

Lectures

- [Introduction to C and C++](#)
- [Object Oriented Basics](#)
- [Operators](#)
- [Templates](#)
- [STL Library](#)
- [Algorithms Library](#)
- [Object Oriented Advanced](#)
- [Smart Pointers](#)
- [Function Pointers & Lambdas](#)
- [Threading Library](#)

Introduction to C and C++

Simply put, C++ adds syntax sugar to make the code easier to write, and provides the benefits of abstraction.

Object Oriented Basics

Topics:

- Classes and objects
- Constructors and destructors
- Inheritance

Classes and Objects

Fundamentally, an object is a collect of data and methods that you can invoke on it. Let's create an object in C, and define APIs on it:

```
typedef struct {
    int age;
} person_s;

void person__set_age(person_s* pointer, int age) {
    pointer->age = age;
}

void c_usage() {
    person_s person;
    person__set_age(&person, 123);
    person.age = 456; // C has no concept of private/public}
```

The C API has the following limitations:

- `struct` members (ie: age), are "public"
- This means that anyone can access the members of `person_s`
- You manually have to create a function whose first parameter is `person_s *pointer` such that this function can modify the object in a mutable way

C++

In C++, the limitations are overcome, and the syntax becomes:

```
class person_class {
    int m_age;

    public:
        void set_age(int age) {
            m_age = age;
        }
};

void usage() {
    person_class person;
    person.set_age(123);

    // p.set_age(123) really resolves to:
    //person::set_age(&person, 123);
    // cannot access private
    //person.m_age = 123;}

```

In a way, C++ just adds a little syntax sugar to achieve the following:

- Class has default visibility of private, hence m_age (as in member variable age) is private
- Public API doesn't need the mutable pointer passed in, it is automatic
 - There is a hidden "this" pointer as a first parameter

Constructors

One of the severe limitations of C is that constructors are not automatic. Let's find out what that means:

```
class person_class {
    int m_age;

    public:

```

```

// "default" constructor
person_class() {
    std::cout << "Constructor of person_class has been called" << std::endl;
    m_age = 0;
}
void set_age(int age) {
    m_age = age;
}
};

```

Default constructor is one you get for "free", and you may not always need to define it, especially if the default constructor has empty code. However, if you specify another constructor with different parameters, default constructor is deleted.

```

class person_class {
    int m_age;

public:
    // constructor
    person_class(int age) {
        std::cout << "Constructor of person_class has been called" << std::endl;
        m_age = age;
    }
    void set_age(int age) {
        m_age = age;
    }
};

```

In the code above, this effectively yields this syntax:

```

class person_class {
    int m_age;

public:
    person_class() = delete;

    // constructor
    person_class(int age) {

```

```
std::cout << "Constructor of person_class has been called" << std::endl;
m_age = age;
}};
```

Destructors

Destructors are intuitively opposite of the constructors. Unlike a constructor when the function is invoked when the object is built, the destructor is called when the object goes out of scope and is thus destroyed.

```
class person_class {
    int m_age;

public:
    ~person_class() {
        std::cout << "Destructor of person_class has been called" << std::endl;
    }
};
```

Practical Example

```
class Vector {
private:
    int* m_array; // Pointer to dynamically allocated array
    int m_size;   // Size of the vector
public:
    // Constructor with size and default value
    Vector(int size) {
        m_size = size;
        m_array = new int[size];
        std::cout << "Vector constructor called. Size: " << size << std::endl;
    }
    // Destructor
    ~Vector() {
        delete[] m_array;
    }
};
```

```
std::cout << "Vector destructor called. Size: " << size << std::endl;
}};
```

Exercise 1

Let's put all the knowledge acquired so far towards an exercise. We will build a simpler version of the `std::vector`, or simply an integer array.

```
// file: vector.hh
class Vector {
private:
    int* m_array;    // Pointer to dynamically allocated array
    int m_max_size;  // Max size of the vector
    int m_size;      // Current size of the vector
public:
    // Constructor with max size
    Vector(int max_size);
    ~Vector();

    bool push_back(int value);
    int pop_back();

    int back();
    int front();

    int get_size();
    int get_max_size();

    void clear();};
```

Self-test framework

Ideally, we would create a unit-test framework, but to keep things simple, we can use the `assert()` API to provide a rudimentary unit-test framework.

```
#include <iostream>
#include <assert>
class Vector;
int main() {
    Vector v(5);

    assert(0 == v.get_size());
    assert(3 == v.get_max_size());
    assert(0 == v.pop_back());

    assert(true == v.push_back(123));
    assert(1 == v.get_size());
    assert(123 == v.pop_back());
    // ...
    return 0;}
```

Copy Problem

There is a problem with our current design of the vector. The code below has an issue; please compile and run the code and see what happens!

```
#include <iostream>
#include <assert>
class Vector;
int main() {
    Vector v1(10);
    // We want another vector with same properties as v1
    // Problem: We did not allocate new memory but are now referring to v1's memory
    Vector v2 = v1;
    return 0;}
```

std::unique Pointer to the Rescue

After gaining advanced C++ experience, you will align to the fact that there should never be "naked pointers" in C++. Pointers should always use more advanced pointers provided by the C++ 11 standard. If we had built our vector like this, we would have caught the problem at compile-time rather than run-time. Note that the code below is just a preview of what we will learn in the future, and this is not

required for the exercises as part of this article.

```
// file: vector.hh
#include <memory>

class Vector {
private:
    std::unique_ptr<int> m_array;    // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
    int m_size;      // Current size of the vector
public:
    // Constructor with max size
    Vector(int max_size);
    ~Vector();

    // ...};
```

Copy Constructor

The solution is that we need to "deep copy" the object which is called the copy constructor. Let's implement the copy constructor and see how it will work.

```
class Vector {
    // RULE: Whenever there is dynamic memory allocation (new operator)
    // There shall always be a copy constructor to perform "deep copy"
    Vector(const Vector& copy) {
        m_max_size = copy.m_max_size;
        // do not :
        //m_array_pointer = copy.m_array_pointer;
        m_array_pointer = new int[m_max_size]; // allocate your own memory, do not reference same memo
        std::cout << "Vector COPY constructor is called for size " << m_max_size << std::endl;
        // Deep copy
        for(int i = 0; i < m_max_size; i++) {
            m_array_pointer[i] = copy.m_array_pointer[i];
        }
    }
}
```

```
// ...};
```

Rule of 3

The "Rule of Three" in C++ refers to a guideline for defining three specific member functions when a class manages resources like **dynamic memory** (e.g., through pointers) to ensure proper behavior regarding copying and destruction. The three key member functions are:

1. **Destructor** (`~ClassName()`):

- The destructor is responsible for releasing resources (like dynamic memory) held by an object when it is destroyed.
- This is crucial to prevent memory leaks and properly clean up allocated resources.

2. **Copy Constructor** (`ClassName(const ClassName& other)`):

- The copy constructor creates a new object as a copy of an existing object.
- It is used when an object is initialized from another object of the same type (e.g., during object initialization, function parameter passing by value).

3. **Copy Assignment Operator** (`ClassName& operator=(const ClassName& other)`):

- The copy assignment operator defines how an existing object can be assigned the value of another object of the same type.
- It is invoked when you assign one object to another using the assignment operator `=`.

