TMP: template meta-programming (III)

Deep dive into C++

scc@teamt5.org

TEAMT5 杜浦數位安全

Persistent Cyber Threat Hunters

Outline

Template 101 and type deduction

Perfect forwarding and reference collapsing

Variadic template and parameter pack

SFINAE

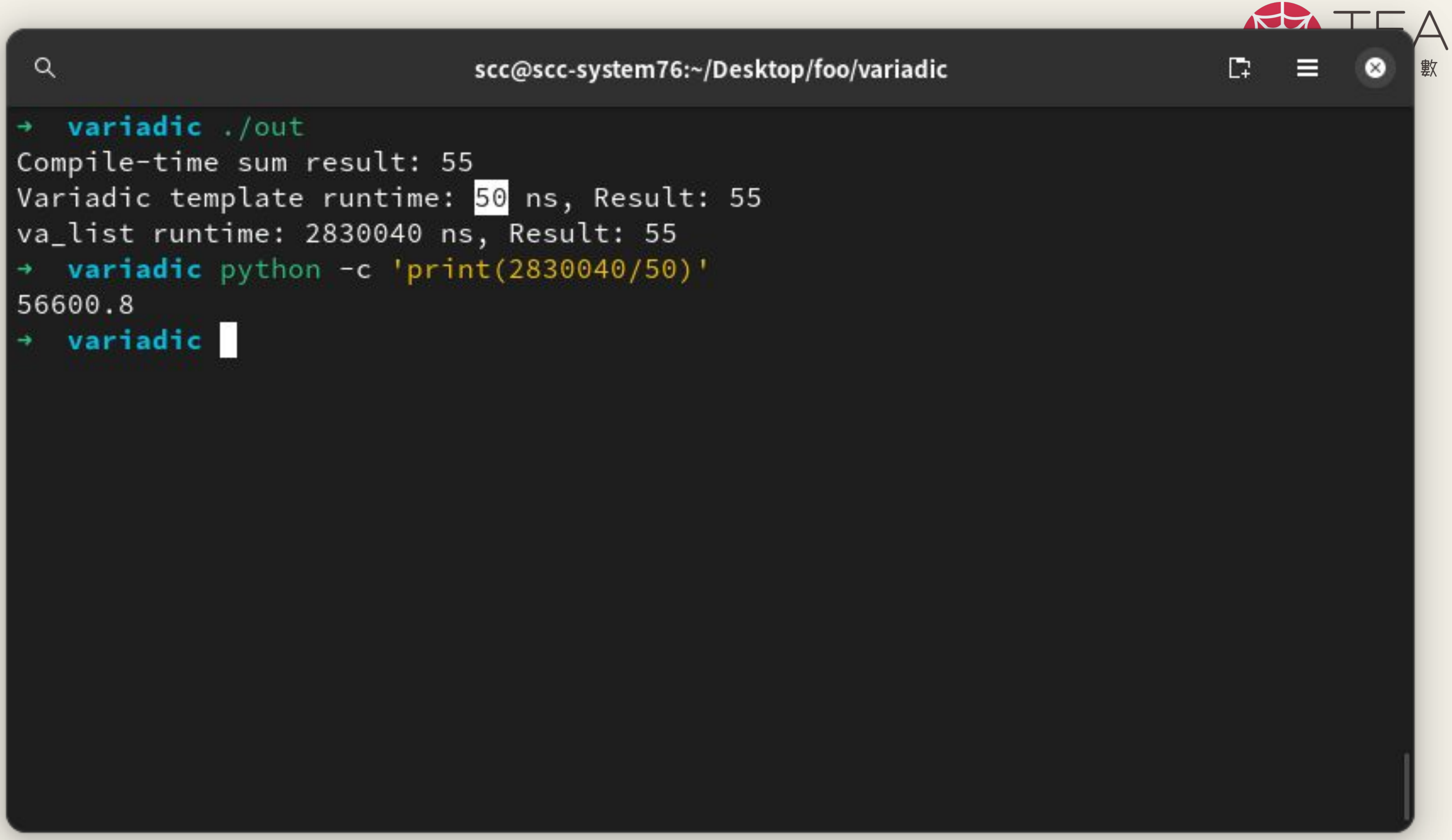
CTAD

Expression template

Concepts and constraints

CRTP and compile time polymorphism

Variadic template



```
➜  variadic ./out
Compile-time sum result: 55
Variadic template runtime: 50 ns, Result: 55
va_list runtime: 2830040 ns, Result: 55
➜  variadic python -c 'print(2830040/50)'
56600.8
➜  variadic
```

https://gist.github.com/25077667/6821c2d3f6a1f029dbae840707320291

Variadic template

```
template <typename T>
constexpr T sum(T a) {
   return a;
}

template <typename T, typename...Args>
constexpr T sum(T first, Args...
args) {
   return first + sum(args...);
}
```

```
int sumCStyle(int count, ...) {
   int total = 0;
   va_list args;
   va_start(args, count);

   for (int i = 0; i < count; ++i)
       total+=va_arg(args,int);
   va_end(args);
   return total;
}
```

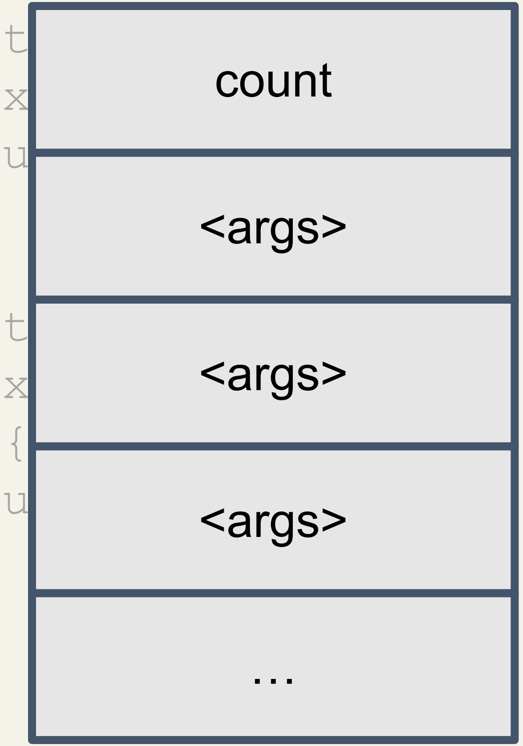
Variadic template

```
template <typename T>
constexpr T sum(T a) {
   return a;
}

template <typename T, typename...Args>
constexpr T sum(T first, Args...
args) {
   return first + sum(args...);
}
```

```
int sumCStyle(int count, ...) {
   int total = 0;
   va_list args;
   va_start(args, count);

   for (int i = 0; i < count; ++i)
       total+=va_arg(args,int);
   va_end(args);
   return total;
}
```

Variadic template

```
templat
constex
    retu
}

templat                    ..Args>
constex                    Args...
args) {
    retu                   gs...);
}
```



count

<args>

<args>

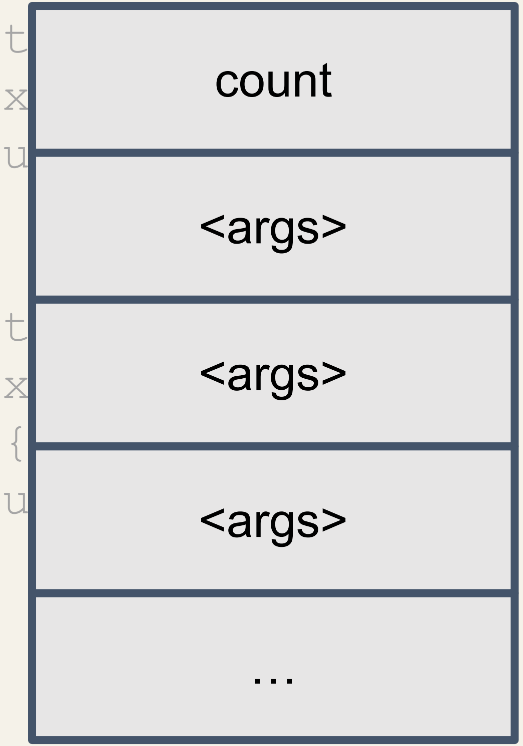
<args>

...

```
int sumCStyle(int count, ...) {
    int total = 0;
    va_list args;
    va_start(args, count);

    for (int i = 0; i < count; ++i)
        total+=va_arg(args,int);
    va_end(args);
    return total;
}
```

