# Lecture 9: Fantastic Friends and Where to Declare Them

Bart Iver van Blokland

(Rune Sætre)

# PSA 1 of 3 - Next week: Insperaøving 1

- Not mandatory, highly recommended.

- Time and date:
  - Available from Monday 27.02 8:00 – Friday 03.03 17:00
  - You can do the test when it suits you
  - After you open the test you have 2 hours to complete it

- Questions are about the material covered thus far
- More information: «insperaøvinger» page on Blackboard

- No lectures and assignments next week, aside from «hjelp i R1/F1»

# PSA 2 of 3 - Coming up: climbing Mont Blanc

- A voluntary competition available to everyone taking the course
- Objective: solve easy to difficult programming tasks using as little power as possible
  - Code is executed on ARM/Exynos (mobile phone) processors
- We have prizes for the best three, and another one selected at random from everyone who solves the easiest two tasks
- 3rd to the 24th of March (tentative)

# PSA 3 of 3 – Lecture Notes in VS Code

- All lecture notes from the previous lectures are available in VS Code (not categorised into examples, however)
- Use “Create new project” > “Lectures” to find them
- All examples we will go through today are also available
  **NOTE:** opening an example will overwrite main.cpp!

  **Tip:** If you want to take notes, consider creating a .h file. It will not be compiled, but VS Code will still applied syntax highlighting
- You may need to do a «Force refresh of course content» first.

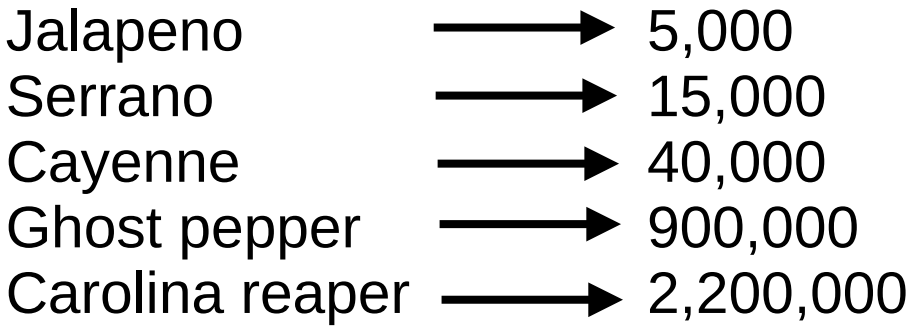
# Last week

- Files and paths
- std::filesystem
- Streams
- std::unordered_map

Example 1 & 2

# std::unordered_map and std::map

- Maps connect a unique “key” value to an associated and not necessarily unique “value”
- Include the `<map>` or `<unordered_map>` header
- The C++ equivalent of a dictionary in Python



