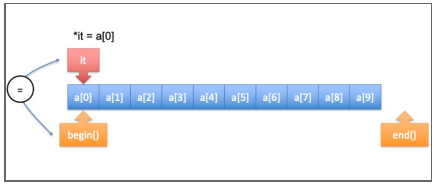
Object-oriented scientific programming with C++

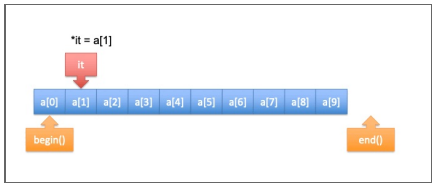
Matthias Möller, Jonas Thies, Cálin Georgescu, Jingya Li (Numerical Analysis, DIAM)
Lecture 6

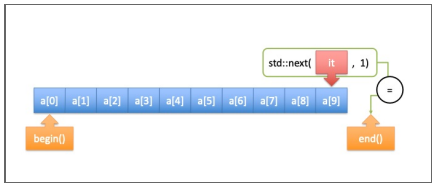
Goal of this lecture

- Self-written iterators for user-defined containers
- Introduction to the concept of expression templates
- Memory management with smart pointers
- Latest and greatest features of C++17 and 20
- ...

Iterators




*it = a[1]
it
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
begin()
end()


std::next(it , 1)
=
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
begin()
end()

Constant iterator over all entries

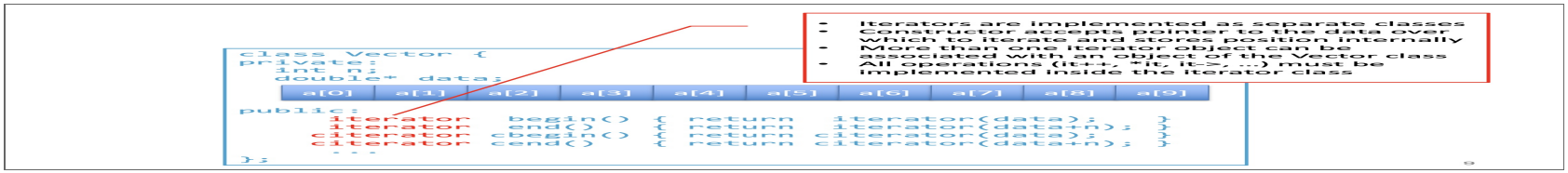
```
for (auto it = a.cbegin(); it != a.cend(); ++it)
      std::cout << *it << "\n";
```

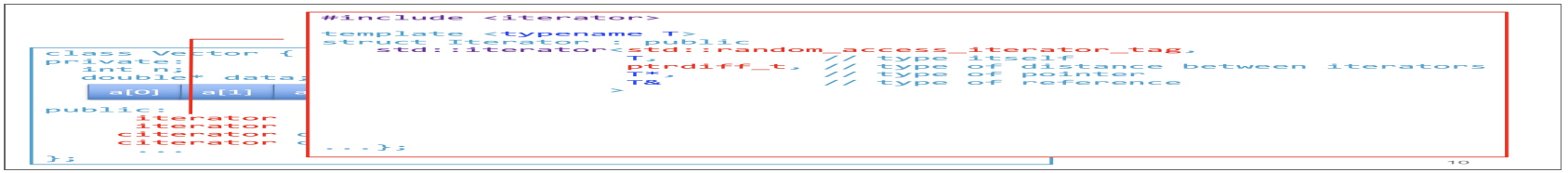
Non-constant iterator over all entries

```
for (auto it = a.begin(); it != a.end(); ++it)
    *it++;
```

