Compound data types

C++ supports a second set of data types called compound data types. **Compound data types** (also sometimes called **composite data types**) are data types that can be constructed from fundamental data types (or other compound data types). Each compound data type has its own unique properties as well.
C++ supports the following compound types:

- Functions
- Arrays
- Pointer types:
  - Pointer to object
  - Pointer to function
- Pointer to member types:
  - Pointer to data member
  - Pointer to member function
- Reference types:
  - L-value references
  - R-value references
- Enumerated types:
  - Unscoped enumerations
  - Scoped enumerations
- Class types:
  - Structs
  - Classes
  - Unions

lvalues and rvalues

An **lvalue** (short for “left value” or “locator value”, and sometimes written as “l-value”) is an expression that evaluates to an identifiable object or function (or bit-field). Entities with identities can be accessed via an identifier, reference, or pointer, and typically have a lifetime longer than a single expression or statement.

```
int main()
{
    int x { 5 };
    int y { x }; // x is an lvalue expression

    return 0;
}
```

a **modifiable lvalue** is an lvalue whose value can be modified. A **non-modifiable lvalue** is an lvalue whose value can’t be modified (because the lvalue is const or constexpr).

```
int x{};
const double d{};

int y { x }; // x is a modifiable lvalue expression
const double e { d }; // d is a non-modifiable lvalue expression
```

An **rvalue** (pronounced “arr-value”, short for “right value”, and sometimes written as r-value) is an expression that is not an lvalue. Rvalue expressions evaluate to a value.

Commonly seen rvalues include literals (except C-style string literals, which are lvalues) and the return value of functions and operators that return by value. Rvalues aren't identifiable (meaning they have to be used immediately), and only exist within the scope of the expression in which they are used.

```
int return5()
{
    return 5;
}

int main()
{
    int x{ 5 }; // 5 is an rvalue expression
    const double d{ 1.2 }; // 1.2 is an rvalue expression

    int y { x }; // x is a modifiable lvalue expression
    const double e { d }; // d is a non-modifiable lvalue expression
    int z { return5() }; // return5() is an rvalue expression (since the result is returned by value)

    int w { x + 1 }; // x + 1 is an rvalue expression
    int q { static_cast<int>(d) }; // the result of static casting d to an int is an rvalue expression

    return 0;
}
```

`return5()`, `x + 1`, and `static_cast<int>(d)` are rvalues, because these expressions produce temporary values that are not identifiable objects.

Lvalue to rvalue conversion

```
int x { 5 };
int y { x }; // x is an lvalue expression
```

lvalue expressions will implicitly convert to rvalue expressions in contexts where an rvalue is expected but an lvalue is provided (assignment and variable definitions expects rvalues)

```
int main()
{
    int x { 2 };

    x = x + 1;

    return 0;
}
```

the variable `x` is being used in two different contexts. On the left side of the assignment operator, `x` is an lvalue expression that evaluates to variable `x`. On the right side of the assignment operator, `x + 1` is an rvalue expression that evaluates to the value `3`.

Lvalue reference

In C++, a **reference** is an alias for an existing object. Although references might seem silly, useless, or redundant at first, references are used everywhere in C++ . An **lvalue reference** (commonly just called a `reference` since prior to C++11 there was only one type of reference) acts as an alias for an existing lvalue (such as a variable).

```
int x { 5 };    // x is a normal integer variable
int& ref { x }; // ref is an lvalue reference variable that can now be used as an alias for variable x
```

the usage of ref or x will be identical and have similar results just like an alias for a variable

the ampersand operator (&) means **lvalue reference to** and not address of (like in pointers)

you can print the references with the reference and also assign new values just like alias

```
int& invalidRef;   // error: references must be initialized
```

all references must be initialized.

Once initialized, a reference in C++ cannot be **reseated**, meaning it cannot be changed to reference another object.

Lvalue references must be bound to a *modifiable* lvalue.

```
int x { 5 };
int& ref { x }; // valid: lvalue reference bound to a modifiable lvalue

const int y { 5 };
int& invalidRef { y };  // invalid: can't bind to a non-modifiable lvalue
int& invalidRef2 { 0 }; // invalid: can't bind to an rvalue
```

reference must match the type of the referent

```
int x { 5 };
int& ref { x }; // okay: reference to int is bound to int variable

double y { 6.0 };
int& invalidRef { y }; // invalid; reference to int cannot bind to double variable
double& invalidRef2 { x }; // invalid: reference to double cannot bind to int variable
```

References and referents have independent lifetimes

```
{
    int x { 5 };

    {
        int& ref { x };   // ref is a reference to x
        std::cout << ref << '\n'; // prints value of ref (5)
    } // ref is destroyed here -- x is unaware of this

    std::cout << x << '\n'; // prints value of x (5)

    return 0;
} // x destroyed here
```

