Object oriented programming in C++ — basic classes and inheritance

Written by:

- Bernard Teo Zhi Yi

Last updated: 17 June 2022

1 C++ classes overview
  1.1 Inline definitions of member functions
  1.2 The `this` pointer
  1.3 Implementation of member functions
  1.4 `const` member functions
  1.5 Encapsulation, `public` and `private` access modifiers
    1.5.1 `struct` vs `class`
2 Inheritance
  2.1 `protected` access modifier
  2.2 Access modifier of the base class
3 Operator overloading
  3.1 Operator overloading in `std::ostream`
    3.1.1 Writing a custom overload of `operator<<`
  3.2 Overloading `operator==` and `operator<=>`
  3.3 Assignment (`operator=`)
    3.3.1 Slicing
  3.4 Table of all operators
4 Friends
5 Static members
  5.1 Inline static fields
  5.2 Implementation of static member functions
6 Other things you can put in a class
7 Additional topics (self study)

---

1 C++ classes overview

At the end of the previous lecture, we introduced structs, which define a new type containing some fields. That abstraction is known as "composition" since we are combining (composing) a type from a few other types.

Like most programming languages, we can go one step further and give the struct some member functions (aka. methods). In this example below, we add two constructors (functions that create and return an object) and a member function `walk`.

```cpp
// animal.h
#pragma once

struct Animal {
  double age;
  int num_legs;
  int energy;

  Animal();
  Animal(double starting_age, int num_legs = 4);

  bool walk(int distance);
};
```

Snippet 1: `animal.h`

Notice that the member functions above are only declarations, as this is a header file. Similar to before, we put the definitions in a `.cpp` file in order to satisfy the One Definition Rule (as the header file may be included by more than one translation unit):

```cpp
// animal.cpp
#include "animal.h"

#include <iostream>

Animal::Animal() : age(0), num_legs(4), energy(100) {}

Animal::Animal(double starting_age, int num_legs)
    : age(starting_age), num_legs(num_legs), energy(100) {}

bool Animal::walk(int distance) {
  std::cout << "Trying to walk..." << std::endl;
  if (distance <= energy) {
    energy -= distance;
    std::cout << "Walked for " << distance << " metres." << std::endl;
    return true;
  } else {
    std::cout << "Not enough energy!" << std::endl;
    return false;
  }
}
```

Snippet 2: `animal.cpp`

▼ Member initializer list

The line

```cpp
  : age(0), num_legs(4), energy(100)
```

that follows the head of the function definition is called the member initializer list. This specifies how each of the fields of `Animal` should be initialized. In this example, we are setting `age` to `0`, `num_legs` to `4`, and `energy` to `100`.

We finally use this struct in the following way:

```cpp
// main.cpp
#include <iostream>

#include "animal.h"

int main() {
  // Construct an animal using the second constructor
  Animal anim(10.0);

  // Call the `walk()` member function
  bool success = anim.walk(30);

  // Output whether the walk succeeded
  std::cout << (success ? "Success" : "Failure") << std::endl;
}
```

Snippet 3: `main.cpp`

1.1 Inline definitions of member functions

Just like in free functions, member functions can be defined inline by placing the function definitions in the header file and adding the `inline` keyword. However, functions with definitions inside the struct definition are implicitly inline, so the `inline` keyword is optional:

```cpp
struct Animal {
  double age;
  int num_legs;
  int energy;

  // same as `inline Animal() : ...`
  Animal() : age(0), num_legs(4), energy(100) {
  }
  Animal(double starting_age, int num_legs = 4)
      : age(starting_age), num_legs(num_legs), energy(100) {
  }

  // same as `inline bool walk(int distance) { ...`
  bool walk(int distance) {
    std::cout << "Trying to walk..." << std::endl;
    if (distance <= energy) {
      energy -= distance;
      std::cout << "Walked for " << distance << " metres." << std::endl;
      return true;
    } else {
      std::cout << "Not enough energy!" << std::endl;
      return false;
    }
  }
};
```

Snippet 4: Same `Animal` struct but with inline function definitions

1.2 The `this` pointer

The `this` pointer is used when referring to fields or member functions from the current instance of the struct. For example, in the `walk` member function, `energy` is actually a shorthand for `this->energy` (`this` can be implicit most of the time).

We can rewrite `walk` to make `this` explicit:

```cpp
  bool walk(int distance) {
    std::cout << "Trying to walk..." << std::endl;
    if (distance <= this->energy) {
      this->energy -= distance;
      std::cout << "Walked for " << distance << " metres." << std::endl;
      return true;
    } else {
      std::cout << "Not enough energy!" << std::endl;
      return false;
    }
  }
```

Snippet 5: `walk` member function in `Animal`, explicitly qualifying member fields with `this`

▼ When do we need to write `this` explicitly?

