DUNE PDELab Tutorial C++ Refresher

Olaf Ippisch

September 3, 2021

Goals of this introduction
A number of advanced C++ features is required to use DUNE or makes its usage much easier. Some of them were introduced in the last C++ language-standards C++11, C++14 and C++17. We will specifically look at:

- Compiling Programs on Linux-based systems
- Object-Oriented Programming
- Namespaces
- Automatic type deduction
- Templates
- Containers of the C++ standard library
- Range-based for loops
- Lambda functions

1 Compiling Programs on Linux

On Linux-based systems, the (GNU-)C++ compiler is called `g++`. In order to compile a program from the source file `main.cc` and create an executable called `myprogram` you run it like this:

```
g++ -o myprogram main.cc
```

With `g++` the C++14 standard is the default since version 6.1. If you want to use C++17 or if you use the Intel compiler or `clang`, you can set the standard with the option `-std=c++14` or `-std=c++17` (on Windows systems the options may be different), e.g.

```
g++ -std=c++17 -o myprogram main.cc
```

If your program was compiled successfully (without error messages), you run it with:

```
./myprogram
```

and get for example the output

```
Hello, world!
```

For more complex software projects usually so-called makefiles are provided. A makefile ensures that all relevant source files for the project are compiled, when the command `make` is executed:

```
make
```

If you build the project after changing only some source files, only these files are recompiled, which can speed up the process considerably.

For DUNE the makefiles are generated by a tool called `cmake`.

2 Object-Oriented Programming

The basic idea of object-oriented programming is to define components with a certain functionality. These components combine both the methods which are needed to provide and control the functionality as well as the relevant data. Different components are connected via interfaces. The same interface is used for specialised components providing the same general functionality.

This approach has several advantages:

- Different components can be developed separately.
- If improved versions of a component become available, they can be used without major changes to the program code.
- Several realisations of the same component can be used easily.

This is also used in the real world, e.g. the tires of a car, the socket of a CPU, the USB interface of computers, printers, mobile phones ...

Object-oriented Programming in C++

In C++ this idea is realised by classes and objects. Classes describe the functionality and structure of a component. Realisations of this blueprint are called objects.

Let us use a two-dimensional vector as example:

- The vector consists of two components.
- The necessary operations are length (norm), sum of two vectors, multiplication of a vector with a scalar,....
- Initialization: Create a vector with a defined state.

$(1, 3)$ and $(5, 0)$ are different vectors, but they share the same structure.

$\Rightarrow$ **C++ language concept**
`class` describes semantics of similar objects (vectors, function spaces, linear solvers,...).

Classes

```
class Point2D {
public:
  Point2D(double x_, double y_)
    : x(x_), y(y_)
  {}

  double norm() const {
    return sqrt(x*x + y*y);
  }

  void add(const Point2D &p) {
    x += p.x;
    y += p.y;
  }

private:
  double x, y;
};
```

Using C++ Objects

Classes define a new data type.

- Variables of **`class`** type are called objects and can be used like other types (**`int`**, **`double`**, ...):

  ```
  // calls constructor, initialises data
  Point2D p = Point2D(3.0,4.1);
  ```

- The member variables and functions of an *object* are accessed over the variable name followed by a dot and the name of the variable/function:

  ```
  std::cout << p.norm() << std::endl;
  ```

- Objects of a **`class`** can be used like ordinary variables:

  ```
  Point2D p2 = p; // create a copy of p
  p.add(p2);
  ```

Encapsulation

- Data members and methods of a **`class`** can be either **`public`** (accessible from outside the **`class`**) or **`private`** (accessible only from methods of the same **`class`**).
- If you try to access a **`private`** member from outside the class, you get a compiler error:

  ```
  std::cout << p.x << std::endl; // compiler error!
  ```

- It is advisable to make the data members of a `class` private. This is called *encapsulation* and facilitates a later change of the way the data is stored, which can often greatly improve performance.

3 Namespaces

- Let us assume you want to use two existing libraries with the header files `linear_solver.h` and `nonlinear_solver.h`.
- Both define a function `solve_problem()`.
- The compiler cannot distinguish between them!
- How can you use both libraries without name conflicts?

