EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Applied Bioinformatics

Programming in C/C++

- Templates -

Class Templates

- Repetition: function templates
- Polymorphism: Inheritance vs. Templates
- Family of classes with class templates
- Template specializations

Generic programming

- **Generic programming** is writing code that is type independent

- **Function templates** are functions that can operate on **generic types T**

- Declaration of a function template with **template parameter** type **T:**

```
template <class T> function_declaration;
template <typename T> function_declaration; // same
```

- We already saw examples for class templates in the STL: container, like `vector<T>`

- How does this magic happen? How can we implement this?
- Are there other applications of templates in classes?

Class templates

- Class templates can save us from a lot of code duplication
  (imagine how much code you would need to implement `vector` for all primitive or STL types…)

- Another way to look at templates: Inheritance vs. Templates
  - Inheritance in object-oriented programming allows **polymorphism at runtime** with virtual functions

  - Templates allow **polymorphism at compile time**
    - From generic code, specialized code is generated (as needed for execution)
    - Implementation at compile time, therefore efficient at runtime
    - Templates can be used for functions and classes

  - What is better? Depends…
    - Templates do not create a class hierarchy
      (which is often considered good as it reduces the strong dependency to the parent)
    - But templates impose compile time overhead
    - and are more complex to write.

Templates for Member Functions

- Within a class, methods can be declared as function templates
- The definition must also be marked as a template

```
class Dummy {
public:
    // declaration
    template< class T > void print(const T &);
};

// definition - implementation
template< class T >
void Dummy::print(const T &_obj) {
    std::cout << "dummy: " << _obj << std::endl;
}
```

Class Templates

- Class templates declare **a family of classes.**
- Without template parameters specified they are an **incomplete type**.

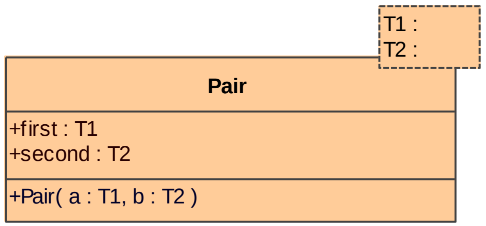
- **Template parameters** can affect attributes, method parameters, return values, …

- Declaration of a template with template parameters, for the type **T** to be used

```
template <class T> class_declaration;
template <typename T> class_declaration; // same
```

- Example (UML and declaration): store a pair of variables with generic types

```
template <class T1, class T2>
class Pair
{
  public:
  Pair(T1 a, T2 b);
  T1 first;
  T2 second;
};
```



Class Template – Example

A Class template declares a family of classes

```
// declaration of class template
template <class T>
class Point2D {
  T d_x, d_y;
public:
  Point2D();
  Point2D( T _x, T _y );
  Point2D<T> add( const Point2D<T>& _oPoint ) const;
  …
};

// usage
Point2D<int> intPt1, intPt2, intPt3;
```

return Type

Class Template – Example

A Class template declares a family of classes

```
// declaration of templated class
template <class T>
class Point2D {
  T d_x, d_y;
public:
  Point2D();
  Point2D( T _x, T _y );
  Point2D<T> add( const Point2D<T>& _oPoint ) const;
  …
};

// usage
Point2D<int> intPt1, intPt2, intPt3;
intPt3 = intPt1.add( intPt2 );
Point2D<double> dPt1, dPt2, dPt3;
dPt3 = dPt1.add( dPt2 );
```

Generic Types

- Classes or methods instantiated from the same template but with **different template parameters** have **different type**

```
Point2D<int> intPt1, intPt2, intPt3;
intPt3 = intPt1.add( intPt2 ); // ok, adding two Point2D<int>

Point2D<double> dPt1, dPt2, dPt3;
dPt3 = dPt1.add( dPt2 );      // ok, adding two Point2D<double>
dPt3 = dPt1.add( intPt2 );    // error: wrong type.
                              //        Can't call add with Point2D<int>
```

