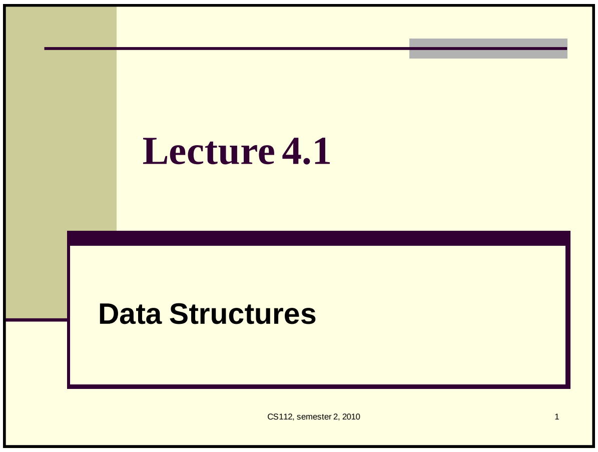
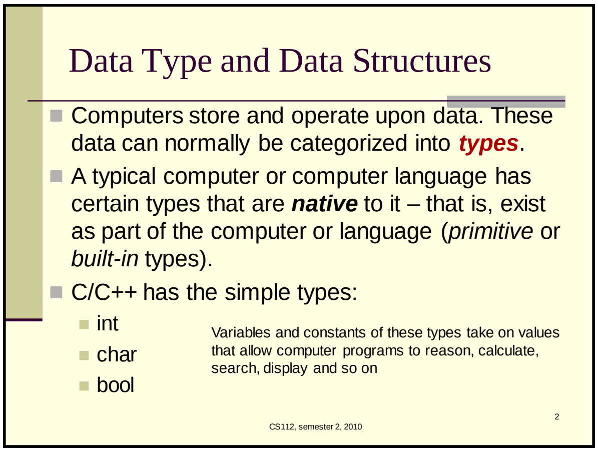


Lecture 4.1

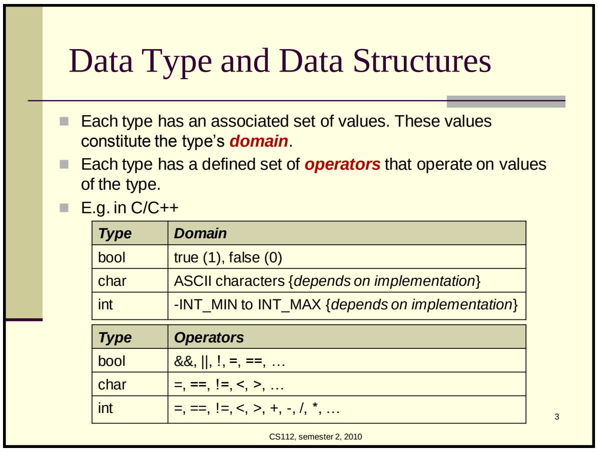
Data Structures



Data Type and Data Structures

- Computers store and operate upon data. These data can normally be categorized into ***types***.
- A typical computer or computer language has certain types that are ***native*** to it – that is, exist as part of the computer or language (*primitive* or *built-in* types).
- C/C++ has the simple types:
  - int
  - char
  - bool

Variables and constants of these types take on values that allow computer programs to reason, calculate, search, display and so on

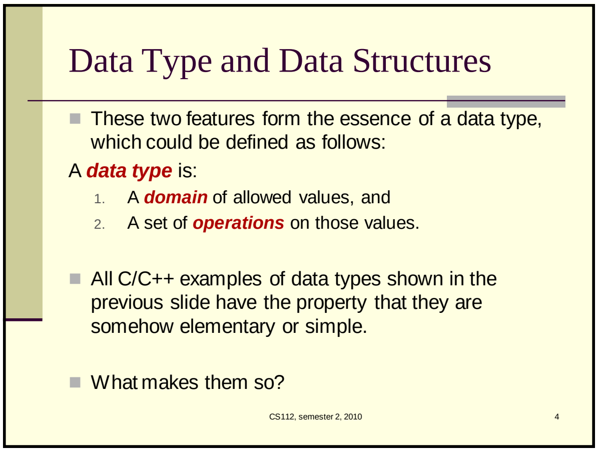


Data Type and Data Structures

- Each type has an associated set of values. These values constitute the type's ***domain***.
- Each type has a defined set of ***operators*** that operate on values of the type.
- E.g. in C/C++

