Overview

In this lesson, we'll look at an overview of what concepts are needed in C++20.

WE'LL COVER THE FOLLOWING

Until C++20, we have two diametral ways to think about functions or user-defined types (classes). Functions or classes can be defined on specific types or on generic types. In the second case, we call them to function or class templates.

There exist two extremes while talking about functions in C++. Let's take a closer look at two function implementations:

Too Specific Versus Too Generic

Too Specific

```
#include <iostream>

void needInt(int i){ // 1
  std::cout<< i << std::endl;
}

int main(){
```

Too Generic

```
#include <iostream>

template<typenameT>
T gcd(T a, T b){
  if( b == 0 ){
    return a;
  }else{
```

```
  double d{1.234};
  needInt(d);
}
```

```
    return gcd(b, a % b);
  }
}

int main(){
  std::cout<< gcd(100, 10)  << st
d::endl;
  std::cout<< gcd(3.5, 4.0) << st
d::endl;
}
```

Too Specific

It's quite a job to define for each specific type a function or a class. To avoid that burden, type conversion comes often to our rescue but it is also part of the problem.

Narrowing Conversion

We have a function `needInt(int a)` in line 3 which we can be invoked with a `double`. Now narrowing conversion takes place.

```
#include <iostream>

void needInt(int i){
  std::cout<< i << std::endl;
}

int main(){
  double d{1.234};
  needInt(d);
}
```

We assume that this is not the behavior we wanted. We started with a `double` and ended with an `int`.

But conversion also works the other way around.

Integral Promotion

We have a user-defined type `MyHouse`. An instance of `MyHouse` can be

constructed in two ways. When invoked without any argument in line 5, its attribute family is set to an empty string. This means the house is still empty.

To quickly check if the house is empty or full, we implemented a conversion operator to `bool` in line 8.

```
#include <iostream>
#include <string>

struct MyHouse{
    MyHouse() = default;
    MyHouse(const std::string& fam): family(fam){}

    operator bool(){ return !family.empty(); }

    std::string family = "";
};

void needInt(int i){
    std::cout << "int: " << i << std::endl;
}

int main(){

    std::cout << std::boolalpha << std::endl;

    MyHouse firstHouse;
    if (!firstHouse){
        std::cout << "The firstHouse is still empty." << std::endl;
    };

    MyHouse secondHouse("grimm");
    if (secondHouse){
        std::cout << "Family grimm lives in secondHouse." << std::endl;
    }

    std::cout << std::endl;

    needInt(firstHouse);
    needInt(secondHouse);

    std::cout << std::endl;

}
```

Now, instances of `MyHouse` can be used, when an `int` is required. *Strange!*

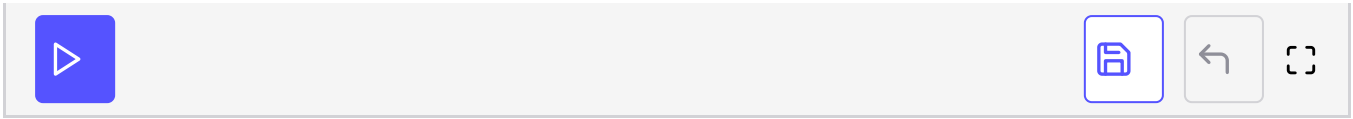
Because of the overloaded operator `bool` in line 8, instances of `MyHouse` can be used as an `int` and can, therefore, be used in arithmetic expressions: `auto res = MyHouse() + 5`. This was not our intention! But we have noted it just for completeness. With C++11, we should declare those conversion operators as

explicit. Therefore, implicit conversions will not take place.

Too Generic

Generic functions or classes can be invoked with arbitrary values. If the values do not satisfy the requirements of the function or class, it is not a problem because you will get a compile-time error.

```
#include <iostream>

template<typename T>
T gcd(T a, T b){
  if( b == 0 ){ return a; }
  else{
    return gcd(b, a % b);
  }
}

int main(){

  std::cout << std::endl;

  std::cout << gcd(100, 10)  << std::endl;

  std::cout << gcd(3.5, 4.0)<< std::endl;
  std::cout << gcd("100", "10") << std::endl;

  std::cout << std::endl;

}
```



What is the problem with this error message?

Of course, it is quite long and quite difficult to understand but our crucial concern is a different one. The compilation fails because neither `double` nor the C-string supports the `%` operator. This means the error is due to the failed template instantiation for `double` and C-string. This is too late and, therefore, really bad. No template instantiation for type `double` or C-strings should even be possible. The requirements for the arguments should be part of the function declaration and not a side-effect of an erroneous template instantiation.

Now concepts come to our rescue.

Concepts to Our Rescue

Concepts to Our Rescue

With concepts, we get something in between. We can define functions or classes which act on semantic categories. Meaning, the arguments of functions or classes are neither too specific nor too generic but named sets of requirements such as `Integral`.

Advantages

- Express the template parameter requirements as part of the interface
- Support the overloading of functions and the specialization of class templates
- Generate drastically improved error messages by comparing the requirements of the template parameter with the template arguments
- Use them as placeholders for generic programming
- Empowers you to define your concepts
- Can be used for all kinds of templates

In the next lesson, we'll learn about the history of C++ and talk about future concepts.