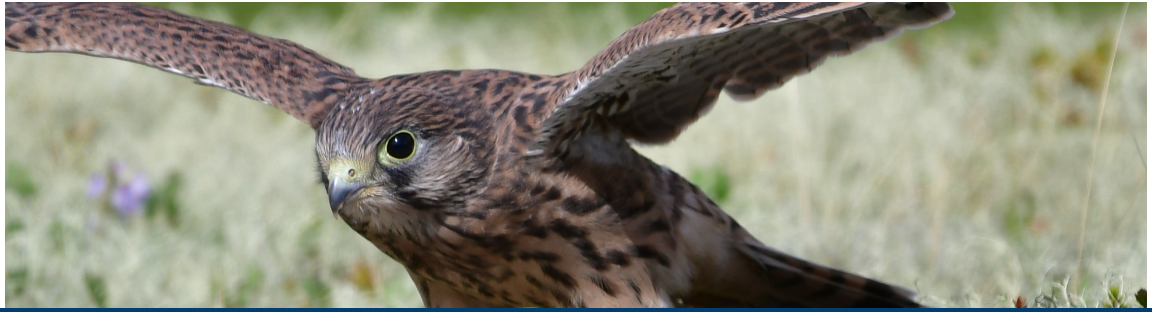


PROGRAMMING IN C++

Jülich Supercomputing Centre

8 – 12 May 2023 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Day 3

```
1  class Example {
2      double x{1.0};
3      auto operator()(unsigned long i) const -> double
4      {
5          auto ans {1.0};
6          while (i--) ans *= x;
7          return ans;
8      }
9  };
10 auto main() -> int
11 {
12     Example one;
13     one(5UL);
14 }
```

Spot the error!

- We are accessing the non-constant member variable `x` in the `const` member function `operator()`
- There are no constructors, so it should not be possible to construct an object of type `Example` in `main()`
- The `operator()` in this class is not accessible in `main()`
- Nothing is wrong, this code is fine.

```
1   class Example {
2       auto inspect(double x) -> bool;
3       auto inspect(const Example& e2) -> bool;
4   };
5   auto main() -> int
6   {
7       Example x, y;
8       double d{7.0};
9
10      x.inspect(d);
11      x.inspect(y);
12  }
```

Based on what you can see above, which variable in `main()` remained unchanged between lines 9 and 12?

1. `x` and `y`
2. Only `y`
3. `y` and `d`
4. Only `d`

Inheritance and class hierarchies

CLASS INHERITANCE

Analogy

- Inherited traits: many properties shared among entities of different related types
- Each branch may add new properties
- Seems like a good fit to different ideas we may want to represent in code

CLASS INHERITANCE

```
1  struct Point {double X, Y;};
2  class Triangle {
3  public:
4    // Constructors etc., and then,
5    void translate();
6    void rotate(double byangle);
7    auto area() const -> double;
8    auto perimeter() const -> double;
9  private:
10   Point vertex[3];
11 };
12 class Quadilateral {
13 public:
14   void translate();
15   void rotate(double byangle);
16   auto area() const -> double;
17   auto perimeter() const -> double;
18   auto is_convex() const -> bool;
19 private:
20   Point vertex[4];
21 };
```

Geometrical figures

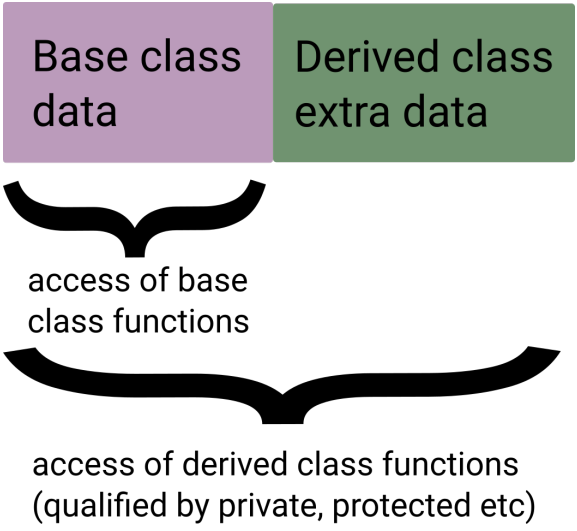
- Many actions (e.g. translate and rotate) will involve identical code
- Properties like `area` and `perimeter` make sense for all, but are better calculated differently for each type
- There may also be new properties (`is_convex`) introduced by a type

INHERITANCE: BASIC SYNTAX

```
1  class SomeBase {
2  public:
3      double f();
4  protected:
5      int i;
6  private:
7      int j;
8  };
9  class Derived : public SomeBase {
10     void haha() {
11     // can access f() and i
12     // can not access j
13     }
14 };
15 void elsewhere()
16 {
17     SomeBase a;
18     Derived b;
19     // Can call a.f(),
20     // but e.g., a.i = 0; is not allowed
21 }
```

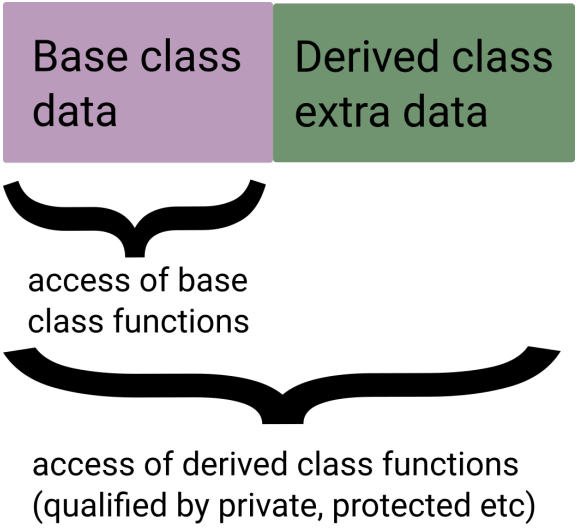
- Class members can be `private`, `protected` or `public`
- `public` members are accessible from everywhere
- `private` members are for internal use in one class
- `protected` members can be seen by derived classes

INHERITANCE



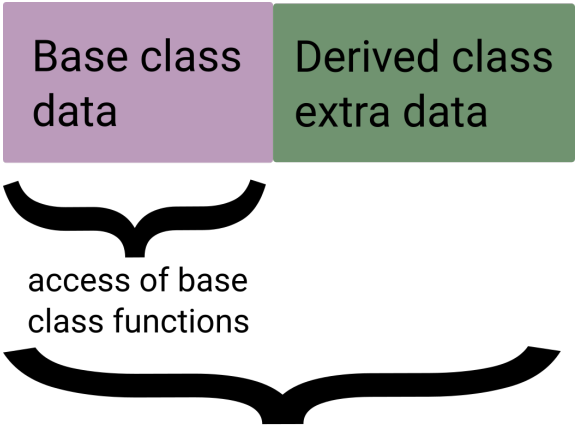
- Inheriting class may add more data, but it retains all the data of the base
- The base class functions, if invoked, will see a base class object
- The derived class object *is a* base class object, but with additional properties

INHERITANCE



- A pointer to a derived class always points to an address which also contains a valid base class object.
- `baseptr=derivedptr` is called "upcasting". Always allowed.
- Implicit downcasting is not allowed. Explicit downcasting is possible with `static_cast` and `dynamic_cast`

INHERITANCE



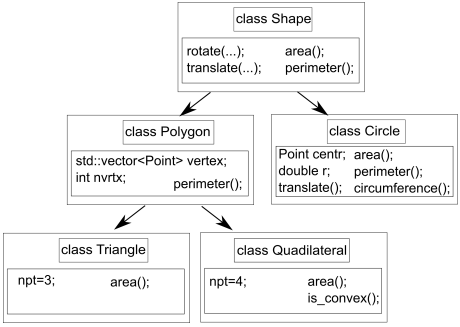

```
1  class Base {
2  public:
3    void f() { std::cout << "Base::f()\n"; }
4  protected:
5    int i{4};
6  };
7  class Derived : public Base {
8    int k{0};
9  public:
10   void g() { std::cout << "Derived::g()\n"; }
11 };
12 int main()
13 {
14   Derived b;
15   Base *ptr = &b;
16   ptr->g(); // Error!
17   static_cast<Derived *>(ptr)->g(); //OK
18 }
```

CLASS INHERITANCE

- We want to write a program to
  - list the area of all the geometric objects
  - select the largest and smallest objects
  - draw

  in our system.
- A loop over a vector of them will be nice. But `vector< ??? >`
- Object oriented languages like C++, Java, Python ... have a concept of "inheritance" for the classes, to describe such conceptual relations between different types.
- 4 ways to solve this problem in C++ will be introduced at various points in this course

INHERITANCE WITH VIRTUAL FUNCTIONS



- Abstract concept class “Shape”
- Inherited classes add/change some properties
- and inherit other properties from “base” class