Most of the time, `this` does not need to be written explicitly when accessing a member.

The most common situation requiring explicit `this` is when there is a local variable of the same name. Explicit `this` is also required when the base class comes from a template parameter (templates will be covered in a later lecture) and so the compiler cannot figure out if there is a member of the given name.

Note that `this` is a *pointer* to the current instance. This is why we use the `->` operator to access a field from it. Later in this lecture, we will see an example where we use `*this` to refer to the current instance.

1.3 Implementation of member functions

We've talked about functions and calling conventions in the previous lecture, so it is clear how functions work at the assembly level. However, how do member functions work? How does the `walk` function access the energy of the correct `Animal`?

A hint comes from the concept of the `this` pointer.

Although not guaranteed by the C++ standard, every major compiler today implements member functions in a similar way — they add an additional "parameter" to the function call to pass a pointer to the current object (i.e. the `this` pointer).

For example, the member function

```cpp
bool walk(int distance)
```

of `Animal` is equivalent to a free function

```cpp
bool walk(Animal* this, int distance)
```

and the member function call `anim.walk(30)` is equivalent to a normal function call `walk(&anim, 30)`. Note that this is "equivalent" in the sense that the `this` pointer is passed into the function as an additional parameter, however you cannot call a member function using the normal function call syntax or vice versa, and the names of a member function and the equivalent free function are mangled differently so you cannot import declarations using the incorrect syntax. In other words, this is an implementation detail that should not be directly visible to the programmer.

Because of the way member functions are implemented, while technically undefined behaviour, it is possible on almost all platforms to reinterpret a member function pointer as a normal function pointer (with an additional pointer argument at the front).

For example, this works in gcc on x86-64, and is equivalent to doing `anim.walk(30);`:

```cpp
bool (*fptr)(Animal*, int) =
    reinterpret_cast<bool (*)(Animal*, int)>(&Animal::walk);
bool success = fptr(&anim, 30);
```

Snippet 6: Casting member function pointer to normal function pointer

1.4 `const` member functions

If the `this` pointer is really just another parameter passed into the function, it follows that we should be able to make it a `const` parameter (i.e. the difference between `Animal*` and `const Animal*`) if we do not intend to modify the current object. A simple getter is a good candidate for being `const`:

```cpp
  int get_energy() const {
    return energy;
  }
```

Snippet 7: `get_energy` member function in `Animal`, which we mark as `const` because it does not need to modify the current object

We can then call `get_energy` with an `Animal` (or a reference of it) that is `const`:

```cpp
Animal anim(10.0);

// Take a const reference to `anim`
const Animal& anim_ref = anim;

// Call a const member function
int energy = anim_ref.get_energy();
```

Snippet 8: Calling `get_energy` with a `const Animal&`

Try removing the `const` from `get_energy` — it will cause a compile error because you cannot call a non-const member function using a const reference. This shows that if your function does not modify any fields in the current object, you can choose whether or not to make it a `const` function. As a non-`const` object may be used to call a `const` function, marking a function as `const` whenever possible means that the function can be called on as many objects as possible.

▼ Should we always make a member function `const` whenever no fields are modified?

For simple classes, this is usually desired. However, for classes that hold pointers to other objects, the class may semantically contain those objects even though they don't syntactically contain them. Functions that modify those objects can technically be const (since they are only held as pointers), but since the class semantically contains those external objects, these functions should not be const. This is related to value semantics, and we will see some examples of such classes in later lectures.

1.5 Encapsulation, `public` and `private` access modifiers

When we add member functions to structs, we are grouping behaviour (the member functions) with state (the fields). Most of the time, the fields should then not be accessed directly by users of the object. This abstraction of state and behaviour ensures that the object can only be interacted with in a few fixed ways, freeing the user from thinking about the internal implementation of the object. This form of abstraction is known as "encapsulation", and is one of the fundamental abstractions of object-oriented programming (OOP).

To limit the access of certain fields and member functions, C++ has access modifiers `public` and `private` (as well as `protected` and `friend`, which we will explain later), like most other OOP languages. For example, we could make the `energy` field private:

```cpp
struct Animal {
  double age;
  int num_legs;

private:
  int energy;

public:
  Animal();
  Animal(double starting_age, int num_legs = 4);

  bool walk(int distance);

  int get_energy() const;
};
```

Snippet 9: `Animal` struct with a private field

Note that the `private:` makes all fields after it private until the next access modifier is encountered — this is slightly different from Java and C♯ where each field or method needs its own access modifier.

