

Department I - C Plus Plus

## Modern and Lucid C++ Advanced for Professional Programmers

Week 7 – Advanced Templates

Prof. Peter Sommerlad / Thomas Corbat  
Rapperswil, 04.04.2019  
FS2019



## Recap Week 6



- **Explicit heap memory allocation**

- new expression

- **Syntax**

`new <type> <initializer>`

- **Allocates memory for an instance of <type>**

- **Returns a pointer to the object or array created (on the heap)**

- of type <type> \*

- **The arguments in the <initializer> are passed to the constructor of <type>**

```
struct Point {  
    Point(int x, int y) :  
        x {x}, y {y} {}  
    int const x, y;  
};  
  
Point * createPoint(int x, int y) {  
    return new Point{x, y}; //constructor  
}  
  
Point * createCorners(int x, int y) {  
    return new Point[2]{{0, 0}, {x, y}};  
}
```

- **Explicit heap memory deallocation**

- delete expression

- **Syntax**

`delete <pointer>`

- **Deallocates the memory (of a single object) pointed to by the <pointer>**

- **Calls the Destructor of the destroyed type**

- **delete nullptr is well defined**

- it does nothing

- **Deleting the same object twice is Undefined Behavior!**

```
struct Point {  
    Point(int x, int y) :  
        x {x}, y {y} {}  
    int const x, y;  
}  
  
void funWithPoint(int x, int y) {  
    Point * pp = new Point{x, y};  
    //pp member access with pp->  
    //pp is the pointer value  
    delete pp; //destructor  
}
```



- **Alternative to allocating and deallocating a resource explicitly**
  - Wrap allocation and deallocation in a class
  - Constructor for allocation
  - Destructor for deallocation
- **The automatic destruction at the end of a scope will take care of the resource deallocation**
- **Works with exceptions**
  - No finally required
- **STL classes for heap memory**
  - `std::unique_ptr` / `std::shared_ptr`

```
struct Resource {  
    Resource() {  
        //Allocate Resource  
    }  
    ~Resource() {  
        //Deallocate Resource  
    }  
    //API for accessing the resource  
    //Don't leak the resource!  
private:  
    WrappedResource * wrappedResource;  
};
```

```
void workWithResource() {  
    Resource item{};  
    functionThatMightThrow();  
    //Resource is released automatically  
}
```

- **Topics:**

- Static vs. Dynamic Polymorphism
- Templates Recap
- Tags for Dispatching
  - Iterators (with Boost)

# Static vs. Dynamic Polymorphism



## Pros of static polymorphism

- Happens at compile-time
- Faster execution time
  - No dynamic dispatch required
  - Easier to optimize (inline)
- Type checks at compile-time

## Cons of static polymorphism

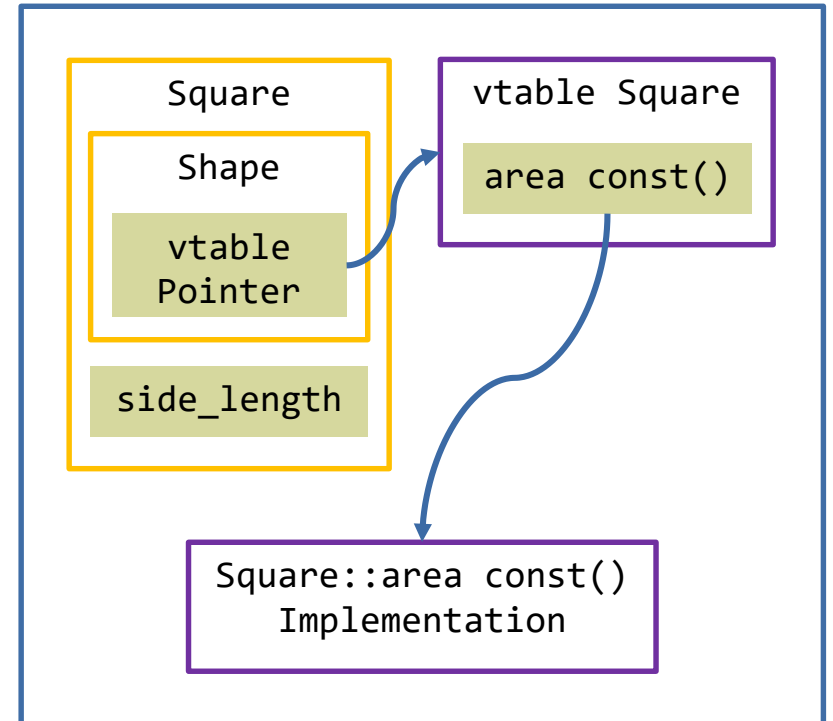
- Longer compile-times
- Template code has to be known when used
- Larger binary size
  - Copy of the used parts for each (template) instance