C++ uses namespaces, which allows an easy resolution of this problem:

```
namespace linsolv {
  #include<linear_solver.h>
}
namespace nonlinsolv {
  #include<nonlinear_solver.h>
}
int main() {
  linsolv::solve_problem();
  nonlinsolv::solve_problem();
}
```

Each `class` automatically defines a namespace with the name of the `class` which contains its data members and functions.

Builtin Namespace `std`

- C++ also contains a library of useful functions and classes the so called standard library.
- Beginners often write

  ```
  using namespace std;
  ```

  at the top of their program, which imports all functions of the standard library into the global namespace.
- You should *NEVER* do this as it can lead to name conflicts and reduces flexibility in further development!

Just write the namespace `std::` in front of the functions and types instead:

```
#include <iostream>
#include <cmath>

int main()
{
  double v = 2.0;
  double s = sqrt(v);
  std::cout << "The␣square␣root␣of␣" << v
            << "␣is␣" << s << std::endl;
}
```

If you use the auto-completion features of modern editors, this is not much typing overhead and much less work than rewriting your code later.

4 Automatic Type Deduction

Since C++11 the type needed for a variable can be detected automatically if it can be deduced from the initialization of the variable. This is indicated with the keyword **`auto`**.

```
#include <vector>

int f() {
  return 1;
}

int main()
{
  auto var1 = 5678;    // var1 has type int
  auto var2 = 'x';     // var2 has type char
  auto var3 = f();     // var3 has return type of f() i.e. int
  auto vector = std::vector<double>(5);
}
```

`auto`: Advantages

- Using **`auto`** instead of manually writing the exact type (e.g. **`int`**) has a lot of advantages (at the expense of a bit of detailed control):
  - If you create a variable with **`auto`**, it will always be initialized with an defined value, which avoids a whole class of very subtle bugs.
  - If you write **`auto`**, the compiler can choose the right type, reducing the risk of errors.
  - When using advanced libraries like DUNE, the types of variables can become very complicated, making it difficult to read code that spells out those types.
- Only since C++14 it is possible to use **`auto`** in function definitions and declarations.

auto and literals

For type deduction with **auto** the usual rules for C++ literals apply:

```
#include<string>

int main()
{
  // upper case letters can be used for suffices as well
  auto var1 = 5.0;    // double
  auto var2 = 5.0f;   // float
  auto var3 = 5.0l;   // long double
  auto var4 = 5;      // int
  auto var5 = 5l;     // long
  auto var6 = 5u;     // unsigned int
  auto var7 = 5ul;    // unsigned long int
  auto var8 = 'a';    // char
  auto var9 = "a";    // const char *
  auto varA = std::string("a"); // string object
  auto varB = new int;  // pointer to int
}
```

References

- If variables are passed to a function in C++, by default a copy of the variable is generated.
- References are an alternative, which only generates a new name for the same data.
- References are generated by adding a `&` between the data type and the variable name, e.g. `int &blub=blob` or `int f(int &a)`.
- You can only generate references to existing variables not to literals!
- If you want to make a variable a reference with automatic type deduction write `auto &` instead of `auto`:

```
/* Copying */
auto i = 4;
auto j = i;  // i = 4, j = 4, j is a copy
i += 1;      // i = 5, j = 4
j -= 1;      // i = 5, j = 3
j  = i;      // i = 5, j = 5

/* Referencing */
auto &k = i; // i = 5, k = 5, k is a reference to i
i  = 2;      // i = 2, j = 5, k = 2
i += 1;      // i = 3, j = 5, k = 3
k  = 8;      // i = 8, j = 5, k = 8
```

5 Templates

- Often the same algorithms are needed for different data types.
- Without generic programming one has to write the same function for all data types, which is tedious and error-prone, e.g.

```
int Square(int x)
{
  return(x*x);
}
```

```
long Square(long x)
{
  return(x*x);
}
```

```
float Square(float x)
{
  return(x*x);
}
```

```
double Square(double x)
{
  return(x*x);
}
```

- Generic programming allows to write the algorithm once and parametrise it with the data type.

