BASIC CONCEPTS

- **Overview**
  - Programming Paradigms
  - Values, Sets, Arrays, Templates
  - Indexer, Lambda Expressions, Iterators, and Recursion
- **References**

  - Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
  - Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel. 2nd Edition. Charles River Media Inc. (2005)
  - Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
  - Scott Meyers: Effective Modern C++. O'Reilly (2014)
  - Anthony Williams: C++ Concurrency in Action - Practical Multithreading. Manning Publications Co. (2012)

© Dr Markus Lumpe, 2023

PROGRAMMING PARADIGMS

- **Imperative style:**

  program = algorithms + data

- **Functional style**:

  program = function • function

- **Logic programming style**:

  program = facts + rules

- **Object-oriented style**:

  program = objects + messages

- Other styles and paradigms:

  blackboard, events, pipes and filters, constraints, lists, …

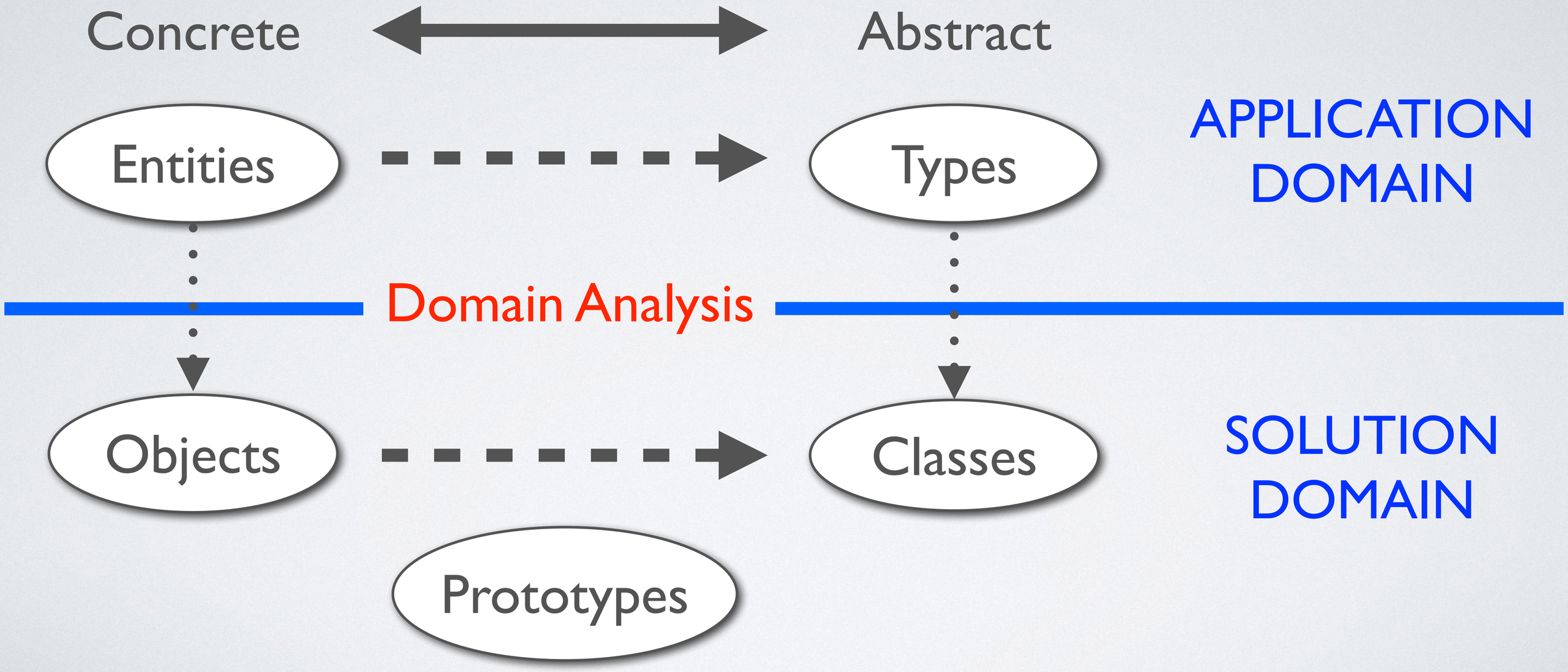
© Dr Markus Lumpe, 2023

OBJECT-ORIENTED SOFTWARE DEVELOPMENT

- Object-oriented programming is about
  - Object-oriented software development
  - Using an object-oriented programming language
- Object-oriented software development is
  - An evolutionary step refining earlier techniques
  - A revolutionary idea perfecting earlier methods

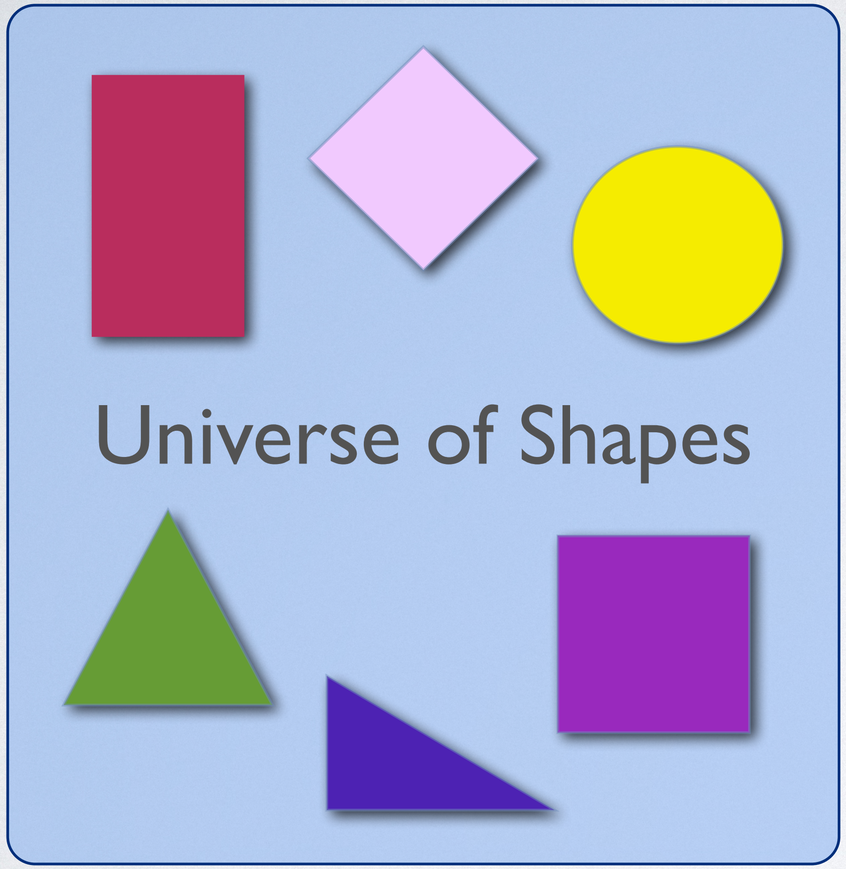
© Dr Markus Lumpe, 2023

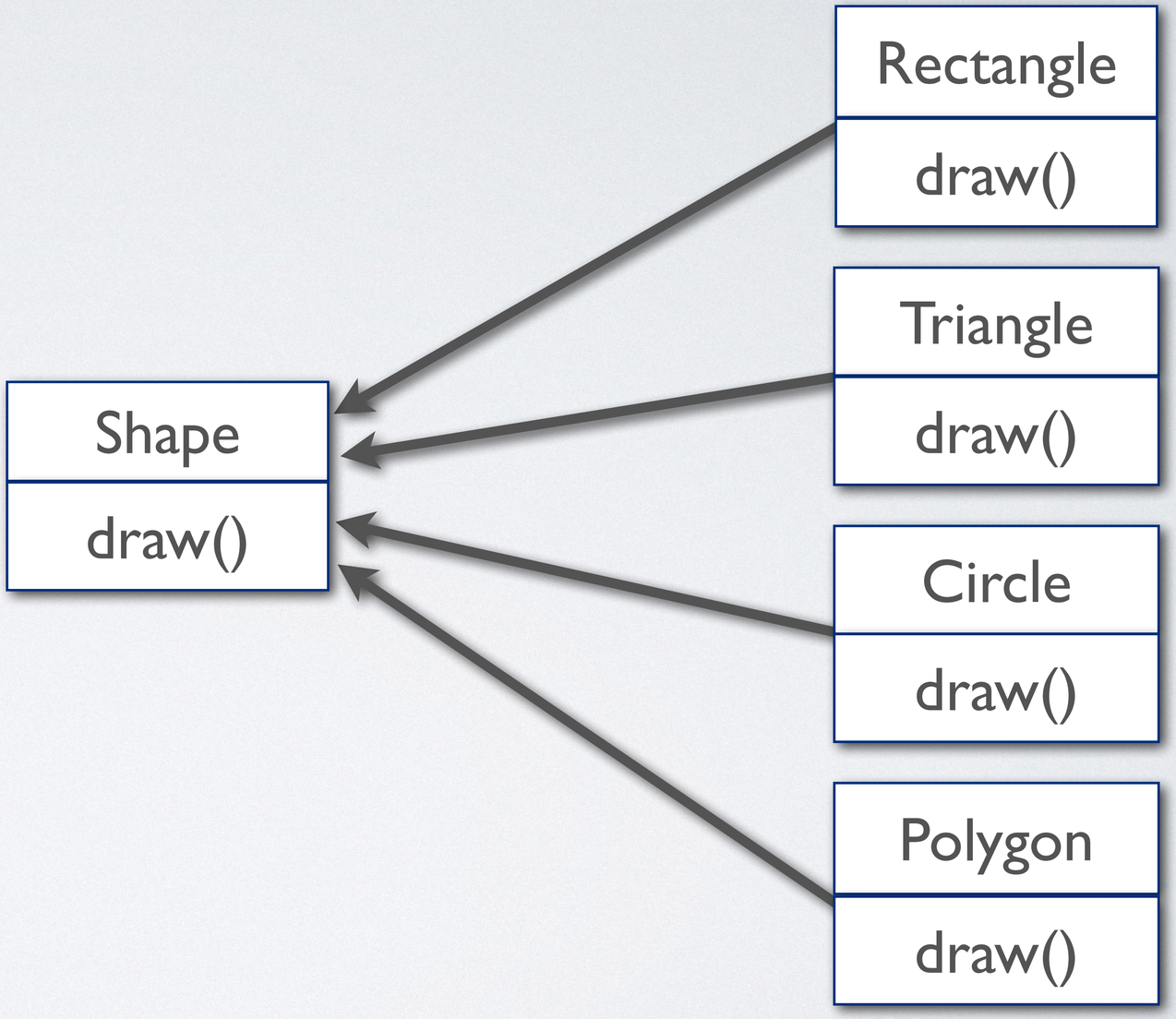
OBJECT-ORIENTED DESIGN



© Dr Markus Lumpe, 2023

CONCRETE VS ABSTRACT





© Dr Markus Lumpe, 2023

WHY IS OBJECT-ORIENTED SOFTWARE DEVELOPMENT POPULAR?

- The object-oriented development approach
  - Naturally captures real life
  - Scales well from trivial to complex tasks
  - Focuses on responsibilities, reuse, and composition

© Dr Markus Lumpe, 2023

VALUES

- In computer science we classify as a value everything that may be evaluated, stored, incorporated in a data structure, passed as an argument to a procedure or function, returned as a function result, and so on.
- We use "expressions" as a means to denote a values, as in mathematics.
- Which kinds of values are supported by a specific programming environment heavily depends on the underlying paradigm and its application domain.
- Most programming environments provide support for some basic sets of values like truth values, integers, real number, records, lists, etc.

© Dr Markus Lumpe, 2023

CONSTANTS

- Constants are named abstractions of values.
- Constants are used to assign a user-defined meaning to a value.
- Examples:
  - **EOF** = -1
  - **TRUE** = 1
  - **FALSE** = 0
  - **PI** = 3.1415927
  - **MESSAGE** = "Welcome to DSP"
- Constants do not have an address, that is, they do not have a defined (programmatically usable) location.
- At compile time, applications of constants are substituted by their corresponding definition.

© Dr Markus Lumpe, 2023

PRIMITIVE VALUES

- Primitive values are values whose representation cannot be further decomposed. We find that some of these values are implementation and platform dependent.

- Examples:
  - Truth values,
  - Integers,
  - Characters,
  - Strings,
  - Enumerators,
  - Real numbers.

-1

"Hello World!"