- A polymorphic call of a virtual function requires lookup of the target function

```
struct Shape {  
    virtual unsigned area() const = 0;  
    virtual ~Shape();  
};  
  
struct Square : Shape {  
    Square(unsigned side_length)  
        : side_length{side_length} {}  
    unsigned area() const {  
        return side_length * side_length;  
    }  
    unsigned const side_length;  
};
```

```
decltype(auto) amountOfSeeds(Shape const & shape) {  
    auto area = shape.area();  
    return area * seedsPerSquareMeter;  
};
```



- Article on this topic: <http://eli.thegreenplace.net/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c>

- Non-virtual calls directly call the target function

Argument Type

```
struct Square {  
    Square(unsigned side_length)  
        : side_length{side_length} {}  
    unsigned area() const {  
        return side_length * side_length;  
    }  
    unsigned const side_length;  
};
```

Template

```
template<typename ShapeType>  
decltype(auto) amountOfSeeds(ShapeType const & shape) {  
    auto area = shape.area();  
    return area * seedsPerSquareMeter;  
};
```

Instance

```
decltype(auto) amountOfSeeds(Square const & shape) {  
    auto area = shape.area();  
    return area * seedsPerSquareMeter;  
};
```

Square

side\_length

Square::area const()  
Implementation

- **Object is smaller**

- No vtable

- **Compiler flag for (cl.exe)**

- /d1reportSingleClassLayout<ClassName>
- /d1reportAllClassLayout

```
class Shape          size(1):
    +---
    +---

class Square         size(4):
    +---
    0    | +--- (base class Shape)
          | +---
    0    | side_length
          +---
```

```
class Shape          size(4):
    +---
    0    | {vfptr}
          +---

Shape::$vtable@:
          | &Shape_meta
          | 0
    0    | &Shape::area
    1    | &Shape::{dtor}

class Square         size(8):
    +---
    0    | +--- (base class Shape)
    0    | | {vfptr}
          | +---
    4    | side_length
          +---

Square::$vtable@:
          | &Square_meta
          | 0
    0    | &Square::area
    1    | &Square::{dtor}
```

- **Copy-pasting at compile-time**

- Instances for Square, Circle and Triangle
- The Optimizer might get rid of it

```
template<typename ShapeType>
decltype(auto) amountOfSeeds(ShapeType const & shape) {
    auto area = shape.area();
    return area * seedsPerSquareMeter;
};
```

```
unsigned amountOfSeeds(Square const & shape) {
    auto area = shape.area();
    return area * seedsPerSquareMeter;
};
```

```
},
```

```
Circle const & shape) {
    a();
    rSquareMeter;
```

```
},
```

```
Triangle const & shape) {
    a();
    rSquareMeter;
```

## Templates Recap



- **Template declaration**

- template Keyword
- Template Parameters

template  
Keyword

Template  
Parameters

```
template<typename ShapeType>  
decltype(auto) amountOfSeeds(ShapeType const & shape) {  
    auto area = shape.area();  
    return area * seedsPerSquareMeter;  
};
```

- **Function is implicitly inline**

- **Template arguments might be deduced from the function call arguments**

ShapeType is deduced  
to be Rectangle

```
Rectangle r{5, 8};  
auto seeds = amountOfSeeds(r);
```

Explicitly specified  
to be Rectangle

```
Rectangle r{5, 8};  
auto seeds = amountOfSeeds<Rectangle>(r);
```

