object relationships

Life is full of recurring patterns, relationships, and hierarchies between objects. By exploring and understanding these, we can gain insight into how real-life objects behave, enhancing our understanding of those objects. Similarly, programming is also full of recurring patterns, relationships and hierarchies. Particularly when it comes to programming objects, the same patterns that govern real-life objects are applicable to the programming objects we create ourselves. There are many different kinds of relationships two objects may have in real-life, and we use specific "relation type" words to describe these relationships. For example: a square "is-a" shape. A car "has-a" steering wheel. A computer programmer "uses-a" keyboard. A flower "depends-on" a bee for pollination. A student is a "member-of" a class. And your brain exists as "part-of" you (at least, we can reasonably assume so if you've gotten this far). All of these relation types have useful analogies in C++.

object composition

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc…object composition models a "has-a" relationship between two objects. A car "has-a" transmission. Your computer "has-a" CPU. You "have-a" heart. **It has two subtypes; composition and aggregation.**

1) composition

To qualify as a **composition**, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

For example, a heart is a part of a person's body. The part in a composition can only be part of one object at a time. A heart that is part of one person's body can not be part of someone else's body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed. But more broadly, it means the object manages the part's lifetime in such a way that the user of the object does not need to get involved. the part doesn't know about the existence of the whole. Your heart operates blissfully unaware that it is part of a larger structure. We call this a **unidirectional** relationship, because the body knows about the heart, but not the other way around.

```
class Fraction
{
private:
    int m_numerator;
    int m_denominator;
```

While object composition models "**has-a"** type relationships (a body has-a heart, a fraction has-a denominator), we can be more precise and say that composition models "part-of" relationships (a heart is part-of a body, a numerator is part of a fraction).

Many games and simulations have creatures or objects that move around a board, map, or screen. One thing that all of these creatures/objects have in common is that they all have a location.

```
#ifndef POINT2D_H
#define POINT2D_H

#include <iostream>

class Point2D
{
private:
    int m_x;
    int m_y;

public:
    // A default constructor
    Point2D()
        : m_x{ 0 }, m_y{ 0 }
    {
    }

    // A specific constructor
    Point2D(int x, int y)
        : m_x{ x }, m_y{ y }
    {
    }

    // An overloaded output operator
    friend std::ostream& operator<<(std::ostream& out, const Point2D& point)
    {
        out << '(' << point.m_x << ", " << point.m_y << ')';
        return out;
    }

    // Access functions
    void setPoint(int x, int y)
    {
        m_x = x;
        m_y = y;
    }

};

#endif
```

The creature is going to have a few properties: a name string and a location, which will be our Point2D class.

```
#ifndef CREATURE_H
#define CREATURE_H

#include <iostream>
#include <string>
#include <string_view>
#include "Point2D.h"

class Creature
{
private:
    std::string m_name;
    Point2D m_location;

public:
    Creature(std::string_view name, const Point2D& location)
        : m_name{ name }, m_location{ location }
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Creature& creature)
    {
        out << creature.m_name << " is at " << creature.m_location;
        return out;
    }

    void moveTo(int x, int y)
    {
        m_location.setPoint(x, y);
    }
};
#endif
```

The creature's name and location have one parent, and their lifetime is tied to the Creature they are part of.

One question that new programmers often ask when it comes to object composition is, "When should I use a class member instead of direct implementation of a feature?". For example, instead of using the Point2D class to implement the Creature's location, we could have instead just added 2 integers to the Creature class and written code in the Creature class to handle the positioning. However, making Point2D its own class (and a member of Creature) has a number of benefits:

1. Each individual class can be kept relatively simple and straightforward, focused on performing one task well. This makes those classes easier to write and much easier to understand, as they are more focused. For example, Point2D only worries about point-related stuff, which helps keep it simple.
2. Each class can be self-contained, which makes them reusable. For example, we could reuse our Point2D class in a completely different application. Or if our creature ever needed another point (for example, a destination it was trying to get to), we can simply add another Point2D member variable.
3. The outer class can have the class members do most of the hard work, and instead focus on coordinating the data flow between the members . This helps lower the overall complexity of the outer class, because it can delegate tasks to its members, who already know how to do those tasks. For example, when we move our Creature, it delegates that task to the Point class, which already understands how to set a point. Thus, the Creature class does not have to worry about how such things would be implemented.

2) aggregation

To qualify as an **aggregation**, a whole object and its parts must have the following relationship:

- The part (member) is part of the object (class)

- The part (member) can (if desired) belong to more than one object (class) at a time
- The part (member) does *not* have its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

Like a composition, an aggregation is still a part-whole **"has-a"** type relationship, where the parts are contained within the whole, and it is a unidirectional relationship. However, unlike a composition, parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts. When an aggregation is created, the aggregation is not responsible for creating the parts. When an aggregation is destroyed, the aggregation is not responsible for destroying the parts.

For example, consider the relationship between a person and their home address. In this example, for simplicity, we'll say every person has an address. However, that address can belong to more than one person at a time: for example, to both you and your roommate or significant other. However, that address isn't managed by the person -- the address probably existed before the person got there, and will exist after the person is gone. Additionally, a person knows what address they live at, but the addresses don't know what people live there. Therefore, this is an aggregate relationship.

In aggregation as well, we also add parts as member variables however, these member variables are typically either references or pointers that are used to point at objects that have been created outside the scope of the class. Consequently, an aggregation usually either takes the objects it is going to point to as constructor parameters, or it begins empty and the subobjects are added later via access functions or operators.

Because these parts exist outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed (but not deleted). Consequently, the parts themselves will still exist.

```
#include <iostream>
#include <string>
#include <string_view>

class Teacher
{
private:
  std::string m_name{};

public:
  Teacher(std::string_view name)
      : m_name{ name }
  {
  }

  const std::string& getName() const { return m_name; }
};

class Department
{
private:
  const Teacher& m_teacher; // This dept holds only one teacher for simplicity, but it could hold many teachers

public:
  Department(const Teacher& teacher)
      : m_teacher{ teacher }
  {
  }
};

```

```cpp
int main()
{
  // Create a teacher outside the scope of the Department
  Teacher bob{ "Bob" }; // create a teacher

  {
    // Create a department and use the constructor parameter to pass
    // the teacher to it.
    Department department{ bob };

  } // department goes out of scope here and is destroyed

  // bob still exists here, but the department doesn't

  std::cout << bob.getName() << " still exists!\n";

  return 0;
}
```

also if there were more than one teachers, we would need to store the values in a std::vector but you cannot store references in a std::vector. so if you had two variables a & b and u tried to store them in an vector; you store only copies of them so if you changed a & b but printed the vector, you would only get the old values. you can use std::reference_wrapper as the elements will act like references but you will be able to assign and copy from it as well.

```
int a = 10;
int b = 20;

std::vector<int> vec = {a, b};                                  //ONLY STORES COPIES
std::vector<std::reference_wrapper<int>> vec = {a, b};          //STORES REFERENCE ADRESS
```

2) association

To qualify as an **association**, an object and another object must have the following relationship:

- The associated object (member) is otherwise unrelated to the object (class)
- The associated object (member) can belong to more than one object (class) at a time
- The associated object (member) does *not* have its existence managed by the object (class)
- The associated object (member) may or may not know about the existence of the object (class)

