Table of Contents

[First Classes 2](#_Toc181966687)

[Functions & Pointers 4](#_Toc181966688)

[Namespaces 6](#_Toc181966689)

[Strings 7](#_Toc181966690)

[Streams 9](#_Toc181966691)

[Statics 10](#_Toc181966692)

[Memory 11](#_Toc181966693)

[LinkedList 12](#_Toc181966694)

[Templates 13](#_Toc181966695)

[Templates20 14](#_Toc181966696)

[Operator Overloading 16](#_Toc181966697)

[Smart Pointers 19](#_Toc181966698)

[Move Semantics 20](#_Toc181966699)

[Inheritance 22](#_Toc181966700)

[Interfaces 26](#_Toc181966701)

[Exceptions 27](#_Toc181966702)

[Containers 28](#_Toc181966703)

[Functional Programming 29](#_Toc181966704)

[Functional Programming with Function Pointers 30](#_Toc181966705)

[Functional Programming with Interfaces 31](#_Toc181966706)

[Functional Programming with Functor 32](#_Toc181966707)

[Functional Programming with Lambdas 33](#_Toc181966708)

[ForEach with Lambdas 34](#_Toc181966709)

[Accumulate with Lambdas 35](#_Toc181966710)

[Threads 36](#_Toc181966711)

[Tasks 40](#_Toc181966712)

# First Classes

## Overview

In this lab, we will code our initial landscaping classes to satisfy the unit tests (Ignore *Color-Statics* test.)

|  |  |
| --- | --- |
| Exercise Folder | Landscaping |
| Builds On | none |
| Time to complete | 40 minutes |

Implement *Color*, *Circle*, and *Rectangle*

### Setup

The starter code for this lab is found in the *Landscaping* folder. If using VS Code, select *Landscaping* as the target run project and run it. 16 All of the tests will fail. This is because all of the functions in the skeleton classes throw exceptions.

### Steps

#### Color

1. Start with *Color.h*
2. Define an instance variable to hold rgb - unsigned rgb;
3. Implement the constructors and setters/getters
4. Run the test again - a little better?

##### Color Add

Color add. Colors can be added, this means that each color component is added. \* Don't let color component exceed 255 \* use the header **<algorithm>** and the **std::min** function to create a new color object that is the sum of the other 2

:::spoiler

#include <algorithm>  
 ...  
   
Color add(const Color &other) const  
{  
 return Color(  
 std::min(255u, Red() + other.Red()),  
 std::min(255u, Green() + other.Green()),  
 std::min(255u, Blue() + other.Blue()));  
}

:::

#### Circle

1. A circle has a name, color, & radius. Create instance variables for those fields
2. Start with the constructor - there are many ways to initialize - assign to the field, call the setter, or use the initializer list.
3. Implement the trivial setters and getters
4. For area and perimeter, you will need PI. There is a PI constant in the std library - we will use it later. For now, use:

constexpr double PI = 3.141592653589;

#### Rectangle

No surprises in *Rectangle*. Just implement it!

# Functions & Pointers

## Overview

Building on previous lab, we will had a *Landscape* class that holds up to 10 circles.

|  |  |
| --- | --- |
| Exercise Folder | FunctionsPointers |
| Builds On | Landscaping |
| Time to complete | 30 minutes |

### Landscape

The starter code for this lab is found in the *FunctionsPointers* project. Set this as your run project and run it. The specifications for the Landscape class are found in the failed tests:

[ FAILED ] 4 tests, listed below:  
[ FAILED ] LandscapeTest.Name  
[ FAILED ] LandscapeTest.Area  
[ FAILED ] LandscapeTest.Perimeter  
[ FAILED ] ColorTest.TestStatics

Implement Landscape.

Note - just use a fixed size array for circles (like 10). The instance variable can be declared as

const Circle\*\* circles;

In the constructor, allocate the array like this:

circles = new const Circle \* [10];

You will also need to keep track of how many circles have been added.

Since Landscape is holding **const** pointers, it will only be able to call **const** functions in *Circle*. Is *Circle* **const** correct?