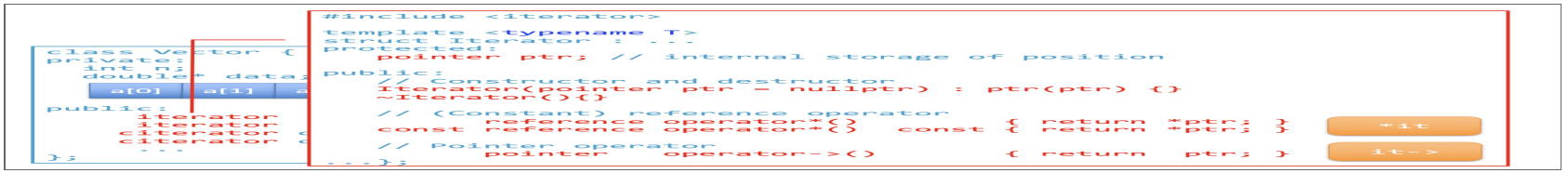
Iterators for user-defined containers

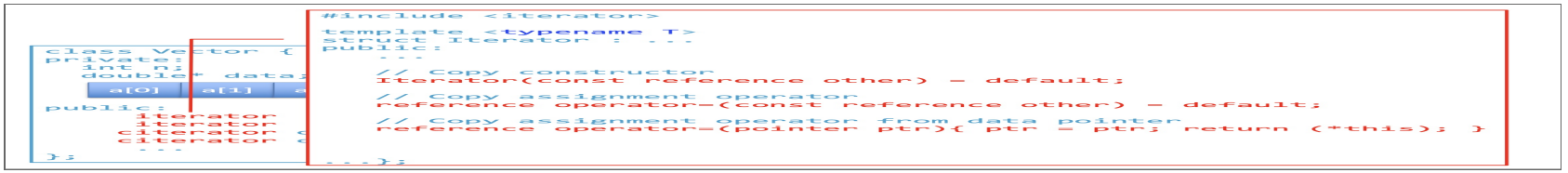
```
class Vector {
private:
  int n;
  double* data;
public:
      iterator   begin() { return   iterator(data); }
      iterator   end()   { return   iterator(data+n); }
      citerator  cbegin() { return citerator(data); }
      citerator  cend()   { return citerator(data+n); }
       ...
};
```

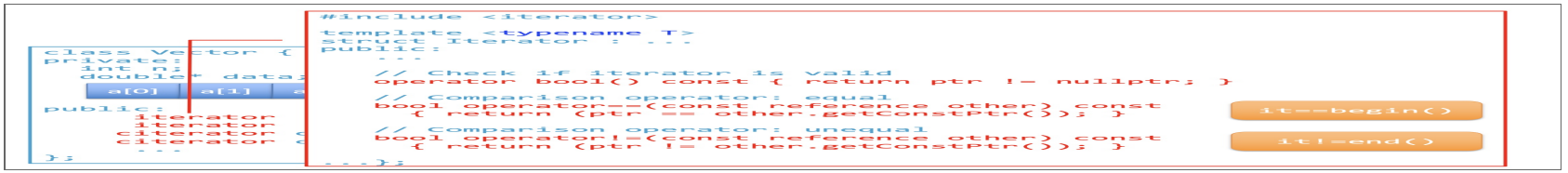

- Iterators are implemented as separate classes
- Constructor accepts pointer to the data over which to iterate and stores position internally
- More than one iterator object can be associated with an object of the Vector class
- All operations (it++, *it, it>, ...) must be implemented inside the iterator class
class Vector {
private:
int n;
double* data;
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
public:
iterator begin() { return iterator(data); }
iterator end() { return iterator(data+n); }
citerator cbegin() { return citerator(data); }
citerator cend() { return citerator(data+n); }
...
};

