all)** return this information by **encoding the error code in a pointer value**.
 - Such functions must be used with care, since their return value cannot simply be compared against NULL.
 - To help in the creation and use of this sort of interface, a small set of functions has been made available (in <linux/err.h>).
 - The caller can use IS_ERR to test whether a returned pointer is an error code or not:
 - `long IS_ERR(const void *ptr);`
 - If you need the actual error code, it can be extracted with:
 - `long PTR_ERR(const void *ptr);`