PROGRAMMING IN C++

Jülich Supercomputing Centre

May 11, 2022 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Day 3

DESIGN GOALS

- Correctness
- Readability
- Extendability
- Speed
- Adaptability

A large scale software project is better off being built out of components which are resilient to unforeseen changes.

DEPENDENCIES

- Impede modifications
- Hamper testing
- Increase rebuild times

- Good design helps us control dependencies.
- Variation points
- Flexible adaptible software

Guideline: Keep dependencies among software components to a minimum.

ENCAPSULATION

- Member functions abstracting properties
- Resilient to internal data reorganisation
- More flexible design

```
class complex_number {
public:
    double real, imag;
    double modulus();
};
```

```
class complex_number {
public:
    auto real() const -> double;
    auto imag() const -> double;
    void real(double x);
    void imag(double x);
    auto modulus() const -> double;
};
```

ENCAPSULATION

- Scott Meyer: degree of encapsulation is gauged by the number of things which break if the internal design changes
- Less member functions : better!
- If a function can be implemented as a non-friend, non-member function, it should be.

```cpp
// Class definition: bare essentials
namespace ns {
class Example {
public:
    auto property1() const -> double;
    auto property2() const -> double;
};
}
```

```cpp
// Use case 1 header
namespace ns {
auto calc(Example & ex) {
  //ex.property1() + ...
  //ex.property2();
  return haha;
}
}
```

THE SOLID PRINCIPLES

- **S**ingle responsibility principle
- **O**pen-closed principle
- **L**iskov's substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

SRP: THE SINGLE RESPONSIBILITY PRINCIPLE

Every class should have a single responsibility and that responsibility should be entirely encapsulated by that class.

- However tempting it might seem, avoid adding member functions not related to the core idea of the class
- Related principle: Don't repeat yourself. Avoids opportunity for bugs and reduces maintenance overhead.

```cpp
class Rectangle {
public:
    auto area() const -> double;
    auto width() const -> double;
    auto height() const -> double;
    void width(double x);
    void height(double x);
    void draw() const;
};
```


JÜLICH Forschungszentrum

OCP: THE OPEN CLOSED PRINCIPLE

A software component should be open for extension, but closed for modifications.

- Closed: can be used by other components. Well defined stable interface.
- Open: Available for extension. Add new data fields, new functionality.
- Inheritance (possibly from abstract base classes)

LSP: LISKOV'S SUBSTITUTION PRINCIPLE

"If, for each object o_1 of type S, there is an object o_2 of type T, such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T."
– Barbara Liskov

- Subtypes must be able to substitute the base type
- Deriving type fully reflects the behaviour of the base class
- True "is a" relationship
- **Guideline:** Don't inherit and then restrict the derived class so that it loses some behaviour expected from the base class

ISP: THE INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to depend on methods they do not use.

- See under "encapsulation" above
- Avoid "fat" classes. When one client forces a change, every other client is affected, even if they are not using the same part of the fat class.
- Think how the functionality available through the namespace `std` is segregated.

DIP: THE DEPENDENCY INVERSION PRINCIPLE

1. High-level modules should not depend on low level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

- High level components own the interface they depend on.
- They specify their requirements.
- If low level components implement that interface, they can be used with the high level client interface.
- Cut the dependency chain
- Adaptor layers

SUMMARY

- Avoiding tight coupling between different components may require extra work at first, but wins out in the life time of a project.
- Assign responsibilities carefully.
- SOLID principles are known to help develop and maintain flexible software.

