In **procedural programming**, the focus is on creating “procedures” (which in C++ are called functions) that implement our program logic. We pass data objects to these functions, those functions perform operations on the data, and then potentially return a result to be used by the caller. In programming, properties are represented by objects, and behaviors are represented by functions. And thus, procedural programming represents reality fairly poorly, as it separates properties (objects) and its behaviors (functions).

object in C++ as, “a piece of memory that can be used to store values”. An object with a name is called a variable. In **object-oriented programming** (often abbreviated as OOP), the focus is on creating program-defined data types that contain both the properties and a set of well-defined behaviours belonging to the object. The term “object” in OOP refers to the objects that we can instantiate from such types.

problem with structs

a **class invariant** is a condition that must be true throughout the lifetime of an object in order for the object to remain in a valid state while an object that has a violated class invariant is said to be in an **invalid state**, and unexpected or undefined behavior may result from further use of that object, so for example if a struct holds numerator and denominator for a fraction, we would prefer the denominator not be zero(this is the class invariant condition) especially if a function exists that calculates the absolute value of fraction but such rules are hard to implement elegantly in structs. Ideally, we’d love to bulletproof our class types so that an object either can’t be put into an invalid state, or can signal immediately if it is (rather than letting undefined behaviour occur at some random point in the future).

Introduction to classes

Just like structs, a **class** is a program-defined compound type that can have many member variables with different types. You have already been using class objects, perhaps without knowing it. Both `std::string` and `std::string_view` are defined as classes. In fact, most of the non-aliased types in the standard library are defined as classes! Classes are really the heart and soul of C++ -- they are so foundational that C++ was originally named “C with classes”! Once you are familiar with classes, much of your time in C++ will be spent writing, testing, and using them.

```
#include <iostream>

class Date       // we changed struct to class
{
public:          // and added this line, which is called an access specifier
    int m_day{}; // and added "m_" prefixes to each of the member names
    int m_month{};
    int m_year{};
};

void printDate(const Date& date)
{
    std::cout << date.m_day << '/' << date.m_month << '/' << date.m_year;
}

int main()
{
    Date date{ 4, 10, 21 };
    printDate(date);

    return 0;
}
```

largely similar syntax to structs

member functions

just like real-life objects where each object has set of functions unique to it, class types (structs, classes, and unions) can also have their own functions called member functions.

```
// Member function version
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() // defines a member function named print
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

int main()
{
    Date today { 2020, 10, 14 }; // aggregate initialize our struct

    today.day = 16; // member variables accessed using member selection operator (.)
    today.print();  // member functions also accessed using member selection operator (.)

    return 0;
}
```

Member functions defined inside the class type definition are implicitly inline, so they will not cause violations of the one-definition rule if the class type definition is included into multiple code files.

complex example to showcase how the objects interact w/ member functions

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

struct Person
{
    std::string name{};
    int age{};

    void kisses(const Person& person)
    {
        std::cout << name << " kisses " << person.name << '\n';
    }
};

int main()
{
    Person joe{ "Joe", 29 };
    Person kate{ "Kate", 27 };

    joe.kisses(kate);

    return 0;
}
```

also all class is seen before compilation so order of declaration does not matter inside a class, things declared later can be used before in class code.

overloading member functions

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

struct Date
{
    int year {};
    int month {};
    int day {};

    void print()
    {
        std::cout << year << '/' << month << '/' << day;
    }

    void print(std::string_view prefix)
    {
        std::cout << prefix << year << '/' << month << '/' << day;
    }
};

int main()
{
    Date today { 2020, 10, 14 };

    today.print(); // calls Date::print()
    std::cout << '\n';

    today.print("The date is: "); // calls Date::print(std::string_view)
    std::cout << '\n';

    return 0;
}
```

History of Structs and member functions

In C++, while designing classes, Bjarne Stroustrup spent some amount of time considering whether structs (which were inherited from C) should be granted the ability to have member functions. Upon consideration, he determined that they should have it. Structs & Classes are effectively the same but different by conventional use with struct being public by default and classes being private by default. Structs are aggregates with no user declared constructors & no private data members, while classes aren't aggregates usually because default itself is private for data-members (being an aggregate allows aggregate initialisation so if a class or struct had a user defined constructor, it cannot be aggregate initialised and aggregate initialisation can only work on public members of the class type)

Class types with no data members

```cpp
#include <iostream>

struct Foo
{
    void printHi() { std::cout << "Hi!\n"; }
};

int main()
{
    Foo f{};
    f.printHi(); // requires object to call

    return 0;
}
```

It will work after f is aggregate initialised of type Foo but its overkill, namespaces are preffered in such case

```cpp
#include <iostream>

namespace Foo
{
    void printHi() { std::cout << "Hi!\n"; }
};

int main()
{
    Foo::printHi(); // no object needed

    return 0;
}
```

const member functions

```cpp
struct Date
{
    int year {};
    int month {};
    int day {};
};

int main()
{
    const Date today { 2020, 10, 14 }; // const class type object

    return 0;
}
```

Once a const class type object has been initialized, any attempt to modify the data members of the object is disallowed, as it would violate the const-ness of the object.

const member functions are needed to used on objects declared const but still work for non-const declared objects while non-const member functions can't work on const objects.

```cpp
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() const // now a const member function
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

int main()
{
    const Date today { 2020, 10, 14 }; // const

    today.print();  // ok: const object can call const member function

    return 0;
}
```

Its also just preffered to pass non const objects via const references then creating const objects

```cpp
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() const // now const
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

void doSomething(const Date& date)
{
    date.print();
}

int main()
{
    Date today { 2020, 10, 14 }; // non-const
    today.print();

    doSomething(today);

    return 0;
}
```

also If you have two member functions with same name(one const and non const) the later intialised objects will call either function based on being const or not.

member access

Each member of a class type has a property called an **access level** that determines who can access that member. C++ has three different access levels: *public*, *private*, and *protected*. In this lesson, we'll cover the two commonly used access levels: public and private.

By default, the members of structs (and unions) are public, and the members of classes are private.