Template Template  
Parameter

Template Type  
Parameter

Template Non-Type  
Parameter

```
template<template<typename, unsigned> class Container, typename Target, std::size_t N>
Target extractMiddleElement(Container<Target, N> & container) {
    Target middleElement{};
    std::swap(container.at(N / 2), middleElement);
    return middleElement;
}
```

```
std::array<int, 3> values{1, 2, 3};
extractMiddleElement(values);
```

```
Container => std::array
Target    => int
N         => 3
```

```
BoundedBuffer<int, 3> values{1, 2, 3};
extractMiddleElement(values);
```

```
Container => BoundedBuffer
Target    => int
N         => 3
```

- Since C++17 template <...> typename is allowed for template template parameters
  - C++ does not recognize it yet, class is ok

- **We could also implement `extractMiddleElement` differently**

- Instead of `N` we might use `size()`
- Instead of `Target` we have the member type `value_type` of `Container`

- **That changes the Concept of the parameter. How?**

Container must have:

- member type `value_type`
- `size()` member function

It does not need to be a template with type and unsigned parameter anymore.

- **The member type `value_type` is a dependent type**

- The compiler does not know whether `Container::value_type` is a member type, function or variable
- To tell the compiler that it is a type the `typename` keyword is required (there is also a `template` keyword for cases where the member is a template)

```
template<typename Container>
auto extractMiddleElement(Container & container) {
    typename Container::value_type middleElement{};
    std::swap(container.at(container.size() / 2), middleElement);
    return middleElement;
}
```



- In specific cases the number of template parameters might not be fix/known upfront
- Thus the template shall take an arbitrary number of parameters

- **Example:**

```
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
    std::cout << first;
    if (sizeof...(Types)) {
        std::cout << ", ";
    }
    printAll(rest...);
}
```

- **Syntax (ellipses everywhere): ...**
  - ... in template parameter list for an arbitrary number of template parameters (Template Parameter Pack)
  - ... in function parameter list for an arbitrary number of function arguments (Function Parameter Pack)
  - ... after sizeof to access the number of elements in template parameter pack
  - ... in the variadic template implementation after a pattern (Pack Expansion)

- Templates allow generic programming in C++
- A template is instantiated for a specific set of template arguments

Type  
Parameter

Non-Type  
Parameter

```
template<typename Freight, unsigned Space>
struct Carriage {
    std::array<Freight, Space> cargo{};
};

decltype(auto) createSmallTankWagon() {
    return Carriage<Oil, 1>{};
}
```

Creates  
Template Instance

```
struct Carriage {
    std::array<Oil, 1> cargo{};
};
```



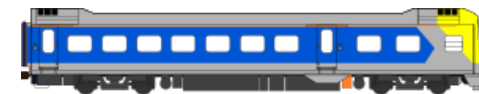
- Templates can be (partially) specialized
- Liskov's Substitution Principle does not apply for specializations, i.e. a specialized template does not need to satisfy the interface of the base template!

```
template<typename Freight, unsigned Space>
struct Carriage;

template<unsigned Space>
struct Carriage<Passenger, Space> {
    unsigned const doors{4};
}

decltype(auto) createPassengerWagon() {
    return Carriage<Passenger, 100>{};
}
```

```
struct Carriage {
    unsigned const doors{4};
}
```



- When a template is instantiated the compiler has to decide whether to use the base template or one of its specializations

## Tags for Dispatching

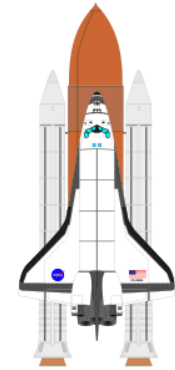


- **Template parameters don't require a specified type hierarchy**
  - but they expect an argument to satisfy a concept
- **If the same operation can be implemented more/less efficiently depending on the capabilities of the argument, tags can be used to find the "best" implementation**
- **Let's have a look at a hypothetical example**
  - Different types of space ships have different means of travel (API)

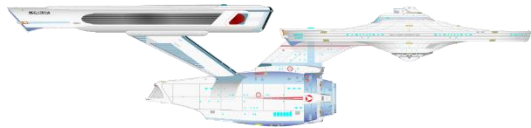
```
template<typename SpaceShip>
void travelTo(Galaxy destination, SpaceShip & ship) {
    ship.$functionUsedToTravel$(destination);
}
```