Definition of Class Templates

- The **definition** must specify that it is a **template** and which **template parameters** belong to the class

```
template <class T> class Point2D {
  // …
};

// Definitions:
template <class T> Point2D<T>::Point2D(T _x, T _y) {
  d_x = _x;
  d_y = _y;
}

template <class T> Point2D<T> Point2D<T>::add(const Point2D<T> &_op) const {
  Point2D<T> res;
  for (int i = 0; i < 2; i++) {
    res.d_components[i] = d_components[i] + _op.d_components[i];
  }
  return res;
}
```

template parameter right after class name

Template default parameter

- Template parameter can have defaults (similar to default arguments in non-template functions)

```
template <typename T1, typename T2 = T1>
class MyPair
{
  public:
  T1 first;
  T2 second;
};
int main()
{
  MyPair<int> iip;         // same as MyPair<int, int>
  MyPair<int, short> isp; // same as MyPair<int, short>
}
```

Template parameter

- Template parameter don't need to be a type, they **can also be a value**:

```
template <typename T, int d = 3>
    class VectorD
    {
      public:
      VectorD() : x_(d, 0) { } // create zero-initialized 3-dimensional vector
      T& operator [] (int i) { return x_[i]; }
      protected:
      vector<T> x_;
    };
    …
    VectorD<double, 4> v;
    v[3] = 7.0;
```

Compiling of Template Code

What happens if we do the usual organization of files?

- Declaration of class template in header file (**`Point2D.hpp`**)
- Definition of the class template in source file (**`Point2D.cpp`**)
- Instantiation in source file that uses the template class (**`example.cpp`**)
  - **`#include "Point2D.hpp"`**.

Problem:

- Compiler also needs **definition** of class template to **generate** code during template instantiation.

Usual separation into header and source file does not work for templates!

A Solution

- Template code with **declaration and definition** in single header file

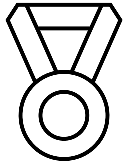
```
#pragma once

template <class T>
class Point2D {
    …
};

// Definition of template class methods in .hpp
template <class T>
class Point2D<T>::Point2D() { };
```

- Problems:
  - Template code may be included from several files
  - Potentially many instantiation
  - Compiler / Linker must resolve this during build
  - Leads to slower build times in C++

Debugging Template Code

- Complex errors
  - Hard to reason
- Compiling the template definition
  - Syntax problems
- Compiling the code that uses the template
  - Number and type of template parameters
  - Template expects other type
- Linking
  - Error concerning the type
  - Template definition is not found
- Error output
  - Errors on templates are often long and confusing

1st place in category longest error message



```
rtmap.cpp: In function `int main()':
rtmap.cpp:19: invalid conversion from `int' to `
   std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:19:   initializing argument 1 of `std::_Rb_tree_iterator<_Val, _Ref,
   _Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
   std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr =
   std::pair<const int, double>*]'
rtmap.cpp:20: invalid conversion from `int' to `
   std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:20:   initializing argument 1 of `std::_Rb_tree_iterator<_Val, _Ref,
   _Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
   std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr =
   std::pair<const int, double>*]'
E:/GCC3/include/c++/3.2/bits/stl_tree.h: In member function `void
   std::_Rb_tree<_Key, _Val, _KeyOfValue, _Compare, _Alloc>::insert_unique(_II,
    _II) [with _InputIterator = int, _Key = int, _Val = std::pair<const int,
   double>, _KeyOfValue = std::_Select1st<std::pair<const int, double> >,
   _Compare = std::less<int>, _Alloc = std::allocator<std::pair<const int,
   double> >]':
E:/GCC3/include/c++/3.2/bits/stl_map.h:272:   instantiated from `void std::map<_
Key, _Tp, _Compare, _Alloc>::insert(_InputIterator, _InputIterator) [with _Input
Iterator = int, _Key = int, _Tp = double, _Compare = std::less<int>, _Alloc = st
d::allocator<std::pair<const int, double> >]'
rtmap.cpp:21:   instantiated from here
E:/GCC3/include/c++/3.2/bits/stl_tree.h:1161: invalid type argument of `unary *
…
```

Templates and Friends

Friend: allows for access to private methods/attributes

- A friendly function or class that has a template class as a parameter/attribute must also be a template again

operator<< is function template

```
template <int N, class T> class Vector {
  T *d_data;
public:
  ...
  template <int FN, class FT>
  friend std::ostream &operator<<(std::ostream &os, const Vector<FN, FT> &_v);
};

template <int FN, class FT>
ostream& operator<<(ostream& out, const Vector<FN, FT>& v) // free function
{
    out << …
    return out;
}
```

