TEMPLATE ARGUMENT DEDUCTION

Template Argument Deduction by Prasanna Ghali

Plan for Today

- Template argument deduction

Template Type Deduction (1/3)

- Entire discussion is based on the excellent material presented here

Template Type Deduction (2/3)

- Consider function template and call to that function template:

```
// function template declaration
template <typename T>
void f(ParamType param);

// call f with some expression
f(expr);
```

- *Template type deduction* is process during compilation when compilers use `expr` to deduce types for `T` and `ParamType`

Template Type Deduction (3/3)

- *Template type deduction* is process during compilation when compilers use `expr` to deduce types for `T` and `ParamType`
- ~~Three~~ Two cases to consider:
  - `ParamType` is pointer or reference type
  - `ParamType` is neither pointer nor reference
  - ~~`ParamType` is forwarding reference~~ [covered in HLP3]

```
// function template declaration
template <typename T>
void f(ParamType param);

// call f with some expression
f(expr);
```

ParamType: Pointer/Reference (1/8)

- If expr's type is reference, ignore reference part and then pattern-match expr's type against ParamType to determine T

```
template <typename T>
void f(ParamType param);

f(expr);
```

```
template <typename T> void f(T& param);

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: ???, param: ???
f(cx); // T: ???, param: ???
f(rx); // T: ???, param: ???
```

ParamType: Pointer/Reference (2/8)

- If expr's type is reference, ignore reference part and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T& param);

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);   // T: int, param: int&
f(cx);  // T: const int, param: const int&
f(rx);  // T: const int, param: const int&
```

ParamType: Pointer/Reference (3/8)

- ParamType's type now changes from T& to T const&
- If expr's type is reference, ignore reference and pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T const& param);

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: ???, param: ???
f(cx); // T: ???, param: ???
f(rx); // T: ???, param: ???
```

ParamType: Pointer/Reference (4/8)

- ParamType's type now changes from T& to T const&
- If expr's type is reference, ignore reference and pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(const T& param);

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: int, param: int const&
f(cx); // T: int, param: int const&
f(rx); // T: int, param: int const&
```

ParamType: Pointer/Reference (5/8)

- ParamType's type is T*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T *param);

int x{27};              // x is an int
int const *px{&x}; // px is pointer to x
                        // as a const int

// what are deduced types for param and T?
f(&x); // T: ???, param: ???
f(px); // T: ???, param: ???
```

ParamType: Pointer/Reference (6/8)

- ParamType's type is T*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T *param);

int x{27};            // x is an int
int const *px{&x}; // px is pointer to x
                      // as a const int

// what are deduced types for param and T?
f(&x); // T: int, param: int*
f(px); // T: int const, param: int const*
```

ParamType: Pointer/Reference (7/8)

- ParamType's type is T const*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T const *param);

int x{27};          // x is an int
int const *px{&x}; // px is pointer to x
                   // as a const int

// what are deduced types for param and T?
f(&x); // T: ???, param: ???
f(px); // T: ???, param: ???
```

ParamType: Pointer/Reference (8/8)

- ParamType's type is T const*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T const *param);

int x{27};           // x is an int
int const *px{&x}; // px is pointer to x
                     // as a const int

// what are deduced types for param and T?
f(&x); // T: int, param: int const*
f(px); // T: int, param: int const*
```

ParamType: Neither Pointer Nor Reference (1/2)

- ParamType's type is T
- Fact that param is newly constructed object motivates rules governing how T is deduced from expr:
  - If expr's type is reference, ignore reference part
  - If expr is now const [or volatile], ignore that too

```
template <typename T> void f(T param);

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to const int
int const * const rprx{&x};
// what are deduced types for param and T?
f(x);      // T: ???, param: ???
f(cx);     // T: ???, param: ???
f(rx);     // T: ???, param: ???
f(rprx);   // T: ???, param: ??
```

ParamType: Neither Pointer Nor Reference (2/2)

- ParamType's type is T
- Fact that param is new object motivates rules governing how T is deduced from expr:
  - If expr's type is reference, ignore reference part
  - If expr is now const (or volatile), ignore that too

```
template <typename T> void f(T param);

int x{27};           // x is an int
int const cx{x};     // cx is const int
int const& rx{x};    // rx is reference to const int
int const * const rprx{&x};
// what are deduced types for param and T?
f(x);      // T: int, param: int
f(cx);     // T: int, param: int
f(rx);     // T: int, param: int
f(rprx);   // T: int const*, param: int const*
```

ParamType: Forwarding Reference

- ParamType's type is T&&
- Situation is bit complicated because expr can be lvalue or rvalue expression!!!

```
// function template declaration
template <typename T>
void f(T&& param);

// call f with some expression
f(expr);
```

Type Deduction: Array Arguments (1/3)

- Array types are different from pointer types – even though they seem interchangeable
- Array *decays* into pointer to its first element:

```
char const name[] = "Clint";

// array decays to pointer
char const *ptr{name};
```

Type Deduction: Array Arguments (2/3)

- What happens if array is passed to template taking by-value parameter?

```
template <typename T>
void f(T param); // param is passed by value

char const name[] = "Clint";

// what type deduced for T and param?
f(name);
```

Type Deduction: Array Arguments (3/3)

- Although functions can't declare parameters that are arrays, they can declare parameters that are references to arrays!

```
template <typename T>
void f(T& param); // param is passed by reference

char const name[] = "Clint";

// what type deduced for T and param?
f(name);
```

Deducing Array Size (1/2)

- Ability to declare references to arrays enables creation of a template that deduces number of elements that an array contains:

```
// return array size as compile-time constant
template <typename T, std::size_t N>
constexpr std::size_t array_size(T (&)[N]) noexcept {
  return N;
}

int keys[] {1,3,5,7,9};

// vals has size 7
std::array<int, array_size(keys)> vals;
```

`noexcept` operator helps compilers generate faster code because the programmer is indicating to compiler that function will not throw exceptions!!!

Deducing Array Size (2/2)

- Ability to declare references to arrays enables creation of a template that deduces number of elements that an array contains:

```
// return array size as compile-time constant
template <typename T, std::size_t N>
constexpr std::size_t array_size(T (&)[N]) noexcept {
  return N;
}

int keys[] {1,3,5,7,9};

// vals has size 7
std::array<int, array_size(keys)> vals;
```

`constexpr` specifier tells compiler that variable or function evaluates to constant expression!!!
`constexpr` is tighter form of `const`!!!

Type Deduction: Function Arguments

- Just like arrays, functions also decay into function pointers
- Type deduction is similar to arrays

```
void func(int, double);

template <typename T> void f1(T param);

template <typename T> void f2(T& param);

// what is type of T and param?
f1(func);
// what is type of T and param?
f2(func);
```