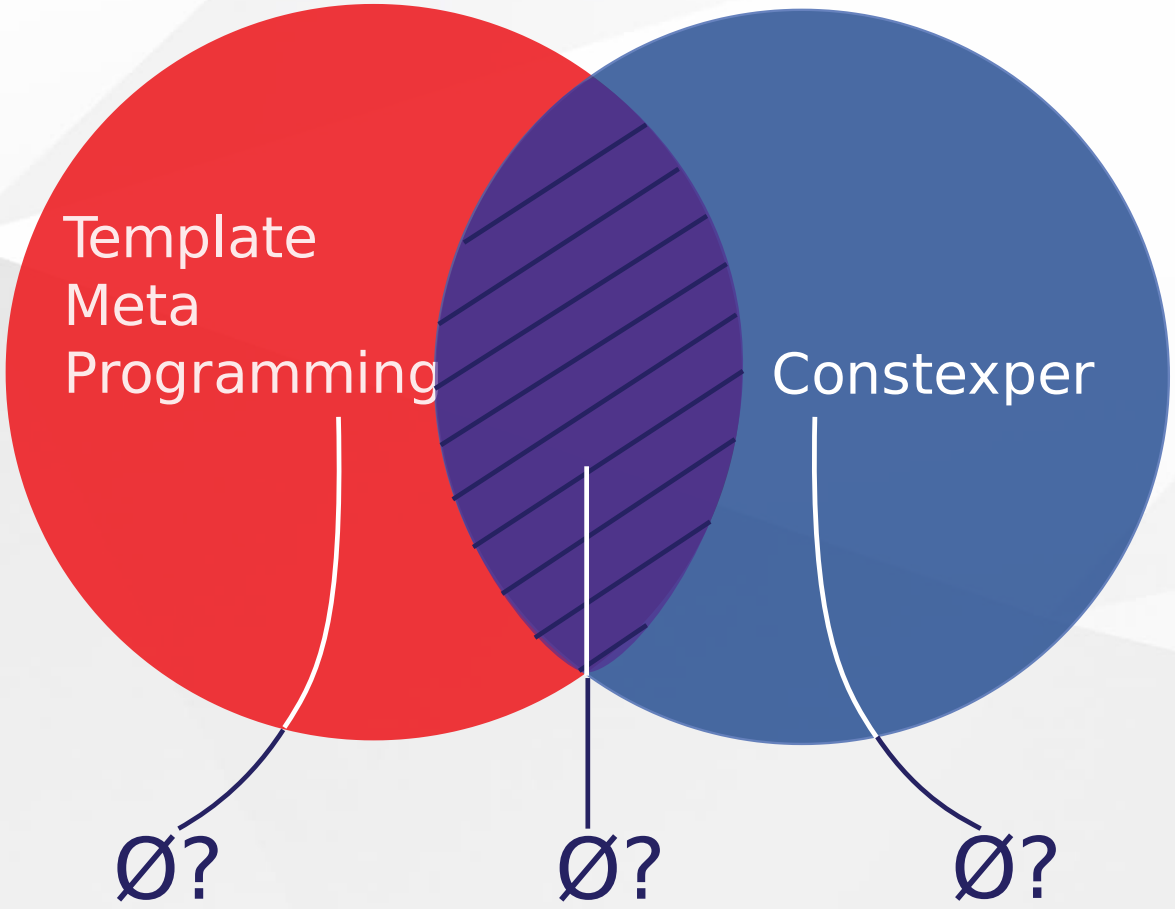

Template Meta-Programming ∩ constexpr

Mochan Shrestha

Templates

- Method of generating code at compile time

```
template<class T>
T square(T n) {
    return n * n;
}

int main(void) {
    square<int>(3);
    square<float>(3.0);
}
```

results in code as

```
template<> int square<int>(int n) { return n * n }
template<> float square<float>(float n) {return n * n;}
```

Template Meta-Programming

- Templates are "Turning complete" language!
  - State variables: template parameters
  - Loops: recursion
  - Conditionals: Conditional expressions
- Bonus: No side effects. Purely functional programming.