Variadic template

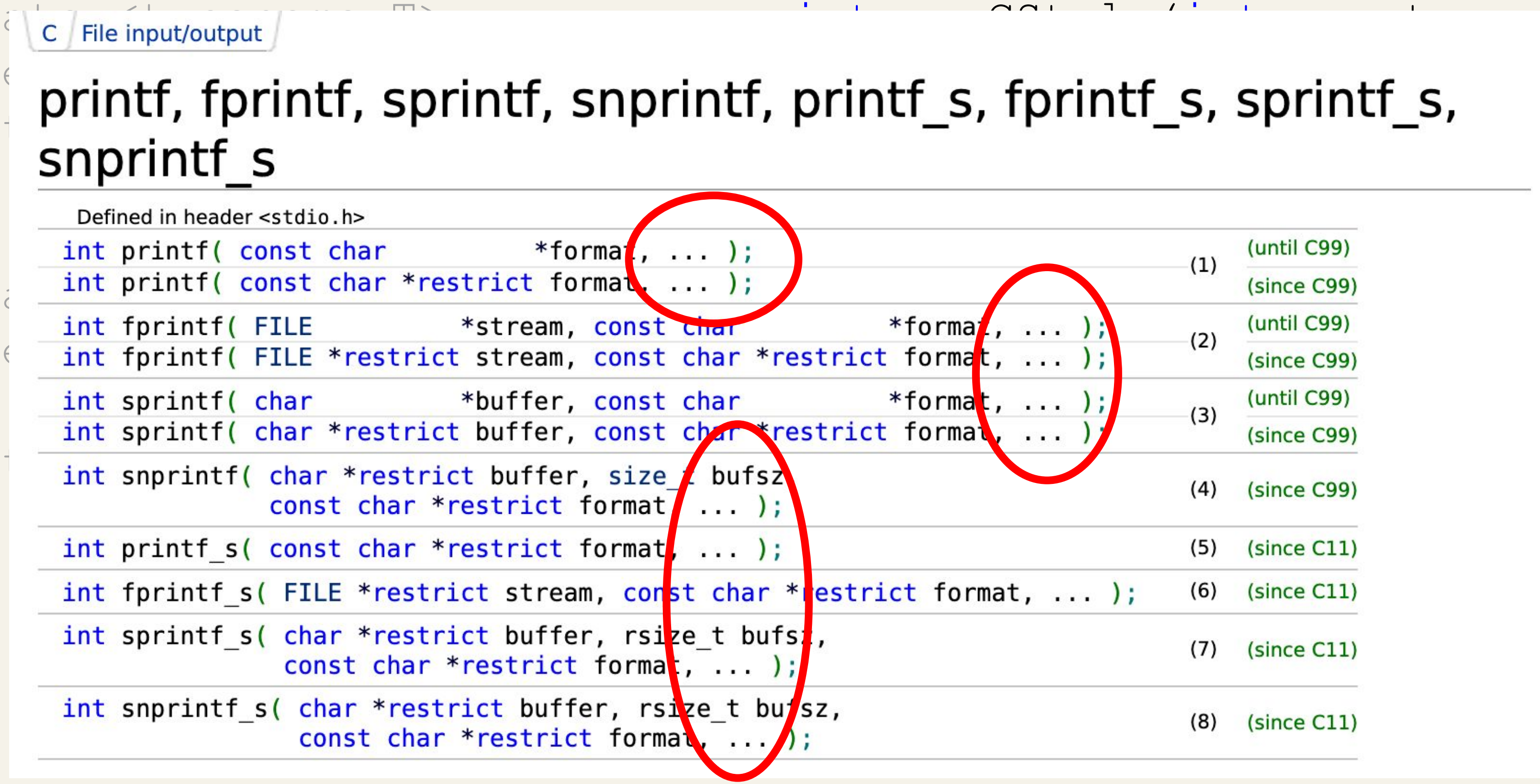
```
templat
constex
    retu
}

templat                ..Args>
constex                Args...
args) {
    retu              gs...);
}
```



```
int sumCStyle(int count, ...) {
    int total = 0;
    va_list args;
    va_start(args, count);

    for (int i = 0; i < count; ++i)
        total+=va_arg(args,int);
    va_end(args);
    return total;
}
```

Variadic template

```
templa                                                    .)  {
conste
    re
}

templa
conste                                                     nt);
args)
    re
}
```



C / File input/output

printf, fprintf, sprintf, snprintf, printf_s, fprintf_s, sprintf_s, snprintf_s

Defined in header `<stdio.h>`

```
int printf( const char          *format, ... );                              (1)  (until C99)
int printf( const char *restrict format, ... );                                   (since C99)
int fprintf( FILE          *stream, const char          *format, ... );     (2)  (until C99)
int fprintf( FILE *restrict stream, const char *restrict format, ... );          (since C99)
int sprintf( char          *buffer, const char          *format, ... );     (3)  (until C99)
int sprintf( char *restrict buffer, const char *restrict format, ... );          (since C99)
int snprintf( char *restrict buffer, size_t bufsz,
              const char *restrict format, ... );                           (4)  (since C99)
int printf_s( const char *restrict format, ... );                           (5)  (since C11)
int fprintf_s( FILE *restrict stream, const char *restrict format, ... );   (6)  (since C11)
int sprintf_s( char *restrict buffer, rsize_t bufsz,
               const char *restrict format, ... );                          (7)  (since C11)
int snprintf_s( char *restrict buffer, rsize_t bufsz,
                const char *restrict format, ... );                         (8)  (since C11)
```

Variadic template

```
template <typename T>
constexpr T sum(T a) {
   return a;
}

template <typename T, typename...Args>
constexpr T sum(T first, Args...
args) {
   return first + sum(args...);
}
```

```
int sumCStyle(int count, ...) {
   int total = 0;
   va_list args;
   va_start(args, count);

   for (int i = 0; i < count; ++i)
       total+=va_arg(args,int);
   va_end(args);
   return total;
}
```