Specialization of a Function

- Custom implementation for a template parameter set
- For some types it may be **necessary, more elegant or more efficient** to deviate slightly from the flow of the original template

```
template <typename T> inline T min(T g, T d) { return ((g < d) ? g : d); }

template <> inline const char *min<const char *>(const char *g, const char *d) {
  if (strcmp(g, d) < 0) { // better because comparing char* (addresses) doesn't make sense
    return g;
  }
  return d;
}

//...

const char strA[] = "A little bigger";
const char strB[] = "A little smaller";
const char *strResult = min(strA, strB);
cout << "min(" << strA << ", " << strB << ") = " << strResult << endl;
```

Specialization of Class Templates

- **Custom** methods**,** functions or classes for **specific** template arguments

Use-cases:

1. Class template works for most types, but needs special implementation for others

2. For some types, a better implementation is possible (faster, more flexible, shorter, ...)

3. New methods for specific types

→ Specialization of class templates for these types

Partial Specialization – `Vector.hpp`

```
#pragma once
#include <iostream>
template< size_t N, class T> class Vector { // N:dimension, T:type
    T * d_data;
public:
    Vector();
    Vector(const Vector<N,T>& _other);
    /** operator for accessing the elements in d_data (get and set) */
    T& operator[](size _i);
    /** operator for accessing the elements in d_data (get only - read
only) */
    const T& operator[](size _i) const;
    /** computes the dot product */
    T dot(const Vector<N,T>& _other) const;
    /** computes the length of the vector */
    T length() const;
     };
…
```

Partial Specialization for 2D – `Vector2.hpp`


```
#include "Vector.hpp"

template<class T> class Vector<2,T> {
    T d_data[2];  // data array can now be of a
                  // fixed size (on stack)
public:
    // we must declare all member functions again
    Vector();
    Vector(const Vector<2,T>& _other);
    /** New constructor added which takes exactly
        two arguments  */
    Vector(const T& _x, const T& _y);
    ~Vector();
    T& operator[](size _i);
    const T& operator[](size _i) const;
...
```

- Changes:
  - Specialized template argument list
  - Specialized attribute type
  - New constructor
- Otherwise, complete declaration of class…

Partial Specialization

- **All methods must be defined once more**

```
template<class T>
Vector<2,T>::Vector() {}

template<class T>
Vector<2,T>::Vector(const Vector<2, T>& _other) {
    d_data[0] = _other[0];
    d_data[1] = _other[1];
}

// novel constructor for two elements
template<class T>
Vector<2,T>::Vector(const T& _x, const T& _y) {
   d_data[0] = _x;
   d_data[1] = _y;
}

template<class T>
Vector<2,T>::~Vector() {
}
...
```

Explicit Full Specialization

- A class template that is fully specialized (=all template parameters are fixed) is no longer a class template → we need to drop the key word **`template<>`** in the definition

```
Vector<2,float>::Vector() { }

Vector<2,float>::Vector(const Vector<2, float>& _other) {
   d_data[0] = _other[0]; d_data[1] = _other[1];
}
```

Specializing Individual Methods

- **Template functions** can only be fully specialized - no partial specialization possible

- **In general:** overloading of a function is preferable to declaration as template and specialization.

- **Recommendation:** only use templates if necessary (e.g., if you are writing a generic library or your design demands it)

Static Attributes in Class Templates

Static member variables of templates follow the rules of static member:

- Static attributes of a template class exist **once for each generated class (type)**

-> Each class in the **type family** defined by a **template class** has its **own static member**

Functors and Templates

- Functor
- Functors as class template
- Functors in the STL

Functor – `operator()`

- An object of a class with **`operator()`** can be called like a function or lambda function:

```
class Doubler  {
public:
  int operator()(int i)
  {
    return 2 * i;
  }
};

…
Doubler d;
int x = d(3); // x = 6
```

- Such objects are called **functors**

Functor – `operator()`

- Can also be a template

```
// declare
template <class T> class equal_to {
public:
  bool operator()(const T &_a, const T &_b);
};

// define
template <class T> bool equal_to<T>::operator()(const T &_a, const T &_b) {
  return (_a == _b);
}

...

int a = 3, b = 4;
equal_to<int> cmp; // construct functor cmp that compares two ints for equality

if (cmp(a, b)) {   // this is a function call (!). Calls operator()
    // ...
}
```

Functors in the STL

- Functors are mostly unary or binary
  - `plus<T>` is a binary functor
  - `negate<T>` is a unary functor

Example from the STL (until C++11):

```
template <class T>
class multiplies : public binary_function<T, T, T>
{
public:
  T operator()(const T& x, const T& y) const
  {
    return x * y;
  }
};
```

- All STL functors have a common base class (**`unary_function<>`**, **`binary_function<>`**) -> can be used polymorphically at compile time.