Template functions

- A function template starts with the keyword `template` and a list of one or more template arguments in angle brackets separated by commas:

```
template<typename T>
T Square(T a)
{
    return(a*a);
}
```

- If a template is used, the compiler can automatically generate the function from the function template according to the function arguments (as with overloading the return type is not relevant).
- The template arguments can also be specified explicitly:

```
std::cout << Square<int>(4) << std::endl;
```

- The argument types must fit the declaration

Example: Unary Template Function

```
#include<cmath>
#include<iostream>

template<typename T>
T Square(T a)
```

```
{
    return(a*a);
}

int main()
{
    std::cout << Square<int>(4) << std::endl;
    std::cout << Square<double>(M_PI) << std::endl;
    std::cout << Square(3.14) << std::endl;
}
```

Example: Binary Template Function

```
#include<cmath>
#include<iostream>

template<class U>
const U &max(const U &a, const U &b) {
 if (a>b)
     return(a);
 else
     return(b);
}

int main()
{
 std::cout << max(1,4) << std::endl;
 std::cout << max(3.14,7.) << std::endl;
 std::cout << max(6.1,4) << std::endl; // compiler error
 std::cout << max<double>(6.1,4) << std::endl; // correct
 std::cout << max<int>(6.1,4) << std::endl;    // warning
}
```

Useful predefined template functions

The C++ standard library already provides some useful template functions:

- `const T& std::min(const T& a, const T& b)` minimum of a and b

  ```
  auto c = std::min(a,b);
  ```

- `const T& std::max(const T& a, const T& b)` maximum of a and b

  ```
  auto c = std::max(a,b);
  ```

- `void std::swap(T& a, T& b)` swap a and b

  ```
  std::swap(a,b);
  ```

Class Templates, Non-type Template arguments, default arguments

```
template<typename T, int dimension = 3>
class NumericalSolver
{
    ...
  protected:
    T variable;
};
```

- Template arguments can be used in class declarations.
- Not only types, but also integer values can be used as template arguments. The values used in the template instantiation have to be compile time constants.
- If templates are used in a class definition, the last template arguments can have default values.
- The name of a class is the class name plus the template parameters

Inheritance from Class Templates

```
template<typename T>
class MyNumericalSolver : public NumericalSolver<T,3>
{
    T myVariable;
  public:
    MyNumericalSolver(T val) : NumericalSolver<T,3>(),
                               myVariable(val)
    {
      std::cout << NumericalSolver<T,3>::variable
                << std::endl;
    };
};
```

- If a class is derived from a template class, the template arguments have to be given as part of the base class name.
- the same is true for the call of base class constructors and the prefixing of base class members and methods.

Using Members of a Template Base Class

```
template<typename T>
class MyNumericalSolver : public NumericalSolver<T,3>
{
    T myVariable;
  public:
    MyNumericalSolver(T val) : NumericalSolver<T,3>(),
                                 myVariable(val)
    {
        this->variable=val;
    };
};
```

- The members of a template base class are often not automatically resolved correctly
- To avoid problems, it is (as a rule of thumb) helpful to always prefix base class members (methods as well as variables) with **`this->`**

Template Compilation

- If templates are not used and therefore are not instantiated, the template code is only checked for crude syntax errors (e.g. missing semicolons) by the compiler.
- The test, if all function calls are valid, is conducted when a template is instantiated. Errors like missing functions are only detected then. The error messages can be rather strange.
- As the code is only created at the template instantiation, the compiler has to know at this time the whole function definition not only its declaration.
- The usual subdivision into header and source files is therefore not possible for templates.
- To save computation time and memory, only class functions which are really called are generated.
- Thus class templates can also be instantiated for types which do not support all necessary operations as long as the methods where they are needed are never called.
- If template classes have long argument lists, typedefs are helpful:

  ```
  typedef Point2D P;
  auto coord = P(8,1); // Now P means the same as Point2D
  ```

Template Aliases

```
template <typname U>    // create partially defined templates
using VectorSpace = Dune::PDELab::Backend::Vector<GFS,U>::type;

int main()
{
  using int32 = int;                  // rename ordinary types
  using Vector =
        typename Dune::PDELab::Backend::Vector<GFS,double>::type;
  auto v = Vector(gfs); // Save lots of typing...
```

```
  using Function = void (*)(double);  // function types
  VectorSpace<float> blub;
}
```

- An alternative method to define abbreviations for long type names is called "template aliasing".
- Also partial template aliasing is possible fixing some of the template arguments.