Lvalue reference to const

binding of a reference to a unmodifiable value is illegal because it would allow us to modify a const variable through a non-const reference. By using the const keyword when declaring an lvalue reference, we tell an lvalue reference to treat the object it is referencing as const.

```
const int x { 5 };    // x is a non-modifiable lvalue
const int& ref { x }; // okay: ref is a an lvalue reference to a const value

std::cout << ref << '\n'; // okay: we can access the const object
ref = 6;                  // error: we can not modify an object through a const reference
```

we can also bind const references to modifiable values, but then it will treat it like a const and we can't assign new values via the reference. we can still change the original identifier.

we can also bind rvalues to LrtC which will create temporary objects and refer to that object when the reference is called. This also extends the lifetime of temporary object to that of the reference bound to it even though temporary objects usually get destroyed at the end of the expression

```
const int& ref { 5 }; // okay: 5 is an rvalue

std::cout << ref << '\n'; // prints 5
```

Lvalue references to const can even bind to values of a different type, so long as those values can be implicitly converted to the reference type

```
char c { 'a' };
const int& r2 { c };     // temporary int initialized with value 'a', r2 binds to temporary

std::cout << r2 << '\n'; // prints 97 (since r2 is a reference to int)
```

expensive price of function passed by value calls

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

void printValue(std::string y)
{
    std::cout << y << '\n';
} // y is destroyed here

int main()
{
    std::string x { "Hello, world!" }; // x is a std::string

    printValue(x); // x is passed by value (copied) into parameter y (expensive)

    return 0;
}
```

argument x is copied into printValue() parameter y, the argument is a std::string which is a class type that is expensive to copy and this expensive copy is made every time printValue() is called!

pass by reference for memory savings

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

void printValue(std::string& y) // type changed to std::string&
{
    std::cout << y << '\n';
} // y is destroyed here

int main()
{
    std::string x { "Hello, world!" };

    printValue(x); // x is now passed by reference into reference parameter y (inexpensive)

    return 0;
}
```

Binding a reference is always inexpensive, and no copy of `x` needs to be made. Because a reference acts as an alias for the object being referenced, when `printValue()` uses reference `y`, it's accessing the actual argument `x` (rather than a copy of `x`).

passed by value does not affect the function parameter

When an object is passed by value, the function parameter receives a copy of the argument. This means that any changes to the value of the parameter are made to the copy of the argument, not the argument itself

```
void addOne(int y) // y is a copy of x
{
    ++y; // this modifies the copy of x, not the actual object x
}

int main()
{
    int x { 5 };

    std::cout << "value = " << x << '\n';

    addOne(x);

    std::cout << "value = " << x << '\n'; // x has not been modified

    return 0;
}
```

passed by reference affects the argument

when using pass by reference, any changes made to the reference parameter *will* affect the argument:

```
#include <iostream>

void addOne(int& y) // y is bound to the actual object x
{
    ++y; // this modifies the actual object x
}

int main()
{
    int x { 5 };

    std::cout << "value = " << x << '\n';

    addOne(x);

    std::cout << "value = " << x << '\n'; // x has been modified

    return 0;
}
```

non const passed by reference functions can only accept modifiable lvalues and passing a const value through such function will result in a error. this significantly limits the usefulness of pass by reference to non-const, as it means we can not pass const variables or literals

Pass by const lvalue reference

Unlike a reference to non-const (which can only bind to modifiable lvalues), a reference to const can bind to modifiable lvalues, non-modifiable lvalues, and rvalues.

```
void printRef(const int& y) // y is a const reference
{
    std::cout << y << '\n';
}

int main()
{
    int x { 5 };
    printRef(x);   // ok: x is a modifiable lvalue, y binds to x

    const int z { 5 };
    printRef(z);   // ok: z is a non-modifiable lvalue, y binds to z

    printRef(5);   // ok: 5 is rvalue literal, y binds to temporary int object

    return 0;
}
```

only downside is that such functions can't modify the function argument values.

Mixing of function passes

```
#include <string>

void foo(int a, int& b, const std::string& c)
{
}

int main()
{
    int x { 5 };
    const std::string s { "Hello, world!" };

    foo(5, x, s);

    return 0;
}
```

the first argument is passed by value, the second by reference, and the third by const reference.

Type deduction drops references

deduced type for functions that return references must have auto& instead of auto

When to pass by (const) reference

Because class types can be expensive to copy (sometimes significantly so), class types are usually passed by const reference instead of by value to avoid making an expensive copy of the argument. Fundamental types are cheap to copy, so they are typically passed by value.

pointers

Pointers are one of C++'s historical boogeymen, and a place where many aspiring C++ learners have gotten stuck. However, pointers are nothing to be scared of.

Although the memory addresses used by variables aren't exposed to us by default, we do have access to this information. The **address-of operator** (&) returns the memory address of its operand as a pointer datatype (not a literal as typeid(&x).name() gives int*)

```
int x{ 5 };
std::cout << x << '\n';  // print the value of variable x
std::cout << &x << '\n'; // print the memory address of variable x
```

```
5
0027FEA0
```

For objects that use more than one byte of memory, address-of will return the memory address of the first byte used by the object.