A triangle *is* a polygon. A polygon *is* a shape. A circle *is* a shape.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```cpp
class Shape {
public:
  virtual ~Shape() = 0;
  virtual void rotate(double) = 0;
  virtual void translate(Point) = 0;
  virtual auto area() const -> double = 0;
  virtual auto perimeter() const -> double = 0;
};
class Circle : public Shape {
public:
  Circle(); // and other constructors
  ~Circle();
  void rotate(double phi) {}
  auto area() const -> double override
  {
    return pi * r * r;
  }
private:
  double r;
};
```

- Circle is a derived class from base class Shape
- A derived class inherits from its base(s), which are indicated in the class declaration.
- Functions marked as `virtual` in the base class *can be re-implemented* in a derived class.

Note: In C++, member functions are not virtual by default.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1  class Shape {
2  public:
3    virtual ~Shape() = 0;
4    virtual void rotate(double) = 0;
5    virtual void translate(Point) = 0;
6    virtual double area() const = 0;
7    virtual auto perimeter() const -> double = 0;
8  };
9  class Circle : public Shape {
10 public:
11   Circle(); // and other constructors
12   ~Circle();
13   void rotate(double phi) {}
14   auto area() const -> double override
15   {
16     return pi * r * r;
17   }
18 private:
19   double r;
20 };
21 Shape a; // Error!
22 Circle b; // ok.
```

- A derived class inherits all member variables and functions from its base.
- `virtual` re-implemented in a derived class are said to be "overriden", and ought to be marked with `override`
- A class with a pure virtual function (with " $= 0$" in the declaration) is an abstract class. Objects of that type can not be declared.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1  class Polygon : public Shape {
2  public:
3    auto perimeter() const -> double final
4    {
5      // return sum over sides
6    }
7  protected:
8    vector<Point> vertex;
9    int npt;
10 };
11 class Triangle : public Polygon {
12 public:
13   Triangle() : npt(3)
14   {
15     vertex.resize(3); // ok
16   }
17   auto area() const -> double override
18   {
19     // return sqrt(s*(s-a)*(s-b)*(s-c))
20   }
21 };
```

Syntax for inheritance

- Triangle implements its own `area()` function, but can not implement a `perimeter()`, as that is declared as **`final`** in `Polygon`. This is done if the implementation from the base class is good enough for intended inheriting classes.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1  class Polygon : public Shape {
2  public:
3    auto perimeter() const -> double final
4    {
5      // return sum over sides
6    }
7  protected:
8    vector<Point> vertex;
9    int npt;
10 };
11 class Triangle : public Polygon {
12 public:
13   Triangle() : npt(3)
14   {
15     vertex.resize(3); // ok
16   }
17   auto area()  -> double override  // Error!!
18   {
19     // return sqrt(s*(s-a)*(s-b)*(s-c))
20   }
21 };
```

- The keyword `override` ensures that the compiler checks there is a corresponding base class function to override.
- Virtual functions can be re-implemented without this keyword, but an accidental omission of a `const` or an `&` can lead to really obscure runtime errors.

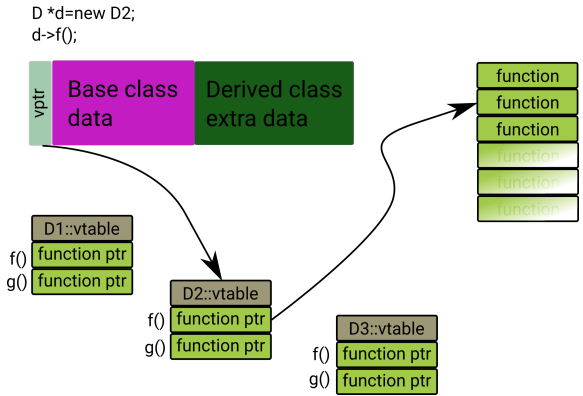
CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1  int main()
2  {
3    vector<std::unique_ptr<Shape>> shape;
4    shape.push_back(std::make_unique<Circle>(0.5, Point(3,7)));
5    shape.push_back(std::make_unique<Triangle>(Point(1,2), Point(3,3), Point(2.5,0)));
6  ...
7    for (size_t i = 0;i < shape.size(); ++i) {
8      std::cout << shape[i]->area() << '\n';
9    }
10 }
```

- A pointer to a base class is allowed to point to an object of a derived class
- Here, `shape[0]->area()` will call `Circle::area()`, `shape[1]->area()` will call `Triangle::area()`

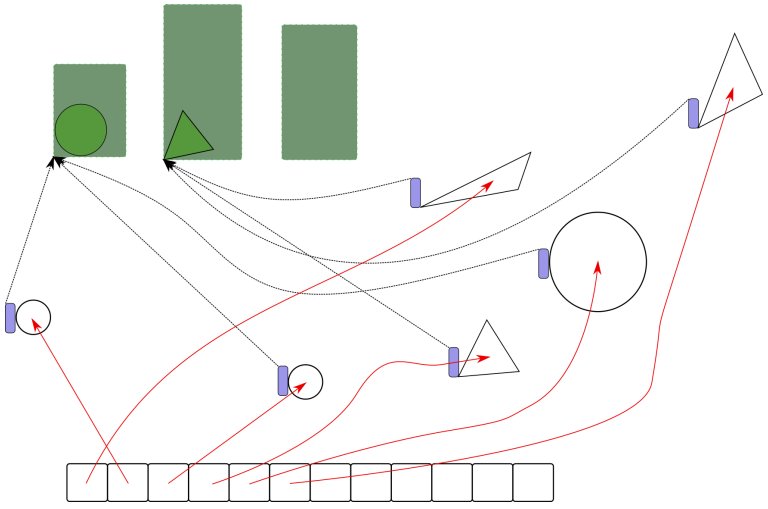
A LITTLE DEMO

CALLING VIRTUAL FUNCTIONS: HOW IT WORKS



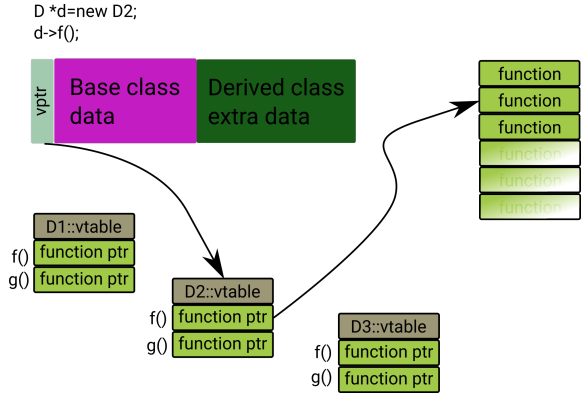
- For classes with virtual functions, the compiler inserts an invisible pointer member to the data and additional book keeping code
- There is a table of virtual functions for each derived class, with entries pointing to function code somewhere
- The `vptr` pointer points to the *vtable* of that particular class

CALLING VIRTUAL FUNCTIONS: HOW IT WORKS



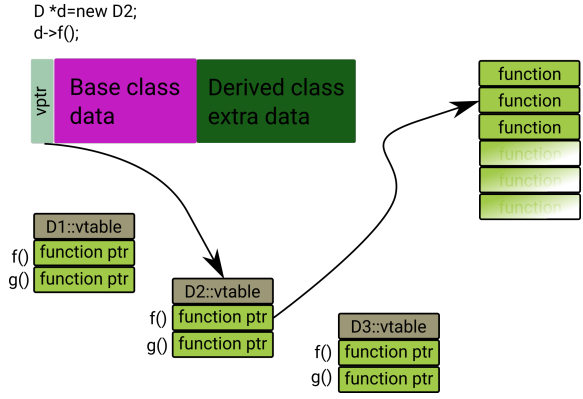
CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual! The overhead, with modern machines and compilers, is not huge. But abusing this feature will hurt performance



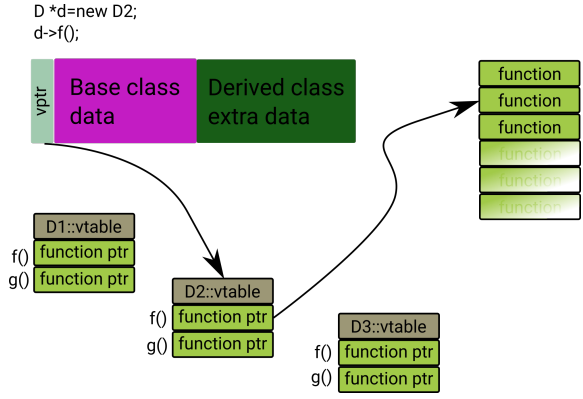
CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual! The overhead, with modern machines and compilers, is not huge. But abusing this feature will hurt performance



CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual! The overhead, with modern machines and compilers, is not huge. But abusing this feature will hurt performance



- But if virtual functions offer the cleanest solution with acceptable performance, don't invent weird things to avoid them!

CLASS INHERITANCE

Inherit or include as data member ?

```
1  class DNA {
2  ...
3    std::valarray<char> seq;
4  };
5  class Cell : public DNA ???
6  or
7  class Cell {
8  ...
9    DNA mydna;
10 };
```

- A derived class extends the concept represented by its base class in some way.
- Although this extension might mean addition of new data members,

$$B = A \oplus newdata$$

does not necessarily mean the class for B should inherit from the class for A

CLASS INHERITANCE

Inherit or include as data member ?

