Slides

Development > Programming Languages > C++

The C++ 20 Masterclass : From Fundamentals to Advanced

Learn and Master Modern C++ From Beginning to Advanced in Plain English : C++11, C++14, C++17, C++20 and More!

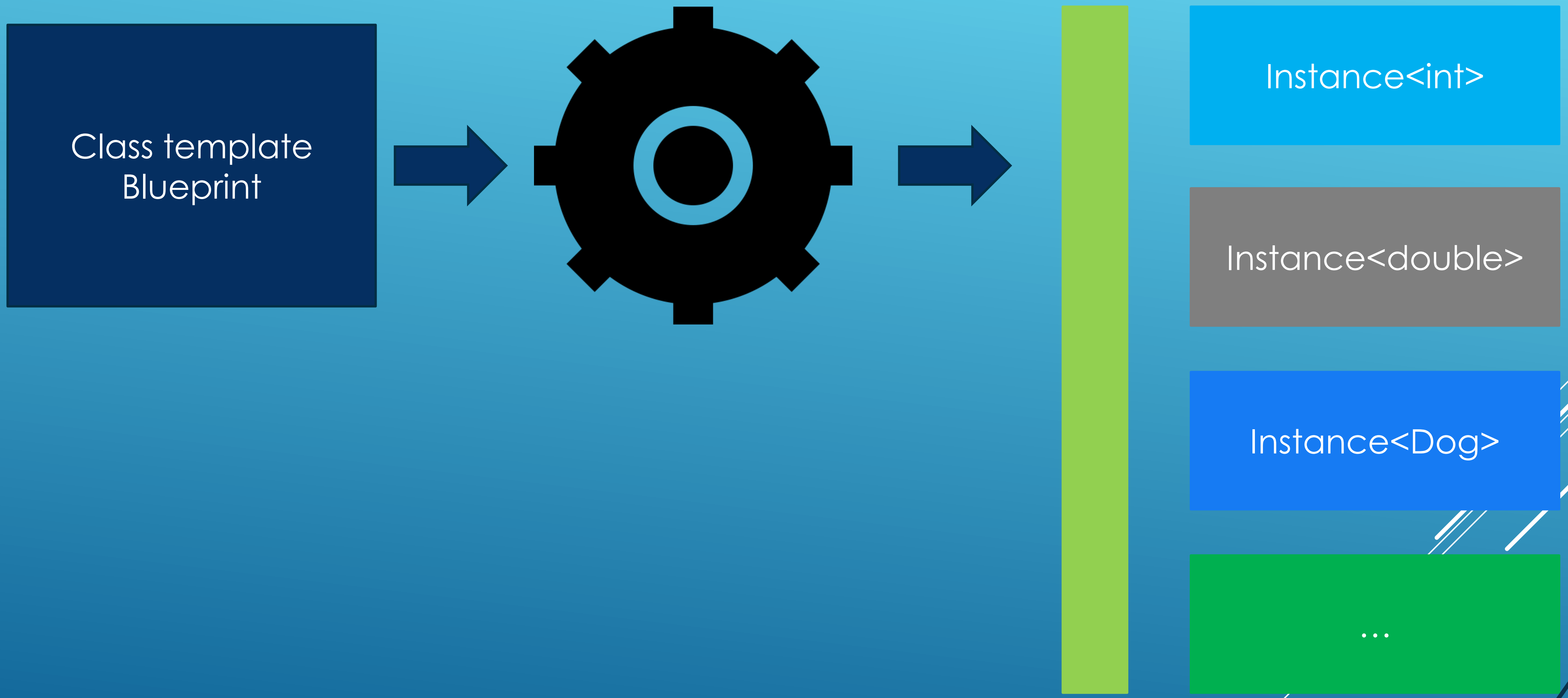
4.7 ★★★★☆

Created by Daniel Gakwaya

Section : Class Templates

Slide intentionally left empty

Class Templates : Introduction


Class template
Blueprint
Instance<int>
Instance<double>
Instance<Dog>
...