Type	*Domain*
bool	true (1), false (0)
char	ASCII characters {*depends on implementation*}
int	-INT_MIN to INT_MAX {*depends on implementation*}

Type	*Operators*
bool	&&, ||, !, =, ==, …
char	=, ==, !=, <, >, …
int	=, ==, !=, <, >, +, -, /, *, …



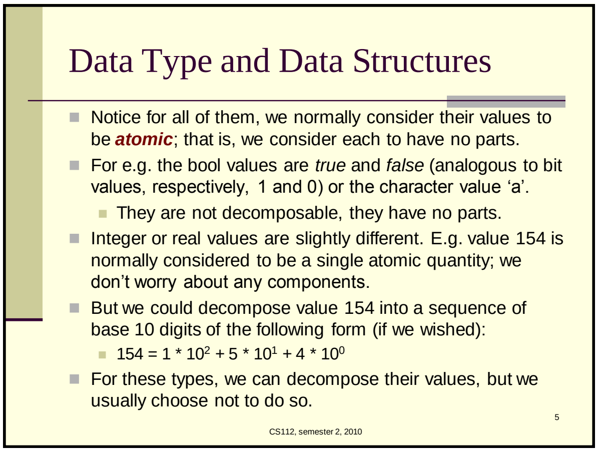
Data Type and Data Structures

- These two features form the essence of a data type, which could be defined as follows:

A ***data type*** is:

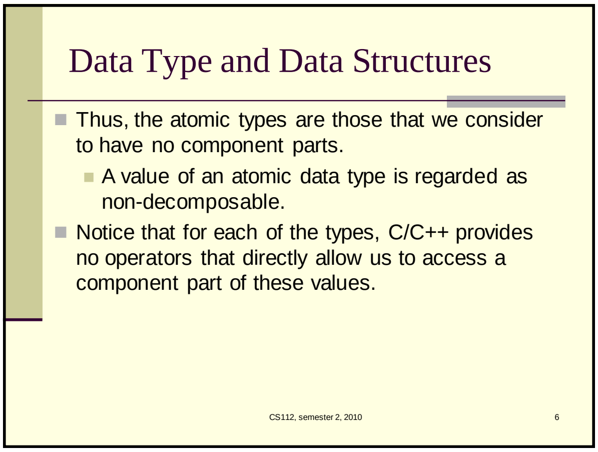
1. A ***domain*** of allowed values, and
2. A set of ***operations*** on those values.

- All C/C++ examples of data types shown in the previous slide have the property that they are somehow elementary or simple.
- What makes them so?



