Constexpr details

By Alexander Cheshmedjiev

Constexpr intro

```
constexpr auto arraySize = 10;
std::array<int, arraySize> data;

constexpr int pow(int base, int exp) noexcept
{
   return (exp == 0 ? 1 : base * pow(base, exp - 1));
}

std::array<int, pow(3, arraySize)> results;
```

Constexpr stripping

```
template <typename T>
void f(T t)
{
   static_assert(t == 1, "");
}

void main()
{
   constexpr int one = 1;
   f(one);
}
```

Constexpr stripping

```
template <typename T>
void f(T t)
{
   static_assert(t == 1, "");
}

void main()
{
   constexpr int one = 1;
   f(one);
}
```

```
// Here, the compiler should generate the
// code for f<int> and store the address
// of that code into fptr.
void(*fptr)(int) = f<int>;
```

Constexpr stripping

```
template <typename T>
void f(T t)
{
   static_assert(t == 1, "");
}

void main()
{
   constexpr int one = 1;
   f(one);
}
```

```
// Here, the compiler should generate the
// code for f<int> and store the address
// of that code into fptr.
void(*fptr)(int) = f<int>;


// assume this was possible
void(*fptr)(int) = f<int>;
int i = ...; // user input
fptr(i);
```

Constexpr stripping

```
template <typename T>
void f(T t)
{
   static_assert(t == 1, "");
}

void main()
{
   constexpr int one = 1;
   f(one);
}
```

```
// Here, the compiler should generate the
// code for f<int> and store the address
// of that code into fptr.
void(*fptr)(int) = f<int>;


// assume this was possible
void(*fptr)(int) = f<int>;
int i = ...; // user input
fptr(i);


template <typename T>
void f(constexpr T t)
{
   static_assert(t == 1, "");
}
```

Constexpr preservation

- To preserve constexpr-ness through argument passing, we have to encode the constexpr value into a type, and then pass a not-necessarily-constexpr object of that type to the function. The function, which must be a template, may then access the constexpr value encoded inside that type.

Side effects

```
template <typename T>
constexpr int f(T& n)
{
   return 1;
}

void main()
{
   int n = 0;
   constexpr int i = f(n);
   static_assert(i == 1, "");
}
```

Side effects

```
template <typename T>
constexpr int f(T& n)
{
   return 1;
}

void main()
{
   int n = 0;
   constexpr int i = f(n);
   static_assert(i == 1, "");
}
```

```
constexpr int sqrt(int i)
{
   if (i < 0)
       throw "i should be non-negative";

   return 1;
}

void main()
{
   constexpr int two = sqrt3(4); // ok
   constexpr int error = sqrt3(-4); // error
}
```

Side effects

```
template <typename T>
constexpr int f(T& n)
{
   return 1;
}

void main()
{
   int n = 0;
   constexpr int i = f(n);
   static_assert(i == 1, "");
}
```

```
template <typename T>
constexpr int f(T& n, bool touch_n)
{
   if (touch_n)
      n++;

   return 1;
}

void main()
{
   constexpr int j = f(n, false); // ok
   constexpr int k = f(n, true); // error
}
```

Side effects

```
template <typename T>
constexpr int f(T n)
{
   return 1;
}

void main()
{
   int n = 0;
   constexpr int i = f(n);
   static_assert(i == 1, "");
}
```

constexpr lambda

```
constexpr auto L = [](int i) constexpr { return i; };

auto L2 = [] { return 0; };
constexpr int I = L2();
```

Static if

```
template<int I>
auto get(const S& x)
{
   if constexpr(I == 0)
      return int(x);
   else if constexpr (I == 1)
      return double(x);
   else
      return string(x);
}
```

Outro