:::spoiler *Landscape class*

class Landscape  
{  
public:  
 Landscape(const std::string &name)  
 {  
 this->name = name;  
 shapes = new const Circle \*[10];  
 count = 0;  
 }  
 std::string Name() const { return name; }  
 void SetName(const std::string &name)  
 {  
 this->name = name;  
 }  
 void Add(const Circle \*circle)  
 {  
 shapes[count++] = circle;  
 }  
 float Area() const  
 {  
 float total = 0;  
 for (int i = 0; i < count; i++)  
 total += shapes[i]->Area();  
 return total;  
 }  
 float Perimeter() const  
 {  
 float total = 0;  
 for (int i = 0; i < count; i++)  
 total += shapes[i]->Perimeter();  
 return total;  
 }  
  
private:  
 std::string name;  
 const Circle \*\*shapes;  
 int count;  
};

# Namespaces

## Overview

In this exercise, we will incorporate namespaces into our code.

|  |  |
| --- | --- |
| Exercise Folder | Namespaces |
| Builds On | FunctionsPointers |
| Time to complete | 15 minutes |

Note - if you did not complete **FunctionsPointers**, you can start with the **Namespaces** project.

Add namespace declarations to all code

training::geometry for *Circle & Rectangle*

training::utility for *Color*

training::landscape for *Landscape*

Test classes should be in a nested *test* namespace under the same namespace as the class under test

# Strings

## Overview

In this lab, we will examine the string class.

|  |  |
| --- | --- |
| Exercise Folder | Strings |
| Builds On | none |
| Time to complete | 15 minutes |

This is a stand-alone console project.

## Steps

1. Start with the code in the *Strings* folder
2. Modify main to include <string>
   * Add **using namespace std;**
   * Experiment with different string operations and declarations

For instance (explain the syntax of each line):

string hello("Hello World");  
string hello2("Hello World");  
if (hello == hello2)   
 cout << "they equal ==" << endl;  
auto great = "Great"s;  
hello += great;  
cout << hello << endl;

Change the project properties to choose C++ 20 standard

Try this:

auto res = hello <=> great;  
if (res < 0)   
 cout << hello << " is < than " << great << endl;  
else if (res > 0)   
 cout << hello << " is > than " << great << endl;  
else   
 cout << hello << " is equal to " << great << endl;

How would you convert a string to upper case?

:::spoiler

void toupper(string& str)  
{  
 for (auto& c : str)  
 c = toupper(c);  
}

:::

# Streams

## Overview

This lab provides a simple demonstration of file streams.

|  |  |
| --- | --- |
| Exercise Folder | Strings |
| Builds On | Strings |
| Time to complete | 15 minutes |

* Extending the string lab, open a file and write some strings to the file
* Prompt the user for a filename
* open the file and display its contents

# Statics

## Overview

Create some common colors as static members of the *Color* class.

|  |  |
| --- | --- |
| Exercise Folder | Namespaces |
| Builds On | Namespaces |
| Time to complete | 30 minutes |

### Steps

1. In *Color.h*, add some static members to the color class. For instance:

static Color RED;  
static Color GREEN;  
static Color BLUE;  
static Color WHITE;  
static Color BLACK;

The statics declarations don't provide storage for the colors, just a forward declaration. The storage must be allocated only once. The best place for this is the .cpp file

1. In *Color.cpp*, define the colors and initialize. :::spoiler

Color Color::RED(255, 0, 0);  
Color Color::GREEN(0, 255, 0);  
Color Color::BLUE(0, 0, 255);  
Color Color::WHITE(255, 255, 255);  
Color Color::BLACK(0, 0, 0);

:::

# Memory

## Overview

Modify Landscape to free memory allocated.

|  |  |
| --- | --- |
| Exercise Folder | Namespaces |
| Builds On | Namespaces |
| Time to complete | 15 minutes |

### Steps

1. Add a destructor ~Landscape() to the *Landscape* class.
2. Delete the memory that was allocated with **new** in the constructor.

* Remember, if an array was allocated use **delete[]** to free the memory.