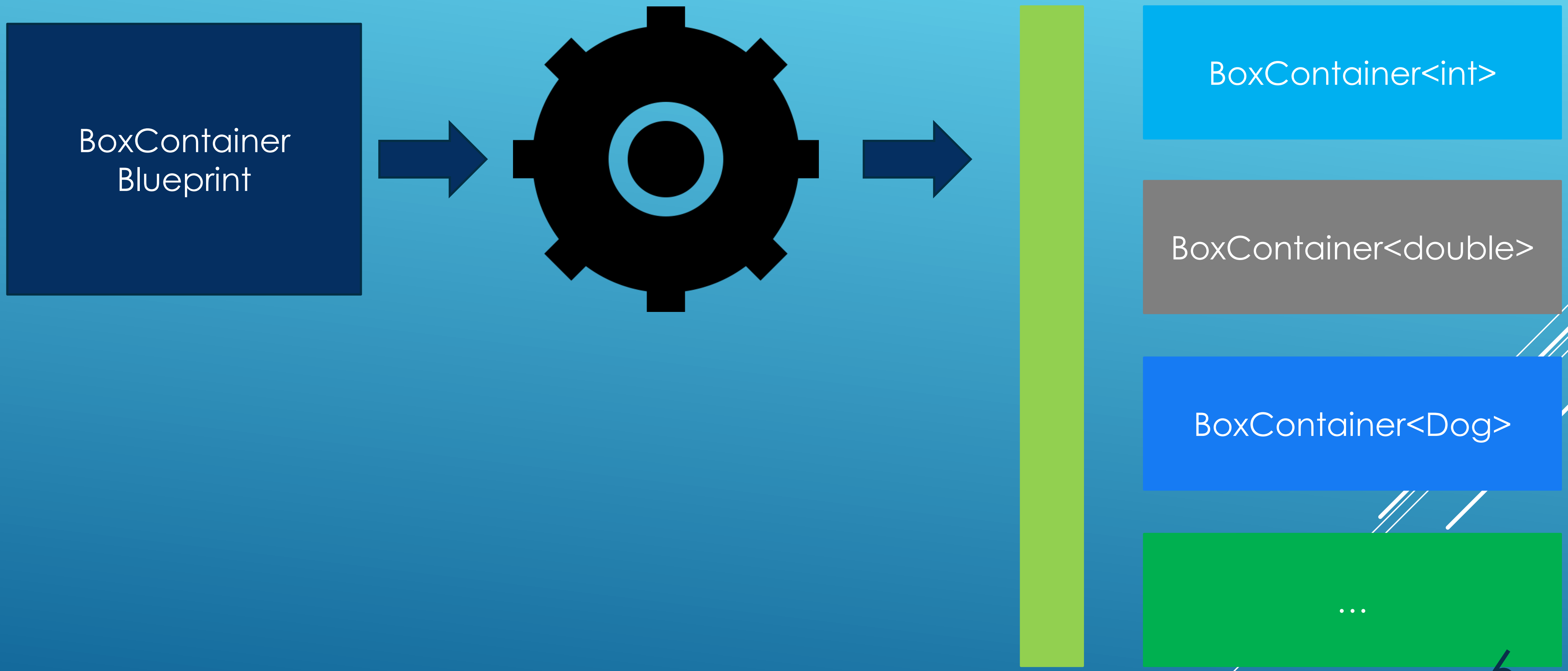
BoxContainer
Type alias

IntContainer

DoubleContainer

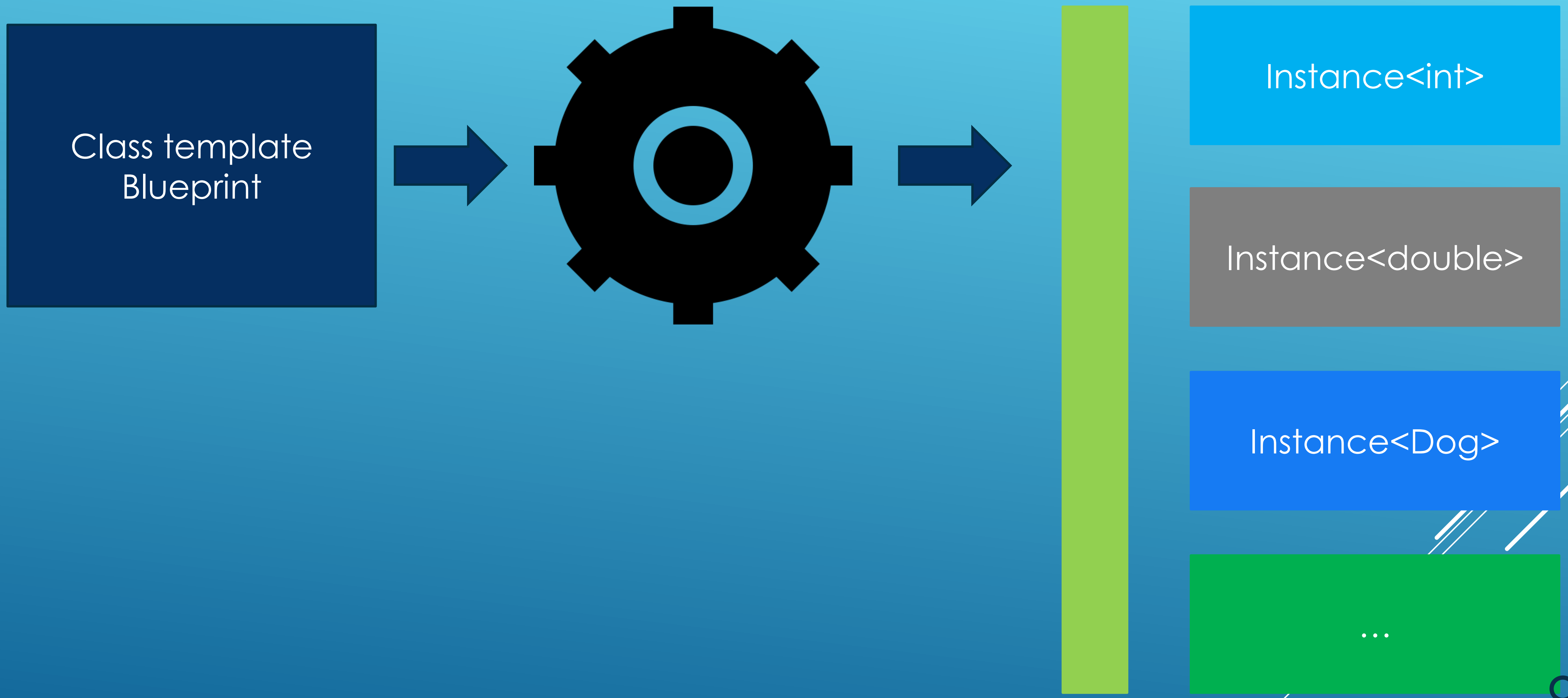
DogContainer

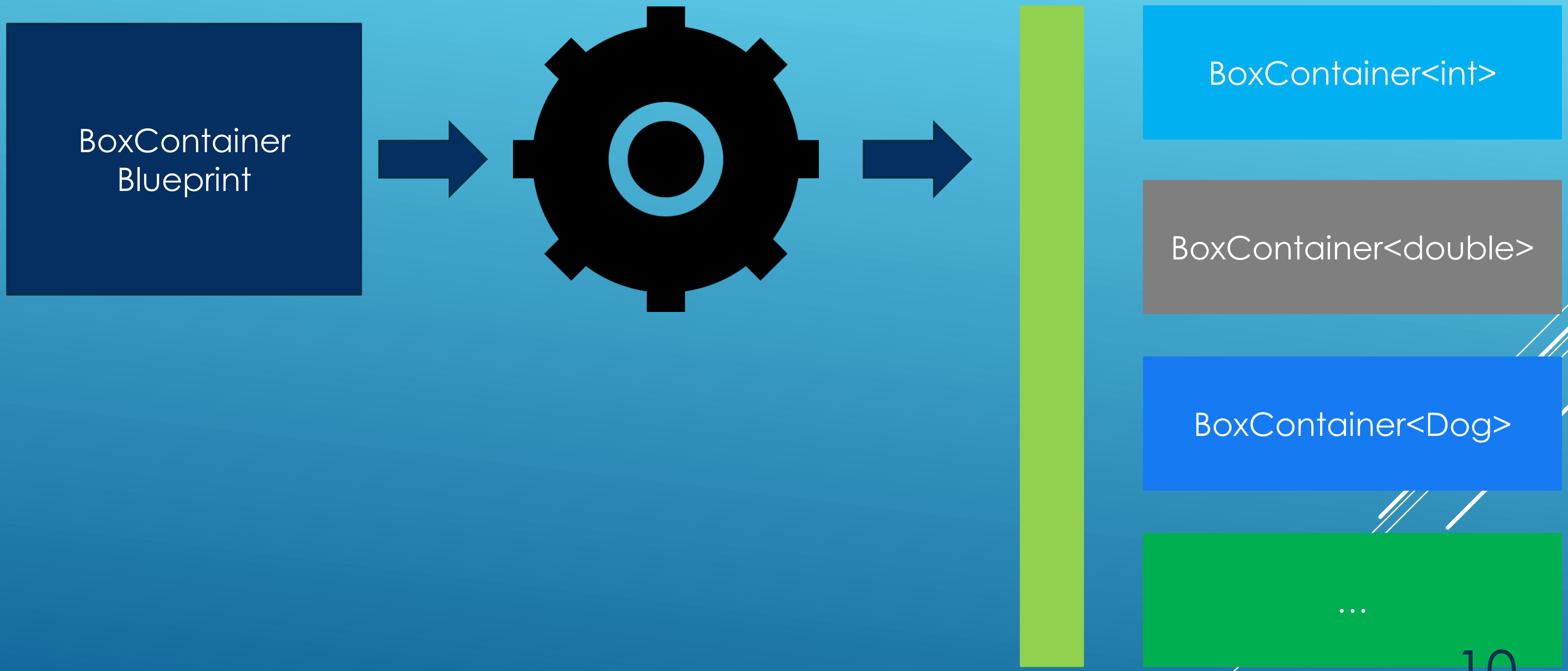
...