The association models has a **“uses-a”** relationship. The doctor “uses” the patient (to earn income). The patient uses the doctor (for whatever health purposes they need).

```cpp
#include <functional> // reference_wrapper
#include <iostream>
#include <string>
#include <string_view>
#include <vector>

// Since Doctor and Patient have a circular dependency, we're going to forward declare Patient
class Patient;

class Doctor
{
private:
    std::string m_name{};
    std::vector<std::reference_wrapper<const Patient>> m_patient{};

public:
    Doctor(std::string_view name) :
        m_name{ name }
    {
    }

    void addPatient(Patient& patient);

    // We'll implement this function below Patient since we need Patient to be defined at that point
    friend std::ostream& operator<<(std::ostream& out, const Doctor& doctor);

    const std::string& getName() const { return m_name; }
};

class Patient
{
private:
    std::string m_name{};
    std::vector<std::reference_wrapper<const Doctor>> m_doctor{}; // so that we can use it here

    // We're going to make addDoctor private because we don't want the public to use it.
    // They should use Doctor::addPatient() instead, which is publicly exposed
    void addDoctor(const Doctor& doctor)
    {
        m_doctor.push_back(doctor);
    }

public:
    Patient(std::string_view name)
        : m_name{ name }
    {
    }

    // We'll implement this function below to parallel operator<<(std::ostream&, const Doctor&)
    friend std::ostream& operator<<(std::ostream& out, const Patient& patient);

    const std::string& getName() const { return m_name; }

    // We'll friend Doctor::addPatient() so it can access the private function Patient::addDoctor()
    friend void Doctor::addPatient(Patient& patient);
};

void Doctor::addPatient(Patient& patient)
{
    // Our doctor will add this patient
    m_patient.push_back(patient);

    // and the patient will also add this doctor
    patient.addDoctor(*this);
}
```

```
std::ostream& operator<<(std::ostream& out, const Doctor& doctor)
{
    if (doctor.m_patient.empty())
    {
        out << doctor.m_name << " has no patients right now";
        return out;
    }

    out << doctor.m_name << " is seeing patients: ";
    for (const auto& patient : doctor.m_patient)
        out << patient.get().getName() << ' ';

    return out;
}

std::ostream& operator<<(std::ostream& out, const Patient& patient)
{
    if (patient.m_doctor.empty())
    {
        out << patient.getName() << " has no doctors right now";
        return out;
    }

    out << patient.m_name << " is seeing doctors: ";
    for (const auto& doctor : patient.m_doctor)
        out << doctor.get().getName() << ' ';

    return out;
}
```

```
int main()
{
    // Create a Patient outside the scope of the Doctor
    Patient dave{ "Dave" };
    Patient frank{ "Frank" };
    Patient betsy{ "Betsy" };

    Doctor james{ "James" };
    Doctor scott{ "Scott" };

    james.addPatient(dave);

    scott.addPatient(dave);
    scott.addPatient(betsy);

    std::cout << james << '\n';
    std::cout << scott << '\n';
    std::cout << dave << '\n';
    std::cout << frank << '\n';
    std::cout << betsy << '\n';

    return 0;
}
```

In general, you should avoid bidirectional associations if a unidirectional one will do, as they add complexity and tend to be harder to write without making errors.

Sometimes objects may have a relationship with other objects of the same type. This is called a **reflexive association**. A good example of a reflexive association is the relationship between a university course and its prerequisites (which are also university courses).

```
class Course
{
private:
    std::string m_name{};
    const Course* m_prerequisite{};

public:
    Course(std::string_view name, const Course* prerequisite = nullptr):
        m_name{ name }, m_prerequisite{ prerequisite }
    {
    }

};
```

This can lead to a chain of associations (a course has a prerequisite, which has a prerequisite, etc...)

we don’t have to use only pointers or references to link objects together

```
#include <iostream>
#include <string>
#include <string_view>

class Car
{
private:
    std::string m_name{};
    int m_id{};

public:
    Car(std::string_view name, int id)
        : m_name{ name }, m_id{ id }
    {
    }

    const std::string& getName() const { return m_name; }
    int getId() const { return m_id; }
};

// Our CarLot is essentially just a static array of Cars and a lookup function to retrieve them.
// Because it's static, we don't need to allocate an object of type CarLot to use it
namespace CarLot
{
    Car carLot[4] { { "Prius", 4 }, { "Corolla", 17 }, { "Accord", 84 }, { "Matrix", 62 } };

    Car* getCar(int id)
    {
        for (auto& car : carLot)
        {
            if (car.getId() == id)
            {
                return &car;
            }
        }

        return nullptr;
    }
};

class Driver
{
private:
    std::string m_name{};
    int m_carId{}; // we're associated with the Car by ID rather than pointer

public:
    Driver(std::string_view name, int carId)
        : m_name{ name }, m_carId{ carId }
    {
    }

    const std::string& getName() const { return m_name; }
    int getCarId() const { return m_carId; }
};

int main()
{
    Driver d{ "Franz", 17 }; // Franz is driving the car with ID 17

    Car* car{ CarLot::getCar(d.getCarId()) }; // Get that car from the car lot

    if (car)
        std::cout << d.getName() << " is driving a " << car->getName() << '\n';
    else
        std::cout << d.getName() << " couldn't find his car\n";

    return 0;
}
```

the CarLot holds our cars. The Driver, who needs a car, doesn’t have a pointer to his Car -- instead, he has the ID of the car, which we can use to get the Car from the CarLot when we need it. In this particular example, doing things this way is kind of silly, since getting the Car out of the CarLot requires an inefficient lookup (a pointer connecting the two is much faster). However, there are advantages to referencing things by a unique ID instead of a pointer. For example, you can reference things that are not currently in memory (maybe they’re in a file, or in a database, and can be loaded on demand). Also, pointers can take 4 or 8 bytes -- if space is at a premium and the number of unique objects is fairly low, referencing them by an 8-bit or 16-bit integer can save lots of memory.

Property	Composition	Aggregation	Association
Relationship type	Whole/part	Whole/part	Otherwise unrelated
Members can belong to multiple classes	No	Yes	Yes
Members’ existence managed by class	Yes	No	No
Directionality	Unidirectional	Unidirectional	Unidirectional or bidirectional
Relationship verb	Part-of	Has-a	Uses-a

3) dependency

A **dependency** occurs when one object invokes another object's functionality in order to accomplish some specific task. This is a weaker relationship than an association, but still, any change to object being depended upon may break functionality in the (dependent) caller. A dependency is always a unidirectional relationship.

A good example of a dependency that you've already seen many times is std::ostream where the classes that use std::ostream, use it in order to accomplish the task of printing something to the console, but not otherwise.

```
#include <iostream>

class Point
{
private:
    double m_x{};
    double m_y{};
    double m_z{};

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x{x}, m_y{y}, m_z{z}
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Point& point); // Point has a dependency on
std::ostream here
};

std::ostream& operator<< (std::ostream& out, const Point& point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly.
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')';

    return out;
}

int main()
{
    Point point1 { 2.0, 3.0, 4.0 };

    std::cout << point1; // the program has a dependency on std::cout here

    return 0;
}
```

Regarding the confusions of Dependencies vs Associations in C++, associations are a relationship where one class keeps a direct or indirect "link" to the associated class as a member. For example, a Doctor class has an array of pointers to its Patients as a member. You can always ask the Doctor who its patients are. The Driver class holds the id of the Car the driver object owns as an integer member. The Driver always knows what Car is associated with it.

Dependencies typically are not members. Rather, the object being depended on is typically instantiated as needed (like opening a file to write data to), or passed into a function as a parameter (like std::ostream in the overloaded operator<< above).

4) container class

a **container class** is a class designed to hold and organize multiple instances of another type (either another class, or a fundamental type) & they implement a **“member-of”** relationship. By far the most commonly used container in programming is the array.

Container classes generally come in two different varieties.

value containers are compositions that store copies of the objects that they are holding (and thus are responsible for creating and destroying those copies).

reference containers are aggregations that store pointers or references to other objects (and thus are not responsible for creation or destruction of those objects).

an array container class written by user would look like.

