Modern C++

Programming

MODERN C++ IS OUR NEW BEETLE?

Modern C++ builds upon Classis C++, which is a reliable, quick, and well-documented language used by developers worldwide. The strength of Modern C++ are the well-known idioms of traditional C++ plus the additional features of C++ 11, C++ 14, and C++ 17.

MODERN C++

For more than thirty years, C++ has been a popular language for business, scientific, and entertainment applications. With the advent of newer languages, such as C# and Java, some of that popularity has eroded. C++ has the advantage of providing developers more control and less abstraction of critical features, such as memory management.

Modern C++ reclaims some of its popularity from adding features, such lambdas and threading, that are common now in other languages. Some of these features were available before only through the native API layer.

LOGISTICS

Class Hours:

- Start time is 8:30am
- End time is 4:30pm
- Class times may vary slightly for specific classes
- Breaks mid-morning and afternoon (10 minutes)

Lunch:

- Lunch is 11:45am to 1pm
- Yes, 1 hour and 15 minutes
- Extra time for email, phone calls, or simply a walk.

Telecommunication:

- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

Miscellaneous

- Courseware
- Bathroom
- Fire drills

BRADLEY NEEDHAM

> 20 years C++ experience

bradley.e.needham@gmail.com

ADVANCED C++

In addition to introducing Modern C++, this courseware is a deep dive of the C++ language. There are literally volumes of documentation, blogs, and articles on advanced concepts, features, and idioms related to C++. This course codifies the most useful and general purpose advanced features.

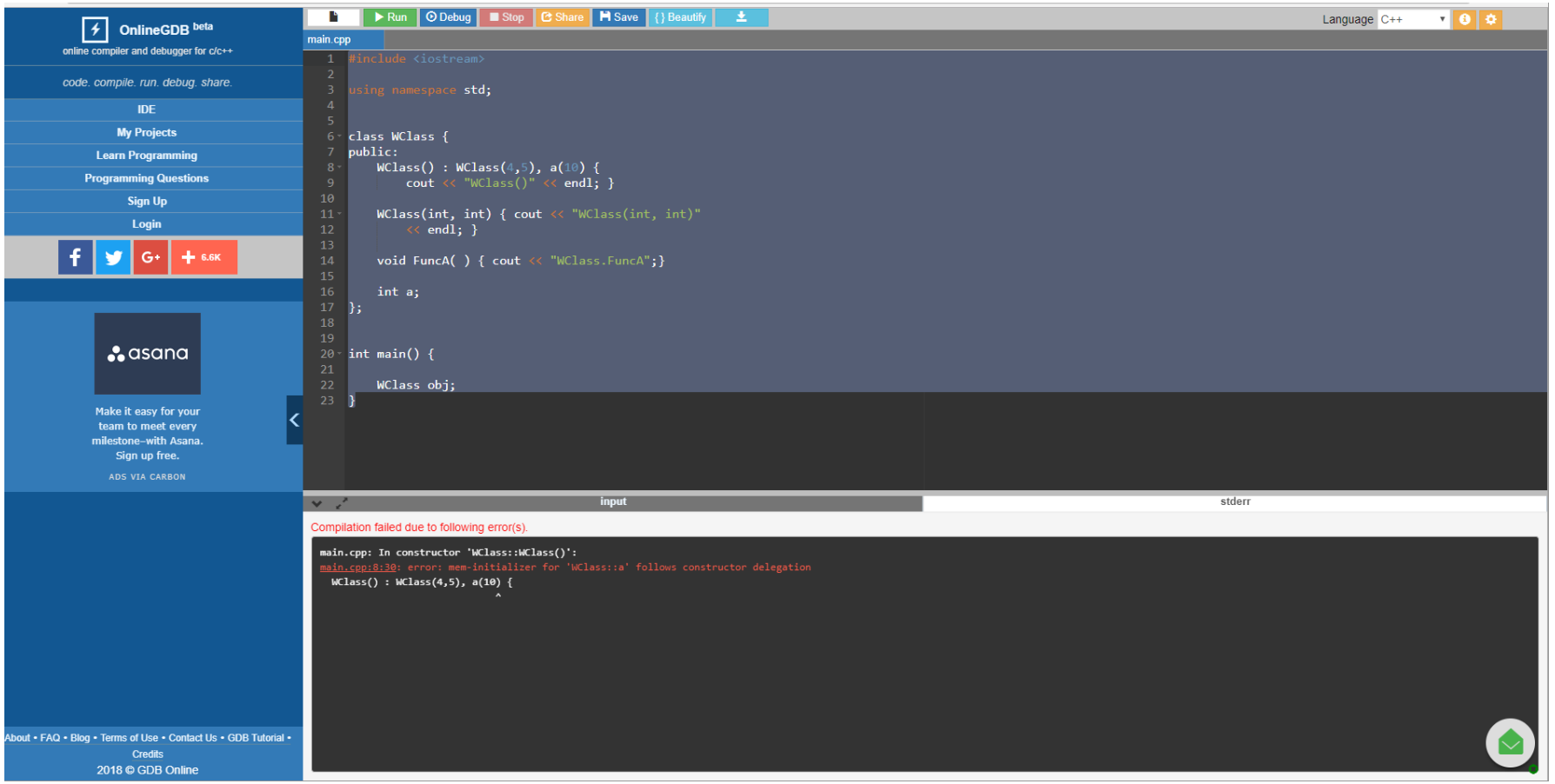
The goal is to provide the necessary skillset to write C++ code that maximizes the capabilities of the language – more efficient, better performance, improved portability, innovative, and scalable.

There is an expectation of 6-months experience but little more.

ADVANCED C++

There are several great online C++ compilers.

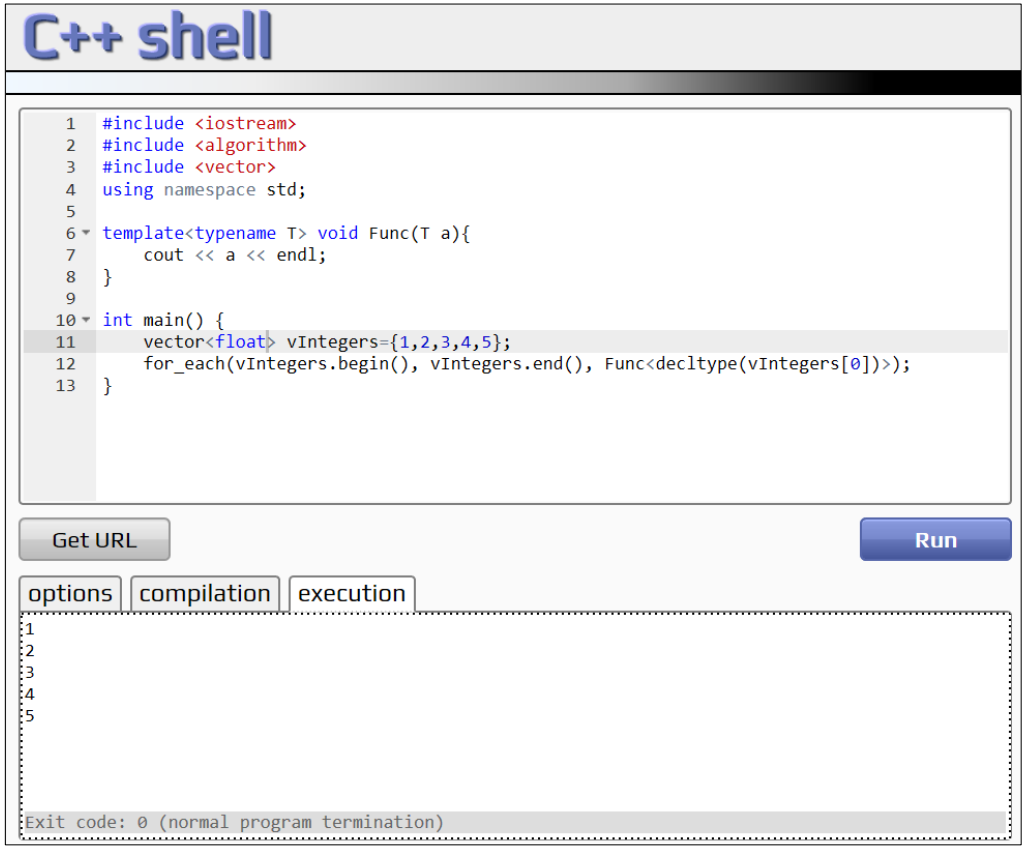
https://www.onlinegdb.com/online_c++_compiler



ADVANCED C++

Here is another:

http://www.cpp.sh/



GOAL

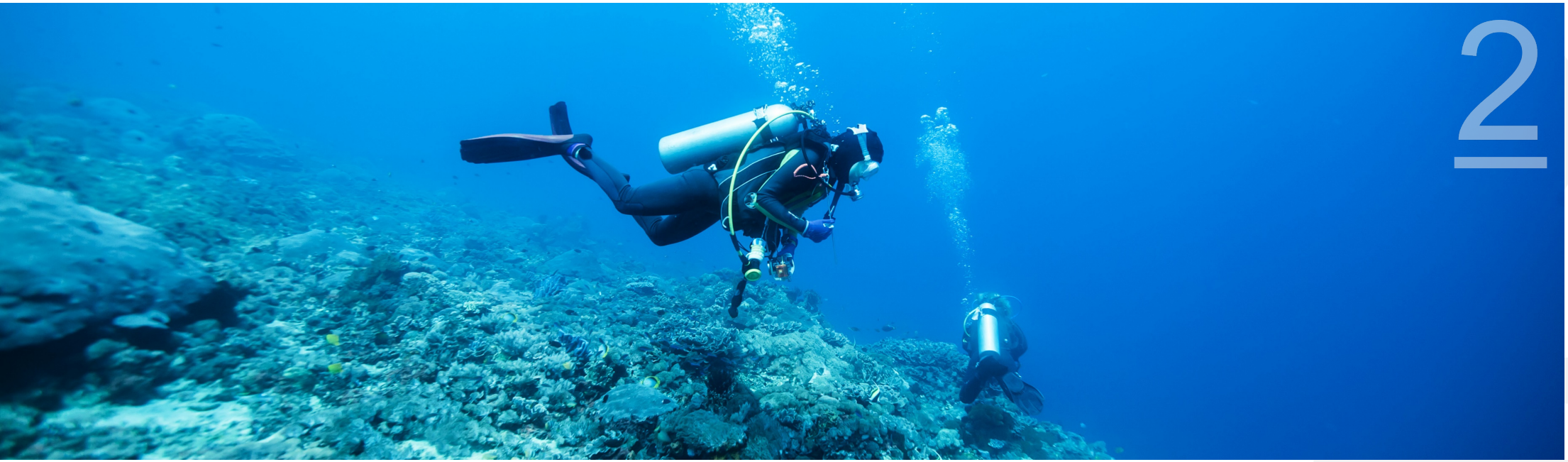
At the conclusion of this class, you will be able to apply the full repertoire of Modern C++ capabilities to effectively and creatively resolve real world problems.

- Return value optimization
- Uniform initialization
- Polymorphism
- Smart pointers
- Optimization
- Type inference
- Lambdas

- Threading
- Move semantics
- Exceptions
- Smart pointers
- Variadic templates
- Advanced construction

Ready, Set, Go

The Warmup



THE WARMUP

FACTORIAL

This code calculates a factorial value. In main, cout displays the result.

The Factorial function returns an integer with a short parameter. Factorial function prototype is declared at the beginning of the source file. The function is called recursively.

What is the significance of the static declaration?

```
#include <iostream>

int Factorial(short value);

int main()
{
    auto result = Factorial(6);

    std::cout << result;
}

int Factorial(short value) {
    static int answer=1, count;
    answer=answer*(++count);
    if (count == value) {
        return answer;
    }
    Factorial(value);
}
```

EMPLOYEE

Employee class inherits from Person class.

Person is an abstract class because of the Display function, which is a pure virtual method. Display is overridden and implemented in the Person class.

Employee class constructor (derived class) calls the Person constructor (base class) and provides the name parameter.

Why is Employee designated as a friend class?

```
#include <iostream>
#include <string>
using namespace std;

class Person {
public:
    friend class Employee;
    Person(string newname) {
        name = newname;
    }
    virtual void Display() = 0;
private:
    string name;
};

class Employee: public Person {
public:
    Employee(string name) : Person(name) {
    }
    void Display() {
        cout << name;
    }
};

int main() {
    Employee bob("Bob Young");
    bob.Display();
}
```

POP QUIZ: WHAT'S WRONG IN THIS CODE?

15 MINUTES

```
class XClass {
public:
    XClass() {
        ptr = new int;
        XClass(42);
    }
    XClass(int a) {
        value = a;
    }
    ~XClass() {
        delete ptr;
    }
    int value;
private:
    int* ptr;
};
class YClass {
public:
    YClass() {
        pObjs = new XClass[3];
    }
    ~YClass() {
        delete [] pObjs;
    }
    XClass* pObjs;
};
```

TWOVALUES

TwoValues class has two integer members. The class has operator overloading to add a float value. There is a two argument constructor to initialize TwoValues objects.

What is the specific purpose of the global operator+? Why is the second parameter both const and by reference?

```
#include <iostream>

class TwoValues {
public:
    friend  TwoValues operator+( float lvalue, const TwoValues& rvalue);
    TwoValues(int one, int two) {
        first = one;
        second = two;
    }
    TwoValues operator+(float value) {
        auto increment = int(value / 2);
        return TwoValues(first + increment, second + increment);
    }
private:
    int first, second;
};

TwoValues operator+(float lvalue,
    const TwoValues& rvalue) {
    auto increment = int(lvalue / 2);
    return TwoValues(rvalue.first + increment, rvalue.second + increment);
}

int main() {
    TwoValues obj1(10, 20);
    TwoValues obj2 = obj1 + 12.4;
}
```

CHESS GAME

ChessGame class implements the singleton pattern. An instance of the board can exist only once but is lazily initialized. This is possible because a pointer to board is contained in the class.

Notice that pBoard is static and initialized at file scope.