BoxContainer Blueprint
BoxContainer<int>
BoxContainer<double>
BoxContainer<Dog>
…

Slide intentionally left empty

Your first class template


Class template
Blueprint
Instance<int>
Instance<double>
Instance<Dog>
...


BoxContainer
Blueprint
BoxContainer<int>
BoxContainer<double>
BoxContainer<Dog>
...

Templated BoxContainer

```
template <typename T>
class BoxContainer
{
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer& source);
    ~BoxContainer<T>();

    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);

    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Templated BoxContainer [GCC 11, Clang 12, msvc 2019]

```
template <typename T>
class BoxContainer
{
public:
    BoxContainer    (size_t capacity = DEFAULT_CAPACITY);
    BoxContainer    (const BoxContainer& source);
    ~BoxContainer    ();

    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);

    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

The definitions should show up in the header file

 © Daniel Gakwaya

- All member function definitions moved into the header file, the compiler needs to see them there to generate proper template instances

- Operator<< set up as a non friend free standing global function. At the moment we don't have enough tools to deal with problems that could pop up if we used our good StreamInsertable interface which befriends global operator<<

- Operator<< is no longer a friend, so it will access BoxContainer private data through public helper getter methods

Constructors

```
template <typename T>
BoxContainer<T>::BoxContainer(size_t capacity)
{
    m_items = new T[capacity];
    m_capacity = capacity;
    m_size =0;
}

template <typename T>
BoxContainer<T>::BoxContainer(const BoxContainer<T>& source)
{
    //Set up the new box
    m_items = new T[source.m_capacity];
    m_capacity = source.m_capacity;
    m_size = source.m_size;

    //Copy the items over from source
    for(size_t i{} ; i < source.size(); ++i){
        m_items[i] = source.m_items[i];
    }
}
```

Destructor

```
template <typename T>
BoxContainer<T>::~BoxContainer()
{
    delete[] m_items;
}
```

Expand

```cpp
template <typename T>
void BoxContainer<T>::expand(size_t new_capacity){
    std::cout << "Expanding to " << new_capacity << std::endl;
    T *new_items_container;

    if (new_capacity <= m_capacity)
        return; // The needed capacity is already there

    //Allocate new(larger) memory
    new_items_container = new T[new_capacity];

    //Copy the items over from old array to new
    for(size_t i{} ; i < m_size; ++i){
        new_items_container[i] = m_items[i];
    }

    //Release the old array
    delete [ ] m_items;

    //Make the current box wrap around the new array
    m_items = new_items_container;

    //Use the new capacity
    m_capacity = new_capacity;
}
```

Adding stuff

```cpp
template <typename T>
void BoxContainer<T>::add(const T& item){
    if (m_size == m_capacity)
        //expand(m_size+5); // Let's expand in increments of 5 to optimize on the calls to expand
        expand(m_size + EXPAND_STEPS);
    m_items[m_size] = item;
    ++m_size;
}
```

Removing stuff

```cpp
template <typename T>
bool BoxContainer<T>::remove_item(const T& item){

    //Find the target item
    size_t index {m_capacity + 999}; // A large value outside the range of the current
                                     // array
    for(size_t i{0}; i < m_size ; ++i){
        if (m_items[i] == item){
            index = i;
            break; // No need for the loop to go on
        }
    }

    if(index > m_size)
        return false; // Item not found in our box here

    //If we fall here, the item is located at m_items[index]

    //Overshadow item at index with last element and decrement m_size
    m_items[index] = m_items[m_size-1];
    m_size--;
    return true;
}
```

Removing stuff

```
//Removing all is just removing one item, several times, until
//none is left, keeping track of the removed items.
template <typename T>
size_t BoxContainer<T>::remove_all(const T& item){

    size_t remove_count{};

    bool removed = remove_item(item);
    if(removed)
        ++remove_count;

    while(removed == true){
        removed = remove_item(item);
        if(removed)
            ++ remove_count;
    }

    return remove_count;
}
```


The C++ 20 Masterclass : From Fundamentals to Advanced © Daniel Gakwaya

operator+= and operator+

```
template <typename T>
void BoxContainer<T>::operator +=(const BoxContainer<T>& operand){

    //Make sure the current box can acommodate for the added new elements
    if( (m_size + operand.size()) > m_capacity)
        expand(m_size + operand.size());

    //Copy over the elements
    for(size_t i{} ; i < operand.m_size; ++i){
        m_items [m_size + i] = operand.m_items[i];
    }

    m_size += operand.m_size;
}

template <typename T>
BoxContainer<T> operator +(const BoxContainer<T>& left, const BoxContainer<T>& right){
    BoxContainer<T> result(left.size( ) + right.size( ));
    result += left;
    result += right;
    return result;
}
```