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <algorithm> // for std::copy_n
#include <cassert> // for assert()
#include <cstddef> // for std::size_t

class IntArray
{
private:
    int m_length{};
    int* m_data{};

public:
    IntArray() = default;

    IntArray(int length):
        m_length{ length }
    {
        assert(length >= 0);

        if (length > 0)
            m_data = new int[static_cast<std::size_t>(length)]{};
    }

    ~IntArray()
    {
        delete[] m_data;
        // we don't need to set m_data to null or m_length to 0 here, since the object will be destroyed immediately
after this function anyway
    }

    IntArray(const IntArray& a): IntArray(a.getLength()) // use normal constructor to set size of array appropriately
    {
        std::copy_n(a.m_data, m_length, m_data); // copy the elements
    }

    IntArray& operator=(const IntArray& a)
    {
        // Self-assignment check
        if (&a == this)
            return *this;

        // Set the size of the new array appropriately
        reallocate(a.getLength());
        std::copy_n(a.m_data, m_length, m_data); // copy the elements

        return *this;
    }

    void erase()
    {
        delete[] m_data;
        // We need to make sure we set m_data to nullptr here, otherwise it will
        // be left pointing at deallocated memory!
        m_data = nullptr;
        m_length = 0;
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }
```

```
    // reallocate resizes the array.  Any existing elements will be destroyed.  This function operates quickly.
    void reallocate(int newLength)
    {
        // First we delete any existing elements
        erase();

        // If our array is going to be empty now, return here
        if (newLength <= 0)
            return;

        // Then we have to allocate new elements
        m_data = new int[static_cast<std::size_t>(newLength)];
        m_length = newLength;
    }

    // resize resizes the array.  Any existing elements will be kept.  This function operates slowly.
    void resize(int newLength)
    {
        // if the array is already the right length, we're done
        if (newLength == m_length)
            return;

        // If we are resizing to an empty array, do that and return
        if (newLength <= 0)
        {
            erase();
            return;
        }

        // Now we can assume newLength is at least 1 element.  This algorithm
        // works as follows: First we are going to allocate a new array.  Then we
        // are going to copy elements from the existing array to the new array.
        // Once that is done, we can destroy the old array, and make m_data
        // point to the new array.


        // First we have to allocate a new array
        int* data{ new int[static_cast<std::size_t>(newLength)] };

        // Then we have to figure out how many elements to copy from the existing
        // array to the new array.  We want to copy as many elements as there are
        // in the smaller of the two arrays.
        if (m_length > 0)
        {
            int elementsToCopy{ (newLength > m_length) ? m_length : newLength };
            std::copy_n(m_data, elementsToCopy, data); // copy the elements
        }

        // Now we can delete the old array because we don't need it any more
        delete[] m_data;

        // And use the new array instead!  Note that this simply makes m_data point
        // to the same address as the new array we dynamically allocated.  Because
        // data was dynamically allocated, it won't be destroyed when it goes out of scope.
        m_data = data;
        m_length = newLength;
    }


    void insertBefore(int value, int index)
    {
        // Sanity check our index value
        assert(index >= 0 && index <= m_length);

        // First create a new array one element larger than the old array
        int* data{ new int[static_cast<std::size_t>(m_length+1)] };

        // Copy all of the elements up to the index
        std::copy_n(m_data, index, data);

        // Insert our new element into the new array
        data[index] = value;

        // Copy all of the values after the inserted element
        std::copy_n(m_data + index, m_length - index, data + index + 1);

        // Finally, delete the old array, and use the new array instead
        delete[] m_data;
        m_data = data;
        ++m_length;
    }
```

```
    void remove(int index)
    {
        // Sanity check our index value
        assert(index >= 0 && index < m_length);

        // If this is the last remaining element in the array, set the array to empty and bail out
        if (m_length == 1)
        {
            erase();
            return;
        }

        // First create a new array one element smaller than the old array
        int* data{ new int[static_cast<std::size_t>(m_length-1)] };

        // Copy all of the elements up to the index
        std::copy_n(m_data, index, data);

        // Copy all of the values after the removed element
        std::copy_n(m_data + index + 1, m_length - index - 1, data + index);

        // Finally, delete the old array, and use the new array instead
        delete[] m_data;
        m_data = data;
        --m_length;
    }

    // A couple of additional functions just for convenience
    void insertAtBeginning(int value) { insertBefore(value, 0); }
    void insertAtEnd(int value) { insertBefore(value, m_length); }

    int getLength() const { return m_length; }
};

#endif
```

```
int main()
{
    // Declare an array with 10 elements
    IntArray array(10);

    // Fill the array with numbers 1 through 10
    for (int i{ 0 }; i<10; ++i)
        array[i] = i+1;

    // Resize the array to 8 elements
    array.resize(8);

    // Insert the number 20 before element with index 5
    array.insertBefore(20, 5);

    // Remove the element with index 3
    array.remove(3);

    // Add 30 and 40 to the end and beginning
    array.insertAtEnd(30);
    array.insertAtBeginning(40);

    // A few more tests to ensure copy constructing / assigning arrays
    // doesn't break things
    {
        IntArray b{ array };
        b = array;
        b = b;
        array = array;
    }

    // Print out all the numbers
    for (int i{ 0 }; i<array.getLength(); ++i)
        std::cout << array[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

This produces the result:

```
40 1 2 3 5 20 6 7 8 30
```

std::initializer_list

when we try to initiate a new object of type IntArray; using a list -> it wont compile because there is no constructor to accept these initializer lists.

```
int main()
{
    // What happens if we try to use an initializer list with this container class?
    IntArray array { 5, 4, 3, 2, 1 }; // this line doesn't compile
    for (int count{ 0 }; count < 5; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

when the compiler sees an initialiser list; it automatically converts it into an object of type std::initializer_list so if our constructor took that as the type of parameter; we will create objects with initialiser lists as an input.

```
IntArray(std::initializer_list<int> list) // allow IntArray to be initialized via list initialization
    : IntArray(static_cast<int>(list.size())) // use delegating constructor to set up initial array
{
    // Now initialize our array from the list
    std::copy(list.begin(), list.end(), m_data);
}
```

and now this should work

```
int main()
{
    IntArray array{ 5, 4, 3, 2, 1 }; // initializer list
    for (int count{ 0 }; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

also non-empty initializer lists will always favor a matching initializer_list constructor over other potentially matching constructors.

```
1  IntArray a1(5);   // uses IntArray(int), allocates an array of size 5
2  IntArray a2{ 5 }; // uses IntArray<std::initializer_list<int>, allocates array of size 1
```

this problem is avoided due to direct initialisation here

```
IntArray(std::initializer_list<int> list)
    : IntArray(static_cast<int>(list.size())) // direct initialization
```

if we had used {} instead of () here; C++ would look for a constructor that takes a std::initializer_list<int> to match the brace syntax but we are already in that constructor so it would be a recursive loop error

```
IntArray(std::initializer_list<int> list)
    : IntArray{static_cast<int>(list.size())} // brace init (list init)
```

adding list constructors to classes can be dangerous; if we had constructed an object with curly braces & down the line we added list constructors; the object will now choose the list constructor over its normal constructor (**()**) changing the final result even though we just added an constructor & dint change anything warranting a change to the result.

assigning class-objects using std::initializer_list

```
#include <algorithm> // for std::copy()
#include <cassert>   // for assert()
#include <initializer_list> // for std::initializer_list
#include <iostream>

class IntArray
{
private:
    int m_length{};
    int* m_data{};

public:
    IntArray() = default;

    IntArray(int length)
        : m_length{ length }
        , m_data{ new int[static_cast<std::size_t>(length)] {} }
    {

    }

    IntArray(std::initializer_list<int> list) // allow IntArray to be initialized via list initialization
        : IntArray(static_cast<int>(list.size())) // use delegating constructor to set up initial array
    {
        // Now initialize our array from the list
        std::copy(list.begin(), list.end(), m_data);
    }

    ~IntArray()
    {
        delete[] m_data;
    }

//  IntArray(const IntArray&) = delete; // to avoid shallow copies
//  IntArray& operator=(const IntArray& list) = delete; // to avoid shallow copies

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() const { return m_length; }
};

int main()
{
    IntArray array{};
    array = { 1, 3, 5, 7, 9, 11 }; // Here's our list assignment statement

    for (int count{ 0 }; count < array.getLength(); ++count)
        std::cout << array[count] << ' '; // undefined behavior

    return 0;
}
```

First, the compiler will note that an assignment function taking a std::initializer_list doesn't exist. Next it will look for other assignment functions it could use, and discover the implicitly provided copy assignment operator. However, this function can only be used if it can convert the initializer list into an IntArray. Because { 1, 3, 5, 7, 9, 11 } is a std::initializer_list, the compiler will use the list constructor to convert the initializer list into a temporary IntArray. Then it will call the implicit assignment operator, which will shallow copy the temporary IntArray into our array object.