3.14159

false

Red

© Dr Markus Lumpe, 2023

COMPOSITE VALUES

- Composite values are built up using primitive values and composite values. The layout of composite values is in general implementation dependent.
- Examples:
  - Records
  - Arrays
  - Enumerations
  - Sets
  - Lists
  - Tuples
  - Files

```
struct Point
{
    int x;
    int y;
} gPoint = { 1, 2 };

enum class Color
{
    Red,
    Blue,
    Green
};
```


© Dr Markus Lumpe, 2023

POINTERS

- Pointers are values that denote locations (or the address) of other values in memory.
- Most programming languages use pointers to provide effective access to objects stored in memory. However, only a few, like C++, also support operations to obtain and manipulate the address of a value.
- In C++, we can obtain the address of any l-value and function. The later is called a pointer to a function, which, prior to the introduction of lambda expression, is the only way to pass a (statically defined) function as argument to an other function.

```
int i = 10;
int* ptr_to_i = &i;          // using & to obtain address of i

int convert( const char* aCString, int(*aMap)(char aChar))
{
   int Result = 0;

   while ( *aCString )       // using * to deference pointer aCString
   {
      Result = Result * 10 + aMap( *aCString );
      aCString++;            // pointer arithmetic, advance to next character
   }

   return Result;
}

int ctoi( char aChar )
{
   assert( isdigit( aChar ) );        // debug-mode assertion testing

   return aChar - '0';                // characters are integers, '0' == 0x30
}

…

const char* lString = "12345";

std::cout << "12345: " << convert( lString, ctoi ) << std::endl;
```


© DPI Markus Lampe, 2023

SETS

- A set is a collection of elements (or values), possibly empty.
- All elements satisfy a possibly complex characterizing property. Formally, we write:

$$\{ x \mid P(x) = \text{True} \}$$

- to define a set, where all elements satisfy the property **P(x)**.
- The basic axiom of set theory is that there exists an empty set, $\varnothing$, with no elements. Formally,

$$\forall x, \; x \notin \varnothing$$

In words, "for every x, x is not an element of $\varnothing$."

© Dr Markus Lumpe, 2023

INDUCTIVE SPECIFICATION

- Sometimes it is difficult to define a set explicitly, in particular if the elements of the set have a complex structure.
- However, it may be easy to define the set in terms of itself. This process is called inductive specification or recursion. Inductive specification yields a finite recipe for the construction of a possibly infinite set of values.
- Example:

  Let the set S be the smallest set of natural numbers divisible by 3 satisfying the following two properties:

  - $0 \in S$, and
  - Whenever $x \in S$, then $x + 3 \in S$.

  The first property is called base clause and the second property is called inductive/recursive clause. An inductive specification may have multiple base and inductive clauses.

© Dr Markus Lumpe, 2023

THE "SMALLEST SET"

- If we use inductive specification, we always define the smallest set that satisfies all given properties. That is, inductive specification is free of redundancy.

- It is easy to see that there can be only one such set:

  If S_1 and S_2 both satisfy all given properties and both are the smallest, then we have $S_1 \subseteq S_2$ (since S_1 is the smallest), and $S_2 \subseteq S_1$ (since S_2 is the smallest), hence $S_1 = S_2$. Q.E.D.

© Dr Markus Lumpe, 2023

INDEXED SETS

- Sets are unordered collections of data elements.

- In order to obtain an ordering relation over the elements of a given set, we can assign each element in that set a unique element of another totally ordered set **I**:

$$S_{\mathbf{I}} = \{ a_i \mid a \in S, i \in \mathbf{I} \}$$

$S_{\mathbf{I}}$ is called the "indexed set" of S.

© Dr Markus Lumpe, 2023

SOME INDEXED SETS

- Let $A = \{ a, b, c, d \}$ and $\mathbf{I} = \{ 1, 2, 3, 4 \} \subset \mathcal{N}$, then

$$A_{\mathbf{I}} = \{ a_1, b_2, c_3, d_4 \}$$

- Let $A = \{ a, b, c, d \}$ and $\mathbf{I} = \{ \text{“1”}, \text{“2”}, \text{“3”}, \text{“4”} \} \subset (\mathbf{S} \times \mathbf{S}, <)$, then

$$A_{\mathbf{I}} = \{ a_{\text{“1”}}, b_{\text{“2”}}, c_{\text{“3”}}, d_{\text{“4”}} \}$$

- Arrays are indexed sets.

© Dr Markus Lumpe, 2023

PAIRS AND MAPS

- Let A and B be sets. The Cartesian product of A and B, denoted by $A \times B$, is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$:

$$A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$$

- A map is an associative container, whose elements are defined as key-value pairs. The key serves as an index into the map, and the value represents the data being stored and retrieved.

© Dr Markus Lumpe, 2023

ASSOCIATIVE ARRAYS (DICTIONARIES)

- An associative array is a map in which elements are indexed by a key rather than by their position. Technically, an associative array is a partial function whose domain is a set of keys and its codomain is a set of values mapped by keys.

- For convenience, we use the standard array indexing syntax to denote a value lookup. If the associative array does not define a corresponding mapping, then an exception has to occur:

$$a[i] = \begin{cases} v & \text{if } i \mapsto v \text{ in } a \\ \perp & \text{otherwise} \end{cases}$$

- Example:

  a = { (“u” ↦345), (“v” ↦ 2), (“w” ↦ 39), (“x” ↦ 5) }

  a[“w”] = 39

  a[“z”] = ⊥

© Dr Markus Lumpe, 2023

AN INDEXER AS CLASS ADAPTER

- We can define an adapter class that defines an indexer:

```
#include <cstddef>
#include <string>

class SimpleAssociativeArray
{
private:
    int* fArray;          // int data array (delegate object)
    size_t fSize;

public:
    SimpleAssociativeArray( int aArray[], size_t aSize ) noexcept;

    size_t size() const noexcept;

    int& operator[]( size_t aIndex ) const;
    int& operator[]( const std::string& aKey ) const;
};
```


© Dr Markus Lumpe, 2023

CONSTRUCTOR

```
SimpleAssociativeArray::SimpleAssociativeArray( int aArray[], size_t aSize ) noexcept :
    fArray(aArray),
    fSize(aSize)
{}
```

- When we pass an array to a function than the array type "decays" to a pointer to the first element in C++.
- Consequently, we have to pass the size of the array also.
- The member variable fArray has to be of pointer type to assign the array passed as argument. pointer variable can be used like an array variable using the selection operator [].

© Dr Markus Lumpe, 2023

SIZE

```
size_t SimpleAssociativeArray::size() const noexcept
{
    return fSize;
}
```

- The function size() simply returns the size of the wrapped array.

© Dr Markus Lumpe, 2023

INDEXED SET ACCESS - ARRAY SEMANTICS

- To define standard array semantics we overload the index operator [].
- The index is of integral type and denotes the position of the element in the array.
- Integral types define an total ordering relation. Hence every element has a unique position in the array.

```
int& SimpleAssociativeArray::operator[]( size_t aIndex ) const
{
    assert( aIndex < fSize );

    return fArray[aIndex];
}
```


© Dr Markus Lumpe, 2023

INDEXED SET ACCESS - ASSOCIATIVE ARRAY SEMANTICS

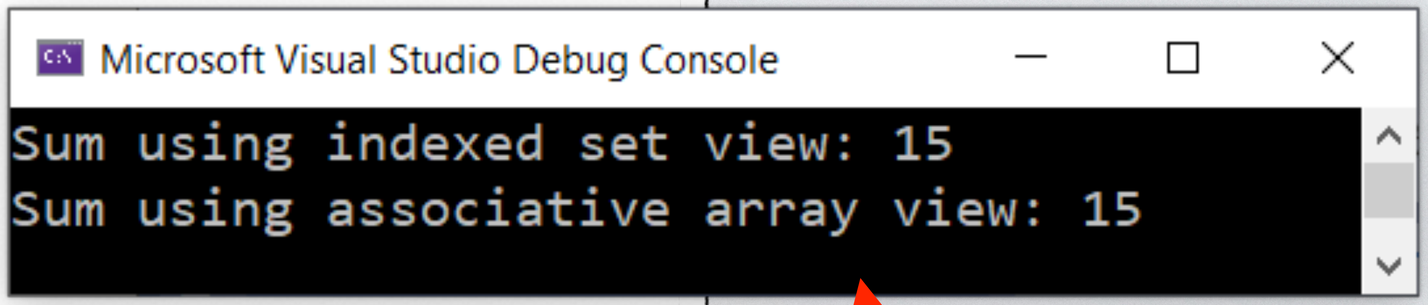
```
int& SimpleAssociativeArray::operator[]( const std::string& aKey ) const
{
    size_t lIndex = 0;

    for ( char c : aKey )                    // scan key string
    {
        if ( isdigit( c ) )
        {
            // convert string to number
            lIndex = lIndex * 10 + static_cast<size_t>(c) - '0';
        }
        else
        {
            break;
        }
    }

    return (*this)[lIndex];          // forward to operator[]( size_t aIndex )
}
```

- We can define a map from strings to numbers.
- The **for**-loop works similar to the function std::stoi().
- We forward to the obtained index to the index operator which yields the element and performs range checking, if necessary.

© Dr Markus Lumpe, 2023

APPLY OPERATIONS

```
int IArray[] = { 1, 2, 3, 4, 5 };
std::string IKeys[] = { "0", "1", "2", "3", "4" };
const int ISize = sizeof( IArray ) / sizeof( int );

SimpleAssociativeArray IDict( IArray, ISize );

int ISum1 = 0;
int ISum2 = 0;

for ( size_t i = 0; i < ISize; i++ )
{
    ISum1 += IDict[i];
    ISum2 += IDict[IKeys[i]];
}

std::cout << "Sum using indexed set view: " << ISum1 << std::endl;
std::cout << "Sum using associative array view: " << ISum2 << std::endl;
```



© Dr Markus Lumpe, 2023

TEMPLATES

© Dr Markus Lumpe, 2023

TEMPLATES: DEFINING GENERIC ABSTRACTIONS IN C++

- Templates are the basis for generic programming in C++.
- C++ requires all variables to have a specific type, either explicitly declared or deduced by the compiler.
- However, many data structures and algorithms are type-agnostic, that is, they work the same irrespective of the types they operate on.
- Templates enable us to define the operations of classes and function as blueprints and let the user specify/instantiate what concrete types those operations should work on. The latter are called specializations.
- **Note**, every time a given template is being instantiated with type parameters that have not been used before, a new version of the class or function is being generated.

© Dr Markus Lumpe, 2023

DEFINING A TEMPLATE

- A template is a parameterized abstraction over a function or class.
- From the language-theoretical perspective, templates are 2nd order functions from types to classes or functions.
- To instantiate a class template we supply the desired types, as actual template parameters, so that the C++ compiler can synthesize a specialized class for the template.
- Further information: Nicolai M. Josuttis: C++17 - The Complete Guide. https://leanpub.com/cpp17 (2019)

© Dr Markus Lumpe, 2023

FUNCTION TEMPLATE

Placeholder for a type.

```
template<typename T1, ..., typename Tn>
T1 AFunction( T2 aParam1, …, Tn aParamn )
{
    // body of function
};

type1 value = AFunction<type1, …, typen>( value1, …, valuen );
```

$T_1, \ldots, T_n$ occur in function signature.

$T_1, \ldots, T_n$ instantiated in function specialization.

Types of $value_1, \ldots, value_n$ must match $type_1, \ldots, type_n$

© Dr Markus Lumpe, 2023

INSERT SORT FUNCTION TEMPLATE

T is element type.

```
template<typename T, typename C = std::greater<T>>
void doInsertSort( T aArray[], size_t aSize, C isOutOfOrder = C{} )
{
    size_t i = 1;

    while ( i < aSize )
    {
        size_t j = i;

        while ( j > 0 && isOutOfOrder( aArray[j-1], aArray[j] ) )
        {
            std::swap( aArray[j-1], aArray[j] );

            j--;
        }

        i++;
    }
}
```

Parameter with default type.

The ordering criterion can be changed programmatically.

std::swap is also a template function

© Dr Markus Lumpe, 2023

SORT AN ARRAY

```
int IArray[] = {45,34,8,6,5,1,0,-2,-3,-100};
const size_t ISize = sizeof(IArray)/sizeof(int);

printArray( IArray, ISize );

doInsertSort( IArray, ISize );

printArray( IArray, ISize );

doInsertSort( IArray, ISize, std::less<int>{} );

printArray( IArray, ISize );
```

Compiler can deduce instantiation types.