Getting the address of a variable isn't very useful by itself. The most useful thing we can do with an address is access the value stored at that address. The **dereference operator** (*) (also occasionally called the **indirection operator**) returns the value at a given memory address as an lvalue:

std::cout << *(&x) << '\n'; // print the value at the memory address, which is 5 for above example of variable x (parentheses not required, but make it easier to read)

A **pointer** is an object that holds a *memory address* (typically of another variable) as its value
int* pointername;   //initialising a pointer to a integer value & asterisk is part of the declaration syntax for pointers, not a use of the dereference operator.
 the type of the pointer has to match the type of the object being pointed to:

Like normal variables, pointers are *not* initialized by default. A pointer that has not been initialized is sometimes called a **wild pointer**. Wild pointers contain a garbage address, and dereferencing a wild pointer will result in undefined behavior.

```
int* ptr;         // an uninitialized pointer (holds a garbage address)
int* ptr2{};      // a null pointer (we'll discuss these in the next lesson)
int* ptr3{ &x }; // a pointer initialized with the address of variable x
```

once we have a pointer holding the address of another object, we can then use the dereference operator (*) to access the value at that address. std::cout << *ptr << '\n';

Initialisation of a pointer with a literal value is illegal

```
int* ptr{ 5 }; // not okay
int* ptr{ 0x0012FF7C }; // not okay, 0x0012FF7C is treated as an integer literal
```

pointer assignment

1. To change what the pointer is pointing at (by assigning the pointer a new address)

```
int x{ 5 };
int* ptr{ &x }; // ptr initialized to point at x
```

```
int y{ 6 };
ptr = &y; // // change ptr to point at y
```

2.To change the value being pointed at (by assigning the dereferenced pointer a new value)

```
int x{ 5 };
int* ptr{ &x }; // initialize ptr with address of variable x

std::cout << x << '\n';    // print x's value
std::cout << *ptr << '\n'; // print the value at the address that ptr is holding (x's address)

*ptr = 6; // The object at the address held by ptr (x) assigned value 6 (note that ptr is dereferenced here)

std::cout << x << '\n';
std::cout << *ptr << '\n'; // print the value at the address that ptr is holding (x's address)
```

```
5
5
6
6
```

size of pointers

The size of a pointer is dependent upon the architecture the executable is compiled for as a 32 bit executable will have all pointers use 32 bit memory addresses regardless of the size of object being pointed to.

null pointers

```
int* ptr {}; // ptr is now a null pointer, and is not holding an address
```

```
int* ptr { nullptr }; // can use nullptr to initialize a pointer to be a null pointer
```

Accidentally dereferencing null and dangling pointers is one of the most common mistakes C++ programmers make, and is probably the most common reason that C++ programs crash in practice.

pointer to const

normal pointer can't point at a const variable as const variables can not be changed

```
const int x{ 5 };
const int* ptr { &x }; // okay: ptr is pointing to a "const int"
```

pointers to const can point to non-const variables as well. such pointers itself are non const but pointing to const values

const pointers

A **const pointer** is a pointer whose address can not be changed after initialization.

```
int x{ 5 };
int y{ 6 };

int* const ptr { &x }; // okay: the const pointer is initialized to the address of x
ptr = &y; // error: once initialized, a const pointer can not be changed.
```

because the *value* being pointed to is non-const, it is possible to change the value being pointed to via dereferencing the const pointer

we can also have a const pointer pointing to const value const int* const ptr1 {&x};

pass by address

```
void printByAddress(const std::string* ptr) // The function parameter is a pointer that holds the address of str
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced pointer
}
```

```
printByAddress(&str);
```

```
printByAddress(ptr);
```

just like pass by reference, pass by address is fast, and avoids making a copy of the argument object.

passing an object by address also gives us the ability to permanently change the value of a certain variable which can obviously be avoided if we use const int* ptr in the function parameter.

pass by address also allows optional arguments

```cpp
#include <iostream>

void printIDNumber(const int *id=nullptr)
{
    if (id)
        std::cout << "Your ID number is " << *id << ".\n";
    else
        std::cout << "Your ID number is not known.\n";
}

int main()
{
    printIDNumber(); // we don't know the user's ID yet

    int userid { 34 };
    printIDNumber(&userid); // we know the user's ID now

    return 0;
}
```

This example prints:

```
Your ID number is not known.
Your ID number is 34.
```

But function overloading is better option to avoid deferencing null pointers

changing what a pointer parameter points at

when we pass an address to a function, that address is copied from the argument into the pointer parameter (which is fine, because copying an address is fast) but this means we can’t change what the pointer is pointing at. we have to pass pointers by reference to achieve this change.

```cpp
#include <iostream>

void nullify(int*& refptr) // refptr is now a reference to a pointer
{
    refptr = nullptr; // Make the function parameter a null pointer
}

int main()
{
    int x{ 5 };
    int* ptr{ &x }; // ptr points to x

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");

    nullify(ptr);

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");
    return 0;
}
```

```
ptr is non-null
ptr is null
```

In modern C++, most things that can be done with pass by address are better accomplished through other methods

“Pass by reference when you can, pass by address when you must”.

void pointer

The **void pointer**, also known as the generic pointer, is a special type of pointer that can be pointed at objects of any data type but it does not know what object type it is pointing to; deferencing or deleting a void pointer is illegal. It is not possible to do pointer arithmetic on a void pointer. This is because pointer arithmetic requires the pointer to know what size object it is pointing to, so it can increment or decrement the pointer appropriately.

