Intermediate Level C++

# C++ Templates

Yong Zhang

## Objectives

**In this chapter you will learn:**

- Introduction to Templates
- Template Function
- Template Class
- Template Specification
- Variadic Templates

## Templates

- **C++ Templates allow functions and classes to operate with generic types.**
  - allow a function or class to work on many different data types
  - are resolved at compile time to speed up the program
  - no runtime checks with limited flexibility
  - rely on Operator overloads
  - Many data structures and algorithms in the standard library are written based on templates

# Template Functions

- **Template function is defined with a placeholder for any possible types**

```
template <class placeholder> function_declaration
```

- **The placeholder is used as the type throughout the function.**
- **The compiler will work out the type during the compilation time.**
- **Demo: FunctionTemplateDemo**

# Function Template Default Arguments

- **Default types are now allowed in function template (C++11)**

```
template<typename T, int N=1>
T& increment(T& i)
```

```
template< typename T, typename C = std::less<T> >
T FindExtreme(std::vector<T> &v, C c = C())
```

- **Demo: FunctionTemplateDefaultArgumentDemo**

# Template Classes

- **Template class is defined with a placeholder for any possible types**

```
template <class placeholder> class_declaration
```

- **The placeholder is used as the type throughout the declaration and definition of the class.**
- **When using the class, specify the type**

```
ClassName<Type> Varname
```

- **Demo: TemplateClassDemo**

## Template Specification

- **Sometimes templates don't work for some types:**
  - Operator overloads are not available.
  - Operator overloads are not what you want.
- **Solutions:**
  - Modify or add necessary operator overloads
  - Specialized the template for a particular type

```
template <>
class_declaration<ParitcularType>
```

- **Demo: TemplateClassDemo**

# Variadic Template

- **Variadic templates allow an unspecified number and type of arguments:**
  - An ellipses is used to indicate 0 or more type parameters
  - The name follows the ellipses is the parameter pack:

```
template<typename Stream, typename... Columns>
class CSVPrinter
{
public:
    void output_line(const Columns&... columns);
    // other methods, constructors etc. not shown
};
```

- **Variadic templates can be used to**
  - Perform type computation at compile time
  - Generate type structure
  - Implement type-safe functions with arbitrary number of arguments
  - Perform argument forwarding

# Variadic Template – Parameter Packs

- **Operations associated with parameter packs:**
  - Perform pack expansion
  - Obtain the type count
- **Pack Expansion:**

```
void output_line(const Columns&... columns)
{
    write_line(validate(columns)...);
}
```

```
template<typename Value, typename... Values>
void write_line(const Value& val, const Values&... values) const
{
    write_column(val, _sep);
    write_line(values...);
}

template<typename Value>
void write_line(const Value &val) const
{
    write_column(val, "\n");
}

void output_line(const Columns&... columns);
```

# Variadic Template – Parameter Packs

- **Obtain the Pack Count**

```
template<typename Stream, typename... Columns>
class CSVPrinter
{
    Stream& _stream;
    array<string, sizeof...(Columns)> _headers;
    // rest of implementation
};
```

# Variadic Template – Parameter Packs

- **Traversing Template Parameter Packs**

```
template<typename... Types>                    // allow zero parameters
struct TupleSize;

template<typename Head, typename... Tail> // traverse types
struct TupleSize<Head, Tail...>
{
    static const size_t value = sizeof(Head) + TupleSize<Tail...>::value;
};

template<> struct TupleSize<>                  // end recursion
{
    static const size_t value = 0;
};


TupleSize<>::value;                       // 0
TupleSize<int, double, char>::value;  // 13 on a 32-bit platform
```

# Variadic Template – Parameter Packs

- **Constraining Parameter Packs to One Type**

```
template<typename... Strings>
void output_strings(const string& s, const Strings&... strings) const
{
    write_column(s, _sep);
    output_strings(strings...);
}

void output_strings(const string& s) const
{
    write_column(s, "\n");
}
```

## Using Variadic Template – A Example: std::tuple

- Like a std::pair, but holds any number of elements.
- Saving wrting little class or struct just to hold a clump of values.
- Comparison operators are already implemented.
- Demo: DemoTuple:
  - create with uniform initialization:

```
std::tuple<int, std::string, double>
   entry { 1, "Kate", 100.0 };
```

  - Use std::make_tuple
  - Use std::get<position>(tupleinstance) to access or set values
- Demo: TupleDemo

# Writing Variadic Templates

- **Declare a class template:**

```
template <class... Ts>
class Foo {
// class details
};
```

- **Declare a function template:**

```
template <class... Ts>
void foo(Ts... vals)
{
// function body
}
```

## Writing Variadic Templates (continued)

- **Parameter Pack: Ts… vals, you can**
  - work with it using the expansion operator, …
  - Pass it to another template or a function
  - Use it in an initializer list
  - Capture it with a lambda
- **Two ways to use the parameter pack:**
  - Forwarding
  - Recursive-ish templates

# Writing Variadic Templates (continued)

- **Forwarding:**

```
template<class _T, class... _Types> inline
unique_ptr<_T> make_unique(_Types&&... _Args)
{
return (unique_ptr<_T>(new _T(std::forward<_Types>(_Args)...)));
}
```

## Writing Variadic Templates (continued)

- **Recursive-ish templates**
  - You can write many templates for the same class or function
  - Specific templates might take 0, 1, or 2 arguments
  - Compiler will choose the appropriate template
  - The variadic template can use the specific templates to get the work done
  - Not really recursion, just using another template with the same name, but feels recursive
- **Demo: MatchesDemo**

# Writing Variadic Templates (continued)

- **The expansion operator …**

```
echo(A<Ts...>::b(vs...)); //calls b once
echo(A<Ts...>::b(vs)...); //calls b(T) Nx A of T,U,V etc
echo(A<Ts>::b(vs)...); //calls N different A::b (each w one T)
```

- **Demo: DemoExpansionOperator**

## Variadic Template – Summary

- **Variadic templates are to templates as template functions are to functions, or template classes are to classes**
- **Writing variadic templates solves some interesting problems**
- **Like all templates, if someone else writes them you don't have to**
- **If you can write a regular template, you can write a variadic one**
  - Just have to learn the expansion rules
  - Be prepared to read a lot of compiler error messages