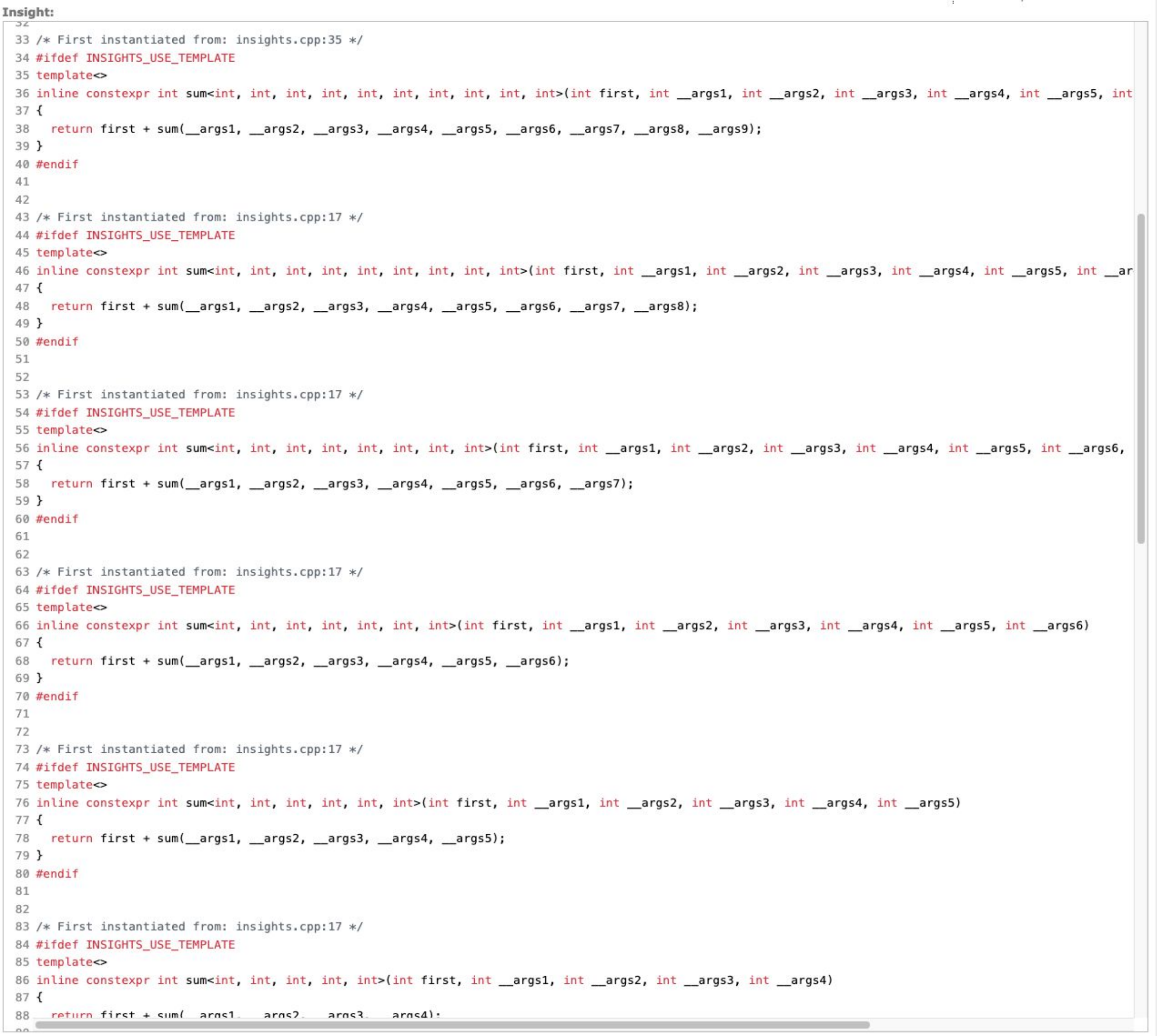
Variadic template

```
template <typename T>
constexpr T sum(T a) {
   return a;
}

template <typename T, typename ...Args>
constexpr T sum(T first, Args... args) {
   return first + sum(args...);
}
```

```
int sumCStyle(int count, ...) {
   int total = 0;
   va_list args;
   va_start(args, count);

   for (int i = 0; i < count; ++i)
       total+=va_arg(args,int);
   va_end(args);
   return total;
}
```

Variadic templa

```
template <typename T>
constexpr T sum(T a) {
   return a;
}

template <typename T, typ
constexpr T sum(T first, 
   return first + sum(arg
}
```

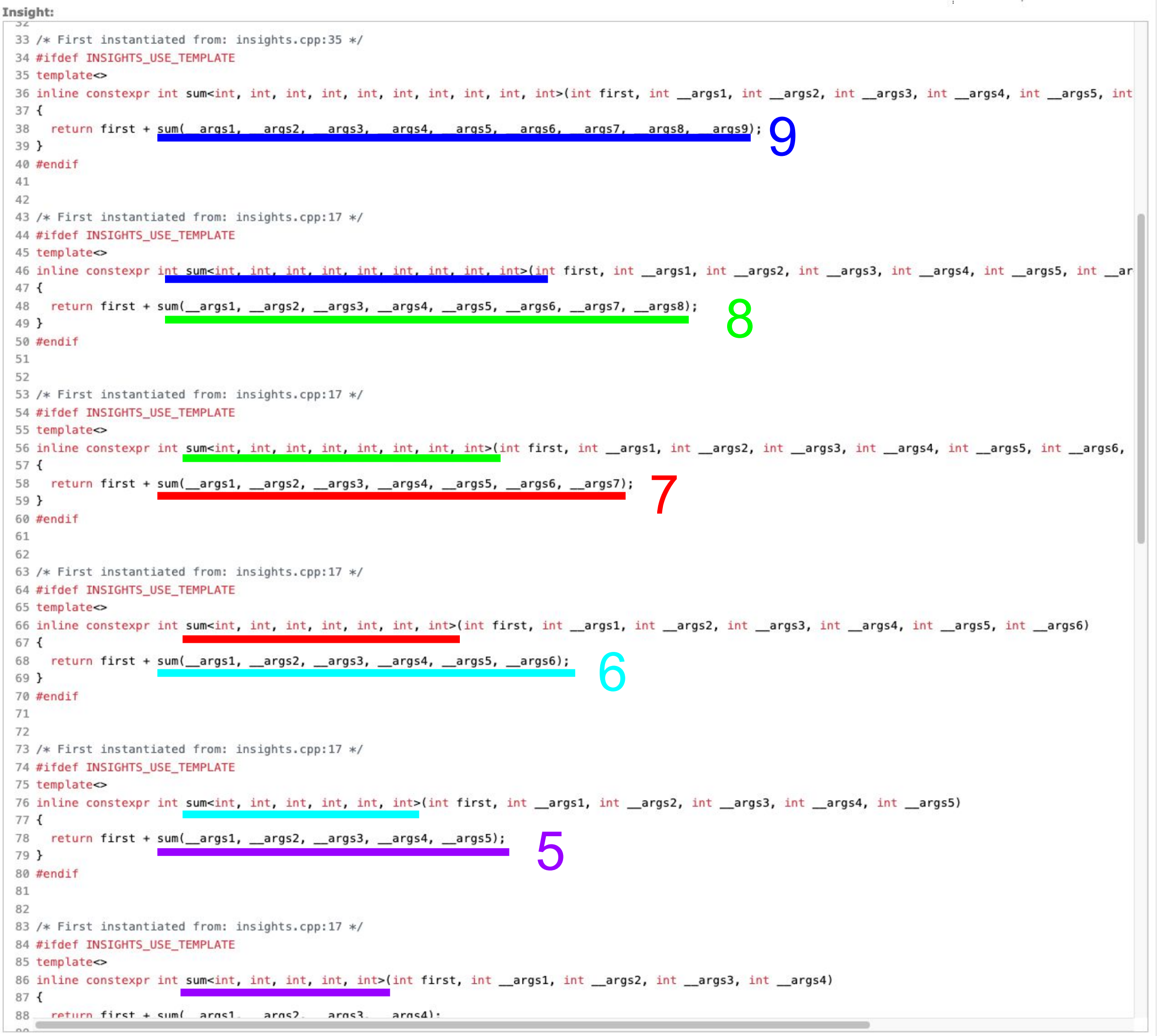


Insight:

```
/* First instantiated from: insights.cpp:35 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7, __args8, __args9);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __ar
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7, __args8);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __args6,
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __args6)
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5)
{
  return first + sum(__args1, __args2, __args3, __args4, __args5);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4)
{
  return first + sum(__args1, __args2, __args3, __args4);
```

Variadic templa

```
template <typename T>
constexpr T sum(T a) {
    return a;
}

template <typename T, typ
constexpr T sum(T first, 
    return first + sum(arg
}
```



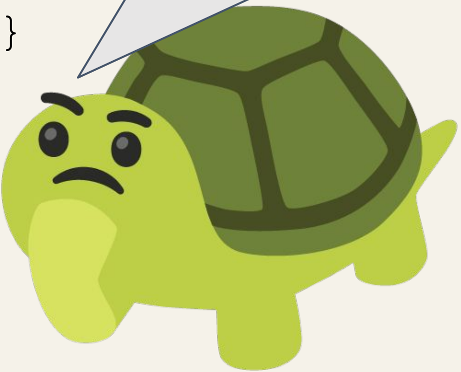
Insight:

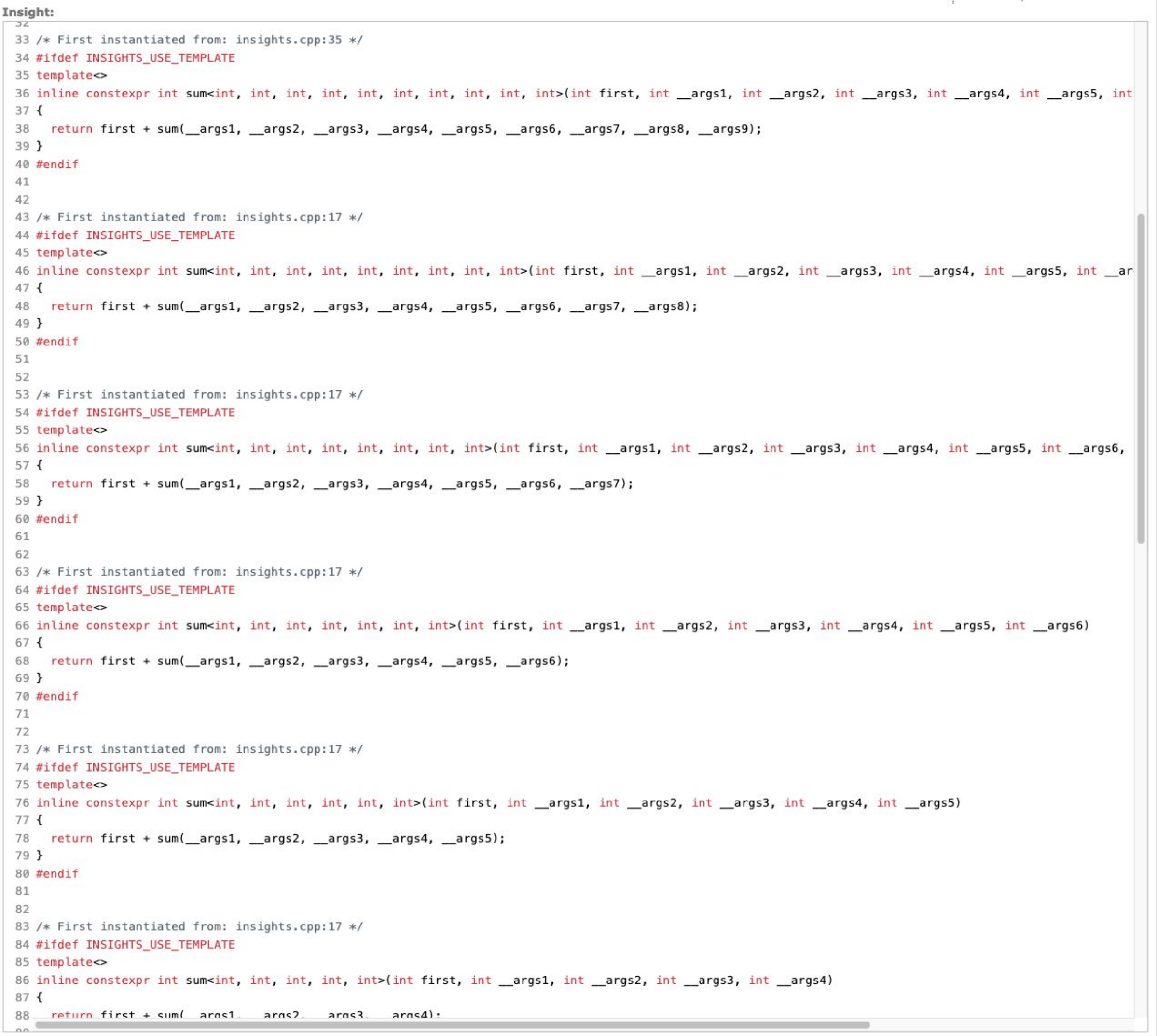
```
/* First instantiated from: insights.cpp:35 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7, __args8, __args9);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __ar
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7, __args8);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __args6,
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __args6)
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5)
{
  return first + sum(__args1, __args2, __args3, __args4, __args5);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4)
{
  return first + sum(__args1, __args2, __args3, __args4);
```

Variadic templa

```
template <typename T>
constexpr T sum(T a) {

}

te                    typ
co                    , 
                first + sum(arg
}
```

The compiler helps you **expend** it into many functions.





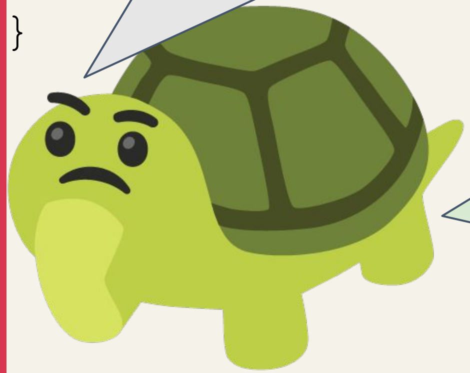
Insight:

```
/* First instantiated from: insights.cpp:35 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7, __args8, __args9);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __ar
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7, __args8);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __args6,
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __args6)
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5)
{
  return first + sum(__args1, __args2, __args3, __args4, __args5);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4)
{
  return first + sum(__args1, __args2, __args3, __args4);
```

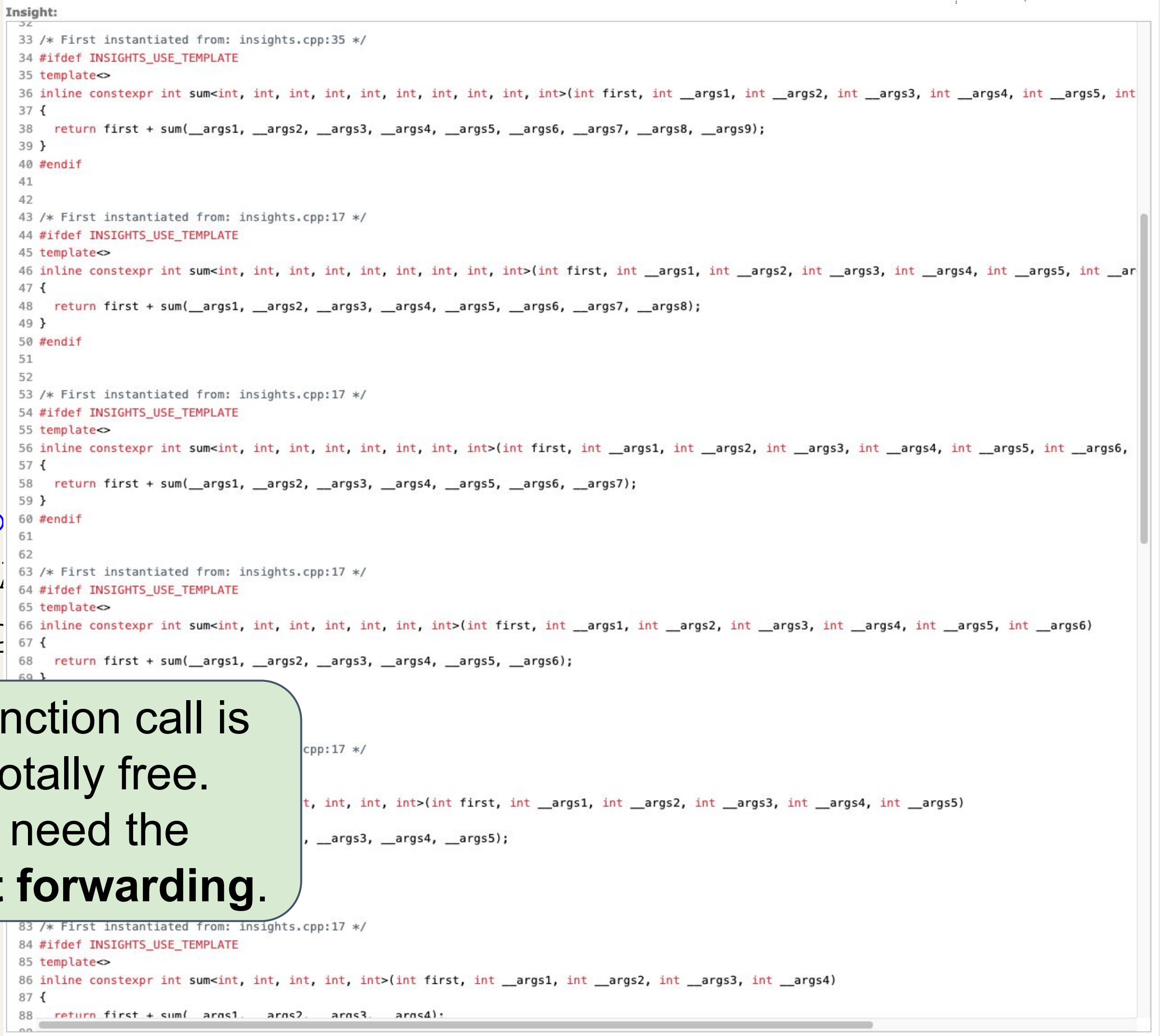
Variadic templa

```
template <typename T>
constexpr T sum(T a) {

}

te                    typ
co                    , 
  re        first + sum(arg
}
```

The compiler helps you **expend** it into many functions.