# std::unordered_map and std::map

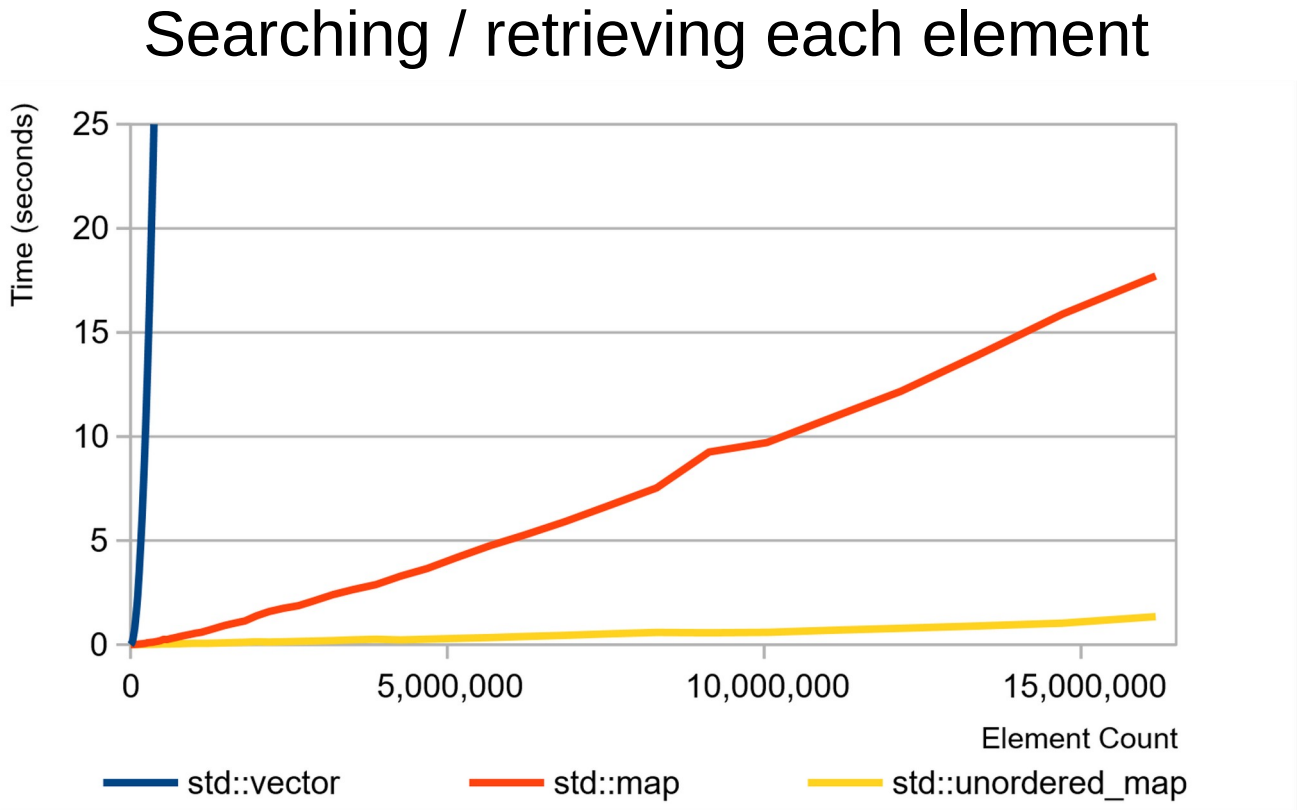
- **Do NOT use the [ ] operator!**

```
std::unordered_map<std::string, double> strengths;
if(strengths["Cayenne"] < 1000) {
    std::cout << "Cayenne is under 1000" << std::endl;
}
```