We could also make all the fields private (as is common for encapsulation), as well as add private member functions:

```cpp
struct Animal {
private:
  double age;
  int num_legs;
  int energy;

public:
  Animal();
  Animal(double starting_age, int num_legs = 4);

  bool walk(int distance);

  int get_energy() const;

private:
  do_something_privately();
};
```

Snippet 10: `Animal` struct with many private fields

There is no need to place the private fields or member functions in any particular order — you can simply sprinkle more `public:` and `private:` in the struct definition. (Note that the order of fields is still important, since it affects the struct layout and padding.) Coding conventions may however prefer a certain order (e.g. public member functions go on top, followed by private member functions, and followed lastly by fields (which are almost always all private)).

1.5.1 `struct` vs `class`

In C++, there is a `class` keyword, that can be used almost interchangeably with the `struct` keyword, save for one difference: By default, fields and member functions in a `struct` are public, while those in a `class` are private. However, by most coding conventions, the `struct` keyword is used for *composition* abstractions, while the `class` keyword is used for *encapsulation* abstractions. We will use this convention for the rest of this course.

2 Inheritance

Inheritance allows us to implement a struct that shares some fields and member functions with another class (i.e. a “base” class). In this lecture, we will only talk about inheritance as a way to extend the functionality of a class. We will cover runtime polymorphism (i.e. virtual functions) in a later lecture.

Let’s start with a cleaned-up version of the `Animal` class from the previous section. For brevity, we use inline definitions here, but they can be placed out-of-line if desired.

```
class Animal {
private:
  double age;
  int num_legs;
  int energy;

public:
  Animal() : age(0), num_legs(4), energy(100) {
  }
  Animal(double starting_age, int num_legs = 4)
      : age(starting_age), num_legs(num_legs), energy(100) {
  }

  bool walk(int distance) {
    /* do stuff */
  }

  int get_energy() const {
    return energy;
  }
};
```

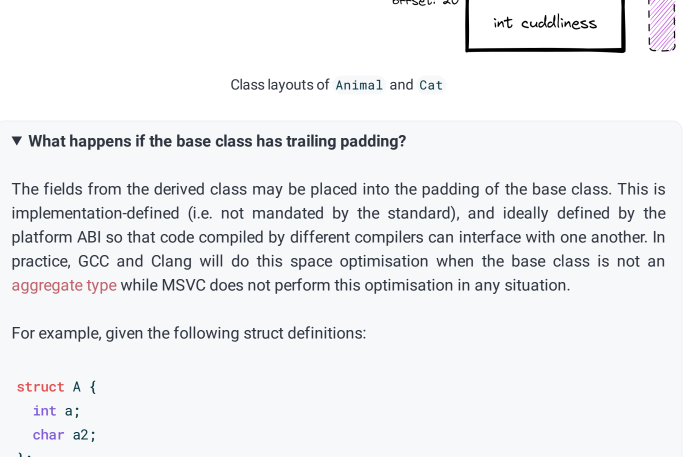
Snippet 11: The base class

We can inherit from this class like this:

```
class Cat : public Animal {
private:
  bool is_sleeping;
  int cuddliness;

public:
  Cat() : Animal(), is_sleeping(false) {
  }
  Cat(double starting_age) : Animal(starting_age), is_sleeping(false) {
  }

  void sleep() {
    is_sleeping = true;
  }

  void wake_up() {
    is_sleeping = false;
    energy = 100;
  }
};
```

Snippet 12: The derived class

This creates a new class with additional fields and member functions. These additional fields are tacked onto the end of the base class (`Animal`) like this (for this and all following examples we assume that `int` is 4 bytes long):



Class layouts of `Animal` and `Cat`

▼ What happens if the base class has trailing padding?

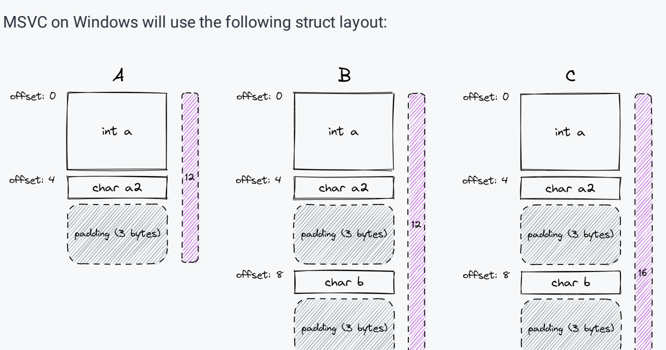
The fields from the derived class may be placed into the padding of the base class. This is implementation-defined (i.e. not mandated by the standard), and ideally defined by the platform ABI so that code compiled by different compilers can interface with one another. In practice, GCC and Clang will do this space optimisation when the base class is not an aggregate type while MSVC does not perform this optimisation in any situation.