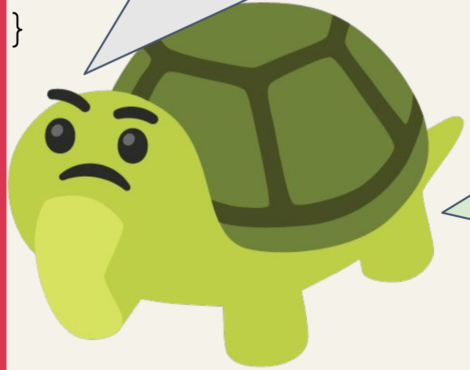
The function call is not totally free. We need the **perfect forwarding**.



```
/* First instantiated from: insights.cpp:35 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7, __args8, __args9);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __ar
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7, __args8);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __args6,
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6, __args7);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5, int __args6)
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6);
}

cpp:17 */

t, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4, int __args5)

, __args3, __args4, __args5);

/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int>(int first, int __args1, int __args2, int __args3, int __args4)
{
  return first + sum(__args1, __args2, __args3, __args4);
```

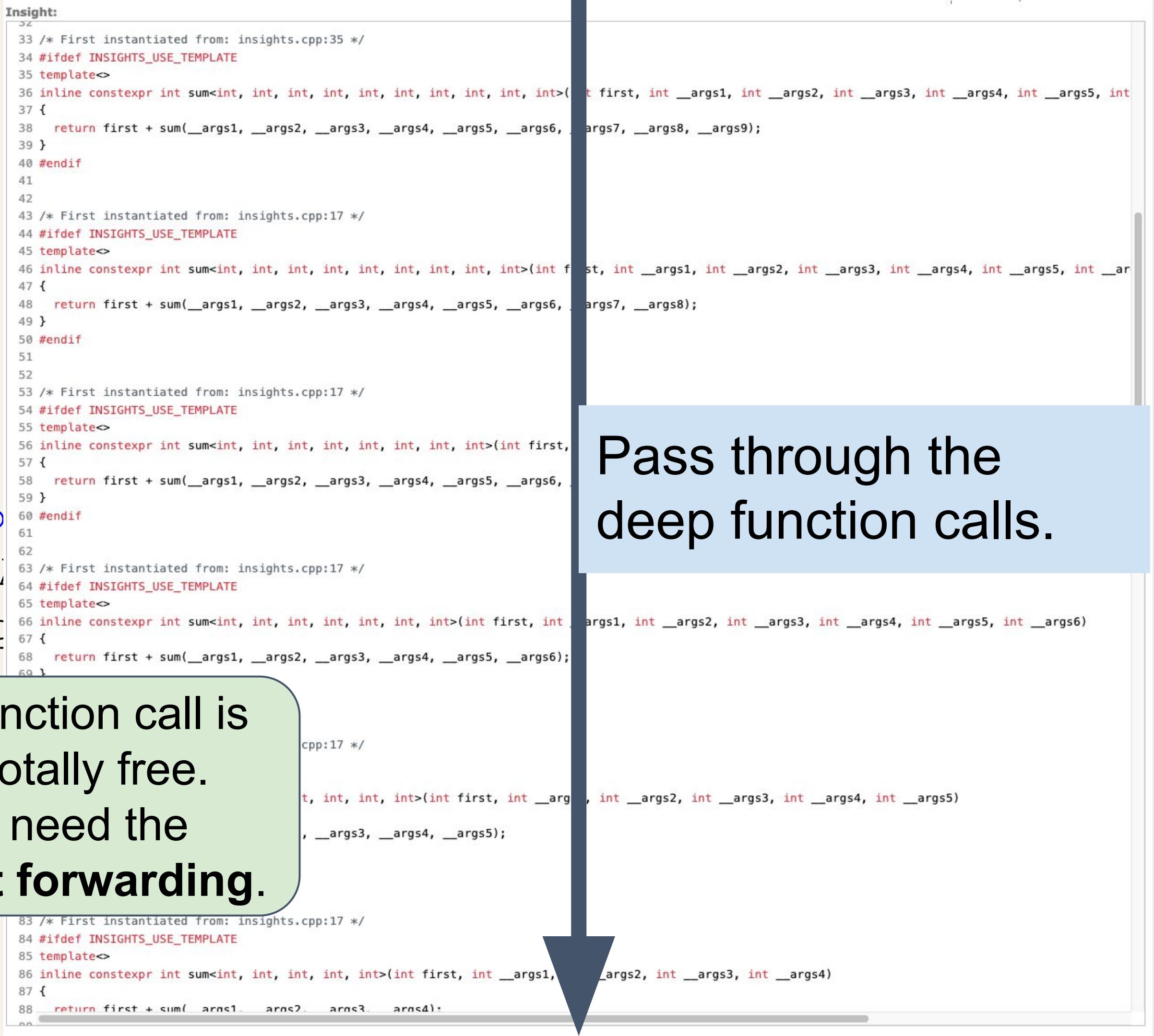
Variadic templa

```
template <typename T>
constexpr T sum(T a) {

}

te                    typ
co                  , 
    re     first + sum(arg
}
```



The compiler helps you **expend** it into many functions.

The function call is not totally free. We need the **perfect forwarding**.

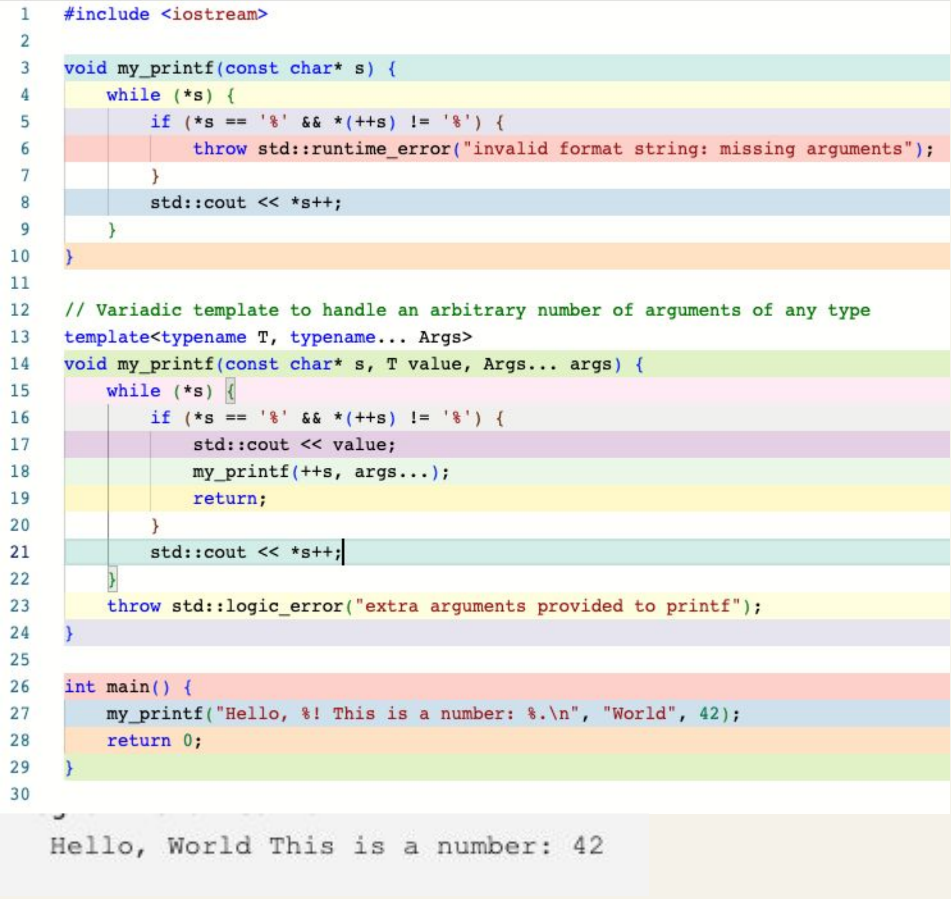


Insight:

```
/* First instantiated from: insights.cpp:35 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int, int>(  t first, int __args1, int __args2, int __args3, int __args4, int __args5, int
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6,  rgs7, __args8, __args9);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int, int>(int f  st, int __args1, int __args2, int __args3, int __args4, int __args5, int __ar
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6,  rgs7, __args8);
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int, int>(int first,
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6,
}
#endif


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int, int, int>(int first, int  rgs1, int __args2, int __args3, int __args4, int __args5, int __args6)
{
  return first + sum(__args1, __args2, __args3, __args4, __args5, __args6);

                              cpp:17 */


                              t, int, int, int>(int first, int __arg  , int __args2, int __args3, int __args4, int __args5)

                              , __args3, __args4, __args5);


/* First instantiated from: insights.cpp:17 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int sum<int, int, int, int, int>(int first, int __args1,    args2, int __args3, int __args4)
{
  return first + sum(__args1, __args2, __args3, __args4);
```

Pass through the deep function calls.