Functors in the STL

Other functors for comparison…

- `equal_to<T>` operator ==
- `not_equal_to<T>` operator !=
- `greater<T>` operator >
- `greater_equal<T>` operator >=
- `less<T>` operator <
- `less_equal<T>` operator <=

… and logical operations:

- `logical_and<T>` operator &&
- `logical_or<T>` operator ||
- `logical_not<T>` operator !

Example: Change sorting direction

```
// fill randomly
vector<int> v(20);
generate(v.begin(), v.end(), random);

// default: sort in non-decreasing order
sort(v.begin(), v.end());

// with greater<int>(): sort in non-increasing order
sort(v.begin(), v.end(), greater<int>());
```

`typedef` and `using`

1. typedef and templates
2. using

typedef

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Recall: `typedef` introduces an alias for a specific type
- Works also inside class templates

```
typedef unsigned long ulong; // simple typedef introduces alias

// the following two objects now have the same type
unsigned long l1;
ulong l2;

// more complicated typedef  (int, int pointer, function pointer, array)
typedef int int_t, *intp_t, (&fp)(int, ulong), arr_t[10];

// the following two objects have the same type
int a1[10];
arr_t a2;

// STL container define typedefs with a specific name (here: value_type)
template< class T>
struct vector {
    typedef T value_type;
};
// Other template functions that work on container can then uniformly use value_type
```

[https://en.cppreference.com]

using

- `using:` Like typedef, can reintroduce the names of a different scope, e.g. from a base class

```cpp
#include <iostream>
struct B {
  virtual void f(int) { std::cout << "B::f\n"; }
  void g(char) { std::cout << "B::g\n"; }
  void h(int) { std::cout << "B::h\n"; }
protected:
  int m; // B::m is protected
  typedef int value_type;
};

struct D : B {          // private inheritance!
  using B::m;           // make B::m public as D::m
  using B::value_type; // make type public
  using B::f;           // make B::f public so we can overwrite it below
  void f(int) override { std::cout << "D::f\n"; }
  using B::g;           // make B::g overload public
  void g(int) {
    std::cout << "D::g\n";
  } // both g(int) and g(char) are visible as members of D
  using B::h;           // make B::h public but it gets hidden in next line
  void h(int) { std::cout << "D::h\n"; } // D::h(int) hides B::h(int)
};
```

```cpp
int main() {
  D d;
  B &b = d;

//b.m = 2;  // error, B::m is protected
  d.m = 1;  // B::m accessible as public
D::m

  b.f(1);   // calls derived f()
  d.f(1);   // calls derived f()
  d.g(1);   // calls derived g(int)
  d.g('a'); // calls base g(char)
  b.h(1);   // calls base h()
  d.h(1);   // calls derived h()
}
```

[https://en.cppreference.com]

Curiously Recurring Template Pattern (CRTP)

- Inheritance without `virtual`
- Found in many code bases
- Advanced topic

Static Polymorphism and Code-reuse

- static polymorphism separates the "polymorphism"-part from the implementation-part.

- It's still recommended to follow the "don't-repeat-yourself"-principle ("DRY") vs "write-everything-twice" ("WET")
  - less mistakes, easier to test
  - less work to change things later
  - easier to understand (more or less)
  - (tougher compiler errors with templates)

- One can use inheritance without virtual functions to reuse code, but this has limitations.

Static Polymorphism and Code-reuse

- static polymorphism -> not allowed to use virtual and dynamic dispatch!

```
struct MyBase {
  void print1() const { std::cout << "Foo"; }
  void print2() const { print1(); std::cout << "Bar"; }
};

struct MyDerived : MyBase {

  void print3() const { print2(); std::cout << "!"; }
};




template <typename T> void myPrint(T const &v) { v.print3(); }

myPrint(MyDerived{}); // what does it print?
```