For example, given the following struct definitions:

```
struct A {
  int a;
  char a2;
};
struct B : A {
  char b;
};
struct C : B {
  char c;
};
```

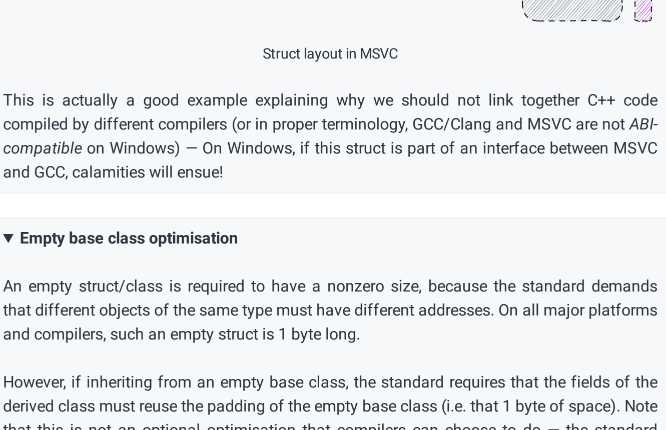
Snippet 13: Code demonstrating reuse of trailing padding

GCC and Clang, on both Unix and Windows, will use the following struct layout:



Struct layout in GCC and Clang

MSVC on Windows will use the following struct layout:



Struct layout in MSVC

This is actually a good example explaining why we should not link together C++ code compiled by different compilers (or in proper terminology, GCC/Clang and MSVC are not *ABI-compatible* on Windows) — On Windows, if this struct is part of an interface between MSVC and GCC, calamities will ensue!

▼ Empty base class optimisation

An empty struct/class is required to have a nonzero size, because the standard demands that different objects of the same type must have different addresses. On all major platforms and compilers, such an empty struct is 1 byte long.

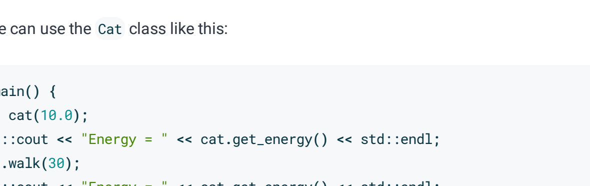
However, if inheriting from an empty base class, the standard requires that the fields of the derived class must reuse the padding of the empty base class (i.e. that 1 byte of space). Note that this is not an optional optimisation that compilers can choose to do — the standard requires it.

For example:

```
struct Empty {};
struct Derived : Empty {
  int x;
};
```

Snippet 14: Deriving from an empty base class

The structs in the code above will be have the following layout:



Struct layout of `Empty` and `Derived`

This feature of C++ is used in all major standard library implementations in allocator-aware containers to ensure that stateless allocators don’t take up any space. (We will be covering containers and allocators in later lectures.)

In C++20, we may alternatively use the `[[no_unique_address]]` attribute to ensure that an empty field does not take up any space.

Then we can use the `Cat` class like this:

```
int main() {
  Cat cat(10.0);
  std::cout << "Energy = " << cat.get_energy() << std::endl;
  cat.walk(30);
  std::cout << "Energy = " << cat.get_energy() << std::endl;
  cat.sleep();
  cat.wake_up();
  std::cout << "Energy = " << cat.get_energy() << std::endl;
}
```

Snippet 15: The main file

Multiple classes can inherit from the same base class, and in this way we can have classes that partially share code. For example, we can have a `Dog` class that has a different way of recharging themselves:

```
class Dog : public Animal {
public:
  Dog() : Animal() {
  }

  void pet() {
    energy += 10;
  }
};
```

Snippet 16: Another derived class

▼ Making a class inherit from multiple base classes

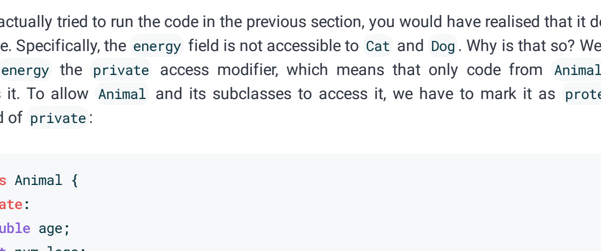
A class can inherit from any number of base classes.