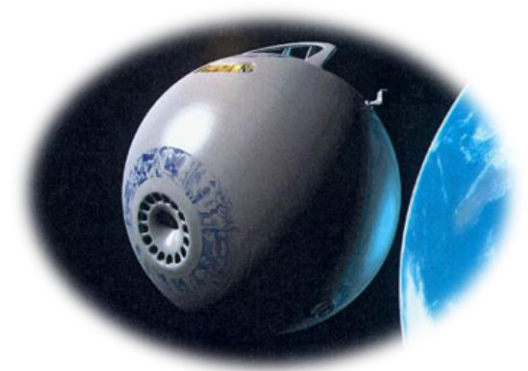
```
struct MultiPurposeCrewVehicle {  
    void travelThroughSpace(Galaxy destination);  
};
```



```
struct GalaxyClassShip {  
    void travelThroughSpace(Galaxy destination);  
    void travelThroughHyperspace(Galaxy destination);  
};
```



```
struct HeartOfGold {  
    void travelThroughSpace(Galaxy destination);  
    Galaxy travelImprobably();  
};
```



```
//Provides travelThroughSpace
struct SpaceDriveTag {};

//Provides travelThroughSpace and travelThroughHyperspace
struct HyperspaceDriveTag : SpaceDriveTag {};

//Provides travelThroughSpace and travelImprobably
struct InfiniteProbabilityDriveTag : SpaceDriveTag {};
```

- Tag types are used mark capabilities of associated types
- Such tag types do not contain any members
- It is possible to derive tag types from each other to "inherit" the capabilities
- Space ship example: Every space ship can somehow travel through space, but some space ships have more advanced technology

```
struct SpaceDriveTag {};  
  
struct HyperspaceDriveTag : SpaceDriveTag {};  
  
struct InfiniteProbabilityDriveTag : SpaceDriveTag {};
```

```
struct MultiPurposeCrewVehicle;  
  
struct GalaxyClassShip;  
  
struct HeartOfGoldPrototype;
```

- **Approach 1: Derive space ship from the associated tag type**
  - This is **not** applicable for all types (e.g. for primitive types, as we will see later)
  - This is **not** extensible (i.e. you cannot specify new kinds of tag kinds as a user of the API)
- **Approach 2: SpaceshipTraits template**

```
template<typename>  
struct SpaceshipTraits {  
    using Drive = SpaceDriveTag;  
};
```

```
template<>  
struct SpaceshipTraits<GalaxyClassShip> {  
    using Drive = HyperspaceDriveTag;  
};
```



```
template<typename Spaceship>
void travelToDispatched(Galaxy destination, Spaceship & ship, SpaceDriveTag) {
    ship.travelThroughSpace();
}

template<typename Spaceship>
void travelToDispatched(Galaxy destination, Spaceship & ship, InfiniteProbabilityDriveTag) {
    while(destination != ship.travelImprobably());
}

template<typename Spaceship>
void travelTo(Galaxy destination, Spaceship & ship) {
    typename SpaceShipTraits<SpaceShip>::Drive drive{}; //instance of the spaceship's Drive
    travelToDispatched(destination, ship, drive);          //call overloaded function
}
```

- A call of `travelTo` with a `HeartOfGold` space ship instance as argument will use the `travelImprobably` function
- A call of `travelTo` with any other space ship with the `SpaceDriveTag` in the `SpaceShipTraits` template will use `travelThroughSpace`

# Iterators



- **Different algorithms require different strengths of iterators**
  - InputIterator - read sequence once
    - operator \* returns const lvalue reference, or rvalue
  - OutputIterator - write results, without designating an end
    - operator \* returns lvalue reference
  - ForwardIterator - read/write sequence, multiple passes
    - const version: operator \* returns const lvalue reference or rvalue
    - non-const: operator \* returns lvalue
  - BidirectionalIterator - read/write sequence, back-forth
  - RandomAccessIterator - read/write/indexed sequence
- **More versatile iterators can be used for more efficient algorithm (like space ships)**
- **Iterator's capabilities can be determined at compile time (with tag types)**

