# Template Argument Deduction

## References:

The material in this handout is collected from the following references:

- Chapter 16 of the text book C++ Primer.
- Chapter 15 of C++ Templates: The Complete Guide.
- Chapters 23-26 of C++ Programming Language.
- Chapter 1 of Effective Modern C++.

## Introduction

A function template looks like this

```
template <typename T>           // T is template type parameter
void foo(FuncParamType param); // FuncParamType contains T
```

where `FuncParamType` contains `T`.

A call to this function template can look like this:

```
foo(expr); // call function template foo with some expression
```

Compilers must instantiate a template for the call `foo(expr)` by substituting a *template argument* for *template parameter* `T`. Compilers use a powerful process called *template argument deduction* to automatically determine the intended template arguments. The basic deduction process compares the type of argument `expr` of function call `foo` with the corresponding parameterized type `FuncParamType` of a function template to determine the template argument for template parameter `T`. The appropriate function template is then instantiated by substituting the deduced template argument for `T` in the parameterized type `FuncParamType`.

During template argument deduction, compilers use the function call argument `expr` to deduce two types: first the template argument for template type parameter `T` and next the function parameter type `FuncParamType`. These types are frequently different, because although `FuncParamType` contains `T`, it often contains `const` and/or reference, pointer, array, function declarators. For example, if the template is declared like this

```
template <typename T>
void foo(T const& param); // FuncParamType is reference-to-read-only-T
```

and we've this call

```
int x {0};
foo(x); // call foo with an int
```

then, the template argument for template parameter `T` is deduced to be `int`, but the function parameter type `FuncParamType` is deduced to be `int const&`.

It is natural to expect that the template argument type deduced for `T` is the same as the argument passed to the function, that is, that template argument for `T` is the type of `expr`. In the above example that's the case: `x` is an `int`, and `T`'s argument is deduced to be an `int`. But it doesn't always work that way. The argument type deduced for `T` is dependent not just on `expr`'s type, but also on the form of `FuncParamType`. There are three cases to consider:

- `FuncParamType` is a pointer or reference
- `FuncParamType` is neither a pointer nor a reference
- `FuncParamType` is a rvalue reference [will not be covered in HLP2]

## Case 1: `FuncParamType` is a Pointer or Reference

Let's start by re-writing the general form of templates and calls to it as:

```
template <typename T>
void foo(FuncParamType param);

foo(expr); // deduce types for T and FuncParamType from expr
```

Template argument deduction is simple and works like this:

1. If `expr`'s type is reference, ignore the reference part.
2. Then pattern-match `expr`'s type against `FuncParamType` to determine `T`'s type.

There are 4 scenarios to be considered: `param` is a reference; `param` is a reference-to-`const`; `param` is a pointer; `param` is a pointer-to-`const`.

### `FuncParamType` is a reference

The first syntactical case to consider is when `FuncParamType` is `T&`. The deduced types for `T` and `param` in various calls are as follows:

```
template <typename T>
void foo(T& param); // param is a reference

int x {1};          // x  is int
int const cx {x};   // cx is read-only-int
int const& rx {x};  // rx is reference-to-read-only-int

foo(x); // expr's type is int
        // param's type is T&
        // pattern-matching between int and T& gives T as int
        // thus, T is int, param's type is int&

foo(cx); // expr's type is int const
         // param's type is T&
         // pattern-matching between int const and T& gives T as int const
         // thus, T is int const, param's type is int const&

foo(rx); // expr's type is int const&
         // ignoring reference:
         // expr's type is int const
         // param's type is T&
         // pattern-matching between int const and T& gives T as int const
         // thus, T is int const, param's type is int const&
```

## `FuncParamType` is a reference-to-`const`

The second case to consider is when `FuncParamType` is `T const&`. If the type of `foo`'s parameter is changed from `T&` to `T const&`, things change a little but not by much. The `const`ness of `cx` and `rx` continue to be respected but because `param` is a reference-to-`const`, there's no longer a need for `const` to be deduced as part of `T`:

```
template <typename T>
void foo(T const& param); // param is a reference-to-const

int x {1};          // as before, x  is int
int const cx {x};   // as before, cx is read-only-int
int const& rx {x}; // as before, rx is reference-to-read-only-int

foo(x); // expr's type is int
        // param's type is T const&
        // pattern-matching between int and T const& gives T as int
        // thus, T is int, param's type is int const&

foo(cx); // expr's type is int const
         // param's type is T const&
         // pattern-matching between int const and T const& gives T as int
         // thus, T is int, param's type is int const&

foo(rx); // expr's type is int const&
         // ignoring reference:
         // expr's type is int const
         // param's type is T const&
         // pattern-matching between int const and T const& gives T as int
         // thus, T is int, param's type is int const&
```