What is wrong with this code?

```
#include <iostream>

class Board {};

class ChessGame {
public:
    ChessGame() {
        if (pBoard) {
            std::cout << "Ending previous game";
            // Do something
        }
        else
            pBoard = new Board;
    }
    ~ChessGame() {
        delete pBoard;
    }
private:
    static Board* pBoard;
};

Board* ChessGame::pBoard = nullptr;

int main(){
    ChessGame game1;
    ChessGame game2;
}
```

STD::ARRAY

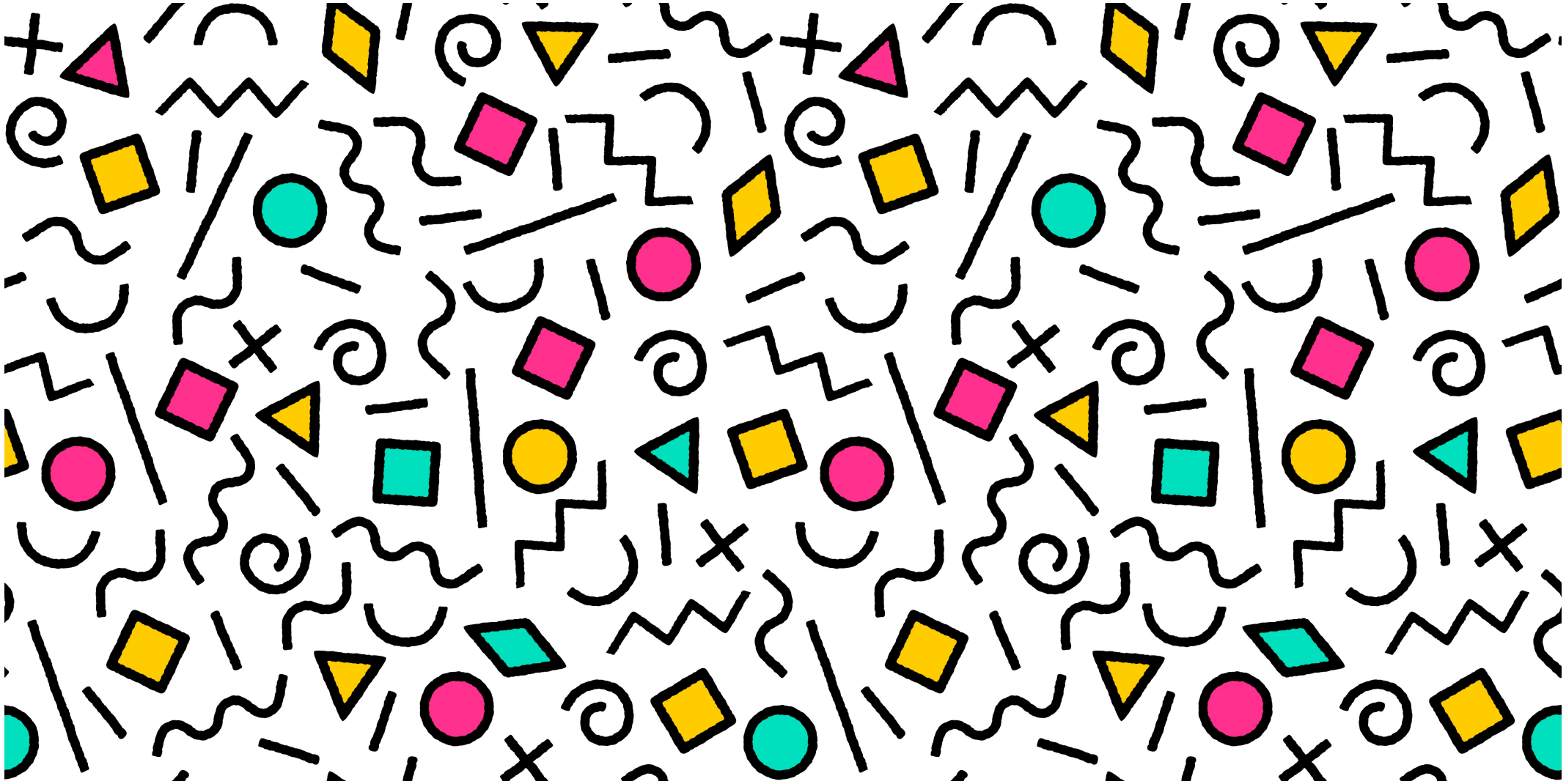
STL is a popular library and often leveraged in C++ code. Most STL artifacts are templated.

In this example, data is a std::array (container) of strings. The strings are listed both forward and backward.

What are the four categories of STL artifacts? For example, containers are one of the categories. How is each category used?

```
#include <array>
#include <iostream>
#include <string>
using namespace std;

int main() {

    array<string, 5> data = { "one", "two", "three",
        "four", "five"};

    cout << "Display forward" << endl;

    for (auto i : data)
        cout << i << ' ';

    cout << "\n\nDisplay reverse" << endl;
    for (auto i{ data.size() }; i-- > 0; )
        std::cout << data[i] << ' ';

    return 0;
}
```

SHAPES



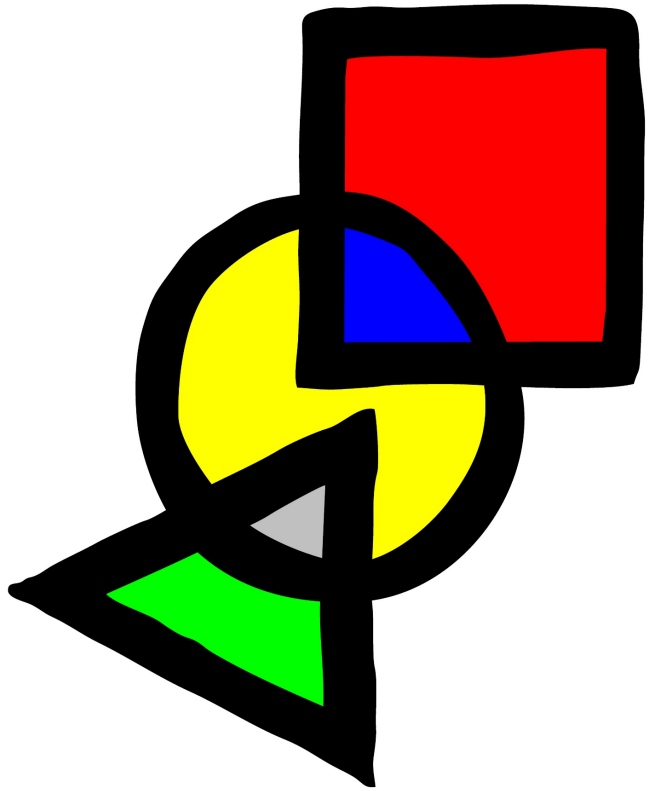
SHAPES

Geometric shapes is one of the most renowned scenarios used for object-oriented programming. In this scenario, Rectangle, Triangle, and Circle are related types derived from Shape, which is the base class.

Draw is a pure virtual method in the Shape class, which should be implemented in a derived class. Other methods, such as GetCenterPoint, are virtual and implemented in the Shape class.

There is a static count for tracking the number of Shape instances.

Each geometric shape has one or more data points. Each data point is stored in a vector as a tuple.



SHAPES - 2

```
typedef tuple<int, int> point;

class Shape {
   public:

   virtual void Draw() = 0;

   Shape() {
      ++count;
   }

   virtual void DrawPoints(bool forward) {
      for (std::vector<point>::iterator iter =
          dataPoints.begin(); iter != dataPoints.end(); ++iter) {
          cout << get<0>(*iter) << "  " << get<1>(*iter) << endl;
      }
   }
```

```
virtual point GetCenterPoint() {
   int totalx = 0;
   int totaly = 0;
   for (std::vector<point>::iterator iter =
      dataPoints.begin(); iter != dataPoints.end(); ++iter) {
      totalx += get<0>(*iter);
      totaly += get<1>(*iter);
   }
return(point(totalx, totaly));
}
static int GetCount() {
   return count;
}

protected:

   string shapeType;
   vector<point> dataPoints;

   private:
   static int count;

};

int Shape::count = 0;
```

TRIANGLE / RECTANGLE CLASS

Here are the Triangle and Rectangle classes that are derived from the Shape class.

Notice that the Draw method is implemented – overriding Draw of the base class.

```
class Triangle : public Shape {
public:

    Triangle(point p1, point p2, point p3) {
        dataPoints.push_back(p1);
        dataPoints.push_back(p2);
        dataPoints.push_back(p3);
    }
    void Draw() {
        cout << "Drawing Triangle" << endl;
    }
};

class Rectangle : public Shape {
public:

    Rectangle(point p1, point p2, point p3, point p4) {
        dataPoints.push_back(p1);
        dataPoints.push_back(p2);
        dataPoints.push_back(p3);
        dataPoints.push_back(p4);
    }
    void Draw() {
        cout << "Drawing Rectangle" << endl;
    }
};
```

LAB: IMPLEMENT CIRCLE

Implement the Circle class similar to the Rectangle and Triangle classes. Define circle with bounding rectangle.

Calculate the center the same as other geometric shapes.

20 Minutes

Practice makes perfect.

LAB: IMPLEMENT CIRCLE - 2

In main, create and display multiple shapes polymorphically.

- Create a vector of shapes
- Add a triangle to the vector
- Add a rectangle to the vector
- Add two circles to the vector
- Iterate through the vector. Call Draw on each item of the vector.

20 Minutes

Practice makes perfect.

LAB: IMPLEMENT CIRCLE - 3

Here are the perimeter points for the shapes:

Triangle 1,1 2,2 3,3,

Rectangle 1,2 2,3 3,4 4,5

Circle1 1,2 1,2 4,3 2,4

Circle2 4,1, 4,2 3,3 2,4

20 Minutes

Practice makes perfect.

Type Inference

Auto and Decltype

INITIALIZATION

Before Modern C++, various initialization techniques were available: equals, parentheses, default constructors, and globals. Initialization is the assignment of value while declaring a variable.

This lack of consistency can cause problems understanding and maintaining C++ source code.

Despite the variety of initialization techniques, there are limitations. Initializing STL containers is one limitation.

```
int a=0;
int b(5);

XClass x1(3,4);
XClass x2=x1;
```

CONTAINER INITIALIZATION

```
1  // Example program
2  #include <iostream>
3  #include <string>
4  #include <vector>
5  using namespace std;
6
7  int main()
8  {
9      int array[]={1,2,3,4};
10     vector<int> values={1,2,3,4};
11 }
12
```

Get URL

Run

options | compilation | execution

```
 In function 'int main()':
10:32: error: in C++98 'values' must be initialized by constructor, not by '{...}'
10:32: error: could not convert '{1, 2, 3, 4}' from '<brace-enclosed initializer list>' to 'std::vector<int>'
9:9: warning: unused variable 'array' [-Wunused-variable]
```

UNIFORM INITIALIZATION

Uniform initialization provides welcomed and standardization of initialization of new variables. Curly braces are used for uniform initializations and, in general, work everywhere.

```
int a={0}

int array[]={1,2,3,4};

vector<int> values={1,2,3,4};
```

NARROWING

Uniform initialization is safer. For example, it warns of narrowing and inadvertent lost of precision. Non-uniform initialization would allow a similar assignment.

```
1  // Example program
2  #include <iostream>
3  #include <string>
4  #include <vector>
5  using namespace std;
6
7  int main()
8  {
9      double x=25, y=45, z=15;
10     int a{x+y+z};
11 }
```

```
10:14: warning: narrowing conversion of '((x + y) + z)' from 'double' to 'int' inside { } [-Wnarrowing]
10:9: warning: unused variable 'a' [-Wunused-variable]
```

WHAT IS WRONG

Run the program. Do you notice any strange behavior?

```
#include <iostream>
using namespace std;

class XClass {
    public:

    XClass() {
        cout << "constructor - no arg";
    }

    XClass(int a) {
        cout << "constructor - n1 arg";
    }
};

int main() {
    XClass x1();
    XClass x2(1);

    return 0;
}
```

INITIALIZER LIST PREFERENCE

For constructors, the affect of parentheses and curly braces are the same with the exception of parameters that are initializer lists. Parentheses will prefer non-initializer list solution, while uniform initialization prefers initializer lists as parameters.

```
1  // Example program
2  #include <iostream>
3  #include <string>
4  #include <vector>
5  using namespace std;
6
7  class XClass {
8      public:
9
10     XClass() {
11         cout << "constructor - no arg" << endl;
12     }
13
14     XClass(int a, int b) {
15         cout << "constructor - 2 arg" << endl;
16     }
17     XClass(initializer_list<int> list) {
18         cout << "constructor - initilizer list" << endl;
19     }
20 };
21
22 int main() {
23     XClass x1(3,4);
24     XClass x2{3,4};
25     return 0;
26 }
27
```

Get URL

options | compilation | execution

```
constructor - 2 arg
constructor - initilizer list
```

DEFAULT CONSTRUCTOR

Default constructors are another example of interesting behavior when using initializer list.

Fortunately, classname() and classname{} calls constructor with an initializer list.

What is the behavior of the four constructors?

```
// Example program
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class XClass {
public:

    XClass() {
        cout << "constructor - no arg" << endl;
    }

    XClass(initializer_list<int> list) {
        cout << "constructor - initilizer list" << endl;
    }
};

int main() {
    XClass x();
    XClass x2{};
    XClass x3({});
    XClass x4;
    return 0;
}
```

CLASS MEMBER INITIALIZATION

Inability to initialize instance members of a class has always been slightly annoying. The problem is finally resolved ☺.

```
class XClass {

    public:

    int a=5;
    int b{10};

};
```

POP QUIZ: TEMPLATED STATIC MEMBERS

How would you initialize the static data members?

```
template<class T> class XClass {

    private:

    static int vara;
    static T varb;
};
```

10 MINUTES

AUTO AND DECLTYPE

The keywords *auto* and *decltype* provide type inference in Modern C++.

- Abstraction: abstracting the type. This can make the code more readable.
- Reduced redundancy: removing redundancy improves code quality (DRY).
- Easy refactoring: without redundancy, the code is more easily refactored and flexible.
- Less Typing. A fan favorite!

AUTO

The term *auto* is a re-"new" keyword; previous use of *auto* removed. The use of *auto* limited to declaring variables. The variable type is not late binding but early binding, implied, and static. For that reason, there must be enough detail to deduce the type.

Hard to write some code without *auto*, such as templates and autogenerated types.

```
std::vector<vector<int>> *myvector = new std::vector<vector<int>>();
```

or

```
auto myvector = new std::vector<vector<int>>();
```

AUTO LIMITATIONS

Cannot use *auto* in the following context:

- Function parameters (before C++ 14)
- Member variables
- static types

AUTO EXAMPLE

The following example demonstrates the *auto* keyword to declare a vector and than an iterator. Do you recall the syntax to declare an iterator for a vector of integers? Auto saves the day and deduces the correct type.

```
int main() {
    auto myvector = vector<int>();
    auto it = myvector.begin();
    myvector.insert(it, 200);

    return 0;
};
```

AUTO REQUIRED (ALMOST)

The lambda type is often undecipherable.

```
auto func = [](->int { return 42; };
auto result=func();
cout << result;
```

WHAT TYPE IS IT?