Copy assignment

```
template <typename T>
void BoxContainer<T>::operator =(const BoxContainer<T>& source){
    T *new_items;

    // Check for self-assignment:
    if (this == &source)
            return;

    if (m_capacity != source.m_capacity)
    {
        new_items = new T[source.m_capacity];
        delete [ ] m_items;
        m_items = new_items;
        m_capacity = source.m_capacity;
    }

    //Copy the items over from source
    for(size_t i{} ; i < source.size(); ++i){
        m_items[i] = source.m_items[i];
    }

    m_size = source.m_size;
}
```

Omitting <T> inside class definition

```
template <typename T>
class BoxContainer
{
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer(const BoxContainer& source);
    ~BoxContainer();
/*   ...
    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);
    //In class operators
    void operator +=(const BoxContainer& operand);
    void operator =(const BoxContainer& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

- A template is only instantiated once, it is reused every time the type is needed in your code
- Also, all the class members are inline by default, so we are safe from ODR issues if the template header is included in many files

© Daniel Gakwaya

Slide intentionally left empty

The C++ 20 Masterclass : From Fundamentals to Advanced © Daniel Gakwaya

Instances of class templates

Templated BoxContainer

```cpp
template <typename T>
class BoxContainer
{
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer& source);
    ~BoxContainer<T>();

    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);

    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Some Facts

- Template instance is created for a given type only once
- Only methods that are used are instantiated

Slide intentionally left empty

Non type template parameters

Non type template parameters

```cpp
template <typename T , size_t maximum>
class BoxContainer
{
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T,maximum>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T,maximum>(const BoxContainer<T,maximum>& source);
    ~BoxContainer<T,maximum>();
    /* ...

    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);
    //In class operators
    void operator +=(const BoxContainer<T,maximum>& operand);
    void operator =(const BoxContainer<T,maximum>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

```
BoxContainer<int,10> int_box1 ; // Generarates a template instance with int and 10
BoxContainer<int,11) int_box2 ; // Generates another template instance.
```

Traditionally, NTTPs could only be of int like types like int,size_t, basically types you could use to model sizes and ranges

Type name as non type template parameter

```
template <typename T, T threashold>
class Point{
public :
    Point(T x , T y);
    ~Point() = default;
private :
    T m_x;
    T m_y;
};

template <typename T, T threashold>
Point<T,threashold>::Point(T x, T y)
    : m_x(x) , m_y(y)
{

}
```

Type name as non type template parameter

```
Point<int,44> point1(10,20); // Works
Point<double,33.1> point2(11.22,22.33);// Compiler error : double not a valid
                                        // non type template parameter : only integral types
                                        //that can represent sizes or ranges allowed.
```

Non type template parameters can make your code hard to read as the definition of each member function outside the class has to be decorated with a template statement with the non type parameters in. This is a requirement from the compiler

As if that wasn't enough, a different instance is generated each time the non type template parameters values are different

```
BoxContainer<int,10> int_box1 ; // Generarates a template instance with int and 10
BoxContainer<int,11) int_box2 ; // Generates another template instance.
```

C++ 20 has relaxed the requirements for a type to be usable as a NTTP
It is now possible to use floating types as NTTPs and even some class types
This feature is still shaky across compilers so you can consider it non existent for now

 © Daniel Gakwaya

Slide intentionally left empty

© Daniel Gakwaya

Default values for template parameters

Default values for template parameters

```
template <typename T = int , size_t maximum = 10>
class BoxContainer
{
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T,maximum>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T,maximum>(const BoxContainer<T,maximum>& source);
    ~BoxContainer<T,maximum>();
    /* ...
    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);
    //In class operators
    void operator +=(const BoxContainer<T,maximum>& operand);
    void operator =(const BoxContainer<T,maximum>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Default values for template parameters

```
BoxContainer int_box; // Defaults to <int,10>
BoxContainer <double> int_box2; // Defaults to <double,10>
BoxContainer <char,5> int_box3; // Defaults are overriden
```

Slide intentionally left empty

Explicit Template Instantiations

```cpp
template <typename T = int , size_t maximum = 10>
class BoxContainer
{
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T,maximum>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T,maximum>(const BoxContainer<T,maximum>& source);
    ~BoxContainer<T,maximum>();
    /* ...
    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);
    //In class operators
    void operator +=(const BoxContainer<T,maximum>& operand);
    void operator =(const BoxContainer<T,maximum>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Explicit template instantiations

```cpp
#include <iostream>
#include <string>
#include "boxcontainer.h"

template class BoxContainer<double,10>;
template class BoxContainer<std::string,5>;

int main(int argc, char **argv)
{
    std::cout << "Hello World in C++20!" << std::endl;
    return 0;
}
```

- The compiler generates the instances based on the arguments you provide
- Template instances are put exactly where your template class statements are in your code
- All members of the class template are instantiated, regardless of whether they are used or not
- This feature is useful to debug your class template code

© Daniel Gakwaya

Slide intentionally left empty

Class Template specialization

```cpp
// Regular class template
template <typename T>
class Adder
{
public:
    Adder(){
    }
    T add(T x, T y);
    void do_something(){
        std::cout << "Doing something..." << std::endl;
    }
};

template <typename T>
T Adder<T>::add(T a, T b)
{
    return a+b;
}
```

Specializing the class template for char*

```
// Template specialization
template <>
class Adder <char*>
{
public:
    Adder(){
    }
    char* add(char* a, char* b);
};

// template <>   <= this is not needed if defined outside of class
char* Adder<char*>::add(char* a, char* b)
{
    return strcat(a,b);
}
```


 © Daniel Gakwaya

```cpp
int main(int argc, char **argv)
{
    Adder<int> adder_int;
    adder_int.do_something();
    std::cout << adder_int.add(10,20) << std::endl;

    char str1[20] {"Hello"};
    char str2[] {" World!"};

    Adder<char*> adder_c_str;
    //adder_c_str.do_something();
    std::cout << adder_c_str.add(str1,str2) << std::endl;
    return 0;
}
```

Class template specializations are FULL classes, they are not templates. If their definitions show up in a header and the header is included in multiple files, you'll get ODR violations

© Daniel Gakwaya

Watch out!

A common mistake for beginners is to think that you can specialize a few methods, leave the others out, hoping the compiler will fill those in from the original template.

© Daniel Gakwaya

Watch out!

- A template specialization is a completely different class from the class template itself , with it's own member variables and functions.
- It just so happens to take precedence over potential regular template instantiations for a given type, because the compiler thinks that if you went through the trouble of specializing an instance of the template for the type, you probably know better, and it uses your specialization.

Slide intentionally left empty

Template specialization for single member functions

Specializing an entire class template