```
1  class DNA {
2  ...
3    std::valarray<char> seq;
4  };
5
6  class Cell : public DNA ???
7
8  or
9
10 class Cell {
11 ...
12   DNA mydna;
13 };
14
```

is* vs *has

- A good guide to decide whether to inherit or include is to ask whether the concept B contains an object A, or whether any object of type B is also an object of type A, like a monkey is a mammal, and a triangle is a polygon.
- is $\Longrightarrow$ inherit . has $\Longrightarrow$ include

CLASS INHERITANCE

Inheritance summary

- Base classes to represent common properties of related types : e.g. all proteins are molecules, but all molecules are not proteins. All triangles are polygons, but not all polygons are triangles.
- Less code: often, only one or two properties need to be changed in an inherited class
- Helps create reusable code
- A base class may or may not be constructable (`Polygon` as opposed to `Shape`)

CLASS DECORATIONS

More control over classes

- Possible to initialise data in class declaration
- Initialiser list constructors
- Delegating constructors allowed
- Inheriting constructors possible

```
1  class A {
2    int v[]{1, -1, -1, 1};
3  public:
4    A() = default;
5    A(std::initializer_list<int> &);
6    A(int i, int j, int k, int l)
7    {
8      v[0] = i;
9      v[1] = j;
10     v[2] = k;
11     v[3] = l;
12   }
13   //Delegate work to another constructor
14   A(int i, int j) : A(i, j, 0, 0) {}
15 };
16 class B : public A {
17 public:
18 // Inherit all constructors from A
19   using A::A;
20   B(string s);
21 };
```

MORE CONTROL OVER CLASSES

- Explicit `default`, `delete`, `override` and `final`
- "Explicit is better than implicit"
- More control over what the compiler does with the class
- Compiler errors better than hard to trace run-time errors due to implicitly generated functions

```
1  class A {
2    // Automatically generated is ok
3    A() = default;
4    // Don't want to allow copy
5    A(const A &) = delete;
6    A & operator=(const A &) = delete;
7    // Instead, allow a move constructor
8    A(const A &&);
9    // Don't try to override this!
10   void getDrawPrimitives() final;
11   virtual void show(int i);
12 };
13 class B : public A
14 {
15   B() = default;
16   void show()override; //will be an error!
17 };
18
```

Exercise 1.1:

The directory `exercises/geometry` contains a set of files for the classes Point, Shape, Polygon, Circle, Triangle, and Quadrilateral. In addition, there is a `main.cc` and a `CMakeLists.txt`. Observe the use of the keywords like **`default`**, **`override`**, **`final`** etc. Familiarise yourself with

- Implementation of inherited classes
- Compiling multi-file projects
- The use of base class pointer arrays to work with heterogeneous types of objects

```
mkdir build
cd build
CXX=g++ cmake ..
make
```

Using STL containers and algorithms

ALGORITHMS

```cpp
// examples/strtrans.cc
#include <iostream>
#include <algorithm>
#include <string>
auto main() -> int {
    std::string name;
    std::cout << "What's your name ? ";
    getline(std::cin, name);
    auto bkpname {name};
    std::transform(begin(name), end(name), begin(name), toupper);
    std::cout << bkpname << " <---------> " << name << "\n";
}
```

- What does this code do ?

ALGORITHMS

```cpp
// examples/strtrans.cc
#include <iostream>
#include <algorithm>
#include <string>
auto main() -> int {
    std::string name;
    std::cout << "What's your name ? ";
    getline(std::cin, name);
    auto bkpname {name};
    std::transform(begin(name), end(name), begin(name), toupper);
    std::cout << bkpname << " <---------> " << name << "\n";
}
```

- What does this code do ?
- `std::transform` transforms each element in an input range, and writes the results to an output range using a given operation

ALGORITHMS

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <ranges>
5  #include <algorithm>
6  #include <string>
7  auto main(int argc, char* argv[]) -> int {
8    std::vector<std::string> names;
9    std::ifstream input_file{argv[1]};
10   std::string name;
11   while (getline(input_file, name))
12     if (not name.empty())
13       names.push_back(name);
14
15   std::ranges::sort(names);
16 //
17 //
18 //
19 //
20
21   for (auto n : names)
22     std::cout << n << "\n";
23 }
```

- What does this code do ?

ALGORITHMS

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <ranges>
5  #include <algorithm>
6  #include <string>
7  auto main(int argc, char* argv[]) -> int {
8    std::vector<std::string> names;
9    std::ifstream input_file{argv[1]};
10   std::string name;
11   while (getline(input_file, name))
12     if (not name.empty())
13       names.push_back(name);
14
15   std::ranges::sort(names);
16 //
17 //
18 //
19 //
20
21   for (auto n : names)
22     std::cout << n << "\n";
23 }
```

- What does this code do ?
- `vector`, `string` grow to accommodate any new element added using `push_back`

ALGORITHMS

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <ranges>
5  #include <algorithm>
6  #include <string>
7  auto main(int argc, char* argv[]) -> int {
8    std::vector<std::string> names;
9    std::ifstream input_file{argv[1]};
10   std::string name;
11   while (getline(input_file, name))
12     if (not name.empty())
13       names.push_back(name);
14
15   std::ranges::sort(names);
16 //
17 //
18 //
19 //
20
21   for (auto n : names)
22     std::cout << n << "\n";
23 }
```

- What does this code do ?
- `vector`, `string` grow to accommodate any new element added using `push_back`
- `sort` sorts a range in increasing order

ALGORITHMS

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <ranges>
5  #include <algorithm>
6  #include <string>
7  auto main(int argc, char* argv[]) -> int {
8    std::vector<std::string> names;
9    std::ifstream input_file{argv[1]};
10   std::string name;
11   while (getline(input_file, name))
12     if (not name.empty())
13       names.push_back(name);
14
15   std::ranges::sort(names);
16 //
17 //
18 //
19 //
20
21   for (auto n : names)
22     std::cout << n << "\n";
23 }
```

- What does this code do ?
- `vector`, `string` grow to accommodate any new element added using `push_back`
- `sort` sorts a range in increasing order
- What is "increasing" order is decided by using the operator `<` to compare elements of the sequence

ALGORITHMS WITH LAMBDA FUNCTIONS

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <ranges>
5  #include <algorithm>
6  #include <string>
7  auto main(int argc, char* argv[]) -> int {
8    std::vector<std::string> names;
9    std::ifstream input_file{argv[1]};
10   std::string name;
11   while (getline(input_file, name))
12     if (not name.empty())
13       names.push_back(name);
14
15   std::ranges::sort(names,
16             [](auto name1, auto name2) {
17                 return name1 > name2;
18             }
19   );
20
21   for (auto n : names)
22     std::cout << n << "\n";
23 }
```

- What does this code do ?

ALGORITHMS WITH LAMBDA FUNCTIONS

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <ranges>
#include <algorithm>
#include <string>
auto main(int argc, char* argv[]) -> int {
  std::vector<std::string> names;
  std::ifstream input_file{argv[1]};
  std::string name;
  while (getline(input_file, name))
    if (not name.empty())
      names.push_back(name);

  std::ranges::sort(names,
            [](auto name1, auto name2) {
                return name1 > name2;
            }
  );

  for (auto n : names)
    std::cout << n << "\n";
}
```

- What does this code do ?
- We can give `std::sort` a comparison function as the sorting criterion

ALGORITHMS WITH LAMBDA FUNCTIONS

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <ranges>
#include <algorithm>
#include <string>
auto main(int argc, char* argv[]) -> int {
  std::vector<std::string> names;
  std::ifstream input_file{argv[1]};
  std::string name;
  while (getline(input_file, name))
    if (not name.empty())
      names.push_back(name);

  std::ranges::sort(names,
            [](auto name1, auto name2) {
                return name1 > name2;
            }
  );

  for (auto n : names)
    std::cout << n << "\n";
}
```

- What does this code do ?
- We can give `std::sort` a comparison function as the sorting criterion
- This can be used to order the elements in lots of different ways. Like sorting in decreasing order.

ALGORITHMS WITH LAMBDA FUNCTIONS

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <string>
auto main(int argc, char* argv[]) -> int
{
  std::vector<std::string> names;
  std::ifstream input_file{argv[1]};
  std::string name;
  while (getline(input_file, name))
    if (not name.empty())
      names.push_back(name);

  std::ranges::sort(names,
            [](auto name1, auto name2) {
              return name1.length() <
                     name2.length();
            }
  );
  for (auto n : names)
    std::cout << n << "\n";
}
```

- What does this code do ?
- We can give `std::sort` a comparison function as the sorting criterion
- This can be used to order the elements in lots of different ways. Like sorting in decreasing order.
- Or, sorting by the length of the strings ...

ALGORITHMS WITH LAMBDA FUNCTIONS

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <string>
auto main(int argc, char* argv[]) -> int
{
  using namespace std;
  vector<std::string> names;
  ifstream input_file{argv[1]};
  string name;
  while (getline(input_file, name))
    if (not name.empty()) names.push_back(name);

  sort(names.begin(), names.end(),
       [](auto name1, auto name2) -> bool {
         return name1.length() < name2.length();
       }
  );

  for (auto n : names) cout << n << "\n";
}
```

- `sort()` needs a function comparing two elements
- If we have such a function, we can pass its name
- If we don't, we can *kind of* write the content of the function, as the argument to the function `sort()`
- These kind of functions, declared as shown are called "lambda functions"
- Notation resembles a mapping $a, b, c... \mapsto value$ from some inputs to an output value, although frequently we skip the trailing return type if the return type is unambiguous

LAMDA FUNCTIONS

```cpp
auto my_cmp(string_view n1, string_view n2)
    -> int
{
  return n1.length() < n2.length();
}

std::sort(names.begin(), names.end(), my_cmp);

std::sort(names.begin(), names.end(),
          [](auto name1, auto name2) {
            return name1.length() <
                   name2.length();
          }
);
}
```

```cpp
double x{1.45};
//
//
//
//
//
y = sin(x);
//
y = sin(1.45);
//
//
//
//
//
//
```

- By themselves, "nameless functions"
- Passed as comparison or filtering criteria etc. to generic functions like `sort`, which can work with any "callable object"

Exercise 1.2:

In the working directory for the course chapter, you will find a file with the often used "lorem ipsum" text. Write a program that takes a text file, and finds all words shorter than 3 letters. If you need to use a lambda function, copy one from one of the slides and modify its code. We will learn its exact syntax later!