Members that have the *public* access level are called *public members*. **Public members** are members of a class type that do not have any restrictions on how they can be accessed. public members can be accessed by anyone (as long as they are in scope).

```cpp
#include <iostream>

struct Date
{
    // struct members are public by default, can be accessed by anyone
    int year {};        // public by default
    int month {};       // public by default
    int day {};         // public by default

    void print() const // public by default
    {
        // public members can be accessed in member functions of the class type
        std::cout << year << '/' << month << '/' << day;
    }
};

// non-member function main is part of "the public"
int main()
{
    Date today { 2020, 10, 14 }; // aggregate initialize our struct

    // public members can be accessed by the public
    today.day = 16; // okay: the day member is public
    today.print();  // okay: the print() member function is public

    return 0;
}
```

Members that have the *private* access level are called *private members*. **Private members** are members of a class type that can only be accessed by other members of the same class.

```
#include <iostream>

class Date // now a class instead of a struct
{
    // class members are private by default, can only be accessed by other members
    int m_year {};      // private by default
    int m_month {};     // private by default
    int m_day {};       // private by default

    void print() const // private by default
    {
        // private members can be accessed in member functions
        std::cout << m_year << '/' << m_month << '/' << m_day;
    }
};

int main()
{
    Date today { 2020, 10, 14 }; // compile error: can no longer use aggregate initialization

    // private members can not be accessed by the public
    today.m_day = 16; // compile error: the m_day member is private
    today.print();    // compile error: the print() member function is private

    return 0;
}
```

In C++, it is a common convention to name private data members starting with an “m_” prefix

access specifier

Structs should avoid access specifiers altogether, meaning all struct members will be public by default. We want our structs to be aggregates, and aggregates can only have public members.

```
class Date
{
// Any members defined here would default to private

public: // here's our public access specifier

    void print() const // public due to above public: specifier
    {
        // members can access other private members
        std::cout << m_year << '/' << m_month << '/' << m_day;
    }

private: // here's our private access specifier

    int m_year { 2020 };  // private due to above private: specifier
    int m_month { 14 }; // private due to above private: specifier
    int m_day { 10 };   // private due to above private: specifier
};

int main()
{
    Date d{};
    d.print();  // okay, main() allowed to access public members

    return 0;
}
```

Because we have private members, we can not aggregate initialize contents of object d without public setter functions or constructors, default initialisation works in this case.

use of an explicit `private:` specifier makes it clear that the following members are private, without having to infer what the default access level is based on whether classtype was defined as a class or a struct.

Access level	Access specifier	Member access	Derived class access	Public access
Public	public:	yes	yes	yes
Protected	protected:	yes	yes	no
Private	private:	yes	no	no

classes normally have public member functions (so those member functions can be used by the public after the object is created). However, occasionally member functions are made private (or protected) if they are not intended to be used by the public namespace.

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

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

public:
    void kisses(const Person& p) const
    {
        std::cout << m_name << " kisses " << p.m_name << '\n';
    }

    void setName(std::string_view name)
    {
        m_name = name;
    }
};

int main()
{
    Person joe;
    joe.setName("Joe");

    Person kate;
    kate.setName("Kate");

    joe.kisses(kate);

    return 0;
}
```

as you can see member-functions have access to private members of all the objects in class!

access function

An **access function** is a trivial public member function whose job is to retrieve or change the value of a private member variable.

Getters (also sometimes called **accessors**) are public member functions that return the value of a private member variable. Getters are usually made const, so they can be called on both const and non-const objects.

Setters (also sometimes called **mutators**) are public member functions that set the value of a private member variable. Setters should be non-const, so they can modify the data members.

```cpp
#include <iostream>

class Date
{
private:
    int m_year { 2020 };
    int m_month { 10 };
    int m_day { 14 };

public:
    void print()
    {
        std::cout << m_year << '/' << m_month << '/' << m_day << '\n';
    }

    int getYear() const { return m_year; }        // getter for year
    void setYear(int year) { m_year = year; }     // setter for year

    int getMonth() const  { return m_month; }     // getter for month
    void setMonth(int month) { m_month = month; } // setter for month

    int getDay() const { return m_day; }          // getter for day
    void setDay(int day) { m_day = day; }         // setter for day
};

int main()
{
    Date d{};
    d.setYear(2021);
    std::cout << "The year is: " << d.getYear() << '\n';

    return 0;
}
```

naming convention

1)either prefix with get and set for clarity

2)no prefix overloading

```cpp
int day() const { return m_day; }  // getter
void day(int day) { m_day = day; } // setter
```

3) set prefix only

```cpp
int day() const { return m_day; }     // getter
void setDay(int day) { m_day = day; } // setter
```

---

returning data members by lvalue reference

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

class Employee
{
    std::string m_name{};

public:
    void setName(std::string_view name) { m_name = name; }
    const std::string& getName() const { return m_name; } //  getter returns by const reference
};

int main()
{
    Employee joe{}; // joe exists until end of function
    joe.setName("Joe");

    std::cout << joe.getName(); // returns joe.m_name by reference

    return 0;
}
```

here the get function returns by const reference reducing cost of function
we can use const auto& instead of std::string& in member function for easy deduction.

problems with returning with rvalue reference

as rvalue terms go out of scope after end of statement, some problems related to rvalues

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

class Employee
{
    std::string m_name{};

public:
    void setName(std::string_view name) { m_name = name; }
    const std::string& getName() const { return m_name; } //  getter returns by const reference
};

// createEmployee() returns an Employee by value (which means the returned value is an rvalue)
Employee createEmployee(std::string_view name)
{
    Employee e;
    e.setName(name);
    return e;
}

int main()
{
    // Case 1: okay: use returned reference to member of rvalue class object in same expression
    std::cout << createEmployee("Frank").getName();

    // Case 2: bad: save returned reference to member of rvalue class object for use later
    const std::string& ref { createEmployee("Garbo").getName() }; // reference becomes dangling when return value of
createEmployee() is destroyed
    std::cout << ref; // undefined behavior