JÜLICH Forschungszentrum

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

```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);
//
//
//
//

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

- What does this code do ?

ALGORITHMS

```
#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);
//
//
//
//

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

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

ALGORITHMS

```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);
//
//
//
//

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

- 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

```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);
//
//
//
//

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

- 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

```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 ?

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

```
#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

```
#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 $a, b, c\ldots \mapsto value$ from some inputs to an output value, although frequently we skip the trailing return type if the return type is unambiguous

JÜLICH Forschungszentrum

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 2.1:

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

```cpp
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));
}
```

```cpp
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

```
void copy(int* start, int* end, int* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
void copy(string* start, string* end,
                string* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
void copy(double* start, double* end,
                double* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
double a[10], b[10];
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

```
void copy(int* start, int* end, int* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
void copy(string* start, string* end,
                 string* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
void copy(double* start, double* end,
                 double* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
double a[10], b[10];
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

```cpp
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`

JÜLICH Forschungszentrum

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.

JÜLICH Forschungszentrum

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

```cpp
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result)
{
  while (first != last) *result++ = *first++;
  return result;
}
```

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!

JÜLICH Forschungszentrum

ORDERED PAIRS

```cpp
struct double_pair
{
    double first, second;
};
...
double_pair coords[100];
...
struct int_pair
{
    int first, second;
};
...
int_pair line_ranges[100];
...
struct int_double_pair
{
    // wait!
    // can I make a template out of it?
};
```

Class templates

- Classes can be templates too

ORDERED PAIRS

```
pair<double,double> coords[100];
pair<int,int> line_ranges[100];
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"

```
template <class T, class U>
struct pair
{
    T first;
    U second;
};
```

ORDERED PAIRS

```
pair<double,double> coords[100];
pair<int,int> line_ranges[100];
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”

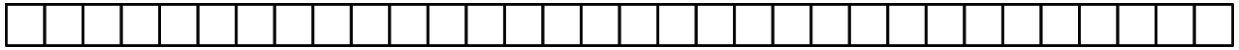
```
template <class T, class U>
struct pair
{
    T first;
    U second;
};
```

- Useful for creating many generic types

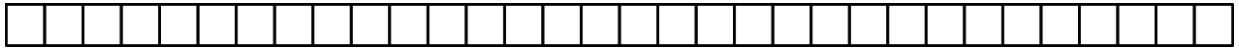
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

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

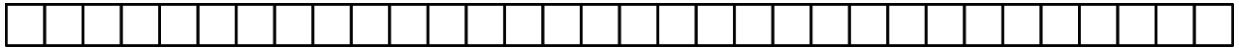
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

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

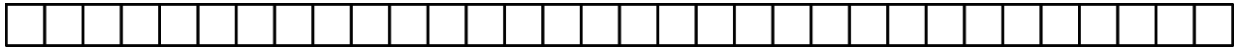
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `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

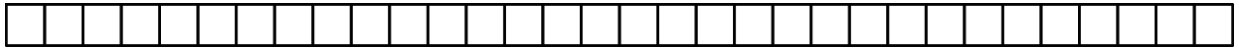
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `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

```
template <class T>
class vector {
  // implementation of a general
  // vector for any type T
};
template <>
class vector<bool> {
  // Store the true false values
  // in a compressed way, i.e.,
  // 32 of them in a single int
};
void somewhere_else()
{
  vector<bool> A;
  // Uses the special implementation
}
```

- 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

```
template <class A, class B>
constexpr auto are_same = false;
template <class A>
constexpr auto are_same<A, A> = true;
static_assert(are_same<int, long>); // Fails
using Integer = int;
static_assert(are_same<int, Integer>); // Passes
```

NOT A TEXT SUBSTITUTION ENGINE!

Recursion and integer arithmetic

```cpp
template <unsigned N> constexpr unsigned fact = N * fact<N-1>;
template <> constexpr unsigned fact<0> = 1U;
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 2.2:

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

```cpp
template <bool Cond, class T> struct enable_if {};
template <class T> struct enable_if<true, T> { using type = T; }

template <class T>
auto func(T x) -> enable_if<sizeof(T) == 8UL, T>::type {
//impl1
}
template <class T>
auto func(T x) -> enable_if<sizeof(T) != 8UL, T>::type {
//impl2
}
```

- **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

```
template <class T>
auto func(T x) -> T requires (sizeof(T) == 8UL) {
//impl1
}
template <class T>
auto func(T x) -> T requires (sizeof(T) != 8UL) {
//impl2
}
```

- 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

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

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

A DYNAMIC ARRAY CLASS TEMPLATE

```cpp
template <class T>
class darray {
public:
    auto operator[](size_t i) const -> T { return arr[i]; }
    auto operator[](size_t i) -> T& { return arr[i]; }
};
```

- 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.