Function and class templates

FUNCTION OVERLOADING

```
auto power(int x, unsigned n) -> unsigned
{
    ans = 1;
    for (; n > 0; --n) ans *= x;
    return ans;
}
auto power(double x, double y) -> double
{
    return exp(y * log(x));
}
```

```
auto someother(double mu, double alpha,
                int rank) -> double
{
    double st=power(mu,alpha)*exp(-mu);

    if (n_on_bits(power(rank,5))<8)
        st=0;

    return st;
}
```

- When specialised strategies are needed to accomplish the same task for different types

FUNCTION OVERLOADING

```
auto power(int x, unsigned n) -> unsigned
{
    ans = 1;
    for (; n > 0; --n) ans *= x;
    return ans;
}
auto power(double x, double y) -> double
{
    return exp(y * log(x));
}
```

```
auto someother(double mu, double alpha,
                int rank) -> double
{
    double st=power(mu,alpha)*exp(-mu);

    if (n_on_bits(power(rank,5))<8)
        st=0;

    return st;
}
```

- When specialised strategies are needed to accomplish the same task for different types
- Static polymorphism: no virtual dispatch, everything resolved at compilation time

FUNCTION OVERLOADING

```
1  void copy(int* start, int* end, int* start2)
2  {
3      for (; start != end; ++start, ++start2) {
4          *start2 = *start;
5      }
6  }
7  void copy(string* start, string* end,
8                   string* start2)
9  {
10     for (; start != end; ++start, ++start2) {
11         *start2 = *start;
12     }
13 }
14 void copy(double* start, double* end,
15                  double* start2)
16 {
17     for (; start != end; ++start, ++start2) {
18         *start2 = *start;
19     }
20 }
21 double a[10], b[10];
22 copy(a, a + 10, b);
```

- When specialised strategies are needed to accomplish the same task for different types
- Static polymorphism: no virtual dispatch, everything resolved at compilation time
- But sometimes we need the opposite: same operations to be performed on different kinds of input

INTRODUCTION TO C++ TEMPLATES

```
1  void copy(int* start, int* end, int* start2)
2  {
3      for (; start != end; ++start, ++start2) {
4          *start2 = *start;
5      }
6  }
7  void copy(string* start, string* end,
8                   string* start2)
9  {
10     for (; start != end; ++start, ++start2) {
11         *start2 = *start;
12     }
13 }
14 void copy(double* start, double* end,
15                  double* start2)
16 {
17     for (; start != end; ++start, ++start2) {
18         *start2 = *start;
19     }
20 }
21 double a[10], b[10];
22 copy(a, a + 10, b);
```

Same operations on different types

- Exactly the same high level code
- Assigning a string to another may involve very different low level operations compared to assigning an integer to another. But once we have written our string class, we may write the exact same code for the string and integer versions of this kind of operations!
- Couldn't we automate the process of writing the 3 variants shown, by perhaps, using a placeholder type, and generating the right variant wherever required ?

INTRODUCTION TO C++ TEMPLATES

```
Dear compiler, in the following, T is a placeholder!
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?

INTRODUCTION TO C++ TEMPLATES

```
Dear compiler, in the following, T is a placeholder!
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy<double>(a, a + 10, b);
string names[10], onames[10];
copy<string>(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?

INTRODUCTION TO C++ TEMPLATES

```
template <class T>
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy<double>(a, a + 10, b);
string names[10], onames[10];
copy<string>(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?
- For the first point : Sure!

INTRODUCTION TO C++ TEMPLATES

```
template <class T>
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy(a, a + 10, b);
string names[10], onames[10];
copy(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?
- For the first point : Sure!
- For the second point: the compiler already knows those types based on the inputs at the point of usage!

INTRODUCTION TO C++ TEMPLATES

```
template <class T>
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy(a, a + 10, b);
string names[10], onames[10];
copy(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?
- For the first point : Sure!
- For the second point: the compiler already knows those types based on the inputs at the point of usage!
- Test it!
  `examples/template_intro.cc`

INTRODUCTION TO C++ TEMPLATES

```
template <class T>
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy(a, a + 10, b);
string names[10], onames[10];
copy(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?
- For the first point : Sure!
- For the second point: the compiler already knows those types based on the inputs at the point of usage!
- Test it!
  `examples/template_intro.cc`

Although we seemingly call a function we only wrote once, with different kinds of inputs, the compiler sees these as calls to two different functions. No runtime decision is needed to find the function to call.

TEMPLATES

Generic code The logic of the copy operation is quite simple. Given a pair of "iterators" (Behaviourally pointer like entities: can be advanced along a sequence, can be dereferenced) `first` and `last` in an input sequence, and a target location `result` in an output sequence, we want to:

- Loop over the input sequence
- For each position of the input iterator, copy the current element to the output iterator position
- Increment the input and output iterators
- Stop if the input iterator has reached `last`

A TEMPLATE FOR A GENERIC COPY OPERATION

```
1  template <class InputIterator, class OutputIterator>
2  OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result)
3  {
4    while (first != last) *result++ = *first++;
5    return result;
6  }
```

C++ template notation

- A template with which to generate code!
- If you had iterators to two kinds of sequences, you could substitute them in the above template and have a nice copy function!
- The compiler does the necessary substitution when you try to use the function
- The compiler needs access to the template source code at the point where it is trying to instantiate it!

ORDERED PAIRS

```
1   struct double_pair
2   {
3       double first, second;
4   };
5   ...
6   double_pair coords[100];
7   ...
8   struct int_pair
9   {
10      int first, second;
11  };
12  ...
13  int_pair line_ranges[100];
14  ...
15  struct int_double_pair
16  {
17      // wait!
18      // can I make a template out of it?
19  };
```

Class templates

- Classes can be templates too

ORDERED PAIRS

```
1  pair<double,double> coords[100];
2  pair<int,int> line_ranges[100];
3  pair<int,double> whatever;
```

`pair<int,double>`, after the template substitutions, becomes

```
struct pair<int, double>
{
    int first;
    double second;
};
```

Class templates

- Classes can be templates too
- Generated when the template is “instantiated”

```
1  template <class T, class U>
2  struct pair
3  {
4      T first;
5      U second;
6  };
```

ORDERED PAIRS

```
1  pair<double,double> coords[100];
2  pair<int,int> line_ranges[100];
3  pair<int,double> whatever;
```

`pair<int,double>`, after the template substitutions, becomes

```
struct pair<int, double>
{
    int first;
    double second;
};
```

Class templates

- Classes can be templates too
- Generated when the template is “instantiated”

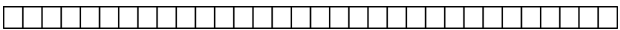
```
1  template <class T, class U>
2  struct pair
3  {
4      T first;
5      U second;
6  };
```

- Useful for creating many generic types

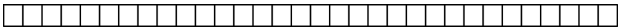
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...

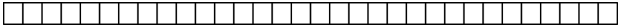
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 

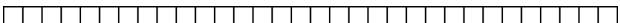
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 
- The code required to write containers of `int`, `double`, `complex_number` or any other class type will only differ by the type of the elements

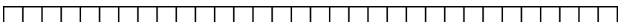
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 
- The code required to write containers of `int`, `double`, `complex_number` or any other class type will only differ by the type of the elements
- The template captures the essential structure, and we don't need to separately develop, debug or test these parametrised types for every possible element type

CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 
- The code required to write containers of `int`, `double`, `complex_number` or any other class type will only differ by the type of the elements
- The template captures the essential structure, and we don't need to separately develop, debug or test these parametrised types for every possible element type
- No inheritance relationship between vectors of different types

CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 
- The code required to write containers of `int`, `double`, `complex_number` or any other class type will only differ by the type of the elements
- The template captures the essential structure, and we don't need to separately develop, debug or test these parametrised types for every possible element type
- No inheritance relationship between vectors of different types
- No inheritance relationship required between entities which can be vector elements

VARIABLE TEMPLATES

```cpp
template <class T> constexpr auto algocategory = 0;
template<> constexpr auto algocategory<int> = 1;
template<> constexpr auto algocategory<long> = 1;
template<> constexpr auto algocategory<int*> = 2;
template<> constexpr auto algocategory<long*> = 2;
template <class T>
auto proc(T x)
{
    if constexpr (algocategory<T> == 2) {
        std::cout << "Using method for category 2 \n";
    } else if constexpr (algocategory<T> == 1) {
        std::cout << "Using method for category 1 \n";
    } else {
        std::cout << "Using method for category 0 \n";
    }
}
```

```cpp
auto main() -> int
{
    int v{7};
    proc(1);
    proc(1.);
    proc(1L);
    proc(v);
    proc(&v);
}
```

- Can be a static data member of a class or a global variable parametrised by template parameters
- Can be used along with `compile time if` statements to decide between different algorithms

NOT A TEXT SUBSTITUTION ENGINE!