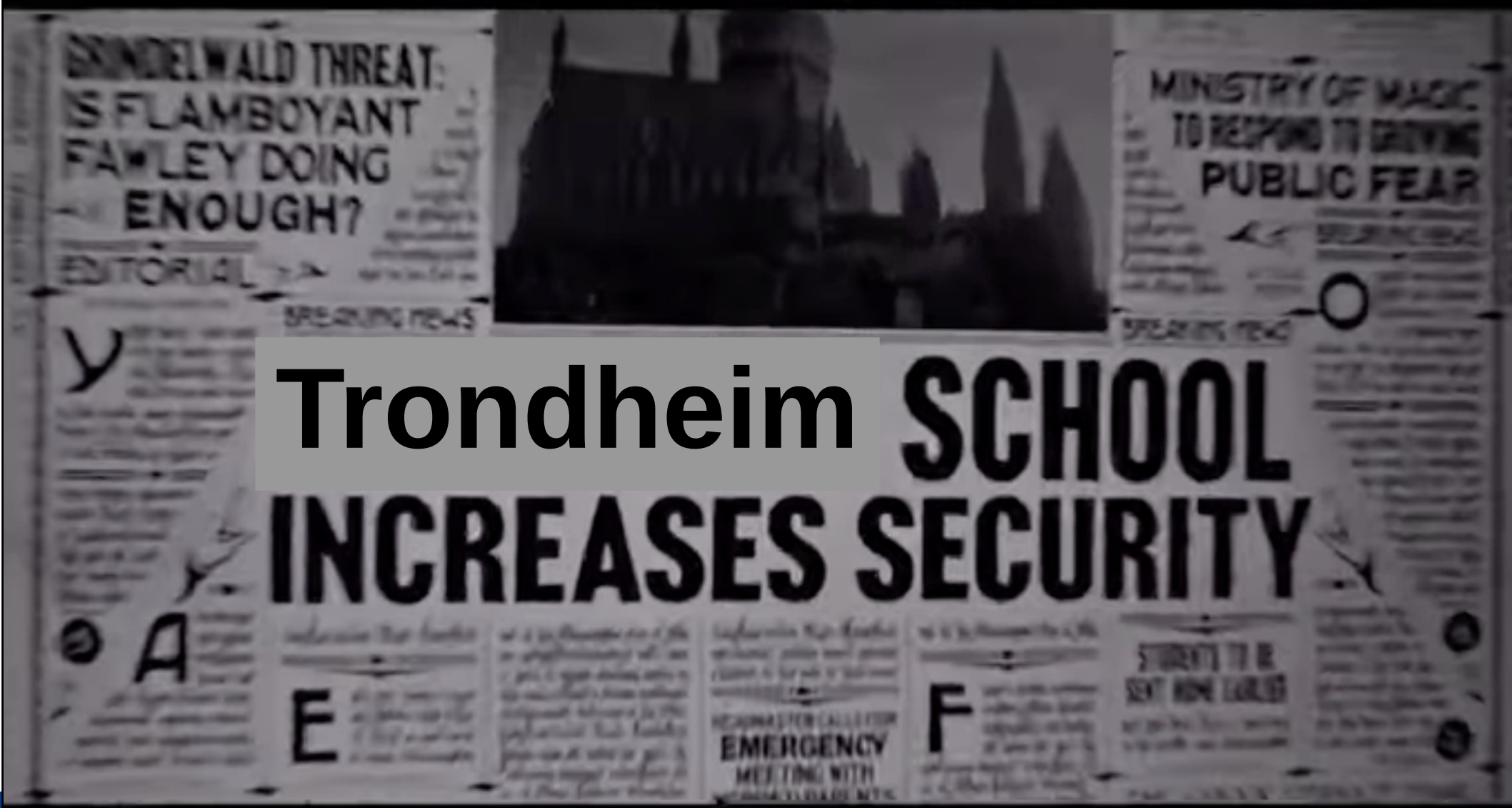
When using the [ ] operator, if the value being requested is not in the map, *it is created implicitly* even when you are not explicitly assigning a value.

Always use insert() and at() instead.

# std::unordered_map is usually faster than std::map



# Black Magic



# We are going to..

- Define functions without writing them
- Call functions that do not exist
- Conjure new C++ language features
- Make friends (hopefully)

# We are going to..

- Define functions without writing them
- Call functions that do not exist
- Conjure new C++ language features
- Make friends (hopefully)

# .. by using:

- Inheritance
- Virtual methods
- Operator overloading
- The friend keyword

# Today

- **Inheritance**
- Virtual methods
- Operator overloading
- Friend keyword

# Inheritance

- Inheritance lets you reuse parts of classes, and change other parts
- A parent class is «copied and pasted» into the child class by the compiler.

```
struct ParentClass {
    void aMethod() {}
    int aField = 10;
};

class ChildClass : public ParentClass {
};

int main() {
    ChildClass child;
    std::cout << child.aField << std::endl;
}
```

The parent class to inherit from is declared after the class name

ChildClass has inherited the aField field from ParentClass.

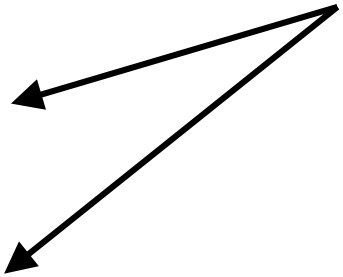
Example 3

# Inheritance: Overriding methods

- Declaring the exact same method in the child class will overwrite the one inherited from the parent class
- Useful for changing portions of the class's behaviour

```
class Lamp {
    bool isLit = false;
    bool isLightOn() {
        return isLit;
    }
};

class BrokenLamp : public Lamp {
    bool isLightOn() {
        return false;
    }
};
```



The overriding method must have the exact same name, parameters, and return type

When creating an instance of `BrokenLamp` and calling `isLightOn()` will run the overridden version!