# LinkedList

## Overview

This exercise is pointer crazy! Use the pictures from the slides to help understand the pointers.

|  |  |
| --- | --- |
| Exercise Folder | LinkedList |
| Builds On | Memory |
| Time to complete | 45 minutes |

Implement a linked list of circles to use with landscape

Use dog list as a starting point and convert to circle:

* Use the dog\_list files in this folder as a start.
* Use the debugger to step through the operations to understand how the linked list and pointers work.

# Templates

## Overview

Linked List is not really type specific. Any kind of pointer could be stored in the list. In this lab, we convert the circle linked list to a generic template. There is some tricky syntax to overcome with this demonstration.

|  |  |
| --- | --- |
| Exercise Folder | Templates |
| Builds On | LinkedList |
| Time to complete | 30 minutes |

### Steps

Note: if you are having problems completing the previous exercise, start with the code in the **Templates** folder

1. To begin, we will modify the declaration of the 3 linked list classes to make them templates.
2. Replace all occurances of **Circle** with a template parameter **T**. For example:

template <typename T>  
class ll\_node  
 private:  
 ll\_node \*next() { return this->right; }  
 const T \*item() { return data; }

1. Modify all three classes, converting them to templates and *lifting* Circle type to the template parameter.
2. Change the name of the linked list class to be more generic (ie *LinkedList*)

Note, the friend declarations are a little different:

template <typename> friend class linked\_list;

1. Update *Landscape* to use the template.

# Templates20

## Overview

In this lab, we build a stand-alone example of the new template syntax

|  |  |
| --- | --- |
| Exercise Folder | Template20 |
| Builds On | na |
| Time to complete | 30 minutes |

1. Open the starter project in *Template20*
2. Copy *Color.h* and *Color.cpp* into the project folder and include them in the project a. In VS code, you can drag with <ctrl> held down to copy files within the explorer panel b. *If Color.cpp contains any code, include it in CMakeList.txt*
3. Create a function named *add* that adds 2 numbers and returns the result (**same file as main**)

int add(int x, int y)  
{  
 cout << "int add" << endl;  
 return x + y;  
}

1. Lift the function to a template
2. In *main*, demonstrate that strings and numbers can be added using the template function
3. Try to add colors by passing color objects into the add template - note the error message
4. Add a constraint to the template requiring number types
5. Add another template function to add colors using *add* instead of +

:::spoiler Here's some examples:

#include <iostream>  
#include <string>  
#include <concepts>  
#include "Color.h"  
  
using namespace std;  
using namespace training::utility;  
  
int add(int x, int y)  
{  
 cout << "int add" << endl;  
 return x + y;  
}  
  
template <floating\_point T>  
T add(const T& x, const T& y)  
{  
 cout << "T add" << endl;  
 return x + y;  
}  
  
template <typename T>  
concept addable = requires (T o1, T o2) {  
 {o1.add(o2)} -> same\_as<T>;  
};  
  
auto add(const addable auto& x, const addable auto& y)  
{  
 cout << "auto add" << endl;  
 return x.add(y);  
}  
  
class foo { public: foo add(const foo& p2) const { return \*this; } };  
  
  
int main()  
{  
 auto result = add(10, 20);  
 auto fresult = add(3.14, 5.12);  
 Color c1;  
 Color c2(Color::RED);  
 auto cresult = add(c1, c2);  
 add(foo(), foo());  
}

:::

# Operator Overloading

## Overview

In this lab, we demonstrate operator overloading by adding selected operators to some of our classes.

|  |  |
| --- | --- |
| Exercise Folder | OperatorOverloading |
| Builds On | Templates |
| Time to complete | 45 minutes |

Note: if you did not complete the last exercise, you can start with the **OperatorOverloading** folder/project.