Template specialisation

```
1  template <class T>
2  class vector {
3    // implementation of a general
4    // vector for any type T
5  };
6  template <>
7  class vector<bool> {
8    // Store the true false values
9    // in a compressed way, i.e.,
10   // 32 of them in a single int
11 };
12 void somewhere_else()
13 {
14   vector<bool> A;
15   // Uses the special implementation
16 }
```

- Templates are defined to work with generic template parameters
- But special values of those parameters, which should be treated differently, can be specified using "template specialisations" as shown
- If a matching specialisation is found, it is preferred over the general template

```
1  template <class A, class B>
2  constexpr auto are_same = false;
3  template <class A>
4  constexpr auto are_same<A, A> = true;
5  static_assert(are_same<int, long>); // Fails
6  using Integer = int;
7  static_assert(are_same<int, Integer>); // Passes
```

NOT A TEXT SUBSTITUTION ENGINE!

Recursion and integer arithmetic

```
1  template <unsigned N> constexpr unsigned fact = N * fact<N-1>;
2  template <> constexpr unsigned fact<0> = 1U;
3  static_assert(fact<7> == 5040)
```

- Templates support recursive instantiation
- Combined with specialisation to terminate recursion
- Recursion and specialisation can be used to emulate “loop” like calculations via tail-recursion

Exercise 1.3:

The example source file `examples/no_textsub.cc` demonstrates recursion and specialisation in templates, and uses `static_assert` to verify that the compiler does the arithmetic.

NOT A TEXT SUBSTITUTION ENGINE!

Because: SFINAE

```
1  template <bool Cond, class T> struct enable_if {};
2  template <class T> struct enable_if<true, T> { using type = T; }
3
4  template <class T>
5  auto func(T x) -> enable_if<sizeof(T) == 8UL, T>::type {
6  //impl1
7  }
8  template <class T>
9  auto func(T x) -> enable_if<sizeof(T) != 8UL, T>::type {
10 //impl2
11 }
```

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror
- If substituting a template parameter results in incomplete or invalid function declarations, that overload is ignored.
- The compiler simply tries to find another template with the same name that might match
- If it can't find any, then you have an error

NOT A TEXT SUBSTITUTION ENGINE!

Because: concepts

```
1  template <class T>
2  auto func(T x) -> T requires (sizeof(T) == 8UL) {
3  //impl1
4  }
5  template <class T>
6  auto func(T x) -> T requires (sizeof(T) != 8UL) {
7  //impl2
8  }
```

- Different implementations can be provided requiring different properties of the input type
- Before C++20, this sort of selection was done using `std::enable_if`. Now, `concepts` provide a far cleaner alternative.

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- The `darray` class we saw earlier in some examples represents a dynamic array, like the `std::vector`. It is a good example to illustrate more about class templates

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- The `darray` class we saw earlier in some examples represents a dynamic array, like the `std::vector`. It is a good example to illustrate more about class templates
- We want to be able to initialise our `darray<T>` like this:

```
darray<double> D(400, 0.);
darray<string> S{"A", "B", "C"};
darray<int> I{1, 2, 3, 4, 5};
```

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- The `darray` class we saw earlier in some examples represents a dynamic array, like the `std::vector`. It is a good example to illustrate more about class templates
- We want to be able to initialise our `darray<T>` like this:

```
darray<double> D(400, 0.);
darray<string> S{"A", "B", "C"};
darray<int> I{1, 2, 3, 4, 5};
```

- And then we want to be able to use it as follows...

```
for (auto i = 0UL; i < D.size(); ++i) {
    D[i] = i * i;
    std::cout << D[i] << "\n";
}
```

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- Making it into a template and writing many of the special functions is easy.

```
template <class T>
class darray {
    std::unique_ptr<T[]> dat;
    size_t sz{};
public:
    darray() = default;
    ~darray() = default;
    darray(const darray& other);
    darray(darray&&) noexcept = default;
    darray& operator=(const darray& other);
    darray& operator=(darray&&) noexcept = default;
};
```

- Using the `unique_ptr` to manage the heap allocation/deallocation means we don't need to do anything special for default constructor, destructor and the move operations. Only copy needs to be carefully implemented!

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- To initialise our `darray<T>` like this:

```
darray<string> S{"A", "B", "C"};
darray<int> I{1, 2, 3, 4, 5};
```

we need an initializer_list constructor

```
darray(initializer_list<T> l) {
    arr = std::make_unique<T[]>(l.size());
    for (auto i{0UL}; auto&& el : l) arr[i++] = el;
}
```

A DYNAMIC ARRAY CLASS TEMPLATE

```
1  template <class T>
2  class darray {
3  public:
4      auto operator[](size_t i) const -> T { return arr[i]; }
5      auto operator[](size_t i) -> T& { return arr[i]; }
6  };
```

- Two versions of the `[]` operator for read-only and read/write access

A DYNAMIC ARRAY CLASS TEMPLATE

```
1  template <class T>
2  class darray {
3  public:
4      auto operator[](size_t i) const -> T { return arr[i]; }
5      auto operator[](size_t i) -> T& { return arr[i]; }
6  };
```

- Two versions of the `[]` operator for read-only and read/write access
- Use `const` qualifier in any member function which does not change the object

TYPE DEDUCTIONS

- Template parameters can be type names or compile time constant values of different types.
- Until C++20, non-type template parameters were limited to integral types. Now, a lot of other types are allowed.

```
1  template <class T, int N>
2  struct my_array {
3      T data[N];
4  };
```

TYPE DEDUCTIONS

- Template parameters can be type names or compile time constant values of different types.
- Until C++20, non-type template parameters were limited to integral types. Now, a lot of other types are allowed.
- Can be used to specify compile time constant sizes

```
1  template <class T, int N>
2  struct my_array {
3      T data[N];
4  };
```

```
1  template <class T,
2            int nrows, int ncols>
3  struct my_matrix {
4      T data[nrows*ncols];
5  };
```

TYPE DEDUCTIONS

- Template parameters can be type names or compile time constant values of different types.
- Until C++20, non-type template parameters were limited to integral types. Now, a lot of other types are allowed.
- Can be used to specify compile time constant sizes
- but also give you a peculiar kind of " function " in effect
- Old uses of template integer arithmetic are by now obsolete. `constexpr` functions constitute a vastly superior alternative.
- But, type-deductions remain an important use for template meta-programs

```
1  template <class T, int N>
2  struct my_array {
3      T data[N];
4  };
```

```
1  template <class T,
2            int nrows, int ncols>
3  struct my_matrix {
4      T data[nrows*ncols];
5  };
```

```
1  template <int i, int j>
2  struct mult {
3      static const int value=i*j;
4  };
5  ...
6  my_array< mult<19,21>::value > vals;
```

EVALUATE DEPENDENT TYPES

- Suppose we want to implement a template function

```
1  template <class T> U f(T a);
```

such that when T is a non-pointer type, U should take the value T. But if T is itself a pointer, U is the type obtained by dereferencing the pointer

EVALUATE DEPENDENT TYPES

- Suppose we want to implement a template function

```
1  template <class T> U f(T a);
```

such that when T is a non-pointer type, U should take the value T. But if T is itself a pointer, U is the type obtained by dereferencing the pointer

- We could use a template function to "compute" the type U like this:

```
1  template <class T> struct remove_pointer { using type = T; };
2  template <class T> struct remove_pointer<T*> { using type = T; };
```

EVALUATE DEPENDENT TYPES

- Suppose we want to implement a template function

```
1  template <class T> U f(T a);
```

such that when T is a non-pointer type, U should take the value T. But if T is itself a pointer, U is the type obtained by dereferencing the pointer

- We could use a template function to "compute" the type U like this:

```
1  template <class T> struct remove_pointer { using type = T; };
2  template <class T> struct remove_pointer<T*> { using type = T; };
```

- We can then declare the function as:

```
1  template <class InputType>
2  auto f(InputType a) -> remove_pointer<InputType>::type ;
```

TYPE FUNCTIONS

- Compute properties of types
- Compute dependent types
- Typically used with convenient alias template declarations for the dependent type or the constant value

```
1  template <class T1, class T2>
2  std::is_same<T1,T2>::value
3
4  template <class T>
5  std::is_integral<T>::value
6
7  template <class T>
8  std::make_signed<T>::type
9
10 template <class T>
11 std::remove_reference<T>::type
12
13 template <class T>
14 using remove_reference_t =
15     typename remove_reference<T>::type;
16
17 template <class T>
18 inline constexpr bool is_integral_v =
19     std::is_integral<T>::value;
```

STATIC_ASSERT WITH TYPE TRAITS

```
1  #include <iostream>
2  #include <type_traits>
3  template < class T, class U>
4  auto some_calc(T x, U y)
5  {
6    static_assert(std::is_convertible_v<T, U>,
7    "The type of the argument x must be convertible to type U");
8  // ...
9  }
10 auto main() -> int
11 {
12   some_calc(4.0, "target"); //Compiler error!
13   ...
14 }
```

- Use `static_assert` and `type_traits` in combination with `constexpr`

Exercise 1.4: static_assert2.cc

TYPETRAITS

Unary predicates

- `is_integral_v<T>` : `T` is an integer type
- `is_const_v<T>` : has a `const` qualifier
- `is_class_v<T>` : struct or class
- `is_pointer_v<T>` : Pointer type
- `is_abstract_v<T>` : Abstract class with at least one pure virtual function
- `is_copy_constructible_v<T>` : Class allows copy construction
- `is_same_v<T1,T2>` : `T1` and `T2` are the same types
- `is_base_of_v<T,D>` : `T` is base class of `D`
- `is_convertible_v<T,T2>` : `T` is convertible to `T2`