Sample Code for Rule of 3

```
class Vector {
    int *m_memory_for_integers;
    int m_max_size;
    int m_current_size;
    void deep_copy(const Vector& source) {
        // Deep copy: Copy each member from one vector to another
        for (int i=0; i < m_current_size; i++) {
            m_memory_for_integers[i] = source.m_memory_for_integers[i];
        }
    }
};
```

```

    }
}

public:
    // Fixed size vector that allocates memory once but cannot grow (by design)
    Vector(int max_size) {
        printf("Constructor is called to allocate %d integers\n", max_size);
        m_max_size = max_size;
        m_memory_for_integers = new int[m_max_size]; // Allocate memory dynamically
    }

    // Rule of 3: If dynamic memory, then:
    // - We must destructor
    // - Copy constructor
    // - Assignment operator
    // 1: Destructor
    ~Vector() {
        std::cout << ("Destructor is called") << std::endl;
        delete [] m_memory_for_integers;
    }

#ifdef 0 /* BUGGY CODE: */
    // You get trivial copy constructor for free:
    // But if you allocate dynamic memory, is this what you want? No!
    Vector(const Vector& source) {
        m_max_size = source.m_max_size;
        m_current_size = source.m_current_size;
        // BUGGY CODE:
        m_memory_for_integers = source.m_memory_for_integers; // THIS IS THE PROBLEM
    }
#endif

    // 2: Copy constructor
    Vector(const Vector& source) {
        printf("Copy constructor called\n");
        m_max_size = source.m_max_size;
        m_current_size = source.m_current_size;
        // THIS IS THE KEY: Allocate our own memory
        // We do not wish to copy memory reference of another object literally
        m_memory_for_integers = new int[m_max_size];
    }

```

```

    deep_copy(source);
}
// 3: Assignment operator
Vector& operator=(const Vector& source) {
    printf("Assignment Operator called\n");

    m_max_size = source.m_max_size;
    m_current_size = source.m_current_size;
    m_memory_for_integers = new int[m_max_size]; // THIS IS THE KEY
    deep_copy(source);
    return *this;
}
// Const APIs that do not modify the Vector instance
int get_size() const { return m_current_size; }
int get_max_size() const { return m_max_size; }
void print_memory_location_of_integers() const {
    printf("Memory allocated at\n");
    for(int i = 0; i < m_max_size; i++) {
        printf(" [%d] = %p\n", i, &m_memory_for_integers[i]);
    }
}
};

```

Exercise 2

Based on our learning so far, let us perform another exercise to create a string library.

```

#include <string.h>
#include <iostream>
// String library
class string {
    char *m_string;
    int m_max_length;

    // Bonus points if you use unique_pointer

```

```

// std::unique_ptr<char> m_string;
public:
    string(int max_length) {
        m_max_length = max_length + 1; // +1 for NULL termination
        m_string = new char[m_max_length];
    }
    // Rule of 3: Because we will allocate memory dynamically
    ~string() {
        delete [] m_string;
    };
    // Implement constructor to allocate as much memory as the c_string
    string(const char *c_string);
    // Rule of 3: 2) Implement the copy constructor
    string(const string& copy);
    // Rule of 3: 3) Implement the assignment operator
    string& operator=(const string& source);

    // Mutable API to make all characters lowercase or uppercase
    void to_upper();
    void to_lower();

    // Adding more data to the string
    void append_char(char c); // Append a char but only if string has memory available
    void append_string(const char *c_string); // Append another c_string only if string has memory available

    // Non-mutable APIs to check certain properties of the string
    bool equals_to(const char *c_string);
    bool contains(const char *c_string);
    bool begins_with(const char *c_string);
    int get_length();
    void print() { std::cout << "String is: '" << m_string << "'" << std::endl; }

    // Other mutable APIs
    void clear();
    void set(const char *string) {
        strncpy(m_string, string, m_max_length);
    }

```

```

        // All 3 lines do the same thing
        m_string[m_max_length - 1] = '\0';
        m_string[m_max_length - 1] = 0;
        //m_string[m_max_length - 1] = NULL;
        // We need the line(s) above because in case strncpy() ran out of space, it won't null terminate
        // "hello" -> 6 spaces
        // [0] = h, [1] = e, [2] = l, [3] = l, [4] = o, [5] = '\0';
    }
};

```

Copy Constructor

The string class you built above has the same problem for the deep copy. Therefore, you will need to create a copy constructor to be able to deep copy the string.

Pay close attention to this code:

```

class string {
    // Rule of 3: 1) Functional destructor to deallocate dynamically allocated memory
    ~string() {
        delete [] m_string;
    };
    // Rule of 3: 2) Implement the copy constructor
    string(const string& copy);
    // Rule of 3: 3) Implement the assignment operator
    string& operator=(const string& source);}

```

Header and Source File

<TODO: Template of header and source file>

Operators

There are different types of operators in C++. More detail can be studied [at this article](#).

Various Operators

1. Arithmetic

```
void arithmetic() {  
    int x = 0;  
  
    x = x + 1;  
    x = x - 1;  
    x = x * 2;  
    x = x / 3;  
    x = x % 2;  
  
    x++;  
    x--;} 
```

Bitwise Operators

```
void bitwise() {  
    int x = 0;  
  
    x = x | 0b0001;  
    x = x & 0b0000;  
    x = x ^ 0b0001;  
}  
void great_example_of_xor_operator() {
```

```
int a = 1;
int b = 0;

// Check if a and b are exclusive from each other
if ((a == 1 && b == 0) || (a == 0 && b == 1)) {
    // ...
}

// We can actually do this:
if (a xor b)
// or
if (a ^ b)}
```

2. Assignment

```
void assignment() {
    int x = 0;

    x = x + 3; // full form
    x += 3;    // shortcut
    x -= 3;
    x *= 3;
    x /= 3;
    x %= 3;
    x &= 3;
    x |= 3;
    x ^= 3;
    x >>= 3;}
```

3. Comparison

```
void comparison() {
    int x = 0;
    int y = 1;
```



```
if (x == y)
if (x != y)
if (x > y)
if (x < y)
if (x >= y)
if (x <= y)}
```

4. Logical

```
void logical() {
    int x = 0;
    int y = 1;

    if (x == 1 && y == 1)
    if (x == 1 || y == 1)
    if (! (x == 1 && y == 1) )}
```

Operator Overloading

The operators would be boring if they were only applied to integers as demonstrated in the examples above. We can actually inform the C++ compiler what operators should do for our classes. Let's reuse the Vector of integers we built before and define some interesting operators.

```
// file: vector.hh
class Vector {
private:
    int* m_array;    // Pointer to dynamically allocated array
    int m_max_size;  // Max size of the vector
    int m_size;      // Current size of the vector
public:
    Vector(int max_size);
    ~Vector();
```

```

    bool push_back(int value);
    int pop_back();

    Vector operator+=(const Vector& other) const; // operator +=
    Vector operator+(const Vector& other) const; // operator +
    bool operator==(const Vector& other) const; // operator ==
    bool operator!=(const Vector& other) const; // operator !=

    Vector operator*=(int multiply_with); // * operator to multiply all integers by a number

    // ...
};

// operator+ definition
Vector Vector::operator+=(const Vector& other) const {
    // allocate memory that can hold data from both vectors
    Vector result(this->m_max_size + other.m_max_size);

    for (int i = 0; i < m_size; ++i) {
        result.push_back(m_array[i]);
    }
    for (int i = 0; i < other.m_size; ++i) {
        result.push_back(other.m_array[i]);
    }

    return result;
}

Vector Vector::operator+(const Vector& other) const {
    Vector result(*this);
    result += other;
    return result;
}

// operator== definition
bool Vector::operator==(const Vector& other) const {
    const bool is_equal = true;
    if (m_size != other.m_size) {
        return !is_equal;
    }
}

```

```

        for (int i = 0; i < m_size; ++i) {
            if (m_array[i] != other.m_array[i]) {
                return !is_equal;
            }
        }
        return is_equal;
    }

// operator!= definition
bool Vector::operator!=(const Vector& other) const {
    return !(*this == other);
}

Vector operator*=(int multiply_with) {
    for (int i = 0; i < m_size; ++i) {
        m_array[i] *= multiply_with;
    }
    return *this;}

```

Here is how the operators may be used:

```

void vector_plus_operator_example() {
    puts("Let's practice strings");
    Vector v1(6);
    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(3);
    Vector v2(3);
    v1.push_back(4);
    v1.push_back(5);
    v1.push_back(6);
    // Use our operator to add contents of two vectors
    v1 = v1 + v2;
    v1.print();
}

void vector_multiply_operator_example() {
    Vector v1(6);
    v1.push_back(1);

```

```
v1.push_back(2);
v1.push_back(3);
// Multiply operator in action
v1 *= 5;
v1.print();}
```