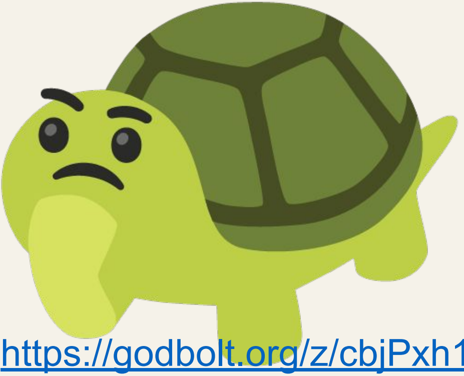
What’s parameter pack?



```cpp
#include <iostream>

void my_printf(const char* s) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') {
            throw std::runtime_error("invalid format string: missing arguments");
        }
        std::cout << *s++;
    }
}

// Variadic template to handle an arbitrary number of arguments of any type
template<typename T, typename... Args>
void my_printf(const char* s, T value, Args... args) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') {
            std::cout << value;
            my_printf(++s, args...);
            return;
        }
        std::cout << *s++;
    }
    throw std::logic_error("extra arguments provided to printf");
}

int main() {
    my_printf("Hello, %! This is a number: %.\n", "World", 42);
    return 0;
}
```

```
Hello, World This is a number: 42
```

printf example



https://godbolt.org/z/cbjPxh1Pa

What’s parameter pack?

```
void my_printf(const char* s) {
   while (*s) {
       if (*s == '%' && *(++s) != '%') return;
       std::cout << *s++;
   }
}
template<typename T, typename... Args>
void my_printf(const char* s, T value, Args... args) {
   while (*s) {
       if (*s == '%' && *(++s) != '%') {
           std::cout << value;
           my_printf(++s, args...);
           return;
       }
       std::cout << *s++;
   }
```

What’s parameter pack?

```
void my_printf(const char* s) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') return;
        std::cout << *s++;
    }
}
```

```
"Hello, %s This is a number: %d\n"
```

```
te
vo
    while (*s) {
        if (*s == '%' && *(++s) != '%') {
            std::cout << value;
            my_printf(++s, args...);
            return;
        }
        std::cout << *s++;
    }
```

What’s parameter pack?

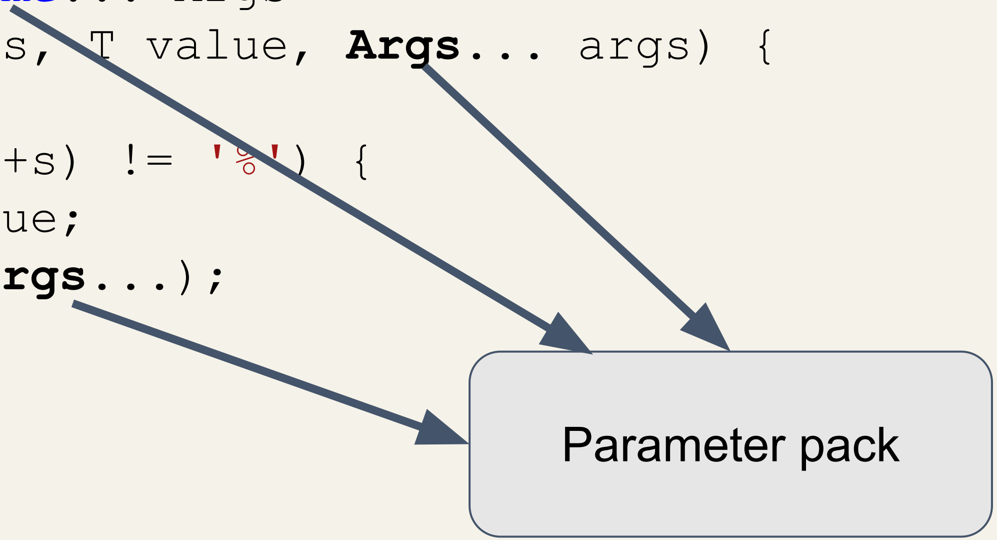
```
void my_printf(const char* s) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') return;
        std::cout << *s++;
    }
}
```

```
template<typename T, typename... Args>
void my_printf(const char* s, T value, Args... args) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') {
            std::cout << value;
            my_printf(++s, args...);
            return;
        }
        std::cout << *s++;
    }
}
```

What’s parameter pack?

```
void my_printf(const char* s) {
   while (*s) {
       if (*s == '%' && *(++s) != '%') return;
       std::cout << *s++;
   }
}
```

```
template<typename T, typename... Args>
void my_printf(const char* s, T value, Args... args) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') {
            std::cout << value;
            my_printf(++s, args...);
            return;
        }
        std::cout << *s++;
    }
}
```



Parameter pack

What’s parameter pack?

```
void my_printf(const char* s) {
   while (*s) {
       if (*s == '%' && *(++s) != '%') return;
       std::cout << *s++;
   }
}
template<typename T, typename... Args>
void my_printf(const char* s, T value, Args... args) {
    while (*s) {
        if (*s == '%' &&
            std::cout <<
            my_printf(++s
            return;
        }
        std::cout << *s++;
    }
}
```

折疊 folding
好多個在這邊**摺**起來

What’s parameter pack?

