B2B: C++ Templates

Part 2

Andreas Fertig
https://AndreasFertig.Info
post@AndreasFertig.Info
@Andreas__Fertig

Variadic templates: Parameter pack

- Syntax:

(A) `typename|class... Ts` generates a type template parameter pack with optional name.

(B) `Args... ts` a function argument parameter pack with optional name.

(C) `sizeof...(ts)` determine the number of arguments passed.

(D) `ts...` in the body of a function to unpack the arguments.

```
template<typename T,
         typename... Ts   (A) Variadic template
         >
constexpr auto
min(const T& a,
    const T& b,
    const Ts&... ts)   (B) Parameter pack
{
  const auto m = a < b ? a : b;
  if constexpr(sizeof...(
                 ts)   (C) size of a pack
                 > 0) {
    return min(m, ts...);   (D) Expand the pack
  }

  return m;
}

static_assert(min(3, 2, 3, 4, 5) == 2);
static_assert(min(3, 2) == 2);
```

Variadic templates

- With C++11, there are *variadic templates*:
  - Variadic templates are templates that take any number of parameters.
  - Already known by variadic macros or variadic functions.

```
A Helper functions to convert everything into a std::string
auto Normalize(const std::string& t) { return t; }
auto Normalize(const QString& t) { return t.toStdString(); }
auto Normalize(const char* t) { return std::string{t}; }

B Catch all others and apply to_string
template<class T> auto Normalize(const T& t) { return std::to_string(t); }

template<typename T, typename... Ts> auto _StrCat(std::string& ret, const T& targ, const Ts&... args)
{
  ret += Normalize(targ);
  if constexpr(sizeof...(args) > 0) {
    _StrCat(ret, args...);  C Do, as long as the pack has elements
  }
}

template<typename T, typename... Ts> auto StrCat(const T& targ, const Ts&... args)
{
  std::string ret{Normalize(targ)};

  _StrCat(ret, args...);  D Start the recursion to expand the pack

  return ret;
}
```


Fold Expressions

C++17

- Used to unpack a parameter pack using an operation.
  - Saves the recursion.
- Syntax:
  - unary { right fold: (pack *op* ...) ; left fold: (... *op* pack) }
  - binary { right fold: (pack *op* ... *op* init) ; left fold: (init *op* ... *op* pack) }
- Note:
  - All *op* must be the same operation.
  - Parenthesis around the epxression are required to make it a fold epxression.

```
template<typename T, typename... Ts>
void Print(const T& targ, const Ts&... args)
{
  std::cout << targ;
  auto coutSpaceAndArg = [](const auto& arg) {
    std::cout << ' ' << arg;
  };

  (..., coutSpaceAndArg(args));  A Unary left fold
}

int main()
{
  Print("Hello", "C++", 20);
}
```


https://AndreasFertig.Info
post@AndreasFertig.Info

C++17

Fold Expressions

```
Ⓐ Normalize functions for 'normal' strings
auto                   Normalize(const std::string& t) { return t; }
auto                   Normalize(const QString& t) { return t.toStdString(); }
auto                   Normalize(const char* t) { return std::string{t}; }
template<class T> auto Normalize(const T& t) { return std::to_string(t); }

Ⓑ Variadic template for concatenating the parts
template<class T, class... Ts> auto BuildCSVLine(const T& targ, const Ts&... args)
{
  auto ret{Normalize(targ)};
  auto addColonAndNormalize = [&](const auto& arg) {
    ret += ',';
    ret += Normalize(arg);
  };

  (..., addColonAndNormalize(args));   Ⓒ A unary left fold

  return ret;
}
```


C++17

Fold Expressions

```
Ⓐ Normalize functions for 'normal' strings
auto                   Normalize(const std::string& t) { return t; }
auto                   Normalize(const QString& t) { return t.toStdString(); }
auto                   Normalize(const char* t) { return std::string{t}; }
template<class T> auto Normalize(const T& t) { return std::to_string(t); }

Ⓑ Variadic template for concatenating the parts
template<class T, class... Ts> auto BuildCSVLine(const T& targ, const Ts&... args)
{
  auto ret{Normalize(targ)};
  auto addColonAndNormalize = [&](const auto& arg) {
    ret += ',';
    ret += Normalize(arg);
  };

  (..., addColonAndNormalize(args));   Ⓒ A unary left fold

  return ret;
}
```



```
void Main()
{
  QString qs(L"@CppCon");
  auto    s = BuildCSVLine("Hello", std::string{"C++"}, 20, qs);
  printf("%s\n", s.c_str());
}
```