Enumerations

An **enumeration** (also called an **enumerated type** or an **enum**) is a compound data type whose values are restricted to a set of named symbolic constants (called **enumerators**). each enumeration needs to be fully defined before we can use it (a forward declaration is not sufficient). Enumerations is like bool datatype except we can have more than two different components as long as we define them before use.

Enumerators are implicitly constexpr.

Unscoped enumerations

Unscoped enumerations are named such because they put their enumerator names into the same scope as the enumeration definition itself which is the global scope.

```
enum Color
{
        red,
        green,
        blue,
};              //don't forget semi colon after user definition for a new datatype

int main()
{
        Color apple {red};
        Color shirt{green};
        Color cup{blue};
        return 0;
}
```

enumerated types can also be used as function return types

```cpp
enum FileReadResult
{
    readResultSuccess,
    readResultErrorFileOpen,
    readResultErrorFileRead,
    readResultErrorFileParse,
};

FileReadResult readFileContents()
{
    if (!openFile())
        return readResultErrorFileOpen;
    if (!readFile())
        return readResultErrorFileRead;
    if (!parseFile())
        return readResultErrorFileParse;

    return readResultSuccess;
}
```

biggest downside to unscoped enumerations is that an object with same name is present in two different enums, they will cause a naming collision error.

we can separate the scope regions for the enums using namespace

```cpp
namespace Color
{
    // The names Color, red, blue, and green are defined inside namespace Color
    enum Color
    {
        red,
        green,
        blue,
    };
}

namespace Feeling
{
    enum Feeling
    {
        happy,
        tired,
        blue, // Feeling::blue doesn't collide with Color::blue
    };
}

int main()
{
    Color::Color paint{ Color::blue };
    Feeling::Feeling me{ Feeling::blue };

    return 0;
}
```

we can also define the enums only in the functions that we will use them in

integral values of enumerators

When we define an enumeration, each enumerator is automatically associated with an integer value based on its position in the enumerator list.

```
enum Color
{
    black,   // 0
    red,     // 1
    blue,    // 2
    green,   // 3
    white,   // 4
    cyan,    // 5
    yellow,  // 6
    magenta, // 7
};
```

we can also self define the integral values and it will start successive naming thereafter

```
enum Animal
{
    cat = -3,     // values can be negative
    dog,          // -2
    pig,          // -1
    horse = 5,
    giraffe = 5, // shares same value as horse
    chicken,      // 6
};
```

two enumerators can also have same value provided which makes them interchangeable

std::couting enumareted type will print the integral value it stores

While the compiler will implicitly convert an unscoped enumeration to an integer, it will *not* implicitly convert an integer to an unscoped enumeration.

```
Pet pet { 2 }; // compile error: integer value 2 won't implicitly convert to a Pet
pet = 3;       // compile error: integer value 3 won't implicitly convert to a Pet
```

this can be overridden using static_cast

```
Pet pet { static_cast<Pet>(2) }; // convert integer 2 to a Pet
pet = static_cast<Pet>(3);       // our pig evolved into a whale!
```

Getting the name of an enumerator

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

enum Color
{
    black,
    red,
    blue,
};

constexpr std::string_view getColorName(Color color)
{
    switch (color)
    {
    case black: return "black";
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

int main()
{
    constexpr Color shirt{ blue };

    std::cout << "Your shirt is " << getColorName(shirt) << '\n';

    return 0;
}
```

value-initializing an enumeration

If an enumeration is zero-initialized (which happens when we use value-initialization), the enumeration will be given value `0` even if there is no enumerator of value 0 in enumeration.

It is always best practice to have the default enumerator at first value (zero) to ensure no semantic consequence in case of value initialising a enumerated variable.

enumeration size

enumerators have values that are stored usually as int but we can specify which integral datatype want to use to store the enumerators especially if we specifically want a smaller datatype storage system.