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

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

```cpp
template <class T, int N>
struct my_array {
    T data[N];
};
```

```cpp
template <class T,
          int nrows, int ncols>
struct my_matrix {
    T data[nrows*ncols];
};
```

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

```cpp
template <class T, int N>
struct my_array {
    T data[N];
};
```

```cpp
template <class T,
          int nrows, int ncols>
struct my_matrix {
    T data[nrows*ncols];
};
```

```cpp
template <int i, int j>
struct mult {
    static const int value=i*j;
};
...
my_array< mult<19,21>::value > vals;
```

EVALUATE DEPENDENT TYPES

- Suppose we want to implement a template function

```
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

```cpp
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:

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

EVALUATE DEPENDENT TYPES

- Suppose we want to implement a template function

```
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:

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

- We can then declare the function as:

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


JÜLICH Forschungszentrum

TYPE FUNCTIONS

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

```cpp
template <class T1, class T2>
std::is_same<T1,T2>::value

template <class T>
std::is_integral<T>::value

template <class T>
std::make_signed<T>::type

template <class T>
std::remove_reference<T>::type

template <class T>
using remove_reference_t =
    typename remove_reference<T>::type;

template <class T>
inline constexpr bool is_integral_v =
    std::is_integral<T>::value;
```

STATIC_ASSERT WITH TYPE TRAITS

```
#include <iostream>
#include <type_traits>
template < class T, class U>
auto some_calc(T x, U y)
{
  static_assert(std::is_convertible_v<T, U>,
  "The type of the argument x must be convertible to type U");
// ...
}
auto main() -> int
{
  some_calc(4.0, "target"); //Compiler error!
  ...
}
```

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

Exercise 2.3: 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

```cpp
template <class T>
auto wrapperfunc( T&&  t)
{
    other(std::forward<T>(t));
}
auto main() -> int
{
    std::string x{"Solar"};
    std::string y{"System"};
    wrapperfunc(x);
    wrapperfunc(x + " " + y);
}
```

- 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

```cpp
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.,

```cpp
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:

```cpp
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.

JÜLICH Forschungszentrum

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.

JÜLICH Forschungszentrum

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>;
    ```


JÜLICH Forschungszentrum

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

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

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

```cpp
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

JÜLICH Forschungszentrum

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

Exercise 2.4:

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 2.5:

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

```cpp
// examples/overload_w_concepts.cc
template <class N>
concept Number = std::is_floating_point_v<N>
                 || std::is_integral_v<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x)
{
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x)
{
    std::cout << "Called proc for non-numbers";
}
auto main() -> int
{
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just "documentation" or decoration.
- Different versions of a function can be chosen based on concepts by writing suitable overload sets.
- 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 more constrained overload is chosen.
- Not based on any inheritance relationships among types
- Not a "quack like a duck, or bust" approach either.
- Entirely compile time mechanism

JÜLICH Forschungszentrum

Exercise 2.6:

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`

VARIADIC TEMPLATES

```cpp
template <class ... Args>
auto countArgs(Args ... args) -> int
{
  return (sizeof ...args);
}

std::cout << "Num args = " << countArgs(1, "one", "ein", "uno", 3.232) << '\n';
```

- Templates with arbitrary number of arguments
- Typical use: template meta-programming
- Recursion, partial specialisation
- The `...` is actual code! Not blanks for you to fill in!