FORWARDING REFERENCES

```
1  template <class T>
2  auto wrapperfunc( T&& t)
3  {
4      other(std::forward<T>(t));
5  }
6  auto main() -> int
7  {
8      std::string x{"Solar"};
9      std::string y{"System"};
10     wrapperfunc(x);
11     wrapperfunc(x + " " + y);
12 }
```

- Function argument written as if it were an R-value reference to a cv-unqualified template parameter
- If `wrapperfunc` is called with a constant L-value, `T` is deduced to be a constant L-value reference, and `other` receives a constant L-value reference
- If `wrapperfunc` is called with an L-value, `T` is deduced to be an L-value reference, and `other` receives an L-value reference
- If the input is an R-value, then `T` is inferred to be a plain type, and `forward` ensures that `other` receives an R-value reference

Constrained templates

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Some way to impose requirements on permissible matches for the template parameters. Something like:

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Some way to impose requirements on permissible matches for the template parameters. Something like:

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

- If we could do that, we can combine the generality of templates with the selectiveness of function overloading

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Some way to impose requirements on permissible matches for the template parameters. Something like:

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

- If we could do that, we can combine the generality of templates with the selectiveness of function overloading
- We can

CONCEPTS

Named requirements on template parameters

- `concept`s are named requirements on template parameters, such as `floating_point`, `contiguous_range`
- If `MyAPI` is a `concept`, and `T` is a type, `MyAPI<T>` evaluates at compile time to either true or false.
- Concepts can be combined using conjunctions (`&&`) and disjunctions (`||`) to make other concepts.
- A `requires` clause introduces a constraint on a template type

A suitably designed set of concepts can greatly improve readability of template code

CREATING CONCEPTS

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of<B, D>;
```

```
class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The **`requires`** *expression* allows creation of syntactic requirements as shown in the last two examples.

CREATING CONCEPTS

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of<B, D>;
```

```
class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The **`requires`** *expression* allows creation of syntactic requirements as shown in the last two examples.

CREATING CONCEPTS

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

```
template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[OUL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The **`requires`** *expression* allows creation of syntactic requirements as shown in the last two examples.

CREATING CONCEPTS

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

```
template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[0UL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The **`requires`** *expression* allows creation of syntactic requirements as shown in the last two examples.
- The requires expression can contain a parameter list and a brace enclosed sequence of requirements, which can be:
  - type requirements, e.g., typename T::value_type;
  - simple requirements as shown on the left
  - compound requirements with optional return type constraints, e.g.,

```
{ A[0UL] } -> convertible_to<int>;
```

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
1  template <class T>
2  auto sqr(const T& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...

USING CONCEPTS

```
1
2   auto sqr(const auto& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...
- Functions with `auto` in their parameter list are implicitly function templates

USING CONCEPTS

```
1
2   auto sqr(const auto& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...
- Functions with `auto` in their parameter list are implicitly function templates

USING CONCEPTS

```
1
2   auto sqr(const auto& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...
- Functions with `auto` in their parameter list are implicitly function templates

Exercise 1.11:

The program `examples/gcd_w_concepts.cc` shows a very small concept definition and its use in a function calculating the greatest common divisor of two integers.

Exercise 1.12:

The series of programs `examples/generic_func1.cc` through `generic_func4.cc` shows some trivial functions implemented with templates with and without constraints. The files contain plenty of comments explaining the rationale and use of concepts.

OVERLOADING BASED ON CONCEPTS

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4
5
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9
10
11
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration.

OVERLOADING BASED ON CONCEPTS

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts

OVERLOADING BASED ON CONCEPTS

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."

OVERLOADING BASED ON CONCEPTS

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.

OVERLOADING BASED ON CONCEPTS

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships

OVERLOADING BASED ON CONCEPTS

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships
- Not a "quack like a duck, or bust" approach either.

OVERLOADING BASED ON CONCEPTS

```
1  template <class N>
2  concept Number = std::floating_point<N>
3                   or std::integral<N>;
4  template <class N>
5  concept NotNumber = not Number<N>;
6  void proc(Number auto&& x) {
7      std::cout << "Called proc for numbers";
8  }
9  void proc(NotNumber auto&& x) {
10     std::cout << "Called proc for non-numbers";
11 }
12 auto main() -> int {
13     proc(-1);
14     proc(88UL);
15     proc("0118 999 88199 9119725   3");
16     proc(3.141);
17     proc("eighty"s);
18 }
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships
- Not a "quack like a duck, or bust" approach either.
- Entirely compile time mechanism

Exercise 1.13:

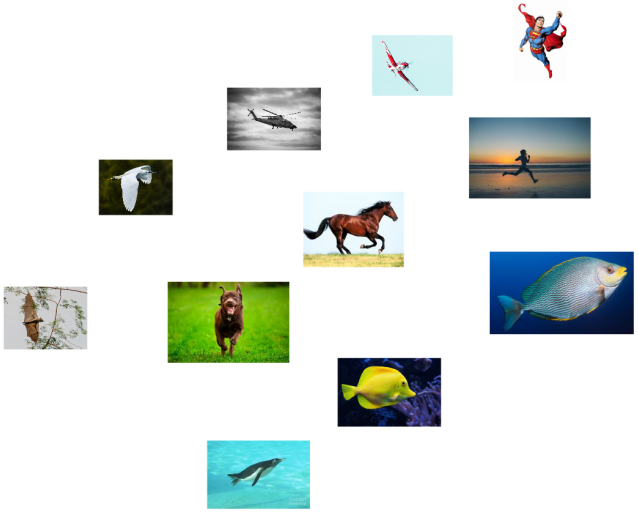
Check how you can use concepts to implement alternative versions of a function based on properties of the input parameters! The program `examples/overload_w_concepts.cc` contains the code just shown. Can you add another overload that is picked if the input type is an array? This means, if `X` is the input parameter, `X[i]` is syntactically valid for unsigned integer `i`. The array version should be picked up if the input is a `vector`, `array`, etc., but also `string`. How would you prevent the `string` and C-style strings picking the array version?

PREDEFINED USEFUL CONCEPTS

Many concepts useful in building our own concepts are available in the standard library header `<concepts>`.

- `same_as`
- `convertible_to`
- `signed_ingegral`, `unsigned_integral`
- `floating_point`
- `assignable_from`
- `swappable`, `swappable_with`
- `derived_from`
- `move_constructible`, `copy_constructible`
- `invocable`
- `predicate`
- `relation`

CONCEPTS: SUMMARY



f(those who can fly)

f(runners)

f(swimmers)

Lambda Functions

FUNCTION LIKE ENTITIES

- In C++, there are a few different constructs which can be used in a context requiring a "function"
- Functions in all varieties constitute one category (`inline` or not, `constexpr` or not, `virtual` or not ...)
- Classes may overload the function call operator `operator()` to give us another type of callable object
- Lambda functions are similar, language provided entities

```
1   class Wave {
2   double A, ome, pha;
3   public:
4   auto operator()(double t) -> double
5   {
6       return A * sin(ome * t + pha);
7   }
8   };
9   void elsewhere()
10  {
11      Wave W{1.0, 0.15, 0.9};
12      for (auto i = 0; i < 100; ++i) {
13          std::cout << i << W(i) << "\n";
14      }
15  }
```

LAMBDA FUNCTIONS

- Locally defined callable entities
- Uses
  - Effective use of STL
  - Initialisation of const
  - Concurrency
  - New loop styles
- Like a function object defined on the spot
- Fine grained control over the visibility of the variables in the surrounding scope

```
1  sort(begin(v), end(v), [](auto x, auto y) {
2      return x > y;
3  });
4
5  const auto inp_file = [](){
6      string resourcefl;
7      cout << "resource file : ";
8      cin >> resourcefl;
9      return resourcefl;
10 }();
11 tbb::parallel_for(0, 1000000, [](int i){
12     // process element i
13 });
```

LAMBDA FUNCTIONS

Function

```
auto sqr(double x) -> double
{
    return x * x;
}
```

Lambda expression

```
auto lsqr = [](double x) -> double
{
    return x * x;
};
```

- Normal C++ functions can not be defined in block scope
- Lambda expressions are expressions, which when evaluated yield callable entities. Like 2^9 is an expression, which when evaluated yields 512.
- Such callable entities can be created in global as well as block scope

LAMBDA FUNCTIONS

Function

```
auto sqr(double x) -> double
{
    return x * x;
}
```

Lambda expression

```
auto lsqr = [](double x) -> double
{
    return x * x;
};
```

- The lambda expression contains information which is used to make the callable entity: such as, expected input, output and the body("recipe").
- Unlike normal functions, which have names, these callable entities themselves are **nameless**, but named variables can be constructed out of them, if desired. Those named variables can then be used like functions.