For example:

```
class A {
  int a;
};
class B {
  int b;
  int b2;
};
class C : public A, public B {
  int c;
};
```

Snippet 17: Multiple inheritance

The base classes are laid out in the order they are specified, before any fields declared in `class C`:



Struct layout for multiple inheritance

2.1 `protected` access modifier

If you actually tried to run the code in the previous section, you would have realised that it doesn’t compile. Specifically, the `energy` field is not accessible to `Cat` and `Dog`. Why is that so? We have given `energy` the `private` access modifier, which means that only code from `Animal` can access it. To allow `Animal` and its subclasses to access it, we have to mark it as `protected` instead of `private`:

```
class Animal {
private:
  double age;
  int num_legs;

protected:
  int energy;

  ...
};
```

Snippet 18: `Animal` struct with a protected field

2.2 Access modifier of the base class

Notice that from `main` you can access all the public member functions from both `Cat` and `Animal` — it is as if all the public member functions of `Animal` automatically became public member functions of `Cat`. This is usually what we want, but sometimes we might want to hide the fact that `Cat` inherits from `Animal`. Changing the access modifier of the base class to `private` means that all the fields and member functions of `Animal` become private fields and member functions of `Cat`, which means that outside users of `Cat` will not be able to access them.

```
class Cat : private Animal {
  ...
};
```

Snippet 19: Private inheritance

Apart from `public` and `private`, we can also make the base class `protected`, which does the natural thing, making the fields and member functions of the base class only accessible to the current class and its descendants.

Note that it is possible to omit the access modifier of the base class, in which case the default access modifier will be used — `private` for classes and `public` for structs.

3 Operator overloading

We now look at a different and more mathematical example — a point in two-dimensional space, consisting of an x- and y-coordinate. In contrast to the previous example where we wanted to hide the implementation of the `Animal` class, we want to make the two coordinates public. This abstraction is closer to composition, even though we may want to add some member functions to make it concise and easy to perform some common tasks. We hence use a `struct` here:

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

Snippet 20: The `Point` struct

We can create a point and do some basic operations on its fields (in some function):

```
Point p{1, 2};
p.x += 5;
p.y += 7;
std::cout << p3.x << ',' << p3.y << std::endl;  // prints "6,9"
```

Snippet 21: Directly operating on fields of `Point`

▼ Aggregate initialization

The code `Point p{1, 2}` does aggregate initialization of the newly declared object `p`.

The C++ Reference page on aggregate initialization provides a precise definition of an aggregate. Broadly speaking, an aggregate is a struct that models composition rather than encapsulation — it simply groups together (i.e. aggregates) some fields and doesn't do anything fancy under the hood.

Aggregate initialization initializes each field in the order the fields are declared — the expression `Point{1, 2}` creates a new `Point` object where `x` is set to `1` and `y` is set to `2`.

However, this quickly gets messy when we want to do more complex operations.

We'd like some member functions for common operations on a point. For example, we might want to add two points together and have it return a new point:

```
  Point add_with(Point other) const { return {x + other.x, y + other.y}; }
```

Snippet 22: `add_with` member function inside the `Point` struct

We can then use it like this:

```
Point p1{2, 3};
Point p2{5, 6};
Point p3 = p1.add_with(p2);
std::cout << p3.x << ',' << p3.y << std::endl;  // prints "7,9"
```

Snippet 23: Calling the `add_with` member function

However, the `add_with` operation is fundamentally an addition operation, and so it would be better if we could write `Point p3 = p1.add_with(p2);` more concisely as an addition, i.e. as `Point p3 = p1 + p2;`. This is called operator overloading — there is a default implementation of the `+` operator for primitive types, but we would like to "overload" this operator for our custom type. We can rewrite our `add_with` member function into a member operator:

```
  Point operator+(Point other) const {
    return {x + other.x, y + other.y};
  }
```

Snippet 24: Member `operator+` inside the `Point` struct

This allows us to write:

```
Point p1{2, 3};
Point p2{5, 6};
Point p3 = p1 + p2;
```

Snippet 25: Calling the member `operator+`

Typically, we would also overload the `+=` operator when we overload the `+` operator, since users generally expect both to be available if one is available (note that this member operator is not `const` since the current object is modified):

```
  Point& operator+=(Point other) {
    x += other.x;
    y += other.y;
    return *this;
  }
```

Snippet 26: Member `operator+=` inside the `Point` struct

In many cases the `+=` operator has a similar implementation to the `+` operator, but there may be optimisations one can do for `+=`, for example in strings (we will talk about them in a later lecture).

Note that while conceptually related, these two operators are not semantically related at all — the language treats these two operators as totally different things, and the presence of one of them does not imply anything about the other.