Determining the deduced type when using auto is mostly straightforward but occasionally surprising.

Type inference is similar to type deduction for templates. For example:

```
template<typename T>
        void f(ParamType param);
f(expr);
```

The are three considerations:

- Reference / pointer
- Value
- Universal reference

SCENARIO 1: REFERENCE / POINTER

1. T int
   TParam int ref
2. T const int
   TParam const int ref
3. T const int
   TParam const int ref

```
template<typename T>
T Func(T& param) {return param;}

int a=42;
const int a_const=a;
const int& a_ref=a;

int main() {

    Func(a);         // 1
    Func(a_const);   // 2
    Func(a_ref);     // 3
}
```

SCENARIO 2: REFERENCE / POINTER (CONST)

1. T int
   TParam int ref
2. T int
   TParam const int ref
3. T int
   TParam const int ref

```
template<typename T>
T Func(const T& param) {return param;}

int a=42;
const int a_const=a;
const int& a_ref=a;

int main() {

    Func(a);         // 1
    Func(a_const);   // 2
    Func(a_ref);     // 3
}
```

SCENARIO 3: BY VALUE

1. T int
   TParam int
2. T int
   TParam int
3. T int
   TParam int

```
template<typename T>
T Func(T param) {return param;}

int a=42;
const int a_const=a;
const int& a_ref=a;

int main() {

    Func(a);         // 1
    Func(a_const);   // 2
    Func(a_ref);     // 3
}
```

DECLTYPE

The *decltype* keyword is more specific than *auto.* It is not limited to capturing the type specification of variables. Unlike *auto*, *decltype* can also deduce the type of expressions. The result of decltype can be used as a substitute for a type name

- Does not evaluate the expression
- Expression must still be valid nonetheless
- Retains the adorned type

DECLTYPE SAMPLE CODE

```
#include <iostream>
#include <typeinfo>
using namespace std;

const int a = 10;

int main() {

          decltype(a) var=5;
          // var=10;  // does not work
          cout << var << " is a " <<
                    typeid(decltype(a)).name();
          return 0;
};
```

TRAILING RETURN TYPE

Demonstrates a trailing return type. Defer establishing the return type until after parameters are stated. The return type can then be derived from the parameter types.

```
template<typename L, typename R>
auto multiply(L l, R r)->decltype(r)
{
        return r;
}
```

DECLTYPE STRANGENESS

What is the expected output? Decltype evaluates the type of expression (the parameter) but does not actually execute the expression.

```
int Func(int &a) {
    a = a + 2;
    return a;
}

int main() {
    int var = 5;
    decltype(Func(var)) var2;
    cout << var;
    return 0;
};
```

INITIALIZER_LIST TYPE

The initializer_list is part of the attempt to standardize initialization in Modern C++ with curly braces (uniform initialization). A list of items with uniform initialization typical evolves into an initialization_list.

Initialization lists are implicitly created when:

- List provided during uniform initialization
- for range loop / auto

Copying an initializer_list does not copy the underlying data.

Include with the STD header: initializer_list. However, often unnecessary since most other STD headers already include initializer_list header.

The underlying type of initializer_list is “const T[N]”.

INITIALIZER_LIST DETAILS

Additional details of the initializer_list.

- Initializer_list is not a container. It is a lightweight wrapper over an compiler generated array.
- Many STL containers offer a constructor with a initializer_list parameter.
- Has an associated begin, end, and size function.
- Does not have at or data methods.

INITIALIZER_LIST SAMPLE CODE

```
#include <initializer_list>
#include <algorithm>
#include <vector>
using namespace std;

class XClass {
    public:
    XClass(initializer_list<int> _list) {
        for_each(begin(_list), end(_list),
            [=](int value)
        {elements.push_back(value); });
    }
    private:
    vector<int> elements;
};

int main() {
    XClass obj({ 5,9,6,4 });
    return 0;
}
```

INITIALIZERS RANGE LOOP / AUTO

In the adjacent code, an initializer_list is implied.

```
#include <iostream>
using namespace std;

int main() {
   for(auto value: {1,2,3,4}) {
       cout << value << endl;
   }
}
```

INITIALIZER INDEXING

Direct indexing is not supported for initializer_list types. However, the end function returns a T*. You can dereference with operator [] and specify an index.

```
#include <initializer_list>
#include <iostream>
using namespace std;


int main() {
    initializer_list<int> il={1,2,3,4,5};

    // cout << il[3] << endl;

            auto *pInt=begin(il);
            cout << pInt[0] << endl;
            cout << pInt[1] << endl;
            cout << pInt[2] << endl;

}
```

ALIAS

Better known as *type alias*.

- Replacement for typedef
  - typedef does not work with templates
  - typedef not as readable, such as function pointers
- Reduce typing
- Add context to code

```
using PlaneId = int;

template<typename T, typename V> using lookup
    = map<T, vector<vector<V>>>;

void main() {
    PlaneId test;

    lookup<string, int> nameLookup;
}
```

ALIAS FOR FUNCTION

Here is an example of defining a type of function pointer using typedef versus an alias.

```
typedef void (*fptr1)(int, int);

using fptr2=void (*)(int, int);

void FuncA(int a, int b) {

}

int main() {
    fptr1 f1=FuncA;
    fptr2 f2=FuncA;
    return 0;
}
```

LAB: SHOW TYPE

30 Minutes

This lab:

Using decltype create a function that accepts a function pointer as a parameter. The function should display the function prototype of the function pointer.

Test the function with:

```
float FuncA(int a) {
return 5;
}
```

and

```
void FuncB() {
}
```

Miscellaneous

NullPtr, Bool, Modern For Loops, and so on

NULLPTR

For pointers, replace the use of NULL with *nullptr.* The benefit is that *nullptr* is type safe.

The problem has been that NULL is zero, which is an integer not a pointer. This can lead to undisciplined code.

Nullptr rules:

- nullptr can be compared to NULL.
- nullptr cannot be assigned to an integer type
- nullptr is equivalent to false (bool type)

```
`=

// does not work

bool bValue = nullptr;


// does not work

int value = nullptr;


// does work

int *pInt= nullptr;


// true

bool bAnswer= NULL == nullptr ? true : false;
```

NULLPTR - 2

Ambiguity is a real possibility with NULL pointers. What happens in the adjacent code? The Func method is overloaded several times.

```
#include <iostream>
#include <typeinfo>
using namespace std;

void Func(int a){}
//void Func(long a){}
void Func(void* a){}

int main() {
    Func(NULL);
}
```

RANGED BASED FOR LOOP

This is the standard syntax of a for loop.

```
for(assignment;conditional;
        expression) { body }
```

If not careful, incorrect code can cause infinite loops, exceptions, and memory leaks.

Range-based for loops have a simpler syntax. Rely more on compiler and less on developer discipline. This improves code quality.

```
for(type value:collection) body
```

```
std::vector<int> myVector =
    { 1, 2, 3, 4, 5 };
for (auto value : myVector) {
    cout << value << endl;
}
```

SUPPORT FOR RANGED-BASED FOR LOOP

You can implement user defined types that support ranged-based for loops.

This is a minimal implementation required to support range-based loop.

- Declare a collection
- Calculate begin and end of collection
- begin method: return pointer to beginning of collection
- end method: return pointer to end of array

The following class generates a sequence from 1 to 10. The begin and end member methods are implemented to return the begin and end of the sequence respectively.

RANGED BASED FOR LOOP - CODE

```
class ZClass {
public:

    ZClass() {
        generate(&v[0], iend,
        [this]() {  return ++i; });
    }

    int * begin() {
        return &v[0];
    }

    int * end() {
        return iend;
    }

int i = 0;
    int v[10];
    int* iend = (&v[0] + (sizeof(v) /
        sizeof(int)));
};

int main() {
    ZClass obj;
    for (auto element : obj) {
        cout << element << endl;
    }
    return 0;
}
```

FOR_EACH LOOP

The for_each loop is a statement that applies a function to iterate a range items in a container. This is one more alternative to the standard for statement with multiple statements. The for_each statement is found in the std algorithm header.

The syntax is:

```
for_each(begin iterator, end iterator, function)
```

The for_each loop will iterate all elements between the begin and end iterators. The beginning iterator is inclusive; the ending iterator is exclusive. The function is often a lambda, which is discussed in a later module.

FOR EACH LOOP WITH TEMPLATES

Here is sample code using a for_each statement with a templated container.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

template<typename T> void Func(T a){
    cout << a << endl;
}

int main() {
    vector<int> vIntegers={1,2,3,4,5};
    for_each(vIntegers.begin(), vIntegers.end(),
        Func<decltype(vIntegers[0])>);
}
```

REGULAR ENUMS

Enum flags are not scoped to a particular enum.

This can cause ambiguity between two enums using the same flags. See right.

The solution are scoped enum classes.

```
enum EvenNumbers
{
        two,
        four,
        six
};
enum Numbers
{
        one,
        two,
        three
};
```

SCOPED ENUMS

To create a scoped enum, add the *class* keyword after *enum*. This is called a scoped enum declaration. With scoped enums, you must specify both the class and enum with the scope resolution operator. Modern C++ still supports traditional enums.

```
enum class EvenNumbers {
        two,
        four,
        six
};

enum class Numbers {
        one,
        two,
        three
};

auto var = Numbers::two;
```

LONG LONG

Modern C++ has introduced the *long long* data type, which is a 64-bit signed integer. The range is:

–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Here is the range for the *unsigned long long:*

0 to 18,446,744,073,709,551,615

This is confirmed in the limits.h file with these two constants:

- LLONG_MIN
- LLONG_MAX

STATIC_ASSERT

static_assert triggers a compile-time assertion, which is conveyed as a compile error and message.

If assert is *false*, there is a compilation error and the message is displayed.

If assert is *true*, there is no compilation error.

```
template <typename T, size_t Size>
class Vector {
   static_assert(Size < 3,
         "Size is too small");
   T _points[Size];
};
int main(){
   Vector<int, 16> a1;
   Vector<double, 2> a2;
   return 0;
};
```

LAB: GUESSING GAME

This lab:

Create a guessing game where the player has a limited number of guesses to discover a target value.

30 Minutes

GUESSING GAME - 2

User gets 5 guesses at a number. Here are the steps of the Guessing Number game:

1. Calculate target number using random number generation and a dynamic seed.
2. Prompt and then enter number to guess (< 100)
3. Prompt for guess
4. Enter guess
5. Save guesses in a vector.
6. If correct, end game. List number of guesses and congratulations.
7. After 5th incorrect guess:
    A. The guesses
    B. Display correct number
    C. Display game over message
    D. End game
8. Display “high” or “low” depending on guess
9. Go to Step 2

GUESSING GAME - 3

Details to help:

- Use srand and rand to calculate a target for each game. Both are found in stdlib.h.
- If the randomization seed is time, include the time.h header.
- Read information from the console using getline. The getline function is located in the iostream header.
- To convert from string to an integer use the stoi function located in the string header.

Have fun!

Extra credit

Numbers to Text

EXTRA CREDIT

Convert any number into a text representation. For example:

12 is

twelve

1,257 is

one thousand two hundred fifty seven

1,234,555 is

one million two hundred thirty four thousand five hundred fifty five

Be ready to present in the morning - last day of class. Team will vote on best solution.

Modern C++

Classes and Unions

FRAGILE BASE CLASS PROBLEM

The Fragile Base Class problem is when base class inheritance is destabilized by the parent / child class hierarchy.

This can occur inadvertently – especially when several developers work in the same code base.

In the following code, a developer accidentally overrides a method in the base class resulting in an unexpected side affect.

Modern C++ introduces solutions to the fragile base class problem.

FRAGILE BASE CLASS PROBLEM- CODE

```
class ZClass  { //  Developer Sally
public:

     void FuncB() {
          cout << "ZClass::FuncB" << endl;
     };
};

class YClass : public ZClass { // Developer Ben
public:

     void FuncA() {
     cout << "YClass::FuncA" << endl;
     FuncB();
};
```

```
     void FuncB() {
          cout << "YClass::FuncB"
               << endl;
     };
};

int main(){
     YClass obj;
     obj.FuncA();
     return 0;
}
```

OVERRIDE AND FINAL

The *override* and *final* keywords now help resolve the fragile base class problem. Allows developers to state their intention.

Either keyword must be applied to virtual methods

The *final* specifier prevents a method from being overridden in a derived class.

The *override* specifier stipulates that a method is intentionally overriding a method in the base class.

POP QUIZ: FINAL

Can the *final* keyword be used elsewhere in C++?

5 MINUTES

PROBLEM AVERTED- CODE

```
class ZClass  { //  Developer Sally
public:

    virtual void FuncB() final {
        cout << "ZClass::FuncB" << endl;
    };
};

class YClass : public ZClass { // Developer Ben
public:

    void FuncA() {
    cout << "YClass::FuncA" << endl;
    FuncB();
};
```

```
    void FuncB() { // Error
        cout << "YClass::FuncB"
            << endl;
    };
};

int main(){
    YClass obj;
    obj.FuncA();
    return 0;
}
```

PROBLEM - MISTAKEN OVERRIDE - CODE 2

```
class ZClass  { //  Developer Sally
public:

    virtual void FuncB() {
        cout << "ZClass::FuncB" << endl;
    };
};

class YClass : public ZClass { // Developer Ben
public:

    void FuncA() {
    cout << "YClass::FuncA" << endl;
    FuncB(5);
};
```

```
    void FuncB(int a) override {
        cout << "YClass::FuncB"
            << endl;
    };
};

int main(){
    YClass obj;
    obj.FuncA();
    return 0;
}
```

DEFAULT

Designates a method as a default. The best example is the default constructor. You can actually lose your default constructor by adding a constructor with parameters.

The default constructor can be added back as shown to the right. You can also use the default keyword to change the attributes of a compiler generated default method.

```
class XClass{
public:
        XClass(double) {}
        XClass() = default;

protected:
        XClass(const XClass &_outer)
                = default;
};
```

DELETE

The *delete* keyword means a particular method cannot be invoked or callable. Delete can be applied to a generated method or an explicit method.

.

```
template<typename T>
void DoSomething(T _arg) {
}


void DoSomething(int _arg) = delete;
void main()
{
        DoSomething(1.5);
        // will not compile
        DoSomething(1);
}
```

DELETE – ANOTHER EXAMPLE

```
class XClass
{
public:
        XClass(double) {}
        XClass(int) = delete;
        XClass() = delete;
};

int main()
{
        XClass obj1(12.0);
        XClass obj2(1);  // not compile
        XClass obj3;  // not compile
}
.
```

INHERITING CONSTRUCTORS

This feature streamlines the invocation of base class constructors in the child class. As the name implies, inheriting constructors allows a child class to directly inherit base class constructors. This is particularly useful when the child has no initialization otherwise. With inheriting constructors, the child class can override any inherited constructor and reimplement.