```cpp
// Template specialization
template <>
class Adder <char*>
{
public:
    Adder(){
    }
    char* add(char* a, char* b);
};

// template <>   <= this is not needed if defined outside of class
char* Adder<char*>::add(char* a, char* b)
{
    return strcat(a,b);
}
```

```cpp
template <typename T = int >
class BoxContainer
{
    /* ...
public:
    BoxContainer<T >(size_t capacity = DEFAULT_CAPACITY);
/* ...
  T get_max() const{
      size_t max_index = 0;

      for(size_t i{0}; i < m_size ; ++i){
          if( m_items[i] > m_items[max_index])
          {
              max_index = i;
          }
      }
      return m_items[max_index];
  }
    /* ...
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

```cpp
BoxContainer<int> int_box;
int_box.add(10);
int_box.add(11);
int_box.add(62);
int_box.add(30);
int_box.add(3);
int_box.add(7);
int_box.add(9);
int_box.add(8);

std::cout << "int_box : " << int_box << std::endl;
std::cout << "int_box.max : " << int_box.get_max() << std::endl;


BoxContainer<const char*> char_ptr_box;

char_ptr_box.add("Zoo");
char_ptr_box.add("World");
std::cout << "char_ptr_box : " << char_ptr_box << std::endl;
std::cout << "char_ptr_box : " << char_ptr_box.get_max() << std::endl;
```

```
// Specializing get_max
template <> inline
const char* BoxContainer<const char*>::get_max() const
{
   size_t max_index = 0;

   for(size_t i{}; i < m_size ; ++i){

       if((strcmp(m_items[i],m_items[max_index])) > 0){
           max_index = i;
       }

   }

   return m_items[max_index];
}
```

Slide intentionally left empty

Friends of class templates

```cpp
template <typename T = int >
class BoxContainer
{
    /* ...
public:
    BoxContainer<T >(size_t capacity = DEFAULT_CAPACITY);
/* ...
  T get_max() const{
      size_t max_index = 0;

      for(size_t i{0}; i < m_size ; ++i){
          if( m_items[i] > m_items[max_index])
          {
              max_index = i;
          }
      }
      return m_items[max_index];
  }
    /* ...
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Friend class : non template

Friend class : template

Friend function : non template

Friend function : template

 © Daniel Gakwaya

Friend functions that take a parameter (or more) of our template class. The goal is so that we are able to model an operator<< for our template class

Operator<< takes a parameter of our class template

```
template < typename T>
inline std::ostream& operator<<(std::ostream& out, const BoxContainer<T>& operand){

    out << "BoxContainer : [ size :  " << operand.size()
        << ", capacity : " << operand.capacity() << ", items : " ;

    for(size_t i{0}; i < operand.size(); ++i){
        out << operand.get_item(i) << " " ;
    }
    out << "]";

    return out;
}
```

Slide intentionally left empty

Friend functions for class templates

The C++ 20 Masterclass : From Fundamentals to Advanced © Daniel Gakwaya

Friend functions that take a parameter (or more) of our template class. The goal is so that we are able to model an operator<< for our template class

```cpp
template<typename T>
class TemplateClass; // forward declare to make function declaration possible

template<typename T> // declaration
void some_func( TemplateClass<T>);

template <typename T>
class TemplateClass{
    friend void some_func<T>(TemplateClass<T>  param);
public :
     explicit TemplateClass<T>(){

     }
     void set_up(T param) {
         m_var = param;
     }
     void do_something(const T a, T b){
         std::cout << "Doing something with " << a << " and " << b << std::endl;
     }
private :
     T m_var;
};
```


The C++ 20 Masterclass : From Fundamentals to Advanced © Daniel Gakwaya

```cpp
template <typename T>
void some_func(TemplateClass<T>  param){
    std::cout << "Inside some_func , accessing private data of TemplateClass : "
        << param.m_var << std::endl;
}

int main(int argc, char **argv)
{

    TemplateClass<int> object1;
    object1.set_up(10);

    TemplateClass<double> object2;
    object2.set_up(12.2);

    some_func(object1);
    some_func(object2);
    return 0;
}
```

Slide intentionally left empty

Operator<< for class templates

 © Daniel Gakwaya

Friend functions that take a parameter (or more) of our template class. The goal is so that we are able to model an operator<< for our template class

```cpp
template <typename T>
class BoxContainer
{
    friend std::ostream& operator<< (std::ostream&, const BoxContainer<T>&);
public:

        ...

};
```

```cpp
// definition
template<typename T>
std::ostream& operator<<(std::ostream& out, const BoxContainer<T>& operand)
{
    out << "BoxContainer : [ size :  " << operand.m_size
        << ", capacity : " << operand.m_capacity << ", items : " ;

    for(size_t i{0}; i < operand.m_size; ++i){
        out << operand.m_items[i] << " " ;
    }
    out << "]";

    return out;
}
```

```
#include <iostream>
#include "boxcontainer.h"

int main(int argc, char **argv)
{
    BoxContainer<int> int_box;

    int_box.add(1);
    int_box.add(2);

    std::cout << "int_box : " << int_box << std::endl;

    return 0;
}
```

Solution1

```
friend std::ostream& operator<< <> (std::ostream&, const BoxContainer<T>&);
friend std::ostream& operator<< <T> (std::ostream&, const BoxContainer<T>&);
```

Solution2

```
public :
    friend std::ostream& operator<<(std::ostream& out, const BoxContainer<T>& operand)
    {
        out << "BoxContainer : [ size :  " << operand.m_size
            << ", capacity : " << operand.m_capacity << ", items : " ;

        for(size_t i{0}; i < operand.m_size; ++i){
            out << operand.m_items[i] << " " ;
        }
        out << "]";

        return out;
    }
```

Slide intentionally left empty

Class templates : Type traits and static asserts

#include <type_traits>

The C++ 20 Masterclass : From Fundamentals to Advanced © Daniel Gakwaya

```cpp
template <typename T>
class Point{
        static_assert(std::is_arithmetic_v<T>,
        "Coordinates of Point can only be numbers.");
public :
    Point<T>() = default;
    Point<T>(T x, T y)
    : m_x(x), m_y(y)
    {
    }
    friend std::ostream& operator<< ( std::ostream& out, const Point<T> operand){
        out << "Point [ x : " << operand.m_x
                    << ", y : " << operand.m_y << "]";
        return out;
    }
private :
    T m_x;
    T m_y;
};
```

Templated BoxContainer