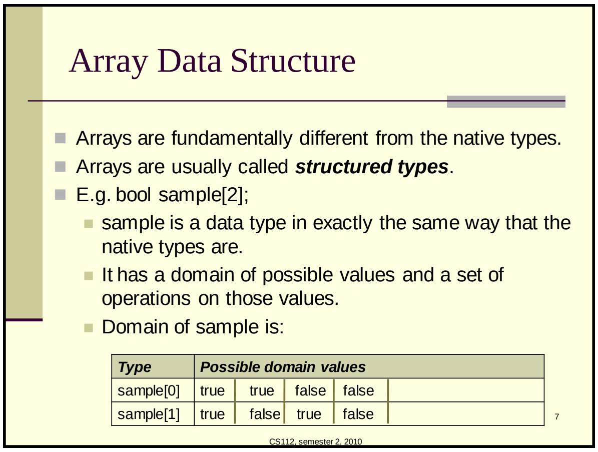
Data Type and Data Structures

- Notice for all of them, we normally consider their values to be ***atomic***; that is, we consider each to have no parts.
- For e.g. the bool values are *true* and *false* (analogous to bit values, respectively, 1 and 0) or the character value 'a'.
  - They are not decomposable, they have no parts.
- Integer or real values are slightly different. E.g. value 154 is normally considered to be a single atomic quantity; we don't worry about any components.
- But we could decompose value 154 into a sequence of base 10 digits of the following form (if we wished):
  - $154 = 1 * 10^2 + 5 * 10^1 + 4 * 10^0$
- For these types, we can decompose their values, but we usually choose not to do so.



Data Type and Data Structures

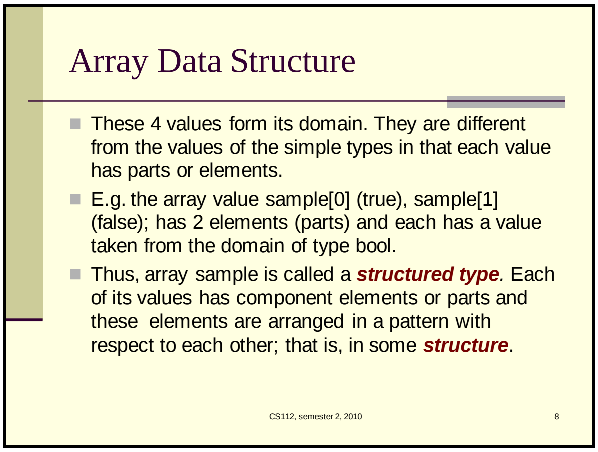
- Thus, the atomic types are those that we consider to have no component parts.
  - A value of an atomic data type is regarded as non-decomposable.
- Notice that for each of the types, C/C++ provides no operators that directly allow us to access a component part of these values.



Array Data Structure

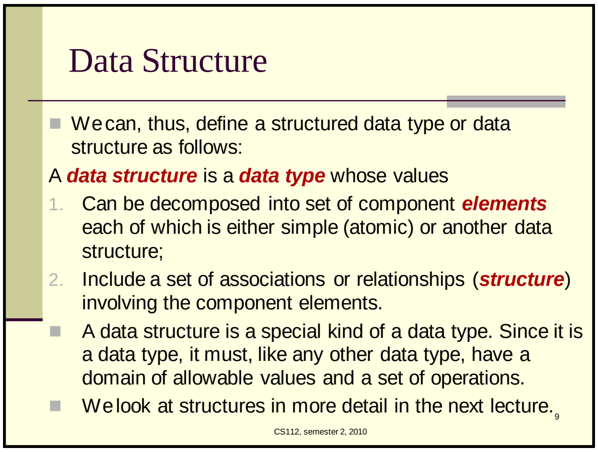
- Arrays are fundamentally different from the native types.
- Arrays are usually called ***structured types***.
- E.g. bool sample[2];
  - sample is a data type in exactly the same way that the native types are.
  - It has a domain of possible values and a set of operations on those values.
  - Domain of sample is:

Type	*Possible domain values*			
sample[0]	true	true	false	false
sample[1]	true	false	true	false



Array Data Structure

- These 4 values form its domain. They are different from the values of the simple types in that each value has parts or elements.
- E.g. the array value sample[0] (true), sample[1] (false); has 2 elements (parts) and each has a value taken from the domain of type bool.
- Thus, array sample is called a ***structured type****.* Each of its values has component elements or parts and these elements are arranged in a pattern with respect to each other; that is, in some ***structure***.



Data Structure

- We can, thus, define a structured data type or data structure as follows:

A ***data structure*** is a ***data type*** whose values

1. Can be decomposed into set of component ***elements*** each of which is either simple (atomic) or another data structure;
2. Include a set of associations or relationships (***structure***) involving the component elements.

- A data structure is a special kind of a data type. Since it is a data type, it must, like any other data type, have a domain of allowable values and a set of operations.
- We look at structures in more detail in the next lecture.