Example 4

# Inheritance: Member access

- Child classes cannot read private values
- Use protected for fields and methods that child classes should be able to use
- Protected members act the same as private members from outside the object

```
class ParentClass {
    void cannotBeCalledFromChild() {}
protected:
    void canBeCalledFromChild() {}
public:
    void canBeCalledFromAnywhere() {}
};
```

Example 4

# Inheritance: Casting

- A child class can be converted its parent class using `static_cast<>()`
- A parent class can't be converted to a child class

```
class Lamp {
};
class BrokenLamp : public Lamp {
};

int main() {
    BrokenLamp brokenLamp;
    Lamp lamp = static_cast<Lamp>(brokenLamp);
    // Not allowed:
    // BrokenLamp b = static_cast<BrokenLamp>(lamp);
    return 0;
}
```

Example 5

# Inheritance: Constructors

- If the parent class has a (non-default) constructor, you have to call that constructor explicitly
- You must call the parent constructor before initialising the fields of the child class

```
class Lamp {
protected:
    bool isOn = false;
    Lamp(bool startState)
        : isOn{startState} {}
};

class FancyLamp : public Lamp {
    FancyLamp(bool startState)
        : Lamp{startState} {}
};
```

The parent constructor is called here

Example 6

# Today

- Inheritance
- **Virtual methods**
- Operator overloading
- Friend keyword

# Virtual methods

- When calling a virtual method using a reference to an object, the data type of the object being referenced determines which variant of that method will be called.

- Basically:
  - If you give me a reference to a coffee machine, and when I use that coffee machine I get coffee, I don’t need to care how the coffee machine works

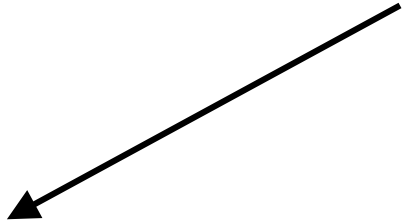
- Purpose:
  - Modular interfaces where parts can be swapped out without having to rewrite your code

# Virtual methods

```
struct Lamp {
    virtual void shine() {}
};

class BlueLamp : public Lamp {
    void shine() {
        std::cout << "[blue light]" << std::endl;
    }
};

int main() {
    BlueLamp blueLamp;
    Lamp& childReference = blueLamp;
    childReference.shine(); // prints [blue light]
    return 0;
}
```



Because shine() is marked as virtual, the BlueLamp version of shine() will be executed, even though the reference is of type Lamp

Example 7

# Abstract classes

- A method is pure virtual if it is marked with virtual and = 0
- An object with at least one pure virtual method is called "abstract", and cannot be instantiated.

```
struct Lamp {
    virtual void shine() = 0;
};

class BlueLamp : public Lamp {
    void shine() override {}
};

int main() {
    BlueLamp blueLamp; // Instances of child classes are allowed
    Lamp& childReference = blueLamp; // References are allowed
    // Lamp lamp; // Not allowed: Lamp is an abstract class
    return 0;
}
```

Example 8

# Virtual methods: usage with std::vector

- Most useful feature of virtual methods
- Allows iterating over a vector containing many different data types which all share some set of methods
  - Example: all moving objects in a video game
- Cannot create a vector of references, so we have to use unique_ptr instead (discussed in a future lecture)
  - Use emplace_back() instead of push_back()