PARAMETER PACK

- The ellipsis (`...`) template argument is called a parameter pack [1]
- It represents 0 or more arguments which could be type names, integers or other templates :

```
template <class ... Args> class mytuple;
// The above can be instantiated with :
mytuple<int, int, double, string> t1;
mytuple<int> t2;
mytuple<> t3;
```

- Definition: A template with at least one parameter pack is called a variadic template

[1] http://en.cppreference.com/w/cpp/language/parameter_pack

PARAMETER PACK

```cpp
//examples/variadic_1.cc
template <class ... Types> void f(Types ... args);
template <class Type1, class ... Types> void f(Type1 arg1, Types ... rest) {
    std::cout << typeid(arg1).name() << ``: '' << arg1 << ``\n'';
    f(rest ...);
}
template <> void f() {}
auto main() -> int
{
  int i{3}, j{};
  const char * cst{"abc"};
  std::string cppst{"def"};
  f(i, j, true, k, l, cst, cppst);
}
```

- Divide argument list into first and rest
- Do something with first and recursively call template with rest
- Specialise for the case with 1 or 0 arguments

PARAMETER PACK EXPANSION

- `pattern ...` is called a parameter pack expansion
- It applies a pattern to a comma separated list of instantiations of the pattern
- If we are in a function :

```cpp
template <class ... Types> void g(Types ... args)
```

- `args...` means the list of arguments used for the function.
- Calling `f(args ...)` in `g` will call `f` with same arguments
- Calling `f(h(args)...)` in `g` will call `f` with an argument list generated by applying function `h` to each argument of `g`
- In `g(true,"abc",1)`, `f(h(args)...)` means `f(h(true),h("abc"),h(1))`

PARAMETER PACK EXPANSION

```
template <class ... Types> void f(Types ... args);
template <class Type1, class ... Types> void f(Type1 arg1, Types ... rest) {
    std::cout <<" The first argument is "<<arg1
             <<". Remainder argument list has "<<sizeof...(Types)<<" elements.\n";
    f(rest ...);
}
template <> void f() {}
template <class ... Types> void g(Types ... args) {
    std::cout << "Inside g: going to call function f with the sizes of "
              << "my arguments\n";
    f(sizeof(args)...);
}
```

- `sizeof...(Types)` retrieves the number of arguments in the parameter pack
- In `g` above, we call `f` with the sizes of each of the parameters passed to `g`
- Similarly, one can generate all addresses as `&args...`, increment all with `++args...` (examples `variadic_2.cc` and `variadic_3.cc`)

PARAMETER PACK EXPANSION: WHERE

```cpp
template <class ... Types> void f( Types & ... args ) {}
template <class ... Types> void h( Types ... args ) {
    f( std::cout << args << ``t'' ... );
    [=, &args ... ]{ return g( args... ); }();
    int t[sizeof...(args)]={ args ... };
    int s = 0;
    for (auto i : t) s += i;
    std::cout << "\nsum = " << s << "\n";
}
```

- Parameter pack expansion can be done in function parameter list, function argument list, template parameter list or template argument list
- Braced initializer lists
- Base specifiers and member initializer lists in classes
- Lambda captures

Exercise 2.7: Parameter packs

Study the examples `variadic_1.cc`, `variadic_2.cc` and `variadic_3.cc`. See where parameter packs are begin expanded, and make yourself familiar with this syntax.

FOLD EXPRESSIONS IN C++17

```cpp
#include <iostream>
template <class ... Args>
auto addup(Args ... args)
{
    return (1 + ... + args);
}
auto main() -> int
{
    std::cout << addup(1, 2, 3) << "\n";
    std::cout << addup(1, 2, 3, 4, 5) << "\n";
}
```

- `... op ppack` translates to reduce from the left with operator `op`
- `ppack op ...` means, reduce from the right with `op`
- `init op ... op ppack` reduces from the left, with initial value `init`
- `pack op ... op init` reduces from the right ...