    // Case 3: okay: copy referenced value to local variable for use later
    std::string val { createEmployee("Hans").getName() }; // makes copy of referenced member
    std::cout << val; // okay: val is independent of referenced member

    return 0;
}
```

don’t ever return non const references to private data members

```cpp
class Foo
{
private:
    int m_value{ 4 }; // private member

public:
    int& value() { return m_value; } // returns a non-const reference (don't do this)
};

int main()
{
    Foo f{};                // f.m_value is initialized to default value 4
    f.value() = 5;          // The equivalent of m_value = 5
    std::cout << f.value(); // prints 5

    return 0;
}
```

a reference acts just like the object being referenced, a member function that returns a non-const reference provides direct access to that member (even if the member is private).

why do we need data hiding (encapsulation)

lets say we have a class type with object members of our name and their first initial, every time a new user changes something in objects of these class types it will be their responsibility to change the first initial accurately in accordance with the new name. this form of access to public class objects could cause various disasters and if these objects were private and could only be change with use of member functions that ensure that initial is automatically changed accurately with respect to new name, it would be a large saver for our code.

When we give users direct access to the implementation of a class, they become responsible for maintaining all invariants -- which they may not do (either correctly, or at all). Putting this burden on the user adds a lot of complexity.

non member functions are preffered

- Non-member functions are not part of the interface of your class. Thus, the interface of your class will be smaller and more straightforward, making the class easier to understand.
- Non-member functions enforce encapsulation, as such functions must work through the public interface of the class. There is no temptation to access the implementation directly just because it is convenient.
- Non-member functions do not need to be considered when making changes to the implementation of a class (so long as the interface doesn't change in an incompatible way).
- Non-member functions tend to be easier to debug.
- Non-member functions containing application specific data and logic can be separated from the reusable portions of the class.

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

class Yogurt
{
    std::string m_flavor{ "vanilla" };

public:
    void setFlavor(std::string_view flavor)
    {
        m_flavor = flavor;
    }

    const std::string& getFlavor() const { return m_flavor; }
};

// Best: non-member function print() is not part of the class interface
void print(const Yogurt& y)
{
        std::cout << "The yogurt has flavor " << y.getFlavor() << '\n';
}

int main()
{
    Yogurt y{};
    y.setFlavor("cherry");
    print(y);

    return 0;
}
```

`print()`non-member function does not directly access any members. If the class implementation ever changes, `print()` will not need to be evaluated at all. Additionally, each application can provide it's own `print()` function that prints exactly how that application wants to.

The order of class member declaration

In code outside classes, it is important to declare variables and functions before we use them, but this limitation does not exist in classes.

In modern C++, listing your public members first, and putting your private members down at the bottom because someone who uses your class is interested in the public interface is fine, putting your public members first makes the information they need up top, and puts the implementation details (which are least important) last.

Constructors

A **constructor** is a special member function that is automatically called when a non-aggregate class type object is created, if a accessible matching constructor is found, object is allocated in memory & if not, then compilation error. it has no return type(not even void) and their name should match the name of class.

```
#include <iostream>

class Foo
{
private:
    int m_x {};
    int m_y {};

public:
    Foo(int x, int y) // here's our constructor function that takes two initializers
    {
        std::cout << "Foo(" << x << ", " << y << ") constructed\n";
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 }; // calls Foo(int, int) constructor
    foo.print();

    return 0;
}
```

```
Foo(6, 7) constructed
Foo(0, 0)
```

When the compiler sees the definition `Foo foo{ 6, 7 }`, it looks for a matching `Foo` constructor that will accept two `int` arguments. `Foo(int, int)` is a match, so the compiler will allow the definition but since private data members aren't initialised yet, `print()` member function will print the `m_x` and `m_y` values which are value initialised.

Constructors will implicitly convert arguments to match existing constructors if no perfect match is found like ('a',true) will be converted to (1,1) to match the (int,int) constructor.

Constructors need to be non-const always (even though non-const functions can't be invoked on a const object, construction process of a const object is exempted from this law)

Constructors are designed to initialize how-much-ever data members but at the point of instantiation of object while Setters are designed to assign a value to data-members of an existing object.

member initialisation list

member initializer list is a constructor function syntax to initialise members of a object

```
#include <iostream>

class Foo
{
private:
    int m_x {};
    int m_y {};

public:
    Foo(int x, int y)
        : m_x { x }, m_y { y } // here's our member initialization list
    {
        std::cout << "Foo(" << x << ", " << y << ") constructed\n";
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 };
    foo.print();

    return 0;
}
```

member initialisation list starts with a colon and doesn't end in a semi colon

If the member initialization list is short/trivial, all initializers can go on one line:

```
Foo(int x, int y)
    : m_x { x }, m_y { y }
{
}
```

Otherwise (or if you prefer), each member and initializer pair can be placed on a separate line (starting with a comma to maintain alignment):

COPY

```
Foo(int x, int y)
    : m_x { x }
    , m_y { y }
{
}
```

In C++ standard, the members are initialised in order they are defined inside the class irrespective of the order they are initialised in member initialisation list.

```
Foo(int x, int y)
    : m_y{ std::max(x, y) }, m_x{ m_y } // issue on this line
```

here m_x is defined first (assume) so it will be initialised first but it can't coz m_y isn't initialised yet causing an error.

we use member-initialisation list for initialisation not assignment, if the values were given inside the constructor function body block scope, it would be assignment, but this would obviously not work on data members that are const or references.