* Implement the following operators in **Color**:
* '''+''' to add colors together
* int conversion - rgb value
* << ostream operator

### Details

#### Step 1

1. Examine *ColorTest* and *CircleTest*
2. There are several tests marked with **DISABLED\_** with commented out code
3. When we run the tests, these will be counted as disabled.
4. Examine the last 3 tests in ColorTest.cpp - these are the requirements
5. In Color.h, add these member functions:  (note, the operations create new objects)
   * operator int()
   * Color operator + (const Color&) const
6. Add a global function after the class definition
   * inline ostream& operator << (ostream&, const Color&)
7. Implement the functions to satisfy the tests
8. Remove the **DISABLED\_** prefix from the tests

Google test ignores tests that are prefixed with **DISABLED\_**. The test is still logged so that we don't forget about it.

:::spoiler *Color.h*

// in Color class  
operator int() const { return Rgb(); }  
Color operator + (const Color& other) const  
{  
 return add(other);  
}  
friend std::ostream& operator << (std::ostream& out, const Color& color);

*Color.cpp*

std::ostream& operator << (std::ostream& out, const Color& color)  
{  
 out << "Color (" << std::hex << color.Red()  
 << ", " << color.Green()  
 << ", " << color.Blue() << ")" << std::dec;  
 return out;  
}

:::

#### Step 2

Implement the following operators in **Circle**

* (+=, ++) to create a new circle with new area.

Note - the radius from area is float radius = std::sqrt(area / PI);

<< ostream

1. Uncomment the tests in CircleTest.cpp
2. implement the required operators as member functions - addition produces new circles.
3. Implement <<; as a friend
4. Remove the **DISABLED\_** prefix from the tests

:::spoiler *Circle.h*

// in Circle class  
Circle operator + (const Circle& c2) const // c1 + c2  
{  
 float area = Area() + c2.Area();  
 float radius = std::sqrt(area / PI);  
 return Circle(radius, "Result", 0);  
}  
Circle& operator += (int value)  
{  
 SetRadius((float)std::sqrt((Area() + value) / PI));  
 return \*this;  
}  
Circle& operator ++() // func (++c)  
{  
 return operator +=(1); // (\*this) += 1;  
}  
void operator ++(int) // postfix func(c++);  
{  
 ++(\*this); // or this->operator++();  
}  
friend std::ostream& operator << (std::ostream& out, const Circle& c);  
  
// after circle class  
inline std::ostream& operator << (std::ostream& out, const Circle& c)  
{  
 out << "A circle with an area of " << c.Area()  
 << " and a radius of " << c.radius;  
 return out;  
}

:::

# Smart Pointers

## Overview

Let's use smart pointers and understand the impact!

|  |  |
| --- | --- |
| Exercise Folder | SmartPointers |
| Builds On | OperatorOverloading |
| Time to complete | 45 minutes |

In this lab, we will examine the impact of using unique\_ptrs and shared\_ptrs in the linked list

Note: if you did not complete the last exercise, you can start with the **SmartPointers** folder

### Unique

We only want to wrap pointers that need to be **delete**d eventurally. For the linked list, if we make *head* a smart pointer and make the *ll\_node.right* pointer smart, the destructors will delete the memory like dominoes falling!

#### Steps

In *LinkedList.h*

1. #include <memory>
2. Change *right* in *ll\_node* to a unique\_ptr<ll\_node>
3. *next()* will need to be modified as a result - what should you do?
4. Change *head* in *LinkedList* to a unique pointer
5. In *add*, in the initial case when *head* is being set, use **reset** to set the managed pointer
6. Remove the destructor
7. Prove that copy construction is not allowed even if *LinkedList* doesn't delete it's copy ctor.

*Challenge*

In the second example, we modify linked list to use shared pointers and we create aliases to simplify usage.