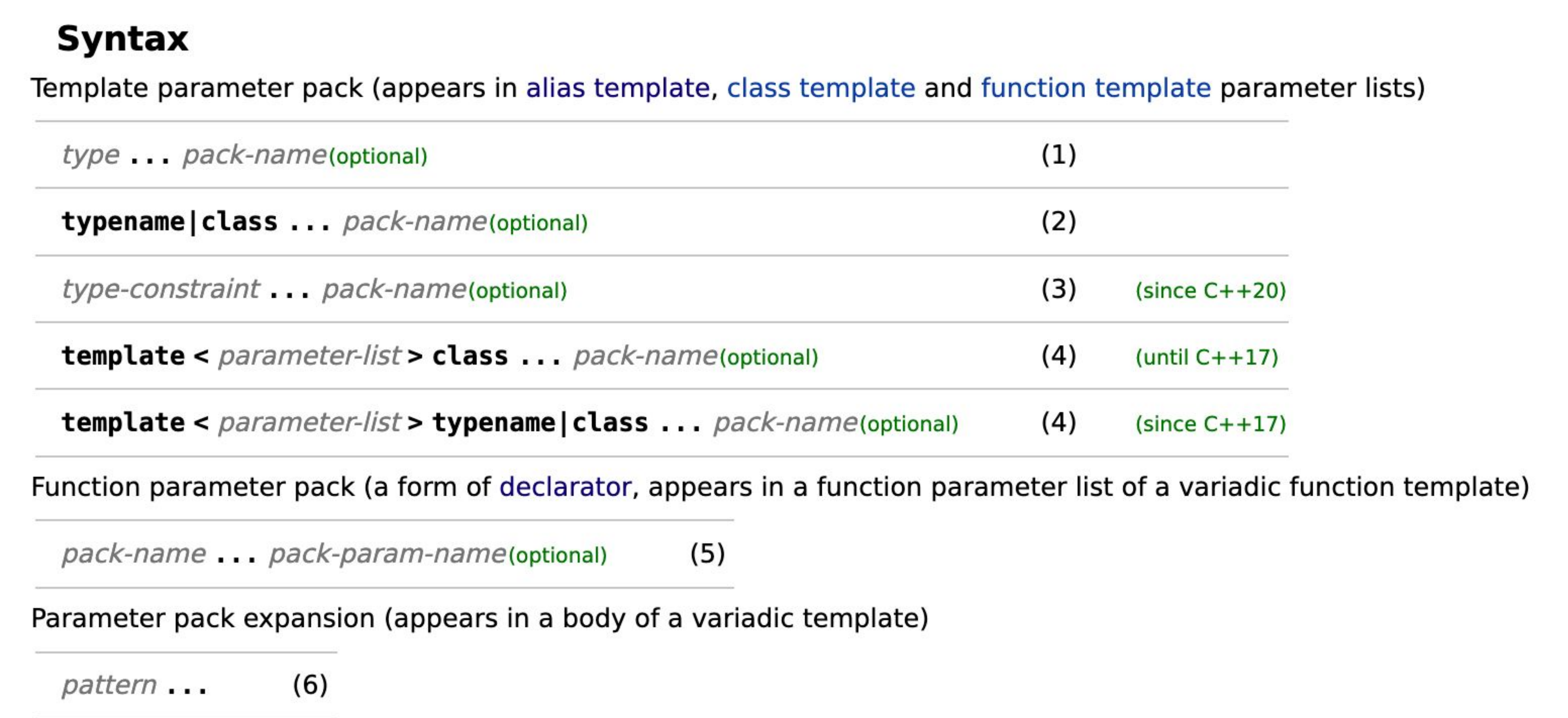
```
void my_printf(const char* s) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') return;
        std::cout << *s++;
    }
}
```

```
template<typename T,
void my_printf(const                                        args) {
    while (*s) {
        if (*s == '%
            std::cout << value;
            my_printf(++s, args...);
            return;
        }
        std::cout << *s++;
    }
}
```

展開 expending
好多個在這邊**展開**

What’s parameter pack?

```
void my_printf(const char* s) {
```



Syntax

Template parameter pack (appears in alias template, class template and function template parameter lists)

type **...** *pack-name*(optional)	(1)	
typename|class ... *pack-name*(optional)	(2)	
type-constraint **...** *pack-name*(optional)	(3)	(since C++20)
template < *parameter-list* **> class ...** *pack-name*(optional)	(4)	(until C++17)
template < *parameter-list* **> typename|class ...** *pack-name*(optional)	(4)	(since C++17)

Function parameter pack (a form of declarator, appears in a function parameter list of a variadic function template)

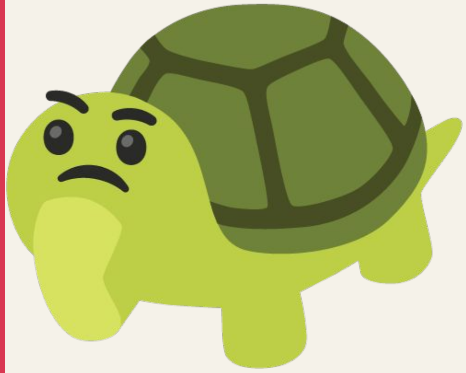
pack-name **...** *pack-param-name*(optional)	(5)

Parameter pack expansion (appears in a body of a variadic template)

pattern **...**	(6)

Expansion loci

Many expandings in a single expression.



Expansion loci

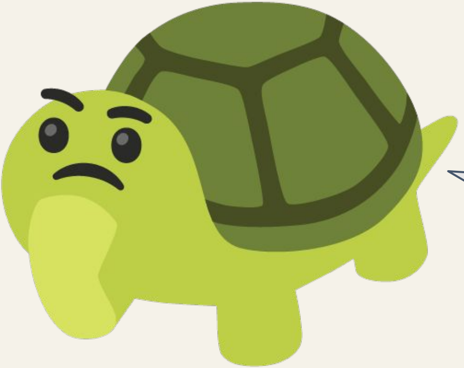
```
f(args...);          // expands to f(E1, E2, E3)
f(&args...);         // expands to f(&E1, &E2, &E3)
f(n, ++args...);     // expands to f(n, ++E1, ++E2, ++E3);
f(++args..., n);     // expands to f(++E1, ++E2, ++E3, n);
f(const_cast<const Args*>(&args)...);
// f(const_cast<const E1*>(&X1), const_cast<const E2*>(&X2), const_cast<const E3*>(&X3))
```

Expansion loci

```
f(h(args...) + args...); // expands to
//f(h(E1, E2, E3) + E1, h(E1, E2, E3) + E2, h(E1, E2, E3) + E3)
```

Expansion loci

```
f(h(args...) + args...); // expands to
//f(h(E1, E2, E3) + E1, h(E1, E2, E3) + E2, h(E1, E2, E3) + E3)
```



Combining for sizeof...(args) times.

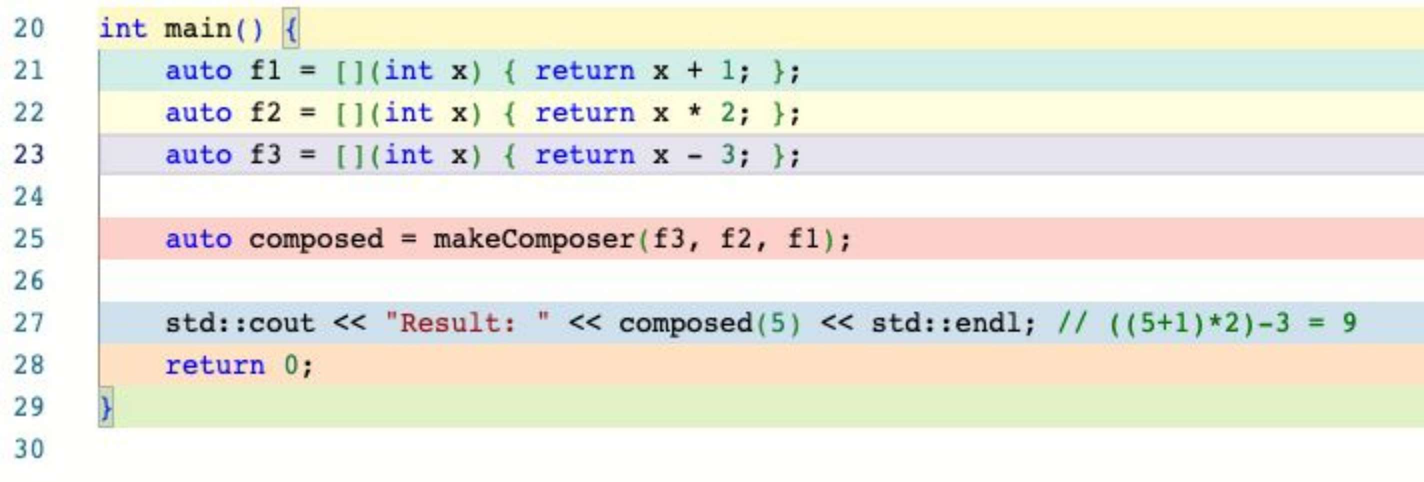
Function composer

```
template<typename Func>
auto makeComposer(Func&& f) {
  return [f](auto&&... args) -> decltype(auto) {
      return f(std::forward<decltype(args)>(args)...);
  };
}

template<typename First, typename... Rest>
auto makeComposer(First&& first, Rest&&... rest) {
  auto composed = makeComposer(std::forward<Rest>(rest)...);
  return [first, composed](auto&&... args) -> decltype(auto) {
      return first(composed(std::forward<decltype(args)>(args)...));
  };
}
```