```
class ZClass {
public:
    ZClass() {
        cout << "ZClass()" << endl; }
    ZClass(int) { cout << "ZClass(int)"
        << endl; }
};
class YClass : public ZClass {
public:
    using ZClass::ZClass;
}
int main() {
    YClass obj(7);
};
```

POP QUIZ: USING

Is the using statement available within a class for normal methods?

If so, what is the behavior?

10 MINUTES

DELEGATING CONSTRUCTORS

A class constructor can delegate to a constructor of the base class through the colon initialization list versus creating a member function, such as Init.

```
class ZClass7 {
public:
      ZClass7() :ZClass7(5) {
            cout << "ZClass()" << endl;
      }

      ZClass7(int) {
            cout << "ZClass(int)"
                  << endl;
      }
};

class YClass7 : public ZClass7 {
public:

};

int main() {
      YClass7 obj;
      return 0;
}
```

POP QUIZ: WHAT DO YOU THINK?

What do you think of the following code?

```
#include <iostream>
using namespace std;

class WClass {
public:
   WClass() : WClass(4,5), a(10) {
      cout << "WClass()" << endl; }

   WClass(int, int) { cout << "WClass(int, int)"
      << endl; }

   void FuncA( ) { cout << "WClass.FuncA";}

   int a;
};

int main() {

    WClass obj;
}
```

5 MINUTES

IMPLICIT CONSTRUCTORS AND OPERATORS

Certain C++ class methods can be called implicitly when not expected, which has been the bane of many developers. This is particularly true of constructors and operator methods.

- Conversion constructor
- Copy constructor
- Default constructor
- Assignment operator

```
class XInt {
public:
      XInt(int _data){
      data = _data;
      }
      int GetData(){ return data; }
private:
      int data;
};

XInt impliedObject = 10;
```

PREVENT IMPLIED INVOCATION

Preface a member method with the *explicit* specifier to prevent implicit invocation.

```
class XInt {
public:
        explicit XInt(int _data){
        data = _data;
}
int GetData(){ return data; }
private:
        int data;
};

XInt impliedObject = 10;  // error
```

POP QUIZ: DOES THIS CODE WORK?

```
class IntWrapper {
public:

     IntWrapper() : pInt(0) {}
     int *pInt;
};

union Integer {
     IntWrapper pInt;
     int varInt;
};

int main() {
     Integer obj;
     return 0;
}
```

5 MINUTES

UNIONS

Modern C++ relaxes some of the rules related to unions – particularly pertaining to union members.

Previously unions could not include members that are types with non-trivial members, such as explicit constructors. That restriction has been removed.

If the instance (non-static) data member has a non-trivial constructor, the union must also explicitly have that constructor.

```
class IntWrapper {
public:

      IntWrapper() : pInt(0) {}
      int *pInt;
};

union Integer {
      Integer() {pInt = IntWrapper2();} // important
      IntWrapper pInt;
      int varInt;
};
```

DISCRIMINATING UNIONS

Discriminating unions are a combination of a type and an anonymous union. The structure contains a member that indicates the active member.

```
class DiscriminatedUnion {
public:

        DiscriminatedUnion(int _var)
        : type(_int), memberi(_var) {}

        DiscriminatedUnion(double _var)
        : type(_double), memberii(_var) {}

        enum { _int, _double} type;

        union {
                int memberi;
                double memberii;
        };
};
```

LAB: IDENTIFICATION

This lab:

Travelers typically must present identification when traveling. In this lab, you will create a Travel class that contains identification that is mutually exclusive and implemented using unions.

45 Minutes

Practice makes perfect.

LAB: IDENTIFICATION - 2

45 Minutes

Create an enumeration for two types of identification called IdType: Passport and DriverLicense.

Define Passport structure that has a country and issuer field. Both strings. Create a constructor to initialize the fields.

Define DriverLicense structure that has a state (string) and glasses (bool). Create a constructor to initialize the fields.

LAB: IDENTIFICATION - 3

45 Minutes

Define a union type called Data. In the type, union the Passport and DriverLicense structures.

In the union type:

- Create a constructor for the Passport member.
- Create a constructor for the DriverLicense member.
- Add an empty destructor.

LAB: IDENTIFICATION - 4

45 Minutes

Create a Traveler type. Here are the private members (name:Type):

- name: string
- id: string
- type: IdType
- data: Data

LAB: IDENTIFICATION - 5

Add two constructors to the Traveler class. The first constructor is for the DriverLicense identification.

```
Traveler(string _name, string _id, string _state,
          bool _glasses) : data(_state, _glasses){
     name = _name;
     id = _id;
     type = IdType::DriverLicense;
}
```

Similarly create the other constructor. This time for the Passport.

45 Minutes

Practice makes perfect.

LAB: IDENTIFICATION - 6

Test the code:

```
int main()
{
      Traveler person("Bob", "Dylan",
            "South Carolina", true);
}
```

Add a traveler that has DriverLicense identification.

45 Minutes

Practice makes perfect.

Lambdas

Anonymous methods

LAMBDAS OVERVIEW

Lambdas are probably one of the most anticipated new features of Modern C++. A lambda is an anonymous method. A simple concept with a major impact. With lambdas, functions are promoted to full variable status.

- Replacement for function objects
- Invokable with an operator()
- Perfect for STL algorithms
- The syntax can range from simple to complex

LAMBDA SYNTAX

Lambda expressions start with square brackets for capturing data, then the parameter list, and finally the lambda body.

```
[ capture-list ] ( params ) mutable
                exception attribute -> ret { body }
[ capture-list ] ( params ) -> ret { body }
[ capture-list ] ( params ) { body }
[ capture-list ] { body }
```

The [] identifier, called the capture specifier, is the clearest indication of a lambda.

LAMBDA INTERNALS

Lambda functions are implemented as classes with call operators (i.e., operator ()). The lambda class is called a closure type.

```
class MyLambda {
public:
        void operator()(int a) {
        }
private:
        int capturedData;
};
```

ANOTHER HELLO WORLD AP!

```
auto func = [] () { cout << "Hello world" << endl; };
func(); // now call the function

([] () { cout << "Hello world"; })();
```

LAMBDA RETURN

- Returns within the lambda method must be the same type
- A lambda can return nothing
- Compiler can determine type

PARAMETERS

- Variable argument list not allowed
- No unnamed parameters
- If there are no parameters, parameter syntax can be omitted.

FIRST CLASS OBJECTS

Functional programming treats functions as first-class objects.

As discussed, this means lambdas as a variable, return type, or even a function parameter.

Alternatively, lambdas can be defined as a std::function type. Shown right.

```
auto Function1() {
     return[]()->int{return 42; };
}
void Function2(std::function<int ()>
     func) {
     cout << func() << endl;
}

int main() {
     cout << Function1()() << endl;
     Function2([]{return 42; });
     return 0;
}
```

MORE ON STD::FUNCTION

The std::function type can be used as a parameter or return type for lambdas. std::function is found in the functional header file.

```
function<int(void)> FuncA()
{
        static int a=10;
        return [&] {return ++a; };
}

int main() {

        auto a = FuncA();
        auto b = FuncA();
        cout << a() << " " << b();

};
```

VARIABLE CAPTURE

Lambdas can capture variables outside the scope of the body of the lambda. As mentioned, [] is the capture specifier.

- You can capture by value or reference
- You can capture as read-only or mutable
- You can capture specific variables or in general

VARIABLE CAPTURE SYNTAX

[] Capture nothing

[&] Capture variable by reference

[=] Capture variable by value

[=, &foo] Default capture any referenced variable by copy, but specifically capture variable foo by reference.

[this] Capture the *this* pointer of the surrounding class

LAMBDAS CAPTURE EXAMPLE

```
class ZClass {
public:

    int member_a;

    void CaptureSpecifier() {
        int a = 5;
        int b = 6;

        []() {  a = 10;
            member_a = 12;
        };

        [&a]() { a = 10;
        cout << a << endl;
        };
        [=, &a]() { a = b;
        };
        [&, a]() { b = a;
        };
        [this]() { member_a = 12;
        };
    }
};
```

MUTABLE LAMBDAS

Use "=" to capture by value. Variables captured by value are read-only. The *mutable* modifier makes capture by value variables editable inside the lambda.

In contrast, capture by reference makes the outside variable editable.

POP QUIZ: WHAT IS THE RESULT?

```
void Func(){
    int a = 5;
    int b = 10;
    [=] () mutable{
        int temp = a;
        a = b;
        b = temp;
    }();
    cout << a << " "
    << b << endl;
}
```

5 MINUTES

LAB: DOING MATH

20 Minutes

This lab:

Create lambdas representing different math operations. Apply operations to a matrix of values.

Practice makes perfect.

DOING MATH

Create an array of lambdas representing math operations that return double types. Each takes two parameters and returns the result.

	operation	param 1	param2	answer
Addition	a	4	5	?
Division	d	0	3	?
Multiplication	m	4	8	?
Subtract	s	1	4	?

Create a two-dimensional array to input the operation type, parameters, and zero for the result.

Iterate the matrix using a range-based for loop. Calculate the operation with parameters and store results back on the same row.

Iterate the matrix and display the operations, parameters, and the result.

.

Classes with Pointers

Base class to derived objects

POLYMORPHISM IMPLEMENTATION

When copying objects, classes with pointers as data members have a specific implementation model. Special implementation for these methods:

- Copy constructor
- Assignment operator
- Destructor

Default copy constructor is a shallow copy which is a binary copy

Binary copy is problematic for objects that contain pointers. This can create dependencies, such as dangling pointers.

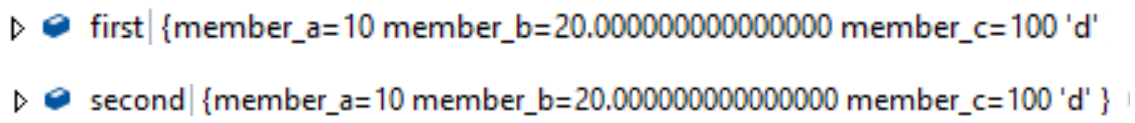
BINARY COPY

The binary copy (shallow copy) is the default:

```
XNormal second(first);
```

C++ does an efficient bitwise copy from *first* into *second*

second is now an identical clone of *first*



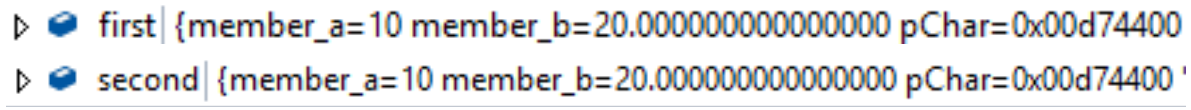

```
class XNormal{
public:
        int member_a=10;
        double member_b=20;
        char member_c='d';
};
int main(){
        XNormal first;
        XNormal second(first);
        return 0;
}
```

OBJECTS WITH POINTERS

When copying objects with pointers, binary copy may be inappropriate.

The result is two or more objects sharing the same pointer.

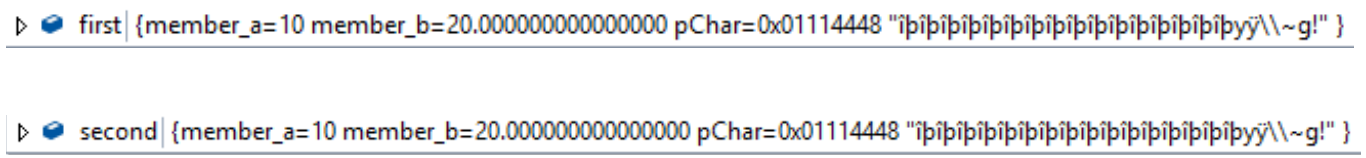
This creates dependencies and potential side affects




```
class XObjectWPointer{
    public:
    int member_a = 10;
    double member_b = 20;
    char* pChar =
            new char{ 'D' };
};

int main(){
    XObjectWPointer first;
    XObjectWPointer second(first);
    return 0;
}
```

THE PROBLEM

What happens when the shared pointer of one of the dependent objects is deleted or leaves scope. This will expose the dependency. The remaining dependent objects now own a invalid pointer.




```
XObjectWPointer first;
XObjectWPointer second(first);


delete first.pChar;
cout << second.pChar << endl;
```

DEEP COPY

- Object with pointers should perform a memberwise copy, which is called a deep copy.
- For objects without pointers, binary copy is typically okay.
- For copying pointer members, allocate new memory and copy existing value to the new location.
- Deep copy is implemented in these three custom methods:
  - Copy constructor
  - Assignment operator
  - Destructor

COPY CONSTRUCTOR

```
class FullName {
    public:
    FullName(const FullName& _inObject)
        :FullName(_inObject.first,
          _inObject.last, inObject.age)
    {
    }
```

```
FullName(char* _first, char* _last, int
_age) {
    delete first;
    delete last;
    age = _age;
    first = new char[strlen(_first) +
1];
    last = new char[strlen(_last) + 1];
    strcpy(first, _first );
    strcpy(last, _last);
    }
int age;
char* first;
char* last;
};
```

ASSIGNMENT OPERATOR

```
FullName& operator=(const FullName &_inObject)
{
        if (this == &_inObject)
        {
                return *this;
        }

        FullName(_inObject.first, _inObject.last, _inObject.age);
        return *this;
}
```

DESTRUCTOR

In the destructor, free the memory created at runtime for data members.

COPY CONSTRUCTOR

```
class FullName {
    public:
    FullName(const FullName& _inObject)

    :FullName(_inObject.first,_inObject.last,
        _inObject.age)
    {
    }
```

```
FullName(char* _first, char* _last, int
_age) {
    delete first;
    delete last;
    age = _age;
    first = new char[strlen(_first) +
1];
    last = new char[strlen(_last) + 1];
    strcpy(first,_first );
    strcpy_s(last,_last);
    }
int age;
char* first;
char* last;
};
```

LAB: CREATE A CLASS PERSON

This lab:

Create a Person class, which has pointers, and the required methods.

30 Minutes

Practice makes perfect.

PERSON

- Define a class called Person. The class has two pointers (char*). 1) Pointer to a Car. 2) Pointer to a House. The Person class also containers a stl::string for storing the person's name.
- The House class has a single member: string for the house address.
- The Car class has a single member: string for car description.
- Implement the methods required for a deep copy in the Person class.
- Implement any other method for the three classes that is required.
- Create names, home address, and car description for two people.
- In main, create a couple of instances of Person. Exercise the objects to demonstrate deep copy.

Rvalue References

Move Semantics and Perfect Forwarding

RVALUE OVERVIEW

Great concept with considerable power; but has caused the most confusion for Modern C++ developers.

Historically there is one reference type – reference. Modern C++ introduces rvalue and lvalue references.

This also introduces move semantics, which allows the stealing of a portion of another object.

WHAT IS AN RVALUE?

Everything that is not a Lvalue. Seriously!

RVALUE

An lvalue is an entity that persist beyond a single expression and the traditional reference. From another perspective, lvalues are named while rvalues are not.

An rvalue is a temporary object that does not persist beyond an expression.

What is a lvalue versus a rvalue in the following code?