default constructors

A **default constructor** is a constructor function that accepts no arguments.

```
class Foo
{
public:
    Foo() // default constructor
    {
        std::cout << "Foo default constructed\n";
    }
};

int main()
{
    Foo foo{}; // No initialization values, calls Foo's default constructor

    return 0;
}
```

an object of datatype `Foo` is created with name foo, since no initialisation value is present, the default constructor function Foo() is called, which just prints “Foo default constructed”

```
Foo foo{}; // value initialization, calls Foo() default constructor
Foo foo2;  // default initialization, calls Foo() default constructor
```

both create objects of type Foo with default constructor because no initialiser

default constructors with default arguments

```
#include <iostream>

class Foo
{
private:
    int m_x { };
    int m_y { };

public:
    Foo(int x=0, int y=0) // has default arguments
        : m_x { x }
        , m_y { y }
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ") constructed\n";
    }
};

int main()
{
    Foo foo1{};     // calls Foo(int, int) constructor using default arguments
    Foo foo2{6, 7}; // calls Foo(int, int) constructor

    return 0;
}
```

```
Foo(0, 0) constructed
Foo(6, 7) constructed
```

implicit/explicit default constructor

```
#include <iostream>

class Foo
{
private:
    int m_x{};
    int m_y{};

    // Note: no constructors declared
};

int main()
{
    Foo foo{};

    return 0;
}
```

will implicitly declare a constructor of form

```
public:
    Foo() // implicitly generated default constructor
    {
    }
```

Explicitly declaring default constructor in public region for no argument use

```
Foo() = default; // generates an explicitly defaulted default constructor
```

There are at least two cases where the explicitly defaulted default constructor behaves differently than an empty user-defined constructor.

1. When value initializing a class, if the class has a user-defined default constructor, the object will be default initialized. However, if the class has a default constructor that is not user-provided (that is, a default constructor that is either implicitly defined, or defined using `= default`), the object will be zero-initialized before being default initialized.

2. Prior to C++20, a class with a user-defined default constructor (even if it has an empty body) makes the class a non-aggregate, whereas an explicitly defaulted default constructor does not. Assuming the class was otherwise an aggregate, the former would cause the class to use list initialization instead of aggregate initialization. In C++20 onward, this inconsistency was addressed, so that both make the class a non-aggregate.

```cpp
#include <iostream>

class User
{
private:
    int m_a; // note: no default initialization value
    int m_b {};

public:
    User() {} // user-defined empty constructor

    int a() const { return m_a; }
    int b() const { return m_b; }
};

class Default
{
private:
    int m_a; // note: no default initialization value
    int m_b {};

public:
    Default() = default; // explicitly defaulted default constructor

    int a() const { return m_a; }
    int b() const { return m_b; }
};

class Implicit
{
private:
    int m_a; // note: no default initialization value
    int m_b {};

public:
    // implicit default constructor

    int a() const { return m_a; }
    int b() const { return m_b; }
};

int main()
{
    User user{}; // default initialized
    std::cout << user.a() << ' ' << user.b() << '\n';

    Default def{}; // zero initialized, then default initialized
    std::cout << def.a() << ' ' << def.b() << '\n';

    Implicit imp{}; // zero initialized, then default initialized
    std::cout << imp.a() << ' ' << imp.b() << '\n';

    return 0;
}
```

```
782510864 0
0 0
0 0
```

calling other member functions in constructor

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

class Employee
{
private:
    std::string m_name{};
    int m_id{ 0 };

    void printCreated() const
    {
        std::cout << "Employee " << m_name << " created\n";
    }

public:
    Employee(std::string_view name)
        : m_name{ name }
    {
        printCreated();
    }

    Employee(std::string_view name, int id)
        : m_name{ name }, m_id{ id }
    {
        printCreated();
    }
};

int main()
{
    Employee e1{ "James" };
    Employee e2{ "Dave", 42 };
}
```

more dangerous case occurs when we try to explicitly call a constructor without any arguments.

```
#include <iostream>
struct Foo
{
    int x{};
    int y{};

public:
    Foo()
    {
        x = 5;
    }

    Foo(int value): y { value }
    {
        // intent: call Foo() function
        // actual: value initializes nameless temporary Foo (which is then discarded)
        Foo(); // note: this is the equivalent of writing Foo{}
    }
};

int main()
{
    Foo f{ 9 };
    std::cout << f.x << ' ' << f.y; // prints 0 9
}
```

when Foo() is called inside constructor body, a **nameless temporary** object is created which has its x = 5, but this temporary object is immediately discarded because its not used. The x member of Foo(int) constructor is never assigned any value and hence the output is x=0 and y=9. Foo(); is seen as a function declaration due to “most vexing parse -> declaration precedence rule” under normal conditions but since u can’t declare functions inside another functions, it takes it as an object declaration constructor.

Delegating constructors

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

class Employee
{
private:
    std::string m_name{};
    int m_id{ 0 };

public:
    Employee(std::string_view name)
        : Employee{ name, 0 } // delegate initialization to Employee(std::string_view, int) constructor
    {
    }

    Employee(std::string_view name, int id)
        : m_name{ name }, m_id{ id } // actually initializes the members
    {
        std::cout << "Employee " << m_name << " created\n";
    }

};

int main()
{
    Employee e1{ "James" };
    Employee e2{ "Dave", 42 };
}
```

a constructor that delegates to another constructor is not allowed to do any member initialization itself. So your constructors can delegate or initialize, but not both. here, Employee(name) constructor is delegating or transferring the initialisation process to Employee(name,int) constructor and not actually member initialising anything.

always delegate to a non-delegating constructor to avoid a delegating infinite loop.

Reducing constructors using default arguments

```
private:
    std::string m_name{};
    int m_id{ 0 }; // default member initializer

public:

    Employee(std::string_view name, int id = 0) // default argument for id
        : m_name{ name }, m_id{ id }
    {
        std::cout << "Employee " << m_name << " created\n";
    }
```

the two m_id = 0 statements serve different purposes 1) default member initialiser 2) allows m_id to be overridden in case there is a value being assigned to it but still getting zero when there is none

conundrum of redundant constructors vs default values

we used delegating constructors and then default arguments to reduce constructor redundancy, there are many opinions on both sides but it's usually more straightforward to have fewer constructors, even if it results in duplication of initialization values.