```
#include <cstdint>  // for std::int8_t
#include <iostream>

// Use an 8-bit integer as the enum underlying type
enum Color : std::int8_t
{
    black,
    red,
    blue,
};

int main()
{
    Color c{ black };
    std::cout << sizeof(c) << '\n'; // prints 1 (byte)

    return 0;
}
```

Because std::int8_t and std::uint8_t are usually type aliases for char types, using either of these types as the enum base will most likely cause the enumerators to print as char values rather than int values.

inputting enumerators

```
int main()
{
    std::cout << "Enter a pet (0=cat, 1=dog, 2=pig, 3=whale): ";

    int input{};
    std::cin >> input; // input an integer

    if (input < 0 || input > 3)
        std::cout << "You entered an invalid pet\n";
    else
    {
        Pet pet{ static_cast<Pet>(input) }; // static_cast our integer to a Pet
        std::cout << "You entered: " << getPetName(pet) << '\n';
    }

    return 0;
}
```

blockscoped enumerations

```cpp
#include <iostream>

int main()
{
    enum Color
    {
        red,
        blue,
    };

    enum Fruit
    {
        banana,
        apple,
    };

    Color color { red };
    Fruit fruit { banana };

    if (color == fruit) // The compiler will compare color and fruit as integers
        std::cout << "color and fruit are equal\n"; // and find they are equal!
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
}
```

since `color` and `fruit` are from different enumerations and are not intended to be comparable. With standard enumerators, there's no easy way to prevent this unless we use scoped enumerations

scoped enumerations

```cpp
#include <iostream>
int main()
{
    enum class Color // "enum class" defines this as a scoped enumeration rather than an unscoped enumeration
    {
        red, // red is considered part of Color's scope region
        blue,
    };

    enum class Fruit
    {
        banana, // banana is considered part of Fruit's scope region
        apple,
    };

    Color color { Color::red }; // note: red is not directly accessible, we have to use Color::red
    Fruit fruit { Fruit::banana }; // note: banana is not directly accessible, we have to use Fruit::banana

    if (color == fruit) // compile error: the compiler doesn't know how to compare different types Color and Fruit
        std::cout << "color and fruit are equal\n";
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
}
```

scoped enumerations offer their own implicit namespacing for enumerators, there's no need to put scoped enumerations inside another scope region (such as a namespace), unless there's some other compelling reason to do so, as it would be redundant

scoped enumerations don't implicitly store the value as integers which avoids semantic errors where we can use these enumerators as integral values when they clearly have no relation to integers that does not mean you can just print enumerators because we would still need to convert them to int which is their only printable option

```
#include <iostream>
#include <utility> // for std::to_underlying() (C++23)

int main()
{
    enum class Color
    {
        red,
        blue,
    };

    Color color { Color::blue };

    std::cout << color << '\n'; // won't work, because there's no implicit conversion to int
    std::cout << static_cast<int>(color) << '\n';   // explicit conversion to int, will print 1
    std::cout << std::to_underlying(color) << '\n'; // convert to underlying type, will print 1 (C++23)

    return 0;
}
```

conversely, you can also input using a `static_cast` integer to a scoped enumerator

```
int input{};
std::cin >> input; // input an integer

Pet pet{ static_cast<Pet>(input) }; // static_cast our integer to a Pet
```

but list initialize a scoped enumeration using an integral value without the static_cast works as well

```
// using enum class Pet from prior example
Pet pet { 1 }; // okay
```

Despite the benefits that scoped enumerations offer, unscoped enumerations are still commonly used in C++ because there are situations where we desire the implicit conversion to int (doing lots of static_casting gets annoying) and we don’t need the extra namespacing.

`using enum` statements (C++20)

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

enum class Color
{
    black,
    red,
    blue,
};

constexpr std::string_view getColor(Color color)
{
    using enum Color; // bring all Color enumerators into current scope (C++20)
    // We can now access the enumerators of Color without using a Color:: prefix

    switch (color)
    {
    case black: return "black"; // note: black instead of Color::black
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

int main()
{
    Color shirt{ Color::blue };

    std::cout << "Your shirt is " << getColor(shirt) << '\n';

    return 0;
}
```

operator overloading to teach input/output enumerators

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

enum Color
{
    black,
    red,
    blue,
};

constexpr std::string_view getColorName(Color color)
{
    switch (color)
    {
    case black: return "black";
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

// Teach operator<< how to print a Color
// std::ostream is the type of std::cout, std::cerr, etc...
// The return type and parameter type are references (to prevent copies from being made)
std::ostream& operator<<(std::ostream& out, Color color)
{
    out << getColorName(color); // print our color's name to whatever output stream out
    return out;                 // operator<< conventionally returns its left operand

    // The above can be condensed to the following single line:
    // return out << getColorName(color)
}

int main()
{
    Color shirt{ blue };
    std::cout << "Your shirt is " << shirt << '\n'; // it works!