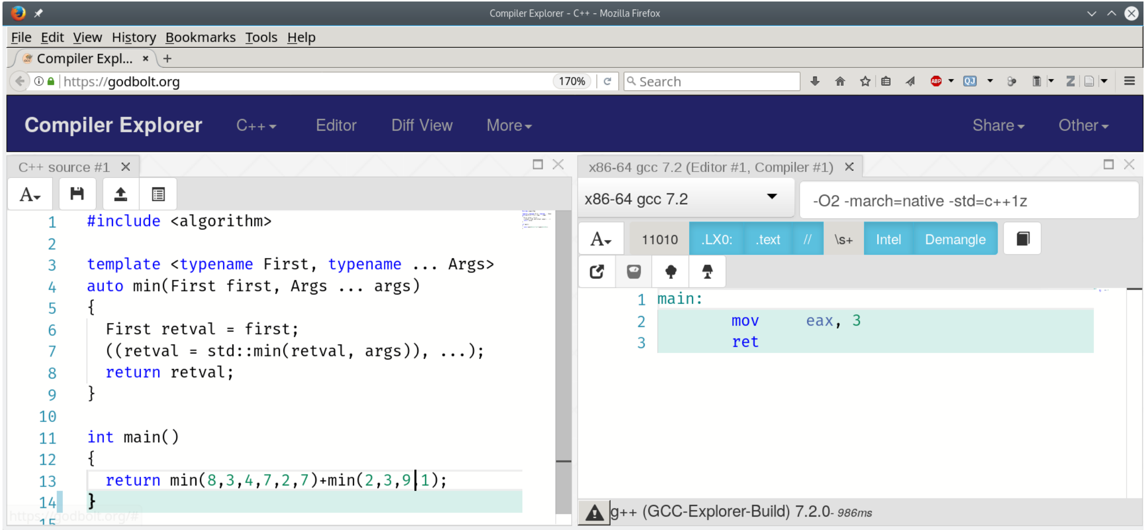
FOLD EXPRESSIONS IN C++17

```cpp
// examples/foldex_3.cc
#include <algorithm>
template <class First, class ... Args>
auto min(First first, Args ... args)
{
    First retval = first;
    ((retval = std::min(retval, args)), ...);
    return retval;
}

auto main() -> int
{
    return min(8, 3, 4, 7, 2, 7)
        + min(2, 3, 9, 1);
}
```

Application

- Fold expression with the comma operator
- Make even the number of arguments abstract



JÜLICH Forschungszentrum

TUPLES

```cpp
#include <tuple>
#include <iostream>
auto main() -> int
{
    std::tuple<int, int, std::string> name_i_j{0, 1, "Uralic"};
    auto t3 = std::make_tuple<int, bool>(2, false);
    auto t4 = std::tuple_cat(name_i_j, t3);
    std::cout << std::get<2>(t4) << '\n';
}
```

- Like `std::pair`, but with arbitrary number of members
- "Structure templates without names"
- Accessor "template functions" `std::get<index>` with `index` starting at 0
- Supports relational operators for lexicographical comparisons
- `tuple_cat(args ...)` concatenates tuples.

TUPLES

```cpp
auto f() -> std::tuple<int, int, string>; // elsewhere
auto main() -> int
{
    int i1;
    std::string name;
    std::tie(i1, std::ignore, name) = f();
}
```

- `tie(args ...)` "extracts a tuple" into pre-existing named variables.
- Some fields may be ignored during extraction using `std::ignore` as shown

PRINTING A TUPLE

```
template <class... Args>
auto operator<<(std::ostream& strm, const std::tuple<Args...>& t) -> std::ostream& {
    using namespace std;
    auto print_one = [&strm](const auto& onearg) -> decltype(strm) {
        using bare_type = remove_cvref_t<decltype(onearg)>;
        if constexpr (is_convertible_v<bare_type, string>)
            strm << quoted(onearg);
        else
            strm << onearg;
        return strm;
    };
    auto print_components = [&](auto&&... args) {
        size_t n {};
        ((print_one(args) << ((++n != sizeof...(args)) ? ", " : "")), ...);
    };
    strm << "[ ";
    apply(print_components, t);
    return strm << " ]";
}
```

- Helper lambda to print one element, quoted when it is a string, plain otherwise
- Fold expression and `std::apply` to print components

Exercise 2.8:

Printing a tuple is demonstrated in `print_tuple.cc`, `print_tuple_cxx17.cc` and `print_tuple_foldex.cc`.