temporary class objects

```
#include <iostream>

class IntPair
{
private:
    int m_x{};
    int m_y{};

public:
    IntPair(int x, int y)
        : m_x { x }, m_y { y }
    {}

    int x() const { return m_x; }
    int y() const{ return m_y; }
};

void print(IntPair p)
{
    std::cout << "(" << p.x() << ", " << p.y() << ")\n";
}

int main()
{
    // Case 1: Pass variable
    IntPair p { 3, 4 };
    print(p);

    // Case 2: Construct temporary IntPair and pass to function
    print(IntPair { 5, 6 } );

    // Case 3: Implicitly convert { 7, 8 } to a temporary Intpair and pass to function
    print( { 7, 8 } );

    return 0;
}
```

more common to see temporary objects used with return values:

```
// Case 1: Create named variable and return
IntPair ret1()
{
    IntPair p { 3, 4 };
    return p;
}

// Case 2: Create temporary IntPair and return
IntPair ret2()
{
    return IntPair { 5, 6 };
}

// Case 3: implicitly convert { 7, 8 } to IntPair and return
IntPair ret3()
{
    return { 7, 8 };
}

int main()
{
    print(ret1());
    print(ret2());
    print(ret3());

    return 0;
}
```

```
(3, 4)
(5, 6)
(7, 8)
```

copy constructor

A **copy constructor** is a constructor that is used to initialize an object with an existing object of the same type. After the copy constructor executes, the newly created object should be a copy of the object passed in as the initializer. If you do not provide a copy constructor for your classes, C++ will create a public **implicit copy constructor** for you.

```
Fraction f { 5, 3 };  // Calls Fraction(int, int) constructor
Fraction fCopy { f }; // What constructor is used here?
```

```
Fraction(5, 3)
Fraction(5, 3)
```

explicitly declared copy constructor

if we want a specific action whenever a constructor function constructing from an existing object reference is used, we can declare a copy constructor.

```
Fraction(const Fraction& fraction)
    // Initialize our members using the corresponding member of the parameter
    : m_numerator{ fraction.m_numerator }
    , m_denominator{ fraction.m_denominator }
{
    std::cout << "Copy constructor called\n"; // just to prove it works
}
```

Unlike the implicit default constructor, which does nothing (and thus is rarely what we want), the member wise initialization performed by the implicit copy constructor is usually exactly what we want. Therefore, in most cases, using the implicit copy constructor is perfectly fine.

passing by value via copy constructor

```
void printFraction(Fraction f) // f is pass by value
{
    f.print();
}

int main()
{
    Fraction f{ 5, 3 };

    printFraction(f); // f is copied into the function parameter using copy constructor

    return 0;
}
```

in this case; the copy constructor is called because when an object is passed by value, the argument is copied into the parameter(parameter value is obtained after calling the object constructor with object in argument as reference)

```
Copy constructor called
Fraction(5, 3)
```

```cpp
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

public:
    // Default constructor
    Fraction(int numerator = 0, int denominator = 1)
        : m_numerator{ numerator }, m_denominator{ denominator }
    {
    }

    // Copy constructor
    Fraction(const Fraction& fraction)
        : m_numerator{ fraction.m_numerator }
        , m_denominator{ fraction.m_denominator }
    {
        std::cout << "Copy constructor called\n";
    }

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
};

void printFraction(Fraction f) // f is pass by value
{
    f.print();
}

void printFraction(Fraction f) // f is pass by value
{
    f.print();
}

Fraction generateFraction(int n, int d)
{
    Fraction f{ n, d };
    return f;
}

int main()
{
    Fraction f2 { generateFraction(1, 2) }; // Fraction is returned using copy constructor

    printFraction(f2); // f2 is copied into the function parameter using copy constructor

    return 0;
}
```

When `generateFraction` returns a `Fraction` back to `main`, a temporary `Fraction` object is created and initialized using the copy constructor & since this temporary is used to initialize `Fraction f2`, this invokes the copy constructor again to copy the temporary into `f2`. Finally when `f2` is passed to `printFraction()`, the copy constructor is called a third time.

```
Copy constructor called
Copy constructor called
Copy constructor called
Fraction(1, 2)
```

explicit default copy constructor

C++ has creates a copy constructor automatically if u don't declare one -> doing a memberwise copy of all datamembers or u can declare it explicitly in case u use "special member functions" later that may not declare copy constructors automatically

```cpp
    // Explicitly request default copy constructor
    Fraction(const Fraction& fraction) = default;

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
```

disallowing copy constructor using =delete

```
Fraction(const Fraction& fraction) = delete;
```
will result in compile error for copy constructor calls

You can also prevent the public from making copies of class object by making the copy constructor private (as private functions can't be called by the public). However, a private copy constructor *can* still be called from other members of the class, so this solution is not advised unless that is desired but =delete; works well always.

class object initialisation techniques (same as variable initialisation techniques)

```
#include <iostream>

class Foo
{
public:

    // Default constructor
    Foo()
    {
        std::cout << "Foo()\n";
    }

    // Normal constructor
    Foo(int x)
    {
        std::cout << "Foo(int) " << x << '\n';
    }

    // Copy constructor
    Foo(const Foo&)
    {
        std::cout << "Foo(const Foo&)\n";
    }
};

int main()
{
    // Calls Foo() default constructor
    Foo f1;           // default initialization
    Foo f2{};         // value initialization (preferred)

    // Calls foo(int) normal constructor
    Foo f3 = 3;       // copy initialization (non-explicit constructors only)
    Foo f4(4);        // direct initialization
    Foo f5{ 5 };      // direct list initialization (preferred)
    Foo f6 = { 6 };   // copy list initialization (non-explicit constructors only)

    // Calls foo(const Foo&) copy constructor
    Foo f7 = f3;      // copy initialization
    Foo f8(f3);       // direct initialization
    Foo f9{ f3 };     // direct list initialization (preferred)
    Foo f10 = { f3 }; // copy list initialization

    return 0;
}
```

copy elision

```
Something s { Something { 5 } };
```

Needlessly inefficient as it requires two constructor calls

Copy elision is a compiler optimization technique that allows the compiler to remove unnecessary copying of objects. In other words, in cases where the compiler would normally call a copy constructor, the compiler is free to rewrite the code to avoid the call to the copy constructor altogether. When the compiler optimizes away a call to the copy constructor, we say the constructor has been **elided**. Unlike other types of optimization, copy elision is exempt from the "as-if" rule. That is, copy elision is allowed to elide the copy constructor even if the copy constructor has side effects (such as printing text to the console)! This is why copy constructors should not have side effects other than copying -- if the compiler elides the call to the copy constructor, the side effects won't execute, and the observable behavior of the program will change!