```
Int main() {
        int x=3+4;
        cout << x << endl;
}
```

LVALUE VERSUS RVALUE

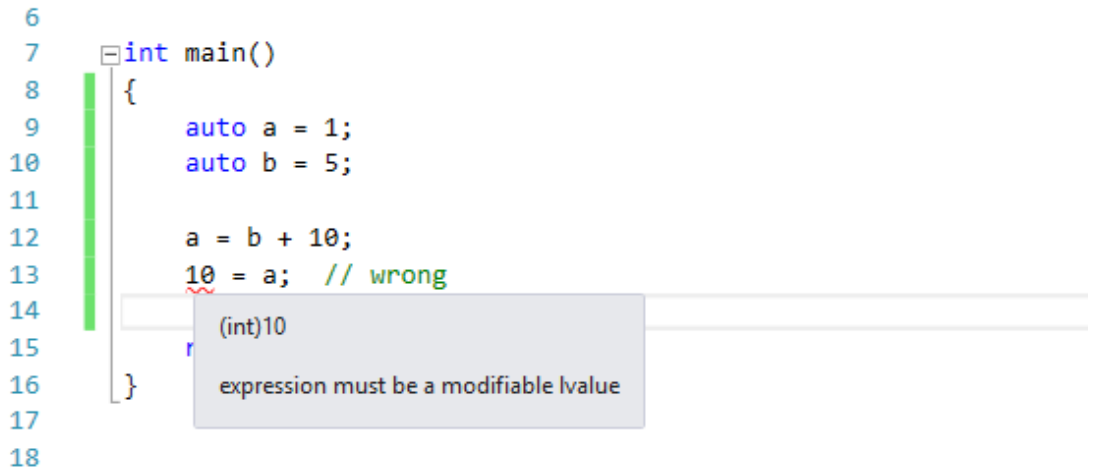
Basic explanation

- Lvalue can appear on the left of an assignment.
- Lvalue can evaluate to an addressable value
- Rvalue can only appear on the right side of an assignment.
- Rvalue cannot be evaluated to an explicit address.

```
int FuncA(){
     return 5;
}
int &FuncB() {
     int b=10;
     return b;
}
void main() {
     int a = 100; // a is lvalue
     100 = a; // 100 is a rvalue
     int c = FuncA(); // FuncA is a
         rvalue
     FuncB() = a; // What about FuncB?
}
```

LITERAL CONSTANTS

Literal constants are an example of Rvalue. They are not addressable. Any attempt to use a Rvalue as an Lvalue will cause a compiler error.



RVALUE REFERENCE OPERATOR

- & indicates a Lvalue reference. Lvalue reference is a constant pointer to another object. Lvalue reference can only reference a Lvalue.
- && indicates a Rvalue reference for referencing Rvalues.

```
int a = 5;
int &b = a;


//int &c = 10;
int &&d = 10;
```

OVERLOAD ON REFERENCE TYPE

You can overload methods based on the reference type: Lvalue versus Rvalue reference.

Which overloaded FuncA method is called in the following code?

```
void FuncA(int &var) {
     cout << "Lvalue reference" << endl;
}

void FuncA(int &&var) {
     cout << "Rvalue reference" << endl;
}

int main() {
     int a = 5;
     FuncA(10);
}
```

POP QUIZ: IS THIS EFFICIENT?

```
FullName(const FullName&
    _inObject)
        :FullName(_inObject.first,
        _inObject.last, _inObject.age)
    {
    }
```

10 MINUTES

MOVE SEMANTICS

- Copy by value can be expensive – especially temporary objects pointing to a lot of data
- Not all temporary variables are obvious
- Why copy when you can steal from a temporary object?
- Stealing is quicker!

REGULAR COPY C'TOR (W/PTR)

- Initialize an internal pointer for the current object
- Copy the value at the pointer to the internal object.

```
XClass(const XClass &var) {
    cout << "XClass(
        const XClass &var)";
    pVar = new T();
    *pVar = *var.pVar;
}
```

RVALUE COPY C'TOR (W/PTR)

- Steal pointer of temporary object
- Notice function parameter is not const
- Set source pointer to null
- No copy!

```
XClass(XClass &&var) {
    cout << "XClass(XClass &&var)";
    pVar = var.pVar;
    var.pVar = nullptr;
}
```

REGULAR OPERATOR= (W/PTR)

- Delete internal pointer
- Initialize internal pointer
- Copy the value at the pointer to the internal object.
- Return current object

```
XClass &operator=(
        const XClass<T> &var) {
    cout << "operator=(XClass<T> &var)";
    delete pVar;
    pVar = new T();
    *pVar = *var.pVar;
    return *this;
}
```

RVALUE OPERATOR= (W/PTR)

- Delete internal variable
- Steal pointer of temporary object
- Notice parameter is not const
- Set source pointer to null
- No copy!
- Return current object

```
XClass &operator=(XClass<T> &&var) {
     cout << "operator=(XClass<T> &&var)";
     delete pVar;
     pVar = var.pVar;
     var.pVar = nullptr;
     return *this;
}
```

POP QUIZ: WHAT AND WHEN

This code exercises the XClass. What is called when?

```
XClass<int> obj1(5);
XClass<int> obj2(obj1);
obj1 = XClass<int>(15);
obj1 = obj2;
```

10 MINUTES

STD::MOVE

The function std::move mandates move semantics.

- If unclear whether Lvalue versus Rvalue, forces move semantics.
- Great for Lvalue references when the object will not be used again.
- Treat a Lvalue reference as a temporary object

UNIVERSAL REFERENCE

The && token can indicate either a rvalue or universal reference.

- When in a type declaration, && refers to a rvalue reference.
- Otherwise, it is a universal reference. For example, with type inference, && is a universal reference.
- With universal reference, the type is determined by reference collapsing rules.

```
auto && a=5
```

REFERENCE COLLAPSING RULES

Here are type reference deduction rules for universal references.

Target	Universal	Result
T&	&	T&
T&	&&	T&
T&&	&	T&
T&&	&&	T&&

STD::MOVE CODE EXAMPLE

```
class XClass {
public:
    XClass() { pA = new int; }
    int GetA() { return *pA; }
    int *pA;
};

__declspec(noinline) void
FuncC(XClass && obj){
// temp XClass
    cout << "FuncC(XClass && obj)"
        << endl;
}
```

```
__declspec(noinline) void FuncC(
        XClass & obj){
    // XClass
    cout << "FuncC(XClass & obj)"
        << endl;
}

int main() {
    XClass obj;
    FuncC(std::move(obj));
};
```

PERFECT FORWARDING

Like move semantics, perfect forwarding reduces overhead associated with a function call. Often, a function call is essentially a delegate to another function.

Calling FuncB is essentially a call to FuncA. However, there is additional overhead of two pass by value instead of one pass by value. If obj is a heavy object, the additional overhead could be considerable.

On the right, three copy by value constructors are called.

```
class ZClass {
public:
      ZClass() { cout << "Regular ctor" << endl; }
      ZClass(const ZClass & obj)
            { cout << "Regular ctor" << endl; }
};
void FuncA(ZClass obj) {
}
void FuncB(ZClass obj) {
      FuncA(obj);
}
int main() {
      ZClass obj;
      FuncB(obj);
    return 0;
}
```

PERFECT FORWARDING - 2

Perfect forwarding removes the potential additional overhead of functions that are thin wrappers for delegating to another function.

Perfect forwarding is accomplished with a combination of move semantics and std::forward to forward parameters through a thin wrapper.

On the next page, one copy by value constructor is called.

PERFECT FORWARDING - 3

```
class ZClass {
public:
    ZClass() {
        cout << "Regular ctor"
             << endl;
    }
    ZClass(const ZClass & obj) {
        cout << "Regular ctor"
        << endl; }
    ZClass(ZClass && obj) {
        cout << "Move ctor" << endl;
    }
};
```

```
void FuncA(ZClass &&obj) {
}

void FuncB(ZClass &&obj) {
    FuncA(std::forward<ZClass>(obj));
}

int main(){
    ZClass obj;
    FuncB(std::move(obj));
   return 0;
}
```

LAB: MODIFY PERSON CLASS

This lab:

Add move semantics to Person class.

30 Minutes

Everything Pointers

Smart Pointers

PROBLEMS WITH POINTERS

Pointers provides unparallel flexibility to developers; but also many challenges. There is a reason that many languages do not make pointers visible.

- Pointer type not evident
- The pointer itself present little information
- Usage pattern not evident
- Double deletes are possible
- Dangling pointers
- Double free
- Buffer overruns
- Dependencies

SMART POINTERS OVERVIEW

Modern C++ helps developers manage pointers with an assortment of smart pointers.

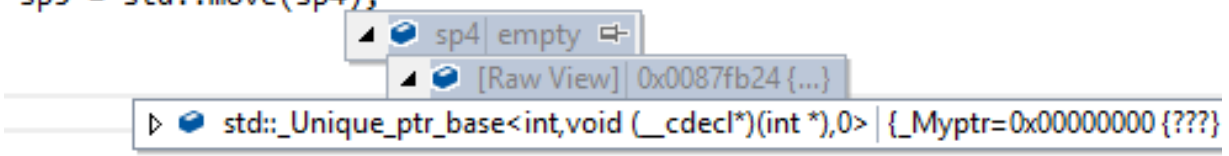
- unique_ptr
- shared_ptr
- weak_ptr

The auto_ptr smart pointer is deprecated for unique_ptr.

UNIQUE_PTR

- Exclusive ownership semantics
- Maintains a unique instance of an object via a pointer
- Same size as the raw pointer
- No reference counting
- unique_ptr owns the referenced inner pointer
- Transferring ownership of a pointer sets your pointer to null.

```
unique_ptr< int, void(*)(int*)>  sp4(new int[10], [](int* obj)
        {cout << "Deleter called" << endl; });

unique_ptr< int, void(*)(int*)>  sp5 = std::move(sp4);

sp5 = std::move(sp4);
```



METHODS

Here are the methods for the unique_ptr. The pointers operators are also implemented.

Method	Description
(constructor)	Instantiate a unique pointer
(destructor)	Deletes a unique pointer
get	Get pointer
get_delete	Gets deleter
operator=	Transfer ownership of pointer
operator bool	Return true if not empty
release	Releases pointer (sets to nullptr)
reset	Destroys the pointer and updates to new pointer

RELEASE UNIQUE_PTR

- Destroyed when unique_ptr goes out of scope
- unique_ptr assigned another pointer
- operator= : assigns unique pointer to another object. Releases the current pointer.
- reset: updates the unique_ptr while deleting the current pointer. If not empty, calls the delete method.

MAKE_UNIQUE VERSUS UNIQUE_PTR

There are two ways to declare a unique pointer: make_unique and unique_ptr.

```
auto ptr1 = make_unique<Circle>();

Rectangle *pRect = new Rectangle();
auto ptr3 = unique_ptr<Rectangle>(pRect);
auto ptr4 = unique_ptr<Rectangle>(new Rectangle);

ptr4->Draw();
```

DELETER

You can customize the delete behavior. The deleter function should have a pointer parameter and return void.

```
void Func(char* p) {
    cout << "deleting";
}

auto a=unique_ptr<char, void(*)(char*)>(
     new char('c'), Func);
```

SAMPLE CODE

In this sample code, FakeThread represents sample threads. Theoretically, you are running a primary thread where multiple person objects are created. You can spin up a thread to handle each person. This requires handing off the person object to another thread.

```
void FakeThread(unique_ptr<string> person) {
    • // do something
}

int main() {
   vector < unique_ptr<string>> persons;

   persons.push_back(make_unique<string>("Bob"));
   persons.push_back(make_unique<string>("Ted"));
   persons.push_back(make_unique<string>("Carol"));
   persons.push_back(make_unique<string>("Alice"));

   FakeThread(std::move(persons[3]));

   return 0;
}
```

SHARED_PTR

The shared_ptr type replaces auto_ptr, which is now deprecated. Supports shared ownership. Counts the number of owners. When count is zero (all owners have released ownership), the object is deleted.

Shared_ptrs are decremented when shared_ptr::release called, reassignment, or out of scope.

Supports reference counting

- Assignment
- Pass by value

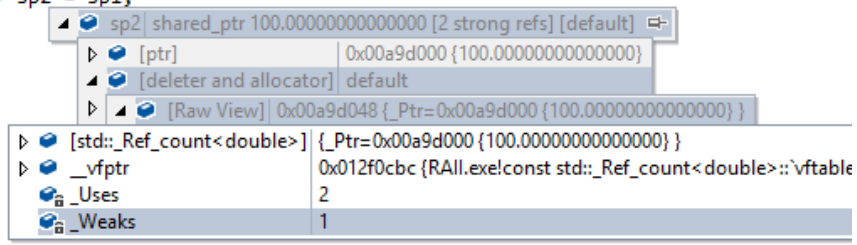
Syntax;

```
shared_ptr<double> sp1(new double(100));
```

or

```
auto x = std::make_shared<int>(5);
```

```
void FuncShare()
{
    shared_ptr<double> sp1(new double(100));
    shared_ptr<double> sp2 = sp1;
}
```



SHARED_PTR

```
#include <iostream>
#include <memory>
using namespace std;

int main() {
     auto x = std::make_shared<int>(5);
     // new block
     {
          decltype(x) a=x;
          cout << a.use_count() << endl;
     }
     cout << x.use_count() << endl;
     x.reset();
     cout << x.use_count() << endl;

    return 0;
}
```

METHODS

Here are the methods for the shared_ptr. The pointers operators are also implemented.

Method	Description
(constructor)	Instantiate a shared pointer
(destructor)	Deletes a shared pointer
get	Get pointer
operator=	sharing ownership of pointer
operator bool	Return true if not empty
use_count	Return usage count
release	Releases pointer (sets to nullptr)
reset	Destroys the ownership of the pointer for a single owner.
unique	Returns true if pointer is unique

WEAK_PTR

Weak pointers refer to a weak reference to memory. A weak pointer alone is not enough to preserve memory. Weak pointers do not take ownership of a pointer reference.

Technically - weak pointers create a shared reference without adding to the reference count.

Create a weak pointer to optional preserve a pointer in memory. You confirm the availability of a weak pointer with the expired method.

WEAK_PTR

Assign a shared_ptr to the weak_ptr.

Before using, convert the weak_ptr to a shared_ptr with lock method.

If pointer no longer available, lock returns an empty shared_ptr.

The expired method return false if pointer remains available.