At this point, both the temporary IntArray's m_data and array->m_data point to the same address (due to the shallow copy). You can already see where this is going.

At the end of the assignment statement, the temporary IntArray is destroyed. That calls the destructor, which deletes the temporary IntArray's m_data. This leaves array->m_data as a dangling pointer. When you try to use array->m_data for any purpose (including when array goes out of scope and the destructor goes to delete m_data), you'll get undefined behavior.

So if you need to implement a constructor that takes a std::initializer_list, you should ensure you do at least one of the following:

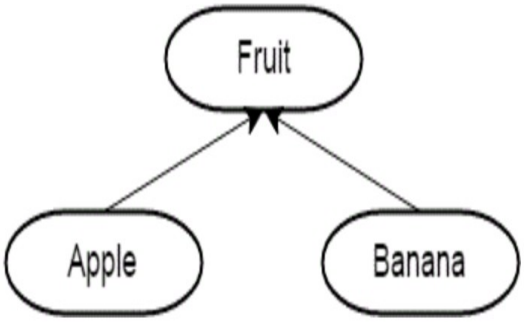
1. Provide an overloaded list assignment operator
2. Provide a proper deep-copying copy assignment operator

3. Delete the copy assignment operator

5)inheritance

object composition to build complex classes from simple structured classes is not the only way build these classes. the second way to do it is inheritance & it models a **“is-a”** relationship.

Consider apples and bananas. Although apples and bananas are different fruits, both have in common that they *are* fruits. And because apples and bananas are fruits, simple logic tells us that anything that is true of fruits is also true of apples and bananas. For example, all fruits have a name, a color, and a size. Therefore, apples and bananas also have a name, a color, and a size. We can say that apples and bananas inherit (acquire) all of the properties of fruit because they *are* fruit. We also know that fruit undergoes a ripening process, by which it becomes edible. Because apples and bananas are fruit, we also know that apples and bananas will inherit the behavior (method function) of ripening.



the class being inherited from is called the **parent class**, **base class**, or **superclass**, and the class doing the inheriting is called the **child class**, **derived class**, or **subclass**. In the above diagram, Fruit is the parent, and both Apple and Banana are children. A child class inherits both behaviors (member functions) and properties (member variables) from the parent (subject to some access restrictions

so for example a person class designed to represent the generic person and needs datamembers like name & age that every person has while the baseballplayer class would have datamembers specific to baseball like batting average & hit homeruns but we also want to track the player’s name & age.

We have three choices for how to add name and age to BaseballPlayer:

1. Add name and age to the BaseballPlayer class directly as members. This is probably the worst choice, as we’re duplicating code that already exists in our Person class. Any updates to Person will have to be made in BaseballPlayer too.
2. Add Person as a member of BaseballPlayer using composition. But we have to ask ourselves, “does a BaseballPlayer have a Person”? No, it doesn’t. So this isn’t the right paradigm.
3. Have BaseballPlayer inherit those attributes from Person. Remember that inheritance represents an is-a relationship. Is a BaseballPlayer a Person? Yes, it is. So inheritance is a good choice here.

```
#include <string>
#include <string_view>

class Person
{
// In this example, we're making our members public for simplicity
public:
    std::string m_name{};
    int m_age{};

    Person(std::string_view name = "", int age = 0)
        : m_name{ name }, m_age{ age }
    {
    }

    const std::string& getName() const { return m_name; }
    int getAge() const { return m_age; }

};
```

```
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage{};
    int m_homeRuns{};

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage{battingAverage}, m_homeRuns{homeRuns}
    {
    }
};
```

BaseballPlayer class objects now will have 4 datamembers: 2 from itself & 2 from person class.

We can also derive more classes from person class like the employee class below

```
#include <iostream>
#include <string>
#include <string_view>

class Person
{
public:
    std::string m_name{};
    int m_age{};

    Person(std::string_view name = "", int age = 0)
        : m_name{name}, m_age{age}
    {
    }

    const std::string& getName() const { return m_name; }
    int getAge() const { return m_age; }

};

// Employee publicly inherits from Person
class Employee: public Person
{
public:
    double m_hourlySalary{};
    long m_employeeID{};

    Employee(double hourlySalary = 0.0, long employeeID = 0)
        : m_hourlySalary{hourlySalary}, m_employeeID{employeeID}
    {
    }

    void printNameAndSalary() const
    {
        std::cout << m_name << ": " << m_hourlySalary << '\n';
    }
};
```

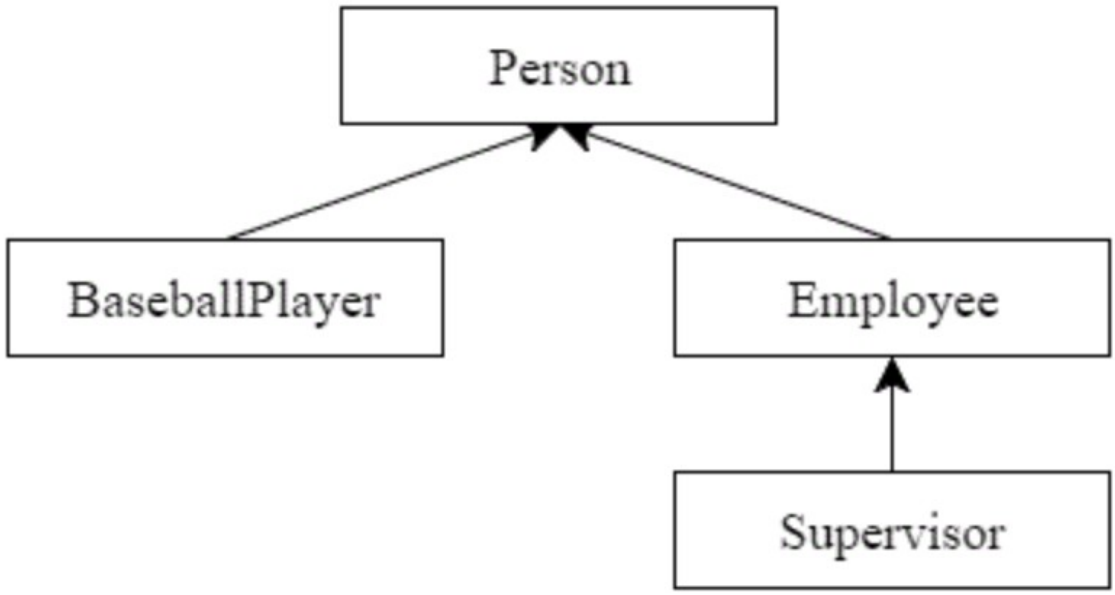
```
int main()
{
    Employee frank{20.25, 12345};
    frank.m_name = "Frank"; // we can do this because m_name is public

    frank.printNameAndSalary();

    return 0;
}
```

its also possible to inherit from a class that is already inheriting some other class

```
class Supervisor: public Employee
{
public:
    // This Supervisor can oversee a max of 5 employees
    long m_overseesIDs[5]{};
};
```



Inheritance is useful because

1) we can have general basic class at top and build specificity as each level of inheritance increases, giving us a set of reusable classes of inheritance chains

2) we don't have to redefine the information from the base class in our derived classes. We automatically receive the member functions and member variables of the base class through inheritance, and then simply add the additional functions or member variables we want. This not only saves work, but also means that if we ever update or modify the base class (e.g. add new functions, or fix a bug), all of our derived classes will automatically inherit the changes!

3) we may not always have permission to access or change the parent base classes(& it could also be risky), so if we want to add functionality to base classes like new functions, we must add it in derived classes to function on data members of base parent class.

one important thing to know while doing this is that, the functions will only work on objects of derived classes so a sole parent object will not be able to access these functions even if the functions literally only use data members from parent class. these functions can only be used on derived objects from classes where the function is defined.