## `FuncParamType` is a pointer

The third case to consider is when `FuncParamType` is `T*`. If the type of `foo`'s parameter is changed from `T&` to `T*`, the template argument deduction would work essentially the same way:

```
template <typename T>
void foo(T* param); // param is a pointer-to-T

int x {1};           // as before, x  is int
int const* px {&x}; // px is pointer-to-read-only-int

foo(&x); // expr's type is int*
         // param's type is T*
         // pattern-matching between int* and T* gives T as int
         // thus, T is int, param's type is int*

foo(px); // expr's type is int const*
         // param's type is T*
         // pattern-matching between int const* and T* gives T as int const
         // thus, T is int const, param's type is int const*
```

### `FuncParamType` is a pointer-to-`const`

The fourth case to consider is when `FuncParamType` is `T const*`. As with references, if the type of `foo`'s parameter is changed from `T*` to `T const*`, template argument deduction changes a little but not by much. Because `param` is a pointer-to-`const`, there's no longer a need for `const` to be deduced as part of `T`:

```
template <typename T>
void foo(T const* param); // param is a pointer-to-const-T

int x {1};          // as before, x is int
int const cx {x};   // cx is read-only-int
int const* px {&x}; // px is pointer-to-read-only-int

foo(&x); // expr's type is int*
         // param's type is T const*
         // pattern-matching between int* and T const* gives T as int
         // thus, T is int, param's type is int const*

foo(&cx); // expr's type is int const*
          // param's type is T const*
          // pattern-matching between int const* and T const* gives T as int
          // thus, T is int, param's type is int const*

foo(px); // expr's type is int const*
         // param's type is T const*
         // pattern-matching between int const* and T const* gives T as int
         // thus, T is int, param's type is int const*
```

## Case 2: `FuncParamType` is Neither Pointer Nor Reference

The general form of templates and calls to it will look like this:

```
template <typename T>
void foo(FuncParamType param);

foo(expr); // deduce T and FuncParamType from expr
```

When `FuncParamType` is neither a pointer nor a lvalue reference, then pass-by-value semantics come into play. This is the syntactic case with equality between `T` and `FuncParamType`. Pass-by-value semantics means that `param` will be a copy of whatever is passed in - a completely new object constructed on the stack. The fact that `param` will be a new object motivates the rules governing the deduction of `T` from `expr`:

1. As before, if `expr`'s type is reference, ignore the reference part.
2. If after ignoring `expr`'s reference-ness, `expr` is `const`, ignore that, too. If its `volatile`, also ignore that.

The deduced types for `param` and `T` in various calls are as follows:

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

int x {1};         // as before, x is int
```

```
int const cx {x};  // as before, cx is read-only-int
int const& rx {x}; // as before, rx is reference-to-read-only-int

foo(x); // expr's type is int
        // param's type is T
        // pattern-matching between int and T gives T as int
        // thus, T is int, param's type is int

foo(cx); // expr's type is int const
         // ignoring top-level const-ness:
         // expr's type is int
         // param's type is T
         // pattern-matching between int and T gives T as int
         // thus, T is int, param's type is int