converting constructor

All constructors by default are converting constructors, hence when we are supplying a integer argument to a function taking class object parameter, the integer will call the constructor to return a constructed object with the single integer as the constructor-function search type (int)

```
class Foo
{
private:
    int m_x{};
public:
    Foo(int x)
        : m_x{ x }
    {
    }

    int getX() const { return m_x; }
};

void printFoo(Foo f) // has a Foo parameter
{
    std::cout << f.getX();
}

int main()
{
    printFoo(5); // we're supplying an int argument

    return 0;
}
```

This will work because 5 will be implicitly converted to type of Foo via converting constructor Foo(int)

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

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

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

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

void printEmployee(Employee e) // has an Employee parameter
{
    std::cout << e.getName();
}

int main()
{
    printEmployee("Joe"); // we're supplying an string literal argument

    return 0;
}
```

This will fail because *only one* convertion is allowed and here the c-style literal is first converted to std::string_view and then to class-object. we can fix this by putting sv after “Joe” to make it a string_view literal.

We can put explicit before the constructor function to ensure it will never be used as a converting constructor in cases where we would rather see an error than see the argument get converted. If the function is already declared explicit but we want to pass a integer argument as an object now, we can just pass constructorName{integer} or constructorName(integer) to make the object there and then, passing it as an rvalue.

Return by value and explicit constructors

When we return a value from a function, if that value does not match the return type of the function, an implicit conversion will occur. Just like with pass by value, such conversions cannot use explicit constructors.

```
class Foo
{
public:
    explicit Foo() // note: explicit (just for sake of example)
    {
    }

    explicit Foo(int x) // note: explicit
    {
    }
};

Foo getFoo()
{
    // explicit Foo() cases
    return Foo{ };   // ok
    return { };      // error: can't implicitly convert initializer list to Foo

    // explicit Foo(int) cases
    return 5;        // error: can't implicitly convert int to Foo
    return Foo{ 5 }; // ok
    return { 5 };    // error: can't implicitly convert initializer list to Foo
}
```

Best practices for use of `explicit`

The modern best practice is to make any constructor that will accept a single argument `explicit` by default. This includes constructors with multiple parameters where most or all of them have default values. This will disallow the compiler from using that constructor for implicit conversions. If an implicit conversion is required, only non-explicit constructors will be considered. If no non-explicit constructor can be found to perform the conversion, the compiler will error.
If such a conversion is actually desired in a particular case, it is trivial to convert the implicit conversion into an explicit definition using direct list initialization.

The following **should not** be made explicit:

- Copy (and move) constructors (as these do not perform conversions).

The following **are typically not** made explicit:

- Default constructors with no parameters (as these are only used to convert `{}` to a default object, not something we typically need to restrict).
- Constructors that only accept multiple arguments (as these are typically not a candidate for conversions anyway)

However, if you prefer, the above can be marked as explicit to prevent implicit conversions with empty and multiple-argument lists.

The following **should usually** be made explicit:

- Constructors that take a single argument.

There are some occasions when it does make sense to make a single-argument constructor non-explicit. This can be useful when all of the following are true:

- The constructed object is semantically equivalent to the argument value.
- The conversion is performant.

constexpr with classes

```
#include <iostream>

struct Pair // Pair is an aggregate
{
    int m_x {};
    int m_y {};

    constexpr int greater() const
    {
        return (m_x > m_y  ? m_x : m_y);
    }
};

int main()
{
    constexpr Pair p { 5, 6 };       // now constexpr
    std::cout << p.greater();        // p.greater() evaluates at runtime or compile-time

    constexpr int g { p.greater() }; // p.greater() must evaluate at compile-time
    std::cout << g;

    return 0;
}
```

For int g to initialise, all of g, p and greater() need to be constexpr to not raise an error.

even constructors need to be constexpr in cases where the object will be used in constexpr functions

```
#include <iostream>

class Pair
{
private:
    int m_x {};
    int m_y {};

public:
    constexpr Pair(int x, int y): m_x { x }, m_y { y } {} // now constexpr

    constexpr int greater() const
    {
        return (m_x > m_y  ? m_x : m_y);
    }
};

int main()
{
    constexpr Pair p { 5, 6 };
    std::cout << p.greater();

    constexpr int g { p.greater() };
    std::cout << g;

    return 0;
}
```

this-> pointer

while writing the function to print a data member of a class inside the class body

```
void print() const { std::cout << m_id; }
```

the public member function has access to all the m_ids of all the objects being created yet the compiler knows that it is needs to std::cout the m_id of the object being referenced in the function even though it is not mentioned in function code. This is because the compiler implicitly prefixes all data members being referenced with this-> which explicitly would look like

```
void print() const { std::cout << this->m_id; } // explicit use of this
```

```
class Simple
{
private:
    int m_id{};

public:
    Simple(int id)
        : m_id{ id }
    {
    }

    int getID() const { return m_id; }
    void setID(int id) { m_id = id; }

    void print() const { std::cout << this->m_id; } // use `this` pointer to access the implicit object and operator-
> to select member m_id
};
```

In this class, when we call setID function for an object Simple simple{1}; simple.setID(2);
The objectprefix in function call is added as an extra argument to the modified function call

```
Simple::setID(&simple, 2); // note that simple has been changed from an object prefix to a function argument!
```

and the function declaration is changed to

```
static void setID(Simple* const this, int id) { this->m_id = id; }
```

where this is a const pointer(cant be repointed but can change pointed value)

Why this a pointer and not a reference

Since the this pointer always points to the implicit object (and can never be a null pointer unless we've done something to cause undefined behavior), you may be wondering why this is a pointer instead of a reference. The answer is simple: when this was added to C++, references didn't exist yet.

If this were added to the C++ language today, it would undoubtedly be a reference instead of a pointer. In other more modern C++-like languages, such as Java and C#, this is implemented as a reference.

use cases of this-> pointer

1) it can be useful when the function parameter has the same name as class data member so when used in the same statement of function body code; we can differentiate them

```
struct Something
{
    int data{}; // not using m_ prefix because this is a struct

    void setData(int data)
    {
        this->data = data; // this->data is the member, data is the local parameter
    }
};
```

2) if we want the object being operated on, returned in the member function for techniques like function chaining where multiple functions can be used in the same line, we can use the this-> pointer to return the object.