4) When a member function is called on a derived class object, the compiler first looks to see if any function with that name exists in the derived class. If so, all overloaded functions with that name are considered, and the function overload resolution process is used to determine whether there is a best match. If not, the compiler walks up the inheritance chain, checking each parent class in turn in the same way. For example if we wanted to call print(int) but in derived class there is only print(double) but a parent base class has its print(int) & print(double), print(double) of derived class will be called on the derived object. one way to workaround this is using parentclassname::print; in the derived class public space to let the code know all versions of print in parent class mentioned are open for function overload resolution process.

We would need to use scope resolution operator in derived classes to determine which function call (parentclassname::functionname();) everytime we want to call parent function while just function name will work while calling the function inside the same class. friend functions are not considered part of base class & hence cannot be called with a scope resolution operator. we would need to use static cast, especially for operator overloading

```
class Base
{
public:
    Base() { }

    friend std::ostream& operator<< (std::ostream& out, const Base&)
    {
        out << "In Base\n";
        return out;
    }
};

class Derived: public Base
{
public:
    Derived() { }

    friend std::ostream& operator<< (std::ostream& out, const Derived& d)
    {
        out << "In Derived\n";
        // static_cast Derived to a Base object, so we call the right version of operator<<
        out << static_cast<const Base&>(d);
        return out;
    }
};
```

because a `Derived` is-a Base, we can `static_cast` our `Derived` object into a `Base` reference, so that the appropriate version of `operator<<` that uses a `Base` is called. In this example specifically we are calling almost the same functionality but with additional function for calling it on derived objects in place of base objects. this is very common use case for inheritance in adding functionality & specificity to existing functions.

order of construction of inherited classes

```
#include <iostream>

class Base
{
public:
    int m_id {};

    Base(int id=0)
        : m_id { id }
    {
        std::cout << "Base\n";
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0)
        : m_cost { cost }
    {
        std::cout << "Derived\n";
    }

    double getCost() const { return m_cost; }
};

int main()
{
    std::cout << "Instantiating Base\n";
    Base base;

    std::cout << "Instantiating Derived\n";
    Derived derived;

    return 0;
}
```

Because Derived inherits functions and variables from Base, you may assume that the members of Base are copied into Derived. However, this is not true. Instead, we can consider Derived as a two part class: one part Derived, and one part Base.

When C++ constructs derived objects, it does so in phases. First, the most-base class (at the top of the inheritance tree) is constructed. Then each child class is constructed in order, until the most-child class (at the bottom of the inheritance tree) is constructed last.

```
Instantiating Base
Base
Instantiating Derived
Base
Derived
```

initialising data members in parent classes

```
class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0, int id=0)
        // does not work
        : m_cost{ cost }
        , m_id{ id }
    {
    }

    double getCost() const { return m_cost; }
};
```

this does not work! as c++ does not allow initialising of inherited member variables in initialiser list of constructor because if the inherited member variables were const; they would need to be given a value at the time of creation potentially changing the value when the derived class calls its constructor. By downright restricting the initialization of variables to the constructor of the class those variables belong to, C++ ensures that all variables are initialized only once.

```
class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0, int id=0)
        : m_cost{ cost }
    {
        m_id = id;
    }

    double getCost() const { return m_cost; }
};
```

this is not downright restricted by the rules of c++ & will work but obviously experiencing similar problems as before will cause errors if parent data members were const or reference variables.

In all of the examples so far, when we instantiate a Derived class object, the Base class portion has been created using the default Base constructor. Why does it always use the default Base constructor? Because we never told it to do otherwise!

Fortunately, C++ gives us the ability to explicitly choose which Base class constructor will be called!

```
class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0, int id=0)
        : Base{ id } // Call Base(int) constructor with value id!
        , m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

it doesn't matter where in the Derived constructor member initializer list the Base constructor is called -- it will always execute first.

an example in inheritance chains

```
#include <iostream>

class A
{
public:
    A(int a)
    {
        std::cout << "A: " << a << '\n';
    }
};

class B: public A
{
public:
    B(int a, double b)
    : A{ a }
    {
        std::cout << "B: " << b << '\n';
    }
};

class C: public B
{
public:
    C(int a, double b, char c)
    : B{ a, b }
    {
        std::cout << "C: " << c << '\n';
    }
};

int main()
{
    C c{ 5, 4.3, 'R' };

    return 0;
}
```

When a derived class is destroyed, each destructor is called in the *reverse* order of construction. In the above example, when c is destroyed, the C destructor is called first, then the B destructor, then the A destructor.

inheritance access specifiers

until now the data members have been public because child classes cannot access the private region of parent class; they would need to use public access functions to access these private data members.

c++ has third access specifier **protected:** that is only used in inheritance context. protected data members unlike private data members can be accessed by derived / child classes but not by public.

we also have three forms of inheritance public, private & protected(switch public with private or protected in inheritance syntax). The form of inheritance denotes how the datamembers from derived classes will be accessed in the derived class (if at all they can be accessed). In public inheritance; the public data members of parent class are accessed publicly by derived class, protected data members of parent class as protected by derived class & private data members are not accessible. now as mentioned before; the private data members of parent class are always inaccessible by derived classes.

protected inheritance; brings both public and protected data members of parent class into the protected region for derived classes & private inheritance brings both types into private region

changing inherited data members access level

using parentclassname::functionname_nobracket types in whatever region of derived class will make the function part of that region for derived objects. so if a function had protected access for a derived object but the above is typed in the public region, the function will be part of public space for the derived object. this will obviously not work for private functions and data members. This is also helpful in hiding functionality; so if we type using parentclassname::publicdatamemebername; in the private region of derived class, the public data member from base class will effectively now become a private variable. This is helpful in encapsulating the data of poorly designed base class. the data member may now be private for derived class but its still public for the parent class, so we can still access it after subverting by static_casting to baseparentclass& & directly accessing the datamember.

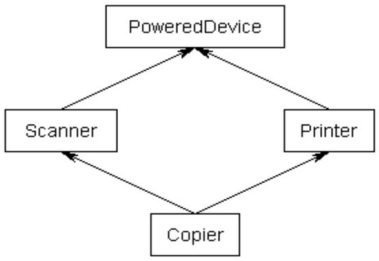
We can also delete functions of parent class for derived objects. so if there is a function called int getValue() const {return m_value;} in the parent class & int getValue() const = delete; in public region of derived class, the function is no longer usable for derived objects.

The only way to now access the getValue() would be
derivedobjectname.parentclassname::getValue(); or
static_cast<parentclassname&>(derivedobjectname).getValue();

multiple inheritance

C++ provides the ability to do multiple inheritance which enables a derived class to inherit members from more than one parent. for ex; class Teacher: public Person, public Employee {}; multiple inheritance is particularly useful in **mixin classes.**

While multiple inheritance seems like a simple extension of single inheritance, multiple inheritance introduces a lot of issues that can markedly increase the complexity of programs and make them a maintenance nightmare. Let's take a look at some of these situations. First, ambiguity can result when multiple base classes contain a function with the same name. Second, and more serious is the diamond problem,



There are many issues that arise in this context, including whether Copier should have one or two copies of PoweredDevice, and how to resolve certain types of ambiguous references. While most of these issues can be addressed through explicit scoping, the maintenance overhead added to your classes in order to deal with the added complexity can cause development time to skyrocket.

As it turns out, most of the problems that can be solved using multiple inheritance can be solved using single inheritance as well. Many object-oriented languages (eg. Smalltalk, PHP) do not even support multiple inheritance.

Many relatively modern languages such as Java and C# restrict classes to single inheritance of normal classes, but allow multiple inheritance of interface classes. The driving idea behind disallowing multiple inheritance in these languages is that it simply makes the language too complex, and ultimately causes more problems than it fixes.

pointers & references to base part of derived objects

```
#include <string_view>

class Base
{
protected:
    int m_value {};

public:
    Base(int value)
        : m_value{ value }
    {
    }

    std::string_view getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    std::string_view getName() const { return "Derived"; }
    int getValueDoubled() const { return m_value * 2; }
};
```

```
#include <iostream>

int main()
{
    Derived derived{ 5 };
    std::cout << "derived is a " << derived.getName() << " and has value " << derived.getValue() << '\n';