Naturally, one would implement other operators such as `-` and `*` for this struct in a similar fashion. (These are the operators that are defined on a vector space.) Note that there is no requirement that the two operands of a binary operator have the same type — for `*`, we probably want to multiply a `Point` with a `double`:

```
  Point operator*(double scale) const { return {x * scale, y * scale}; }
```

Snippet 27: Member `operator*` inside the `Point` struct

Note that the operators are not by default commutative in C++ — to successfully call the operator above, the left side must be a `Point` and the right side must be a `double`, for example `Point{2, 3} * 5`. If we want to make `5 * Point{2, 3}` work, we have to use a non-member operator (i.e. declared outside the class definition), since the first argument is not a `Point`:

```
Point operator*(double scale, Point original) {
  return {scale * original.x, scale * original.y};
}
```

Snippet 28: Non-member `operator*(double, Point)`

The more common way of defining such an operator is using a friend function inside the class definition:

```
  friend Point operator*(double scale, Point original) {
    return {scale * original.x, scale * original.y};
  }
```

Snippet 29: Friend `operator*(double, Point)` inside the `Point` struct

Note that a friend function does not have an implicit `this` parameter even though it is declared inside the class definition.

▼ Should we use member operators or friend operators when we have a choice?

When the first argument has the same type as the struct, we can choose to use a member operator or a friend operator. Both ways are equivalent, but which is better?

This generally boils down to your coding convention. Operators that modify the left-hand side (e.g. `operator+=`) are almost always written as a member operator, but both styles are common for non-modifying operators. Writing a non-modifying operator as a friend operator gives the function more symmetry between its two arguments, and hence is preferred by some.

We can declare non-member friend functions using the same syntax, but it is less common.

Most other operators can also be overloaded. C++'s input and output (`std::cin` and `std::cout`) use operator overloading too!

3.1 Operator overloading in `std::ostream`

You would have noticed that we use `<<` and `>>` with `std::cin` and `std::cout` respectively, and by now you would have realised that these are operators. These operators work not on any language magic — they are simply operator overloads of `<<` and `>>` defined in the standard library.

For example, we could write this:

```
int a, b;
std::cin >> a >> b;
std::cout << "You typed " << a << " and " << b << std::endl;
```

Snippet 30: Operator overloading of `std::cin` and `std::cout`

How exactly are these operators overloaded?

We'll take a look at `std::cout` here, but `std::cin` is similar and you should be able to work it out on your own.

Since we can chain any number of printable arguments to `std::cout` in a single line, it looks as if we are using some kind of operator that takes any number of arguments. However, that is not the case. C++ operator precedence and associativity roles specify that `operator<<` is evaluated from left to right, which means that

```
std::cout << "You typed " << a << " and " << b << std::endl;
```

is equivalent to

```
((((std::cout << "You typed ") << a) << " and ") << b) << std::endl;
```

We can now see that `operator<<` is overloaded with a left-hand side of `std::cout` and a right-hand side of any printable type. In fact, there are multiple overloads, one for each printable right-hand side type. Since `std::cout` has type `std::ostream`, the operator has a signature `operator<<(std::ostream&, const char*)` (the `std::ostream` is taken by mutable reference since data is being written to that stream).

To make chaining work, this operator must return a reference to the same stream that it was given. Hence, the full function declaration looks something like:

```
std::ostream& operator<<(std::ostream& out, const char* val);
```

3.1.1 Writing a custom overload of `operator<<`

Since `std::cout` works on operator overloading, we can create an overload of `operator<<` for `Point` so that we can directly serialise a `Point`. As the first argument of `operator<<` is `std::ostream&` (not `Point`), we can't have it be a member function of `Point`. Let's make it a free function:

```
  friend std::ostream& operator<<(std::ostream& out, const Point& pt) {
    return out << "{" << pt.x << ", " << pt.y << "}";
  }
```

Snippet 31: Friend `operator<<` inside the `Point` struct

Note that since the return value of each of our calls to `operator<<` is a reference to the same `std::ostream`, the return value here is also a reference to the original `std::ostream`.

```
Point pt{5, 7};
std::cout << pt << std::endl;  // prints "{5, 7}"
```

Snippet 32: Printing a `Point` using `operator<<`

3.2 Overloading `operator==` and `operator<=>`

Since `Point` represents a point in two-dimensional space, testing if two points are equal is a conceptually reasonable operation. Just like the other operators, we can define `operator==`:

```
  friend bool operator==(const Point& a, const Point& b) {
    return a.x == b.x && a.y == b.y;
  }
```

Snippet 33: Friend `operator==` inside the `Point` struct

```
Point pt1{8, 9};
Point pt2{8, 9};
if (pt1 == pt2) {
  std::cout << "Points are equal" << std::endl;
}
```