Exercises

Vector library operators

Let's implement a few more operators for your vector library. Typically, the `[]` operator is implemented such that it returns a reference to one of the elements of the vector, but in our case, we will return a read-only value.

```
class Vector {
private:
    int* m_array;    // Pointer to dynamically allocated array
    int m_max_size;  // Max size of the vector
    int m_size;      // Current size of the vector
public:
    Vector(int max_size);
    ~Vector();

    // First implement an "at()" API
    // Return an element at a particular index
    int at(int index);

    // Secondly, implement the [] operator:
    const int operator[](int index);}
```

String library operators

Implement the following string operators, and also write unit-test code in `main.cpp` to test that the code you wrote actually functions correctly.

```

class string
{
    std::unique_ptr<char[]> m_string;
    int m_max_length;
public:
    string(int max_length);
    string(const char *c_string);

    // -----
    // Implement the following operators
    // -----

    // Adds two strings together
    string operator+=(const string& other) const;
    // "hello world" - "world" = "hello"
    string operator-=(const string& other) const;
    // "hello" * 2 = "hellohello"
    string operator*=(int how_many_times) const;
    // Implement shift operators to trim beginning or end of string
    // 0b1101 >> 1 ==> 0b0110
    // "hello" >>= 1 ==> "hell"
    // "hello" >>= 3 ==> "he"
    string operator>>=(int shift_right_by) const;
    // similar to python for slice operation
    string operator<<=(int shift_right_by) const;
    // All comparison operators are applicable
    // string s1("hello"); string s2("world")
    // if (s1 == s2)
    // if (s1 != s2)
    bool operator!=(const string &compare_with) const {
        return !(*this == compare_with);
    }
    bool operator==(const string &compare_with) const {
        // todo
        return false;
    }
}

```

```
// Implement comparison operators
bool operator<=(const string &compare_with) const;
bool operator>=(const string &compare_with) const;
bool operator<(const string &compare_with) const;
bool operator>(const string &compare_with) const;
bool operator!=(const char* compare_with) const;
bool operator==(const char* compare_with) const;
}
```

Templates

Templates and the need for header only code

In one of our [previous lessons](#), we built our own "vector", but it was specifically designed to only hold integers. But what if we wanted to store `float`, or `char`, or `bool`? This can be accomplished by using templates in C++.

Without a template

```
// file: vector.hh
class Vector {
private:
    int* m_array;    // Pointer to dynamically allocated array
    int m_max_size;  // Max size of the vector
    int m_size;      // Current size of the vector
public:
    Vector(int max_size);
    ~Vector();

    bool push_back(int value);
    int pop_back();

    // ...};
```

With a template

```
// file: vector.hh
template <typename your_type>
class Vector {
private:
```

```

    your_type* m_array;    // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
    int m_size;        // Current size of the vector
public:
    Vector(int max_size);
    ~Vector();

    bool push_back(your_type value);
    your_type pop_back();

    // ...};

```

Here is how you would use the code and have the C++ compiler multiply your code for different types:

```

int usage() {
    Vector<int> my_int_vector_v2(1);
    Vector<char> my_char_vector_v2(1);
    Vector<float> my_float_vector_v2(1);}

```

Sample 1

When you design your header file, especially with a template, your code can be input right within the class itself.

```

// file: vector.hh
template <typename your_type>
class Vector {
private:
    your_type* m_array;    // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
    int m_size;        // Current size of the vector
public:
    // code for function can be right here
    Vector(int max_size) {
        m_array = new int[max_size];
    }

```



```

    m_max_size = max_size;
    m_size = 0;
}

void push_back(your_type value) {
    if (m_size < m_max_size) {
        m_array[m_size] = value;
        m_size++;
    }
}

// ...};

```

Sample 2

```

// file: vector.hh
template <typename your_type>
class Vector {
private:
    your_type* m_array;    // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
    int m_size;      // Current size of the vector
public:
    // we can declare functions but "define" them below
    // this improves code readability
    Vector(int max_size);

    void push_back(your_type value);

    // ...
};

// We must use "scope" operator to define which class the function belongs to
Vector::Vector(int max_size) {
    m_array = new int[max_size];
    m_max_size = max_size;
}

```

```
m_size = 0;
}
void Vector::push_back(your_type value) {
    if (m_size < m_max_size) {
        m_array[m_size] = value;
        m_size++;
    }
}
```

Source Code Organization

Template code in header file only

The way templates work is that for each type, such as `vector<int>` or `vector<bool>`, the entire header file is copied and pasted by the compiler for the new type. Because of this reason, classes that use templates must be in header file only. This means that you cannot have `vector.hh` and also `vector.cc` because the entire code has to exist in the header file only.

Standard code

Standard code that doesn't use templates can be in `file.hh` and also `file.cc` and doesn't need to be in a header. Although not that many libraries sometimes tend to be header only and the code split to `file.cc` is for cosmetic reasons only.

```
// Header file:
// File: adder.hh
#pragma once
// Example of header file and source file
// Class should only declare functions, but not define them
class adder {
public:
    int x;
    int y;
    // Only declare functions, do not "define" functions
    int get_sum();};
```

And then there should be a separate *.cc file or *.cpp file:

```
// File: adder.cc
#include "adder.hh"
int adder::get_sum() {
    return x + y;
}
```

STL Library

Before you read about the STL library, it is important to understand the [Templates](#), so ensure that you have covered that section before you start here.

Various Containers

There are different types of containers available in the STL library:

1. Sequence containers

array

An STL array is a fixed-size array, much like `int array[5]`. The difference is that the STL array is more of a "first class citizen" in terms of providing APIs on this array which are iterators, and other accessors such as `size()`, `front()` and `back()`.

```
#include <iostream>
#include <array>
int main() {
    // Create and initialize an std::array with 5 integers
    std::array<int, 5> numbers = {1, 2, 3, 4, 5};
    // Accessing elements using operator[]
    std::cout << "The first element is: " << numbers[0] << std::endl;
    // Accessing elements using at() with bounds checking
    try {
        std::cout << "Element at index 4 is: " << numbers.at(4) << std::endl;
        // This will throw an exception if the index is out of bounds
        std::cout << "Element at index 5 is: " << numbers.at(5) << std::endl;
    }
```

```

    } catch (std::out_of_range& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }
    // Iterating over elements using a range-based for loop
    std::cout << "All elements: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    return 0;}

```

Further examples

```

#include <iostream>
#include <array>
#include <algorithm>
void c_array() {
    // Standard C int array doesn't offer any APIs
    int int_array[] = {1, 2, 3, 4, 5};
    int_array[5] = 123; /* Uncaught memory exception */
}
void cpp_array__access_api() {
    // Better thing to do this:
    std::array<int, 5> cpp_int_array{1, 2, 3, 4, 5};
    // std::array offers some useful APIs
    cpp_int_array.at(0);
    cpp_int_array[1] = 22;
    //cpp_int_array[5] = 123; /* [] operator will not catch memory exception */
    //cpp_int_array.at(5);    /* C++ library catches memory exception */
    cpp_int_array.front() = 11; // Same as index[0] or .at(0)
    //cpp_int_array.end() = 67890;    // End is actually the first element out of bound
}
void cpp_array__iterate_api() {
    std::array<int, 5> cpp_int_array{1, 2, 3, 4, 5};
    // Let us iterate through each element of the array:

```

```

for (int element_iterator : cpp_int_array) {
    std::cout << "Element value: " << element_iterator << std::endl;
}
// The loop above is the same as the following:
for (auto it = cpp_int_array.begin();
     it < cpp_int_array.end(); /* end() is the first element address out of bound */
     it++) { /* end() is thus an element address following the last element of the array */
    std::cout << "Element value using iterator: " << *it << std::endl;
}
}

void cpp_array__other_apis() {
    std::array<int, 5> cpp_int_array;
    cpp_int_array.fill(123);
    for (int element_iterator : cpp_int_array) {
        std::cout << "Element value: " << element_iterator << std::endl;
    }
}

void cpp_array__powerful_things_about_containers_with_iterators() {
    std::array<int, 5> arr {10, 50, 4, -5, -100};
    std::sort(arr.begin(), arr.end());
    for (int element_iterator : arr) {
        std::cout << "Element value: " << element_iterator << std::endl;
    }
}

