# Constrained genericity in ConceptC++ and D

**Sources:**

- [Czarnecki and Eisenecker 2000] §6 (Generic programming)
- Perform web-search with term *ConceptC++*
- Perform web-search with term *D programming language*

# Polymorphism

The word **polymorphism** means "the ability to have many forms".

**Parametric polymorphism:** C++ templates

**Inclusion polymorphism:** C++ virtual functions

**Overloading:** C++ function overloading including partial specialization

**Coercion:** C++ built-in or user defined conversion operators or constructors to coercion

# Bounded polymorphism

**Bounded parametric polymorphism** (or **constrained genericity**) means that we can specify some constraints on type parameters.

In C++, there is no way to specify constrains on type parameters, but many clever tricks and workarounds exist to support generic programming (including type mappings, tag dispatching, and SFINAE)

There are two approaches to specify constraints on type parameters:

1. use an interface defined elsewhere

```
template <LessThanComparable T>
class point {
  // ...
}
```

2. list all the required operations in place

```
template <typename T>
  requires {bool operator<(T const&, T const&);}
class point {
  // ...
}
```

# Problems with C++ templates

It is theoretically interesting that the template level of C++ has the power of a Turing machine, but template meta-programming has its problems, particularly in the areas of

- error reporting,
- debugging,
- code readability,
- code maintainability,
- separate compilation,
- compilation speed,
- internal capacity and robustness of compilers, and
- portability.

Most problems seem to be related to unbounded parametric polymorphism.

# Motivating example

```
#include <list>
#include <algorithm>
using namespace std;

void f() {
  list<int> l;
  sort(l.begin(), l.end());
}
```

```
sort.C:7: error: no matching function for call to 'sort(std::List_iterator< ↩
  int>,␣std::List_iterator<int>)'
<path>: note: candidates are: void std::sort(Iter, Iter) [with Iter = std:: ↩
  List_iterator<int>] <requires clause>
sort.C:7: note: unsatisfied model requirement 'std:: ↩
  MutableRandomAccessIterator<std::List_iterator<int>␣>'
```

# Four definitions

"A **concept** is a set of requirements [on types] bundled together under a single name." [Gregor 2006]

"a type system—called **concepts**—for C++ types and values that can be used for template arguments" [Reis & Stroustrup 2006]

"concepts are compile-time predicates on types and values (e.g. integral constant values). They can be combined with the usual logical operators (*and*, *or*, *not*)." [Reis & Stroustrup 2006]

"Everybody's first idea for [defining the predicates] is to specify a concept as a set of operations" [Reis & Stroustrup 2006]

# Old style

```
template <typename R>
void stable_sort(R a, R z);
```

## Requirements for types

- `R` is a model of random-access iterator.
- `R` is mutable.
- `R`'s `value_type` is strict weakly comparable.

...

## Complexity guarantees

Let $N$ be `z`$-$`a`. The worst-case behaviour is $O(N(\lg N)^2)$ if no auxiliary memory is available, and $O(N \lg N)$ if a large enough auxiliary memory buffer is available.

# New style

```
template <MutableRandomAccessIterator R>
  requires LessThanComparable<R::value_type>
void stable_sort(R a, R z);
```

## Semantic requirements

**operator**<() on the set of elements of `R`'s `value_type` is a strict weak ordering.

. . .

## Complexity guarantees

. . . $O(N(\lg N)^2)$ . . .

# Pseudo-signatures

**Pseudo-signatures** permit conversions of the argument and result types.

```
concept LessThanComparable<typename T> {
  bool operator<(T const&, T const&);
  bool operator>(T const&, T const&);
  bool operator≤(T const&, T const&);
  bool operator≥(T const&, T const&);
};
```

The declaration of **operator**<() requires the existence of a $<$ operator, either built in, as a free function, or as a member function, that can be passed two values convertible to type `T` and returns a value convertible to `bool`.

# Associated types

**Associated types** are represented as nested types within the concept; they replace traits and permit checking of template definitions.

```
concept IteratorAssociatedTypes<typename X> {
  typename value_type = X::value_type;
  typename difference_type = X::difference_type;
  typename reference = X::reference;
  typename pointer = X::pointer;
};
```

If a model does not specify a type definition for an associated type, then the model uses the default.

# Some standard concepts

```
concept Assignable<typename T, typename U = T> {
  T& operator=(T&, U const&);
};

concept EqualityComparable<typename T, typename U = T> {
  bool operator≡(T const&, U const&);
  bool operator≢(T const&, U const&);
};

concept Convertible<typename T, typename U> {
  operator U(T const&);
}; // built-in, constructor, or member operation

concept DefaultConstructible<typename T> {
  T::T();
  T::~T();
};

concept Dereferenceable<typename PtrLike, typename Value> {
  Value operator*(PtrLike&);
};
```

# Refinement

```
concept InputIterator<typename X>
  : Assignable<X>, EqualityComparable<X>, IteratorAssociatedTypes<X> {

  requires SignedIntegral<difference_type> &&
           Convertible<reference, value_type> &&
           Arrowable<pointer, const value_type*>;

  typename postincrement_result = X;
  requires Dereferenceable<postincrement_result, value_type>;

  pointer operator→(X);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
  reference operator*(X const&);
};
```

# Concept-based overloading

```
template <InputIterator Iter>
void advance(Iter& i, difference_type n) {
  while (n ≠ 0) {
    ++i; --n;
  }
}

template <BidirectionalIterator Iter>
void advance(Iter& i, difference_type n) {
  while (n > 0) {
    ++i; --n;
  }
  while (n < 0) {
    --i; ++n;
  }
}

template <RandomAccessIterator Iter>
void advance(Iter& i, difference_type n) {
  i += n;
}
```

# Concept maps

A **model declaration** illustrates how a set of types will model a particular concept.

```
template <typename T>
concept_map ForwardIterator<T*> {
  typedef T value_type;
}
```

Each model must meet all of the requirements in the concept.

# D supports (a limited form of) constraints

```
class B { ... }
interface I { ... }

class F(
  R,              // R can be any type
  P:P*,           // P must be a pointer type
  T:int,          // T must be int type
  S:T*,           // S must be pointer to T
  C:B,            // C must be of class B or derived from B
  U:I,            // U must be a class that implements interface I
  char[] string = "hello", // string literal, default is "hello"
  alias A = B     // A is any symbol (including template symbols), defaulting to B
  ) {
    ...
}
```

# Pros and cons of constrained genericity

+ improved error messages
+ debugging easier for library authors
+ explicit descriptions of the import interfaces
+ new opportunities for overloading
+ separate compilation possible
+ improved static type checking
+ lower barrier to novices

− more to learn
− more to type
− duplication of the interface information
− flexibility of lazy type checking lost
− possibility for over-specification

# Conclusion

In this course, we cannot any more concentrate only on C++.