```
auto pShared = make_shared<int*>(new int);
weak_ptr<int*> pWeak = pShared;

// Use pWeak

if (auto pAvailable = pWeak.lock()) {
     cout << "Pointer available";
}
else {
     cout << "Pointer not available";
}
```

LAB: BUILDING

1 hour

This lab:

You will simulate people entering and leaving a building. You must open and close the building. Create a shared pointer for tracking people entering the building. You cannot close the building at the end of day if not empty (people in the building).

BUILDING

- Create a forward reference to Building class.
- Create a class called Person. Here are the members:
    - name:string
    - pBuilding:shared pointer to Building class. Initiallize to a null pointer.
    - C'tor to set name
    - Accessor to get name
    - SetBuilding is a mutator method to initialize pBuilding
    - GetBuilding is an accessor method that returns pBuilding
- Create a class called building with these members:
    - pBuilding:shared pointer to Building class. Initiallize to a null pointer.
    - Create the Open method: returns nothing and has a Person pointer as parameter (the landlord)
        - If building not open,
            - Construct a new building at runtime and initialize pBuilding pointer
            - Set Building pointer for Person
            - Display message "Building opened: {name of person}"
        - If building open,
            - Display message that building is already open

BUILDING - 2

- Create the Enter method: returns nothing and has a Person pointer as parameter.
  - If building open:
    - Create another Building shared pointer from the current building pointer.
    - Assign new Building pointer to the person.
    - Display a message that the person has entered the building.
  - If building not open:
    - Display a message that the building is closed.
- Create the Close method: returns nothing and has a Person pointer as parameter.
  - If shared pointer for person not unique, display message “Can’t close building”.
  - If shared pointer is not unique, reset pBuilding. Set pBuilding to null pointer. Display a message that the building is now closed.

BUILDING - 3

Create the Leave method: returns nothing and has a Person pointer as parameter.

Get building pointer for person. If null, display message that person is not in the building and cannot leave.

If pointer non-null, reset the building pointer for the person.

Set the building pointer for person to null.

Display message that person has left building

Test application

```
auto pShared = make_shared<int*>(new int);
weak_ptr<int*> pWeak = pShared;

// Use pWeak

if (auto pAvailable = pWeak.lock()) {
    cout << "Pointer available";
}
else {
    cout << "Pointer not available";
}
```

BUILDING - 2

- Create the Leave method: returns nothing and has a Person pointer as parameter.
  - Get building pointer for person. If null, display message that person is not in the building and cannot leave.
  - If pointer non-null, reset the building pointer for the person.
  - Set the building pointer for person to null.
  - Display message that person has left building
- Test application

```
int main()
{
    Person Bob("Bob");
    Building building;
    building.Open(&Bob);

    Person Sally("Sally");
    building.Enter(&Sally);
    Person Fred("Fred");
    building.Enter(&Fred);
    building.Close(&Bob);
    building.Leave(&Sally);
    building.Leave(&Sally);
    building.Leave(&Fred);
    building.Close(&Bob);

    return 0;
}
```

Noexcept

Exception Handling

OVERVIEW

Modern C++ makes it easier to define whether or not a function raises an exception. Label functions with noexcept to guarantee that an exception is not emitted. Clients can then query the exception status of a method.

This is checked at runtime and not at compile time. If exception is raised, std::unexpected not called.

The noexcept causes compilers to more optimize the binary. This is because the call stack is only potentially unwound.

DEFAULT FUNCTIONS

Compiler generated methods are noexcept, if every function directly invoked is also noexcept. Examples are:

- Default constructor
- Default destructor
- Default assignment operator
- And so on

USER DEFINED FUNCTIONS

noexcept can be applied to custom methods.

- Promise method will not raise an exception
- If exception raised, std::terminate is called. This will cause an immediate termination.
- For this reason, all exceptions must be caught in the method.

```
void Amphibious(Vehicle* _pVehicle,
        Boat* _pBoat) noexcept {
    try {
        SetPointers(_pVehicle,
            _pBoat);
    }
    catch (char* message) {
        cout << message << endl;
    }
}
```

EXCEPTION NEUTRAL

Most user defined methods are actually exception neutral. Neutral means while exceptions are not likely raised in the function. However, exceptions may *pass through* the method while propagating up the call stack.

Exception neutral methods should not have the noexcept attribute.

NO_EXCEPT(BOOLEAN)

NO_EXCEPT can deny or accept exception handling within a method. You can provide a single constant Boolean expression as a parameter. Default is true.

```
Amphibious(Vehicle* _pVehicle, Boat*
_pBoat) noexcept(false)
{
    SetPointers(_pVehicle,
        pBoat);
}
```

NO_EXCEPT(BOOLEAN)

NO_EXCEPT can deny or accept exception handling within a method. You can provide a single constant Boolean expression as a parameter. Default is true.

```
Amphibious(Vehicle* _pVehicle, Boat*
_pBoat) noexcept(false)
{
    SetPointers(_pVehicle,
        pBoat);
}
```

NOEXCEPT OPERATOR

The NOEXCEPT operator is used within the NOEXCEPT specifier. If expression should not throw an exception, operator returns true. Otherwise, returns false.

```
void ZeroFunc(int a, int b)
        noexcept(noexcept(a / b)) {
    a /= b;
    cout << "Worked " << a;
}
```

POP QUIZ: WHAT DOES THIS MEAN?

```
void FuncA() noexcept {
   cout << "test" << endl;
}

void FuncB()
     noexcept(noexcept(FuncA())) {
   cout << "test2" << endl;
}
```

10 MINUTES

EXCEPTION_PTR

A shared pointer for saving exception objects.

Exception objects created with these methods:

- current_exception
- make_exception_ptr
- nested_exception::nested_ptr

The implementation of the exception object is environment specific (data1, data2).

```
int main(){
    try {
        throw "exception";
    }
    catch (...) {
        exception_ptr ePtr =
            current_exception();
    }
    return 0;
}
```

Variadic Templates

Variable Length Templated Types

IN THE PAST

Functions with variable number of parameters (i.e., variadic methods) have always been available in C++. The best example is the ever popular printf. However there have been several shortcomings with the implementation:

- Not naturally type safe
- Overly complicated
- Required the use of macros
- Not templated
- Runtime resolution

In Modern C++, variadic templates are a major step forward when compared to previous solutions.

VARIADIC METHOD

Here is sample code for a variadic method for non-Modern C++.

```
int FindMax(int n, ...) {
   int i, val, largest;
   va_list vl;
   va_start(vl, n);
   largest = va_arg(vl, int);
   for (i = 1; i<n; i++) {
      val = va_arg(vl, int);
      largest = (largest>val) ? largest : val;
   }
   va_end(vl);
   return largest;
}

int main(){
   int m;
   m = FindMax(7, 702, 422, 631, 834, 892, 104, 772);
   printf("The largest value is: %d\n", m);
   return 0;
}
```

VARIADIC TEMPLATES

Variadic templates provides a model for writing functions that take an variable number of type safe parameters. Since the call sites are known, safeness is resolved at compile time; not runtime.

You specify a variadic template using the typename…*Args* syntax, which is called a function parameter pack. Variadic templates are executed similar to a recursive method.

When creating variable length variadic templates for functions, you actually create two functions. First, a function for the last element(s). Second, a function to reduce the variadic list.

VARIADIC METHOD - SAMPLE

Here is sample code for a variadic method:

- The variadic method is called (Args… args) performs list reduction – removing one item at a time.
- The one-argument adder is called for the final item, which typically ends the recursiveness.

In this example, adder just sums the items in the list.

```
template<typename T>
T adder(T v) {
    cout << "One Arg Method " << v << endl;
    return v;
}

template<typename T, typename... Args>
T adder(T first, Args... args) {
    cout << "Two Arg Method " << first << endl;
        return first + adder(args...);
}

int main()
{
    long sum = adder(1, 2, 3, 4, 5);
        cout << sum << endl;
     return 0;
}
```

VARIADIC METHOD – SAMPLE 2

This depicts the order of invocation and list reduction:

```
template<typename T, typename... Args> T adder(T first, Args... args) { … }
```

First : 1 Args: 2, 3, 4, 5

First : 2 Args: 3, 4, 5

First : 3 Args: 4, 5

First : 4 Args: 5

```
template<typename T>T adder(T v) { … }
```

v : 5

ANOTHER METHOD - SAMPLE

You can reduce the variadic list by multiples; not single items. Just add additional parameters to the variadic list.

In the adjacent code, the list is reduced two items at a time.

The compares two items at a time and then “and” && with the previous comparison. The result is then returned.

```
template<typename T>
bool pair_comparer(T a, T b) {
return a == b;
}

template<typename T, typename...Args>
bool pair_comparer(T a, T b, Args...args) {
return a == b && pair_comparer(args...);
}


int main() {

cout << pair_comparer(1, 1, 2, 2, 3, 3);
}
```

ANOTHER METHOD – SAMPLE 2

This depicts the order of invocation and list reduction:

```
template<typename T> bool pair_comparer(T a, T b, Args...args) { …
}
```

a : 1 b: 1 Args: 2, 2, 3, 3

a: 2 b: 2 Args: 3, 3

```
bool pair_comparer(T a, T b) { … }
```

a: 3 b: 3

POP QUIZ: WHAT DOES THIS MEAN?

In the previous example code, is this a problem?

```
pair_comparer(1, 1, 2, 2, 3, 3, 5);
```

10 MINUTES

VARIADIC TYPES

Unlike variadic methods, variadic types is an entirely new concept introduce in Modern C++.

Variadic types create a type with a variable number of fields.

In-depth discussion of variadic types is beyond the scope of this course.

VARIADIC CLASS TYPE

On the most basic level, variadic class types expand on the concept of variadic templated functions, which have variable number of parameters.

You use the class template to gather variable data for the functions.

In this example, the variadic class type is gathering data for the write_line method. The write_line method is a variadic templated function.

```
. . .
template<typename Stream, typename... Columns>
class CSVPrinter {
public:
    void output_line(const Columns&... columns) {
        write_line(columns...);
    }
    template<typename Value, typename... Values>
    void write_line(const Value &val, const Values&... values) {
        cout << val << endl;
        write_line(values...);
    }
    template<typename Value>
    void write_line(const Value &val) {
        cout << val << endl;
    }
};

int main() {
    CSVPrinter<int, int, int, int, std::string> obj;
    obj.output_line(1, 2, 3, "test");
}
```

VARIADIC CLASS TYPE

A variety of advanced concepts can be applied to variadic class types. The use of variadic class types unlocks virtually unlimited capability for creating custom classes, where the variadic class type is scaffolding for building a variety of actual classes.

You can templatize inheritance, the number of fields, and so much more.

```
template<class... __Policies>
class GenericPolicyAdapter : public __Policies... {
public:
    template<class... _Args>
        GenericPolicyAdapter(_Args... args) :
        __Policies(args...)...
    {}
};

struct T1 {
    T1(){}
    T1(int, int, int) {}
};

struct T2 {
    T2(){}
    T2(int, int, int) {}

};
int main() {
    GenericPolicyAdapter<T1, T2> gp1(1, 2, 3);
    GenericPolicyAdapter<T1> gp2(1, 3);

    return 0;
}
```

Threading

Asynchronous Function Calls

POP QUIZ: COMMON TERM: THREAD

What is a thread?

10 MINUTES

I AM A THREAD. WHAT AM I?

Asynchronous function call

- I have parameters
- I have a return value
- When the function exits, I am done.

I own a stack frame:

- Locals
- Parameters
- Return address
- so on

I own thread specific storage: Thread Local Storage (TLS)

If I own a window, I also have a GUI queue.

THREAD CONCEPTS

- Multi-Threading
- Preemptive Multi-Threading
- Parallelism
- Impact on performance
- Scheduling
- Process / Thread Priority
- Synchronization
- Maintainability

MULTI-THREADING VERSUS PARALLELISM

Multi-threading

- Responsiveness
- UI Thread
- Task specific allocation

Parallelism

- Multi-threading
- Performance
- Maximize multiple cores
- Thread Pools

THREAD SCHEDULING (WINDOWS)

Thread scheduling is a combination of process priority, thread priority, and round-robin preemptive scheduling.

Preemptive scheduling means each thread receives one or more quantums of execution; but can be preempted if a high priority thread starts.

A thread is preempted after completing the quantum(s). The schedule then looks for the next thread to schedule – either a higher priority thread or round-robin fashion.

Threads running over their base priority are eroded 1 priority when completing a quantum(s). Threads may also receive a priority boost for a variety of reasons.

POP QUIZ: THREAD SCHEDULING

How is thread scheduling handled in your environment?

15 MINUTES

THREAD SYNCHRONIZATION

- Use C++ or Platform SDK (C++ preferred)
- Coordinates activities of multiple threads
- Use hybrid synchronization
- Context switches
- Kernel versus User-mode synchronization
- Critical Sections
- Spinning
- Avoid if possible

MAINTAINABILITY

- Creating threads is easy
- Maintainable threads are much easier
  - Normal priority threads
  - No global variables
  - Limited if any synchronization
  - Using thread pool when possible
  - No dependences, such as file access.

CREATE A THREAD

- First – create a function
- Initialize a thread object
- Synchronize thread to prevent the thread from simply ending.

```
void FuncA() {
        cout << "Hello, world";
}

int main() {
        thread t1{ FuncA};
        t1.join();
        return 0;
}
```

POP QUIZ: LAMBDA FOR A THREAD

Replicate the code on the previous page. However, use a lambda instead of FuncA.

10 MINUTES

THREAD PARAMETERS

When using functions, thread parameters are passed in a comma separated list. For lambdas, use standard syntax: parameters or captured variables.

```
void FuncA(int a){
    cout << a;
}
int main() {
    int i = 2;
    thread t1{ FuncA, 4 };
    thread t2{ [=] {cout << i; } };
    t1.join();
    t2.join();
    return 0;
}
```

THREAD RETURN

Variety of techniques to return a value from a thread.

- Parameters that are references
- Wrap thread in a structure or class
- Globals convenient but bad

THREAD WITH STATE

Thread parameters allow minimal transfer of thread specific data. You can also use a class or structure to create a thread object. The data members provide rich information available to the threads.