FUN WITH FOLD EXPRESSIONS

We have an uncertain number of containers of arbitrary types, with arbitrary element types (which are known to be $<$ comparable), containing an arbitrary number of elements each. We need a tuple consisting of the largest element of each container. Write a function which will create that tuple from our inputs.

FUN WITH FOLD EXPRESSIONS

We have an uncertain number of containers of arbitrary types, with arbitrary element types (which are known to be $<$ comparable), containing an arbitrary number of elements each. We need a tuple consisting of the largest element of each container. Write a function which will create that tuple from our inputs.

```
auto max_of_multiple(auto&& ... containers)
{
    return std::make_tuple(std::ranges::max(containers) ...);
}
```

FUN WITH FOLD EXPRESSIONS

We need a function to replace each element of a vector with the averages of neighbours separated by some shifts. Write a function that takes the vector and the shifts as function arguments, and returns the smoothed vector. It should be possible to use the function for any given number of shifts.

```
auto conv(const std::vector<double>& inp, auto ... shift)
{
    std::vector<double> out(inp.size(), 0.);
    auto res_exp = std::views::iota(0, static_cast<int>(inp.size()))
        | std::views::transform([inp, shift...](auto index){
            auto S = inp.size();
            return (inp[
            (index + shift) > 0 ? (index + shift) % S : S + (index + shift) % S
                      ] + ...)
                  / (sizeof ... (shift));
        });

    std::ranges::copy(res_exp, out.begin());
    return out;
}
```

Exercise 2.9:

`fold_xpr_demo[2-4].cc` demostrate the last few applications of variadic templates, fold expressions and the new C++20 syntax for `auto` in function parameters. Build them with the proper include paths for printing tuples and ranges. The necessary headers for this functionality is in the `include` folder for the course.

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

```
class Wave {
double A, ome, pha;
public:
auto operator()(double t) -> double
{
    return A * sin(ome * t + pha);
}
};
void elsewhere()
{
    Wave W{1.0, 0.15, 0.9};
    for (auto i = 0; i < 100; ++i) {
        std::cout << i << W(i) << "\n";
    }
}
```

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

```cpp
sort(begin(v), end(v), [](auto x, auto y) {
    return x > y;
});

const auto inp_file = []{
    string resourcefl;
    cout << "resource file : ";
    cin >> resourcefl;
    return resourcefl;
}();
tbb::parallel_for(0, 1000000, [](int i){
    // process element i
});
```

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

```cpp
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:

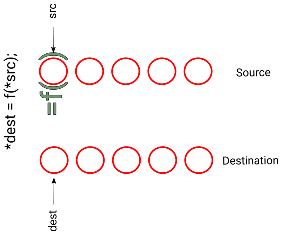
```
template <class InputIterator, class UnaryFunction>
void for_each(InputIterator start, InputIterator end, UnaryFunction f)
{
    for (auto it = start; it != end; ++it) f(*it);
}
```

What do the following lines do ?

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

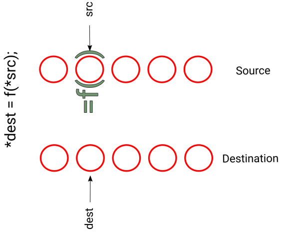
LAMBDA FUNCTIONS WITH ALGORITHMS

`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:



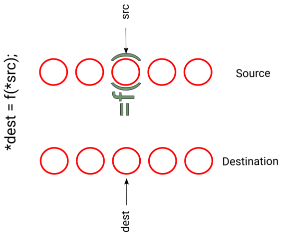
LAMBDA FUNCTIONS WITH ALGORITHMS

`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:



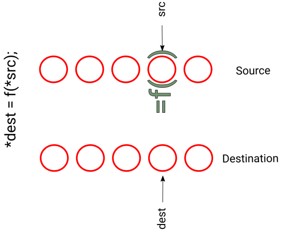
LAMBDA FUNCTIONS WITH ALGORITHMS

`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:



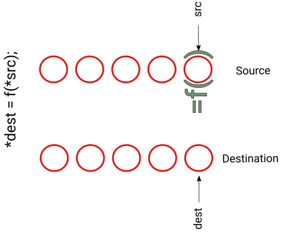
LAMBDA FUNCTIONS WITH ALGORITHMS

`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:

```
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);
}
```


JÜLICH Forschungszentrum

LAMBDA FUNCTIONS WITH ALGORITHMS

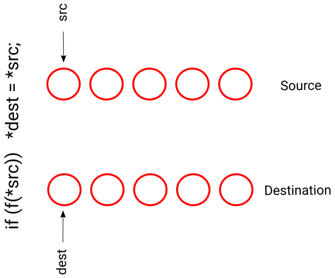
`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:

What do the following lines do ?

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

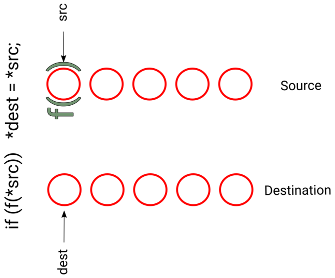
LAMBDA FUNCTIONS WITH ALGORITHMS

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



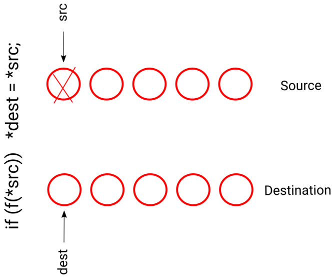
LAMBDA FUNCTIONS WITH ALGORITHMS

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



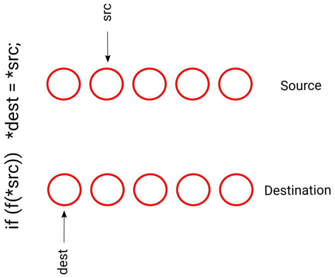
LAMBDA FUNCTIONS WITH ALGORITHMS

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



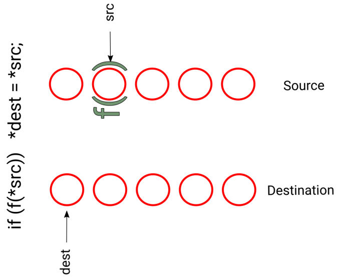
LAMBDA FUNCTIONS WITH ALGORITHMS

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



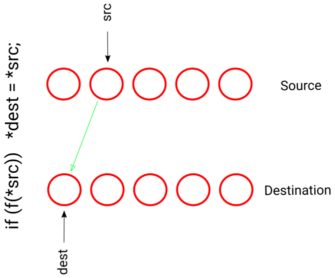
LAMBDA FUNCTIONS WITH ALGORITHMS

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



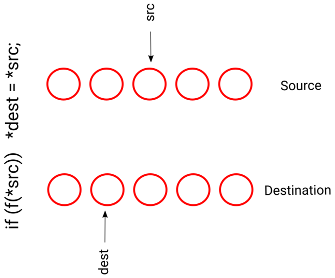
LAMBDA FUNCTIONS WITH ALGORITHMS

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



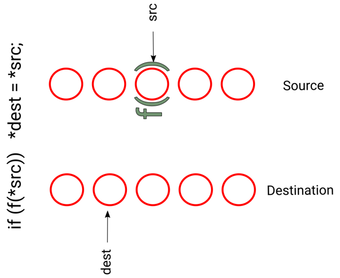
LAMBDA FUNCTIONS WITH ALGORITHMS

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



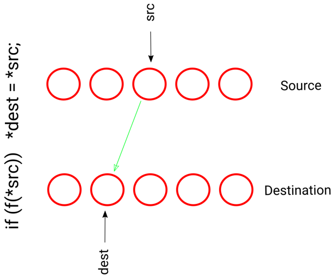
LAMBDA FUNCTIONS WITH ALGORITHMS

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



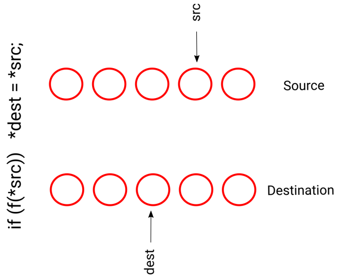
LAMBDA FUNCTIONS WITH ALGORITHMS

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



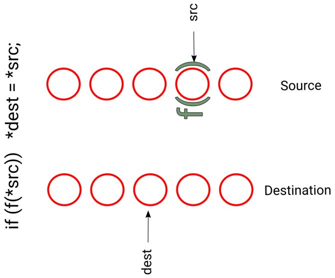
LAMBDA FUNCTIONS WITH ALGORITHMS

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