Microsoft Visual Studio Debu...

```
[45,34,8,6,5,1,0,-2,-3,-100]
  [34,45,8,6,5,1,0,-2,-3,-100]
  [8,34,45,6,5,1,0,-2,-3,-100]
  [6,8,34,45,5,1,0,-2,-3,-100]
  [5,6,8,34,45,1,0,-2,-3,-100]
  [1,5,6,8,34,45,0,-2,-3,-100]
  [0,1,5,6,8,34,45,-2,-3,-100]
  [-2,0,1,5,6,8,34,45,-3,-100]
  [-3,-2,0,1,5,6,8,34,45,-100]
  [-100,-3,-2,0,1,5,6,8,34,45]
[-100,-3,-2,0,1,5,6,8,34,45]
  [-3,-100,-2,0,1,5,6,8,34,45]
  [-2,-3,-100,0,1,5,6,8,34,45]
  [0,-2,-3,-100,1,5,6,8,34,45]
  [1,0,-2,-3,-100,5,6,8,34,45]
  [5,1,0,-2,-3,-100,6,8,34,45]
  [6,5,1,0,-2,-3,-100,8,34,45]
  [8,6,5,1,0,-2,-3,-100,34,45]
  [34,8,6,5,1,0,-2,-3,-100,45]
  [45,34,8,6,5,1,0,-2,-3,-100]
[45,34,8,6,5,1,0,-2,-3,-100]
```


© Dr Markus Lumpe, 2023

CLASS TEMPLATE

Placeholder for a type.

```
template<typename T1, ..., typename Tn>
class AClassTemplate
{
    // class specification
};

AClassTemplate<type1, …, typen> obj{};
```

$T_1, \ldots, T_n$ occur in class specification.

$T_1, \ldots, T_n$ instantiated in class specialization.

new initializer syntax

© Dr Markus Lumpe, 2023

INSERTION SORTER CLASS TEMPLATE

```
template<typename T>
class InsertionSorter
{
private:
    std::vector<T> fData;

public:
    InsertionSorter( const std::vector<T>& aData ) : fData(aData) {}

    template<typename C = std::greater<T>>
    void operator()( C alsOutOfOrder = C{} )
    {
        size_t i = 1;

        while ( i < fData.size() )
        {
            size_t j = i++;

            while ( j > 0 && alsOutOfOrder(fData[j - 1], fData[j]) )
            {
                std::swap( fData[j - 1], fData[j] );
                j--;
            }
        }
    }

    const std::vector<T>& data() const { return fData; }
};
```

T is element type.

Sorter orders vectors.

Parameter with default type.

The ordering criterion is a parameter of the sorting process, not the class itself.

© Dr Markus Lumpe, 2023

SORT A VECTOR

Compiler deduces all instantiation types.

```
std::vector IData{ 45, 34, 8, 6, 5, 1, 0, -2, -3, -100 };
InsertionSorter ISorter{IData};

printVector(ISorter.data());

ISorter();

printVector(ISorter.data());

ISorter(std::less{});

printVector(ISorter.data());
```

Microsoft Visual Studio De...

```
[45,34,8,6,5,1,0,-2,-3,-100]
  [34,45,8,6,5,1,0,-2,-3,-100]
  [8,34,45,6,5,1,0,-2,-3,-100]
  [6,8,34,45,5,1,0,-2,-3,-100]
  [5,6,8,34,45,1,0,-2,-3,-100]
  [1,5,6,8,34,45,0,-2,-3,-100]
  [0,1,5,6,8,34,45,-2,-3,-100]
  [-2,0,1,5,6,8,34,45,-3,-100]
  [-3,-2,0,1,5,6,8,34,45,-100]
  [-100,-3,-2,0,1,5,6,8,34,45]
[-100,-3,-2,0,1,5,6,8,34,45]
  [-3,-100,-2,0,1,5,6,8,34,45]
  [-2,-3,-100,0,1,5,6,8,34,45]
  [0,-2,-3,-100,1,5,6,8,34,45]
  [1,0,-2,-3,-100,5,6,8,34,45]
  [5,1,0,-2,-3,-100,6,8,34,45]
  [6,5,1,0,-2,-3,-100,8,34,45]
  [8,6,5,1,0,-2,-3,-100,34,45]
  [34,8,6,5,1,0,-2,-3,-100,45]
  [45,34,8,6,5,1,0,-2,-3,-100]
[45,34,8,6,5,1,0,-2,-3,-100]
```


© Dr Markus Lumpe, 2023

TYPE DEDUCTION

- Before C++17, we always had to explicitly specify all template parameter types for all class templates. For example, we had to write:

```
std::vector<int> IData{ 45, 34, 8, 6, 5, 1, 0, -2, -3, -100 };
```

- This constraint has been relaxed in C++17. By using class template argument deduction, we can skip defining template arguments explicitly if the constructor is able to deduce all template parameters:

vector of int

```
std::vector IData{ 45, 34, 8, 6, 5, 1, 0, -2, -3, -100 };
```

- **Note**: Template parameters have to be unambiguously deducible.

© Dr Markus Lumpe, 2023

NOTE OF TEMPLATE PARAMETER DEDUCTION

The rules for template parameter deduction are rather complex. We also have to take in consideration default parameters (e.g., **typename** C = std::greater<T>) that allow us to skip explicit argument specification.

In addition, C++ allows for the specification of deduction guides to assist the compiler. (This feature is beyond the scope of COS30008.)

© Dr Markus Lumpe, 2023

TEMPLATE IMPLEMENTATION

- When using templates, the C++ compiler must have access not only the specification of a template class but also to all method implementations in order to properly instantiate the template.
- Templates are not to be confused with library classes. Templates are blueprint that have to be instantiated for each separate type application.
- Think of templates as special forms of macros.
- When defining a template class, we need to implement everything within the class definition, usually in a header file. Templates have no .cpp.

© Dr Markus Lumpe, 2023

INDEXER

© Dr Markus Lumpe, 2023

ALLOW ANY DATA TYPE TO BE INDEXED LIKE AN ARRAY

- In order to allow any data type to be indexed like an array, we have to define an overridden index operator [].
- Overloading [] may be useful
  - to check for index out of bounds, or
  - to allow for selectors other than integers.
- [] must return an L-value reference so that the indexed value can be used as L-value.
- We can enforce read-only semantics by decorating the the returned L-value reference as const.

© Dr Markus Lumpe, 2023

A BASIC INDEXER

```
template<typename T, size_t N = 20>
class BasicIndexer
{
private:
   T fElements[N];

public:

   BasicIndexer() noexcept :
      fElements()
   {}

   const size_t size() const noexcept
   {
      return N;
   }

   T& operator[]( size_t aIndex )
   {
      assert( aIndex < N );

      return fElements[aIndex];
   }
};
```

trivially copyable

We can use non type parameters, e.g., N = 20.

read-write L-value reference

debug mode test

© Dr Markus Lumpe, 2023

POPULATE BASIC INDEXER

```
BasicIndexer<int> IIndexer;

for ( size_t i = 0; i < IIndexer.size(); i++ )
{
    IIndexer[i] = static_cast<int>(i) + 1;
}

std::cout << "Indexer in reverse order:" << std::endl;

for ( size_t i = IIndexer.size(); i > 0; )
{
    std::cout << "\t" << IIndexer[--i] << std::endl;
}
```

Microsoft Visual St...

```
Indexer in reverse order:
        20
        19
        18
        17
        16
        15
        14
        13
        12
        11
        10
        9
        8
        7
        6
        5
        4
        3
        2
        1
```


© Dr Markus Lumpe, 2023

USING STRINGS AS INDICES

```
#include "BasicIndexer.h"

#include <string>

template<typename T, size_t N = 20>
class StringIndexer
{
private:
    BasicIndexer<T, N> fElements;

public:

    const size_t size() const noexcept
    {
        return fElements.size();
    }

    T& operator[]( const std::string& aKey )
    {
        size_t lIndex = 0;

        for ( size_t i = 0; i < aKey.size(); i++ )
        {
            lIndex = lIndex * 10 + (static_cast<size_t>(aKey[i]) - '0');
        }

        assert( lIndex < fElements.size() );

        return fElements[lIndex];
    }
};
```

trivially copyable delegate instance

forward request

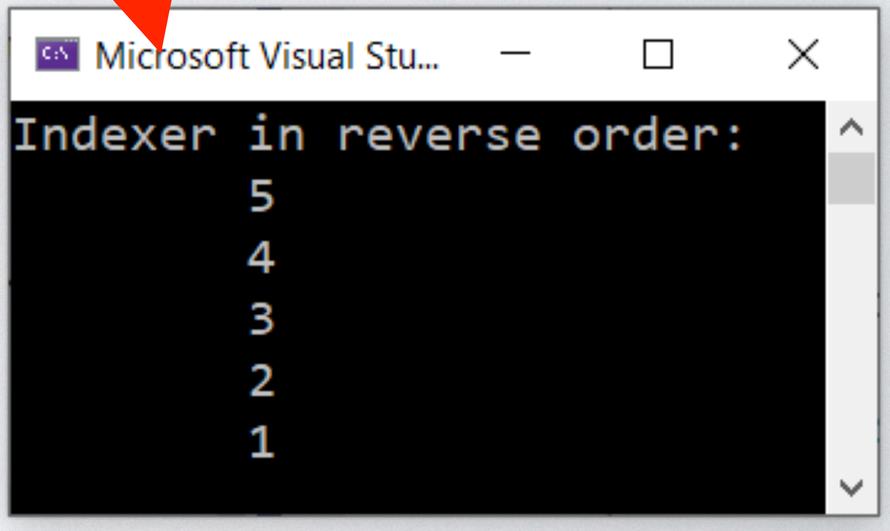
stoi

forward request

© Dr Markus Lumpe, 2023

POPULATE BASIC INDEXER

```
StringIndexer<int, 5> IIndexer;
std::string IKeys[] = { "0", "1", "2", "3", "4" };

for ( size_t i = 0; i < IIndexer.size(); i++ )
{
    IIndexer[IKeys[i]] = static_cast<int>(i) + 1;
}

std::cout << "Indexer in reverse order:" << std::endl;

for ( size_t i = IIndexer.size(); i > 0; )
{
    std::cout << "\t" << IIndexer[IKeys[--i]] << std::endl;
}
```



© Dr Markus Lumpe, 2023

ADDITIONAL FLEXIBILITY: LAMBDA EXPRESSIONS

© Dr Markus Lumpe, 2023

HARD-CODED CONVERSION

```
T& operator[]( const std::string& aKey )
{
    size_t lIndex = 0;

    for ( size_t i = 0; i < aKey.size(); i++ )
    {
        lIndex = lIndex * 10 + (static_cast<size_t>(aKey[i]) - '0');
    }

    return fElements[lIndex];
}
```

hard-coded stoi

- The string indexer uses a hard-coded conversion from string to size_t.
- This limits the application of the indexer.
- A better option is to pass a conversion function explicitly to the indexer, so that we can change the conversion procedure at runtime.