    return 0;
}
```

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

enum Pet
{
    cat,   // 0
    dog,   // 1
    pig,   // 2
    whale, // 3
};

constexpr std::string_view getPetName(Pet pet)
{
    switch (pet)
    {
    case cat:   return "cat";
    case dog:   return "dog";
    case pig:   return "pig";
    case whale: return "whale";
    default:    return "???";
    }
}

constexpr std::optional<Pet> getPetFromString(std::string_view sv)
{
    if (sv == "cat")   return cat;
    if (sv == "dog")   return dog;
    if (sv == "pig")   return pig;
    if (sv == "whale") return whale;

    return {};
}
```

```
// pet is an in/out parameter
std::istream& operator>>(std::istream& in, Pet& pet)
{
    std::string s{};
    in >> s; // get input string from user

    std::optional<Pet> match { getPetFromString(s) };
    if (match) // if we found a match
    {
        pet = *match; // set Pet to the matching enumerator
        return in;
    }

    // We didn't find a match, so input must have been invalid
    // so we will set input stream to fail state
    in.setstate(std::ios_base::failbit);

    // On an extraction failure, operator>> zero-initializes fundamental types
    // Uncomment the following line to make this operator do the same thing
    // pet = {};

    return in;
}

int main()
{
    std::cout << "Enter a pet: cat, dog, pig, or whale: ";
    Pet pet{};
    std::cin >> pet;

    if (std::cin) // if we found a match
        std::cout << "You chose: " << getPetName(pet) << '\n';
    else
    {
        std::cin.clear(); // reset the input stream to good
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << "Your pet was not valid\n";
    }

    return 0;
}
```

structs

structs are a good method to store informative variables that are related to each other

```cpp
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe {};
    joe.id = 14;
    joe.age = 32;
    joe.wage = 60000.0;

    Employee frank {};
    frank.id = 15;
    frank.age = 28;
    frank.wage = 45000.0;

    int totalAge { joe.age + frank.age };

    if (joe.wage > frank.wage)
        std::cout << "Joe makes more than Frank\n";
    else if (joe.wage < frank.wage)
        std::cout << "Joe makes less than Frank\n";
    else
        std::cout << "Joe and Frank make the same amount\n";

    // Frank got a promotion
    frank.wage += 5000.0;

    // Today is Joe's birthday
    ++joe.age; // use pre-increment to increment Joe's age by 1

    return 0;
}
```

using pointers to structs for member selection

```cpp
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << (*ptr).id << '\n'; // Not great but works: First dereference ptr, then use member selection

    return 0;
}
```

Aggregate Initialization

```cpp
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee frank = { 1, 32, 60000.0 }; // copy-list initialization using braced list
    Employee joe { 2, 28, 45000.0 };     // list initialization using braced list (preferred)

    return 0;
}
```

order of declaration is used for initialising values

Prefer the list braced initialisation method

Designated initializers (C++20)

We can also designate the members of struct based on a personal order where we don't have to rely on member declaration because if we add new members to a struct, it would mess up all declared values

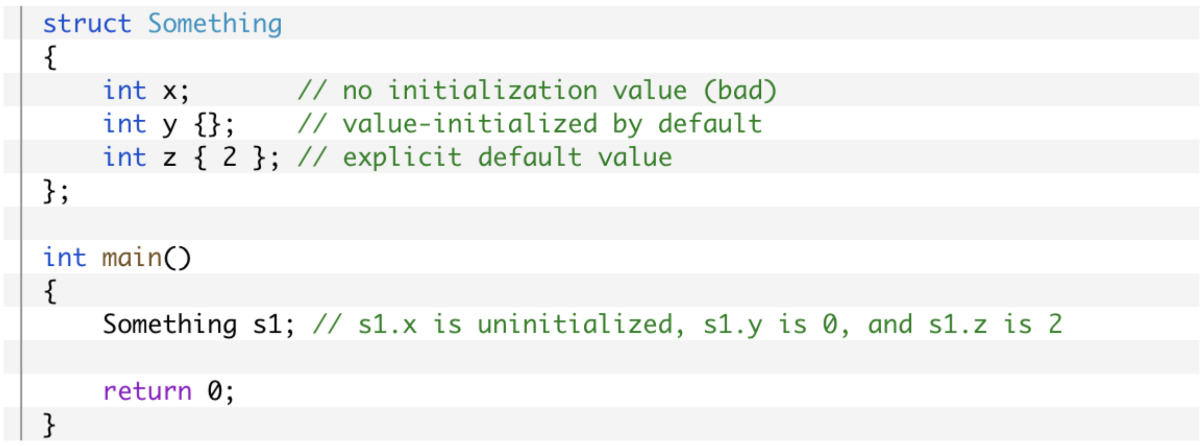
Aggregate Member Reassignment

```cpp
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 1, 32, 60000.0 };
    joe = { joe.id, 33, 66000.0 }; // Joe had a birthday and got a raise

    return 0;
}
```

Initialisation of one struct with another of same type

```cpp
#include <iostream>

struct Foo
{
    int a{};
    int b{};
    int c{};
};

int main()
{
    Foo foo { 1, 2, 3 };

    Foo x = foo; // copy initialization
    Foo y(foo);  // direct initialization
    Foo z {foo}; // list initialization

    std::cout << x.a << ' ' << y.b << ' ' << z.c << '\n';

    return 0;
}
```

Default Initialisation

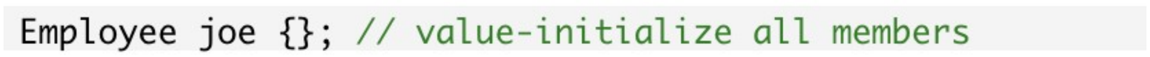


```
struct Something
{
    int x;       // no initialization value (bad)
    int y {};    // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s1; // s1.x is uninitialized, s1.y is 0, and s1.z is 2

    return 0;
}
```