Static Polymorphism and Code-reuse

- static polymorphism -> not allowed to use virtual and dynamic dispatch!

```
struct MyBase {
  void print1() const { std::cout << "Foo"; }
  void print2() const { print1(); std::cout << "Bar"; }
};

struct MyDerived : MyBase {

  void print3() const { print2(); std::cout << "!"; }
};


…


template <typename T> void myPrint(T const &v) { v.print3(); }

myPrint(MyDerived{}); // what does it print? Nothing new: FooBar!
```

Static Polymorphism and Code-reuse

```
struct MyBase {
  void print1() const { std::cout << "Foo"; }
  void print2() const { print1(); std::cout << "Bar"; }
};

struct MyDerived : MyBase {
  void print1() const { std::cout << "Chocolate"; }
  void print3() const { print2(); std::cout << "!"; }
};




template <typename T> void myPrint(T const &v) { v.print3(); }

myPrint(MyDerived{}); // what does it print?
```

Static Polymorphism and Code-reuse


```
struct MyBase {
  void print1() const { std::cout << "Foo"; }
  void print2() const { print1(); std::cout << "Bar"; }
};

struct MyDerived : MyBase {
  void print1() const { std::cout << "Chocolate"; }
  void print3() const { print2(); std::cout << "!"; }
};




template <typename T> void myPrint(T const &v) { v.print3(); }

myPrint(MyDerived{}); // what does it print? FooBar! again
```

Observation:

- The (non-virtual) member functions in the base class cannot call overrides or other member functions of derived classes.
- **Think:** in `print2` the address of the function `print1` is hardcoded at compile time.

Curiously Recurring Template Pattern (CRTP)

Problems when using inheritance for code-reuse:

- We saw: **non-virtual member functions** in a base class that call other member functions cannot call overrides or other member functions of derived classes

- Some member functions need to return the type itself (e.g., `operator=`), but functions in a base class return the type of the base class instead

Core Problem: The derived type "knows" the base type, but the base type does not know anything about the derived type

Solution: The **"curiously recurring template pattern"** (CRTP):
- base class becomes a template, and the derived type passes its own (incomplete) type to the base type as a template argument.
- Sounds confusing? Let's take a look at the code…

Curiously Recurring Template Pattern (CRTP)

```
template<typename Derived>
struct MyBase {
  // ...
};

struct MyDerived : MyBase<MyDerived> {
  // ...
};
```

- `MyBase` is a class template that takes a type `Derived`
- `MyDerived` specializes `MyBase` with its own type **and** then inherits from that specialization
- `MyDerived` is *incomplete* at the time of specialization (that's ok…).
- It looks like a recursive definition, but in fact it's not. Think of it as the compiler just needs to do some (simple) text substitutions in the base class.

Curiously Recurring Template Pattern (CRTP)

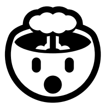
```
template <typename Derived> struct MyBase {
  Derived const & toDerived() const // const version
  {
    return static_cast<Derived const &>(*this);
  }

  Derived & toDerived()               // non-const version
  {
    return static_cast<Derived &>(*this);
  }
  // ...
};
```

- `toDerived()` returns a reference to an object of the derived type (more precisely a reference to the current object, but cast to the derived type)
- The `static_cast` works because we ensured that the object is of type `Derived`
- Usually, you have a **`const`** and a non-**`const`**-overload

Curiously Recurring Template Pattern (CRTP)

```
template <typename Derived> struct MyBase {
  Derived const &toDerived() const
  {
    return static_cast<Derived const &>(*this);
  }
  …
  void print1() const { std::cout << "Foo"; }
  void print2() const { toDerived().print1(); std::cout << "Bar"; }
};
```

- Calls **`toDerived().print1()`** instead of calling **`print1()`**
- This casts the local object to the derived type and calls the derived type's **`print1()`** member
- If **`print1()`** is overridden in the derived type, the override is executed; if there is no override, the base's member is called (because that’s inherited by the derived type)



Curiously Recurring Template Pattern (CRTP)

```cpp
template <typename Derived> struct MyBase {
  Derived const &toDerived() const { return static_cast<Derived const &>(*this); }

  Derived &toDerived() { return static_cast<Derived &>(*this); }

  void print1() const { std::cout << "Foo"; }
  void print2() const { toDerived().print1(); std::cout << "Bar";
  }
};

struct MyDerived : MyBase<MyDerived> {
  void print1() const { std::cout << "Chocolate"; }
  void print3() const { print2();  std::cout << "!"; }
};

template <typename T> void myPrint(T const &v) { v.print3(); }

myPrint(MyDerived{}); // what does it print?
```