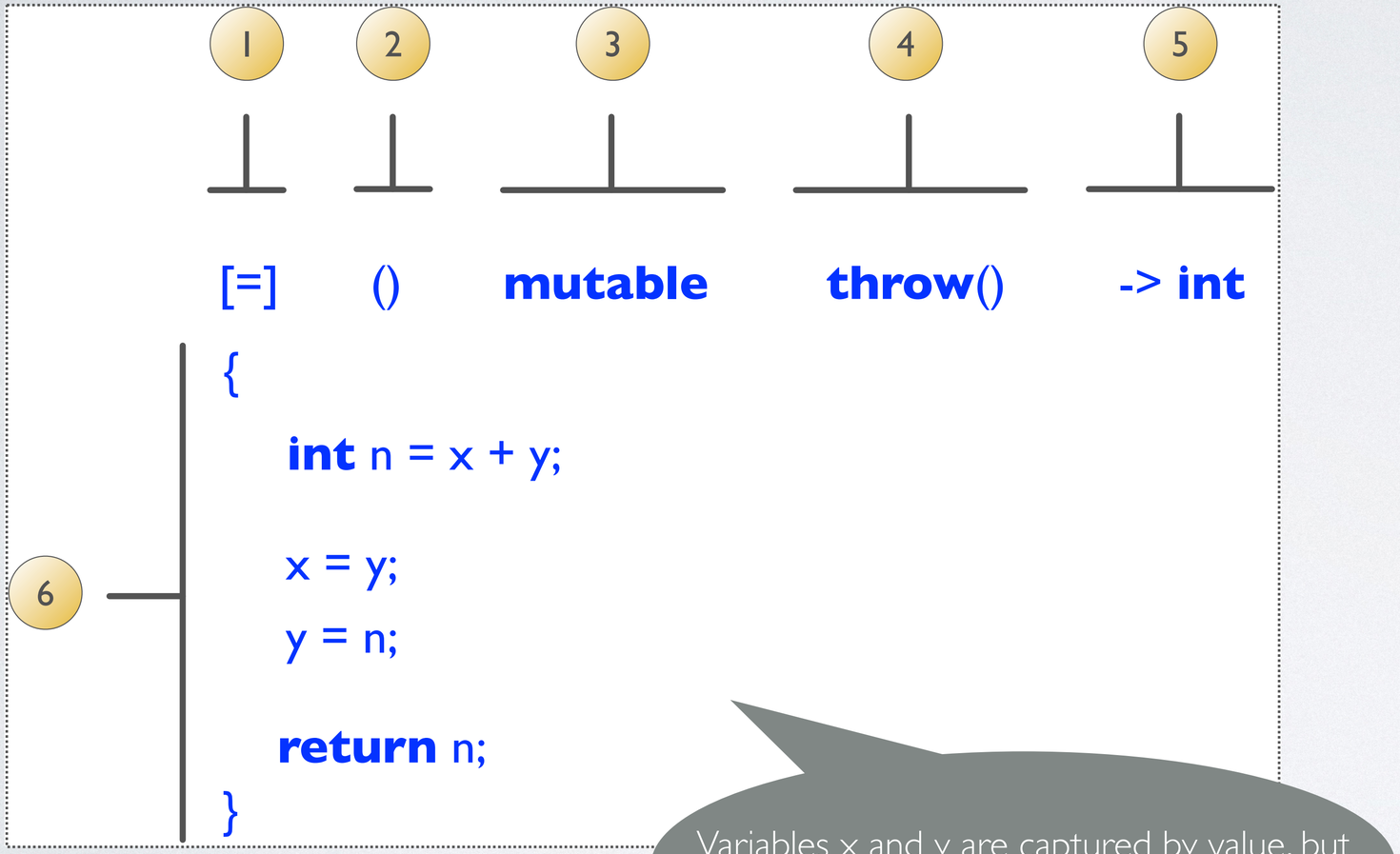
© Dr Markus Lumpe, 2023

C++ LAMBDA EXPRESSIONS

- Since C++11, lambda expressions are available in C++.
- A lambda expression, or just lambda, is an anonymous function object (closure) that represents a callable unit of code.
- Like any function, a lambda has a return type, a parameter list, and a function body.
- Unlike a function, lambdas may be defined inside a function.
- Note, C++ supports two kinds of callable objects: objects of classes that override the call operator (i.e., operator()) and lambda expressions.

© Dr Markus Lumpe, 2023

C++ LAMBDA



Variables x and y are captured by value, but can be altered within the body of lambda.

1. Capture clause
2. Parameter list, optional
3. Mutable specification, optional
4. Exception specification, optional
5. Trailing return type, optional
6. Lambda body

© Dr Markus Lumpe, 2023

C++ LAMBDA EXAMPLES

function declaration with lambda

```
auto f = [] { return 42; };
cout << f() << endl;        // prints 42
```

```
[] (const std::string& aLHS, const std::string& aRHS) { return aLHS.size() < aRHS.size() };
```

```
[lSize] (const std::string& aString) { return aString.size() < lSize; };
```

capture lSize

© Dr Markus Lumpe, 2023

LAMBDA CAPTURES

[]	Lambda does not use variables from the enclosing environment. Variables from the environment cannot be accessed.
[identifier list]	The variables listed in the comma-separated identifier list are captured by value and copied into the body of lambda. The lambda sees only stored values. Updates in the environment have not effect on lambda.
[&]	All variables in the environment are implicitly captured by reference. Updates in the environment affect lambda.
[=]	All variables in the environment are implicitly captured by value. Values are copied into the body of lambda. Updates in the environment have no effect on lambda.
[this]	Capture current object by reference.
[*this]	Capture current object by value (copy).
[&, identifier list]	Implicit capture by reference of all variables in the environment, except those that occur in identifier list. Identifier list must not contain &. Can contain ***this**.
[=, reference list]	Implicit capture by value (copied into the body of lambda) of all variables in the environment, except those that occur in identifier list. In reference list all names must be preceded by &.

© Dr Markus Lumpe, 2023

INDEXER WITH LAMBDA

```
...

#include <functional>

template<typename T, size_t N = 20>
class StringIndexerLambda
{
    ...
public:
    ...

  T& operator[]( const std::string& aKey ) { return item( aKey ); }

  T& item( const std::string& aKey,
        std::function<size_t( const std::string& )> aMap = [] ( const std::string& aKey )
        {
          size_t lIndex = 0;

          for ( size_t i = 0; i < aKey.size(); i++ ) { lIndex = lIndex * 10 + (static_cast<size_t>(aKey[i]) - '0'); }

          return lIndex;
        } )
  {
    size_t lIndex = aMap( aKey );

    assert( lIndex < fElements.size() );

    return fElements[lIndex];
  }
};
```

new item function, operator[] expects one argument only

© Dr Markus Lumpe, 2023

STD::FUNCTION

```
template<typename R, typename… Args>
class function<R(Args…)>
```

- Technically, a variable that stores a lambda is a function pointer. However, function pointers may lead to unreadable specifications or worse.
- Since C++11, C++ offers a function wrapper std::function for this purpose.
- Technically, std::function is a variadic template (it takes a variable number of arguments) that allows us to capture any function signature.

© Dr Markus Lumpe, 2023

AUTO

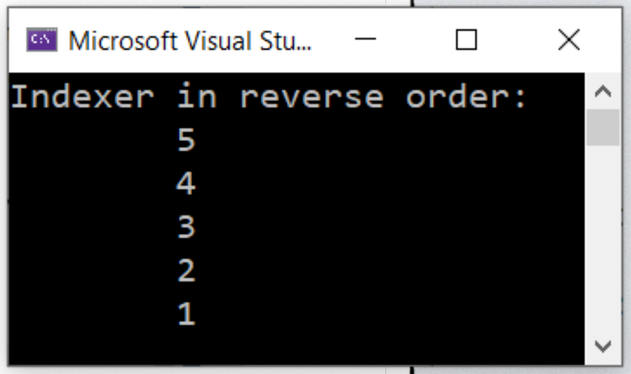
```
auto f = [] (const string& aLHS, const string& aRHS) { return aLHS.size() < aRHS.size() };
```

- C++ has also introduced auto typing, that is, we can declare variables using auto as type name:
- Using auto saves typing and prevents correctness and performance issues when dealing with complex types.
- Automatic type deduction via auto is no free lunch. The programmer has to guide the compiler to produce the right answer. Failing to do so, can result in a wrong type altogether.
- Unfortunately, auto type specifies cannot be used in function parameters. For parameters we need to specify the actual type. It is available in C++20.

© Dr Markus Lumpe, 2023

INDEXER WITH BINARY KEYS

```
StringIndexerLambda<int, 5> IIndexer;
std::string IBinaryKeys[] = { "000", "001", "010", "011", "100" };

auto IMapBinary = [] ( const std::string& aNumber )
{
    size_t IIndex = 0;

    for ( size_t i = 0; i < aNumber.size(); i++ ) { IIndex = (IIndex << 1) + (static_cast<size_t>(aNumber[i]) - '0'); }

    return IIndex;
};

for ( size_t i = 0; i < IIndexer.size(); i++ )
{
    IIndexer.item( IBinaryKeys[i], IMapBinary ) = static_cast<int>(i) + 1;
}

std::cout << "Indexer in reverse order:" << std::endl;

for ( size_t i = IIndexer.size(); i > 0; )
{
    std::cout << "\t" << IIndexer.item( IBinaryKeys[--i], IMapBinary ) << std::endl;
}
```



© Dr. Markus Lumpe, 2023

ITERATORS

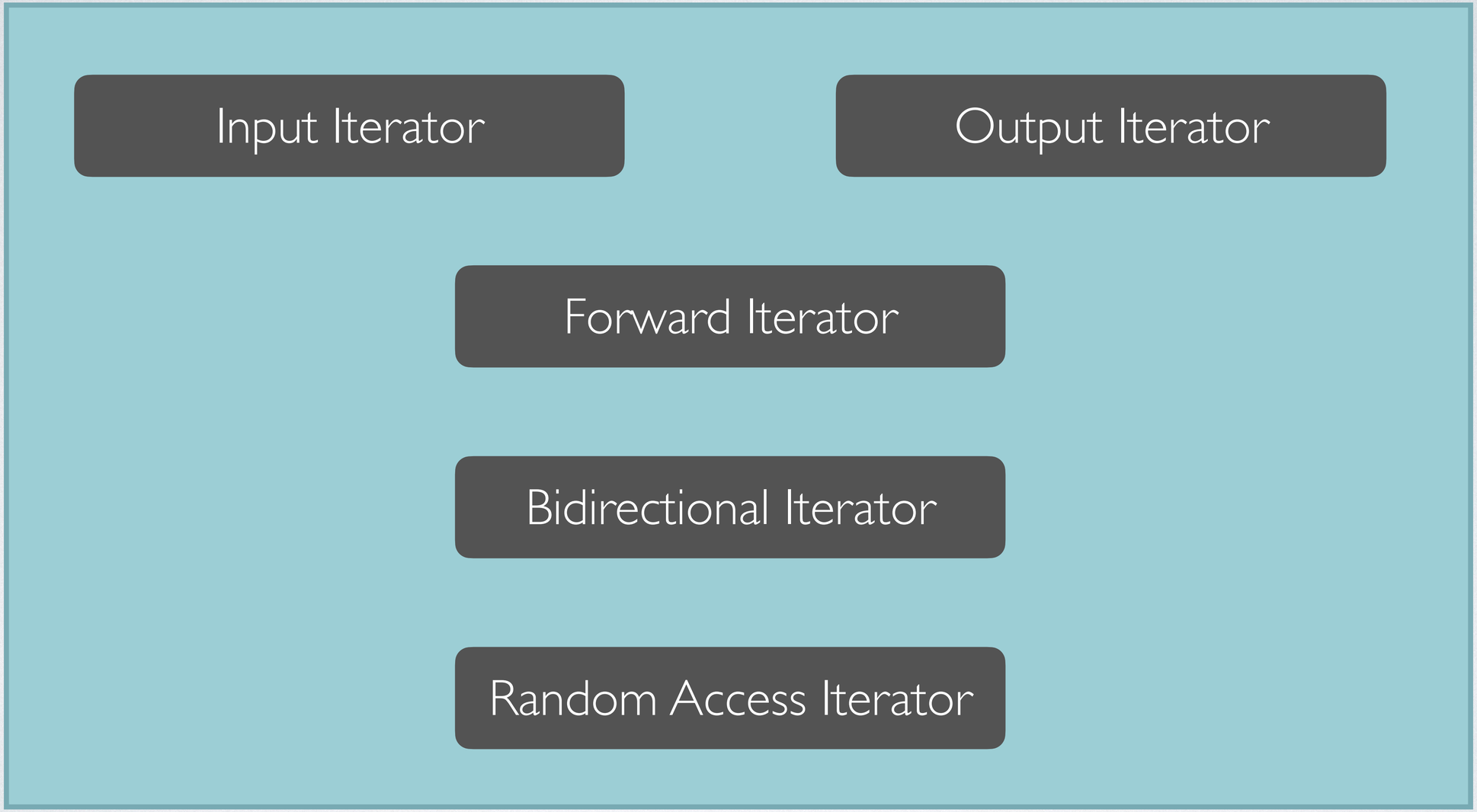
© Dr Markus Lumpe, 2023

SYSTEMATIC ITERATION OVER ELEMENTS: ITERATORS

- We can use a loop statement and a loop counter to visit all elements of an array in sequence.
- However, not all data types are arrays and simple indexing may not suffice.
- Iterators offer programmers a suitable alternative to define iteration in a data type agnostic way.
- Conceptually, iterators are objects that implement the necessary infrastructure to iterate over elements of a sequence. They do this via a common interface.
- The C++ ecosystem has popularized and uses five types of iterators. Iterator objects have the look-and-feel of pointers, and advancing an iterator means to increment/decrement a pointer-like object.

© Dr Markus Lumpe, 2023

C++ ITERATORS



© Dr Markus Lumpe, 2023

ABILITIES OF ITERATORS

Category	Abilities	Providers
Input Iterator	read forward	stream
Output Iterator	write forward	ostream, inserter
Forward Iterator	read and write forward	
Bidirectional Iterator	+read and write backward	STL containers
Random Access Iterator	+read and write with random access	STL containers

© Dr Markus Lumpe, 2023

INPUT ITERATOR

Expression	Effect
*iter	provides read access to element at current iterator position
iter->member	provides read access to a member of the element at current iterator position
++iter	steps forward (returns new position; ITER& **operator++()**)
iter++	steps forward (returns old position; ITER **operator++(int)**)
iter1 == iter2	iterators iter1 and iter2 positioned at same element
iter1 != iter2	iterators iter1 and iter2 not positioned at same element

© Dr Markus Lumpe, 2023

OUTPUT ITERATOR

Expression	Effect
*iter = value	provides write access to element at current iterator position
++iter	steps forward (returns new position; ITER& **operator++()**)
iter++	steps forward (returns old position; ITER **operator++(int)**)

An output iterator is like a "black hole".