Missing initializers in Aggregate

If an aggregate is initialized but the number of initialization values is fewer than the number of members, then all remaining members are initialized with an empty initializer list which is zero valued (value evaluation)

which is why the following will value initialise all



```
Employee joe {}; // value-initialize all members
```

in case of default initialisers existing, they will be valued over zero value evaluation and remainder of variables will be value initialised

passing structs as references in functions

```cpp
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

void printEmployee(const Employee& employee) // note pass by reference here
{
    std::cout << "ID:   " << employee.id << '\n';
    std::cout << "Age:  " << employee.age << '\n';
    std::cout << "Wage: " << employee.wage << '\n';
}

int main()
{
    Employee joe { 14, 32, 24.15 };
    Employee frank { 15, 28, 18.27 };

    // Print Joe's information
    printEmployee(joe);

    std::cout << '\n';

    // Print Frank's information
    printEmployee(frank);

    return 0;
}
```

we are passing the struct object rather than individual members so we need only one parameter and even if we change the struct, we wont have to change the function code

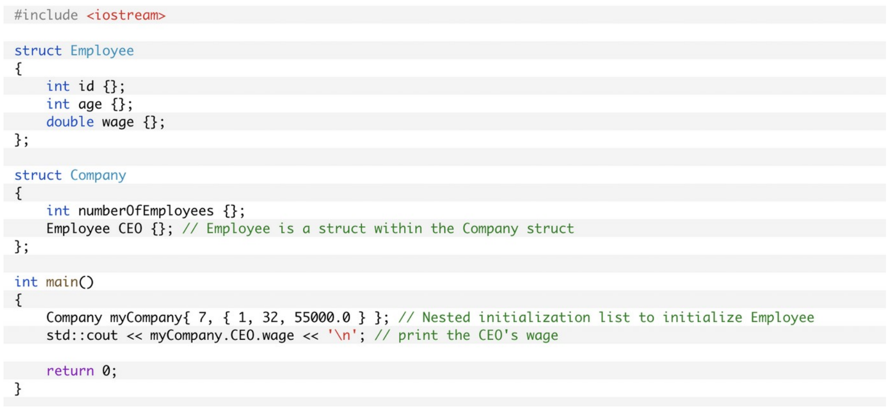
```cpp
int main()
{
    // Print Joe's information
    printEmployee(Employee { 14, 32, 24.15 }); // construct a temporary Employee to pass to function (type explicitly
specified) (preferred)

    std::cout << '\n';

    // Print Frank's information
    printEmployee({ 15, 28, 18.27 }); // construct a temporary Employee to pass to function (type deduced from
parameter)

    return 0;
}
```

we can also pass temporary structs through such functions

```cpp
Point3d getZeroPoint()
{
    return Point3d { 0.0, 0.0, 0.0 }; // return an unnamed Point3d
}
```

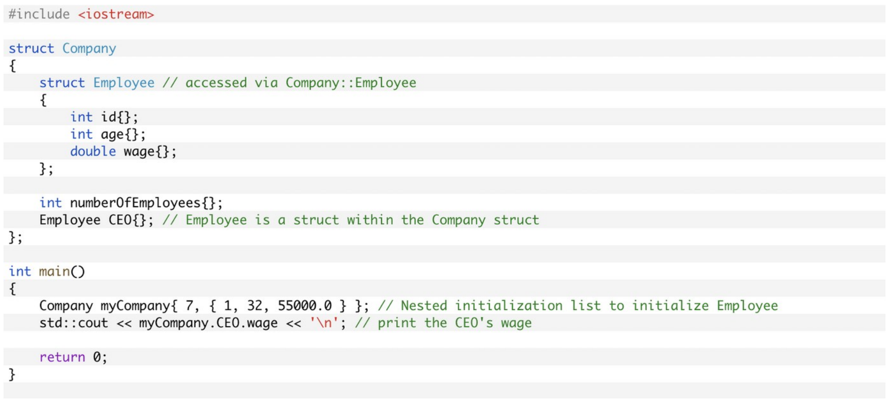
however its preferred to return unnamed temp structs in such cases (return {}; will also return same the zero valued struct

nested structs



```cpp
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

struct Company
{
    int numberOfEmployees {};
    Employee CEO {}; // Employee is a struct within the Company struct
};

int main()
{
    Company myCompany{ 7, { 1, 32, 55000.0 } }; // Nested initialization list to initialize Employee
    std::cout << myCompany.CEO.wage << '\n'; // print the CEO's wage

    return 0;
}
```