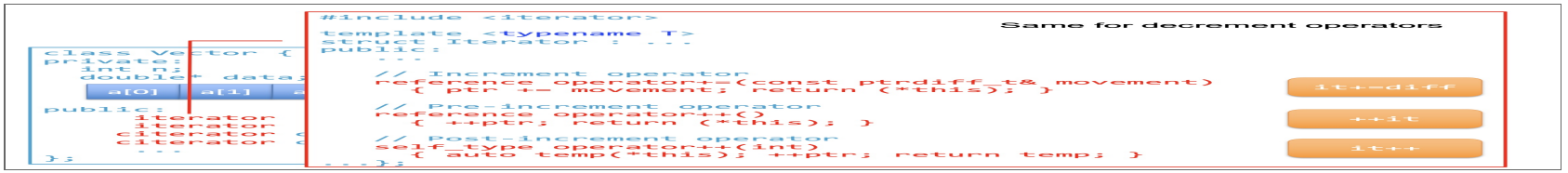

class Vector {
private:
int n;
double* data;
a[0]
a[1]
public:
iterator
iterator
citerator
citerator
...
};
#include <iterator>
template <typename T>
struct iterator : public
std::iterator<std::random_access_iterator_tag,
T, // type itself
ptrdiff_t, // type of distance between iterators
T*, // type of pointer
T& // type of reference
>
...};

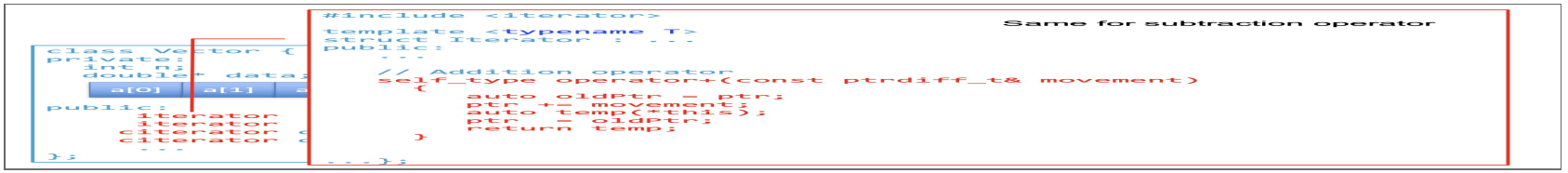

class Vector {
private:
int n;
double* data;
a[0]
a[1]
public:
iterator
iterator
iterator
iterator
...
};
#include <iterator>
template <typename T>
struct iterator : public
std::iterator<std::random_access_iterator_tag,
T, // type of itself
ptrdiff_t, // type of distance between iterators
T*, // type of pointer
T& // type of reference
>
typedef iterator
typedef std::random_access_iterator_tag
typedef ptrdiff_t
typedef T*
typedef T&
...};
self_type;
iterator_category;
difference_type;
value_type;
pointer;
reference;
11

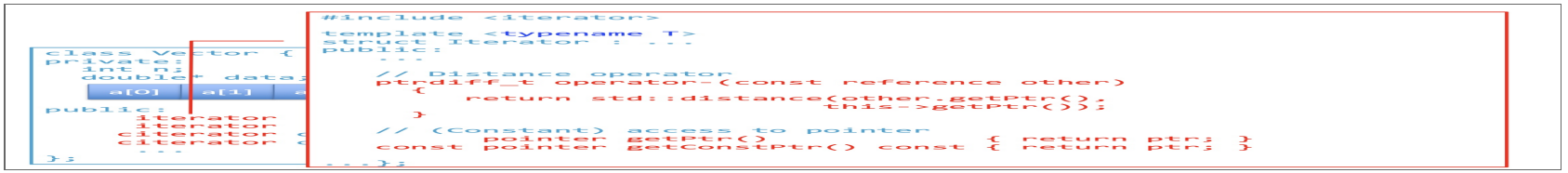

class Vector {
private:
int n;
double* data;
a[0]
a[1]
public:
iterator
iterator
iterator
iterator
};
#include <iterator>
template <typename T>
struct Iterator : .
protected:
pointer ptr; // internal storage of position
public:
// constructor and destructor
Iterator(pointer ptr = nullptr) : ptr(ptr) {}
~Iterator(){}
// (Constant) reference operator
reference operator*() { return *ptr; }
const reference operator*() const { return *ptr; }
// Pointer operator
pointer operator->() { return ptr; }
};
*it
it->
a[2]


class Vector {
private:
int ni
double* data;
a[0]
a[1]
public:
iterator
iterator
citerator
citerator
};
#include <iterator>
template <typename T>
struct Iterator : ...
public:
// Copy constructor
iterator(const reference other) = default;
// Copy assignment operator
reference operator=(const reference other) = default;
// Copy assignment operator from data pointer
reference operator=(pointer ptr){ ptr = ptr; return (*this); }
...};