Curiously Recurring Template Pattern (CRTP)

```cpp
template <typename Derived> struct MyBase {
  Derived const &toDerived() const { return static_cast<Derived const &>(*this); }

  Derived &toDerived() { return static_cast<Derived &>(*this); }

  void print1() const { std::cout << "Foo"; }
  void print2() const { toDerived().print1(); std::cout << "Bar";
  }
};

struct MyDerived : MyBase<MyDerived> {
  void print1() const { std::cout << "Chocolate"; }
  void print3() const { print2();  std::cout << "!"; }
};

template <typename T> void myPrint(T const &v) { v.print3(); }

myPrint(MyDerived{}); // what does it print? ChocolateBar!
```

CRTP – Issues:

- You cannot construct an object of the base class nor a pointer to the base class, it is a class template

- Therefore, you cannot have a container collecting objects of different derived classes **and** the benefits of having no overhead from the virtual function calls

Example Applications:

- Enforce a static interface
- Implement functionality (e.g., serialization) in base function that needs type of derived
- Implementing iterators require to write >20 member functions, with CRTP this can be reduced to 5.
- Implementing three comparison operators(==, <, >) and "default" the others.
  Detail: Not needed anymore if the proposed (starship) operator<=> gets standardized.
- …

Summary

- Follow the "DRY"-principle, i.e. reduce code-duplication, reuse code as much as possible.
- When using static polymorphism, do not combine it with virtual functions.
- You may still want to use inheritance to "DRY" your code.
- In many situations you will need to use CRTP to achieve this. You will find it in a lot of code bases.
- **"curiously recurring template pattern" (CRTP):** the base class becomes a template and the derived type passes its own (incomplete) type to the base type as a template argument.
- This way the base type can access functionality of derived types **without run-time overhead.**
- In the future, **concepts** might replace some current applications of CRTP

- Further reading:
  - https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
  - https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern
  - https://www.grimm-jaud.de/index.php/blog/c-ist-doch-lazy

Concepts

1. Compile time contracts with static_assert and concepts
2. Making template code safer

static_assert

- performs compile-time assertion checking, compile time contracts

```
template <class T, int Size> class Vector {
  // Compile time assertion any vector is declared whose size is less
  // than 4, the assertion will fail
  static_assert(Size > 3, "Vector size is too small!");
  T m_values[Size];
};

int main() {
  Vector<int, 4> four;  // This will work
  Vector<short, 2> two; // This will fail "Vector size is too small!"
  return 0;
}
```

[https://www.geeksforgeeks.org/understanding-static_assert-c-11/]

Concepts

- **A concept is a named set of requirements to enforce constraints on template parameters**

- Syntax:

```
template < template-parameter-list >
concept concept-name = constraint-expression;
```

- Example:

```
class Base {…} // just some class with name Base

// define concept DerivedFromBase
template <class T>
concept DerivedFromBase = std::is_base_of<Base, T>::value;

// apply concept
template < DerivedFromBase<T> > void f(T); // -> T must be derived from Base
```

Constraints

- We can enforce multiple constraints by combining concepts:

```
template <class T>
concept Integral = std::is_integral<T>::value;                      // a single constraint

template <class T>
concept SignedIntegral = Integral<T> && std::is_signed<T>::value; // two constraints

template <class T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

```
template <typename T> constexpr bool get_value() { return T::value; }
template <typename T>
requires(sizeof(T) > 1 && get_value<T>()) void f(T) {...}; // #1
void f(int) {...};                                          // #2

void g() {
  f('A'); // OK, calls #2. When checking the constraints of #1,
          // 'sizeof(char) > 1' is not satisfied, so get_value<T>() is not
          // checked
}
```

Different Ways of Using a Concept

```
template <class T>
concept MyConcept = std::is_integral<T>::value;                    // define concept MyConcept<T>
auto func1(MyConcept auto param) { return param + 3; }

template <typename T>
requires MyConcept<T>                                              // after template
auto func2(T param) { return param + 3; }

template <typename T> auto func3(T param) requires MyConcept<T> { // after signature
  return param + 3;
}