© Dr Markus Lumpe, 2023

FORWARD ITERATOR

Expression	Effect
*iter	provides read access to element at current iterator position
iter->member	provides read access to a member of the element at current iterator position
++iter	steps forward (returns new position; ITER& **operator++()**)
iter++	steps forward (returns old position; ITER **operator++(int)**)
iter1 == iter2	iterators iter1 and iter2 positioned at same element
iter1 != iter2	iterators iter1 and iter2 not positioned at same element
iter1 = iter2	iterator assignment

© Dr Markus Lumpe, 2023

BIDIRECTIONAL ITERATOR

Expression	Effect
*iter	provides read access to element at current iterator position
iter->member	provides read access to a member of the element at current iterator position
++iter	steps forward (returns new position; ITER& **operator++()**)
iter++	steps forward (returns old position; ITER **operator++(int)**)
—iter	steps backward (returns new position; ITER& **operator—()**)
iter—	steps backward (returns old position; ITER **operator—(int)**)
iter1 == iter2	iterators iter1 and iter2 positioned at same element
iter1 != iter2	iterators iter1 and iter2 not positioned at same element
iter1 = iter2	iterator assignment

© Dr Markus Lumpe, 2023

RANDOM ACCESS ITERATOR

- Refinement of bidirectional iterator

Expression	Effect
iter[n]	provides read access to element at position n
iter += n	steps n elements forward (or backward)
iter -= n	steps n elements backward (or forward)
n+iter	returns iterator positioned n elements forward
n-iter	returns iterator position n elements backward
iter1 - iter2	returns distance between iter1 and iter2

© Dr Markus Lumpe, 2023

A BASIC INPUT-OUTPUT ITERATOR

```
#pragma once

template<typename T, size_t N = 20>
class BasicIterator
{
private:
   T fElements[N];                                                    // underlying collection
   size_t fPosition;                                                  // iterator position

public:

   using Iterator = BasicIterator<T,N>;                               // type alias

   BasicIterator() noexcept;

   T& operator*() noexcept;                                           // dereference

   Iterator& operator++() noexcept;                                   // prefix increment
   Iterator operator++( int ) noexcept;                               // postfix increment

   bool operator==( const Iterator& aOther ) const noexcept; // iter1 == iter2
   bool operator!=( const Iterator& aOther ) const noexcept;  // iter1 != iter2

   Iterator begin() const noexcept;                                   // return fresh iterator positioned at start
   Iterator end() const noexcept;                                     // return fresh iterator position after last
};
```

ITERATOR CONSTRUCTOR

BasicIterator is trivially copyable

```
BasicIterator() noexcept :
    fElements(),
    fPosition(0)
{}
```

initialize member variables

© Dr Markus Lumpe, 2023

DEREFERENCE OPERATOR - ELEMENT ACCESS

```
T& operator*() noexcept
{
    return fElements[fPosition];
}
```

- The deference operator returns a read-write reference to the element the iterator is currently positioned on. BasicIterator is an input-output iterator.
- No range check is required. The **operator***() should only be called if the iterator has not yet reached the end of the underlying collection.

© Dr Markus Lumpe, 2023

PREFIX INCREMENT

- The prefix increment operator advances the iterator and returns a reference to the current iterator object:

```
Iterator& operator++() noexcept
{
    fPosition++;

    return *this;
}
```

return a reference to current object

© Dr Markus Lumpe, 2023

POSTFIX INCREMENT

- The postfix increment operator advances the iterator and returns a copy of the old iterator:

```
Iterator operator++( int ) noexcept
{
    Iterator old = *this;

    ++(*this);        // forward to prefix

    return old;
}
```

dummy argument; used by compiler

return a copy of old iterator (old position)

© Dr Markus Lumpe, 2023

EQUIVALENCE

```
bool operator==( const Iterator& aOther ) const noexcept
{
    // domain of == is position
    return fPosition == aOther.fPosition;
}
```

- Two iterators are equal if and only if they refer to the same element (this may require considering the context of ==):
  - The domain of equivalence only considers the iterator positions. This design decision worth in the current context.
  - The tests need to be consistent with the definitions in the auxiliary methods.

© Dr Markus Lumpe, 2023

INEQUALITY

```
bool operator!=( const Iterator& aOther ) const noexcept
{
    return !(*this == aOther);
}
```

We implement != in terms of ==.

© Dr Markus Lumpe, 2023

AUXILIARY METHODS

```
Iterator begin() const noexcept
{
    Iterator Result = *this;

    Result.fPosition = 0;

    return Result;
}

Iterator end() const noexcept
{
    Iterator Result = *this;

    Result.fPosition = N;

    return Result;
}
```

- The methods begin() and end() return fresh iterators positioned at the first element and after last element, respectively.
- The names and implementation of these auxiliary methods follow standard practices. The compiler may look for them when you use a range-loop.

© Dr Markus Lumpe, 2023

ITERATOR APPLICATION

```
BasicIterator<int> Iter1;
int i = 1;

for ( ; Iter1 != Iter1.end(); Iter1++ )
{
    *Iter1 = i * i;
    i++;
}

std::cout << "Iterator values:" << std::endl;

for ( BasicIterator<int> Iter2 = Iter1.begin(); Iter2 != Iter2.end(); Iter2++ )
{
    std::cout << "\t" << *Iter2 << std::endl;
}
```

Mic...

```
Iterator values:
        1
        4
        9
        16
        25
        36
        49
        64
        81
        100
        121
        144
        169
        196
        225
        256
        289
        324
        361
        400
```


© Dr Markus Lumpe, 2023

C++ RANGE-LOOP

- The traditional for statement in C++ reads:

for (init-statement; condition; expression) statement

- This form uses explicit loop variables, conditions, and increments over loop variables.
- C++11 introduces a simpler form, called range for statement, to iterate through the elements of a container or other sequence:

for (declaration : expression) statement

- This form is called range-loop, and expression must denote a sequence used to define a range-loop variable in declaration advanced in each step of the iteration.

© Dr Markus Lumpe, 2023

UNDERSTANDING THE RANGE LOOP

for (declaration : expression) statement

```
auto&& __range = expression;                    // C++ forwarding (move)
for ( auto __begin = begin-expression,          // begin() of expression object
          __end = end-expression;               // end() of expression object
      __begin != __end;                         // range-loop condition
      ++__begin )                               // advance iterator
{
  declaration = *__begin;                       // set loop variable to current element
  statement;                                    // range-loop body
}
```


© Dr Markus Lumpe, 2023

USING THE RANGE-LOOP

```
int IArray[] = {-100,-3,-2};

std::cout << "Copy: " << std::endl;

for ( int i : IArray )
{
	std::cout << std::hex << &i << "," << sizeof(i) << std::endl;
}

std::cout << "Const reference: " << std::endl;

for ( const int& i : IArray )
{
	std::cout << std::hex << &i << "," << sizeof(i) << std::endl;
}

std::cout << "Auto copy: " << std::endl;

for ( auto i : IArray )
{
	std::cout << std::hex << &i << "," << sizeof(i) << std::endl;
}

std::cout << "Const auto reference: " << std::endl;

for ( const auto& i : IArray )
{
	std::cout << std::hex << &i << "," << sizeof(i) << std::endl;
}
```

- Case 1: read-write variable
- Case 2: constant reference
- Case 3: auto variable
- Case 4: constant auto reference

© Dr Markus Lumpe, 2023

RANGE-LOOP BEHAVIOR

```
int IArray[] = {-100,-3,-2};

std::cout << "Copy: " << std::endl;

for ( int i : IArray )
{
    std::cout << std::hex << &i << "," << std::dec << sizeof(i) << ", i = " << i << std::endl;
}

std::cout << "Const reference: " << std::endl;

for ( const int& i : IArray )
{
    std::cout << std::hex << &i << "," << std::dec << sizeof(i) << ", i = " << i  << std::endl;
}

std::cout << "Auto copy: " << std::endl;

for ( auto i : IArray )
{
    std::cout << std::hex << &i << "," << std::dec << sizeof(i) << ", i = " << i  << std::endl;
}

std::cout << "Const auto reference: " << std::endl;

for ( const auto& i : IArray )
{
    std::cout << std::hex << &i << "," << std::dec << sizeof(i) << ", i = " << i  << std::endl;
}
```

Microsoft Visual Studio De...

```
Copy:
0000004D15FAF424, 4, i = -100
0000004D15FAF424, 4, i = -3
0000004D15FAF424, 4, i = -2
Const reference:
0000004D15FAF398, 4, i = -100
0000004D15FAF39C, 4, i = -3
0000004D15FAF3A0, 4, i = -2
Auto copy:
0000004D15FAF524, 4, i = -100
0000004D15FAF524, 4, i = -3
0000004D15FAF524, 4, i = -2
Const auto reference:
0000004D15FAF398, 4, i = -100
0000004D15FAF39C, 4, i = -3
0000004D15FAF3A0, 4, i = -2
```

© Dr Markus Lumpe, 2023

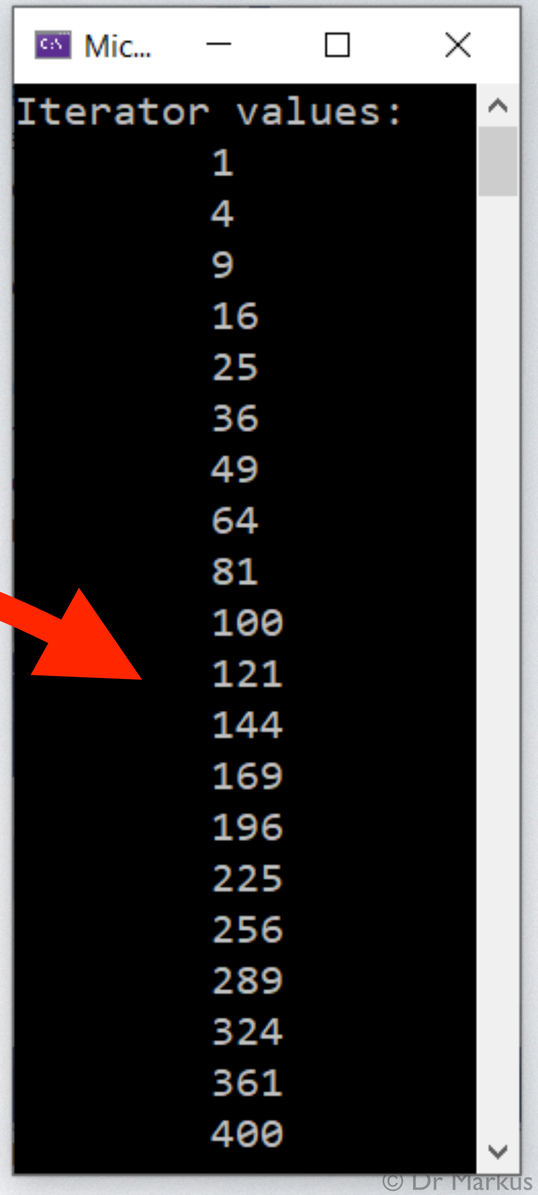
WHICH RANGE-LOOP PATTERN TO USE

- In a range-loop always use a reference variable. This avoids unnecessary copies.
- Prefer auto to explicit type declarations. Iterator types can be quite complex and hard to express. Using auto - automatic type deduction - simplifies things dramatically.
- You still need to understand what type deduction means and what the results are. Code becomes less readable, as fewer explicit detail is available.

© Dr Markus Lumpe, 2023

ITERATOR APPLICATION WITH RANGE-LOOP

```
BasicIterator<int> Iter1;
int i = 1;

// populate underlying collection
for ( ; Iter1 != Iter1.end(); Iter1++ )
{
    *Iter1 = i * i;
    i++;
}

std::cout << "Iterator values:" << std::endl;

// fetch elements from underlying collection (copy of Iter1)
for ( const auto& item : Iter1 )
{
    std::cout << "\t" << item << std::endl;
}
```



© Dr Markus Lumpe, 2023

NOTE ON AUTO

- Auto saves typing and prevents correctness and performance issues when dealing with complex types.
- Automatic type deduction via auto is no free lunch. The programmer has to guide the compiler to produce the right answer. Failing to do so, can result in a wrong type altogether.
- The use of auto can hamper program comprehension. We may have to perform an in-depth study of the code base to understand what type we are dealing with and which methods can be safely invoked on a given object.
- Using auto can produce undefined behavior. The code still compiles fine, but there is a chance the result will be unpredictable. In this case an explicit type specification is required.

