Variable Template, Alias Template & Variadic Template

Variable Templates

- A variable template defines a family of variables or static data members
- They are constexpr and perform compile time computation

Syntax:

```
template < parameter-list > variable-declaration
```

- When not explicitly specialized or explicitly instantiated, it is implicitly instantiated where a specialization of the variable template is used.
- Variable templates can be partially/explicitly specialized

Typedef

- Introduces a name for an existing type
- Useful to construct shorter more meaningful name for types
- Encapsulates implementation details that may change
- Only introduces new name for existing type – not a new type

```
typedef unsigned int UINT;
typedef long long LLONG;
```

```
typedef std::vector<int> IntVector;
void foo(const IntVector& s);
IntVector istack[10];
```

```
struct Employee {};
typedef std::vector<std::list<Employee>> Org;
Org team;
```

Type Alias (C++ 11)

- Creates a name that is a synonym of existing type
- Does not introduce a new type
- Creates through using keyword
- Natural, similar to declaring variables

```
using UINT = unsigned int;
using LLONG = long long;
```

```
using IntVector = std::vector<int>;
void foo(const IntVector& s);
IntVector istack[10];
```

```
struct Employee {};
using Org = std::vector<std::list<Employee>>;
Org team;
```

Alias Templates

- Typedef cannot be templatized

```
struct Employee {};
typedef std::vector<std::list<std::string>> Org;
Org team;
Org<Employee> names; // list of Employee.. Does not work
```

- Alias templates allow templatization of different types while allowing typedef type synonyms.

```
template<typename T>
using Org = std::vector<std::list<T>>;
Org<std::string> names; // list of strings
Org<Employee> name1;  // list of Employee
```

Variadic Templates

- Since C++11, templates can have parameters that accept a variable number of template arguments.
- C variadic functions which is generally referred to as the varargs construct is the first implementation of a mechanism to support variable arguments
- With C++11 implementation using templates, the compiler is able to guarantee type safety and deduce the number of applied parameters following a function call.

Variadic Templates – Parameter Packs

- **Syntax**

```
template<typename T1, typename… Args>
void println(std::ostream& out, T1 t, Args… args) {
    sizeof...(args);
};
```

```
println(std::cout, 10, 20.3f, "Hello World!")  // expands to below:
println(std::ostream& std::cout, int arg1, float arg2, const char* arg3);
```

Folding in C++11

- In C++11, folding may be implemented using recursive templates
- Folding is the implicit destructuring of the parameter pack by passing a head argument as a separate parameter to the function
- To correctly terminate the recursion, an identity function is needed

Folding in C++11

```
void println(std::ostream& out) {
    out << "\n";
}

template<typename T1, typename ... T2>
void println(std::ostream& out, T1 t1, T2 ...t2) {
    out << t1 << " ";
    println(out, t2...);
}
```

```
println(std::cout, 10, 20.3f, "Hello World!");
10 println(std::cout, 20.3f, "Hello World!");
10 20.3f println(std::cout, "Hello World!");
10 20.3f "Hello World!" println(std::cout);
```

Fold Expressions

- A fold expression performs a fold of a template parameter pack over a binary operator.
- Solves the problem of recursive template folding in C++

An expression of the form (... op e) or (e op ...), where op is a fold-operator and e is an unexpanded parameter pack, are called unary folds.

An expression of the form (e1 op ... op e2), where op are fold-operators, is called a binary fold. Either e1 or e2 is an unexpanded parameter pack, but not both.

- https://en.cppreference.com/w/cpp/language/fold

Fold Expressions


```cpp
template<typename ... T2>
void println(std::ostream& out, T2 ...t2) {
    ((std::cout << sep << t2), ...);
}
println(std::cout, 10, 20.3f, "Hello World!");
```

```cpp
template<typename... Arg>
auto adder(Arg...args) {
    std::cout << "Func called";
    return (args + ... + 0);
}
std::cout << adder(10, 20, 30, 40) << "\n";
std::cout << adder();
```

Types of Templates

- **Function Template** - defines a family of functions

```
template <typename T>
void Hello(T arg);
```

- **Class Template** - defines a family of classes

```
template <typename T>
class Hello;
```

- **Variable templates** (C++ 14) – variables that are templated

```
template<class T>
constexpr T pi = T(3.14);
```

- **Alias Templates** (C++11) – alias to a family of types

```
template<class T>
using ptr = T*;
```