```
template <typename T>
class BoxContainer
{
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer& source);
    ~BoxContainer<T>();

    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);

    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Slide intentionally left empty

Class templates : Concepts

#include <concepts>

```
template <typename T>
requires std::is_arithmetic_v<T>

class Point{
    /* ...
public :
    //Point<T>() = default;
    Point<T>(T x, T y)
    : m_x(x), m_y(y)
    {
    }
    friend std::ostream& operator<< ( std::ostream& out, const Point<T> operand){
        out << "Point [ x : " << operand.m_x
                    << ", y : " << operand.m_y << "]";
        return out;
    }
private :
    T m_x;
    T m_y;
};
```

Constraint(s) attached to a class template

```
template <typename T>
requires std::is_default_constructible_v<T>
class BoxContainer
{
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer(const BoxContainer<T>& source) requires std::copyable<T>;
    ~BoxContainer();
    friend std::ostream& operator<<(std::ostream& out, const BoxContainer& operand){ ...

    // Helper getter methods
    size_t size( ) const { return m_size; }
    size_t capacity() const{return m_capacity;};
    T get_item(size_t index) const{ ...
    /* ...
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Concept declaration shows up for each method definition

```
template <typename T> requires std::is_default_constructible_v<T>
BoxContainer<T>::BoxContainer(size_t capacity)
{
    m_items = new T[capacity];
    m_capacity = capacity;
    m_size =0;
}

template <typename T> requires std::is_default_constructible_v<T>
BoxContainer<T>::BoxContainer(const BoxContainer<T>& source) requires std::copyable<T>
{ ...

template <typename T> requires std::is_default_constructible_v<T>
BoxContainer<T>::~BoxContainer()
{
    delete[] m_items;
}
```

Slide intentionally left empty

Built In Concepts

 © Daniel Gakwaya

Number concepts	
Concept	**Description**
std::integral<T>	int,char,bool,unsigned_int,…
std::floating_point<T>	float,double,long double
std::signed_integral<T>	int, char
std::unsigned_integral<T>	unsigned int,…

Comparison concepts

Concept	Description
std::ordered<T>	Type has operator== and !=
std::ordered_with<T,U>	T and U comparable with == and !=
std::totally_ordered<T>	Operators >,<,>=,<=,== and !=
std::totally_ordered_with<T,U>	T and U comparable with perators >,<,>=,<=,== and !=

Other concepts	
Concept	**Description**
std::same_as<T,U>	Two types are the same
std::destructible<T>	Has a destructor
std::derived_from<T,U>	Is one type derived from another
std::copyable<T>	Has a copy constructor
…	

Slide intentionally left empty

Concepts Example1

 © Daniel Gakwaya

Constrain for an operator<< to be there

```
template <typename T>
concept OutputStreamable = requires(std::ostream& o , T d){
    o << d;
};

//Constrain the content of the vector to have operator<<
template <OutputStreamable T>
std::ostream& operator<<( std::ostream& out,const  std::vector<T>& vec){
    out << " [ ";
    for(auto i : vec){
        out << i  << " ";
    }
    out << "]";
    return out;
}
```

```cpp
struct Point{
    double mx;
    double my;
    friend std::ostream& operator<<( std::ostream& o, const Point p){
        o << "Point [ x : " << p.mx << ", y : " << p.my << "]";
        return o;
    }
};

int main(int argc, char **argv)
{

    std::vector<int> numbers {1,2,3,4,5};
    std::vector<Point> points {{10,20} , {59,45}};
    std::cout << "numbers : " << numbers << std::endl;
    std::cout << "points : " << points << std::endl;

    std::cout << "Done!" << std::endl;
    return 0;
}
```

Slide intentionally left empty

Concepts Example2

© Daniel Gakwaya

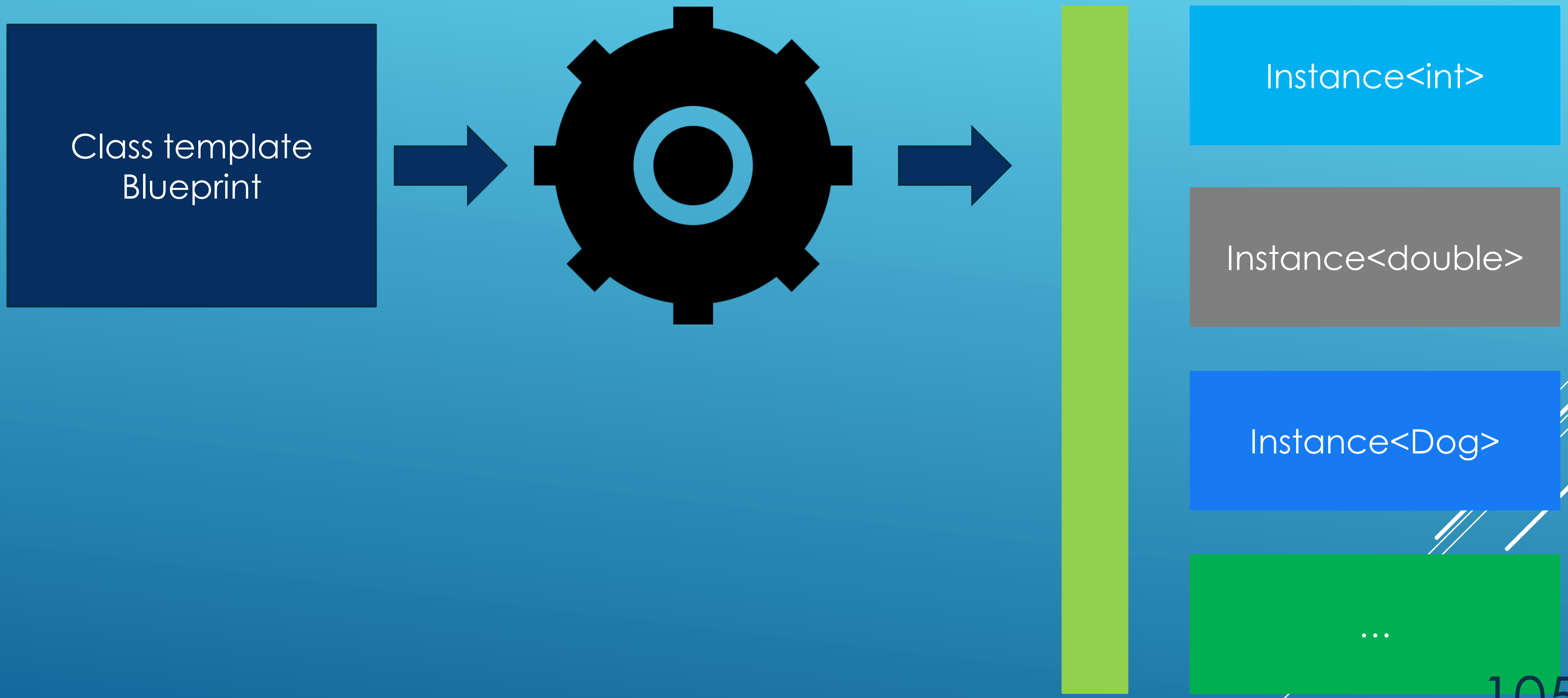
Number