```
class FortyTwo {
public:
     int vara = 42;
     void operator()() { cout << vara; }
};

int main() {
     thread t1{ FortyTwo() };
     t1.join();
     return 0;
}
```

POP QUIZ: THREAD RETURN VALUE

Based on the content presented in this module, how would you obtain a "return" value from a thread in a thread-safe manner?

5 MINUTES

LAB: FACTORIAL OR FIBONACCI

Create a thread that can calculate either a factorial or Fibonacci sequence.

Run both threads twice simultaneously. Display the results.

This lab is purposely vague.

30 Minutes

LESS THAN OR GREATER THAN - 2

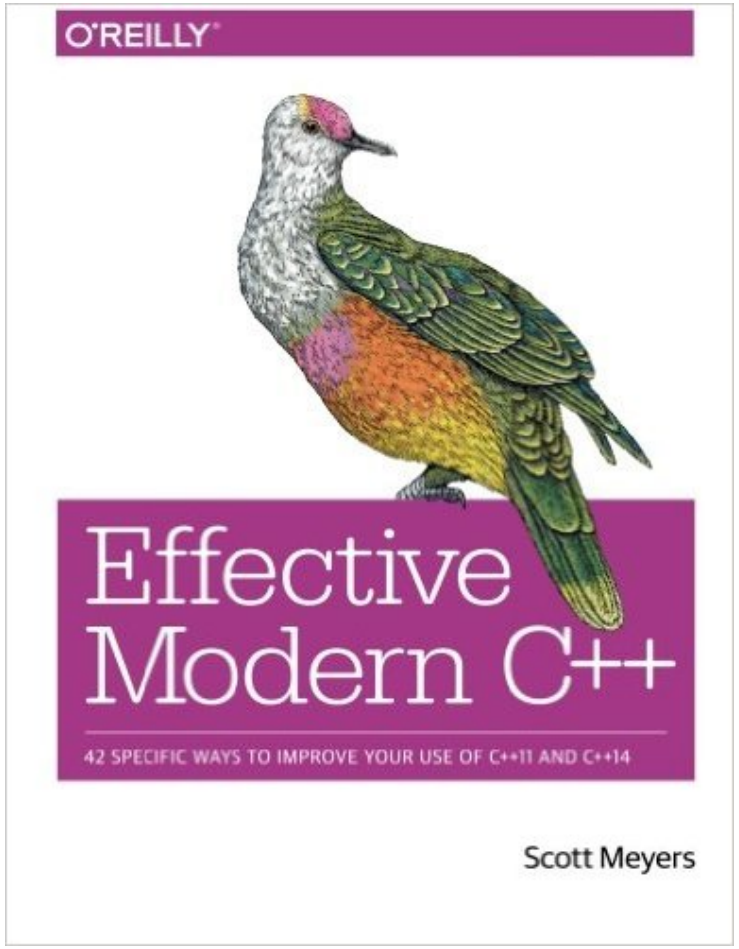
- FYI. The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, and so on.
- Create a thread that accepts a flag and parameter.
  - Based on the flag calculate either a Factorial or Fibonacci
  - For the Factorial function, calculate !parameter.
  - For the Fibonacci function, calculate and save Fibonacci numbers <= parameter.
- Run thread twice simultaneously: one for Factorial and one for Fibonacci.
  - For Factorial, calculate !5.
  - For Fibonacci, calculate sequence up to 55.
- After both threads done, display results.

Employ as much Modern C++ as possible.

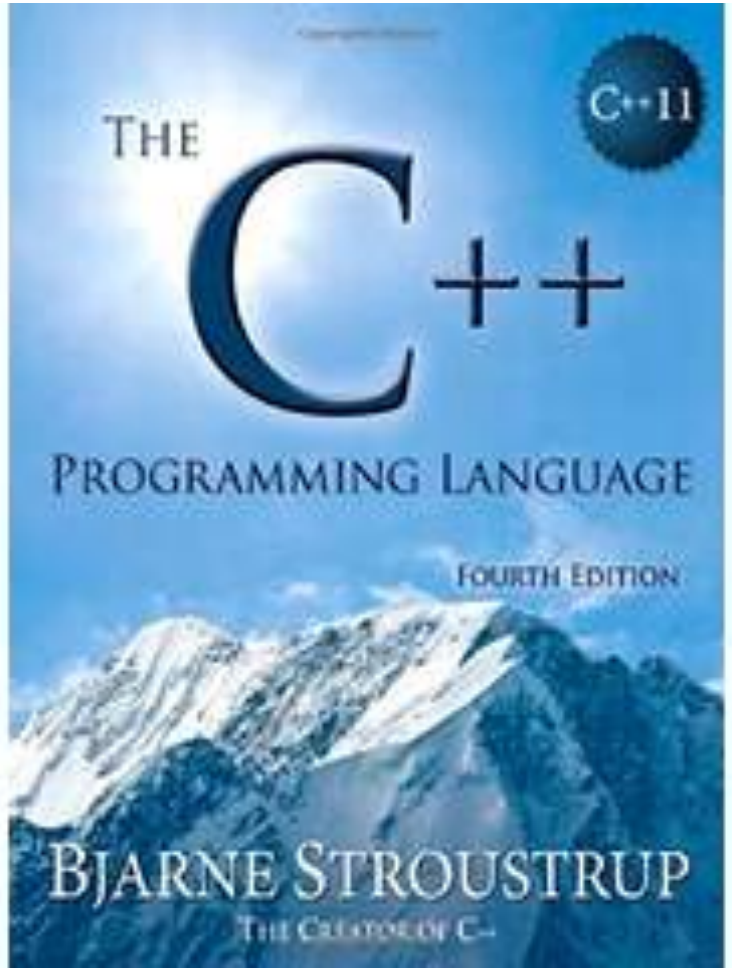
Reference content

Books and websites

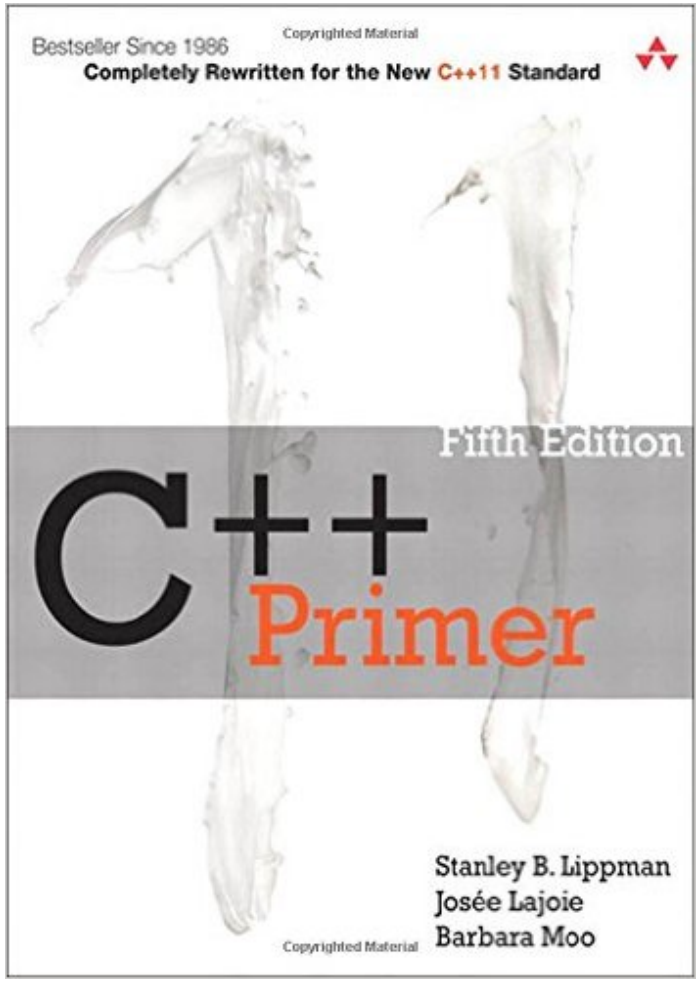
EFFECTIVE MODERN C++
SCOTT MEYERS



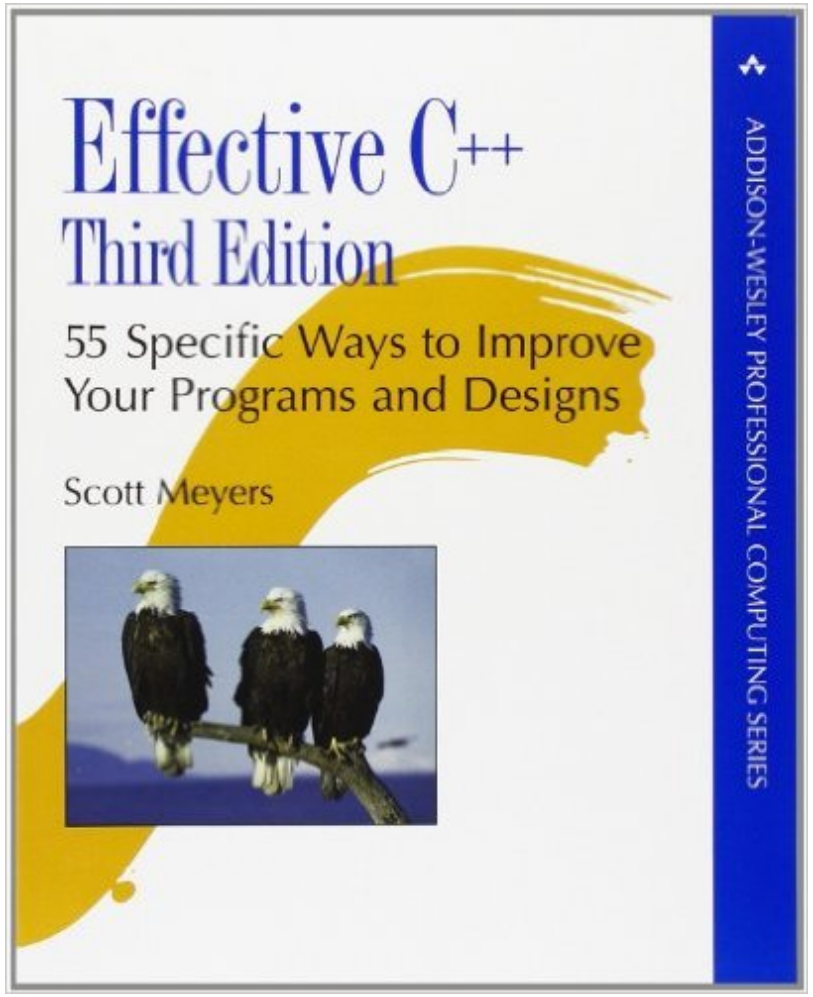
C++ PROGRAMMING LANGUAGE
BJARNE STROUSTRUP



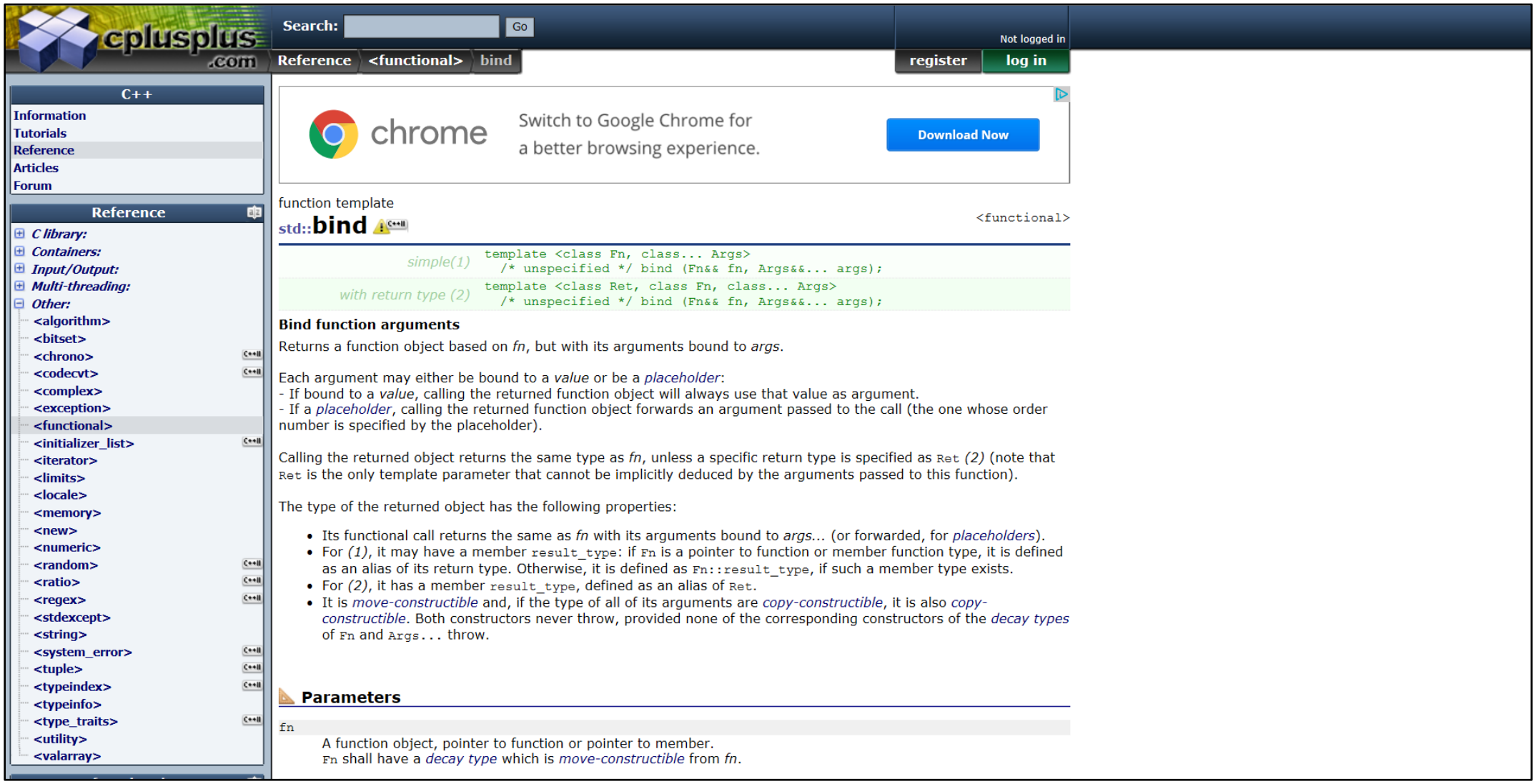
C++ PRIMER
STANLEY LIPPMAN



EFFECTIVE C++
SCOTT MYERS



CPLUSPLUS.COM



cplusplus.com

Search: Go

Not logged in register log in

Reference <functional> bind

C++
- Information
- Tutorials
- Reference
- Articles
- Forum

Reference
- C library:
- Containers:
- Input/Output:
- Multi-threading:
- Other:
  - <algorithm>
  - <bitset>
  - <chrono>
  - <codecvt>
  - <complex>
  - <exception>
  - <functional>
  - <initializer_list>
  - <iterator>
  - <limits>
  - <locale>
  - <memory>
  - <new>
  - <numeric>
  - <random>
  - <ratio>
  - <regex>
  - <stdexcept>
  - <string>
  - <system_error>
  - <tuple>
  - <typeindex>
  - <typeinfo>
  - <type_traits>
  - <utility>
  - <valarray>

chrome Switch to Google Chrome for a better browsing experience. Download Now

function template

std::bind <functional>