// like a type
template <MyConcept T> auto func4(T param) { return param + 3; }

// vs. totally unconstraint
auto func5(auto param) { return param + 3; }
```

```
// valid:
cout << "func1(10) = " << func1(10) << endl;
cout << "func2(10) = " << func2(10) << endl;
cout << "func3(10) = " << func3(10) << endl;
cout << "func4(10) = " << func4(10) << endl;
cout << "func5(10) = " << func5(10) << endl;

string s{"txt"};
// compile error: candidate template ignored: constraints not satisfied
// cout << "func1( \"txt\" ) = " << func1("txt") << endl;

// compile error when trying to {s + 3}

// cout << "func5( s ) = " << func5(s) << endl;
// compiles fine (!) but does not produce any useful
// output ... (it is a const char*)
cout << "func5( \"txt\" ) = " << func5("txt") << endl;
```

Expressing Constraints

```
template<typename T>
concept Addable =
requires (T a, T b) {
    a + b;  // require that "the expression a+b is a valid expression that will compile"
};

template <class T, class U = T>
concept Swappable = requires(T&& t, U&& u) {          // require that values of type T and U can be swapped
    swap(std::forward<T>(t), std::forward<U>(u));    // adv. detail: instantiates std::swap but passes u and t (both lvalues) with their
    swap(std::forward<U>(u), std::forward<T>(t));    //     original value category (lvalue or rvalue) given by T and U.
};

template<typename T> concept C =
requires {
    typename T::inner; // required nested member with name inner
    typename S<T>;     // required class template specialization
};

template<typename T> concept C2 =
requires(T x) {
    {*x} -> std::convertible_to<typename T::inner>; // the expression *x must be valid
                                                    // AND the type T::inner must be valid
                                                    // AND the result of *x must be convertible to T::inner

    {x + 1} -> std::same_as<int>;   // the expression x + 1 must be valid
                                    // AND std::same_as<decltype((x + 1)), int> must be satisfied
                                    // i.e., result of (x + 1) must be of type int

    {x * 1} -> std::convertible_to<T>; // the expression x * 1 must be valid
                                       // AND its result must be convertible to T
};
```

<type_traits>

- Check for properties of a specific type

```
is_void
is_null_pointer
is_integral
is_floating_point
is_array
is_pointer
is_lvalue_reference
is_rvalue_reference
is_member_object_pointer
is_member_function_pointer
is_enum
is_union
is_class
is_function
```

```
is_reference
is_arithmetic
is_fundamental
is_object
is_scalar
is_compound
is_member_pointer
```

```
is_const
is_volatile
is_trivial
is_trivially_copyable
is_standard_layout
is_empty
is_polymorphic
is_abstract
is_final
is_aggregate

is_signed
is_unsigned
is_bounded_array
is_unbounded_array
is_scoped_enum
```

…

Template Metaprogramming

Template Metaprogramming

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Templates define a powerful (turing complete) meta language that can be (mis-)used for many things.
- Using it is called **Template-Metaprogramming.** Allows compile time calculations.

```
template <int N>
class fac
{
  public:
  static int value() { return N * fac<N-1>::value(); }
};
// Break out of compile time recursion with partial specialization for 0!
template <>
class fac <0>
{
  public:
  static int value() { return 1; }
};
…
fac<7>::value(); // == 5040 = 7!
```

- Many of TMP techniques get easier or can be replaced in newer C++ versions.

Template Metaprogramming

- Many of TMP techniques get easier or can be replaced in newer C++ versions.

```
clang 15.0  with -std=c++20
```

```cpp
consteval unsigned factorial(unsigned n) {
    return n < 2 ? 1 : n * factorial(n - 1);
}

int main()
{
  return factorial(7); // 7! = 5040 generated at compile time
}
```

Proof: generated assembly code has *compile-time* value baked in.

```
main: # @main
  push rbp
  mov rbp, rsp
  mov dword ptr [rbp - 4], 0
  mov eax, 5040
  pop rbp
  ret
```

Summary - Polymorphism

	dynamic	static
how	Inheritance + virtual functions	overloaded functions + templates
model	object-oriented	generic programming
dispatch at	run-time	compile-time
run-time	slower	faster
good for	complex and deep type hierarchies; frameworks; large “in-house” solutions	flat hierarchies; “external types”; generic libraries; high-performance
difficulty	easier to program	more difficult to program

Summary

- Class Templates
- Specialization
- Functors
- Curiously Recurring Template Pattern (CRTP)
- Concepts
- Template Metaprogramming