    Derived& rDerived{ derived };
    std::cout << "rDerived is a " << rDerived.getName() << " and has value " << rDerived.getValue() << '\n';

    Derived* pDerived{ &derived };
    std::cout << "pDerived is a " << pDerived->getName() << " and has value " << pDerived->getValue() << '\n';

    return 0;
}
```

```
derived is a Derived and has value 5
rDerived is a Derived and has value 5
pDerived is a Derived and has value 5
```

```
#include <iostream>

int main()
{
    Derived derived{ 5 };

    // These are both legal!
    Base& rBase{ derived }; // rBase is an lvalue reference (not an rvalue reference)
    Base* pBase{ &derived };

    std::cout << "derived is a " << derived.getName() << " and has value " << derived.getValue() << '\n';
    std::cout << "rBase is a " << rBase.getName() << " and has value " << rBase.getValue() << '\n';
    std::cout << "pBase is a " << pBase->getName() << " and has value " << pBase->getValue() << '\n';

    return 0;
}
```

```
derived is a Derived and has value 5
rBase is a Base and has value 5
pBase is a Base and has value 5
```

It turns out that because rBase and pBase are a Base reference and pointer, they can only see members of Base (or any

classes that Base inherited). So even though Derived::getName() shadows (hides) Base::getName() for Derived objects, the Base pointer/reference can not see Derived::getName(). Consequently, they call Base::getName(), which is why rBase and pBase report that they are a Base rather than a Derived. this also means it is not possible to call Derived::getValueDoubled() using rBase or pBase. They are unable to see anything in Derived.

```
#include <iostream>
#include <string_view>
#include <string>

class Animal
{
protected:
    std::string m_name;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string_view name)
        : m_name{ name }
    {
    }

    // To prevent slicing (covered later)
    Animal(const Animal&) = delete;
    Animal& operator=(const Animal&) = delete;

public:
    std::string_view getName() const { return m_name; }
    std::string_view speak() const { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const { return "Woof"; }
};

int main()
{
    const Cat cat{ "Fred" };
    std::cout << "cat is named " << cat.getName() << ", and it says " << cat.speak() << '\n';

    const Dog dog{ "Garbo" };
    std::cout << "dog is named " << dog.getName() << ", and it says " << dog.speak() << '\n';

    const Animal* pAnimal{ &cat };
    std::cout << "pAnimal is named " << pAnimal->getName() << ", and it says " << pAnimal->speak() << '\n';

    pAnimal = &dog;
    std::cout << "pAnimal is named " << pAnimal->getName() << ", and it says " << pAnimal->speak() << '\n';

    return 0;
}
```

```
cat is named Fred, and it says Meow
dog is named Garbo, and it says Woof
pAnimal is named Fred, and it says ???
pAnimal is named Garbo, and it says ???
```

these tools are particularly useful for using the attributes of base class in need without having to write identical functions for each class level to use those specific attributes.

another usecase could be that; we want to store the objects in an array (can only store one type of item in one array); so if there are multiple objects of different subclass but all of them had same parent class – we can store them as pointers to their common parent class.

```
#include <array>
#include <iostream>

// Cat and Dog from the example above

int main()
{
    const Cat fred{ "Fred" };
    const Cat misty{ "Misty" };
    const Cat zeke{ "Zeke" };

    const Dog garbo{ "Garbo" };
    const Dog pooky{ "Pooky" };
    const Dog truffle{ "Truffle" };

    // Set up an array of pointers to animals, and set those pointers to our Cat and Dog objects
    const auto animals{ std::to_array<const Animal*>({&fred, &garbo, &misty, &pooky, &truffle, &zeke }) };

    // Before C++20, with the array size being explicitly specified
    // const std::array<const Animal*, 6> animals{ &fred, &garbo, &misty, &pooky, &truffle, &zeke };

    for (const auto animal : animals)
    {
        std::cout << animal->getName() << " says " << animal->speak() << '\n';
    }

    return 0;
}
```

This should help in storing the array comfortably but ultimately; the function will be called on the base part of these objects – not on their derived parts; to call the derived part for the function we use **virtual functions**

virtual functions

special type of member functions that call the most derived function version of the object.

```cpp
#include <iostream>
#include <string_view>

class Base
{
public:
    virtual std::string_view getName() const { return "Base"; } // note addition of virtual keyword
};

class Derived: public Base
{
public:
    virtual std::string_view getName() const { return "Derived"; }
};

int main()
{
    Derived derived {};
    Base& rBase{ derived };
    std::cout << "rBase is a " << rBase.getName() << '\n';

    return 0;
}
```

```
rBase is a Derived
```

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    virtual std::string_view getName() const { return "B"; }
};

class C: public B
{
public:
    virtual std::string_view getName() const { return "C"; }
};

class D: public C
{
public:
    virtual std::string_view getName() const { return "D"; }
};

int main()
{
    C c {};
    A& rBase{ c };
    std::cout << "rBase is a " << rBase.getName() << '\n';

    return 0;
}
```

```
rBase is a C
```

```cpp
C c{};
std::cout << c.getName(); // will always call C::getName

A a { c }; // copies the A portion of c into a (don't do this)
std::cout << a.getName(); // will always call A::getName
```

virtual functions resolution only happen when the function is called through a pointer or reference, calling a function even if virtual on a object will only call the member function belonging to the same object type.

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Animal
{
protected:
    std::string m_name {};

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string_view name)
        : m_name{ name }
    {
    }

public:
    const std::string& getName() const { return m_name; }
    std::string_view speak() const { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const { return "Meow"; }
};
```

```cpp
class Dog: public Animal
{
public:
    Dog(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const { return "Woof"; }
};

void report(const Animal& animal)
{
    std::cout << animal.getName() << " says " << animal.speak() << '\n';
}

int main()
{
    Cat cat{ "Fred" };
    Dog dog{ "Garbo" };

    report(cat);
    report(dog);

    return 0;
}
```

```
Fred says ???
Garbo says ???
```

if we make all the speak() functions in three classes -> virtual; we would call the accurate speak() calls Woof & Meow.

important points

1) virtual function resolution only works as intended -> when the parameter datatypes & number of parameters are exactly same. also the virtual keyword must be used in the function of base / parent class most importantly and this makes all the functions with similar name / parameters to the most derived class; automatically virtual regardless of, if you use keyword or not. This does not work in reverse so if there is a base pointer pointing to base part of the object but the base function is not virtual; the function will always call the base function and not the most virtualised derived function even if the later classes have virtual functions with similar names.

2) virtual function should not called inside the body of constructors; because when the base parent object part is constructed; the derived parts aren't yet constructed so even if the virtual function is called inside the body -> it can not execute derived version of the function.

3) Since most of the time you'll want your functions to be virtual, why not just make all functions virtual? The answer is because it's inefficient -- resolving a virtual function call takes longer than resolving a regular one. Furthermore, to make virtual functions work, the compiler has to allocate an extra pointer for each object of a class that has virtual functions. This adds a lot of overhead to objects that otherwise have a small size.

polymorphism

In programming, **polymorphism** refers to the ability of an entity to have multiple forms (the term "polymorphism" literally means "many forms"). **Compile-time polymorphism** refers to forms of polymorphism that are resolved by the compiler. These include function overload resolution, as well as template resolution. **Runtime polymorphism** refers to forms of polymorphism that are resolved at runtime. This includes virtual function resolution.

override & final specifier

these identifiers are not keywords -- they are normal words that have special meaning only when used in certain contexts. The C++ standard calls them "identifiers with special meaning", but they are often referred to as "specifiers".