```
template <typename T>
concept Number = (std::integral<T> || std::floating_point<T>)
                 && !std::same_as<T, bool>
                 && !std::same_as<T, char>;

template <Number T, Number U>
T add( T a, U b){
    return a + b;
}

int main(int argc, char **argv)
{

    //static_assert(Number<bool>);
    auto result = add (10, 44.1);
    std::cout << "result : " << result << std::endl;
    return 0;
}
```


The C++ 20 Masterclass : From Fundamentals to Advanced © Daniel Gakwaya

Slide intentionally left empty

Class Templates : Summary


Class template
Blueprint
Instance<int>
Instance<double>
Instance<Dog>
...

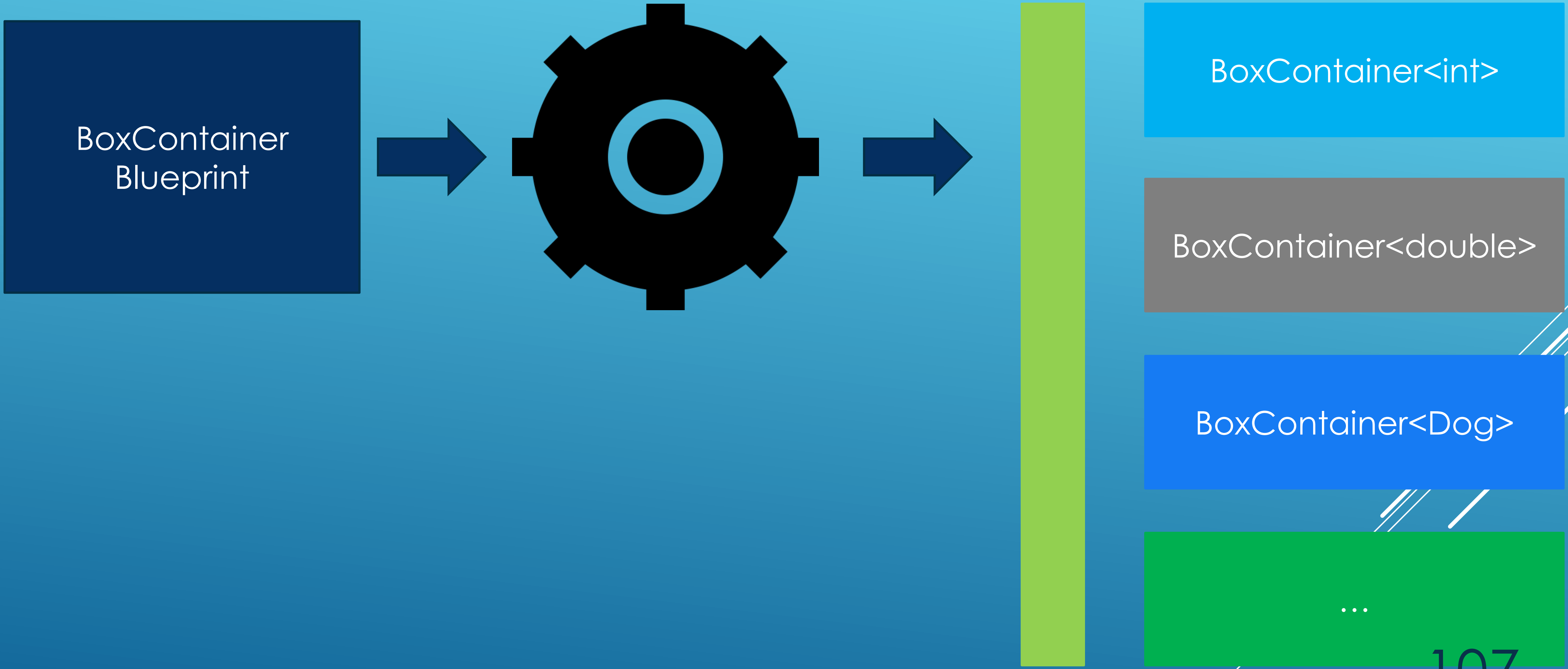
BoxContainer
Type alias

IntContainer

DoubleContainer

DogContainer

...


BoxContainer
Blueprint
BoxContainer<int>
BoxContainer<double>
BoxContainer<Dog>
...

Your first class template

```
template <typename T>
class BoxContainer
{
public:
    BoxContainer<T>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T>(const BoxContainer& source);
    ~BoxContainer<T>();

    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);

    //In class operators
    void operator +=(const BoxContainer<T>& operand);
    void operator =(const BoxContainer<T>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Template instances

- Template instance is created for a given type only once
- Only methods that are used are instantiated

Non type template parameters

```
template <typename T , size_t maximum>
class BoxContainer
{
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T,maximum>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T,maximum>(const BoxContainer<T,maximum>& source);
    ~BoxContainer<T,maximum>();
    /* ...

    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);
    //In class operators
    void operator +=(const BoxContainer<T,maximum>& operand);
    void operator =(const BoxContainer<T,maximum>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```


The C++ 20 Masterclass : From Fundamentals to Advanced © Daniel Gakwaya

Default values for template parameters

```
template <typename T = int , size_t maximum = 10>
class BoxContainer
{
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer<T,maximum>(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer<T,maximum>(const BoxContainer<T,maximum>& source);
    ~BoxContainer<T,maximum>();
    /* ...
    //Method to add items to the box
    void add(const T& item);
    bool remove_item(const T& item);
    size_t remove_all(const T& item);
    //In class operators
    void operator +=(const BoxContainer<T,maximum>& operand);
    void operator =(const BoxContainer<T,maximum>& source);
private :
    void expand(size_t new_capacity);
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Explicit template instantiations

```cpp
#include <iostream>
#include <string>
#include "boxcontainer.h"

template class BoxContainer<double,10>;
template class BoxContainer<std::string,5>;

int main(int argc, char **argv)
{
    std::cout << "Hello World in C++20!" << std::endl;
    return 0;
}
```