```

Inconvenience of std::array

The usage of std::array becomes inconvenient when we have to pass it as parameters to a function because not only your code has to account for the type in the template, but also the size. The problem is that if and when we change the size of the array, then we have to change it in a number of places.

```

#include <iostream>
#include <array>

// Function that modifies the array elements
void modify_array(std::array<int, 5>& arr) {
    for (int& num : arr) {
        num *= 2; // Double each element
    }
}

```

```

    }
}
// Function that prints the array elements (passed by const reference to prevent modification)
void print_array(const std::array<int, 5>& arr) {
    std::cout << "Array elements: ";
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
int main() {
    // Create and initialize an std::array with 5 integers
    std::array<int, 5> numbers = {1, 2, 3, 4, 5};
    // Print the original array
    print_array(numbers);
    // Modify the array elements
    modify_array(numbers);
    return 0;}

```

We can improve the code above utilizing the "using" keyword, but functions still get bound to an array of a fixed size, and functions are not reusable for other types of arrays.

```

#include <iostream>
#include <array>
// Array size is fixed, and we could use this statement to improve our code
using fixed_size_array = std::array<int, 3>;
// Function that modifies the array elements
void modify_array(fixed_size_array& arr) {
    for (int& num : arr) {
        num *= 2; // Double each element
    }
}
// Function that prints the array elements (passed by const reference to prevent modification)
void print_array(const fixed_size_array& arr) {
    std::cout << "Array elements: ";
    for (int num : arr) {

```

```

        std::cout << num << " ";
    }
    std::cout << std::endl;
}

void yet_another_problem_with_aliased_type() {
    std::array<int, 10> another_array = { 1, 2, 3, 4, 5, };
    /* This won't work because print_array() is bound to array of 3 integers only */
    print_array(another_array);
}

int main() {
    // Create and initialize an std::array with 5 integers
    fixed_size_array numbers = {1, 2, 3};
    // Print the original array
    print_array(numbers);
    // Modify the array elements
    modify_array(numbers);
    return 0;
}

```

vector

A vector is a more flexible container than an `std::array` because it can dynamically grow (or shrink). Let us practice a code snippet that uses various APIs that this class provides.

A vector is simply an array that can grow as needed. Try out the following code:

```

#include <iostream>
#include <vector>
int main() {
    std::vector<int> v;
    v.reserve(10);
    for (int i = 0; i < 100; i++) {
        std::vector<int>::iterator before = v.begin();
        v.push_back(i);
        auto after = v.begin();
        if (before != after) {

```



```

        printf("Memory got re-allocated when we tried to insert element #%d\n", (i+1));
    }
}
return 0;
}

```

Here are more examples of a vector with more APIs that it offers:

```

#include <iostream>
#include <vector>
int main() {
    // Create a vector of integers
    std::vector<int> vec;
    // Check if the vector is initially empty
    std::cout << "Initially, is vector empty? " << (vec.empty() ? "Yes" : "No") << std::endl;
    // Add elements to the vector
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    // Size and capacity after adding elements
    std::cout << "Size after adding 3 elements: " << vec.size() << std::endl;
    std::cout << "Capacity after adding 3 elements: " << vec.capacity() << std::endl;
    // Print current elements in the vector
    std::cout << "Current elements in vector: ";
    for (int i : vec) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
    return 0;}

```

A vector is definitely more powerful than `std::array`, and unlike the `std::array`, you can pass the vector to functions more conveniently. Let us modify our earlier example to a vector and see the difference.

```

#include <iostream>
#include <vector>
// Function that modifies the array elements

```

```

void modify_array(std::vector<int>& arr) {
    for (int& num : arr) {
        num *= 2; // Double each element
    }
}

// Function that prints the array elements (passed by const reference to prevent modification)
void print_array(const std::vector<int>& arr) {
    std::cout << "Array elements: ";
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

int main() {
    // Create and initialize an std::array with 5 integers
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    // Print the original array
    print_array(numbers);
    // Modify the array elements
    modify_array(numbers);
    return 0;}

```

deque

Double Ended Queue is pronounced as "deck" (ue is missing in deque), and it works similar to a vector, except that we can insert data at the beginning or at the end meaning that there is a `push_front()` as well as `push_back()`. One key difference is that a `<vector>` maintains contiguous memory access to all its elements. If we have a pointer to the first element, we can offset the pointer by N to get to the Nth element. In other words, we have $O(1)$ access to any of the element of the vector. Dequeue also has $O(1)$ access to any of its elements through a complex `[]` operator function, however, its data is in series of chunks or blocks of contiguous memory, **but not one large contiguous memory** like a vector.

```

int main() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
}

```

```

v.push_back(3);
int *pointer_to_first_element = v.data(); // data returns memory reference/pointer to first memory of vector
pointer_to_first_element[0] = 11;
pointer_to_first_element[1] = 22;
pointer_to_first_element[2] = 33;
for(size_t i = 0; i < v.size(); i++) {
    std::cout << "[" << i << "] = " << v[i] << std::endl;
}
return 0;}

```

The other key difference is insertion operation. When a vector runs out of memory, new memory is allocated and data is copied to the new memory. However, a deque can not only grow in both directions (front and back), but when the container runs out of memory, it can grow the memory without incurring the full cost of memory allocation or copying data from old memory to new memory.

The following APIs are in addition to what a vector would provide:

- `emplace_front()`
- `push_front()`
- `pop_front()`

However, the following APIs are lacking because they are not needed based on the implementation of the deque:

- `capacity()`
- `reserve()`

forward_list

Forward list is a singly linked list; it has "forward" links to its elements and only forward links and not backward links of a `<list>` container. It's operation is actually very similar to the `<list>`, however, because it is a singly linked-list, the "end()" iterator is lacking because one has to iterate through the beginning of the list to reach the end of the list. Hence, you will find these APIs missing from this container:

- `back()`
- `emplace_back()`
- `push_back()`
- `pop_back()`

list

List is a doubly linked list. API wise, we have all the functionality of a vector, however because it is a linked-list, the following APIs are lacking:

- operator[]
- at()

2. Associative containers

set and multiset

`set` is a container that is meant to hold sorted and unique elements. For example, an array of unique integers. It's job is to be able to hold your objects in a sorted manner.

`multiset` is very similar to a set with the exception that you can have duplicate members, such as duplicate integers `{1, 1, 2, 2, 3, 4}`.

```
#include <iostream>
#include <set>
int main() {
    // Create a set of integers
    std::set<int> numbers;
    // Insert elements
    numbers.insert(10);
    numbers.insert(40);
    numbers.insert(30);
    numbers.insert(20);
    numbers.insert(10); // This will not be added again, as sets do not allow duplicates
    // Print the elements of the set
    std::cout << "Elements of the set: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";
}
```

```
return 0;}
```

Practical Example

```
#include <iostream>
#include <set>
#include <string>
class Book {
public:
    std::string title;
    std::string author;
    Book(const std::string& title, const std::string& author)
        : title(title), author(author), year(year) {}
    // Define the operator '<' that will be used by the 'std::set' to
    // insert elements in a sorted order
    bool operator<(const Book& other) const {
        return title < other.title;
    }
};
int main() {
    std::set<Book> library;
    library.insert(Book("1984", "George Orwell"));
    library.insert(Book("The Great Gatsby", "F. Scott Fitzgerald"));

    // A duplicate entry will not be inserted:
    library.insert(Book("1984", "George Orwell"));
    std::cout << "Books in the library:\n";
    for (const auto& book : library) {
        std::cout << book.title << " by " << book.author << "\n";
    }
    return 0;}
```

map and multimap

map is a container that holds data mapped to a key. For example, a map of integers (key) mapped to a string (value). Hence, it is essentially a key/value map. The keys must be unique that map to their

corresponding values.

```
#include <iostream>
#include <map>
#include <string>
class Book {
public:
    std::string title;
    std::string author;
    Book(const std::string& title, const std::string& author)
        : title(title), author(author) {}
    // Unlike the previous example with std::set, we do not need < operator
    // because the map is sorted by the "key" which in this case is the book title (std::string)
};
int main() {
    std::map<std::string, Book> library;
    // Insert books into the library; the key is the book title, and the value is the Book instance
    library.insert(std::make_pair("1984", Book("1984", "George Orwell")));
    library.insert(std::make_pair("The Great Gatsby", Book("The Great Gatsby", "F. Scott Fitzgerald")));

    // Here are other ways to insert a "pair" (although since it is a map, duplicate entries won't be .
    library.insert({"1984", Book("1984", "George Orwell")});
    library.insert(std::make_pair("1984", Book("1984", "George Orwell")));
    library.insert(std::pair<std::string, Book>("1984", Book("1984", "George Orwell")));
    std::cout << "Books in the library:\n";
    for (const auto& entry : library) {
        const auto& book = entry.second;
        std::cout << book.title << " by " << book.author << "\n";
    }
    return 0;
}
```