# Move Semantics

## Overview

Demonstration of string's move semantics.

|  |  |
| --- | --- |
| Exercise Folder | MoveSemantics |
| Builds On | None |
| Time to complete | 30 minutes |

* Add the following code, run and examine the results (note - **add #includes and usings as needed**). Use any file you wish as the infile

string getInfo()  
{  
 ifstream infile("anyfile");  
 string txt = string((istreambuf\_iterator<char>(infile)), istreambuf\_iterator<char>());  
 infile.close();  
 cout << "txt buffer loc: " << ((void\*)txt.c\_str()) << endl;  
 return txt;  
}  
  
void func(string data)  
{  
 cout << "data buffer loc: " << ((void\*)data.c\_str()) << endl;  
}  
  
int main()  
{  
 auto result = getInfo();  
 cout << "result buffer loc: " << ((void\*)result.c\_str()) << endl;  
 func(result);  
  
 func(getInfo());  
}

* Move the declaration of ***txt***in *getInfo()* to be a global
* Rerun the application
* Explain the changes

### Part 2

* Create a class (same .cpp file as above)
* Add a noisy copy constructor, move constructor, assignment operator, and move assignment operator
* Create a function that returns an instance of the class
* Create a function that accepts an instance of the class by value
* Call the functions from main()

# Inheritance

## Overview

We will implement an inheritance hierarchy for shapes.

|  |  |
| --- | --- |
| Exercise Folder | Inheritance |
| Builds On | SmartPointers |
| Time to complete | 30 minutes |

Note - if you did not complete the *SmartPointers* lab, you can start with the *Inheritance* project.

* Create a new class in the geometry namespace called *Shape*
* Modify *Circle* to inherit from *Shape*
* Promote members common to both *Rectangle* and *Circle* to *Shape*
* Modify *Rectangle* to inherit from *Shape*
* Re-run tests
* Modify *Landscape* to accept *Shapes* rather than Circles
* Modify *LandscapeTest* to add both Circle and Rectangle to the landscape

:::spoiler *Shape.h*

#pragma once  
#include <string>  
#include "Color.h"  
  
namespace training::geometry {  
 using namespace training::utility;  
  
 class Shape  
 {  
 public:  
 Shape(std::string name, Color c) : name(name), color(c) {}  
 std::string Name() const { return name; }  
 Color GetColor() const { return color; }  
 void SetName(std::string name)  
 {  
 this->name = name;  
 }  
 void SetColor(Color color)  
 {  
 this->color = color;  
 }  
  
 virtual float Area() const = 0;  
 virtual float Perimeter() const = 0;  
  
 private:  
 std::string name;  
 Color color;  
 };  
}

*Circle.h*

#pragma once  
#include <string>  
#include <numbers>  
#include <cmath>  
#include "Color.h"  
#include "Shape.h"  
  
constexpr double PI = std::numbers::pi;  
  
namespace training::geometry  
{  
 using namespace training::utility;  
 class Circle : public Shape  
 {  
 public:  
 Circle() : Circle(0, "", Color::BLACK) {}  
  
 Circle(float radius, std::string name, Color color)  
 :Shape(name, color), radius(radius)  
 {}  
  
 float Radius() const { return radius; }  
 void SetRadius(float radius)  
 {  
 this->radius = radius;  
 }  
  
 float Area() const override  
 {  
 return Radius() \* Radius() \* PI;  
 }  
 float Perimeter() const override  
 {  
 return 2 \* PI \* Radius();  
 }  
  
 Circle operator + (const Circle& c2) const // c1 + c2  
 {  
 float area = Area() + c2.Area();  
 float radius = (float)std::sqrt(area / PI);  
 return Circle(radius, "Result", 0);  
 }  
 Circle& operator += (int value)  
 {  
 SetRadius((float)std::sqrt((Area() + value) / PI));  
 return \*this;  
 }  
 Circle& operator ++()  
 {  
 return operator +=(1); // (\*this) += 1;  
 }  
 void operator ++(int) // postfix  
 {  
 ++(\*this); // or this->operator++();  
 }  
 friend std::ostream& operator << (std::ostream& out, const Circle& c);  
 private:  
 float radius;  
 };  
   
 inline std::ostream& operator << (std::ostream& out, const Circle& c)  
 {  
 out << "A circle with an area of " << c.Area()  
 << " and a radius of " << c.radius;  
 return out;  
 }  
}