LAMBDA FUNCTIONS

Function

```
auto sqr(double x) -> double
{
    return x * x;
}
```

Lambda expression

```
auto lsqr = [](double x) -> double
{
    return x * x;
};
```

- The lambda expression contains information which is used to make the callable entity: such as, expected input, output and the body("recipe").
- Unlike normal functions, which have names, these callable entities themselves are **nameless**, but named variables can be constructed out of them, if desired. Those named variables can then be used like functions.

```
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = 0.;
for (auto i = 0UL; i < X.size(); ++i) {
    sqsum += sqr(X[i]);
}
```

LAMBDA FUNCTIONS

Function

```
auto sqr(double x) -> double
{
    return x * x;
}
```

Lambda expression

```
auto lsqr = [](double x) -> double
{
    return x * x;
};
```

- The lambda expression contains information which is used to make the callable entity: such as, expected input, output and the body("recipe").
- Unlike normal functions, which have names, these callable entities themselves are **nameless**, but named variables can be constructed out of them, if desired. Those named variables can then be used like functions.

```
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = 0.;
for (auto i = 0UL; i < X.size(); ++i) {
    sqsum += lsqr(X[i]);
}
```

LAMBDA FUNCTIONS

```
template <Callable F>
auto aggregate(const std::vector<double>& inp, F f) -> double
{
    auto s{0.};
    for (auto i = 0UL; i < inp.size(); ++i) { s += f(inp[i]); }
    return s;
}
```

- Typical use: arguments to higher order functions. Function parameter that specifies an operation to be performed on a value or (as in this case) a range of values

LAMBDA FUNCTIONS

```
template <Callable F>
auto aggregate(const std::vector<double>& inp, F f) -> double
{
    auto s{0.};
    for (auto i = 0UL; i < X.size(); ++i) { s += f(X[i]); }
    return s;
}
// ...
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = aggregate(X, sqr);
```

- Typical use: arguments to higher order functions. Function parameter that specifies an operation to be performed on a value or (as in this case) a range of values
- Named callable entities can be used when available.

LAMBDA FUNCTIONS

```
template <Callable F>
auto aggregate(const std::vector<double>& inp, F f) -> double
{
    auto s{0.};
    for (auto i = 0UL; i < X.size(); ++i) { s += f(X[i]); }
    return s;
}
// ...
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = aggregate(X, lsqr);
```

- Typical use: arguments to higher order functions. Function parameter that specifies an operation to be performed on a value or (as in this case) a range of values
- Named callable entities can be used when available.

LAMBDA FUNCTIONS

```
template <Callable F>
auto aggregate(const std::vector<double>& inp, F f) -> double
{
    auto s{0.};
    for (auto i = 0UL; i < X.size(); ++i) { s += f(X[i]); }
    return s;
}
// ...
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = aggregate(X, [](double x) -> double { return x * x; });
```

- Typical use: arguments to higher order functions. Function parameter that specifies an operation to be performed on a value or (as in this case) a range of values
- Named callable entities can be used when available.
- Often it is more convenient to pass a lambda expression, and let the higher order function create the callable entity it needs!

LAMBDA FUNCTIONS WITH ALGORITHMS

`std::for_each` is a higher order function, similar to this:

```
template <class InputIterator, class UnaryFunction>
void for_each(InputIterator start, InputIterator end, UnaryFunction f)
{
    for (auto it = start; it != end; ++it) f(*it);
}
```

LAMBDA FUNCTIONS WITH ALGORITHMS

`std::for_each` is a higher order function, similar to this:

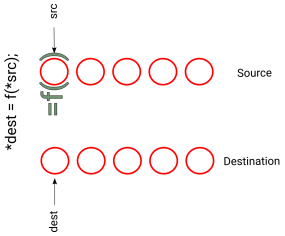
```
template <class InputIterator, class UnaryFunction>
void for_each(InputIterator start, InputIterator end, UnaryFunction f)
{
    for (auto it = start; it != end; ++it) f(*it);
}
```

What do the following lines do ?

```
1  std::vector X{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
2  for_each(X.begin(), X.end(), [](int& elem){ elem = elem * elem; });
3  for_each(X.begin(), X.end(), [](int& elem){ elem -= 100; });
4  for_each(X.begin(), X.end(), [](int elem){ std::cout << elem << "\n"; });
```

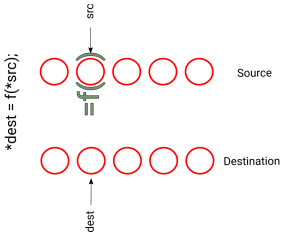
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



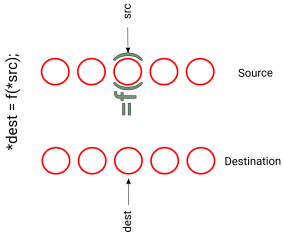
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



LAMBDA FUNCTIONS WITH ALGORITHMS

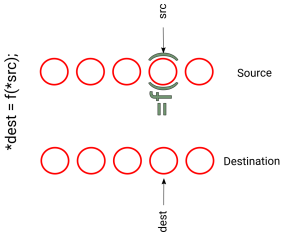
`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



JÜLICH Forschungszentrum

LAMBDA FUNCTIONS WITH ALGORITHMS

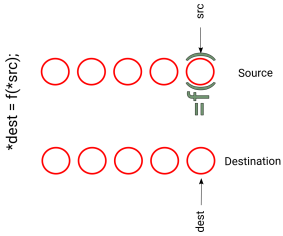
`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



JÜLICH Forschungszentrum

LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



JÜLICH Forschungszentrum

LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:

```
template <class InputIt, class OutputIt,
          class UnaryFunction>
void transform(InputIt start, InputIt end,
               OutputIt out,
               UnaryFunction f)
{
    for (; start != end; ++start, ++out)
        *out = f(*start);
}
```

LAMBDA FUNCTIONS WITH ALGORITHMS

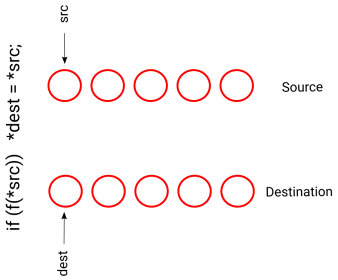
`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:

What do the following lines do ?

```
1  std::vector X{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
2  std::vector<int> Y;
3  transform(X.begin(), X.end(), std::back_inserter(Y),
4            [](int elem){ return elem * elem; });
```

LAMBDA FUNCTIONS WITH ALGORITHMS

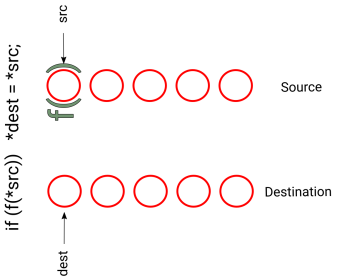
`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



JÜLICH Forschungszentrum

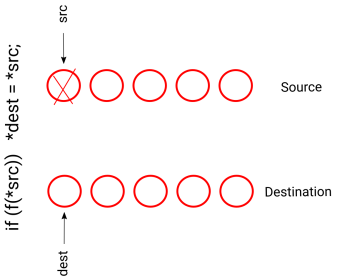
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



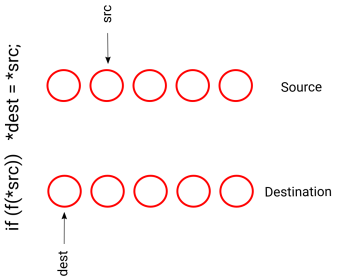
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



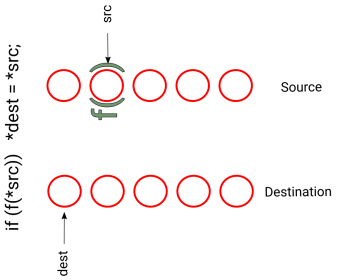
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



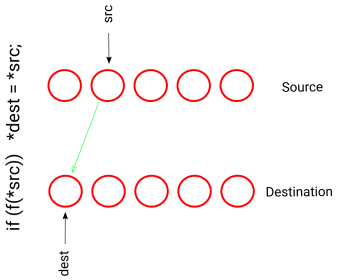
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



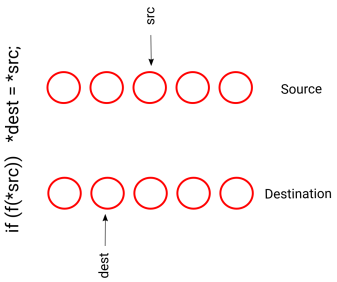
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



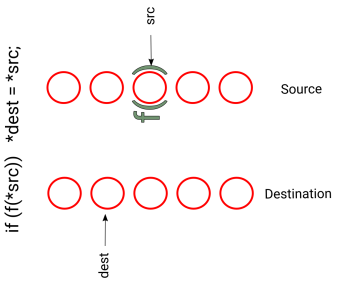
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



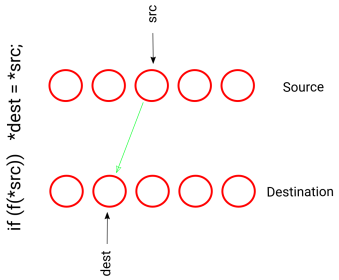
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



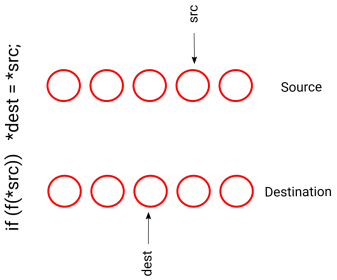
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



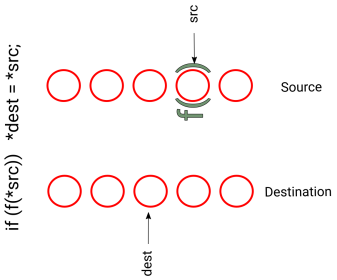
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



