NEW THINK TANK

C++ Tutorial 14

Posted by Derek Banas on May 6, 2018 in C Video Tutorial | 0 comments

In this tutorial I cover many topics many people have asked for. I show how to include outside files, Preprocessor Directives, Macro Constants, Macro Functions, Template Functions, Template Classes, and Iterators.

Like always all of the code follows the video below. It is heavily commented and it is recommended to print it out and take notes on it as you watch.

If you like videos like this consider donating $1, or simply turn off Ad Blocking software. Either helps me to continue making free tutorials.

Code from the Video

```
--- ANIMAL.H ---

// This guards against including this header in multiple
// files that make up the program along with #endif
#ifndef ANIMAL_H

// Read the following lines once
#define ANIMAL_H

#include <string>

class Animal {
public:
    Animal();
    Animal(const Animal& orig);
    virtual ~Animal();
    std::string name;
private:

};

#endif /* ANIMAL_H */

--- ANIMAL.CPP ---

#include "Animal.h"

Animal::Animal() {
}

Animal::Animal(const Animal& orig) {
}

Animal::~Animal() {
```

```
}

#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <ctime>
#include <numeric>
#include <cmath>
#include <sstream>

#include <deque>
#include <iterator>

// ----- PREPROCESSOR DIRECTIVES -----

// Anything that starts with a # is a preprocessor
// directive and they run before the program compiles

// Right click Header Files -> New -> C++ Header File
// and then include it here
#include "Animal.h"

// This is a macro constant that will replace
// PI with 3.14159 in the code before execution
#define PI 3.14159

// This is a macro function that will do the same with
// a function
#define AREA_CIRCLE(radius) (PI * (std::pow(radius, 2)))

// ----- END PREPROCESSOR DIRECTIVES -----

// ----- TEMPLATE FUNCTIONS -----

// We use templates to create functions or classes
// that can work with many types
// Templates differ from function overloading in that
// instead of having a function that does similar
// things with different objects a template does the
// same thing with different objects

// This says this is a function template that generates
// functions that except 1 parameter
template <typename T>
void Times2(T val){
    std::cout << val << " * 2 = " <<
            val * 2 << "\n";
}

// Receive multiple parameters and return a value
template <typename T>
T Add(T val, T val2){
    return val + val2;
}

// Work with chars and strings
template <typename T>
T Max(T val, T val2){
    return (val < val2) ? val2 : val;
}

// ----- END OF TEMPLATE FUNCTIONS -----

// ----- TEMPLATE CLASSES -----
```

```cpp

// Template classes are classes that can work with
// different data types

// You can define that you may receive parameters
// with different types, but they don't have to
// be different
template <typename T, typename U>
class Person{
public:
    T height;
    U weight;
    static int numOfPeople;
    Person(T h, U w){
        height = h, weight = w;
        numOfPeople++;
    }
    void GetData(){
        std::cout << "Height : " <<
            height <<
            " and Weight : " <<
            weight << "\n";
    }
};

// You have to initialize static class members
template<typename T, typename U> int Person<T, U>::numOfPeople;


// ----- END OF TEMPLATE CLASSES -----

int main()
{
    Animal spot = Animal();
    spot.name = "Spot";
    std::cout << "The Animal is named " <<
            spot.name << "\n";

    std::cout << "Circle Area : " <<
            AREA_CIRCLE(5) << "\n";

    // ----- TEMPLATE FUNCTIONS -----
    // The template function can receive ints or floats
    Times2(5);
    Times2(5.3);

    // Multiple parameters and returned value
    std::cout << "5 + 4 = " <<
            Add(5,4) << "\n";
    std::cout << "5.5 + 4.6 = " <<
            Add(5.5,4.6) << "\n";

    // Get biggest value
    std::cout << "Max 4 or 8 = " <<
            Max(4, 8) << "\n";
    std::cout << "Max A or B = " <<
            Max('A', 'B') << "\n";
    std::cout << "Max Dog or Cat = " <<
            Max("Dog", "Cat") << "\n";

    // ----- END OF TEMPLATE FUNCTIONS -----

    // ----- TEMPLATE CLASSES -----

    // When creating the object you must define the
```

```
    // data types used
    Person<double, int> mikeTyson (5.83, 216);
    mikeTyson.GetData();

    // You access static values using the object
    // and not the class
    std::cout << "Number of people : " <<
            mikeTyson.numOfPeople << "\n";

    // ----- END OF TEMPLATE CLASSES -----

    // ----- CONTAINERS -----
    // We have already seen the STL container vector
    // There are many other special ways of storing data

    // ----- DOUBLE ENDED QUEUE -----

    // A double ended queue (Deck) is a dynamic array that can
    // be expanded or contracted on both ends
    std::deque<int> nums = {1,2,3,4};
    nums.push_front(0);
    nums.push_back(5);
    for(int x: nums)
        std::cout << x << "\n";

    // You can access index values, but they are costly
    // because values aren't stored contigously, but
    // instead use multiple arrays
    std::cout << nums[0] << "\n";

    // ----- END DOUBLE ENDED QUEUE -----

    // ----- ITERATORS -----
    // Iterators are used to point at container
    // memory locations
    std::vector<int> nums2 = {1,2,3,4};

    // Define an iterator as the same type
    std::vector<int>::iterator itr;

    // Refer to the vectors begin and end while
    // incrementing the iterator
    for(itr = nums2.begin();
            itr < nums2.end();
            itr++){

        // Get value at the pointer
        std::cout << *itr << "\n";
    }

    // You can also increment a set number of spaces
    // Create an iterator and point it at the beginning
    // of the vector
    std::vector<int>::iterator itr2 = nums2.begin();

    // Advance 2 spaces
    advance(itr2, 2);
    std::cout << *itr2 << "\n";

    // Next works like advance, but it returns an
    // iterator
    auto itr3 = next(itr2, 1);
    std::cout << *itr3 << "\n";

    // Previous moves a set number of indexes and
```

```
    // returns an iterator
    auto itr4 = prev(itr2, 1);
    std::cout << *itr4 << "\n";

    // You can also insert at a defined index
    std::vector<int> nums3 = {1,4,5,6};
    std::vector<int> nums4 = {2,3};
    auto itr5 = nums3.begin();
    advance(itr5, 1);
    copy(nums4.begin(), nums4.end(),
            inserter(nums3, itr5));

    for(int &i: nums3)
        std::cout << i << "\n";

    // ----- END ITERATORS -----

    // ----- END OF CONTAINERS -----

    return 0;
}
```

