Examples - Template Code Simplifications

Let's see a couple of examples!

WE'LL COVER THE FOLLOWING

Line Printer

You might have already seen the below example in the Jump Start section at the beginning of this part of the course. Here, we'll dive into the details.

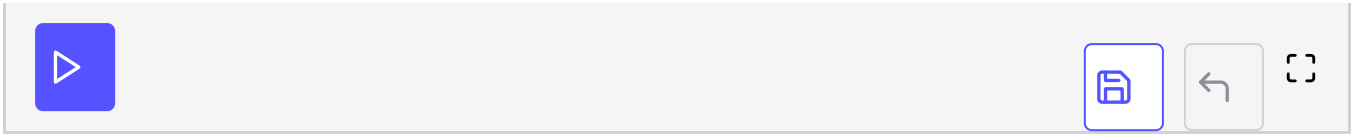
```
#include <iostream>
using namespace std;

template<typename T> void linePrinter(const T& x)
{
  if constexpr (std::is_integral_v<T>){
    std::cout << "num: " << x << '\n';
  }
  else if constexpr (std::is_floating_point_v<T>){
    const auto frac = x - static_cast<long>(x);
    std::cout << "flt: " << x << ", frac " << frac << '\n';
  }
  else if constexpr(std::is_pointer_v<T>){
    std::cout << "ptr, ";
    linePrinter(*x);
  }
  else{
    std::cout << x << '\n';
  }
}

template<typename ... Args>
void PrintWithInfo(Args ... args)
{
  (linePrinter(std::forward<Args>(args)), ...); // fold expression over the comma operator
}

int main(){
  std::cout << "-- extra info: \n";
  int i = 10;
  PrintWithInfo(&i, std::string("hello"), 10, 20.5, 30);
```

```
}
```



`linePrinter` uses `if constexpr` to check the input type. Based on that, we can output additional messages. An interesting thing happens with the pointer type - when a pointer is detected the code dereferences it and then calls `linePrinter` recursively.

Declaring Custom `get<N>` Functions

Structured Bindings works for simple structures that have all public members, like

```
struct S
{
  int n;
  std::string s;
  float d;
};

int main(){
  S s;
  auto [a, b, c] = s;
}
```

However, if you have a custom type (with private members), then it's also possible to `override` `get<N>` functions so that structured bindings can also work.

```
class MyClass{
  public:
  int GetA() const { return a; }
  float GetB() const { return b; }
  private:
  int a;
  float b;
};
template <std::size_t I> auto get(MyClass& c)
{
  if constexpr (I == 0) return c.GetA();
  else if constexpr (I == 1) return c.GetB();
}
// specialisations to support tuple-like interface
namespace std
{
```

```
  template <> struct tuple_size<MyClass> : std::integral_constant<size_t, 2> { };
  template <> struct tuple_element<0,MyClass> { using type = int; };

  template <> struct tuple_element<1,MyClass> { using type = float; };
}
```

In the above code you have the advantage of having everything in one function. It's also possible to do it as template specialisations:

```
template <> auto& get<0>(MyClass &c) { return c.GetA(); }
template <> auto& get<1>(MyClass &c) { return c.GetB(); }
```

For more examples you can read the chapter about `Replacing std::enable_if with if constexpr` and also the chapter Structured Bindings - the section about custom `get<N> specialisations`.

You can also see the following article: Simplify code with if constexpr in C++17

> **Extra Info:** The change was proposed in: P0292R2

Head over to the next lesson to learn about declaring Non-Type Template Parameters With `auto`.