- **Pre C++17: Inherit from `std::iterator<tag, value_type>`**

- Now deprecated

- **Provide member types**

Member	Description
<code>iterator_category</code>	Specifies the iterator category by tag
<code>value_type</code>	Specifies the type of the elements the iterator iterates over
<code>difference_type</code>	Specifies the type used to specify iterator distance (usually <code>ptrdiff_t</code> )
<code>pointer</code>	Specifies the pointer type for the elements the iterator iterates over
<code>reference</code>	Specifies the reference type for the elements the iterator iterates

- **Example:**

```
struct IntIterator {  
    using iterator_category = std::input_iterator_tag;  
    using value_type = int;  
    using difference_type = ptrdiff_t;  
    using pointer = int *;  
    using reference = int &;  
};
```

- Implement members required by your `?_iterator_tag`
- Example: InputIterator (Concept)

```
struct IntIterator {  
    using iterator_category = std::input_iterator_tag;  
    using value_type = int;  
    using difference_type = ptrdiff_t;  
    using pointer = int *;  
    using reference = int &;  
  
    //operator *  
    //operator ->  
    //operator ++ (prefix)  
    //operator ++ (postfix)  
    //operator ==  
    //operator !=  
};
```

```
struct IntIterator { /* Member Types Omitted */
    explicit IntIterator(int const start = 0) :
        value { start } {}
    bool operator==(IntIterator const & r) const {
        return value == r.value;
    }
    bool operator!=(IntIterator const & r) const {
        return !(*this == r);
    }
    value_type operator*() const {
        return value;
    }
    IntIterator & operator++() {
        ++value;
        return *this;
    }
    IntIterator operator++(int) {
        auto old = *this;
        ++(*this);
        return old;
    }
private:
    value_type value;
};
```

Explicit constructor

Implement != through  
operator ==

Implement postfix through  
prefix operators

Reuse pre-defined type

```
struct input_iterator_tag{};
struct output_iterator_tag{};
struct forward_iterator_tag : public input_iterator_tag{};
struct bidirectional_iterator_tag : public forward_iterator_tag{};
struct random_access_iterator_tag : public bidirectional_iterator_tag{};
```

- Iterators define type aliases for common usage
- `std::iterator<>` base class provides defaults (Pre C++17)

C++03 Style  
with typedef

```
template<typename Category, typename Tp, typename Distance = ptrdiff_t,
        typename Pointer = Tp *, typename Reference = Tp >
struct iterator {
    /// One of the iterator_tags tag types.
    typedef Category iterator_category;
    /// The type "pointed to" by the iterator.
    typedef Tp value_type;
    /// Distance between iterators is represented as this type.
    typedef Distance difference_type;
    /// This type represents a pointer-to-value_type.
    typedef Pointer pointer;
    /// This type represents a reference-to-value_type.
    typedef Reference reference;
};
```

using iterator\_category = Category

using value\_type = Tp

```
template<typename InputIterator, typename Tp>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const Tp& value) {
    ...
}
```

- STL algorithms often want to determine the type of some specific thing related to an iterator -> use optimal solution!
- However, not all iterator types are actually classes, i.e., subclasses of `std::iterator<>`.
- Default `iterator_traits` just pick the type aliases from those provided by base class `std::iterator`
- Specialization of `iterator_traits` also allows "naked pointers" to be used as iterators in algorithms (that is the main reason for the separate traits mechanism)

```
template<typename _Tp>
struct iterator_traits<_Tp*> {
    typedef random_access_iterator_tag  iterator_category;
    typedef _Tp                        value_type;
    typedef ptrdiff_t                  difference_type;
    typedef _Tp *                       pointer;
    typedef _Tp &                      reference;
};
```



```
#include <boost/iterator/counting_iterator.hpp>  
#include <boost/iterator/filter_iterator.hpp>  
#include <boost/iterator/transform_iterator.hpp>
```