C++17

Variadic templates: data structure comparisons

- Comparison of data structures often paperwork with a tendency to errors.
  - For example, checking whether one MAC address is valid or equal to another.
- Goal:
  - Code, as shown on the right.
    - A Simple comparison of two equal-sized arrays.
    - B Compare all fields of an array to a specific value.

```
struct MACAddress
{
  unsigned char value[6];
};

void Main()
{
  constexpr MACAddress macA{2, 2, 2, 2, 2, 2};
  constexpr MACAddress macB{2, 2, 2, 2, 2, 4};
  constexpr MACAddress macC{2, 2, 2, 2, 2, 2};

  A Compare each element against value
  static_assert(Compare(macA.value, 2));
  static_assert(!Compare(macB.value, 2));

  B Compare two equally sized arrays
  static_assert(!Compare(macA.value, macB.value));
  static_assert(Compare(macA.value, macC.value));
}
```


C++17

Variadic templates

```
namespace details::array_single_compare {
  template<typename T, size_t N, typename U, size_t... I>
  constexpr bool Compare(const T (&a)[N], const U& b, std::index_sequence<I...>)
  {
    return ((a[I] == b) && ...);
  }
} /* namespace details::array_single_compare */

template<typename T, size_t N, typename U>  A Compare each element against value
constexpr bool Compare(const T (&a)[N], const U& b)
{
  return details::array_single_compare::Compare(a, b, std::make_index_sequence<N>{});
}

namespace details::array_compare {
  template<typename T, size_t N, size_t... I>
  constexpr bool Compare(const T (&a)[N], const T (&b)[N], std::index_sequence<I...>)
  {
    return ((a[I] == b[I]) && ...);
  }
} /* namespace details::array_compare */

template<typename T, size_t N>  B Compare two equally sized arrays
constexpr bool Compare(const T (&a)[N], const T (&b)[N])
{
  return details::array_compare::Compare(a, b, std::make_index_sequence<N>{});
}
```

C++20

Variadic templates

```
template<typename T, size_t N, typename U>  (A) Compare each element against value
constexpr bool Compare(const T (&a)[N], const U& b)
{
  return [&]<size_t... I>(std::index_sequence<I...>) { return ((a[I] == b) && ...); }
  (std::make_index_sequence<N>{});
}

template<typename T, size_t N>  (B) Compare two equally sized arrays
constexpr bool Compare(const T (&a)[N], const T (&b)[N])
{
  return [&]<size_t... I>(std::index_sequence<I...>) { return ((a[I] == b[I]) && ...); }
  (std::make_index_sequence<N>{});
}
```


C++14

Variable templates

- Variables can now also become templates.
  - With them, we can define constants like π or `true_type`
- This makes some template metaprogramming (TMP) code more readable.
  - (B) is only an alias.

```
(A) Helper to store a value at compile-time
template<class T, T v>
struct integral_constant
{
  static constexpr T value = v;
};

(B) Aliases for clean TMP
using true_type  = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

(C) Base is_pointer template
template<class T>
struct is_pointer : false_type
{
};

(D) is_pointer specialization for T*
template<class T>
struct is_pointer<T*> : true_type
{
};

(E) Test it
static_assert(is_pointer<int*>::value);
static_assert(not is_pointer<int>::value);
```

C++14

Variable templates

- Variables can now also become templates.
  - With them, we can define constants like π or `true_type`
- This makes some TMP code more readable.
  - With this new version, (B) defines a new variable.
  - The two together make TMP much more readable in a lot of places.

```
(A) As seen before
template<class T, T v>
struct integral_constant
{
  static constexpr T value = v;
};

using true_type  = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

template<class T>
struct is_pointer : false_type
{
};

template<class T>
struct is_pointer<T*> : true_type
{
};

(B) A variable template to access ::value
template<typename T>
constexpr auto is_pointer_v = is_pointer<T>::value;

(C) is_pointer_v looks cleaner than ::value
static_assert(is_pointer_v<int*>);
static_assert(not is_pointer_v<int>);
```