class Vector {
private:
int n;
double* data;
a[0]
a[1]
public:
iterator
iterator
iterator
iterator
};
#include <iterator>
template <typename T>
struct iterator : ...
public:
...
// Check if iterator is valid
operator bool() const { return ptr != nullptr; }
// Comparison operator: equal
bool operator==(const reference other) const
{ return (ptr == other.getConstPtr()); }
// Comparison operator: unequal
bool operator!=(const reference other) const
{ return (ptr != other.getConstPtr()); }
...};
it==begin()
it!=end()


#include <iterator>
template <typename T>
struct Iterator : ...
public:
...
// Increment operator
reference operator+=(const ptrdiff_t& movement)
{ ptr += movement; return (*this); }
// Pre-increment operator
reference operator++()
{ ++ptr; return (*this); }
// Post-increment operator
self_type operator++(int)
{ auto temp(*this); ++ptr; return temp; }
...};
Same for decrement operators
it+=diff
++it
it++
class Vector {
private:
int n;
double* data;
a[0] a[1]
public:
iterator
iterator
const_iterator
const_iterator
...
};


class Vector {
private:
int n;
double* data;
a[0]
a[1]
public:
iterator
iterator
iterator
iterator
};
#include <iterator>
template <typename T>
struct Iterator : ...
public:
...
// Addition operator
self_type operator+(const ptrdiff_t& movement)
{
auto oldPtr = ptr;
ptr += movement;
auto temp(*this);
ptr = oldPtr;
return temp;
}
...};
Same for subtraction operator


class Vector {
private:
int n;
double* data;
a[0]
a[1]
public:
iterator
iterator
const_iterator
const_iterator
...
};
#include <iterator>
template <typename T>
struct Iterator : ...
public:
...
// Distance operator
ptrdiff_t operator-(const reference other)
{
return std::distance(other.getPtr(),
this->getPtr());
}
// (Constant) access to pointer
pointer getPtr() { return ptr; }
const pointer getConstPtr() const { return ptr; }
...};

Example: Sorting of Vectors

Create unsorted Vector object and sort it

In [ ]:

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<double> v = { 4, 2, 3, 7, 5, 8, 9, 1, 10, 6 };
std::sort(v.begin(), v.end());

std::cout << "v = [";
for (auto it = v.begin(); it != v.end(); it++) {
        std::cout << *it << (std::next(it, 1) != v.end() ? ", " : "]\n");
    }
```

Linear algebra library

Task: Let us design our own linear algebra library that supports the following operations:

- Element-wise Add, Sub, Div, Mul of vectors
- Add, Sub, Div, Mul of a vector with a scalar
- Copy assignment (other functionality can be added later)

Additional requirements:

- Support for arbitrary types: template metaprogramming
- Mathematical notation: operator overloading

Vector class: Attributes

For the whole program: **https://www.online-cpp.com/3DwpTrm4qB**

```cpp
template<typename T> class vector {
private:
     // Attributes
     T* data;
     std::size_t n;
```

Vector class: Constructors

```
template<typename T> class vector {
public:
     // Default constructor
     vector()
     : n(0), data(nullptr) {}
     // Size constructor
     vector(std::size_t n)
     : n(n), data(new T[n]) {}
...
```

Vector class: Constructors

```
template<typename T>
class vector {
    // Copy constructor
    vector(const vector<T>& other)
    : vector(other.size())
    {
      for (std::size_t i=0; i<other.size(); i++)
          data[i] = other.data[i];
    }
 ...
```

Vector class: Constructors

```
template<typename T>
class vector {
    // Move constructor
    vector(vector<T>&& other)
    {
      data = other.data;
      other.data = nullptr;
      n = other.n;
      other.n    = 0;
    }
    ...
```

Vector class: Destructor and helper functions