Example: Factorial

```
template <unsigned int n>
struct factorial {
    static const long long value = n * factorial<n - 1>::value;
};

template <>
struct factorial<0> {
    static const long long value = 1;
};
```

Problems

- Slow compile times
  - Recursive instantiations (n types are created for factorial)
- Separate compile time and run time algorithm

Why not compile time computation?

```
template<unsigned int n>
int factorial()
{
    return n * factorial<n-1>();
}

template<>
int factorial<0>()
{
    return 1;
}
```

- Creates n functions. Does not evaluate them.

SFINAE

- "Substitution Failure Is Not An Error"
- Way to a compile time if statement

```
template <class T> int f(typename T::foo*);
template <class T> int f(T);
int i = f<int>(0); // uses second overload
```

Hybrid Meta-Programming

- Combine template types and functions (all recursive)
  - `std::tuple`
  - `std::variant`

constexpr

- Specifies that a value or function can be computed at compile time
- Does not force compile time computation
- Compiler will throw an error if there is a runtime modification possible of a constexpr value

Example

```
constexpr long long factorial(int n)
{
    return n <= 1 ? 1 : (n * factorial(n - 1));
}

constexpr long long value = factorial(18);
```

Example 2

```
constexpr long long factorial(int n)
{
    long long f = 1;
    for (int i = 1; i <= n; i++)
    {
        f = f * i;
    }
    return f;
}
```

constexpr context

- Same code can be used at runtime and compile time
- Code runs on constexpr context at compile time

constexpr-ing

- Following code does not work

```
constexpr val = tan(45_deg);
```

- Libraries have to explicitly say they support constexpr context
- `tan` has a side effect - sets `errno`

constexpr Constructors

- To declare classes as constexpr we need constexpr constructor

Compile Time Lookup Tables

```
template<int N>
struct Factorial {
  constexpr Factorial() :values()
  {
    for (auto i=0; i<N; i++) {
      values[i] = factorial(i);
    }
  }

  constexpr long long factorial(int n)
  {
    long long f = 1;
    for (int i = 1; i <= n; i++) {
      f = f * i;
    }
    return f;
  }

  long long values[N];
};
```

Usage: `constexpr Factorial<10> table;`

Doom Trigonometric Tables

```
int finetangent[4096] =
{
    -170910304,-56965752,-34178904,-24413316,-18988036,-15535599,-13145455,-11392683,
    -10052327,-8994149,-8137527,-7429880,-6835455,-6329090,-5892567,-5512368,
    -5178251,-4882318,-4618375,-4381502,-4167737,-3973855,-3797206,-3635590,
    -3487165,-3350381,-3223918,-3106651,-2997613,-2895966,-2800983,-2712030,
    ...
```

C++14 constexpr

- Relaxed restrictions on constexpr functions
  - Local variables
  - `if`, `switch`, `for`, `while` inside constexpr functions

C++17 constexpr

- `if constexpr(cond)`

```
if constexpr(cond)
     statement1; // Discarded if cond is false
else
     statement2; // Discarded if cond is true
```

- if `cond` is true, `statement2` is never looked at by the compiler
- Can do SFINAE like functions

```
if constexpr (std::is_pointer_v<T>)
         return *t;
    else
         return t;
```

C++20 constexpr

- Lift more restrictions on constexpr
  - virtual functions
  - try/catch
- Standard library implementation of `std::string` and `std::vector`
- Check if in constexpr context: `is_constant_evaluated`
- `consteval` - Must be compile time function

TMP \ constexpr

- Type based meta-programming
- `std::tuple` and `std::variant` can't be created with `constexpr`
  - Can all of its methods be constexpr?

```
...
template<typename First>
struct Tuple<First> {
  Tuple(First first): first(first) {}
  First first;
};

template<>
struct Tuple<double> {
  Tuple(double first): first(static_cast<int>(first)+1) {}
  int first;
};
```

constexpr \ TMP

- Code that can run both in compile time and at run time
  - Future will have most libraries support constexpr context

constexpr ∩ TMP

- Compile time computations.
  - Use constexpr over template meta-programming

Questions?