© Dr Markus Lumpe, 2023

LIMITATIONS OF BASICITERATOR

- Equivalence of BasicIterator is solely defined in terms of iterator position. The values of the underlying collection are not in the domain of equivalence. It work in the current scenario, but may not be transferrable to other applications.

- It is possible to share the underlying collection via a reference data member. However, reference data members can become invalid when the lifetime of the object holding the reference exceeds that of the referenced object. The C++ Core Guidelines explicitly discourage the use of reference members.

- If we require shared ownership, then we need to use std::shared_ptr, which provides us with a reference counting. However, the pervasive use of std::shared_ptr has a cost.

- Unfortunately, there exist no enforceable pattern for the reliable representation of shared ownership and, hence, a standard way to handle the underlying collection in an iterator.

© Dr Markus Lumpe, 2023

RECURSION

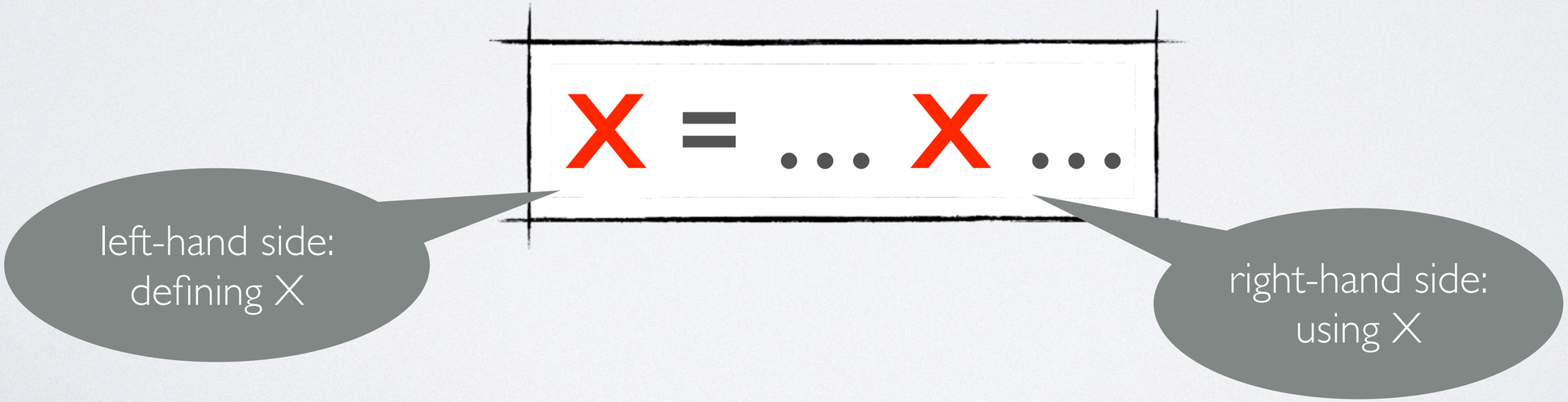
- If a procedure contains within its body calls to itself, then this procedure is said to be recursively defined.
- This approach of program specification is called recursion and is found not only in programming.
- If we the define a procedure recursively, then there must exist at least one sub-problem that can be solved directly, that is without calling the procedure again.
- A recursively defined procedure must always contain a directly solvable sub-problem. Otherwise, this procedure does not terminate.

© Dr Markus Lumpe, 2023

PROBLEM-SOLVING WITH RECURSION

- Recursion is an important problem-solving technique in which a given problem is reduced to smaller instances of the same problem.

- The general structure of a recursive definition is



© Dr Markus Lumpe, 2023

BASIC RECURSIVE PROBLEMS

© Dr Markus Lumpe, 2023

FIBONACCI (PROGRAMMATIC REPRESENTATION)

- The Fibonacci numbers are the following sequence of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

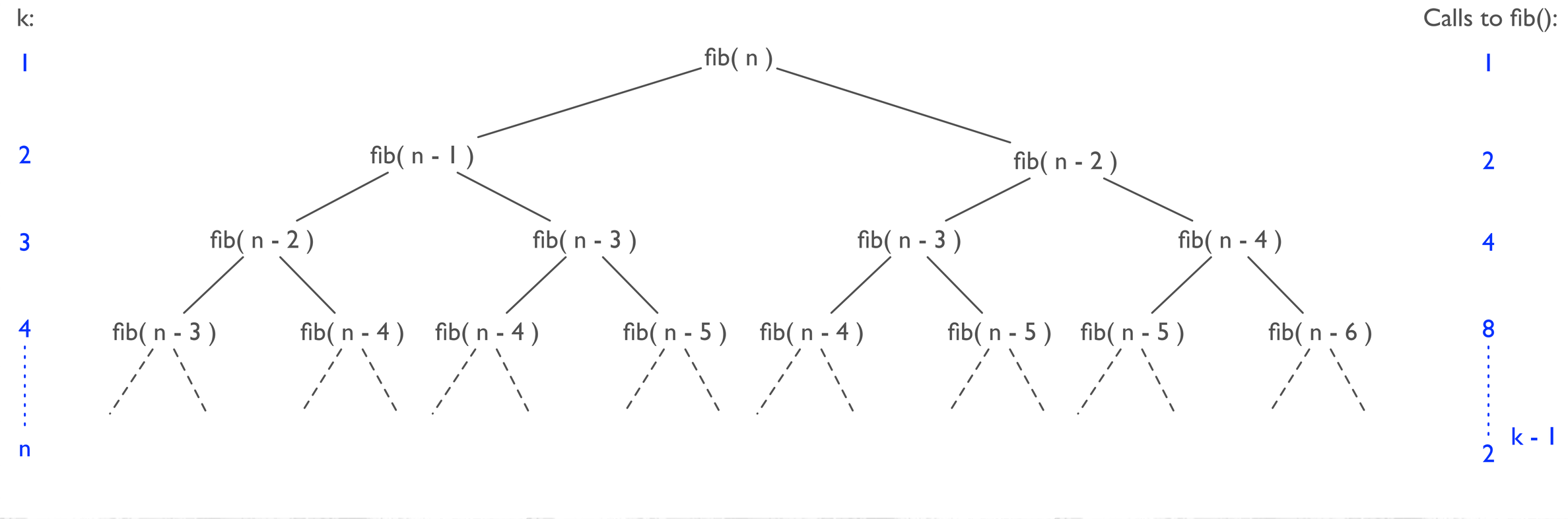
- In mathematical terms, the sequence F(n) of Fibonacci numbers is defined recursively as follows:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-i) + F(n-2)$$

```
size_t fib( size_t n ) noexcept
{
    if ( n <= 1 )
        return n;
    else
        return fib( n - 1 ) + fib( n - 2 );
}
```


© Dr Markus Lumpe, 2023

UNFOLDING RECURSIVE FIBONACCI



© Dr Markus Lumpe, 2023

FACTORIAL

- The factorial for positive integers is

$$n! = n * (n - 1) * \ldots * 1$$

- The recursive definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

© Dr Markus Lumpe, 2023

FACTORIAL CALCULATION

- The recursive definition tells us exactly how to calculate a factorial:

$4! = 4 * 3!$	Recursive step: n=4
$= 4 * (3 * 2!)$	Recursive step: n=3
$= 4 * (3 * (2 * 1!))$	Recursive step: n=2
$= 4 * (3 * (2 * (1 * 0!)))$	Recursive step: n=1
$= 4 * (3 * (2 * (1 * \mathbf{1})))$	Stop condition: n=0
$= 4 * (3 * (2 * \mathbf{1}))$	
$= 4 * (3 * \mathbf{2})$	
$= 4 * \mathbf{6}$	
$= \mathbf{24}$	

© Dr Markus Lumpe, 2023

FACTORIAL IMPLEMENTATION

```
size_t factorial( size_t n ) noexcept
{
    if ( n == 0 )
        return 1;
    else
        return n * factorial( n - 1 );
}

int main()
{
    std::cout << "6! = "  << factorial( 6 ) << std::endl;

    return 0;
}
```


© Dr Markus Lumpe, 2023

TAIL RECURSION

- A function is called tail-recursive if it ends in a recursive call that does not build-up any deferred operations.

```
long gcd( long x, long y ) noexcept
{
    if ( y == 0 )
        return x;
    else
        return gcd( y, x % y );
}
```

```
gcd( 1246, 234 ):
    gcd( 234, 76 )
    gcd( 76, 6 )
    gcd( 6, 4 )
    gcd( 4, 2 )
    gcd( 2, 0 )
    2
```


© Dr Markus Lumpe, 2023

TAIL RECURSION TO LOOP

- A tail-recursive function can be converted into a loop.

```
long gcd( long x, long y ) noexcept
{
    if ( y == 0 )
        return x;
    else
        return gcd( y, x % y );
}
```

```
long gcdLoop( long x, long y ) noexcept
{
    while ( y != 0 )
    {
        long temp  = x;
        x = y;
        y = temp % y;
    }

    return x;
}
```


© Dr Markus Lumpe, 2023

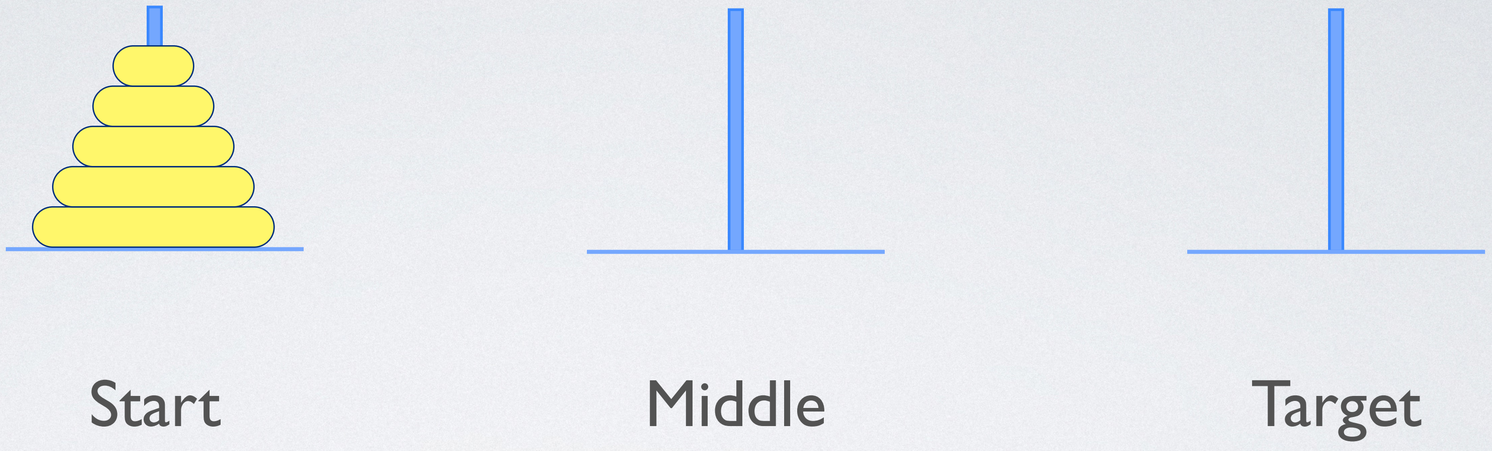
TOWERS OF HANOI

- **Problem**:
  - Move disks from a start peg to a target peg using a middle peg.
- **Challenge**:
  - All disks have a unique size and at no time must a bigger disk be placed on top of a smaller one.