```
template<typename T>
class vector {
    // Destructor
    ~vector() { delete[] data; data=nullptr; n=0; }
    // Return size
    inline const std::size_t size() const { return n; }
```

Vector class: Helper functions

```
template<typename T>
class vector {
    // Get data pointer (by constant reference)
    inline const T& get(const std::size_t& i) const {
            return data[i];
    }
    // Get data pointer (by reference)
    inline T& get(const std::size_t& i) {
      return data[i];
    }
```

Vector class: Assignment operator

```
template<typename T>
class vector {
    // Scalar assignment operator
    vector<T>& operator=(const T& value) {
      for (std::size_t i = 0; i < size(); i++)
        data[i] = value;
      return *this;
}
```

Vector class: Assignment operator

```
template<typename T>
class vector {
    // Copy assignment operator
    const vector<T>& operator=(const vector<T>& other) const {
        if (this != &other) {
        delete[] data; n = other.size(); data = new T[n];
        for (std::size_t i = 0; i < other.size(); i++)
              data[i] = other.data[i];
  }
  return *this;
}
```

Vector class: Assignment operator

```
template<typename T>
class vector {
    // Move assignment operator
    vector<T>& operator=(vector<T>&& other) {
        if (this != &other) {
            std::swap(this->data, other.data);
            std::swap(this->n, other.n);
            other.n = 0; delete[] other.data; other.data = nullptr;
  }
  return *this;
}
```

Vector class: Binary vector-vector operators **(outside class!)**

```
template<typename T1, typename T2>
auto operator+(const vector<T1>& v1,
               const vector<T2>& v2) {
    vector<typename std::common_type<T1,T2>::type>
      v(v1.size());
    for (std::size_t i=0; i<v1.size(); i++)
      v.get[i] = v1.get[i] + v2.get[i];
    return v;
}
```

Similar for `operator-`, `operator*` and `operator/`

Vector class: Binary vector-scalar operators **(outside class!)**

```
template<typename T1, typename T2>
auto operator+(const vector<T1>& v1,
               const T2& s2) {
    vector<typename std::common_type<T1,T2>::type>
      v(v1.size());
    for (std::size_t i=0; i<v1.size(); i++)
      v.get[i] = v1.get[i] + s2;
    return v;
}
```

Similar for `operator-`, `operator*` and `operator/`

Quiz: Linear algebra library

What is the **theoretical complexity** (#memory transfers and #floating-point operations) of the expression `z=2*x+y/(x-3)`?

- Two loads (`x`, `y`) and one store (`z`) each of length `n`
- Four times `n` floating-point operations (`*`, `+`, `/`, `-`)

What is the **practical complexity** of the following code? (**https://www.online-cpp.com/CszH10vqGJ**)

```
int main() {
    Vector<double> x(10), y(10), z(10); x=1.0; y=2.0;
    z=2*x+y/(x-3);
}
```

- Five loads and three stores each of length `n`

Expression template library for linear algebra

- **Aim:** We want that the **expression** `z=2*x+y/(x-3)` is evaluated in a single for loop without temporaries.
- **Strategy:** Build up an **expression tree** that holds the steps to be performed to evaluate the expression but delay the actual evaluation until the assignment operator is used.
- The expression tree is a **symbolic** and **lightweight** representation of the expression and does not carry data.

Implement templated vector class (derived from base class `vectorBase`) as before but remove all operators (`+`, `-`, `*`, `/`)
Implement an additional **assignment operator**, which loops through the expression e and assigns the value to `data[i]`

```
template <typename E>
vector<T>& operator=(const E& e) {
      for (std::size_t i = 0; i < size(); i++)
          data[i] = e.get(i);
      return *this;
}
```

For each **binary operator** we need to implement a helper class that represents the operation in the expression tree

```
template<typename E1, typename E2>
class VectorAdd : vectorBase {
private:
    const E1& e1;
    const E2& e2;

public:
    VectorAdd(const E1& e1, const E2& e2)
    : e1(e1), e2(e2)
{}
...
```

For each **binary operator** we need to implement a helper class that represents the operation in the expression tree