- **Several pre-defined adapters with factory functions, for example**

- Counting
- Filtering
- Transforming

- **See also**

- [http://www.boost.org/doc/libs/1\\_66\\_0/libs/iterator/doc/index.html](http://www.boost.org/doc/libs/1_66_0/libs/iterator/doc/index.html)

```
struct odd {  
    bool operator()(int n) const {  
        return n % 2;  
    }  
};  
  
int main() {  
    using counter = boost::counting_iterator<int>;  
    std::vector<int> v(counter{ 1 }, counter{ 11 });  
    std::ostream_iterator<int> out { std::cout, ", " };  
    copy(v.begin(), v.end(), out);  
    std::cout << '\n';  
  
    copy(boost::make_filter_iterator(odd{}, v.begin(), v.end()),  
         boost::make_filter_iterator(odd{}, v.end(), v.end()), out);  
    std::cout << '\n';  
  
    auto sq = [](auto i) {return i * i;};  
    copy(boost::make_transform_iterator(v.begin(), sq),  
         boost::make_transform_iterator(v.end(), sq), out);  
}
```

Functor for filtering

Counting iterator

Filter iterator only odd values  
provided

transform iterator applies  
function/functor/lambda for  
each value

- **Inherit and provide own iterator**
  - Class as first template argument
  - Second argument for `value_type`
  - Other template arguments are usually defaulted and OK
- **`input_iterator_helper<T, V>`**
- **`forward_iterator_helper<T, V>`**
- **`bidirectional_iterator_helper<T, V>`**
- **`random_access_iterator_helper<T, V>`**
- **`output_iterator_helper<T>`**
  - Output only is special, no value type, special requirements!

- Using `boost/operators.hpp` shortens definition

Pass own type  
CRTP = Curiously Recurring Template Parameter

Explicit  
Constructor

Reuse  
predefined  
type

```
struct IntIteratorBoost
: boost::input_iterator_helper<IntIteratorBoost, int> {

    explicit IntIteratorBoost(int start = 0)
    : value { start } {}

    bool operator==(IntIteratorBoost const & r) const {
        return value == r.value;
    }

    value_type operator*() const { return value; }

    IntIteratorBoost & operator ++() {
        ++value;
        return *this;
    }

private:
    value_type value;
};
```

Inherit to obtain types and operations  
(through CRTP)

`operator==`  
required

- Rarely needed, special tricks required
- `operator*` just returns `this`, `operator++` is a no-op
- `operator=` defines output of value
- `*out++ = 42; // works`

```
struct MyIntOutIter {

    using iterator_category = std::output_iterator_tag;
    using value_type = int;
    /* Other Member Types Omitted */

    MyIntOutIter & operator++() {
        return *this;
    }
    MyIntOutIter operator++(int) {
        return *this;
    }
    MyIntOutIter const & operator*() const {
        return *this;
    }
    void operator=(value_type val) const {
        std::cout << "val = " << val << '\n';
    }
};
```

- Even simpler with Boost
- `operator=` defines output of value
- `*out++ = 42; // works`

```
struct MyIntOutIterBoost : boost::output_iterator_helper<MyIntOutIterBoost> {  
    void operator=(int val) const {  
        std::cout << "value = " << val << '\n';  
    }  
};
```

- An output iterator for summing and averaging

```
struct SummingIter {  
    using iterator_category = std::output_iterator_tag;  
    using value_type = int;  
    /* Other Member Types Omitted */  
  
    void operator++() { ++counter; }  
    SummingIter & operator*() {  
        return *this;  
    }  
    void operator=(int val) {  
        sum += val;  
    }  
    double average() const {  
        return sum / counter;  
    }  
    double sum{};  
    size_t counter{};  
};
```

```
std::vector<int> v {3, 1, 4, 1, 5, 9, 2};  
auto res = copy(v.begin(), v.end(), SummingIter{});  
std::cout << res.sum << " average: " << res.average();
```

## Deduction Guides





- Class template arguments can usually be determined by the compiler

```
template <typename T>
struct Box {
    Box(T content)
        : content{content}{}
    T content;
};

int main() {
    Box<int> b0{0}; //Before C++17
    Box      b1{1}; //Since C++17
}
```


- The behavior is similar to pretending as if there was a factory function for each constructor

```
template <typename T>
Box<T> make_box(T content) {
    return Box<T>{content};
}
```

```
auto gift = make_box(teddy);
```