```
class Calc
{
private:
    int m_value{};

public:
    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() const { return m_value; }
};

#include <iostream>

int main()
{
    Calc calc{};
    calc.add(5).sub(3).mult(4); // method chaining

    std::cout << calc.getValue() << '\n';

    return 0;
}
```

3) we can use this-> pointer to reset the object to its initial state.

```
#include <iostream>

class Calc
{
private:
    int m_value{};

public:
    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() const { return m_value; }

    void reset() { *this = {}; }
};


int main()
{
    Calc calc{};
    calc.add(5).sub(3).mult(4);

    std::cout << calc.getValue() << '\n'; // prints 8

    calc.reset();

    std::cout << calc.getValue() << '\n'; // prints 0

    return 0;
}
```

classes and header files

you can include the class definition in header files for easy use across all files;
it is still advised to not define member functions in the class headerfile because if we had to update one of the member functions, it would cause a ripple effect where every file including the headerfile would need to be recompiled which in a large project could cost hours.

It is wise to include member functions in the headerfile, only for cases where its unlikely for the class to change a lot or it's a trivial class that will be included in few files only.

class definitions and member functions defined inside the class are exempt from ODR (One Definition Rule) so even If they are included multiple times in a cpp file, it would not cause a error. member function outside the class body will not be exempt and we would need to use *inline* to exempt them.

Date.h:

```h
#ifndef DATE_H
#define DATE_H

#include <iostream>

class Date
{
private:
    int m_year{};
    int m_month{};
    int m_day{};

public:
    Date(int year, int month, int day);

    void print() const;

    int getYear() const { return m_year; }
    int getMonth() const { return m_month; }
    int getDay() const { return m_day; }
};

inline Date::Date(int year, int month, int day) // now inline
    : m_year{ year }
    , m_month{ month }
    , m_day{ day }
{
}

inline void Date::print() const // now inline
{
    std::cout << "Date(" << m_year << ", " << m_month << ", " << m_day << ")\n";
};

#endif
```

Libraries

Throughout your programs, you've used classes that are part of the standard library, such as `std::string`. To use these classes, you simply need to #include the relevant header (such as `#include <string>`). Note that you haven't needed to add any code files (such as `string.cpp` or `iostream.cpp`) into your projects. The header files provide the declarations that the compiler requires in order to validate that the programs you're writing are syntactically correct. However, the implementations for the classes that belong to the C++ standard library are contained in a precompiled file that is linked in automatically at the link stage. You never see the code.

Many open source software packages provide both .h and .cpp files for you to compile into your program. However, most commercial libraries provide only .h files and a precompiled library file. There are several reasons for this: 1) It's faster to link a precompiled library than to recompile it every time you need it, 2) A single copy of a precompiled library can be shared by many applications, whereas compiled code gets compiled into every executable that uses it (inflating file sizes), and 3) Intellectual property reasons (you don't want people stealing your code). While you probably won't be creating and distributing your own libraries for a while, separating your classes into header files and source files is not only good form, it also makes creating your own custom libraries easier.

nested class

it is fairly uncommon & complicated to use and does not have outer class' this pointer.

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

class Employee
{
public:
    using IDType = int;

    class Printer
    {
    public:
        void print(const Employee& e) const
        {
            // Printer can't access Employee's `this` pointer
            // so we can't print m_name and m_id directly
            // Instead, we have to pass in an Employee object to use
            // Because Printer is a member of Employee,
            // we can access private members e.m_name and e.m_id directly
            std::cout << e.m_name << " has id: " << e.m_id << '\n';
        }
    };

private:
    std::string m_name{};
    IDType m_id{};
    double m_wage{};

public:
    Employee(std::string_view name, IDType id, double wage)
        : m_name{ name }
        , m_id{ id }
        , m_wage{ wage }
    {
    }

    // removed the access functions in this example (since they aren't used)
};

int main()
{
    const Employee john{ "John", 1, 45000 };
    const Employee::Printer p{}; // instantiate an object of the inner class
    p.print(john);

    return 0;
}
```

```
John has id: 1
```

forward declaration is possible inside outer class

```
#include <iostream>

class outer
{
public:
    class inner1;   // okay: forward declaration inside the enclosing class okay
    class inner1{}; // okay: definition of forward declared type inside the enclosing class
    class inner2;   // okay: forward declaration inside the enclosing class okay
};

class inner2 // okay: definition of forward declared type outside the enclosing class
{
};

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

but not before the outer class is declared

```
#include <iostream>

class outer;         // okay: can forward declare non-nested type
class outer::inner1; // error: can't forward declare nested type prior to outer class definition

class outer
{
public:
    class inner1{}; // note: nested type declared here
};

class outer::inner1; // okay (but redundant) since nested type has already been declared as part of outer class
definition

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

destructors

special member functions that are called automatically when a non-aggregate class type object is destroyed (constructors are called automatically when a non-aggregate class type object is created)