```
template<typename E1, typename E2>
class VectorAdd : vectorBase {
...
    inline const
    typename std::common_type<typename E1::value_type,
                              typename E2::value_type>::type
    get(std::size_t index) const {
      return e1.get(index) + e2.get(index);
    }
};
```

For each **binary operator** we need to implement a helper class that represents the operation in the expression tree

```
template<typename E1, typename E2>
class VectorAdd : vectorBase { ... };
```

Base class `vectorBase` has no functionality and is used to check if a template argument belongs to the expression tree

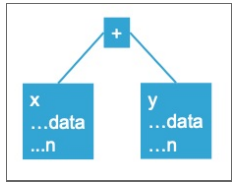
```
typename std::enable_if<
            std::is_base_of<vectorBase, E1>::value
              >::type
```

Implement the corresponding binary operator overload

```
template<typename E1, typename E2>
auto operator+(const E1& e1, const E2& e2) {
      return VectorAdd<E1,E2>(e1,e2);
  };
```

Now, expression `auto e=x+y` creates a new item in the expression tree and keeps constant references to `x` and `y`

Expression evaluation for a single index is triggered by `e.get(index)`



The expression auto `e=x+y+z` has the type
`VectorAdd&lt VectorAdd&lt Vector&ltX>, Vector&ltY> >, Vector&ltZ> >`
During the assignment of this expression to a vector object `vector<double> v=e` the `get(index)` function is called for each single index. The overall computation takes place in a **single for loop** as if we had written the following code

```
for (std::size_t i=0; i&ltx.size(); i++)
    v.get(i) = x.get(i) + y.get(i) + z.get(i);
```

A word of caution: Expression templates (ET) are very powerful (for you as a user) but writing an **efficient** ET linear algebra library takes much time and is not trivial
ET eliminate multiple for-loops but they do by no means imply parallelism, use of SIMD intrinsics, etcetera
A recent trend is to use the ET concept as high-level user- interface and implement all the dirty tricks (inlined assembler code, vector intrinsics, ...) in helper classes

Overview of linear algebra expression template libraries

Armadillo: MATLAB-like notation
ArrayFire: broad support on ARM/x86 CPUs and GPUs
Blaze: aims for ultimate performance
Eigen: easy usage and broad user community
IT++: grandfather of all ET LA libraries
MTL4: distributed HPC + fast linear solvers
VexCL: broad support on CPUs and GPUs
ViennaCL:.) fast linear solvers

Example: Eigen library

```
#include <Eigen/Dense>
using namespace Eigen;
int main() {
    MatrixXd A(3,3), B(3,3);
    VectorXd x(3), y(3), z(3);
    A << 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0;
    B << 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0;
    x <<-1.0,-2.0,-3.0;
    y << 4.0, 5.0, 6.0;
    z = A.transpose() * x.array().abs().sqrt().matrix();
    std::cout "z = A^T * sqrt(|x|)\n" << z << "\n\n";
  }
```

Smart Pointers

Dynamically allocated memory must be deallocated explicitly by the user to prevent memory leaks

https://www.online-cpp.com/ueBnpEWSU9

```cpp
{

    // Traditional dynamic memory allocation with raw pointer
    double* raw_ptr = new double[42];

    // Perform operations on raw_ptr
    for (int i = 0; i < 42; ++i)
        raw_ptr[i] = i;

    // Explicitly deallocating memory
    delete[] raw_ptr;

} // raw_ptr goes out of scope but is not destroyed automatically
```

```cpp
#include <memory>

{
    std::unique_ptr<double> scoped_ptr(new double[42]);

    // Perform operations on the scoped_ptr
    for (int i = 0; i < 42; ++i)
        scoped_ptr[i] = i;

} // scoped_ptr goes out of scope and is destroyed automatically
```

Smart Pointers – use-case

```
double* d = new double[42];
try {
  foo(d, 42); // do something that may
fail
} catch (std::exception e){
  std::cerr << e.what() << std::endl;
  return -1; // forgot to delete d!
}
if (d[0]>100) return +1; // again...
delete[] d;
return 0;
```