Exercise

Here is the code structure of a "Bookstore" class. Note that the inventory of books is a private member, and we are exposing public APIs to be able to manage the Bookstore class.

```

#include <iostream>
#include <map>
#include <string>
class Bookstore {
private:
    std::multimap<std::string, int> inventory;
public:
    // Add or update books in the inventory
    void add_or_update(const std::string& title, int quantity) {
        inventory[title] += quantity; // Adds quantity to existing or initializes with quantity if not
    }
    // Remove a book from inventory
    bool remove_book(const std::string& title) {
        auto it = inventory.find(title);
        if (it != inventory.end()) {
            inventory.erase(it);
            return true;
        }
        return false;
    }
    // Check the stock of a specific book
    bool is_book_present(const std::string& title) const {
        auto it = inventory.find(title);
        return (it != inventory.end());
    }
    // Print the current inventory
    void print_inventory() const {
        std::cout << "Current Inventory:\n";
        for (const auto& pair : inventory) {
            std::cout << "Title: " << pair.first << ", Quantity: " << pair.second << "\n";
        }
    }
};

```

For this exercise, please build a menu system to complete the following code:

```

int main() {
    Bookstore store;
    int choice;

    do {
        std::cout << "\nBookstore Inventory Management:\n";
        std::cout << "1. Add/Update Book\n";
        std::cout << "2. Remove Book\n";
        std::cout << "3. Print Inventory\n";
        std::cout << "4. Check Stock\n";
        std::cout << "5. Exit\n";
        std::cout << "Enter choice: ";
        std::cin >> choice;
        switch (choice) {
            case 1:
                // Implement adding/updating a book
                break;
            case 2:
                // Implement removing a book
                break;
            case 3:
                store.print_inventory();
                break;
            case 4:
                // Implement checking stock
                break;
            case 5:
                std::cout << "Exiting...\n";
                break;
            default:
                std::cout << "Invalid choice. Please try again.\n";
        }
    } while (choice != 5);
    return 0;}

```


3. Unordered associative containers

The unordered associative containers are very similar to ordered ones; here are the differences:

- Ordered containers store data in sorted order
- Unordered containers store data using hash memory

Pseudo implementation of hash

```
#include <iostream>
#include <memory>
#include <string>
template <typename type>
class unordered_set {
    const size_t bucket_size = 100;
    std::unique_ptr<type> elements[100];
public:
    void insert(const type& key) {
        const auto hashed_value = std::hash<type>{}(key);
        const auto mapped_index = hashed_value % bucket_size;
        std::cout << "Hashed value: " << hashed_value << std::endl;
        std::cout << "Mapped index: " << mapped_index << std::endl;
        // This mapped index is empty, hence add the element here
        if (elements[mapped_index] == nullptr) {
            std::cout << "Adding a new element to the set: " << key << std::endl;
            elements[mapped_index] = std::make_unique<type>(key);
        }
        // There is an existing key mapped already
        else {
            std::cout << "Mapped index (" << key << ") already exists" << std::endl;
            // Is there an existing key already?
            if (*elements[mapped_index] == key) {
                std::cout << "Duplicate element, will not add it" << std::endl;
            }
            else {
```

```

        std::cout << *elements[mapped_index] << " and "
                << key <<
                " share the same mapped memory" << std::endl;

        // TODO: Handle this case
    }
}
};

void line() { std::cout << "-----" << std::endl; }
void test_string() {
    unordered_set<std::string> set;
    set.insert("hello"); line();
    set.insert("world"); line();
    set.insert("world"); line();
}

void test_int() {
    unordered_set<int> set;
    set.insert(123); line();
    set.insert(456); line();
    set.insert(456); line();
    set.insert(223); line();
}

int main(int argc, char **argv) {
    std::cout << "Hello world!" << std::endl;
    test_string();
    test_int();
    return 0;
}

```

The code above doesn't handle the case when we have "collisions". The collisions occur when there aren't enough buckets or containers or when multiple items by chance happen to map to the same memory. In this case, we have to change our unique pointer of elements to a vector of elements.

```

#include <iostream>
#include <vector>

```

```

#include <algorithm>
template <typename type>
class unordered_set {
    const size_t bucket_size = 100;
    // When "collisions" occur, we need to be able to store more than one element at the "mapped memory"
    // Therefore, let us change the single element storage to multiple element storage
    // std::unique_ptr<type> elements[100];
    std::vector<type> elements[100];
public:
    void insert(const type& key) {
        const auto hashed_value = std::hash<type>{}(key);
        const auto mapped_index = hashed_value % bucket_size;
        std::cout << "Hashed value: " << hashed_value << std::endl;
        std::cout << "Mapped index: " << mapped_index << std::endl;
        // This mapped index is empty, hence add the element here
        auto& mapped_container = elements[mapped_index];
        if (mapped_container.size() == 0) {
            std::cout << "Adding a new element to the set: " << key << std::endl;
            mapped_container.push_back(key);
        }
        // There is an existing key mapped already
        else {
            std::cout << "Mapped entry (" << key << ") already exists" << std::endl;
            // Is there an existing key already?
            const auto& found_entry = std::find(mapped_container.begin(), mapped_container.end(), key);
            if (found_entry != mapped_container.end() ) {
                std::cout << "Duplicate entry, will not add it" << std::endl;
            }
            else {
                std::cout << key << "has to share the same mapped memory" << std::endl;
                mapped_container.push_back(key);
            }
        }
    }
};

void line() { std::cout << "-----" << std::endl; }
void test_string() {

```

```

unordered_set<std::string> set;
set.insert("hello"); line();
set.insert("world"); line();
set.insert("world"); line();
}

void test_int() {
    unordered_set<int> set;
    set.insert(123); line();
    set.insert(456); line();
    set.insert(456); line();
    set.insert(223); line();
}

int main(int argc, char **argv) {
    test_string();
    test_int();
    return 0;
}

```

Now that you have a deeper understanding of hashing and memory storage (STL calls it "buckets"), please try the following program and make sense out of the output.

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_set>

void test_int() {
    std::unordered_multiset<int> set;
    set.insert(123);
    set.insert(456);
    set.insert(456); // duplicate
    for(int i = 0; i < 30; i++) {
        set.insert(i);
    }

    std::cout << "bucket count      : " << set.bucket_count() << std::endl;
    std::cout << "max_bucket_count  : " << set.max_bucket_count() << std::endl;
    std::cout << "bucket_size of 123: " << set.bucket_size(123) << std::endl;
}

```

```

std::cout << "bucket_size of 456: " << set.bucket_size(123) << std::endl;
std::cout << "bucket of 123      : " << set.bucket(123) << std::endl;
std::cout << "bucket of 456      : " << set.bucket(456) << std::endl;
std::cout << "load_factor        : " << set.load_factor() << std::endl;
std::cout << "max_load_factor    : " << set.max_load_factor() << std::endl;
}

int main(int argc, char **argv) {
    test_int();
    return 0;}

```

Exercise

Please extend the `unordered_set` class by adding a new method that would print all elements in the set:

- `void unordered_set::print_all_elements()`

Note that some containers may be empty vectors, and you would have to go through each container and print the data if the container is not empty.

unordered_set and unordered_multiset

Similar to `std::set` and `std::multiset`, these containers provide the ability to store unique elements for `std::unordered_set` and possibly duplicate elements also using the `std::unordered_multiset`. In fact, the same example from our previous section would work if we merely change the container type.

One key difference would be that when the `std::set` is iterated for each element, it would print the elements in a sorted order. That is because fundamentally the data is stored in a sorted data structure inside. The "unordered" as its name implies stores data in a bit arbitrary way and elements are not sorted. Therefore, the data is iterated in a random order.