Like constructors, destructors have specific properties

1. The destructor must have the same name as the class, preceded by a tilde (~).
2. The destructor can not take arguments.
3. The destructor has no return type.
4. A class can only have a single destructor.
5. If no user-declared destructor is present; the compiler will implicitly generate a destructor with an empty body which works fine in most cases unless you want some action to be taken when a object is destructed
6. If a std::exit() function is called, the program terminates and no object is destructed which can be a problem in cases where you need the necessary clean-up to ensure memory resources are not held unnecessarily.

```
#include <iostream>

class Simple
{
private:
    int m_id {};

public:
    Simple(int id)
        : m_id { id }
    {
        std::cout << "Constructing Simple " << m_id << '\n';
    }

    ~Simple() // here's our destructor
    {
        std::cout << "Destructing Simple " << m_id << '\n';
    }

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

int main()
{
    // Allocate a Simple
    Simple simple1{ 1 };
    {
        Simple simple2{ 2 };
    } // simple2 dies here

    return 0;
} // simple1 dies here
```

class template usage syntax

```
#include <ios>       // for std::boolalpha
#include <iostream>

template <typename T>
class Pair
{
private:
    T m_first{};
    T m_second{};

public:
    // When we define a member function inside the class definition,
    // the template parameter declaration belonging to the class applies
    Pair(const T& first, const T& second)
        : m_first{ first }
        , m_second{ second }
    {
    }

    bool isEqual(const Pair<T>& pair);
};

// When we define a member function outside the class definition,
// we need to resupply a template parameter declaration
template <typename T>
bool Pair<T>::isEqual(const Pair<T>& pair)
{
    return m_first == pair.m_first && m_second == pair.m_second;
}

int main()
{
    Pair p1{ 5, 6 }; // uses CTAD to infer type Pair<int>
    std::cout << std::boolalpha << "isEqual(5, 6): " << p1.isEqual( Pair{5, 6} ) << '\n';
    std::cout << std::boolalpha << "isEqual(5, 7): " << p1.isEqual( Pair{5, 7} ) << '\n';

    return 0;
}
```

Since the bool Pair<T> member function definition is separate from the class template definition, we need to resupply a template parameter declaration (`template <typename T>`) so the compiler knows what `T` is. Also, when we define a member function outside of the class, we need to qualify the member function name with the fully templated name of the class template (`Pair<T>::isEqual`, not `Pair::isEqual`).

The name of a constructor must match the name of the class but in our class template for `Pair<T>` above, we named our constructor `Pair`, not `Pair<T>`. Somehow this still works, even though the names don't match. Within the scope of a class, the unqualified name of the class is called an **injected class name**. In a class template, the injected class name serves as shorthand for the fully templated name because `Pair` is the injected class name of `Pair<T>`, within the scope of our `Pair<T>` class template, any use of `Pair` will be treated as if we had written `Pair<T>` instead. Therefore, although we named the constructor `Pair`, the compiler treats it as if we had written `Pair<T>` instead. The names now match!

This means that we can also define our `isEqual()` member function like this:

```
template <typename T>
bool Pair<T>::isEqual(const Pair& pair) // note the parameter has type Pair, not Pair<T>
{
    return m_first == pair.m_first && m_second == pair.m_second;
}
```

With member functions for class templates, the compiler needs to see both the class definition (to ensure that the member function template is declared as part of the class) and the template member function definition (to know how to instantiate the template). Therefore, we typically want to define both a class and its member function templates in the same location.

static member variables

```
struct Something
{
    static int s_value; // declare s_value as static (initializer moved below)
};

int Something::s_value{ 1 }; // define and initialize s_value to 1 (we'll discuss this section below)

int main()
{
    Something first{};
    Something second{};

    first.s_value = 2;

    std::cout << first.s_value << '\n';
    std::cout << second.s_value << '\n';
    return 0;
}
```

This program produces the following output:

```
2
2
```

static member variables are shared by all objects and retain their value throughout the program; they exist even if no objects have been instantiated which is why their preffered method of access is via the class name not the object.

```
class Something
{
public:
    static int s_value; // declare s_value as static
};

int Something::s_value{ 1 }; // define and initialize s_value to 1 (we'll discuss this section below)

int main()
{
    // note: we're not instantiating any objects of type Something

    Something::s_value = 2;
    std::cout << Something::s_value << '\n';
    return 0;
}
```

the definition of value of static member variables can be done anywhere even If the static data member itself is private or protected as definition isn't considered a form of access.

```
#include <iostream>

class Something
{
private:
    static inline int s_idGenerator { 1 };
    int m_id {};

public:
    // grab the next value from the id generator
    Something() : m_id { s_idGenerator++ }
    {
    }

    int getID() const { return m_id; }
};
```

another classic static ID example for class objects

static member getter functions are used for dealing with private static data members that cannot be accessed outside the class blockspace & they don't have any this-> pointers because they are not associated with an object but the whole class & static constructors are not allowed in c++.

friendship is magic

sometimes we may not want people to write code in our public regions in classes to avoid misuse but without public access; they can not access private data members.

friend declaration of another class or function gives selective access to friendly members.

friend non-member function

```
#include <iostream>

class Accumulator
{
private:
    int m_value { 0 };

public:
    void add(int value) { m_value += value; }

    // Here is the friend declaration that makes non-member function void print(const Accumulator& accumulator) a
friend of Accumulator
    friend void print(const Accumulator& accumulator);
};

void print(const Accumulator& accumulator)
{
    // Because print() is a friend of Accumulator
    // it can access the private members of Accumulator
    std::cout << accumulator.m_value;
}

int main()
{
    Accumulator acc{};
    acc.add(5); // add 5 to the accumulator

    print(acc); // call the print() non-member function

    return 0;
}
```

even if we put the function inside the class after friend; it would still be a non member

```
// Friend functions defined inside a class are non-member functions
friend void print(const Accumulator& accumulator)
{
    // Because print() is a friend of Accumulator
    // it can access the private members of Accumulator
    std::cout << accumulator.m_value;
}
```

friend functions are useful when a function needs to access data members of more than one class by being friends of many classes while not having to be in the class blockscope of any one particular class. the later class would need a forward declaration before the initial class so that both classes are available when the function is parsing through the initial class.

friendly class

class is forwardly declared by class className;

friendship is not reciprocal or transitive(a->b doesn't mean b->a **&** a->b b->c doesn't mean a->c)

instead of making whole class your friend, you can declare another class' function as your friend by friend + whole class declaration in compiled form with classname::fnname(arg);

the friending class needs to be after the class with function being friended (forward declaration would not work in this case as the whole class needs to be seen in this case)

and the forward declaration of friending class should be above the class with function being friended.