```cpp
#include <string_view>

class A
{
public:
    virtual std::string_view getName1(int x) { return "A"; }
    virtual std::string_view getName2(int x) { return "A"; }
    virtual std::string_view getName3(int x) { return "A"; }
};

class B : public A
{
public:
    std::string_view getName1(short int x) override { return "B"; } // compile error, function is not an override
    std::string_view getName2(int x) const override { return "B"; } // compile error, function is not an override
    std::string_view getName3(int x) override { return "B"; } // okay, function is an override of A::getName3(int)

};

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

sometimes we expect the base function to be overridden to execute some derived virtual function but since the data parameter is slightly different or the number of parameters are different, in this case we would want to know during compilation if the virtual functions in later derived classes cannot actually be overridden to; from some base function being called. If the derived function has no similar base function but has override specified; it will give an error.

also the **coding practice** is to write virtual for base function but not for derived functions (as they automatically are virtual) & only mention override; always (no performance penalty)

```cpp
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B : public A
{
public:
    // note use of final specifier on following line -- that makes this function not able to be overridden in derived
classes
    std::string_view getName() const override final { return "B"; } // okay, overrides A::getName()
};

class C : public B
{
public:
    std::string_view getName() const override { return "C"; } // compile error: overrides B::getName(), which is
final
};
```

final keyword prevents any overrides from that point onwards; here the code will not compile at all because the existence of a function with similar signature (paremeter types and number) is present in derived class. removing the override specifier from C::getName() will not fix this & we would need to delete the whole function itself. now calling A::getName() will call B::getname().

```cpp
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B final : public A // note use of final specifier here
{
public:
    std::string_view getName() const override { return "B"; }
};

class C : public B // compile error: cannot inherit from final class
{
public:
    std::string_view getName() const override { return "C"; }
};
```

you can also specify classes as final; making inheriting from these classes uncompilable.

finally for cases when we want to call the Base version of function for a Base pointer pointing to a derived object but the function is virtual in the base; we can just use scope resolution operator. pointer_pointing_to_object.Base::functionname()

to understand the implementation process of virtual functions deeply
https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/
https://www.learncpp.com/cpp-tutorial/the-virtual-table/

covariant return types – one special case where you can do virtual function resolution override to a derived version of function even if the return types are different. Its when the return type is a pointer or reference to base and derived objects.

```cpp
#include <iostream>
#include <string_view>

class Base
{
public:
    // This version of getThis() returns a pointer to a Base class
    virtual Base* getThis() { std::cout << "called Base::getThis()\n"; return this; }
    void printType() { std::cout << "returned a Base\n"; }
};

class Derived : public Base
{
public:
    // Normally override functions have to return objects of the same type as the base function
    // However, because Derived is derived from Base, it's okay to return Derived* instead of Base*
    Derived* getThis() override { std::cout << "called Derived::getThis()\n";  return this; }
    void printType() { std::cout << "returned a Derived\n"; }
};

int main()
{
    Derived d{};
    Base* b{ &d };
    d.getThis()->printType(); // calls Derived::getThis(), returns a Derived*, calls Derived::printType
    b->getThis()->printType(); // calls Derived::getThis(), returns a Base*, calls Base::printType

    return 0;
}
```

```
called Derived::getThis()
returned a Derived
called Derived::getThis()
returned a Base
```

virtual destructors

```cpp
#include <iostream>
class Base
{
public:
    virtual ~Base() // note: virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    virtual ~Derived() // note: virtual
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

if the destructors here in Base & Derived classes weren't virtual; deleting base would only call the Base class destructor while the Derived destructor will not called causing m_array to not be deleted.

Since destuctors are mostly implicitly created; we don't have to write the destructor code just to make it virtual every time in all the classes. just need to write one virtual destructor in base class & all the derived destructors (even implicitly created ones) will be virtual.

if the base class destructor isn't marked as virtual, then the program is at risk for leaking memory if a programmer later deletes a base class pointer that is pointing to a derived object which can be easily avoided by making your destructors virtual but the performance penalty must be taken into the balance sheet

If a class isn't explicitly designed to be a base class, then it's generally better to have no virtual members and no virtual destructor. The class can still be used via composition. If a class is designed to be used as a base class and/or has any virtual functions, then it should always have a virtual destructor.

- If you intend your class to be inherited from, make sure your destructor is virtual and public.
- If you do not intend your class to be inherited from, mark your class as final. This will prevent other classes from inheriting from it in the first place, without imposing any other use restrictions on the class itself.

pure virtual functions

```
#include <string_view>

class Base
{
public:
    std::string_view sayHi() const { return "Hi"; } // a normal non-virtual function

    virtual std::string_view getName() const { return "Base"; } // a normal virtual function

    virtual int getValue() const = 0; // a pure virtual function

    int doSomething() = 0; // Compile error: can not set non-virtual functions to 0
};
```

pure virtual functions are virtual functions with no body & acts as a placeholder for derived version of function to execute. a class with pure virtual functions (one or more) is considered an **abstract base class.** you can not instantiate objects of abstract base class separately. any class with a pure virtual function should have a virtual destructor.

pure virtual functions can have definitions if provided separately. this separate definition can then be called by derived version of function if they want to choose to, (like a default case that you don't have to repeatedly write, in cases where the class does not have any specific function to execute.

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Animal // This Animal is an abstract base class
{
protected:
    std::string m_name {};

public:
    Animal(std::string_view name)
        : m_name(name)
    {
    }

    const std::string& getName() const { return m_name; }
    virtual std::string_view speak() const = 0; // note that speak is a pure virtual function

    virtual ~Animal() = default;
};

std::string_view Animal::speak() const
{
    return "buzz"; // some default implementation
}

class Dragonfly: public Animal
{

public:
    Dragonfly(std::string_view name)
        : Animal{name}
    {
    }

    std::string_view speak() const override// this class is no longer abstract because we defined this function
    {
        return Animal::speak(); // use Animal's default implementation
    }
};

int main()
{
    Dragonfly dfly{"Sally"};
    std::cout << dfly.getName() << " says " << dfly.speak() << '\n';

    return 0;
}
```

interface class

An **interface class** is a class that has no data member variables, and where *all* of the functions are pure virtual! Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class. Interface classes have become extremely popular because they are easy to use, easy to extend, and easy to maintain. In fact, some modern languages, such as Java and C#, have added an “interface” keyword that allows programmers to directly define an interface class without having to explicitly mark all of the member functions as abstract. Furthermore, although Java and C# will not let you use multiple inheritance on normal classes, they will let you multiple inherit as many interfaces as you like. Because interfaces have no data and no function bodies, they avoid a lot of the traditional problems with multiple inheritance while still providing much of the flexibility.

diamond problem

```
#include <iostream>

class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
        std::cout << "PoweredDevice: " << power << '\n';
    }
};

class Scanner: public PoweredDevice
{
public:
    Scanner(int scanner, int power)
        : PoweredDevice{ power }
    {
        std::cout << "Scanner: " << scanner << '\n';
    }
};

class Printer: public PoweredDevice
{
public:
    Printer(int printer, int power)
        : PoweredDevice{ power }
    {
        std::cout << "Printer: " << printer << '\n';
    }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power)
        : Scanner{ scanner, power }, Printer{ printer, power }
    {
    }
};
```

If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class -- one from Printer, and one from Scanner. While this is often desired, other times you may want only one copy of PoweredDevice to be shared by both Scanner and Printer. To share a base class, simply insert the “virtual” keyword in the inheritance list of the derived class. This creates what is called a **virtual base class**, which means there is only one base object. The base object is shared between all objects in the inheritance tree and it is only constructed once.