*Landscape.h*

#pragma once  
#include <string>  
#include "Shape.h"  
#include "LinkedList.h"  
  
namespace training::landscape  
{  
 using namespace training::geometry;  
  
 class Landscape  
 {  
 public:  
 Landscape(const std::string& name)  
 {  
 this->name = name;  
 }  
 std::string Name() const { return name; }  
 void SetName(const std::string& name)  
 {  
 this->name = name;  
 }  
 void Add(const Shape\* shape)  
 {  
 shapes.add(shape);  
 }  
 float Area() const  
 {  
 float total = 0;  
 auto it = shapes.begin();  
 const Shape\* s;  
 while ((s = it.next()) != nullptr)  
 total += s->Area();  
 return total;  
 }  
 float Perimeter() const  
 {  
 float total = 0;  
 auto it = shapes.begin();  
 const Shape\* s;  
 while ((s = it.next()) != nullptr)  
 total += s->Perimeter();  
 return total;  
 }  
  
 private:  
 std::string name;  
 training::templates::LinkedList<Shape> shapes;  
 };  
  
}

:::

# Interfaces

## Overview

Create an interface to support Landscape named Measurable

|  |  |
| --- | --- |
| Exercise Folder | Interfaces |
| Builds On | Inheritance |
| Time to complete | 20 minutes |

### Steps

* Create a new file *Measurable.h*
* Create class *Measurable*
* Determine what methods are required by landscape - those are the methods required for *Measurable*
* Refactor *Landscape* to depend on *Measurable*
* Make *Measurable* the base class of *Shape*

Bonus - refactor LandscapeTest to use a fake object rather than circle and rectangle

Bonus Bonus - Make Landscape a *Composite Pattern*

*Need a* [*hint*](Interfaces//api/user/file/901)*?*

# Exceptions

## Overview

This lab adds validation to our project

|  |  |
| --- | --- |
| Exercise Folder | Exceptions |
| Builds On | Interfaces |
| Time to complete | 30 minutes |

### Step 1

Add the following to the test suites:

**ColorTest.cpp**

TEST\_F(ColorTest, InvalidRed) {  
 ASSERT\_THROW(c1.SetRed(-5), std::invalid\_argument);  
}  
TEST\_F(ColorTest, InvalidGreen){  
 ASSERT\_THROW(c1.SetGreen(-5), std::invalid\_argument);  
}  
TEST\_F(ColorTest, InvalidBlue){      
 ASSERT\_THROW(c1.SetBlue(270), std::invalid\_argument);  
}

**CircleTest.cpp**

TEST\_F(CircleTest, InvalidRadius){  
 ASSERT\_THROW(c1.SetRadius(-1), std::invalid\_argument);  
}

**RectangleTest.cpp**

TEST\_F(RectangleTest, SizeError) {   
 ASSERT\_THROW(r1.SetDimension(-1, -1), std::invalid\_argument);  
}

Run all tests and verify the new ones **fail**

### Step 2

Modify classes to get the tests to succeed.

# Containers

## Overview

Let's dump the linked list and switch to a standard container!

|  |  |
| --- | --- |
| Exercise Folder | Containers |
| Builds On | Exceptions |
| Time to complete | 15 minutes |

1. Modify *Landscape* to use either **vector** or **list**
2. Re-write *Area* and *Perimeter* to use a ranged for

# Functional Programming

## Overview

Get ready to explore functional programming with this starter project.

|  |  |
| --- | --- |
| Exercise Folder | MenuSystemStarter |
| Builds On | None |
| Time to complete | 15 minutes |

1. Review the code in *MenuSystemStarter*
2. Run the application

This will be the basis for the function programming examples.

# Functional Programming with Function Pointers

## Overview

Functional Programming with function pointers.

|  |  |
| --- | --- |
| Exercise Folder | MenuSystemFunctionPtr |
| Builds On | None |
| Time to complete | 15 minutes |

Create a function pointer callback for the menu activation

# Functional Programming with Interfaces

## Overview

Functional Programming with Interfaces.

|  |  |
| --- | --- |
| Exercise Folder | MenuSystemInterface |
| Builds On | None |
| Time to complete | 15 minutes |

Define a command interface to use for the callbacks

# Functional Programming with Functor

## Overview

Functional Programming with Functor.

|  |  |
| --- | --- |
| Exercise Folder | MenuSystemFunctor |
| Builds On | None |
| Time to complete | 15 minutes |

Modify the command interface to use an operator () to execute the command.

