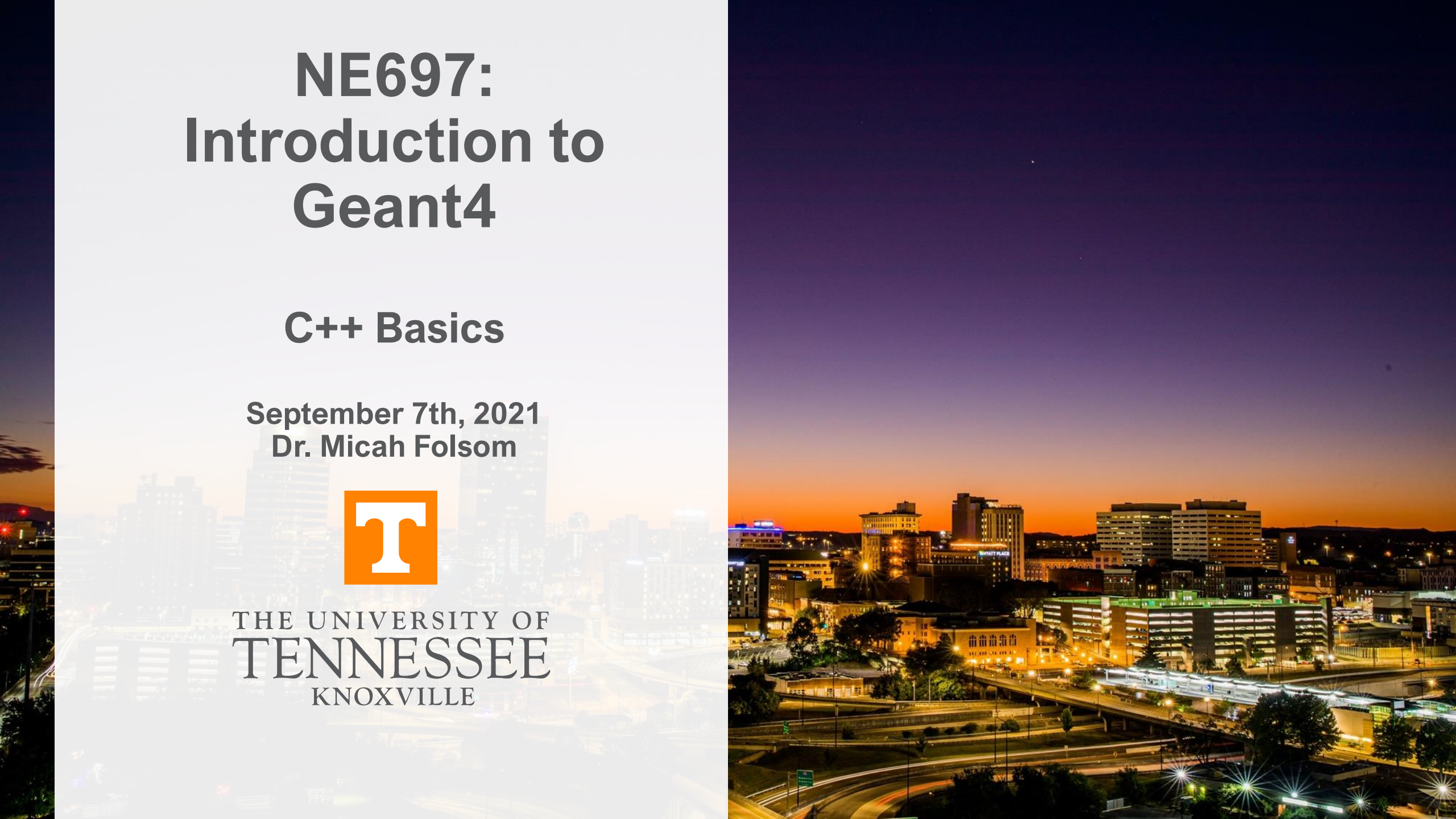
NE697: Introduction to Geant4

C++ Basics

September 7th, 2021

Dr. Micah Folsom



Today's Agenda

- Server and Canvas access for auditors – may have an option!

- Any other administrative items?

Assignment 2

- Demonstrate the C++ development workflow we've learned using our new tools
  - Make a new folder in your Github repo called **assignment2**
  - Create a CMakeLists.txt file
  - Write a simple program (1 source file) that takes the first command-line argument *N* and prints 2^N to standard out
    - Headers for: power function and converting text to numbers
  - Commit your code and push to Github
    - Add your build directory to .gitignore if the folder is in the repo

Arrays + Vectors

- [DEMO]
  - Instantiating each
  - Accessing elements
  - Iterating over the elements
    - Standard for-loop
    - Ranged-based for-loop
    - For-loop with iterators (pointer/C-style)
  - Element access ([] vs .at()), out-of-bounds access
  - **Multidimensional arrays**

Pointers

- The bane of every C++ student's existence. They're confusing!
- People write some extremely ugly and obtuse code abusing pointers and references
- **Pointer**: declare with *, e.g. **int* my_var;**
  - my_var is a "pointer to an int", stores the address as its value
  - Can be nullptr (== NULL == 0)
  - Must be **dereferenced** to get the value (***my_var**, see also [] and ->)
- **Reference**: declare with &, e.g. **int& my_var = another_var;**
  - my_var is a "reference to an int", also stores the address, but the value is the value
  - Cannot be nullptr!
  - Fixed once set, but no need to dereference
  - You will almost never declare these; you will "take the reference of" a variable
    - **int* my_var = &another_var;**

Pointers

- If you're nice to them, they'll be nice to you
  - Stay away from expressions that involve multiple (de)references/pointers (**, &&)
  - Always remember: they just store the address of the thing, instead of the value of the thing (the value is stored at that address!)
- [DEMO]
  - Declaring a pointer
  - Assigning/copying to other variables
  - Pointer = &thing

Memory

- C++ requires you to manage *some* of your own memory
- Your variables/objects are kept in 2 places: **stack** and **heap**
  - **Stack**: your local stash. Typically limited to ~8 MB
    - “local”: in this function
    - Where local variables go
    - Vanishes as soon as your function returns!
  - **Heap**: as much as the OS will give you
    - Use 1) when you have large objects, 2) need to pass it around efficiently (pointer), and/or 3) want/need to control its lifetime
    - Variables created with **new** (or smart pointers) will be on the heap
    - Don’t forget to **delete**!

Memory

- **new** returns the address of the memory it allocated
  - This is the **value** of a pointer. An address (also called a reference)
- Most often, you'll use new with objects so you can pass around their pointers
  - Efficient, because it's just the address instead of the whole object
- [DEMO]
  - Allocating a single int
  - Allocating an array of ints
  - What is the value of the variables (pointers)?
  - Take a peek at Geant4 example B1

Pointers and Memory

- Reminder that there are many different ways to do things
- Often times, it's a design choice for specific app/library
- You're stuck with what you're given
  - If the library uses raw pointers, it's hard to use smart pointers, vice versa
- In addition, Geant4 uses the **singleton** pattern frequently
  - One and only one instance of an object (think "manager" types)
  - Effectively a global variable
- My strategy: avoid pointers where possible
- The std:: collections have pointers underneath, so passing them around is cheap