```

#include <iostream>
// #include <set>
#include <unordered_set>

int main() {
    // Create a set of integers

```

```

//std::set<int> numbers;
std::unordered_set<int> numbers;
// Insert elements
numbers.insert(10);
numbers.insert(40);
numbers.insert(30);
numbers.insert(20);
numbers.insert(10); // This will not be added again, as sets do not allow duplicates
// Print the elements of the set
std::cout << "Elements of the set: ";
for (int num : numbers) {
    std::cout << num << " ";
}
std::cout << "\n";
return 0;}

```

unordered_map and unordered_multimap

The "map" version of the unordered container is obviously a map, and not just a collection of keys. For the `std::unordered_map`, there is a unique key and only one mapped value to the key. The `multimap` version offers the ability to store multiple values mapped to a single key.

Let us modify our code to enable it for key/value type.

```

#include <iostream>
#include <vector>
#include <algorithm>
template <typename KeyType, typename ValueType>
class unordered_map {
    const size_t bucket_size = 100;
    // Use a vector of vector of pairs to handle collisions and store key-value pairs.
    std::vector<std::pair<KeyType, ValueType>> elements[100];
public:
    void insert(const KeyType& key, const ValueType& value) {
        const auto hashed_value = std::hash<KeyType>{}(key);
        const auto mapped_index = hashed_value % bucket_size;
    }
};

```

```

std::cout << "Hashed value: " << hashed_value << std::endl;
std::cout << "Mapped index: " << mapped_index << std::endl;
auto& mapped_container = elements[mapped_index];
bool key_exists = false;
// The key maps to this container, but multiple keys could be stored here due to collisions
// Hence, search the container to check if our key exists
for (auto& elem : mapped_container) {
    if (elem.first == key) {
        // Key exists, update value
        std::cout << "Key (" << key << ") exists, updating value to " << value << std::endl;
        elem.second = value;
        key_exists = true;
        break;
    }
}
if (!key_exists) {
    // Key does not exist, add new key-value pair
    std::cout << "Adding new key-value pair to the map: " << key << ": " << value << std::endl;
    mapped_container.emplace_back(key, value);
}
}

};

void line() { std::cout << "-----" << std::endl; }

void test_string() {
    unordered_map<std::string, int> map;
    map.insert("hello", 1); line();
    map.insert("world", 2); line();
    map.insert("world", 3); line();
}

void test_int() {
    unordered_map<int, std::string> map;
    map.insert(123, "abc"); line();
    map.insert(456, "def"); line();
    map.insert(456, "ghi"); line();
    map.insert(223, "xyz"); line();
}

int main(int argc, char **argv) {

```

```
test_string();  
test_int();  
return 0;  
}
```


Algorithms Library

[Algorithms library](#)

[Other Content](#)

[Streams](#)

Object Oriented Advanced

Basics

Syntax of Inheritance

```
#include <iostream>
using namespace std;
// Base class (Parent class)
class Base {
public:
    void parent_class_function() {
        cout << "This is the base class method." << endl;
    }
};
// Derived class (Child class) that inherits from the Base class publically
class Derived : public Base {
public:
    void child_class_function() {
        cout << "This is the derived class method." << endl;
    }
};
// Derived -> Base
int main() {
    Derived derived_obj;
    // Calling method from the base class
    derived_obj.parent_class_function();
    // Calling method from the derived class
    derived_obj.child_class_function();
    return 0;}
```

Advanced: Adapter pattern

We can get a little creative with the inheritance syntax and deploy an "Adapter pattern":

```
#include <iostream>
#include <string>
using namespace std;
// Existing class (Adaptee)
class LegacyMediaPlayer {
public:
    void play_mp3(const string& filename) {
        cout << "Playing MP3 file: " << filename << endl;
    }
};
// Target interface
class AdvancedMediaPlayer {
public:
    void play_mp4(const string& filename) {
        cout << "This player does not support MP4 files." << endl;
    }
    void play_vlc(const string& filename) {
        cout << "This player does not support VLC files." << endl;
    }
};
// Adapter class using private inheritance
class MediaPlayerAdapter : private LegacyMediaPlayer, public AdvancedMediaPlayer {
public:
    void play_mp4(const string& filename) {
        // For simplicity, convert MP4 to MP3 (dummy implementation)
        cout << "Converting MP4 to MP3..." << endl;
        LegacyMediaPlayer::play_mp3(filename + ".converted.mp3");
    }
    void play_vlc(const string& filename) {
        // For simplicity, convert VLC to MP3 (dummy implementation)
        cout << "Converting VLC to MP3..." << endl;
    }
};
```

```

        LegacyMediaPlayer::play_mp3(filename + ".converted.mp3");
    }
};

// Client code
int main() {
    MediaPlayerAdapter player;
    player.play_mp4("example.mp4");
    player.play_vlc("example.vlc");
    return 0;
}

```

Basic Inheritance pattern

Let us learn by studying an example.

```

#include <iostream>
#include <string>
using namespace std;
class Vehicle {
    // protected says that inheriting class will have access to these members
    // but users of the class will not
protected:
    string brand;
    int year;
    void protected_method() { std::cout << "protected method" << std::endl; }
public:
    Vehicle(string b, int y) : brand(b), year(y) {}
    virtual void display_info() {
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};

void example() {
    Vehicle v("name", 2020);
    //v.brand = "123"; // Can't do because brand is private, only inheriting child class can access it
    //v.protected_method(); // Can't do this because this method is not public
}

```

```

}

class Car : public Vehicle {
private:
    int doors;
public:
    Car(string b, int y, int d) : Vehicle(b, y), doors(d) {}
    void display_info() override {
        cout << "Car - Brand: " << Vehicle::brand <<
            ", Year: " << Vehicle::year <<
            ", Doors: " << doors << endl;
    }
};

class Motorcycle : public Vehicle {
private:
    string type;
public:
    Motorcycle(string b, int y, string t) : Vehicle(b, y), type(t) {}
    void display_info() override {
        cout << "Motorcycle - Brand: " << brand << ", Year: " << year << ", Type: " << type << endl;
    }
};

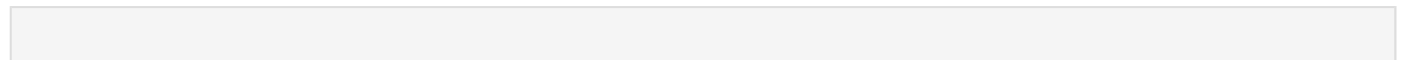
int main() {
    Car car("Toyota", 2020, 4);
    Motorcycle motorcycle("Yamaha", 2019, "Sport");
    car.display_info();
    motorcycle.display_info();
    return 0;
}

```

Abstract classes

Override

A base class can define method(s) that could have override behavior:



```

class Animal {
public:
    virtual void speak() {
        std::cout << "Some generic animal sound" << endl;
    }
};

class another_animal : public Animal {
public:
    void speak() { /* Intent is to override the base, parent class */
        std::cout << "I speak differently" << std::endl;
    }
};

```

One question to ask in the code snippet above is that what happens if the base class changes its method name? In this case, the intent was for `another_animal` to override, but it no longer overrides anything. This is the reason the `override` keyword was invented.

