C++ 23 Language Features

Contents

[a) Core Language Features 2](#_Toc191145249)

[1. Compile-time conditional execution - if consteval 2](#_Toc191145250)

[2. Deducing this 3](#_Toc191145251)

# Core Language Features

### Compile-time conditional execution - if consteval

* C++ already has constexpr and consteval to enforce compile-time evaluation:
  + constexpr allows a function to be evaluated at compile-time or runtime.
  + consteval forces a function to be evaluated only at compile-time.
* Sometimes, you want to do optimizations or use different algorithms depending on whether a value is known at compile time. For example, if you know a value is constant, you might be able to precalculate something during compilation, making your program run faster.

#### Enter if consteval

* if consteval is a conditional statement that is evaluated at compile time. It's like a regular if statement, but the compiler checks the condition while compiling the code, not while the program is running.

|  |
| --- |
| #include <iostream>  using namespace std;  constexpr int square(int x) {  if consteval { // Compile-time calculation  return x \* x;  }  else { // Runtime calculation  return x \* x;  }  }  int main() {  // Computed at compile-time  constexpr int result1 = square(5);  cout << "Compiletime Result1: " << result1 << "\n";  int y = 6;  // Computed at runtime  int result2 = square(y);  cout << "Runtime Result2: " << result2 << "\n";  } |

### Deducing this

#### The ‘this’ pointer

* In C++, when you write a **member function** (a function that belongs to a class), the ‘this’ pointer is implicitly passed to the function. This ‘this’ pointer points to the current object instance, allowing you to access its members.

|  |
| --- |
| class MyClass {  public:  void print() {  std::cout << "Hello from MyClass!\n";  }  };  MyClass obj;  obj.print(); |

* When you call obj.print(), the this pointer inside print() points to obj.

#### The "Deducing this" feature

* The "Deducing this" feature allows you to explicitly capture the ‘this’ pointer as a parameter in a member function template.
* Before C++23, if you wanted to write a member function template that could work with different types of objects (e.g., base class or derived class), you had to rely on workarounds like Curiously Recurring Template Pattern (CRTP) or manually deducing the type. This could get messy and hard to read.

#### CRTP

* What is the CRTP?
  + The CRTP is a design pattern in C++ where a class derives from a class template, and the template argument is the class itself. It sounds a bit circular, and that's why it's called "curiously recurring."
* Why is it used?
  + The CRTP allows you to achieve static polymorphism (compile-time polymorphism). This means that the specific behaviour is determined at compile time, rather than at runtime.
* A Simple Example

|  |
| --- |
| #include <iostream>  using namespace std;  template <typename Derived>  class Base {  public:  void interface() {  static\_cast<Derived\*>(this)->implementation();  }  void implementation() {  cout << "Base implementation\n";  }  };  class Derived : public Base<Derived> {  public:  void implementation() {  cout << "Derived implementation\n";  }  };  int main() {  Derived d;  d.interface(); // Calls Derived::implementation  return 0;  }  // Derived implementation |

* Explanation:
  + template <typename Derived>: Base is a class template. The template parameter Derived is what makes this the **CRTP**.
  + class Derived:public Base<Derived>: This is the "**curiously recurring**" part. Derived inherits from Base, but the template argument to Base is Derived itself. So, Base is instantiated with Derived as its template parameter.
  + void interface(): This function is in the base class. It's the key to how the CRTP works.
  + static\_cast<Derived\*>(this): Inside interface(), we're casting this (which is a pointer to Base) to a pointer to Derived. This is safe because, in this specific CRTP usage, we know that this is actually pointing to a Derived object. This is because Derived inherits from Base<Derived>.
  + ->implementation(): After the cast, we call the implementation() function. Because of the cast, the compiler knows we are calling the implementation() function of the Derived class.
* How it works:
  + When you create a Derived object and call d.interface(), the following happens:
    - d.interface() is called. This is the interface() function in the Base<Derived> class.
    - Inside interface(), this points to the Derived object d.
    - static\_cast<Derived\*>(this) converts the Base\* to a Derived\*.
    - ->implementation() calls the implementation() function of the Derived class.

#### Coming back to ‘Deducing this’

|  |
| --- |
| #include <iostream>  using namespace std;  class Base {  public:  template <typename Self>  void print(this Self&& self) {  std::cout << "Hello from Base!\n";  }  };  class Derived : public Base {  public:  void print() {  std::cout << "Hello from Derived!\n";  }  };  int main() {  Base obj;  obj.print(); // Calls Base::print  Derived derivedObj;  derivedObj.print(); // Calls Derived::print  }  // Hello from Base!  // Hello from Derived! |

* Key Points
  + this Self&& self:
    - This is the new syntax introduced in C++23.
    - Self is a template parameter that deduces the type of the object (Base, Derived, etc.).
    - self is the explicit this pointer, which can be used to access the object's members.
  + Flexibility:
    - The print function template can now work with any type of object, including derived classes.
    - If you call print on a Derived object, Self will be deduced as Derived.
  + No More CRTP:
    - Before C++23, you might have used CRTP (Curiously Recurring Template Pattern) to achieve similar behaviour. Now, you can avoid that complexity.

#### How This Feature Helps

* Deducing this allows us to write more generic member functions.
* It enables CRTP-like behaviours without explicitly using CRTP.