Here is an example (without constructors for simplicity) showing how to use the virtual keyword to create a shared base class:

```
class PoweredDevice
{
};

class Scanner: virtual public PoweredDevice
{
};

class Printer: virtual public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```

However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it? The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly:

Few Important Details -

First, for the constructor of the most derived class, virtual base classes are always created before non-virtual base classes, which ensures all bases get created before their derived classes.

Second, note that the Scanner and Printer constructors still have calls to the PoweredDevice constructor. When creating an instance of Copier, these constructor calls are simply ignored because Copier is responsible for creating the PoweredDevice, not Scanner or Printer. However, if we were to create an instance of Scanner or Printer, those constructor calls would be used, and normal inheritance rules apply.

Third, if a class inherits one or more classes that have virtual parents, the *most* derived class is responsible for constructing the virtual base class. In this case, Copier inherits Printer and Scanner, both of which have a PoweredDevice virtual base class. Copier, the most derived class, is responsible for creation of PoweredDevice. Note that this is true even in a single inheritance case: if Copier singly inherited from Printer, and Printer was virtually inherited from PoweredDevice, Copier is still responsible for creating PoweredDevice.

Fourth, all classes inheriting a virtual base class will have a virtual table, even if they would normally not have one otherwise, and thus instances of the class will be larger by a pointer. Because Scanner and Printer derive virtually from PoweredDevice, Copier will only be one PoweredDevice subobject. Scanner and Printer both need to know how to find that single PoweredDevice subobject, so they can access its members (because after all, they are derived from it). This is typically done through some virtual table magic (which essentially stores the offset from each subclass to the PoweredDevice subobject).

object slicing

```
int main()
{
    Derived derived{ 5 };
    Base base{ derived }; // what happens here?
    std::cout << "base is a " << base.getName() << " and has value " << base.getValue() << '\n';

    return 0;
}
```

assigning a derived object to a base object only assigns the base part of the object.

slicing can happen accidentally like a function having parameter of base type in value (not reference as passing by reference will just give the whole object up for virtual function resolution) and then passing a derived object in the function call.

another problem is when we have a array/vector; we need to declare the type which if it was base parent type, even if input elements of derived objects are assigned – they would be sliced and only the base objects will be copied. since array of references is not allowed, we can just use an array of pointers pointing to the base type.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<Base*> v{};

    Base b{ 5 }; // b and d can't be anonymous objects
    Derived d{ 6 };

    v.push_back(&b); // add a Base object to our vector
    v.push_back(&d); // add a Derived object to our vector

    // Print out all of the elements in our vector
    for (const auto* element : v)
        std::cout << "I am a " << element->getName() << " with value " << element->getValue() << '\n';

    return 0;
}
```

```
I am a Base with value 5
I am a Derived with value 6
```

frankenobject

```
int main()
{
    Derived d1{ 5 };
    Derived d2{ 6 };
    Base& b{ d2 };

    b = d1; // this line is problematic

    return 0;
}
```

The first three lines in the function are pretty straightforward. Create two Derived objects, and set a Base reference to the second one. The fourth line is where things go astray. Since b points at d2, and we're assigning d1 to b, you might think that the result would be that d1 would get copied into d2 -- and it would, if b were a Derived. But b is a Base, so only the base portion of d1 is copied into d2 & the derived portion remains same.

dynamic_casting

C++ will implicitly let you convert a Derived pointer into a Base pointer. This process is sometimes called **upcasting**.

```
Derived d;
Base* basePtr = &d;  // Implicit upcast
```

C++ provides a casting operator named **dynamic_cast** that can be used for just this purpose. Although dynamic casts have a few different capabilities, by far the most common use for dynamic casting is for converting base-class pointers into derived-class pointers. This process is called **downcasting**.

```
Derived* d{ dynamic_cast<Derived*>(b) }; // use dynamic cast to convert Base pointer into Derived pointer

std::cout << "The name of the Derived is: " << d->getName() << '\n';

delete b;
```

here b is base* pointing to a derived object

it is also good practice to check if the dynamic cast was succesfull

```
Derived* d{ dynamic_cast<Derived*>(b) }; // use dynamic cast to convert Base pointer into Derived pointer

if (d) // make sure d is non-null
    std::cout << "The name of the Derived is: " << d->getName() << '\n';

delete b;
```

Also note that there are several cases where downcasting using dynamic_cast will not work:

1. With protected or private inheritance.
2. For classes that do not declare or inherit any virtual functions (and thus don't have a virtual table).
3. In certain cases involving virtual base classes

It turns out that downcasting can also be done with static_cast. The main difference is that static_cast does no runtime type checking to ensure that what you're doing makes sense. This makes using static_cast faster, but more dangerous. If you cast a Base* to a Derived*, it will "succeed" even if the Base pointer isn't pointing to a Derived object. This will result in undefined behavior when you try to access the resulting Derived pointer (that is actually pointing to a Base object).

```
    if (b->getClassID() == ClassID::derived)
    {
        // We already proved b is pointing to a Derived object, so this should always succeed
        Derived* d{ static_cast<Derived*>(b) };
        std::cout << "The name of the Derived is: " << d->getName() << '\n';
    }

    delete b;
```

we can also use dynamic_cast with references.

```
   Derived& d{ dynamic_cast<Derived&>(b) }; // dynamic cast using a reference instead of a pointer

   std::cout << "The name of the Derived is: " << d.getName() << '\n'; // we can access Derived::getName through d
```

virtualising operator overload functions

```
#include <iostream>

class Base
{
public:
    virtual void print() const { std::cout << "Base"; }

    friend std::ostream& operator<<(std::ostream& out, const Base& b)
    {
        out << "Base";
        return out;
    }
};

class Derived : public Base
{
public:
    void print() const override { std::cout << "Derived"; }

    friend std::ostream& operator<<(std::ostream& out, const Derived& d)
    {
        out << "Derived";
        return out;
    }
};


int main()
{
    Derived d{};
    Base& bref{ d };
    std::cout << bref << '\n';

    return 0;
}
```

```
Base
```

This happens because our version of operator<< that handles Base objects isn't virtual, so std::cout << bref calls the version of operator<< that handles Base objects rather than Derived objects. we cannot make operator overload functions virtual because most of the times they are implemented as friends & we can only make member functions virtual (friend functions are not member functions). even if we made them member function and then proceed to virtualise it, there's the problem that the function parameters for Base::operator<< and Derived::operator<< differ (the Base version would take a Base parameter and the Derived version would take a Derived parameter) so to solve this issue

First, we set up `operator<<` as a friend in our base class as usual. But rather than have `operator<<` determine what to print, we will instead have it call a normal member function that *can* be virtualized!

```
#include <iostream>

class Base
{
public:
    // Here's our overloaded operator<<
    friend std::ostream& operator<<(std::ostream& out, const Base& b)
    {
        // Call virtual function identify() to get the string to be printed
        out << b.identify();
        return out;
    }

    // We'll rely on member function identify() to return the string to be printed
    // Because identify() is a normal member function, it can be virtualized
    virtual std::string identify() const
    {
        return "Base";
    }
};
```

```
class Derived : public Base
{
public:
    // Here's our override identify() function to handle the Derived case
    std::string identify() const override
    {
        return "Derived";
    }
};

int main()
{
    Base b{};
    std::cout << b << '\n';

    Derived d{};
    std::cout << d << '\n'; // note that this works even with no operator<< that explicitly handles Derived objects

    Base& bref{ d };
    std::cout << bref << '\n';

    return 0;
}
```

```
Base
Derived
Derived
```

The above solution works great, but has two potential shortcomings:

1. It makes the assumption that the desired output can be represented as a single std::string.
2. Our `identify()` member function does not have access to the stream object.

In the previous version, virtual function `identify()` returned a string to be printed by `Base::operator<<`. In the next version, we'll instead define virtual member function `print()` and delegate responsibility for printing *directly* to that function.

```
#include <iostream>

class Base
{
public:
    // Here's our overloaded operator<<
    friend std::ostream& operator<<(std::ostream& out, const Base& b)
    {
        // Delegate printing responsibility for printing to virtual member function print()
        return b.print(out);
    }

    // We'll rely on member function print() to do the actual printing
    // Because print() is a normal member function, it can be virtualized
    virtual std::ostream& print(std::ostream& out) const
    {
        out << "Base";
        return out;
    }
};

// Some class or struct with an overloaded operator<<
struct Employee
{
    std::string name{};
    int id{};

    friend std::ostream& operator<<(std::ostream& out, const Employee& e)
    {
        out << "Employee(" << e.name << ", " << e.id << ")";
        return out;
    }
};

class Derived : public Base
{
private:
    Employee m_e{}; // Derived now has an Employee member

public:
    Derived(const Employee& e)
        : m_e{ e }
    {
    }

    // Here's our override print() function to handle the Derived case
    std::ostream& print(std::ostream& out) const override
    {
        out << "Derived: ";

        // Print the Employee member using the stream object
        out << m_e;

        return out;
    }
};
```