SFINAE

- With templates, we have a technique called substitution failure is not an error (SFINAE).
  - When the compiler looks into an instantiation of a template, and it turns out that instantiation fails, this failure is silently discarded.
  - In the end, there needs to be a template that works for the type otherwise we will see a compiler error.
  - However, this allows us to do more than just specialisation or overloads.
- Here we have `equal` and a specialization for **`double`**. But what is with **`float`**?
  - We can add another specialization, or we can use SFINAE.

```
template<typename T>
bool equal(const T& a, const T& b)
{
  return a == b;
}

template<>
bool equal(const double& a, const double& b)
{
  return std::abs(a - b) < 0.00001;
}

void Main()
{
  int a = 2;
  int b = 1;

  printf("%d\n", equal(a, b));

  double d = 3.0;
  double f = 4.0;

  printf("%d\n", equal(d, f));
}
```

SFINAE

- With templates, we have a technique called SFINAE.
  - When the compiler looks into an instantiation of a template, and it turns out that instantiation fails, this failure is silently discarded.
  - In the end, there needs to be a template that works for the type otherwise we will see a compiler error.
  - However, this allows us to do more than just specialisation or overloads.
- Here we have `equal` and a specialization for **`double`**. But what is with **`float`**?
  - We can add another specialization, or we can use SFINAE.
- Where to put the SFINAE condition?
  - You can put it as an additional default parameter in the template-head, as a default function parameter or on the return-type.
  - As a guideline, start by putting it somewhere, a user does not see it. Only if that is not possible, put it in another place.

```cpp
template<typename T>
std::enable_if_t<not std::is_floating_point_v<T>, bool>
equal(const T& a, const T& b)
{
  return a == b;
}

template<typename T>
std::enable_if_t<std::is_floating_point_v<T>, bool>
equal(const T& a, const T& b)
{
  return std::abs(a - b) < 0.00001;
}

void Main()
{
  int a = 2;
  int b = 1;

  printf("%d\n", equal(a, b));

  double d = 3.0;
  double f = 4.0;

  printf("%d\n", equal(d, f));
}
```


Tag dispatch

- Another option instead of SFINAE is tag dispatch.
  - We use an empty class *tag*.
  - These tags are passed to functions as an additional parameter.
  - This makes overloading of functions for otherwise identical parameters possible.
  - In terms of performance, this comes for free. As we talk about templates, the compiler can see that the tag is never used and optimizes the parameter away.

```cpp
namespace internal {
  struct notFloatingPoint {};
  struct floatingPoint {};

  template<typename T>
  bool equal(const T& a, const T& b, notFloatingPoint)
  {
    return a == b;
  }

  template<typename T>
  bool equal(const T& a, const T& b, floatingPoint)
  {
    return std::abs(a - b) < 0.00001;
  }
}  // namespace internal

template<typename T>
bool equal(const T& a, const T& b)
{
  using namespace internal;

  if constexpr(std::is_floating_point_v<T>) {
    return equal(a, b, floatingPoint{});
  } else {
    return equal(a, b, notFloatingPoint{});
  }
}
```

C++17

Tag dispatch & fold expressions

- Do you remember this example form earlier?
  - It lacks wide-string support.
  - Let’s use tag dispatch to add the missing functionality.

```
A Normalize functions for ’normal’ strings
auto                    Normalize(const std::string& t) { return t; }
auto                    Normalize(const QString& t) { return t.toStdString(); }
auto                    Normalize(const char* t) { return std::string{t}; }
template<class T> auto Normalize(const T& t) { return std::to_string(t); }

B Variadic template for concatenating the parts
template<class T, class... Ts> auto BuildCSVLine(const T& targ, const Ts&... args)
{
  auto ret{Normalize(targ)};
  auto addColonAndNormalize = [&](const auto& arg) {
    ret += ’,’;
    ret += Normalize(arg);
  };

  (..., addColonAndNormalize(args));  C A unary left fold

  return ret;
}
```


