Function Templates: Nontype Parameters

References:

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

- Chapter 16 of the text book C++ Primer.
- Chapters 1 and 3 of C++ Templates: The Complete Guide.

Nontype template parameters: Example 1

For function and class templates, template parameters don't have to be types. They can also be ordinary values. As with templates using type parameters, you define code for which a certain detail remains open until the code is used. However, the detail that is open is a value instead of a type. When using such a template, you've to specify the value explicitly. The resulting code then gets instantiated.

Let's begin by thinking of how to specify arrays as parameters. Just as we define a variable that is a reference to an array, we can define a parameter that is a reference to an array. In the following code fragment, we define a function that takes a reference to `int [5]`:

```
int sum(int const (&arr)[5]) {
  int total {0};
  for (int x : arr) { total += x; }
  return total;
}

int arr[5] {1, 2, 3, 4, 5};
std::cout << "sum: " << sum(arr) << '\n';
```

The `sum` algorithm is more convenient if it can accumulate the values in arrays of *any* size. This is possible by defining a function template with a template parameter representing the size of the array:

```
template<size_t N>
int sum(int const (&arr)[N]) {
  int total {0};
  for (int x : arr) { total += x; }
  return total;
}
```

When we call this version of `sum`:

```
int iarr[5] {1, 2, 3, 4, 5};
std::cout << "sum:  " << sum(iarr) << '\n';
```

the compiler will use the static size of array `iarr` to instantiate a version of the template with the size `5` substituted for `N`:

```
int sum(int const (&arr)[5]);
```

and when we make the call with array `iarr2`:

```
int iarr2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::cout << "sum2: " << sum(iarr2) << '\n';
```

the compiler will use the static size of array `iarr2` to instantiate a version of the template with the size `10` substituted for `N`:

```
int sum(int const (&arr)[10]);
```

The `sum` algorithm can be made more generic by adding a second template parameter that represents the type of values:

```
template <size_t N, typename T>
T sum(T const (&arr)[N]) {
  T total = T();
  for (T x : arr) { total += x; }
  return total;
}
```

When we call this version of `sum`:

```
int iarr[5] {1, 2, 3, 4, 5};
std::cout << "sum:  " << sum(iarr) << '\n';
```

the compiler will use the size of array `iarr` and the type of each element of `iarr` to instantiate a version of the template with the size `5` substituted for `N` and type `int` substituted for `T`:

```
int sum(int const (&arr)[5]);
```

This call to `sum`:

```
double darr[7] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
std::cout << "sum: " << sum(darr) << '\n';
```

the compiler will instantiate a version of the template with the size `7` substituted for `N` and type `double` substituted for `T`:

```
double sum(double const (&arr)[7]);
```

Finally this call to `sum`:

```
std::string sarr[5] = {"h", "e", "l", "l", "o"};
std::cout << "sum: " << sum(sarr) << '\n';
```

will make the compiler to instantiate this version of the template:

```
std::string sum(std::string const (&arr)[5]);
```

Nontype template parameters: Example 2

Consider the following template to compare any two values of the same type:

```
template<typename T>
int compare(T const& lhs, T const& rhs) {
  if (lhs < rhs) return -1;
  if (rhs < lhs) return 1;
  return 0;
}
```

We can write a version of `compare` that will handle string literals. Such literals are arrays of `char const`. Because we cannot copy an array, we'll define our parameters as references to an array. Because we'd like to be able to compare literals of different lengths, we'll give our template two nontype parameters. The first template parameter will represent the size of the first array, and the second parameter will represent the size of the second array:

```
template<size_t M, size_t N>
int compare(char const (&lhs)[M], char const (&rhs)[N]) {
  return std::strcmp(lhs, rhs);
}
```

When we call this version of `compare`:

```
compare("math", "mathematics");
```

the compiler will use the size of the literals to instantiate a version of the template with the size `5` of literal `"math"` substituted for `M` and the size `12` of literal `"mathematics"` substituted for `N`:

```
int compare(char const (&lhs)[5], char const (&rhs)[12]);
```

Nontype template parameters: Example 3

The following function template defines a group of functions for which a certain value can be added:

```
template<int Val, typename T>
T add(T x) {
  return x + Val;
}
```

These kinds of functions can be useful if functions or operations are used as parameters. For example, if you can pass an instantiation of this function template to add a value to each element of a collection:

```
template<typename In, typename Out, typename Func>
Out xform(In first, In last, Out result, Func f) {
  while (first != last) {
    *result = f(*first); ++first; ++result;
  }
  return result;
}

int iarr[5] {1, 2, 3, 4, 5}, iarr2[5];
// iarr2 will now have values: 6, 7, 8, 9, 10
xform(iarr, iarr+5, iarr2, add<5, int>);
```

The last argument to the function call `xform` instantiates the function template `add<5, int>` to add `5` to a passed `int` value. The resulting function is called for each element in the source collection `iarr`, while it is translated into the destination collection `iarr2`. Note that you've to specify the argument `int` for the template parameter `T` of `add<>`. Deduction only works for immediate calls and `xform` needs a complete type to deduce the type of its fourth parameter. There is no support to substitute/deduce only some template parameters and then see, what could fit, and deduce the remaining parameters.

Restrictions for nontype template parameters

A nontype parameter may be an integral type, or a pointer or lvalue reference to an object or to a function type. Floating-point numbers are not allowed as nontype template parameters:

```
template<double VAT>       // ERROR: floating-point values are not
double process(double v) { // allowed as template parameters
  return v * VAT;
}
```

A template nontype parameter is a constant value inside the template definition. A nontype parameter can be used when constant expressons are required, for example, to specify the size of an array.

Default function template arguments

Just as we can supply default arguments to function parameters, we can also supply default template arguments. As an example, we'll write the function template `add` to use a default value:

```
template<typename T, int Val = 5>
T add(T x) {
  return x + Val;
}
```

We can use this function template to transform the values of an array and store the values in an other array:

```
template<typename In, typename Out, typename Func>
Out xform(In first, In last, Out result, Func f) {
  while (first != last) {
    *result = f(*first); ++first; ++result;
  }
  return result;
}

int iarr[5] {1, 2, 3, 4, 5}, iarr2[5];
// iarr2 will now have values: 11, 12, 13, 14, 15
xform(iarr, iarr+5, iarr2, add<int, 10>);
// iarr2 will now have values: 6, 7, 8, 9, 10
xform(iarr, iarr+5, iarr2, add<int>);
```

We can also provide a default value to the type parameter `T` for `add`:

```
template<typename T = int, int Val = 5>
T add(T x) {
  return x + Val;
}

int iarr[5] {1, 2, 3, 4, 5}, iarr2[5];
// iarr2 will now have values: 11, 12, 13, 14, 15
xform(iarr, iarr+5, iarr2, add<int, 10>);
// iarr2 will now have values: 6, 7, 8, 9, 10
xform(iarr, iarr+5, iarr2, add<>); // add<int, 5>

double darr[5] = {1.1, 2.2, 3.3, 4.4, 5.5}, darr2[5];
xform(darr, darr+5, darr2, add<double, 1>);
```