Snippet 34: Comparing that two points are equal using `operator==`

C++20 brings two improvements to `operator==`, which you should use if possible:

1. If `operator!=` is not declared but `operator==` is declared, then `operator!=` is automatically generated from `operator==` (in the only reasonable manner). Previously, where you had to write a separate definition for `operator!=`, its definition invariably looked something like this:

   ```
     friend bool operator!=(const Point& a, const Point& b) {
       return !(a == b);
     }
   ```

   Snippet 35: Friend `operator==` inside the `Point` struct

2. `operator==` can be defaulted by writing this:

   ```
     friend bool operator==(const point& a, const point& b) = default;
   ```

   Snippet 36: Defaulted friend `operator==` inside the `Point` struct

   The defaulted `operator==` returns true if all base classes (if any) and member fields compare equal using `operator==`, and false otherwise — this is usually the behaviour you want.

There is another new operator introduced in C++20 — `operator<=>` (the *three-way comparison* operator), or more colloquially the *spaceship* operator. This operator does a three-way comparison on its two arguments, returning whether the first argument is *less than*, *equal to*, or *more than* the second argument. This operator is used to automatically generate `operator<`, `operator<=`, `operator>`, and `operator>=`. `operator<=>` may also be defaulted, in which it does a lexicographical comparison of its base classes (if any) and member fields; a defaulted `operator<=>` will automatically generate a defaulted `operator==`.

As points in two-dimensional space do not form a total ordering, we would likely choose not to declare an `operator<=>` (or any of the four operators that are generated from it) for such a general-purpose point struct. However, `operator<=>` actually has provisions for partial orderings, which may be reasonable depending on how you are using the `Point` struct. We will not cover it here, but do look it up if interested.

▼ Why aren't `operator==` and `operator!=` also generated from non-defaulted `operator<=>`?

This is for performance reasons. For some structs, there are quick ways to tell if two objects are not equal, even if deciding whether one is smaller or larger than the other is slow (for example, if figuring out if two strings are equal, we can first check if the two strings have equal length).

3.3 Assignment (`operator=`)

Recall that we can reassign structs just like we do for primitive types. For example, we can do this:

```
Point p1{2, 3};
Point p2{5, 6};
std::cout << p1 << std::endl;  // prints "{2, 3}"
p1 = p2;                       // <-- reasssignment here
std::cout << p1 << std::endl;  // prints "{5, 6}"
```

Snippet 37: Reassignment of `Point`

This is yet another operator, just like all the others we have seen, and it does a member-wise copy of the fields in the struct. This assignment operator however is implicitly generated, even when not declared as default.

Just like the other operators, we can overload it to change its behaviour (covered in a later lecture) — think about when this might be useful!

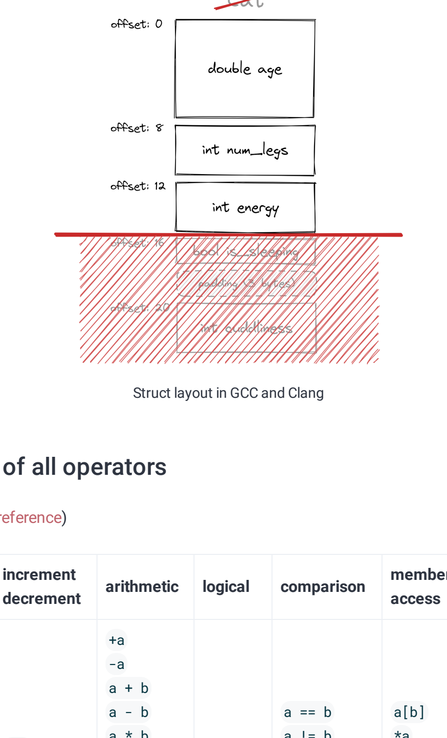
3.3.1 Slicing

Let's go back to the `Animal` and `Cat` example. Consider the following code:

```
Cat cat(10.0);
Animal anim = cat;  // <-- what happens here?
```

Snippet 38: Slicing a cat — the code

The `cat` object is truncated, and only the `Animal` part is copied into `anim`! This may have its uses, but is often not what we want, especially when there are member functions that use dynamic dispatch (covered in a later lecture).