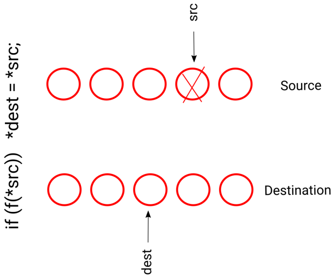
LAMBDA FUNCTIONS WITH ALGORITHMS

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



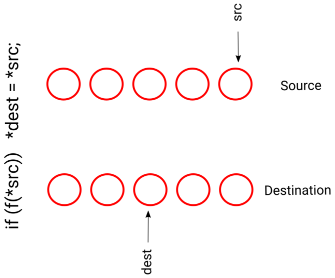
LAMBDA FUNCTIONS WITH ALGORITHMS

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



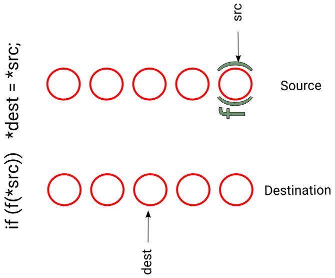
LAMBDA FUNCTIONS WITH ALGORITHMS

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



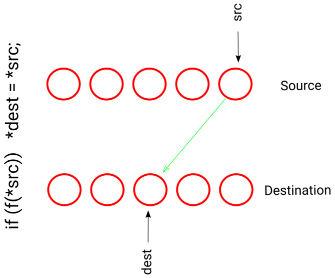
LAMBDA FUNCTIONS WITH ALGORITHMS

`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:

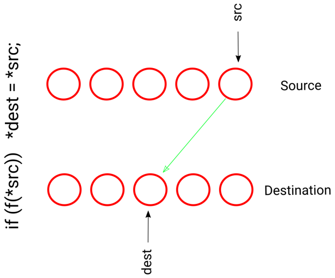


LAMBDA FUNCTIONS WITH ALGORITHMS

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

What do the following lines do ?

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



Exercise 2.10:

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

```cpp
// examples/saxpy_2.cc
// includes ...
auto main() -> int {
    const std::vector inp1 { 1., 2., 3., 4., 5. };
    const std::vector inp2 { 9., 8., 7., 6., 5. };
    std::vector outp(inp1.size(), 0.);

    auto saxpy = [] <class T, class T_in, class T_out>
                    (T a, const T_in& x, const T_in& y, T_out& z) {
        std::transform(x.begin(), x.end(), y.begin(), z.begin(),
                       [a](T X, T Y){ return a * X + Y; });
    };

    std::ostream_iterator<double> cout { std::cout, "\n" };
    saxpy(10., inp1, inp2, outp);
    copy(outp.begin(), outp.end(), cout);
}
```

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 2.11:

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

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

Exercise 2.12:

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