Keyword `typename`

```
template<typename T, int dimension = 3>
class NumericalSolver : public T::ClassType
{
    ...
    typename T::SubType doSomething(
                              typename T::OtherSubType argument);
  private:
    typename T::SubType variable;
}
```

- In C++ by default a member of a class template is assumed not to be a type but a (static) variable.
- The **`typename`** keyword is needed to indicate that the member of a class (which is given as or depends on a template parameter) is a type.
- It is only needed/allowed inside a template.
- It is not used in a list of base class specifications or in a list of member initializers in a constructor definition

Keyword `.template`

```
class A
{
  public:
    template<class T> T doSomething() { };
};

template<class U> void doSomethingElse(U variable)
{
    char result = variable.template doSomething<char>();
}

template<class U,typename V> V doSomethingMore(U *variable)
{
    return variable->template doSomething<V>();
}
```

- C++ assumes by default that every `<` character following an object is the start of a comparison.
- The keyword **`template`** in front of such a method name indicates that an explicit template parameter follows.

6 The C++ Standard Library

The standard library (sometimes called STL for Standard Template Library) is

- a collection of useful template functions and classes.
- available for all modern C++ compilers.
- optimised for efficiency.
- a lot safer than using plain C libraries and data structures.

STL-Containers

- Data representation is often crucial for the efficiency of algorithms
- The STL defines containers and algorithms to use them.
- Containers are used to manage a collection of elements
- Iterators provide a common interface to traverse the elements of a container
- There is a wide variety of containers optimised for different purposes:

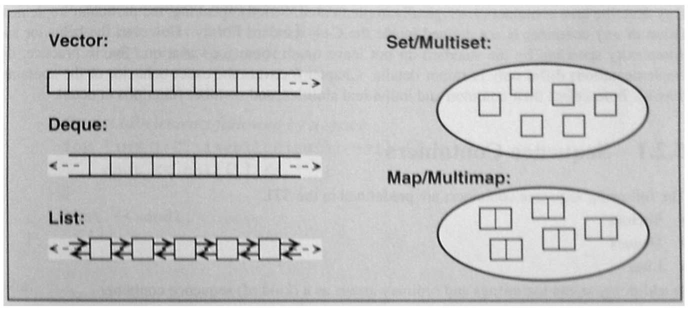

figure from: [?]

Library Example: dynamic array

The STL contains a dynamic array called `std::vector` which is a lot better than its plain C counterpart:

- It automatically frees the memory when you don't need it anymore, avoiding memory leaks.

- It knows about its size, so you don't have to remember it.
- It can automatically resize itself if you need a larger vector.
- It is a template: You specify the type of object to store inside of it.
- It is fast.

Vector Example

The following example shows how to use a `std::vector`:

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

int main()
{
    auto b = std::vector<double>(7); // a vector for 7 doubles
    std::cout << b.size() << std::endl; // output the size
    for (int i = 0; i < b.size() ; ++i)
        b[i] = i*0.1; // assign some data
    auto c = b; // create a copy, automatically copying all data
    c.resize(15); // Make c bigger
    std::cout << b.size() << std::endl; // still 7
    b.push_back(3.8); // make b larger and append the value 3.8
}
```

Iterators

- An essential part of the Container concept is a generalised scheme to iterate over all elements stored in a container, which is independent of the container type using so-called iterators.
- The syntax of iterator usage is derived from the usage of pointers in ordinary C.
- Each container has a method `begin()`, which returns an iterator pointing to the first member of the container.
- The iterator has the data type `containerclass::iterator` (but you don't need to know this if you use auto).
- If you call the increment operator `++` of the iterator, it will afterwards point to the next element.
- You can check if you reached the end of the container by comparing the iterator to `container.end()`.
- To access the element to which the iterator points, you have to dereference it, e.g. `*it`

- If the element of the container is an object of a class, you can also use the operator `it->` to access data members or functions of the class.

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

int main(int argc, char** argv)
{
  std::vector<int> vec;
  for (int i=0; i<10 ;++i)
    vec.push_back(i);

  for (auto it=vec.begin(); it!=vec.end(); ++it)
    std::cout << *it << "␣";
  std::cout << std::endl;
}
```

7 Range-based for Loops

With range-based **for** loops the same operation (iterating over a whole container) can be written much simpler:

```
auto vec = std::vector<double>(8);
for (auto d : vec)
  std::cout << d << std::endl; // prints all entries of vec
```

- Works for C arrays and all STL containers.
- Also important for using DUNE!
- Careful: what you get in a range-based **for**-loop is not an iterator. You don't have to dereference it. You can directly use it to access the content.