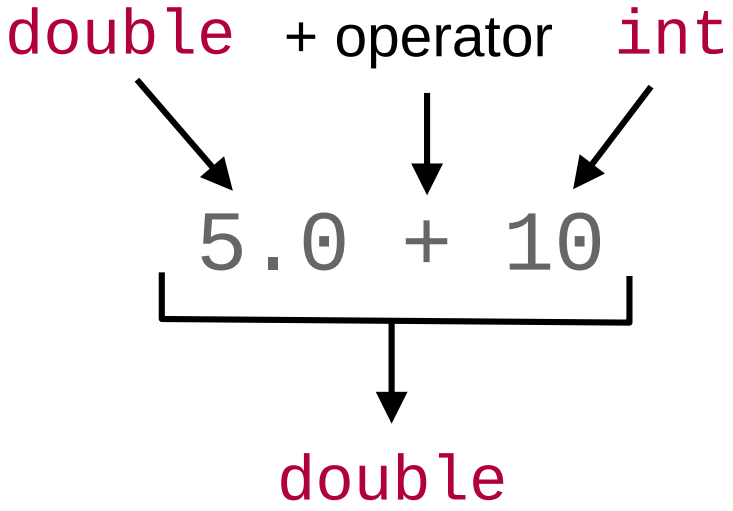
Example 9

# Today

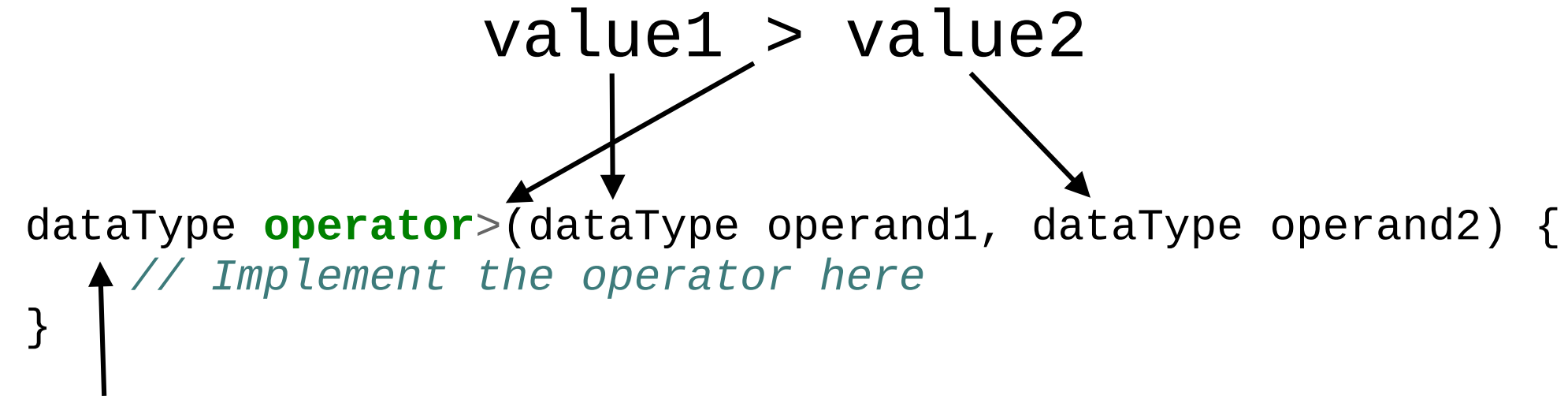
- Inheritance
- Virtual methods
- **Operator overloading**
- Friend keyword

# Operator overloading

You can think of an operator as a function that takes two parameters and returns another value



# Operator overloading: syntax




The data type of the result of the operator
(for > that is usually **bool**)

Example 10

# Operators are functions

- Operators are normal functions, except:
  - The name must be `operator` with the desired operator appended to it (e.g. `operator*` or `operator!=`)
  - They are called by using the operator

```
std::string operator+(std::string text, int number) {
    return text + std::to_string(number);
}

int main() {
    std::string message = "The number is: ";
    std::string combined = message + 5;
    return 0;
}
```

Using the operator calls the operator function!

# Operators: Order matters

- The parameter order matters!
  - The example below will not compile, as the operator is only defined for «int + string», not «string + int»

```
std::string operator+(int number, std::string text) {
    return text + std::to_string(number);
}

int main() {
    std::string message = "The number is: ";
    std::string combined = message + 5;
    return 0;
}
```

# Operators: There are many!

```
type operator+ (type a, type b) {}
type operator- (type a, type b) {}
type operator* (type a, type b) {}
type operator/ (type a, type b) {}
type operator% (type a, type b) {}
type operator^ (type a, type b) {}
type operator& (type a, type b) {}
type operator| (type a, type b) {}
type operator, (type a, type b) {}
type operator>> (type a, type b) {}
type operator<< (type a, type b) {}
type operator~ (type a) {}
type operator! (type a) {}
```

```
type operator== (type a, type b) {}
type operator!= (type a, type b) {}
type operator&& (type a, type b) {}
type operator|| (type a, type b) {}
type operator< (type a, type b) {}
type operator> (type a, type b) {}
type operator<= (type a, type b) {}
type operator>= (type a, type b) {}
```

For all of these, `type` can be replaced with **any** data type!