```

class Animal {
public:
    //virtual void speak() { /* Case changes: */
    virtual void Speak() {
        std::cout << "Some generic animal sound" << endl;
    }
};

class another_animal : public Animal {
public:
    void speak() { /* This overrides the base class function */
        std::cout << "I speak differently" << std::endl;
    }
};

```

Let us use C++11 syntax to use the `override` keyword and see what happens when a method is intending to override a function but it doesn't do so:

```

class Animal {
public:
    //virtual void speak() { /* Case changes: */
    virtual void Speak() {

```

```

        std::cout << "Some generic animal sound" << endl;
    }
};
class another_animal : public Animal {
public:
    void speak() override { /* This overrides the base class function */
        std::cout << "I speak differently" << std::endl;
    }
};

```

Pure Virtual

A base class can provide default functionality or mandate that the child class inheriting the base (parent) class has to provide a certain functionality. It would look like this:

```

class Animal {
public:
    virtual void speak() = 0; /* I do not know how to speak, someone else tell me how */
};
class dolphin : public Animal {
public:
    void speak() override {
        std::cout << "I am a dolphin and I can whistle" << std::endl;
    }
};

```

Example 1

```

#include <iostream>
using namespace std;
class Animal {
public:
    virtual void speak() = 0;
};
class Dog : public Animal {
public:

```

```

        void speak() override { // Overriding the base class function
            cout << "Woof!" << endl;
        }
};
class Cat : public Animal {
public:
    void speak() override { // Overriding the base class function
        cout << "Meow!" << endl;
    }
};
void speak(Animal& animal) {
    animal.speak();
}

```

Example 2

```

#include <iostream>
#include <vector>
class Shape {
public:
    // Virtual function to be overridden by derived classes
    virtual void draw() const {
        std::cout << "Drawing a shape\n";
    }
    // Virtual destructor to ensure proper cleanup of derived objects
    virtual ~Shape() {}
};
class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle\n";
    }
};
class Rectangle : public Shape {
public:

```



```

    void draw() const override {
        std::cout << "Drawing a rectangle\n";
    }
};

int main() {
    // Create a vector of Shape pointers
    std::vector<Shape*> shapes;
    // Add different shapes to the vector
    shapes.push_back(new Circle());
    shapes.push_back(new Rectangle());
    shapes.push_back(new Circle());
    // Iterate through the vector and call the draw method
    for (const auto& shape : shapes) {
        shape->draw(); // Polymorphic call
    }
    // Clean up the allocated memory
    for (auto& shape : shapes) {
        delete shape;
    }
    return 0;}

```

Video Game Example

```

#include <iostream>
#include <vector>
#include <memory>
// Base class
class Character {
public:
    virtual void attack() const {
        std::cout << "Character attacks in a generic way.\n";
    }
    virtual ~Character() {}
};
// Derived class: Warrior

```

```

class Warrior : public Character {
public:
    void attack() const override {
        std::cout << "Warrior swings a sword!\n";
    }
};

// Derived class: Mage
class Mage : public Character {
public:
    void attack() const override {
        std::cout << "Mage casts a fireball!\n";
    }
};

// Derived class: Archer
class Archer : public Character {
public:
    void attack() const override {
        std::cout << "Archer shoots an arrow!\n";
    }
};

int main() {
    // Create a vector of unique_ptr to Character
    std::vector<std::unique_ptr<Character>> characters;
    // Add different types of characters to the vector
    characters.push_back(std::make_unique<Warrior>());
    characters.push_back(std::make_unique<Mage>());
    characters.push_back(std::make_unique<Archer>());
    // Iterate through the vector and call the attack method
    for (const auto& character : characters) {
        character->attack(); // Polymorphic call
    }
    // No need to delete characters; unique_ptr handles it automatically
    return 0;
}

```

Interfaces

An interface abstracts the implementation and enables unit-testing on the code. In other words, an interface is an abstract class that decouples the implementation from the interface, which allows changing or adding new implementations without affecting the rest of the program.

Polymorphism: The code below demonstrates polymorphism through the use of the interface. The `test_communication` function can take any object that implements the Communicable interface, showcasing how polymorphism enables the writing of more flexible and reusable code.

Example 1

In the example below, the interface ensures that any class that implements it will have a `send` method, but does not commit to how the sending is done, which is polymorphic.

```
#include <iostream>
// Interface class with a single responsibility
class Communicable {
public:
    virtual void send(const std::string& message) = 0; // Only one pure virtual function
    virtual ~Communicable() {} // Virtual destructor for proper cleanup
};
class WiFiDevice : public Communicable {
public:
    void send(const std::string& message) override {
        std::cout << "Sending over WiFi: " << message << std::endl;
    }
};
class BluetoothDevice : public Communicable {
public:
    void send(const std::string& message) override {
        std::cout << "Sending over Bluetooth: " << message << std::endl;
    }
};
void test_communication(Communicable& device, const std::string& message) {
```

```

        device.send(message); // Use the send method to demonstrate communication
    }
    int main() {
        WiFiDevice wifi;
        BluetoothDevice bluetooth;
        testCommunication(wifi, "Hello WiFi World");
        testCommunication(bluetooth, "Hello Bluetooth World");
        return 0;
    }

```

Example 2

```

#include <iostream>
#include <vector>
#include <memory>
// Interface class
class GameCharacter {
public:
    virtual void attack() = 0;           // Pure virtual function
    virtual void defend() = 0;           // Pure virtual function
    virtual void heal() = 0;             // Pure virtual function
    virtual int get_health() const = 0; // Pure virtual function to get health
    virtual ~GameCharacter() {}
};

void perform_action(GameCharacter& character) {
    if (character.get_health() < 30) {
        character.heal();
    } else if (rand() % 2 == 0) { // 50% chance to attack or defend
        character.attack();
    } else {
        character.defend();
    }
}

```

Exercise

Let us continue working on the examples below and practice more uses of interfaces.

1. Create an interface that returns low threshold level for health
2. Create an interface that returns when a character should attack
3. Write two classes that inherit and implement the GameCharacter class
4. Write a main function to invoke `perform_action()` on the two classes

Smart Pointers

C++ 11 standard solved a major safety problem with the language: Managing memory

- Creating memory
- Deleting memory
- Managing who owns memory and how many owners are there?

Example

Problematic code

Memory becomes difficult to manage when you give the pointers or references away. Here is an example.

```
// Interface for an internet socket API
class socket_i {
public:
    virtual void send() = 0;
    virtual void recv() = 0;
    virtual ~socket_i() {}
};

// Child class that implements an interface
class socket : public socket_i {
public:
    void send() override {
    }

    void recv() override {
    }
};

// An HTTP library that uses a socket interface to communicate
```

```

class http {
public:
    http(socket_i *sock) : m_socket(sock) {
    }
private:
    socket_i *m_socket;
};
// Multiple problems with this code
http* create_http() {
    socket s;
    http http_obj(&s);
    return http_obj;
}
int main(void) {
    http* http_obj = create_http();}

```

One of the problems can be simplified to this:

```

int* get() {
    int x;
    return &x;
}
void problem() {
    int* pointer = get();
    *pointer = 123;}

```

Fixed Version 1

The partially corrected version creates a new memory reference, and it leaves it up to the consumer of the code to actually delete the object. This is just one of the issues: memory management problem. Because you are relying on the user of the code to delete your memory, it creates traps in your code.

```

// Let us partially fix our code
http* create_http() {
    socket s;

```

```

// Let us create new memory which will not go out of scope
// even when this function exits
// BUT:
// We have now created memory management problem:
// ie:
// - Who deletes this memory?
// - When do they do it?
// - Can you ensure that they delete it?
http * http_obj = new http(&s);
return http_obj;
}

int main(void) {
    http* http_obj = create_http();

    // We better delete our object or else it is memory leak
    delete http_obj;}

```

std::unique_ptr

Let us introduce the concept of using a unique pointer such that we do not have to worry about deleting it.

```

// Let us partially fix our code
std::unique_ptr<http> create_http() {
    socket s;
    std::unique_ptr<http> http_obj = std::make_unique<http>(&s);
    return http_obj;
}

int main(void) {
    std::unique_ptr<http> http_obj = create_http();
    // No need to worry about deleting our allocated resource
    //delete http_obj}

```

The unique_ptr is also special because it will force you to maintain "one owner".

std::shared_ptr

If we continue in our example, we will notice that there is still a fundamental flaw which is that the socket that was created would go out of scope, and the http class would then end up using a reference to memory that no longer is alive (no longer in scope).