- In the following example the only template parameter is T, which can be deduced from `std::initializer_list<T>`

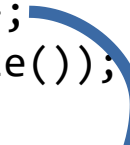
```
template <typename T>
class Sack {
    //...
    Sack(std::initializer_list<T> values)
        : theSack(values) {
    }
    //...
};
```



A blue arrow originates from the `T` in `std::initializer_list<T>` and points to the `T` in the template parameter `<typename T>`.

```
void testImplicitDeductionGuide () {
    Sack charSack{'a', 'b', 'c'};
    ASSERT_EQUAL(3, charSack.size());
}
```


`std::initializer_list<char>`



A blue arrow originates from the `char` in `std::initializer_list<char>` and points to the `char` in `Sack charSack`.

- There is no direct relation from `Iter` to `T` (constructor parameters to template parameter)

```
template <typename T>
struct BoundedBuffer {
    //...
    template <typename Iter>
    BoundedBuffer(Iter begin, Iter end);
    //...
};
```



```
void testDeductionFromIterators() {
    std::vector values{3, 1, 4, 1, 5, 9, 2, 6};
    BoundedBuffer buffer{begin(values), end(values)};
    ASSERT_EQUAL(values.size(), buffer.size());
}
```

```
error: class template argument deduction failed:
      Sack aSack(begin(values), end(values));
```

- **User-defined deduction guides can be specified in the same scope as the template**
  - Usually, after the template definition itself

```
TemplateName(ConstructorParameters) -> TemplateID;
```

- **Might be necessary for complex cases, e.g. template constructors if the constructor template parameters do not map directly to the class template parameters**
- **The deduction guide can be (and usually is) a template itself**
- **It looks similar to a free-standing constructor**
- **Unfortunately, C<sub>evelop</sub> does not recognize the deduction guides yet**

Template declaration for  
Iter

```
template <typename Iter>  
BoundedBuffer(Iter begin, Iter end) -> BoundedBuffer<typename std::iterator_traits<Iter>::value_type>;
```

Constructor signature

Deduced template instance

- Test for deducing template argument from iterator works

```
void testDeductionFromIterators() {  
    std::vector values{3, 1, 4, 1, 5, 9, 2, 6};  
    BoundedBuffer buffer{begin(values), end(values)};  
    ASSERT_EQUAL(values.size(), buffer.size());  
}
```

- **Function calls resolved at compile-time can be much faster**
- **Tag types can be used for static dispatching (For example in iterators)**
- **DIY Iterators are used much less often than functors for parameterizing the standard library algorithm**
  - Try one of the boost adapters first
- **They need pre-defined member types to work with the standard algorithms**
  - as well as a set of operators
  - if DIY `<boost/operators.hpp>` provides boilerplate code

# Self-Study



## Variadic Template Instances Unfolded (Recap)

- **Template declaration:**

```
template<typename First, typename...Types>  
void printAll(First const & first, Types const &...rest);
```

- **Implicit instantiation:**

```
int i{42}; double d{1.25}; std::string book{"Lucid C++"};  
printAll(i, f, book);
```

- **Template instance:**

```
void printAll(int const & first, double const & __rest0,  
              std::string const & __rest1) {  
    std::cout << first;  
    if (2) { //sizeof...(Types) - Number of arguments in the pack  
        std::cout << ", ";  
    }  
    printAll(__rest0, __rest1); //rest... expansion  
}
```

- **sizeof...(<PACK>) will be replaced by the number of arguments in the pack parameter**

- 0, 1, 2, ...



```
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
    //...
    printAll(rest...);
}
```

- **Pattern: rest**
- **The pattern must contain at least one pack parameter**
- **An expansion is a coma-separated list of instances of the pattern**
- **For each argument in that pack an instance of the pattern is created**
- **In an instance of the pattern the parameter pack name is replaced by an argument of the pack**

```
void printAll(int const & first, double const & __rest0, std::string const & __rest1) {
    //...
    printAll(__rest0, __rest1); //rest...
}
```

- For the call `printAll(__rest0, __rest1): printAll<double, std::string>`

```
void printAll(double const & first, std::string const & __rest0) {  
    std::cout << first;  
    if (1) { //sizeof...(Types) - Number of arguments in the pack  
        std::cout << ", ";  
    }  
    printAll(__rest0); //rest... expansion  
}
```

- For the call `printAll(<rest0>): printAll<std::string>`

```
void printAll(std::string const & first) {  
    std::cout << first;  
    if (0) { //sizeof...(Types) - Number of arguments in the pack  
        std::cout << ", ";  
    }  
    printAll(); //rest... expansion  
}
```

- What about `printAll()`?