Template specialization : Entire class

```cpp
// Template specialization
template <>
class Adder <char*>
{
public:
    Adder(){
    }
    char* add(char* a, char* b);
};

// template <>    <= this is not needed if defined outside of class
char* Adder<char*>::add(char* a, char* b)
{
    return strcat(a,b);
}
```

Template specialization : Single Method

```
template <typename T = int >
class BoxContainer
{
    /* ...
public:
    BoxContainer<T >(size_t capacity = DEFAULT_CAPACITY);
/* ...
  T get_max() const{
      size_t max_index = 0;

      for(size_t i{0}; i < m_size ; ++i){
          if( m_items[i] > m_items[max_index])
          {
              max_index = i;
          }
      }
      return m_items[max_index];
  }
    /* ...
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Operators for class templates

```
friend std::ostream& operator<< <> (std::ostream&, const BoxContainer<T>&);
friend std::ostream& operator<< <T> (std::ostream&, const BoxContainer<T>&);
```

Operators for class templates

```cpp
public :
    friend std::ostream& operator<<(std::ostream& out, const BoxContainer<T>& operand)
    {
        out << "BoxContainer : [ size :  " << operand.m_size
            << ", capacity : " << operand.m_capacity << ", items : " ;

        for(size_t i{0}; i < operand.m_size; ++i){
            out << operand.m_items[i] << " " ;
        }
        out << "]";

        return out;
    }
```

Type traits and static asserts

```
template <typename T>
class Point{
        static_assert(std::is_arithmetic_v<T>,
        "Coordinates of Point can only be numbers.");
public :
    Point<T>() = default;
    Point<T>(T x, T y)
    : m_x(x), m_y(y)
    {
    }
    friend std::ostream& operator<< ( std::ostream& out, const Point<T> operand){
        out << "Point [ x : " << operand.m_x
                    << ", y : " << operand.m_y << "]";
        return out;
    }
private :
    T m_x;
    T m_y;
};
```

```
template <typename T>
requires std::is_arithmetic_v<T>

class Point{
    /* ...
public :
    //Point<T>() = default;
    Point<T>(T x, T y)
    : m_x(x), m_y(y)
    {
    }
    friend std::ostream& operator<< ( std::ostream& out, const Point<T> operand){
        out << "Point [ x : " << operand.m_x
                  << ", y : " << operand.m_y << "]";
        return out;
    }
private :
    T m_x;
    T m_y;
};
```

Constraint(s) attached to a class template

```
template <typename T>
requires std::is_default_constructible_v<T>
class BoxContainer
{
    static const size_t DEFAULT_CAPACITY = 5;
    static const size_t EXPAND_STEPS = 5;
public:
    BoxContainer(size_t capacity = DEFAULT_CAPACITY);
    BoxContainer(const BoxContainer<T>& source) requires std::copyable<T>;
    ~BoxContainer();
    friend std::ostream& operator<<(std::ostream& out, const BoxContainer& operand){ ...

    // Helper getter methods
    size_t size( ) const { return m_size; }
    size_t capacity() const{return m_capacity;};
    T get_item(size_t index) const{ ...
    /* ...
private :
    T * m_items;
    size_t m_capacity;
    size_t m_size;
};
```

Concept declaration shows up for each method definition

```
template <typename T> requires std::is_default_constructible_v<T>
BoxContainer<T>::BoxContainer(size_t capacity)
{
    m_items = new T[capacity];
    m_capacity = capacity;
    m_size =0;
}

template <typename T> requires std::is_default_constructible_v<T>
BoxContainer<T>::BoxContainer(const BoxContainer<T>& source) requires std::copyable<T>
{ ...

template <typename T> requires std::is_default_constructible_v<T>
BoxContainer<T>::~BoxContainer()
{
    delete[] m_items;
}
```

Built In Concepts

Number concepts	
Concept	**Description**
std::integral<T>	int,char,bool,unsigned_int,…
std::floating_point<T>	float,double,long double
std::signed_integral<T>	int, char
std::unsigned_integral<T>	unsigned int,…

Comparison concepts

Concept	**Description**
std::ordered<T>	Type has operator== and !=
std::ordered_with<T,U>	T and U comparable with == and !=
std::totally_ordered<T>	Operators >,<,>=,<=,== and !=
std::totally_ordered_with<T,U>	T and U comparable with perators >,<,>=,<=,== and !=

Other concepts	
Concept	**Description**
std::same_as<T,U>	Two types are the same
std::destructible<T>	Has a destructor
std::derived_from<T,U>	Is one type derived from another
std::copyable<T>	Has a copy constructor
…	

Constrain for an operator<< to be there

```
template <typename T>
concept OutputStreamable = requires(std::ostream& o , T d){
    o << d;
};

//Constrain the content of the vector to have operator<<
template <OutputStreamable T>
std::ostream& operator<<( std::ostream& out,const  std::vector<T>& vec){
    out << " [ ";
    for(auto i : vec){
        out << i  << " ";
    }
    out << "]";
    return out;
}
```

```cpp
struct Point{
    double mx;
    double my;
    friend std::ostream& operator<<( std::ostream& o, const Point p){
        o << "Point [ x : " << p.mx << ", y : " << p.my << "]";
        return o;
    }
};

int main(int argc, char **argv)
{

    std::vector<int> numbers {1,2,3,4,5};
    std::vector<Point> points {{10,20} , {59,45}};
    std::cout << "numbers : " << numbers << std::endl;
    std::cout << "points : " << points << std::endl;

    std::cout << "Done!" << std::endl;
    return 0;
}
```

Number

```
template <typename T>
concept Number = (std::integral<T> || std::floating_point<T>)
                 && !std::same_as<T, bool>
                 && !std::same_as<T, char>;

template <Number T, Number U>
T add( T a, U b){
    return a + b;
}

int main(int argc, char **argv)
{

    //static_assert(Number<bool>);
    auto result = add (10, 44.1);
    std::cout << "result : " << result << std::endl;
    return 0;
}
```

Slide intentionally left empty