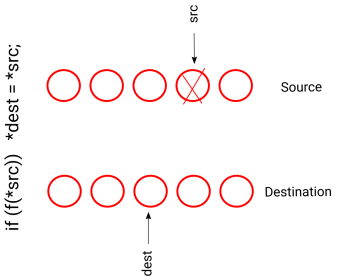
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



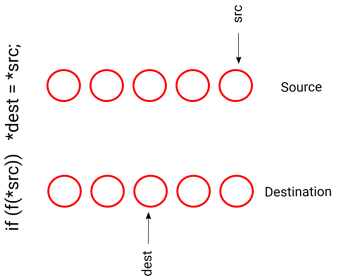
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



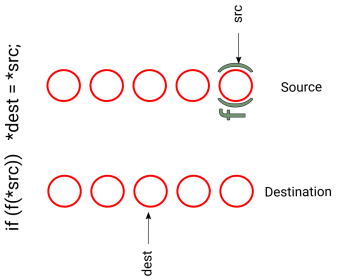
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



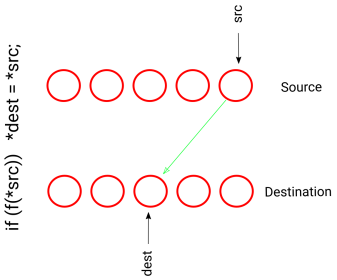
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:

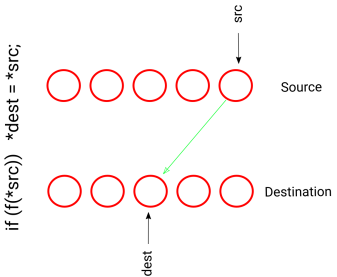


LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:

What do the following lines do ?

```
1  std::vector X{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
2  std::vector<int> Y;
3  copy_if(X.begin(), X.end(), std::back_inserter(Y),
4            [](int elem){ return elem % 3 == 0; });
```



Exercise 1.14:

Use the notebook `lambda_practice_0.ipynb` to quickly practice writing a few small lambdas and using them with a few standard library algorithms.

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

- Copy to `w` all positive elements

```
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i>0; });
```

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

- Copy to `w` all positive elements

```
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i>0; });
```

- Copy to `w` all elements larger than a user specified value

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

- Copy to `w` all positive elements

```
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i>0; });
```

- Copy to `w` all elements larger than a user specified value
- This does not work

```
std::cin >> lim;
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){  return i > lim ; });
// Lambda function has its own scope, and lim is not visible
```

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

- Copy to `w` all positive elements

```
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i>0; });
```

- Copy to `w` all elements larger than a user specified value
- This does not work

```
std::cin >> lim;
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){  return i > lim ; });
// Lambda function has its own  scope , and lim is not visible
```

- A way to make the lambda selectively aware of chosen variables in its context:

```
std::cin >> lim;
copy_if(v.begin(), v.end(), back_inserter(w),
                                [lim](int i){  return i > lim;  });
// Lambda function "captures" lim, and lim is now visible inside the lambda
```

LAMBDA EXPRESSIONS: SYNTAX

```
[capture] <templatepars> (arguments) lambda-specifiers { body }
```

- Variables in the body of a lambda function are either passed as function arguments or "captured", or are global variables
- Function arguments field is optional if empty. e.g. `[&cc]{ return cc++; }`
- The *lambda-specifiers* field can contain a variety of things: Keywords `mutable`, `constexpr` or `consteval`, exception specifiers, attributes, the return type, and any `requires` clauses. All of these are optional.
- The return type is optional if there is one return statement. e.g.
  `[a,b,c](int i) mutable { return a*i*i + b*i + c; }`
- The optional keyword `mutable` can be used to create lambdas with state
- `auto` can be used to declare the formal input parameters of the lambda (since C++14)
- Template parameters can be optionally provided where shown (since C++20)

EXPLICIT TEMPLATE PARAMETERS FOR LAMBDA FUNCTIONS

```
1  // examples/saxpy_2.cc
2  // includes ...
3  auto main() -> int {
4      const std::vector inp1 { 1., 2., 3., 4., 5. };
5      const std::vector inp2 { 9., 8., 7., 6., 5. };
6      std::vector outp(inp1.size(), 0.);
7
8      auto saxpy = [] <class T, class T_in, class T_out>
9                   (T a, const T_in& x, const T_in& y, T_out& z) {
10         std::transform(x.begin(), x.end(), y.begin(), z.begin(),
11                        [a](T X, T Y){ return a * X + Y; });
12     };
13
14     std::ostream_iterator<double> cout { std::cout, "\n" };
15     saxpy(10., inp1, inp2, outp);
16     copy(outp.begin(), outp.end(), cout);
17 }
```

For normal function templates, we could easily express relationships among the types of different parameters. With C++20, we can do that for generic lambdas.

LAMBDA CAPTURE SYNTAX I

```
[capture]<templatepars> (arguments) lambda-specifiers { body }
```

- `[ ](int a, int b) -> bool { return a > b;}` : Capture nothing. Work only with the arguments passed, or global objects.
- `[=](int a) -> bool {return a > somevar;}` : Capture everything needed by value.
- `[&](int a){somevar += a;}` : Capture everything needed by reference.
- `[=,&somevar](int a){ somevar += max(a,othervar); }` : `somevar` by reference, but everything else as value.
- `[a,&b]{ f(a,b); }` : `a` by value, `b` by reference.
- `[a=std::move(b)]{ f(a,b); }` : Init capture. Create a variable `a` with the initializer given in the capture brackets. It is as if there were an implicit `auto` before the `a`.

Exercise 1.15:

The program `lambda_captures.cc` (alternatively, notebook `lambda_practice_1.ipynb`) declares a variable of the `Vbose` type (with all constructors, assignment operators etc. written to print messages), and then defines a lambda function. By changing the capture type, and the changing between using and not using the `Vbose` value inside the lambda function, try to understand, from the output, the circumstances under which the captured variables are copied into the lambda. In the cases where you see a copy, where does the copy take place ? At the point of declaration of the lambda or at the point of use ?

LAMBDA FUNCTIONS: CAPTURES

- Imagine there is a variable `int p=5` defined previously

LAMBDA FUNCTIONS: CAPTURES

- Imagine there is a variable `int p=5` defined previously
- We can "capture" `p` by value and use it inside our lambda

```
auto L = [p](int i){ std::cout << i*3 + p; };
L(3); // result : prints out 14
auto M = [p](int i){ p = i*3; }; // syntax error! p is read-only!
```

LAMBDA FUNCTIONS: CAPTURES

- Imagine there is a variable `int p=5` defined previously
- We can "capture" `p` by value and use it inside our lambda

```
auto L = [p](int i){ std::cout << i*3 + p; };
L(3); // result : prints out 14
auto M = [p](int i){ p = i*3; }; // syntax error! p is read-only!
```

- We can capture `p` by value (make a copy), but use the `mutable` keyword, to let the lambda function change its local copy of `p`

```
auto M = [p](int i) mutable { return p += i*3; };
std::cout << M(1) << " "; std::cout << M(2) <<" "; std::cout << p <<"\n";
// result : prints out "8 14 5"
```

LAMBDA FUNCTIONS: CAPTURES

- Imagine there is a variable `int p=5` defined previously
- We can "capture" `p` by value and use it inside our lambda

```
auto L = [p](int i){ std::cout << i*3 + p; };
L(3); // result : prints out 14
auto M = [p](int i){ p = i*3; }; // syntax error! p is read-only!
```

- We can capture `p` by value (make a copy), but use the `mutable` keyword, to let the lambda function change its local copy of `p`

```
auto M = [p](int i) mutable { return p += i*3; };
std::cout << M(1) << " "; std::cout << M(2) <<" "; std::cout << p <<"\n";
// result : prints out "8 14 5"
```

- We can capture `p` by reference and modify it

```
auto M = [&p](int i){ return p += i*3; };
std::cout << M(1) << " "; std::cout << M(2) << " "; std::cout << p << "\n";
// result : prints out "8 14 14"
```

NO DEFAULT CAPTURE!

`[]`	Capture nothing
`[=]`	Capture used by value (copy)
`[=,&x]`	Capture used by value, except x by reference
`[&]`	Capture used by reference
`[&,x]`	Capture used by reference, except x by value
`[a=init]`	Init capture

- A lambda with empty capture brackets is like a local function, and can be assigned to a regular function pointer. It is not aware of identifiers defined previously in its context
- When you use a (non-global) variable defined outside the lambda in the lambda, you have to capture it

STATEFUL LAMBDAS

- Mutable lambdas have "state", and remember any changes to the values captured by value
- Combined with "init capture", gives us interesting generator functions

```
1  vector<int> v, w;
2  generate_n(back_inserter(v), 100, [i=0]() mutable {
3      ++i;
4      return i*i;
5  });
6  // v = [1, 4, 9, 16 ... ]
7  generate_n(back_inserter(w), 50, [i=0, j=1]() mutable {
8      i = std::exchange(j, j+i); // exchange(a,b) sets a to b and returns the old value of a
9      return i;
10 });
11 // See the videos on Fibonacci sequence on the
12 // YouTube channel "C++ Weekly" by Jason Turner
13 // w = [1, 1, 2, 3, 5, 8, 11 ...]
```

Exercise 1.16:

The program `mutable_lambda.cc` shows the use of mutable lambdas for sequence initialisation.