```
std::unique_ptr<double>
  d(new double[42]);
try {
  foo(d, 42); // do something that may
fail
} catch (std::exception e){
  std::cerr << e.what() << std::endl;
  return -1; // d goes out of scope
}
if (d[0]>00) return +1; // also fine
return 0;
```

Smart Pointers

The `unique_ptr` owns and manages its data exclusively, i.e. the data can be owned by only one pointer at a time

```
std::unique_ptr<double> data(new double[42]);
std::unique_ptr<double> other(data.release());
```

One can easily check if managed data is associated

```
if (data)
    std::cout << "Data is associated\n";
else
    std::cout << "No data is associated\n";
```

Smart Pointers

Managed data can be updated

```
data.reset(new double[42]);
```

Managed data can be swapped

```
data.swap(other);
```

A class with a `unique_ptr` member is not default copiable
A `unique_ptr<T>[]` argument may allow more aggressive compiler optimizatiohn in a function than a raw pointer `T*`

Smart Pointers with reference counting

The `shared_ptr` shares ownership of the managed content with other smart pointers

```
std::shared_ptr<double> data(new double[42]);
std::shared_ptr<double> other(data);
```

Content is deallocated when last pointer goes out of scope

```
std::cout << data.use_count(); // -> 2
```

Example: multiple data sets 'living' on the same mesh

```
class DataSet {
      std::shared_ptr<Mesh> mesh;
  };
```

Bundle references with `std::tie`

Example: use tuple comparison for own structure

```
struct S {
      int n;
      std::string s;
      float d;
      bool operator<(const S& rhs) const {
        // true if any comparison yields true
           return   std::tie(n, s, d) <
                    std::tie(rhs.n, rhs.s, rhs.d);
    }
```

Bundle multiple return values with `std::tie`

Inserting an object into `std::set` with C++11/14

```cpp
std::set<S> Set;
S value{42, "Test", 3.14};
std::set<S>::iterator iter;
bool inserted;
// unpacks return val of insert
std::tie(iter, inserted) = Set.insert(value);
```

Structured bindings (C++17)

Inserting an object into `std::set` with C++17

```cpp
std::set<S> Set;
S value{42, "Test", 3.14};

// structured bindings
auto [iter, inserted] = Set.insert(value);
```

Structured bindings (C++17)

Retrieving individual elements of an array

```
double Array[3] = { 1.0, 2.0, 3.0 };
auto [ a, b, c ] = Array;
```

Retrieving individual elements of a `std::map`

```
std::map Map;
for (const auto & [k,v] : Map)
{
    // k – key
    // v – value
}
```

Constexpr (C++11)

C++11 introduced the constexpr specifier (constant expression), which declares that the value of the function or variable can be evaluated at compile time.

```
constexpr int factorial(int n)
{ return n <= 1 ? 1 : (n * factorial(n - 1)); }
```

Such functions or variables can then be used where only compile-time expressions are allowed

```
Vector<factorial(6)> v;
```

Constexpr (C++14)

C++14 further allows `constexpr` functions to have local variables, which is not allowed in C++11

```
constexpr int factorial(int n)
{
      int val = (n <= 1 ? 1 : (n * factorial(n - 1)));
      return val;
}
```

If Constexpr (C++17)

C++17 introduces if constexpr, which allows to discard branches of an if statement at compile-time based on a `constexpr` condition (no more SFINAE and specialization?).