C++17

Tag dispatch & fold expressions

```
struct locale_s {};  A The tag-types
struct locale_ws {};
B Normalize functions for ’normal’ strings
auto                    Normalize(const std::string& t, locale_s) { return t; }
auto                    Normalize(const std::wstring& t, locale_s) { return to_string(t); }
auto                    Normalize(const QString& t, locale_s) { return t.toStdString(); }
auto                    Normalize(const char* t, locale_s) { return std::string{t}; }
auto                    Normalize(const wchar_t* t, locale_s) { return Normalize(std::wstring{t}, locale_s{}); }
template<class T> auto Normalize(const T& t, locale_s) { return std::to_string(t); }
C Normalize functions for ’wide’ strings
auto                    Normalize(const std::string& t, locale_ws) { return to_wstring(t); }
auto                    Normalize(const std::wstring& t, locale_ws) { return t; }
auto                    Normalize(const QString& t, locale_ws) { return t.toStdWString(); }
auto                    Normalize(const char* t, locale_ws) { return Normalize(std::string{t}, locale_ws{}); }
auto                    Normalize(const wchar_t* t, locale_ws) { return std::wstring{t}; }
template<class T> auto Normalize(const T& t, locale_ws) { return std::to_wstring(t); }
D Variadic template for concatenating the parts
template<class LT = locale_s, class T, class... Ts> auto BuildCSVLine(const T& targ, const Ts&... args)
{
  auto ret{Normalize(targ, LT{})};
  auto addColonAndNormalize = [&](const auto& arg) {
    ret += ’,’;
    ret += Normalize(arg, LT{});
  };

  (..., addColonAndNormalize(args));  E A unary left fold

  return ret;
}
F Wide string version of the above variadic template for concatenating the parts
template<class... Ts> auto BuildWCSVLine(const Ts&... args) { return BuildCSVLine<locale_ws>(args...); }
```

C++20

requires

- With C++20 we can replace most SFINAE with Concepts.
  - They may look like SFINAE but are much more powerful.
  - Plus, it makes our code even cleaner and more expressive.

```
template<typename T>
requires(not std::is_floating_point_v<T>) bool equal(
  const T& a,
  const T& b)
{
  return a == b;
}

template<typename T>
requires(std::is_floating_point_v<T>) bool equal(
  const T& a,
  const T& b)
{
  return std::abs(a - b) < 0.00001;
}

void Main()
{
  int a = 2;
  int b = 1;

  printf(”%d\n”, equal(a, b));

  double d = 3.0;
  double f = 4.0;

  printf(”%d\n”, equal(d, f));
}
```


Template template parameters

- They can be seen as nested templates.
- When we have a template parameter which itself is a template, and its parameters are deduced, we have a template template parameter.
  - We first declare the template template parameter, the names of the template parameter are omitted (A). Only the name we give this template template parameter matters.
  - Now a template can take the arguments for the template template parameter (B). We can use defaults as usual (C).
  - After that, the template parameter can be instantiated by applying the template parameters to the template template parameter.

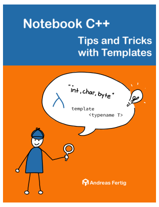
```
template<
  (A) A template template parameter
  template<class, class>
  class Container,
  (B) 1. Parameter for Container
  class T,
  (C) 2. Parameter for Container
  class Allocator = std::allocator<T>>
void Fun(const Container<T, Allocator>& c)
{
  for(const auto& e : c) {
    printf(”%d\n”, e);
  }
}

int main()
{
  std::vector<int> v{2, 3, 4};
  Fun(v);

  std::list<char> l{’a’, ’b’, ’c’};
  Fun(l);
}
```

}

I am Fertig.



bit.ly/cppcon2020

Discounted version for you!

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - g++ 10.2.0
  - clang version 10.0.0 (https://github.com/llvm/llvm-project.git d32170dbd5b0d54436537b6b75beaf44324e0c28)

Typography

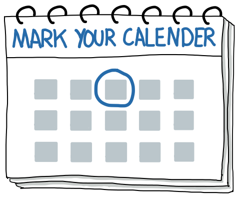
- Main font:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code font:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/

References

Images:
22: Franziska Panter

Upcoming Events

For my upcoming talks you can check https://andreasfertig.info/talks/.
For my training services you can check https://andreasfertig.info/training/.



https://AndreasFertig.Info
post@AndreasFertig.Info

About Andreas Fertig

Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig is the CEO of Unique Code GmbH, which offers training and consulting for C++ specialized in embedded systems. He worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.
Andreas is involved in the C ++ standardization committee. He is a regular speaker at conferences internationally. Textbooks and articles by Andreas are available in German and English.
Andreas has a passion for teaching people how C++ works, which is why he created C++ Insights (cppinsights.io).

https://AndreasFertig.Info
post@AndreasFertig.Info