Leave a Reply

Your email address will not be published.

Comment

Name

Email

Website

Submit Comment

Search

Search

Social Networks

Facebook
YouTube
Twitter
LinkedIn

Buy me a Cup of Coffee
"Donations help me to keep the site running. One dollar is greatly appreciated." - (Pay Pal Secured)

Archives

- November 2014
- October 2014
- September 2014
- August 2014
- July 2014
- June 2014
- May 2014
- April 2014
- March 2014
- February 2014
- January 2014
- December 2013
- November 2013
- October 2013
- September 2013
- August 2013
- July 2013
- June 2013
- May 2013
- April 2013
- March 2013
- February 2013
- January 2013
- December 2012
- November 2012
- October 2012
- September 2012
- August 2012
- July 2012
- June 2012
- May 2012
- April 2012
- March 2012
- February 2012
- January 2012
- December 2011
- November 2011
- October 2011
- September 2011
- August 2011
- July 2011
- June 2011
- May 2011
- April 2011
- March 2011
- February 2011
- January 2011
- December 2010
- November 2010
- October 2010
- September 2010
- August 2010
- July 2010
- June 2010

- May 2010
- April 2010
- March 2010
- February 2010
- January 2010
- December 2009

Powered by WordPress | Designed by Elegant Themes
About the Author Google+