foo(rx); // expr's type is int const&
         // ignoring top-level const-ness and reference:
         // expr's type is int
         // param's type is T
         // pattern-matching between int and T gives T as int
         // thus, T is int, param's type is int
```

Note that even though `cx` and `rx` represent `const` values, `param` isn't `const`. That makes sense: because of pass-by-value semantics, `param` is a copy of `cx` or `rx` and is therefore completely independent of them. That's why `expr`'s `const`ness (and `volatile`ness, if any) is ignored when deducing a type for `param`: just because `expr` can't be modified doesn't mean a copy of it can't be.

Note that only "top-level" `const`ness is ignored. Pointers can not only point to `const` objects but can themselves be `const`. Consider the case where `expr` is a `const` pointer to a `const` object, and pass-by-value semantics are used to pass `expr` to a by-value `param`:

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

int x {1};                 // as before, x is int
int const* const px {&x}; // px is read-only pointer to read-only int
                 // px can't be made to point to a different int nor can
                 // px be used to change the int value of the object it
                 // is pointing to
                 // the const to the left of the * says that what px points
                 // to (int object x) is const, and cannot be modified
                 // the const to the right of the * says that px is a
                 // const - that is, px is a read-only object

foo(px); // expr type is int const* const
         // ignoring "top-level" const:
         // expr type is int const*
         // param's type is T
         // pattern-matching between int const* and T gives T as int const*
         // thus, T is int const*, param's type is int const*
```

To sum up, the `const`ness of what `px` points to is preserved during type deduction, but the `const`ness of `px` itself is ignored when copying it to create the new pointer `param`.

## Niche Scenario: Array Arguments

Array types are different from pointer types, even though they're sometimes interchangeable. In some contexts, an array decays into a pointer to its first element, creating the illusion among novice C/C++ programmers that "arrays are the same as pointers." This decay is what allows code like this to compile:

```
// type of aci is "array of 5 read-only ints"
int const aci[] {1, 2, 3, 4, 5};
int const* pi {aci}; // array decays to pointer
```

Here, the `int const*` pointer `pi` is being initialized with `aci`, which is a `int const [5]`. The types `int const*` and `int const [5]` are not the same, but because of the array-to-pointer decay rule, the code compiles.

There are two cases to consider when passing an array to a function template:

- passing an array to a function template taking a by-value parameter
- passing an array to a function template taking a by-reference parameter

### Passing array to function template taking by-value parameter

Consider the following code fragment depicting a function template with by-value parameter:

```
template <typename T>
void foo(T param); // param is a by-value parameter

// type of aci is "array of 5 read-only ints"
int const aci[] {1, 2, 3, 4, 5};

foo(aci); // what types are deduced for T and param?
```

First, there is no such thing in C/C++ that allows a function parameter that is an array. The syntax `void some_func(int param[]);` is syntactic sugercoating for `void some_func(int *param);`. Because array parameter declarations are treated as if they were pointer parameters, the type of an array that's passed to a template function by value is deduced to be a pointer type. That mean that in the call to the function template `foo`, its type parameter `T` is deduced to be `int const*`:

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

// type of aci is "array of 5 read-only ints"
int const aci[] {1, 2, 3, 4, 5};

foo(aci); // aci is int const [5], but expr decays to int const*
          // param's type is T
          // pattern-matching between int const* and T gives T as int const*
          // thus, T is int const*, param's type is int const*
```

## Passing array to function template taking by-reference parameter

What happens if an array is passed to a function template taking a by-reference parameter? Although functions can't declare parameters that are arrays, they can declare parameters that are references to arrays! So, if function template `foo` is modified to take its argument by reference and `foo` is passed an array, what types are deduced for `T` and `param`?

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

// type of aci is "array of 5 read-only ints"
int const aci[] {1, 2, 3, 4, 5};

foo(aci); // pass array of 5 read-only ints to foo
          // now, what types are deduced for T and param
```

Since `foo` is passed array `aci` by reference, the type deduced for `T` is the actual type of the array! That type includes the size of the array so in this example, `T` is deduced to be `int const [5]`, and the type of `param`'s parameter is `int const (&)[5]`:

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

// type of aci is "array of 5 read-only ints"
int const aci[] {1, 2, 3, 4, 5};

foo(aci); // since foo is declared pass-by-reference,
          // type deduced for T is int const [5] and
          // param's type is int const (&)[5]
```

If `foo` is declared to take reference to `const`, that is, `FuncParamType` is `T const&`, the types deduced for `T` and `param` are:

```
template <typename T>
void foo(T const& param); // param is passed by read-only-reference

int ai[] {1, 2, 3, 4, 5}; // type of ai is "array of 5 ints"

foo(ai); // because foo is declared pass-by-const-reference,
         // type deduced for T is int [5] and
         // type deduced for param is int const (&) [5]
```