You can get either a copy or a reference (if you write **auto &**) of the element currently accessed.

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

int main()
{
  std::vector<double> x(5);
  int i=0;
  for (auto &y : x) {       // with reference
    y = i * 1.2;            // can be changed
    ++i;
  }
  for (auto y : x) {        // with copy
```

```
        y *= y;                  // original container unchanged
        std::cout << y << std::endl;
      }
    }
```

8 Type Deduction for Function Argumetns

Decltype

- `decltype` determines the result type of an expression (not the result value). It can be used to make matching variables.

  ```
  int a,b;
  decltype(a+b) c;
  ```

- Together with template aliasing it can be used to store types: `using type = decltype(expression)`
- Difference to `auto`: Does not create a variable of the same type but stores the type, also preserves references.
- Good if you want to store the result of a function call in a container.

```
template<typename Vector>
auto squareroot_values(const Vector& v)
{
  using R = decltype(sqrt(v[0])); // result type of std::sqrt()
  auto result = std::vector<R>();
  for (auto d : v)           // create a vector of square roots
    result.push_back(std::sqrt(d));
  return result;
}
```

Historic: `decltype` and return values in C++11

- In C++11 `auto` could not be used directly for function definitions or declarations. This is only possible since C++14.
- The return type of a function could be determined depending on the type of the function arguments with `decltype`
- `decltype` determines the type of a given operation.
- In the easiest version, `auto` is used as return type and `-> decltype(op)` is added after the argument list of the function, where `op` is a given operation:

```
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u);
```

9 Lambda Functions

- C++11 introduced a simplified possibility to create (mostly temporary) functions, so-called lambda expressions or lambdas.
- Example:

```
auto f = [](auto x) -> double
{
  double y = x;
  return std::sin(y);
};
```

- There is an even shorter version for single expressions (return type auto-deduced):

```
auto f = [](auto x) { return std::sin(x); };
```

- Lambdas can be stored like variables, the type of lambda functions is implementation-defined. Thus `auto` variables have to be used.

Lambda functions: Syntax

```
[capture-list](parameter-list) -> return-type { code; }
```

- Lambdas do not have a function name.
- They start with a capture list, determining which variables from the surrounding scope are available inside the lambda
- This is followed by a ordinary parameter list and a return (specified similar to `decltype`) and finally the code block of the function.
- Two types of capture: by value (creates a copy) or by reference (points to the original variable)
- Usage: List variable names, add ampersand (`&`) for references
- Shortcuts: `[=]` capture all variables by value, `[&]` capture all variables by reference
- Since C++17 it is also possible to capture a variable as constant reference: `[&v=std::as_const(v)]`
- Caution: When using capture by reference, make sure the original variable still exists when calling the lambda function!

Lambda functions: Capture Example

```
auto pi = 3.14;
auto f = [pi](auto x) { return pi * x; };
auto g = [&pi](auto x) { return pi * x; };
f(1); // returns 3.14;
g(1); // returns 3.14;
pi = 3.141;
f(1); // returns 3.14 - value was copied!
g(1); // returns 3.141 - value was referenced!

int call_count = 0;
auto h = [&](auto x)
{
  ++call_count;
  return pi + x;
}
h(1); // returns 4.141
std::cout << call_count << std::endl; // prints 1
```

Accumulation using `for_each`: lambda function

Lambda functions are very useful for template algorithms as e.g. defined by the standard template library. The `for_each` algorithm applies a function object to each of the elements of a container.

```
auto coll = std::vector<double>();
// fill coll with data...
int count = 0;
double sum = 0;
std::for_each(coll.begin(),coll.end(),
              [&](auto x){ ++count; sum += x; } );
std::cout << sum / count << std::endl;
```

10 References

References

[Stroustrup (2014)]	Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Addison Wesley, 2014.
[Stroustrup(2013)]	Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2013.
[cppreference.comm]	The cppreference community. *The community C++ reference web site.* *`http://cppreference.com`*.