```
simple(1)          template <class Fn, class... Args>
                     /* unspecified */ bind (Fn&& fn, Args&&... args);
with return type (2) template <class Ret, class Fn, class... Args>
                     /* unspecified */ bind (Fn&& fn, Args&&... args);
```

Bind function arguments

Returns a function object based on *fn*, but with its arguments bound to *args*.

Each argument may either be bound to a *value* or be a *placeholder*:
- If bound to a *value*, calling the returned function object will always use that value as argument.
- If a *placeholder*, calling the returned function object forwards an argument passed to the call (the one whose order number is specified by the placeholder).

Calling the returned object returns the same type as *fn*, unless a specific return type is specified as `Ret` *(2)* (note that `Ret` is the only template parameter that cannot be implicitly deduced by the arguments passed to this function).

The type of the returned object has the following properties:

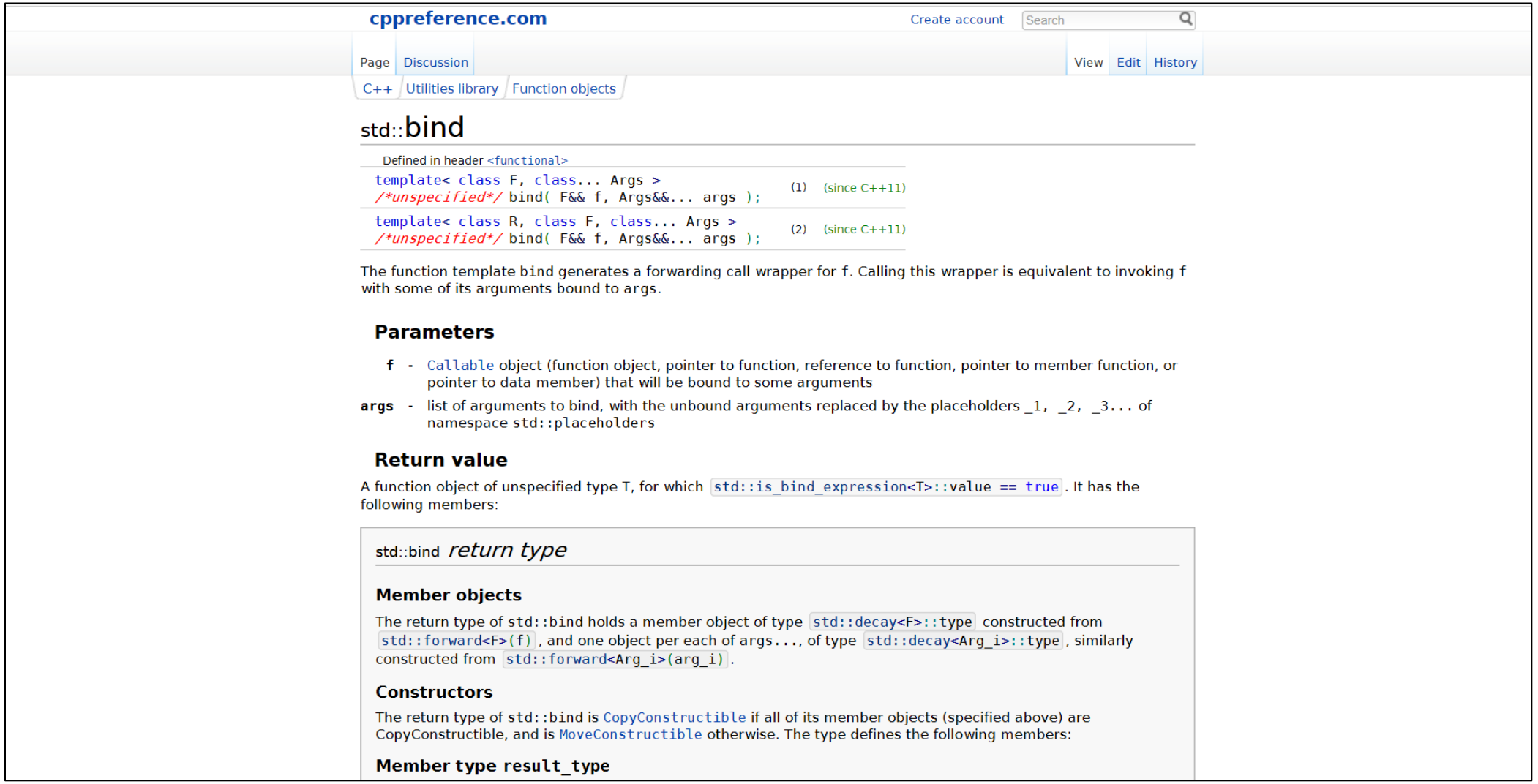
- Its functional call returns the same as *fn* with its arguments bound to *args...* (or forwarded, for *placeholders*).
- For *(1)*, it may have a member `result_type`: if `Fn` is a pointer to function or member function type, it is defined as an alias of its return type. Otherwise, it is defined as `Fn::result_type`, if such a member type exists.
- For *(2)*, it has a member `result_type`, defined as an alias of `Ret`.
- It is *move-constructible* and, if the type of all of its arguments are *copy-constructible*, it is also *copy-constructible*. Both constructors never throw, provided none of the corresponding constructors of the *decay types* of `Fn` and `Args...` throw.

Parameters

`fn`

A function object, pointer to function or pointer to member.
`Fn` shall have a *decay type* which is *move-constructible* from *fn*.

CPPREFERENCE.COM



cppreference.com

Create account | Search

Page | Discussion | View | Edit | History

C++ / Utilities library / Function objects

std::bind

Defined in header <functional>

```
template< class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );                (1) (since C++11)

template< class R, class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );                (2) (since C++11)
```

The function template bind generates a forwarding call wrapper for f. Calling this wrapper is equivalent to invoking f with some of its arguments bound to args.

Parameters

f - Callable object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments

args - list of arguments to bind, with the unbound arguments replaced by the placeholders _1, _2, _3... of namespace std::placeholders

Return value

A function object of unspecified type T, for which `std::is_bind_expression<T>::value == true`. It has the following members:

std::bind *return type*

Member objects

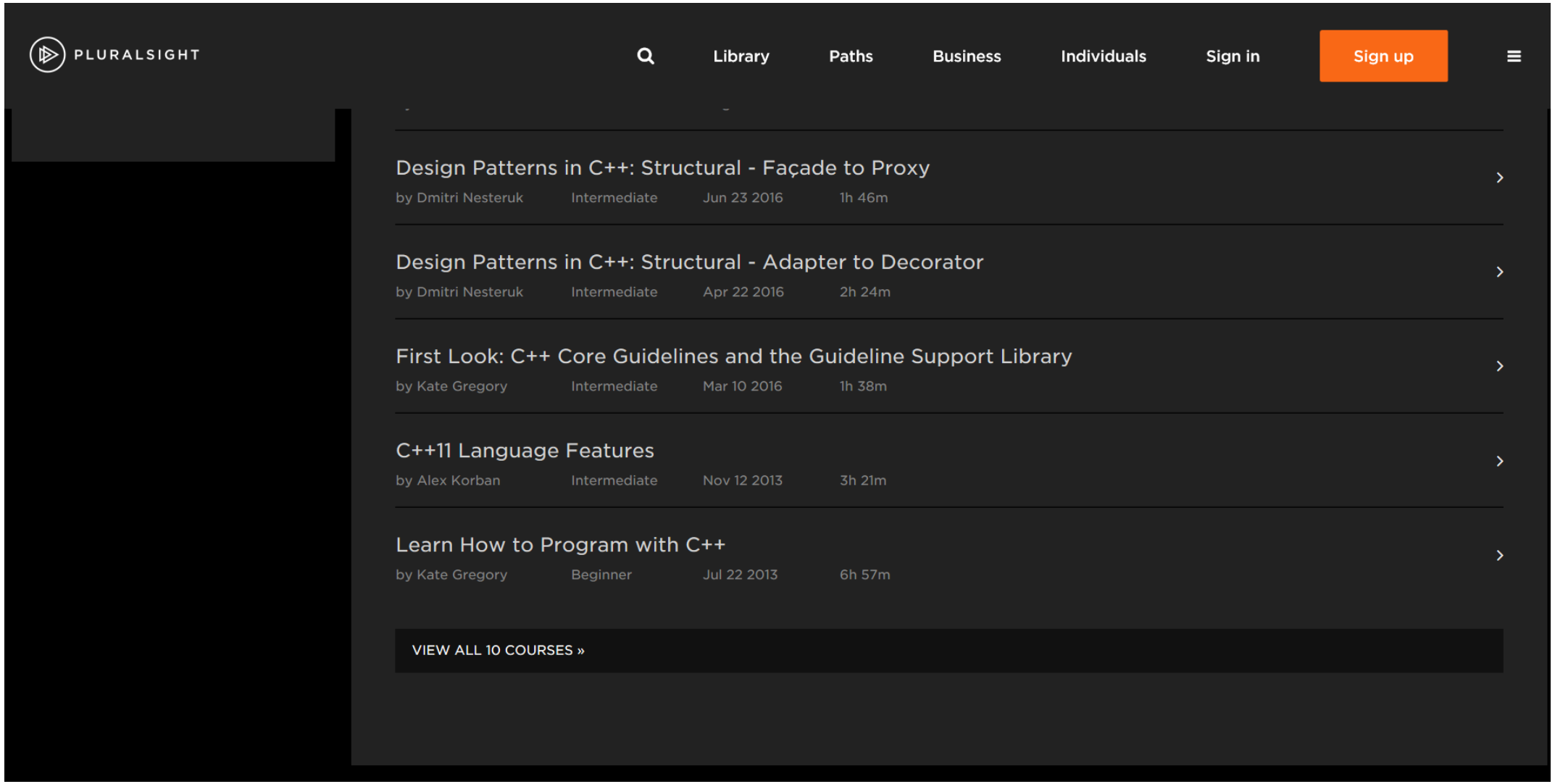
The return type of std::bind holds a member object of type `std::decay<F>::type` constructed from `std::forward<F>(f)`, and one object per each of args..., of type `std::decay<Arg_i>::type`, similarly constructed from `std::forward<Arg_i>(arg_i)`.

Constructors

The return type of std::bind is CopyConstructible if all of its member objects (specified above) are CopyConstructible, and is MoveConstructible otherwise. The type defines the following members:

Member type result_type

PLURALSIGHT.COM



Extras

Lots of stuff

OFFSETOF

#define offsetof(TYPE, MEMBER)
((size_t) &((TYPE *)0)->MEMBER)

- Get offset within user defined type
- Great for partial initialization
- Perfect for link lists

```
#include <cstddef>

struct Record {
     double a=5;
     int b=23;
     double c=34;
};

int main() {
     Record aRecord;
     memset(&aRecord, 0,
          offsetof(Record, b));
     return 0;
}
```

FUNCTION TRY BLOCK

Catch exceptions from initialization lists, destructors, and so on. Provides an opportunity to handle an exception in an initialization list in a orderly manner.

```
class XClass {
public:
XClass() try : obj(new YClass) {


}
catch(...)
{


}


YClass *obj;
};
```

TO_STRING

The to_string method was recently introduced and converts various types to std::string.

string to_string (int val);

string to_string (long val);

string to_string (long long val);

string to_string (unsigned val);

string to_string (unsigned long val);

string to_string (unsigned long long val);

string to_string (float val);

string to_string (double val);

string to_string (long double val);

```
int main(){
    string str = to_string(42);
    return 0;
}
```

PREDEFINED MACROS

These predefined macros are not new but nonetheless useful for diagnostics, debugging, and logging.

- __COUNTER__
- __DATE__
- __TIME__
- __FILE__
- __LINE__
- __func__

```
void example() {
     printf("%d\n", __COUNTER__);
     printf("%s\n", __func__);
     printf("%s\n", __FILE__);
     printf("%s\n", __DATE__);
     printf("%d\n", __LINE__);
     printf("%d\n", __COUNTER__);
}

int main(){
     example();
     printf("\n%s\n", __func__);
     printf("%d\n", __COUNTER__);
     return 0;
}
```

REF QUALIFIERS

Overload a method based on lvalue or rvalue object type.

```
class XClass {
    public:
    void FuncA() & {
        int a = 5;
        ++a;
    }

    void FuncA() && {
        int a = 5, b = 10;
        a /= b;
    }
};
```

OPTIONAL

The optional feature of Modern C++ indicates whether an object exists. If are returning an object for example, you can return the object or indicate the object does not exist for some reason.

This is a much better solution than using a arbitrary value, such as -1.

This requires C++ 17.

```
#include <optional>
#include <string>
#include <iostream>
using namespace std;


std::optional<unsigned> FuncA(bool b) {
	if (b) {
		return make_optional<int>(42);
	}
	else {
		return nullopt;
	}
}
int main() {
	std::optional<unsigned> opt = FuncA(true);
	if (opt.has_value()) {
	 cout << opt.value();
	}
}
```

STRING_VIEW

The string_view type is for immutable strings; similar to a const string.

This requires C++ 17.

```
#include <iostream>

using namespace std;



int main() {

std::string str = "lllloooonnnnggggg sssstttrrriiinnnggg";
//A really long string


//Bad way - 'string::substr' returns a new string
(expensive if the string is long)

std::cout << str.substr(15, 10) << '\n';


//Good way - No copies are created!

std::string_view view = str;


// string_view::substr returns a new string_view

std::cout << view.substr(15, 10) << '\n';

}
```

OPERATOR COMMA

Some surprising operators, such as the comma operator, can be overloaded. This can lead to some imaginative solutions.

Other available operators:

- operator &
- operator &&
- operator ||
- operator->

COMMA OPERATOR EXAMPLE CODE

```
class XInt {

public:

XInt(int _value) : value(_value)
{}


XInt operator ,(const XInt & _rhs)
{

return XInt(value + _rhs.value);

}

int value;

};
```

```
int main() {

XInt obj1(10), obj2(20), obj3(30);


obj1 = (obj1, obj2, obj3);


cout << obj1.value;

return 0;

}
```