# Functional Programming with Lambdas

## Overview

Functional Programming with Lambda.

|  |  |
| --- | --- |
| Exercise Folder | MenuSystemLambda |
| Builds On | None |
| Time to complete | 30 minutes |

Delete the command interface and replace it with a function<>.

In main, demonstrate that callbacks can be functions, functors, and lambda expressions.

# ForEach with Lambdas

## Overview

Demonstrate for\_each and lambdas.

|  |  |
| --- | --- |
| Exercise Folder | ForEach |
| Builds On | Containers |
| Time to complete | 20 minutes |

*Landscape* uses a ranged for loop to calculate the total area and perimeter.

Modify these functions to use a for\_each loop instead.

Use a lambda expression to capture the running total

# Accumulate with Lambdas

## Overview

Demonstrate accumulate and lambdas.

|  |  |
| --- | --- |
| Exercise Folder | ForEach |
| Builds On | Containers |
| Time to complete | 15 minutes |

Modify the *Area* and *Perimeter* functions in *Landscape* to use **accumulate** instead of **for\_each**

Can you do it in one statement?

What's the difference with **reduce**?

:::spoiler

float Perimeter() const  
{  
 return std::accumulate(shapes.begin(), shapes.end(), 0.0f,  
 [](Accumulate/float total, const Measurable \*m) {  
 return total + m->Perimeter();  
 });  
}

:::

# Threads

## Overview

In this lab, we convert a synchronous function to a thread.

|  |  |
| --- | --- |
| Exercise Folder | Threads |
| Builds On | None |
| Time to complete | 20-40 minutes |

### Setup

Review the code in the *Threads* folder. In *main.cpp*, the function *Monitor* accepts a path to a directory. The function initially displays the contents of the directory and monitors for changes. The loop has no blocks, so it will be checking for changes at top speed, consuming CPU time and rapid IO.

Follow the steps below to convert the function to use threads.

### Steps - Monitor

1. So that we can signal when to stop, declare a global bool running = true; and change the while loop in *Monitor* to loop on running instead of true
2. At the bottom of the loop (inside), add a sleep for a second or two so the function isn't running wide open this\_thread::sleep\_for(1s);

### - Main

1. Allow for multiple directories from the command line, so define a container for multiple threads vector<thread> threads;
2. Loop through the arguments (1 through argc) creating thread objects to call *Monitor* with the path
3. Add the thread to the list
4. Prompt the user to press enter to end
5. Change running to false to stop the Monitor loop
6. Loop through the threads, joining them so that they have all completed before terminating main

\*\* Bonus - if the user provides a bad path, *Monitor* will throw an exception and cause the whole application to terminate. Add a try/catch so that *Monitor* doesn't crash the program. \*\*\* Bonus Bonus - create a mutex so that the threads aren't writing to **cout** simultaneously.

:::spoiler

#include <iostream>  
#include <thread>  
#include <mutex>  
#include <filesystem>  
#include <vector>  
  
using namespace std;  
  
mutex cout\_mutex;  
bool running = true;  
  
void Watch(string path) noexcept  
try  
{  
 filesystem::file\_time\_type lastTime;  
 while (running)  
 {  
 filesystem::file\_time\_type time = filesystem::last\_write\_time(path);  
 if (time != lastTime)  
 {  
 lock\_guard<mutex> lock(cout\_mutex);  
 cout << "File changed " << path << endl;  
 lastTime = time;  
 for (auto file : filesystem::directory\_iterator(path))  
 {  
 cout << file.path() << endl;  
 }  
 }  
 this\_thread::sleep\_for(1s);  
 }  
}  
catch (exception &e)  
{  
 cout << "Error: " << e.what() << endl;  
}  
  
int main(int argc, char \*argv[])  
{  
 cout << "Threads Project" << endl;  
 if (argc < 2)  
 {  
 cout << "Usage: " << argv[0] << " <path> ..." << endl;  
 return 1;  
 }  
 vector<thread> threads;  
 for (int i = 1; i < argc; i++)  
 {  
 string path = argv[i];  
 threads.push\_back(thread(Watch, path));  
 }  
  
 cout << "Press enter to exit" << endl;  
 cin.get();  
 running = false;  
 for (auto &thread : threads)  
 thread.join();  
 return 0;  
}