```
template<int N>
constexpr int fibonacci()
{
   if constexpr (N>=2)
      return fibonacci<N-1>() + fibonacci<N-2>();
   else
      return N;
}
```

Template argument deduction

C++11/14: Template arguments can be deduced automatically by the compiler for functions only

```
template <typename A, typename B>
auto sum (A a, B b)
    -> typename std::common_type<A,B>::type
{ return a + b; }

auto result1 = sum<double, float>(1.0, 2.0f);
auto result2 = sum(1.0, 2.0f);
```

Template argument deduction

C++11/14: Template argument deduction for classes and structures requires the use of auxiliary maker functions

```
auto t = std::make_tuple("Hello", 42);
auto p = std::make_pair ("Hello", 42);
```

C++17 enables automatic template argument deduction directly for classes and structures

```
auto t = std::tuple("Hello", 42);
auto p = std::pair("Hello", 42);
```

Fold expressions (C++17)

C++11 introduced variadic templates, which often require recursive calls and an overloaded "termination condition"

```
template<typename T>
auto static sum(T arg)
{ return a; }

template<typename T, typename... Ts>
auto sum(T arg, Ts... args)
{ return arg + sum(args...); }
```

Fold expressions (C++17)

C++17 introduces fold expressions, which reduce (=fold) a parameter pack over a binary operator

```
template<typename... Ts>
auto sum(Ts... args)
{ return (args + ...); }
```

Fold expressions can be used in more complex expressions

```
template<typename... Ts>
auto expression(Ts... args)
{ return (3 * args + 1 + ... + 10); }
```

Fold expressions (C++17)

Associativity of fold expressions

```
return (args + ...); // arg0 + (arg1 + arg2)
return (... + args);  // (arg0 + arg1) + arg2
```

An example where associativity matters

```
return (args - ...); // arg0 - (arg1 - arg2)
return (... - args); // (arg0 - arg1) - arg2
```

Fold expressions (C++17)

Incorrect treatment of empty parameter packs

```
template<typename... Ts>
auto sum(Ts... args)
  {
    return (args + ...);
  }
```

What happens for `sum()`?
-> error: fold of empty expansion over sum

Fold expressions (C++17)

Correct treatment of empty parameter packs

```
template<typename... Ts>
auto sum(Ts... args)
  {
    return (0 + ... + args);
  }
```

Note that the 0 must be inside of the expression, i.e. **not**

```
return 0 + (... + args);
```

Fold expressions (C++17)

Print an arbitrary number of arguments

```
template<typename ...Args>
void FoldPrint(Args&&... args)
{ (std::cout <<  ... <<  args) << std::endl; }
```

Fold over a comma operator

```
template<typename T, typename... Args>
void push_back_vec(std::vector<T>& v, Args&&... args)
{ (v.push_back(args), ...); }
```

Concepts (C++20)

Recall homework assignment

```
template<typename m1, typename m2>
struct Measure_add
  {
     static const int value = m1::value+m2::value;
     static const Unit unit = m1::unit;
  };
```

We assume that `m1` and `m2` provide a `value` and `unit` attribute. If another type without these attributes is passed the compiler throws an error.

Concepts (C++20)

Manual workaround

```
template<typename m1, typename m2>
struct Measure_add; // leave unimplemented

template<int v1, Unit u1, int v2, Unit u2>
struct Measure_add<Measure<v1, u1>, Measure<v2,u2>>
  {
     static const int value = v1+v2;
     static const Unit unit = u1;
  };
```

Concepts (C++20)

```cpp
#include <concepts>

template<typename T>
concept Measurable = requires(){T::value; T::unit;};

template<Measurable m1, Measurable m2>
struct Measure_add{...};
```

Happy Holidays!

Please bring any questions you have about the projects to the office hour after Christmas!

In [8]:

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

int treeheight = 10;
int height = 3;
for (int i = 1; i <= treeheight; ++i) {
        // Print spaces
        std::cout << std::string(treeheight - i, ' ');
        // Print asterisks for the tree
        std::cout << std::string(2 * i - 1, '*');
        std::cout << std::endl;
}
for (int i = 1; i <= height; ++i) {
        // Print spaces
        std::cout << std::string(treeheight/2+2, ' ');
        // Print asterisks for the tree
        std::cout << std::string(treeheight/2, '*');
        std::cout << std::endl;
}
std::cout << "   Happy Holidays!" << std::endl;
```

```
            *
           ***
          *****
         *******
        *********
       ***********
      *************
     ***************
    *****************
   *******************
          *****
          *****
          *****
     Happy Holidays!
```

In [ ]:

```

```