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 : Concepts

Slide intentionally left empty

Concepts : Introduction

A mechanism to place constraints on your template type parameters

C++ 20

An alternative to static asserts and type traits

```
template <typename T>
void print_number(T n){
    static_assert(std::is_integral<T>::value , "Must pass in an integral argument");
    std::cout << "n : " << n << std::endl;
}
```

Standard built in concepts

Custom concepts

Some built in concepts

Core language concepts

same_as (C++20)	specifies that a type is the same as another type (concept)
derived_from (C++20)	specifies that a type is derived from another type (concept)
convertible_to (C++20)	specifies that a type is implicitly convertible to another type (concept)
common_reference_with (C++20)	specifies that two types share a common reference type (concept)
common_with (C++20)	specifies that two types share a common type (concept)
integral (C++20)	specifies that a type is an integral type (concept)
signed_integral (C++20)	specifies that a type is an integral type that is signed (concept)
unsigned_integral (C++20)	specifies that a type is an integral type that is unsigned (concept)
floating_point (C++20)	specifies that a type is a floating-point type (concept)

Slide intentionally left empty

Concepts

A mechanism to place constraints on your template type parameters

Standard built in concepts

Custom concepts

Some built in concepts

Core language concepts

same_as (C++20)	specifies that a type is the same as another type (concept)
derived_from (C++20)	specifies that a type is derived from another type (concept)
convertible_to (C++20)	specifies that a type is implicitly convertible to another type (concept)
common_reference_with (C++20)	specifies that two types share a common reference type (concept)
common_with (C++20)	specifies that two types share a common type (concept)
integral (C++20)	specifies that a type is an integral type (concept)
signed_integral (C++20)	specifies that a type is an integral type that is signed (concept)
unsigned_integral (C++20)	specifies that a type is an integral type that is unsigned (concept)
floating_point (C++20)	specifies that a type is a floating-point type (concept)

Syntax1

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

```cpp
char a_0{10};
char a_1{20};

auto result_a = add(a_0,a_1);
std::cout << "result_a : " << static_cast<int>(result_a) << std::endl;

int b_0{11};
int b_1{5};
auto result_b = add(b_0,b_1);
std::cout << "result_b : " << result_b << std::endl;

double c_0 {11.1};
double c_1 {1.9};
auto result_c = add(c_0,c_1); // Error std::integral concept not satisfied.
```

Syntax1

```cpp
template <typename T>
requires std::integral<T>
T add (T a, T b){
    return a + b;
}
```

Syntax1 : using type traits

```cpp
template <typename T>
requires std::is_integral_v<T>// Using a type trait
T add (T a, T b){
    return a + b;
}
```

Syntax2

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

Syntax3

```
auto add (std::integral auto a,std::integral auto b){
    return a + b;
}
```

Syntax4

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

Slide intentionally left empty

Concepts : Building your own

Standard built in concepts

Custom concepts

Different ways to build concepts

```
template <typename T>
concept MyIntegral = std::is_integral_v<T>;

template <typename T>
concept Multipliable =  requires(T a, T b) {
    a * b; // Just makes sure the syntax is valid
};

template <typename T>
concept Incrementable = requires (T a) {
    a+=1;
    ++a;
    a++;
};
```

Using custom concepts

```
//Syntax 1
template <typename T>
requires MyIntegral<T>
T add_1(T a, T b){
    return a + b;
}

//Syntax2
template <MyIntegral T>
T add_2(T a ,T b){
    return a + b;
}

auto add_3(MyIntegral auto a, MyIntegral auto b){
    return a + b;
}
```

Slide intentionally left empty

requires clause : Zooming in

The requires clause can take in four kinds of requirements :

- Simple requirements
- Nested Requirements
- Compound Requirements
- Type Requirements

Simple requirement | Expressions only checked for valid syntax

```
template <typename T>
concept TinyType = requires ( T t){
    sizeof(T) <=4; // Simple requirement : Only checks syntax
};
```

Nested requirement

```
template <typename T>
concept TinyType = requires ( T t){
    sizeof(T) <=4; // Simple requirement : Only checks syntax
    requires sizeof(T) <= 4; // Nested requirement : checks the if the expression is true
};
```

Compound requirement

```
template <typename T>
concept Addable = requires (T a, T b) {
    //noexcept is optional
    {a + b} noexcept -> std::convertible_to<int>; //Compound requirement
    //Checks if a + b is valid syntax, doesn't throw expetions(optional) , and the result
    //is convertible to int(optional)
};
```

Slide intentionally left empty

Logical combinations of concepts

Concepts can be combined with the logical operators && and | |

Combining concepts

```
template <typename T>
concept TinyType = requires ( T t){
    sizeof(T) <=4;
    requires sizeof(T) <= 4;
};

template <typename T>
//T func(T t) requires std::integral<T> || std::floating_point<T>
//T func(T t) requires std::integral<T> && TinyType<T>
T func(T t) requires std::integral<T> &&
                                    requires ( T t){
                                        sizeof(T) <=4;
                                        requires sizeof(T) <= 4;
                                    }
{
    std::cout << "value : " << t << std::endl;
    return (2*t);
}
```

Slide intentionally left empty

Concepts and auto

Concepts and auto

```
//This syntax constrains the auto parameters you pass in
//to comply with the std::integral concept
std::integral auto add (std::integral auto a,std::integral auto b){
    return a + b;
}
```

Concepts and auto

```
//Constrain declared auto var
std::integral auto x = add(10,20);
//std::floating_point auto x = add(10,20); // Compiler error
std::cout << "x : " << x << std::endl;
```

Concepts and auto

```
//std::integral auto y = 7.7;
std::floating_point auto y = 7.7;
std::cout << "y : " << y << std::endl;
```

Slide intentionally left empty

Concepts : Summary

A mechanism to place constraints on your template type parameters

C++20

Standard built in concepts

Custom concepts

Some built in concepts

Core language concepts

`same_as` (C++20)	specifies that a type is the same as another type (concept)
`derived_from` (C++20)	specifies that a type is derived from another type (concept)
`convertible_to` (C++20)	specifies that a type is implicitly convertible to another type (concept)
`common_reference_with` (C++20)	specifies that two types share a common reference type (concept)
`common_with` (C++20)	specifies that two types share a common type (concept)
`integral` (C++20)	specifies that a type is an integral type (concept)
`signed_integral` (C++20)	specifies that a type is an integral type that is signed (concept)
`unsigned_integral` (C++20)	specifies that a type is an integral type that is unsigned (concept)
`floating_point` (C++20)	specifies that a type is a floating-point type (concept)

. Using Concepts
. Building your own concepts
. Zooming in on the requires clause
. Combining concepts : Conjunction(&&) and disjunction(||)
. Concepts and auto

Head to the IDE and show all this off