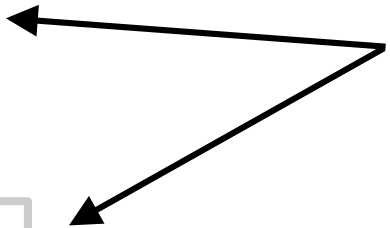
(exception: operators with data types that already exist, like `int + int`)

# Operators: Can be member functions

- When declaring an operand as a member function, the containing class becomes the operator's first parameter

```
struct Point {
    double x = 0;
    double y = 0;

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



These are equivalent!

```
Point operator+(Point a, Point b) {
    Point sum {a.x + b.x, a.y + b.y};
    return sum;
}
```

Example 11

# Operators: There are even more!

```
struct vec3 {
    vec3 operator++ () {}
    vec3 operator-- () {}
    vec3 operator[] (vec3 index) {}
    vec3 operator() (vec3 other) {}
    vec3 operator=  (vec3 other) {}
    vec3 operator+= (vec3 other) {}
    vec3 operator-= (vec3 other) {}
    vec3 operator*= (vec3 other) {}
    vec3 operator/= (vec3 other) {}
    vec3 operator%= (vec3 other) {}
    vec3 operator^= (vec3 other) {}
    vec3 operator&= (vec3 other) {}
    vec3 operator|= (vec3 other) {}
    vec3 operator>>=(vec3 other) {}
    vec3 operator<<=(vec3 other) {}
};
```

.. But these can **only** be declared as a member function!

As before, `vec3` can be replaced with any other data type!

# Operators: Tidbits

- Operator overloading extends the language beyond what it is normally able to do.
- For some operators (such as >> and << for reading and writing to streams) you usually want to take in the parameters as references.
- As an operator is just another function, it can return any data type, including `void` where that makes sense (for example += ).

# Today

- Inheritance
- Virtual methods
- Operator overloading
- **Friend keyword**

# Friend

- Allows an object to give another object or function access to its private members.
- Primary use case: printing classes using `std::cout` or `std::ofstream`
- Friends can only be declared while declaring an object. They cannot be added after the fact.

# Using friend: Functions

- Case 1 of 3: Giving access to a specific function

```
class House {
    bool plantsHaveBeenWatered = false;
    void openFrontDoor();

    friend void useHouseKey(House &house);
};

void useHouseKey(House &house) {
    house.openFrontDoor();
    house.plantsHaveBeenWatered = true;
}
```

Simply copy and paste the function declaration after the friend keyword

The function can now use private fields and methods!

Example 12

# Using friend: Classes

- Case 2 of 3: Giving access to all methods in another class

```
class House {
    bool plantsHaveBeenWatered = false;
    void openFrontDoor();

    friend class Buddy;
};

class Buddy {
public:
    void openHouseDoor(House &house) {
        house.openFrontDoor();
        house.plantsHaveBeenWatered = true;
    }
};
```

Simply copy and paste the class declaration after the friend keyword

All methods in Buddy can now use the private members of House

Example 12

# Using friend: Methods

- Case 3 of 3: Giving access to a specific method in another class

```
class Buddy;

class House {
    bool plantsHaveBeenWatered = false;
    void openFrontDoor();

    friend void Buddy::openHouseDoor(House &house);
};

class Buddy {
public:
    void openHouseDoor(House &house) {
        house.openFrontDoor();
        house.plantsHaveBeenWatered = true;
    }
};
```

You may need to forward declare the class!

Copy the method, and add the class name in front (here: `Buddy::`)

The openHouseDoor() method in Buddy can now access private House members. Other methods can't.

Example 12

# Today

- Inheritance
- Virtual methods
- Operator overloading
- Friend keyword

# Next week:

- Insperaøving 1!
- No regular / assignment lectures
- There will be «hjelp-i-R1/F1»