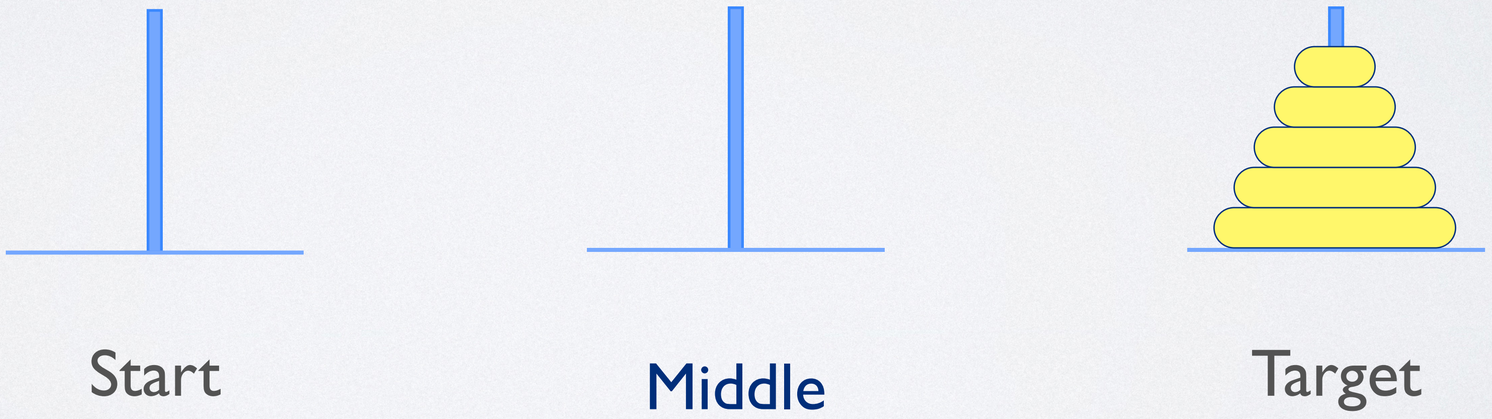
© Dr Markus Lumpe, 2023

TOWERS OF HANOI: CONFIGURATION

Start:



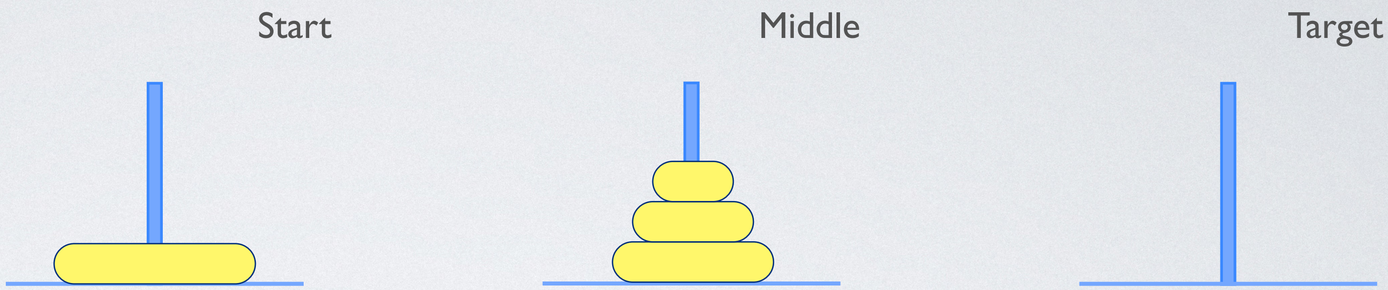
Finish:



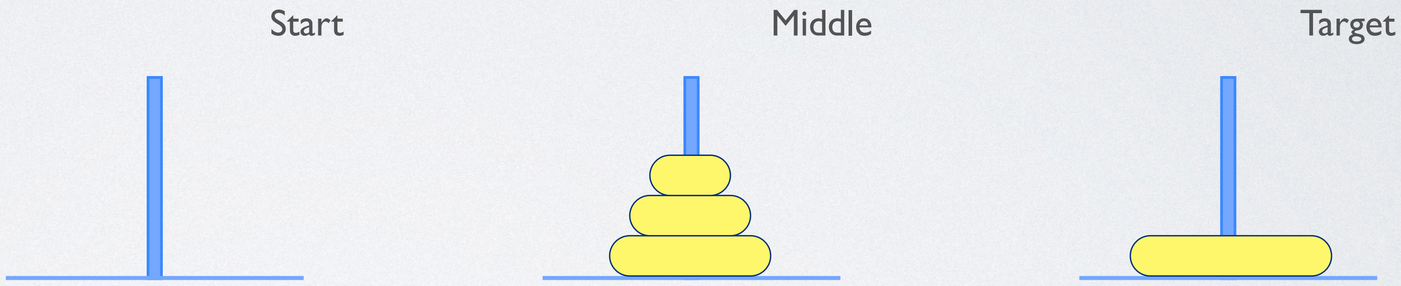
© Dr Markus Lumpe, 2023

RECURSIVE SOLUTION

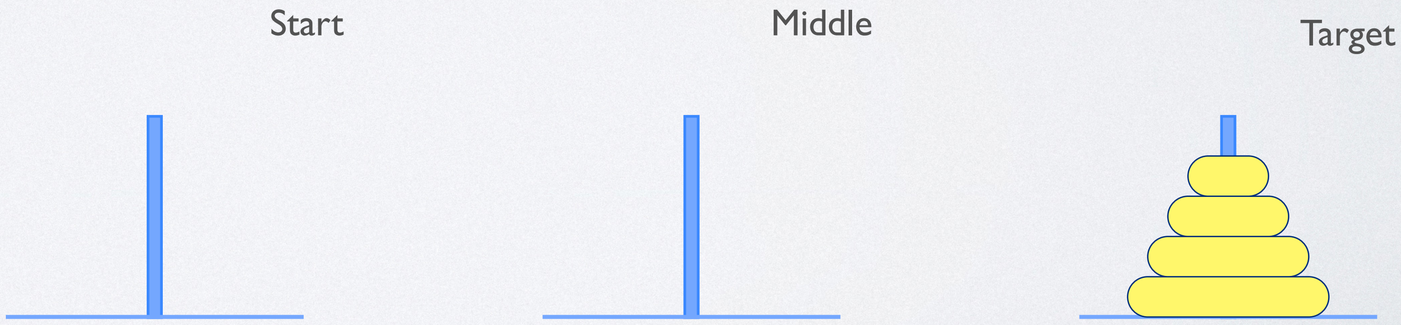
1. Move n-1 disks from Start to Middle:



2. Move 1 disk from Start to Target:

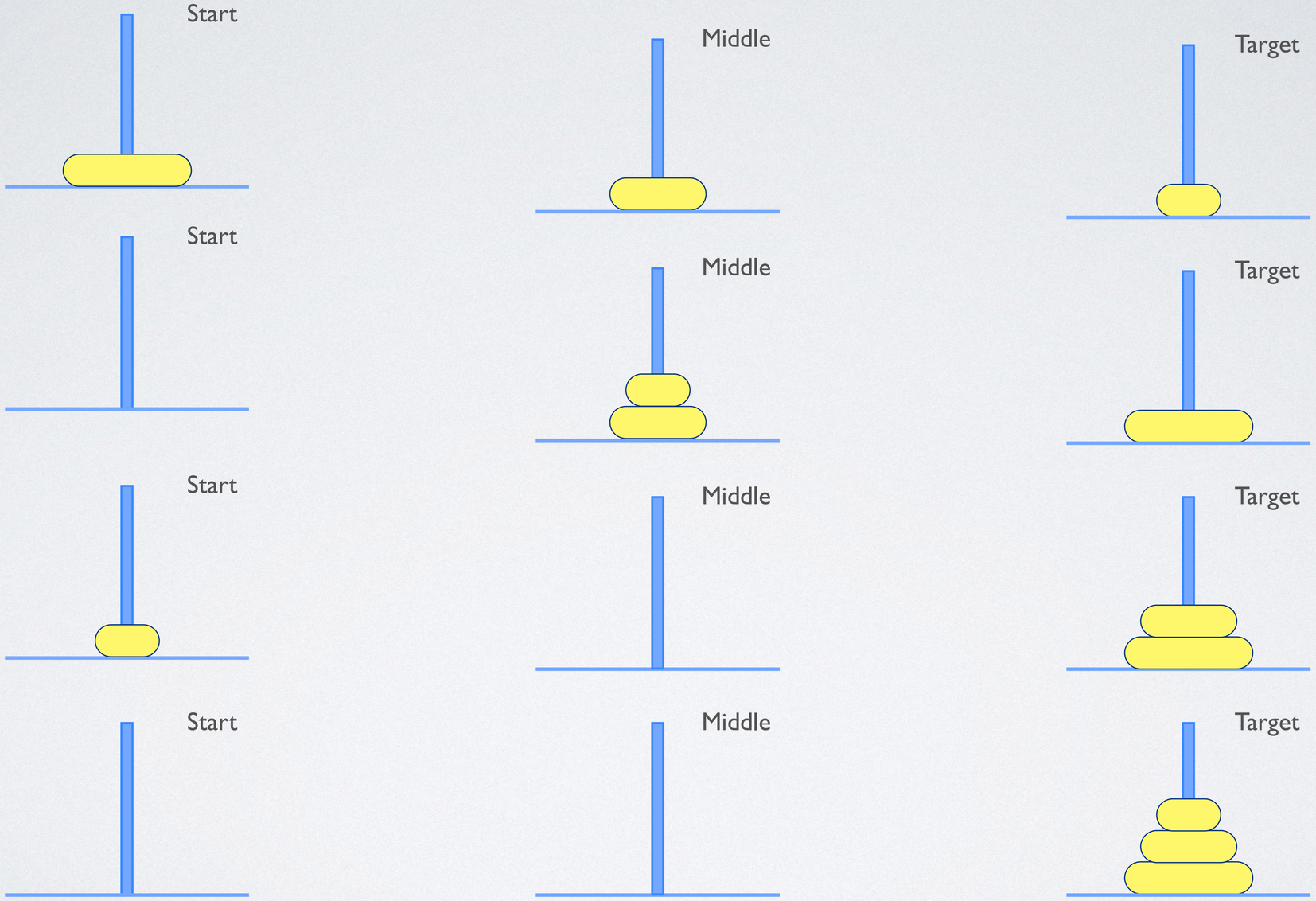


3. Move n-1 disks from Middle to Target:



© Dr Markus Lumpe, 2023

RECURSIVE SOLUTION: INTERMEDIATE



© Dr Markus Lumpe, 2023

TOWERS OF HANOI: RECURSIVE PROCEDURE

```
void doTowersOfHanoi( size_t n,
                      const std::string& aStart,
                      const std::string& aMiddle,
                      const std::string& aTarget
                     ) noexcept
{
   if ( n > 0 )
   {
      doTowersOfHanoi( n - 1, aStart, aMiddle, aTarget );

      std::cout << "Move disk " << n << " from " << aStart
                << " to " << aTarget << std::endl;

      doTowersOfHanoi( n - 1, aMiddle, aTarget, aStart );
   }
}

int main()
{
   doTowersOfHanoi( 3, "Start", "Middle", "Target" );

   return 0;
}
```

Microsoft Visual Studio Debug Cons...

```
Move disk 1 from Start to Target
Move disk 2 from Start to Target
Move disk 1 from Middle to Start
Move disk 3 from Start to Target
Move disk 1 from Middle to Start
Move disk 2 from Middle to Start
Move disk 1 from Target to Middle
```

© Dr Markus Lumpe, 2023

MERGE SORT

- Merge sort is a classical example of an algorithm based on the divide-and-conquer design strategy.

- Given a list L[low:high], let x be any index between low and high. Further, let A, B, and C denote the sublists A=L[low:x], B=L[x+1:high], and C=L[low:high], respectively. The problem of sorting C can be solved by first sorting A (recursively), then sorting B (recursively), and finally calling the procedure Merge, which merges the sorted lists A and B to obtain a sorted list C.