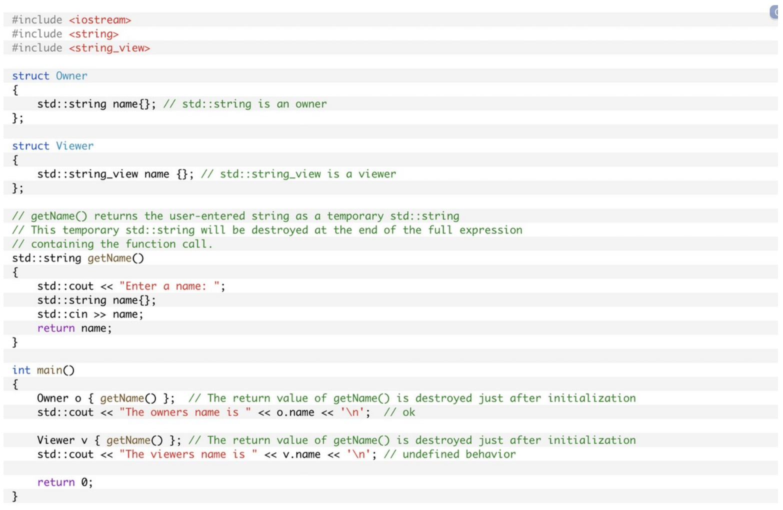
we can declare struct data members of one struct directly in another



```cpp
#include <iostream>

struct Company
{
    struct Employee // accessed via Company::Employee
    {
        int id{};
        int age{};
        double wage{};
    };

    int numberOfEmployees{};
    Employee CEO{}; // Employee is a struct within the Company struct
};

int main()
{
    Company myCompany{ 7, { 1, 32, 55000.0 } }; // Nested initialization list to initialize Employee
    std::cout << myCompany.CEO.wage << '\n'; // print the CEO's wage

    return 0;
}
```

we can also nest the declaration directly

struct data members should be owners (not viewer datatypes)



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

struct Owner
{
    std::string name{}; // std::string is an owner
};

struct Viewer
{
    std::string_view name {}; // std::string_view is a viewer
};

// getName() returns the user-entered string as a temporary std::string
// This temporary std::string will be destroyed at the end of the full expression
// containing the function call.
std::string getName()
{
    std::cout << "Enter a name: ";
    std::string name{};
    std::cin >> name;
    return name;
}

int main()
{
    Owner o { getName() };  // The return value of getName() is destroyed just after initialization
    std::cout << "The owners name is " << o.name << '\n';  // ok

    Viewer v { getName() }; // The return value of getName() is destroyed just after initialization
    std::cout << "The viewers name is " << v.name << '\n'; // undefined behavior

    return 0;
}
```

this ensures that the data members will be valid even after declarations and not create dangling struct errors

using pointers to structs for member selection

```
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << (*ptr).id << '\n'; // Not great but works: First dereference ptr, then use member selection

    return 0;
}
```

However, this is a bit ugly, especially because we need to parenthesize the dereference operation so it will take precedence over the member selection operation. C++ offers a **member selection from pointer operator (->)** (also sometimes called the **arrow operator**) that can be used to select members from a pointer to an object

```
    Employee* ptr{ &joe };
    std::cout << ptr->id << '\n'; // Better: use -> to select member from pointer to object
```

we can also use it twice to navigate through nested struct

```
#include <iostream>

struct Point
{
    double x {};
    double y {};
};

struct Triangle
{
    Point* a {};
    Point* b {};
    Point* c {};
};

int main()
{
    Point a {1,2};
    Point b {3,7};
    Point c {10,2};

    Triangle tr { &a, &b, &c };
    Triangle* ptr {&tr};

    // ptr is a pointer to a Triangle, which contains members that are pointers to a Point
    // To access member y of Point c of the Triangle pointed to by ptr, the following are equivalent:

    // access via operator.
    std::cout << (*(*ptr).c).y << '\n'; // ugly!

    // access via operator->
    std::cout << ptr -> c -> y << '\n'; // much nicer
}
```

Good Example to understand pointers and member selection

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

struct Paw
{
    int claws{};
};

struct Animal
{
    std::string name{};
    Paw paw{};
};

int main()
{
    Animal puma{ "Puma", { 5 } };

    Animal* ptr{ &puma };

    // ptr is a pointer, use ->
    // paw is not a pointer, use .

    std::cout << (ptr->paw).claws << '\n';

    return 0;
}
```

Class template argument deduction

```cpp
std::pair<int, int> p1{ 1, 2 }; // explicitly specify class template std::pair<int, int> (C++11 onward)
std::pair p2{ 1, 2 };           // CTAD used to deduce std::pair<int, int> from the initializers (C++17)
```

we need to provide the datatype parameter for both the member types and not doing so will result in a error

```cpp
std::pair<> p1 { 1, 2 };    // error: too few template arguments, both arguments not deduced
std::pair<int> p2 { 3, 4 }; // error: too few template arguments, second argument not deduced
```

type alias

```cpp
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

template <typename T>
void print(const Pair<T>& p)
{
    std::cout << p.first << ' ' << p.second << '\n';
}

int main()
{
    using Point = Pair<int>; // create normal type alias
    Point p { 1, 2 };        // compiler replaces this with Pair<int>

    print(p);

    return 0;
}
```