```
// Interface for an internet socket API
class socket_i {
public:
    virtual void send() = 0;
    virtual void recv() = 0;
    virtual ~socket_i() {}
};

// Child class that implements an interface
class socket : public socket_i {
public:
    void send() override {
    }

    void recv() override {
    }
};

// An HTTP library that uses a socket interface to communicate
class http {
public:
    http(std::shared_ptr<socket_i> sock) : m_socket(sock) {
    }
private:
    std::shared_ptr<socket_i> m_socket;
};
```

Cleaned up code

```
#include <iostream>
```

```

#include <memory>
// Interface for an internet socket API
class socket_i {
public:
    virtual bool open(const std::string& hostname) = 0;
    virtual void send(const std::string& data_to_transit) = 0;
    virtual std::string recv() = 0;
    virtual ~socket_i() {}
};

// An HTTP library that uses a socket interface to communicate
class http {
public:
    http(std::shared_ptr<socket_i> sock) : m_socket(sock) {
    }
    ~http() {
        std::cout << "Destructor of http class has been called" << std::endl;
    }
    void send_request(const std::string& host, const std::string& resource) {
        std::string request = "GET " + resource + " HTTP/1.1\r\n";
        request += "Host: " + host + "\r\n";
        request += "Connection: close\r\n\r\n";
        m_socket->send(request);
        std::string response = m_socket->recv();
        std::cout << "Response:\n" << response << std::endl;
    }
private:
    std::shared_ptr<socket_i> m_socket;
};

class linux_socket : public socket_i {
public:
    bool open(const std::string& hostname) override {
        return false;
    }
    void send(const std::string& data_to_transit) override {
    }

    std::string recv() override {

```

```

        return std::string{};
    }
};

// Let us partially fix our code
std::unique_ptr<http> create_http() {
    auto socket = std::make_shared<linux_socket>();
    // References to the shared pointer:
    std::cout << "reference count before: " << socket.use_count() << std::endl;
    // Let us create new memory which will not go out of scope
    // even when this function exits
    // BUT:
    // We have now created memory management problem:
    // ie:
    // - Who deletes this memory?
    // - When do they do it?
    // - Can you ensure that they delete it?
    std::unique_ptr<http> http_obj = std::make_unique<http>(socket);
    std::cout << "reference count after: " << socket.use_count() << std::endl;
    return http_obj;
}

int main(void) {
    auto http_obj = create_http();
    // This uses "socket" which was created at the first line inside create_http()
    http_obj->send_request("google.com", "index.html");
    // We better delete our object or else it is memory leak
    //delete http_obj;
    std::cout << "End of main()" << std::endl;}

```

Function Pointers & Lambdas

Function pointers are essential in C and C++ programming.

Function Pointers in C

```
#include <stdio.h>
// Function prototypes for the operations
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
int main() {
    // Define function pointers for the operations
    int (*operation)(int, int);
    int x = 10;
    int y = 5;
    int result;
    // Use the function pointer to point to the add function
    operation = add;
    result = operation(x, y);
    printf("Adding: %d + %d = %d\n", x, y, result);
    // Change the function pointer to point to the subtract function
    operation = subtract;
    result = operation(x, y);
    printf("Subtracting: %d - %d = %d\n", x, y, result);
    return 0;}
```

We can make the code more intuitive by using a "typedef".

```

#include <stdio.h>

// Function prototypes for the operations
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

// Using typedef to simplify function pointer declarations
typedef int (*function_pointer_for_operation)(int, int);

int main() {
    // Declare a function pointer using the typedef
    function_pointer_for_operation operation;

    int x = 10;
    int y = 5;
    int result;

    // Use the function pointer to point to the add function
    operation = add;
    result = operation(x, y);
    printf("Adding: %d + %d = %d\n", x, y, result);

    // Change the function pointer to point to the subtract function
    operation = subtract;
    result = operation(x, y);
    printf("Subtracting: %d - %d = %d\n", x, y, result);

    return 0;
}

```

Function Pointers in C++

```

#include <iostream>
#include <functional>

// Function prototypes for the operations
int add(int a, int b) {
    return a + b;
}

```

```

int subtract(int a, int b) {
    return a - b;
}

int main() {
    // Using std::function to declare function pointers
    std::function<int(int, int)> operation;

    int x = 10;
    int y = 5;
    int result;

    // Assign the add function to the operation std::function
    operation = add;
    result = operation(x, y);
    std::cout << "Adding: " << x << " + " << y << " = " << result << std::endl;
    // Change the operation to subtract
    operation = subtract;
    result = operation(x, y);
    std::cout << "Subtracting: " << x << " - " << y << " = " << result << std::endl;
    return 0;
}

```

Similar to C, we can improve our code by using a typedef:

```

#include <iostream>
#include <functional>

// Define the function type using typedef for clarity and reusability
typedef std::function<int(int, int)> MathOperation;

// Function prototypes for the operations
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int main() {
    // Declare a variable of type MathOperation
    MathOperation operation;
}

```

```

int x = 10;
int y = 5;
int result;
// Assign the add function to the operation
operation = add;
result = operation(x, y);
std::cout << "Adding: " << x << " + " << y << " = " << result << std::endl;
// Change the operation to subtract
operation = subtract;
result = operation(x, y);
std::cout << "Subtracting: " << x << " - " << y << " = " << result << std::endl;
return 0;
}

```

Example 1

```

#include <iostream>
#include <vector>
#include <algorithm>
// Comparison functions:
bool ascending_compare(int a, int b) {
    return a < b; // Change this to a > b for descending order
}
bool descending_compare(int a, int b) {
    return a > b; // Change this to a > b for descending order
}
int main() {
    // Create a vector of integers
    std::vector<int> vec = {-8, 5, 2, 9, 1, 5, 6, 3};
    // Sort the vector using the comparison function
    std::sort(vec.begin(), vec.end(), descending_compare);
    // Print the sorted vector
    std::cout << "Sorted vector: ";
    for (int value : vec) {
        std::cout << value << " ";
    }
}

```

```
}  
std::cout << std::endl;  
return 0;}
```

Example 2

```
#include <iostream>  
#include <chrono>  
#include <functional>  
#include <thread>  
class Timer {  
public:  
    void callback_after_delay(std::function<void()> callback, int delay) {  
        std::this_thread::sleep_for(std::chrono::milliseconds(delay));  
        callback();  
    }  
};  
void function_to_callback() {  
    std::cout << "Callback invoked!" << std::endl;  
}  
int main() {  
    Timer t;  
    t.callback_after_delay(function_to_callback, 3000);  
    return 0;}
```

Lambda

Before we learn what is a lambda, let us consider naive code that achieves similar functionality of a lambda but it uses function pointer:

```
#include <algorithm>  
#include <iostream>  
#include <vector>  
// A simple comparison function
```



```

bool compare(int a, int b) {
    return a < b;
}

int main() {
    std::vector<int> data = {5, 3, 9, 1, 6};
    // Sort using a function pointer
    std::sort(data.begin(), data.end(), compare);
    for (int n : data) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
    return 0;
}

```

In super naive terms, a lambda is a function pointer. Just remember this syntax: `[] () {}`

```

#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    std::vector<int> data = {5, 3, 9, 1, 6};
    // Sort using a lambda expression
    auto compare_function_lambda = [](int a, int b) {return a < b; };
    std::sort(data.begin(), data.end(), compare_function_lambda);
    for (int n : data) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
    return 0;
}

```

Threading Library

Basics

- What is multithreading?
- Why multithreading?

Overview of C+ thread library

Let us start with an easy example:

```
#include <iostream>
#include <thread>
void helloFunction() {
    std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 1000));
    std::cout << "Hello from a thread!\n";
}
int main() {
    std::thread t1(helloFunction);
    std::thread t2(helloFunction);

    std::cout << "We have launched two threads to operate in parallel!" << std::endl;
    t1.join(); // Wait for the thread to finish
    t2.join();

    return 0;
}
```