:::

### JThreads

A jthread (join thread) is a thread that has a join in the destructor. With the jthread, we don't need to join before letting the parent thread die. The thread's destructor will execute in the parent thread when it goes out of scope. The join will make sure the thread has completed.

The thread function can also have an optional first parameter of type stop\_token that can be used instead of the boolean running to control the loop. The threads destructor signals the token to stop. In the next pass of the loop, *Monitor* can stop looping and exit. The destructor can then complete.

Modify the Monitor solution to use *jthreads*

:::spoiler

#include <iostream>  
#include <thread>  
#include <mutex>  
#include <filesystem>  
#include <vector>  
  
using namespace std;  
  
mutex cout\_mutex;  
  
void Watch(stop\_token stop\_token, string path) noexcept  
try  
{  
 filesystem::file\_time\_type lastTime;  
 while (!stop\_token.stop\_requested())  
 {  
 filesystem::file\_time\_type time = filesystem::last\_write\_time(path);  
 if (time != lastTime)  
 {  
 lock\_guard<mutex> lock(cout\_mutex);  
 cout << "File changed " << path << endl;  
 lastTime = time;  
 for (auto file : filesystem::directory\_iterator(path))  
 {  
 cout << file.path() << endl;  
 }  
 }  
 this\_thread::sleep\_for(1s);  
 }  
}  
catch (exception &e)  
{  
 cout << "Error: " << e.what() << endl;  
}  
  
int main(int argc, char \*argv[])  
{  
 cout << "Threads Project" << endl;  
 if (argc < 2)  
 {  
 cout << "Usage: " << argv[0] << " <path> ..." << endl;  
 return 1;  
 }  
 vector<jthread> threads;  
 for (int i = 1; i < argc; i++)  
 {  
 string path = argv[i];  
 threads.push\_back(jthread(Watch, path));  
 }  
  
 cout << "Press enter to exit" << endl;  
 cin.get();  
  
 // no need to join - jthreads join on destructor  
 // for (auto &thread : threads)  
 // thread.join();  
 return 0;  
}

:::

# Tasks

## Overview

In this lab, we convert a synchronous function to a task.

|  |  |
| --- | --- |
| Exercise Folder | Tasks |
| Builds On | None |
| Time to complete | 20-40 minutes |

### Setup

Review the code in *main.cpp*.

The function *Convert* reads an input file and creates an output file of all uppercase letters. It returns the name of the file and the number of characters that were actually changed to uppercase.

*main* loops through the command line arguments, calling *Convert* for each file sequentially.

Change *main* to create tasks for each file and then wait for them to complete in parallel.

### Steps - main

1. Create a vector (or list) of future<ConvertResult>.
2. In the loop, call **async** to create the task on *ConvertFileUpper* and add the future to the vector
3. Remove the output statement showing the result
4. Create a second loop to loop through the vector of futures
5. Call **get()** to get the result of the task
6. Display the result of the conversion

:::spoiler

int main(int argc, char\* argv[])   
{  
 cout << "Tasks Project" << endl;  
 if (argc < 2)  
 {  
 cout << "Usage: " << argv[0] << " file [file] ..." << endl;  
 return 1;  
 }  
 vector<future<ConvertResult>> futures;  
 for\_each(argv + 1, argv + argc, [&futures](Tasks/const char\* fileName)  
 {  
 futures.push\_back(async(ConvertFileToUpper, fileName));  
 });  
 for (auto& f : futures)  
 {  
 auto result = f.get();  
 cout << "File " << result.first << " converted " << result.second << " characters" << endl;  
 }  
  
 return 0;  
}

:::