Struct layout in GCC and Clang

3.4 Table of all operators

(taken from cppreference)

<table>
<tr><th>assignment</th><th>increment decrement</th><th>arithmetic</th><th>logical</th><th>comparison</th><th>member access</th><th>other</th></tr>
<tr>
<td>a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &= b
a |= b
a ^= b
a <<= b
a >>= b</td>
<td>++a
--a
a++
a--</td>
<td>+a
-a
a + b
a - b
a * b
a / b
a % b
~a
a & b
a | b
a ^ b
a << b
a >> b</td>
<td>!a
a && b
a || b</td>
<td>a == b
a != b
a < b
a > b
a <= b
a >= b
a <=> b</td>
<td>a[b]
*a
&a
a->b
a.b
a->*b
a.*b</td>
<td>a(...)
a, b
a ? b : c</td>
</tr>
</table>

Apart from the ternary conditional operator (`a ? b : c`) and the `.` and `.*` operators, all other operators can be overloaded.

There are also a number of operators that use words (such as `new` and `delete`), as well as user-defined conversion functions (that enable implicit and explicit conversions from your struct/class to some other type).

4 Friends

We have seen earlier the use of the `friend` keyword to define operators within a class. We did that so that we could place a non-member function inline in the `Point` struct, but there is another, and usually more important, reason why we don't simply place the operator as a free function outside the class.

In C++, a friend of some class `X` is a function that can access all members of `X`, including those that it would have not been able to access due to access modifiers. Friends are usually free functions (i.e. normal functions), but we can also friend a member function of another class, or friend the entire class.

```
class Animal;  // forward declaration of `Animal`

struct A {
  void mess_around(Animal& anim);
};

struct B {
  void mess_around(Animal& anim);
};

class Animal {
  double age;
  int num_legs;
  int energy;

  // inline friend definition
  friend void inline_friend(Animal& anim) {
    anim.energy += 10;
  }

  // friend declaration for an out-of-line function
  friend void outofline_friend(Animal&);

  // friend an entire struct/class
  friend A;

  // friend a member function of another class
  friend void B::mess_around(Animal&);
};

void outofline_friend(Animal& anim) {
  anim.energy += 10;
}

void A::mess_around(Animal& anim) {
  anim.age = 1;
}

void B::mess_around(Animal& anim) {
  anim.age = 1;
}
```

Snippet 39: Friend functions of `Animal`

The use of friend operators is common because most classes have private fields which the operator needs to access.

Note that out-of-line friends, especially friends from other header files, should be used sparingly — it is rare to need to use friends, and having too many friends is likely a symptom of bad software design (subverting the abstraction barrier formed by encapsulation).

5 Static members

Note: Static members of structs/classes are different from static variables at namespace or global scope (covered last lecture) and static local variables in functions!

Classes may also contain static member fields and functions. These are not associated with a particular instance of a class.

For example, we could add these three static members to our `Point` struct:

```
  static double static_field;

  static const Point origin;

  static double shoelace(Point a, Point b, Point c);
```

Snippet 40: Declaration of a static member field and static member function inside the `Point` struct

We could then define them in a separate .cpp file as such:

```
double Point::static_field = 0;

const Point Point::origin{0, 0};

double Point::shoelace(Point a, Point b, Point c) {
  return a.x * b.y + b.x * c.y + c.x * a.y  //
         - a.y * b.x - b.y * c.x - c.y * a.x;
}
```

Snippet 41: Definition of the static member field and static member function of `Point`

Note that defining fields in a separate .cpp file is necessary, otherwise we would flout the One Definition Rule if multiple translation units include the same header file. However, placing the definition of `shoelace` into the struct definition makes it implicitly inline (just like for member functions), and so it would work.

5.1 Inline static fields

In C++17, it is possible to define variables in header files, and have them be merged by the linker if they are defined in multiple translation units, by writing `inline`. For example, we can replace the definition of `static_field` to this:

```
  inline static double static_field = 0;
```

Snippet 42: Inline static definition

However, the `origin` field cannot be declared inline because it is illegal to instantiate a `Point` object before the end of the class definition of `Point` (because `Point` would be an incomplete type).

5.2 Implementation of static member functions

Since static member functions are not associated with an instance of the class, it does not have a `this` pointer. As such, there is no additional parameter (unlike non-static member functions), and on all major platforms and compilers static member functions use the same calling convention as free functions.

6 Other things you can put in a class

- `using` declarations (type aliases) (or `typedefs` in pre-C++11)
- nested structs/classes (note that nested structs/classes in C++ are not associated with a particular instance of the outer class, so they are more like *static* inner classes in Java)

7 Additional topics (self study)

- Delegating constructors (i.e. calling one constructor from another)
- When are fields uninitialized?
  - What happens if we change `Animal() : age(0), num_legs(4), energy(100) {}` to `Animal() {}`?
  - What about `Cat() : Animal(), is_sleeping(false) {}` to `Cat() : is_sleeping(false) {}`?
- User defined conversion functions
- Three way comparisons (`operator<=>`)
- Incomplete types