- What about printAll()?
  - The variadic template printAll is not viable, as it requires at least one parameter
- We need a base case for the recursion

```
void printAll() {  
}
```

- Wouldn't it be feasible to just rearrange the code in the variadic template?

```
template<typename First, typename...Types>  
void printAll(First const & first, Types const &...rest) {  
    std::cout << first;  
    if (sizeof...(Types)) {  
        std::cout << ", ";  
        printAll(rest...);  
    }  
}
```

?

Self-Study

Stream Iterators



```
struct IntInputter {
    using iterator_category = std::input_iterator_tag;
    using value_type = int;
    /* Other Member Types Omitted */

    IntInputter();
    explicit IntInputter(std::istream & in)
        : input { in } {}
    value_type operator*();
    IntInputter & operator++() {
        return *this;
    }
    IntInputter operator++(int) {
        IntInputter old{*this};
        ++(*this);
        return old;
    }
    bool operator==(IntInputter const & other) const;
    bool operator!=(IntInputter const & other) const {
        return !(*this == other);
    }
private:
    std::istream & input;
};
```

Default Constructor  
for EOF

++ does nothing

Equal only if both  
EOF

Caller must guarantee survival  
of object, otherwise "dangling"  
reference!

```
IntInputter();  
value_type operator*();  
bool operator==(IntInputter const & other) const;
```

- How do we initialize the reference in the default constructor?
- What should happen in operator\*()?
- How can we compare iterators to guarantee equality for EOF-condition?

```
IntInputter();
```

- **We need a dirty trick: A global variable to initialize the reference!**
  - Put it into an anonymous namespace to hide it
  - Not good for multi-threading -> bad for production code

```
namespace {  
    // a global helper needed...  
    std::istream empty{};  
}  
IntInputter::IntInputter()  
    : input { empty } {  
    // guarantee the empty stream is not good()  
    input.clear(std::ios_base::eofbit);  
}
```

```
value_type operator*();  
bool operator==(IntInputter const & other) const;
```

- **Just input**

```
IntInputter::value_type IntInputter::operator*() {  
    value_type value{};  
    input >> value;  
    return value;  
}
```

- **And Compare. Only equal if both are !good()**

- Both eof() would result in problems on failing input when using standard algorithms

```
bool IntInputter::operator==(const IntInputter & other) const {  
    return !input.good() && !other.input.good();  
}
```



- **Alternative to global variable: Naked pointer that might point to "nothing"**
  - A pointer can be empty, which requires a check
- **Boost can be used for brevity**

```
struct IntInputterPtrBoost
: boost::input_iterator_helper<IntInputterPtrBoost, int> {
IntInputterPtrBoost() = default;
explicit IntInputterPtrBoost(std::istream & in)
: input {&in} {}
IntInputterPtrBoost::value_type operator*();
IntInputterPtrBoost & operator++() {
return *this;
}
bool operator==(IntInputterPtrBoost const & other) const;
private:
std::istream * input{};
};
```

Caller must guarantee survival of object, otherwise "dangling" reference!

Initialized with nullptr

```
value_type operator*();  
bool operator==(IntInputter const & other) const;
```

- **Input only if defined**

```
IntInputterPtrBoost::value_type IntInputterPtrBoost::operator*() {  
    value_type value{};  
    if (input) (*input) >> value;  
    return value;  
}
```

- **And Compare. Only equal if both are either nullptr or !good()**

```
bool IntInputterPtrBoost::operator==(IntInputterPtrBoost const & other) const {  
    return (!input || !input->good()) && (!other.input || !other.input->good());  
}
```