© Dr Markus Lumpe, 2023

PROCEDURE MERGE SORT

```
procedure MergeSort( L[0:n-1], low, high )
begin
  if low < high then
    mid ← low + (high - low)/2          // prevent overflow bug
    MergeSort( L[0:n-1], low, mid )
    MergeSort( L[0:n-1], mid+1, high )
    Merge( L[0:n-1], low, mid, high )
  endif
end
```


© Dr Markus Lumpe, 2023

PROCEDURE MERGE (DECREASING ORDER)

```
procedure Merge( L[0:n-1], low, mid, high )
begin
  for k ← low to mid do
    Temp[k] ← L[k]                                  // copy left-to-right
  endFor
  for k ← mid to high do
    Temp[(high - k) + mid] ← L[k+1]                 // copy right-to-left (prevent overflow)
  endFor
  CurPos1 ← low; CurPos2 ← high                     // CurPos1 move right, CurPos2 move left
  for k ← low to high do
    if Temp[CurPos1] < Temp[CurPos2] then
      L[k] ← Temp[CurPos1]; CurPos1 ← CurPos1 + 1
    else
      L[k] ← Temp[CurPos2]; CurPos2 ← CurPos2 - 1
    endIf
  endFor
end
```


s Lumpe, 2023

A POSSIBLE SOLUTION: MERGESORTER

```
#include <functional>

template<typename T>
class MergeSorter
{
private:
    using Comparator = std::function<bool(T,T)>;

    T* fBuffer;
    T* fMergeBuffer;
    Comparator fComparator;

    void doMergeSort( size_t aLow, size_t aHigh ) noexcept;
    void doMerge( size_t aLow, size_t aMid, size_t aHigh ) noexcept;

public:
    MergeSorter() noexcept;

    template<typename C = std::greater<T>>
    void operator()( T aArray[], size_t aSize, C alsOutOfOrder = C{} );
};
```

type of relational operator

parameterized over ordering relation

© Dr Markus Lumpe, 2023

USING MERGE SORT

```
int IArray[] = { 45, 34, 8, 6, 5, 1, 0, -2, -3, -100 };
const size_t ISize = sizeof(IArray)/sizeof(int);
MergeSorter<int> ISorter{};

printArray( IArray, ISize );

ISorter( IArray, ISize );

printArray( IArray, ISize );

ISorter( IArray, ISize, std::less{} );

printArray( IArray, ISize );
```

Microsoft Visual Studio Debug Console

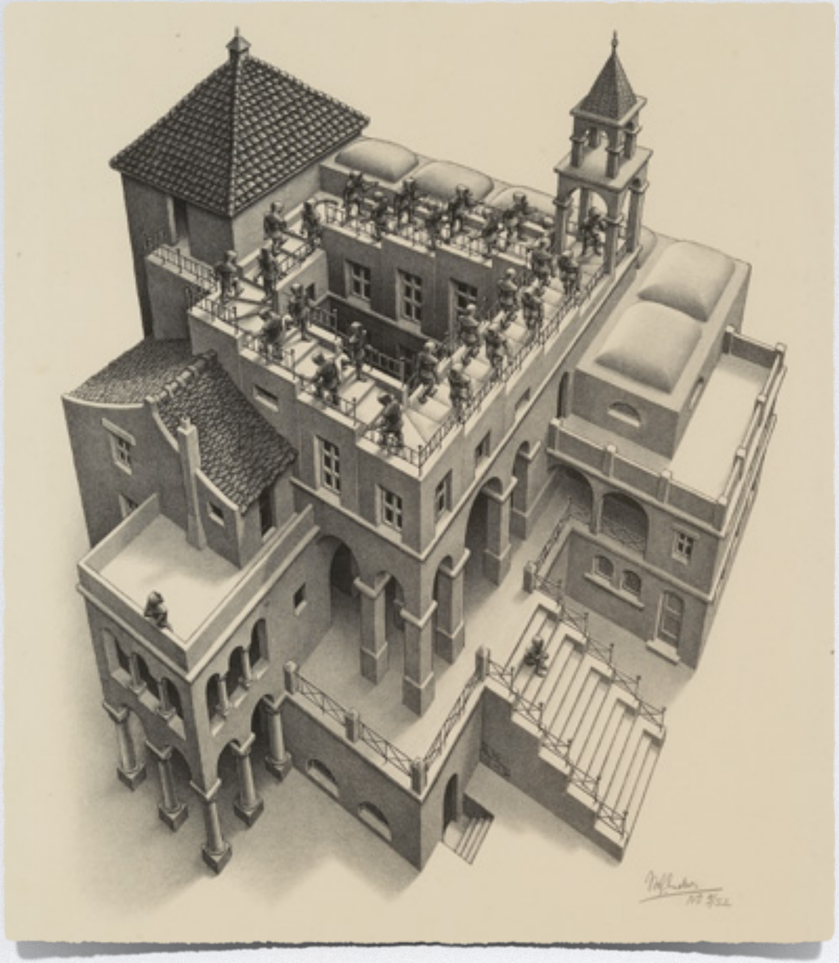
```
[45,34,8,6,5,1,0,-2,-3,-100]
  [45,34,8,6,5,1,0,-2,-3,-100]
    [45,34,8,6,5]
      [45,34,8]
        [45,34]
        [34,45]
      [8,34,45]
      [6,5]
      [5,6]
    [5,6,8,34,45]
    [1,0,-2,-3,-100]
      [1,0,-2]
        [1,0]
        [0,1]
      [-2,0,1]
      [-3,-100]
      [-100,-3]
    [-100,-3,-2,0,1]
  [-100,-3,-2,0,1,5,6,8,34,45]
[-100,-3,-2,0,1,5,6,8,34,45]
  [-100,-3,-2,0,1,5,6,8,34,45]
    [-100,-3,-2,0,1]
      [-100,-3,-2]
        [-100,-3]
        [-3,-100]
      [-2,-3,-100]
      [0,1]
      [1,0]
    [1,0,-2,-3,-100]
    [5,6,8,34,45]
      [5,6,8]
        [5,6]
        [6,5]
      [8,6,5]
      [34,45]
      [45,34]
    [45,34,8,6,5]
  [45,34,8,6,5,1,0,-2,-3,-100]
[45,34,8,6,5,1,0,-2,-3,-100]
```

Lumpe, 2023

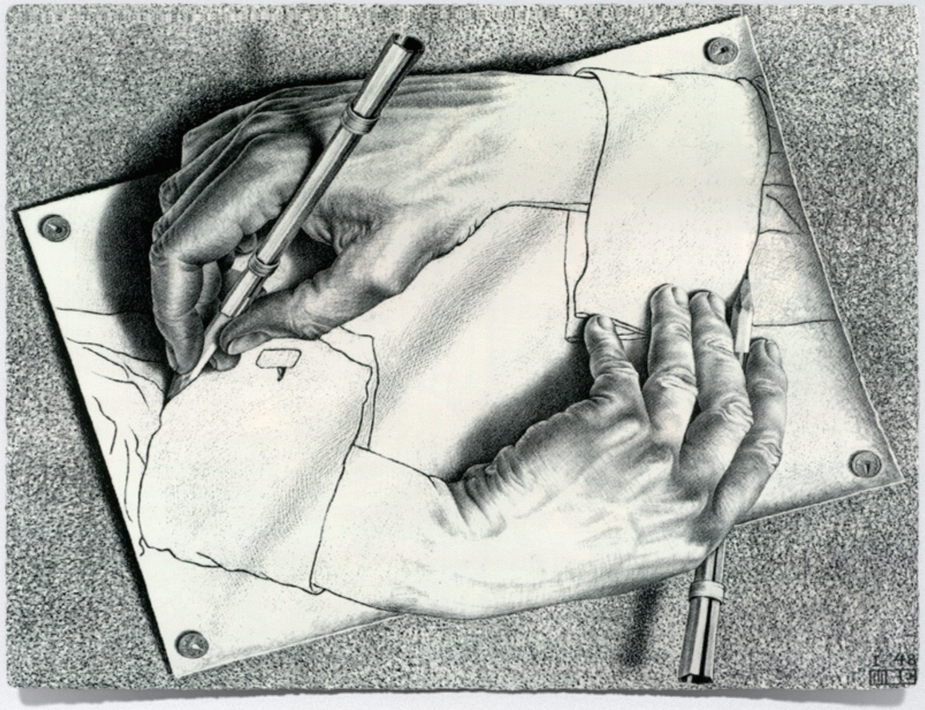
RECURSION IS A PREREQUISITE FOR LINKED LISTS

© Dr Markus Lumpe, 2023

IMPOSSIBLE STRUCTURES



http://www.mcescher.com



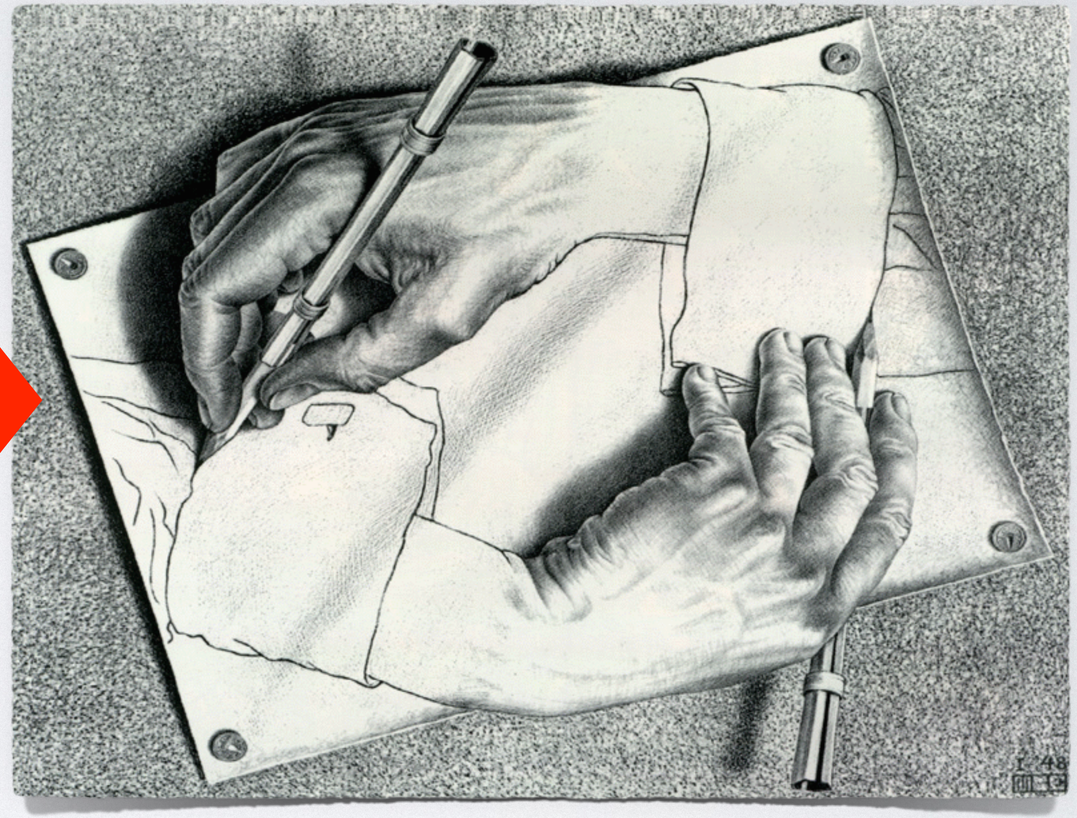
Fair use of material subject to copyright: Low resolution images for educational presentation on recursion.

© Dr Markus Lumpe, 2023

IMPOSSIBLE STRUCTURES IN C++

```
#include "ClassB.h"

class ClassA
{
    use ClassB
};
```

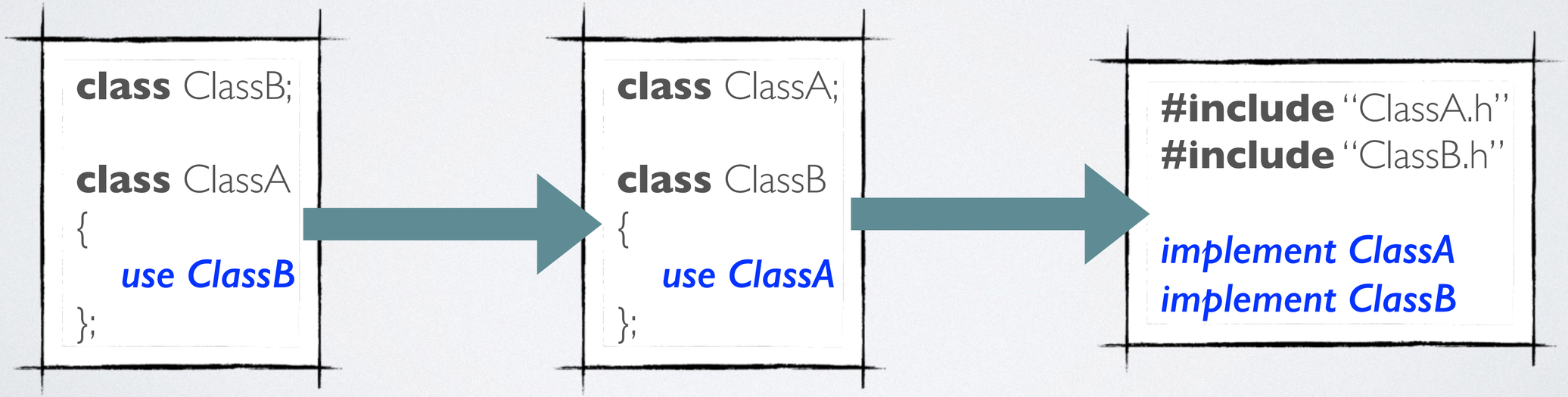
```
#include "ClassA.h"

class ClassB
{
    use ClassA
};
```



© Dr Markus Lumpe, 2023

FORWARD DECLARATION

- Unlike in Java or C#, mutual recursive classes in C++ are not automatically resolved.
- We must resolve the dependency manually and use forward declarations in specifications.
- In the implementations, we must include all specifications, so that the compiler can resolve the dependencies.



© Dr Markus Lumpe, 2023