https://godbolt.org/z/fvcev9oYv

Function composer

```
template
auto mak
   return
       re
   };
}
```



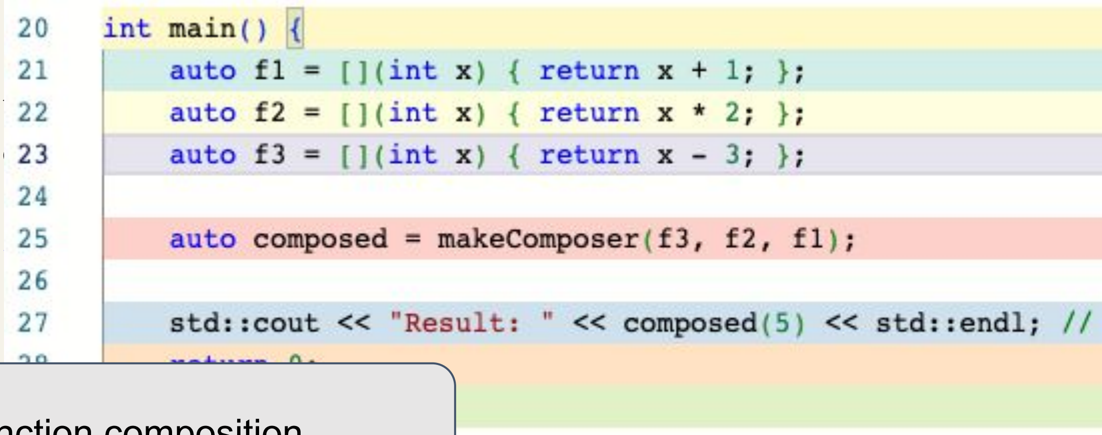
```
int main() {
    auto f1 = [](int x) { return x + 1; };
    auto f2 = [](int x) { return x * 2; };
    auto f3 = [](int x) { return x - 3; };

    auto composed = makeComposer(f3, f2, f1);

    std::cout << "Result: " << composed(5) << std::endl; // ((5+1)*2)-3 = 9
    return 0;
}
```

```
template
auto makeComposer(First&& first, Rest&&... rest) {
   auto composed = makeComposer(std::forward<Rest>(rest)...);
   return [first, composed](auto&&... args) -> decltype(auto) {
       return first(composed(std::forward<decltype(args)>(args)...));
   };
}
```

https://godbolt.org/z/fvcev9oYv

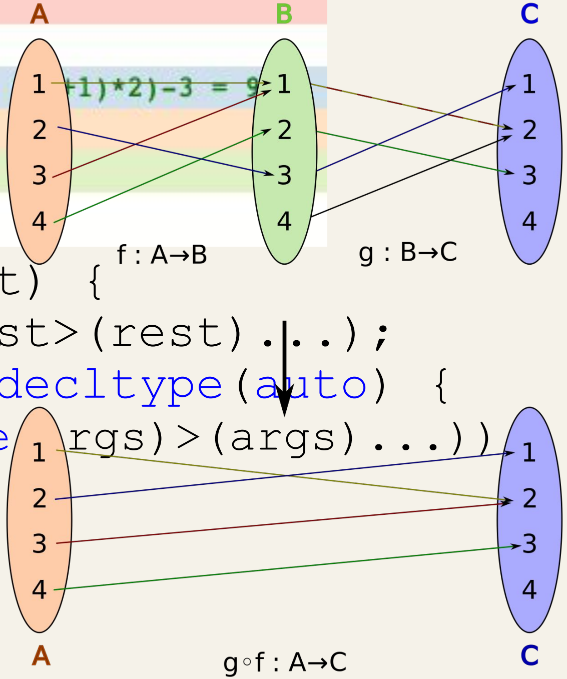
Function composer

```
template
auto mak
  return
      re
  };
}

te
auto makeComposer(First&& first, Rest&&... rest) {
  aut     ed = makeComposer(std::forward<Rest>(rest)...);
  r         t, composed](auto&&... args) -> decltype(auto) {
              st(composed(std::forward<decltype   rgs)>(args)...))

}
```



```
int main() {
    auto f1 = [](int x) { return x + 1; };
    auto f2 = [](int x) { return x * 2; };
    auto f3 = [](int x) { return x - 3; };

    auto composed = makeComposer(f3, f2, f1);

    std::cout << "Result: " << composed(5) << std::endl; //  +1)*2)-3 = 9
    return 0;
```

Function composition.

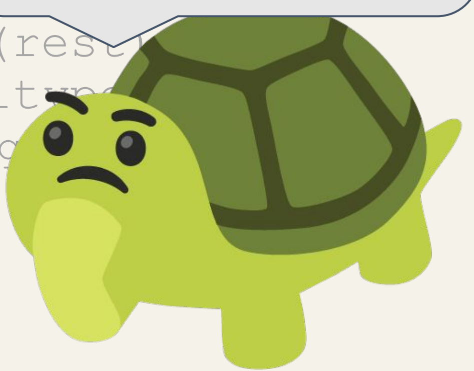


https://godbolt.org/z/fvcev9oYv

Function composer

```
template<typename Func>
auto makeComposer(Func&& f) {
  return [f](auto&&... args) -> decltype(auto) {
      return f(std::forward<decltype(args)>(args)...);
  };
}

template<typename First, typename... Rest>
auto makeComposer(First&& first, Rest&&... rest)
  auto composed = makeComposer(std::forward<Rest>(rest
  return [first, composed](auto&&... args) -> declty
      return first(composed(std::forward<decltype(arg
  };
}
```

數學歸納法需要一個基底。
Template 遞迴也要



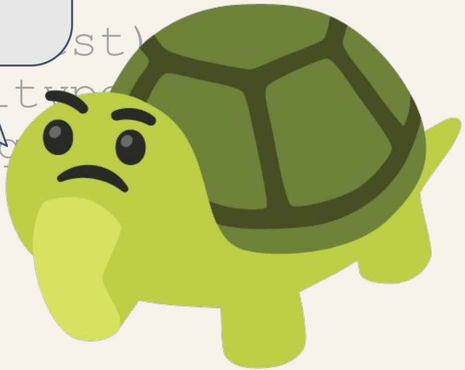
https://godbolt.org/z/fvcev9oYv

Function composer

```
template<typename Func>
auto makeComposer(Func&& f) {
  return [f](auto&&... args) -> decltype(auto) {
      return f(std::forward<decltype(args)>(args)...);
  };
}
```

```
template<typename First, typename
auto makeComposer(First&& first,
  auto composed = makeComposer(st                    st)
  return [first, composed](auto&&... args) -> d      ltyp
      return first(composed(std::forward<decltype(ar
  };
}
```

Forwarding references



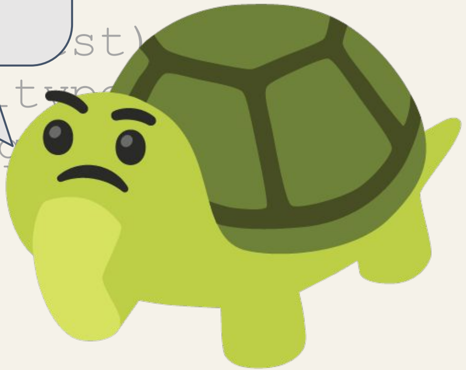
https://godbolt.org/z/fvcev9oYv

Function composer

```
template<typename Func>
auto makeComposer(Func&& f) {
  return [f](auto&&... args) -> decltype(auto) {
      return f(std::forward<decltype(args)>(args)...);
  };
}
```

```
template<typename First, typename.
auto makeComposer(First&& first, Re
  auto composed = makeComposer(std.                       st)
  return [first, composed](auto&&... args) -> d          ltype
      return first(composed(std::forward<decltype(arg
  };
}
```

Variadic template



https://godbolt.org/z/fvcev9oYv

Function composer

```cpp
template<typename Func>
auto makeComposer(Func&& f) {
 return [f](auto&&... args) -> decltype(auto) {
   return f(std::forward<decltype(args)>(args)...);
 };
}

template<typename First, typename...
auto makeComposer(First&& first, Rest
 auto composed = makeComposer(std::forward<Rest      est)
 return [first, composed](auto&&... args) -> declt
   return first(composed(std::forward<decltype(arg
 };
}
```

https://godbolt.org/z/fvcev9oYv

Function composer

```
template<typename Func>
auto makeComposer(Func&& f) {
  return [f](auto&&... args) -> decltype(auto) {
      return f(std::forward<decltype(args)>(args)...);
  };
}

template<typename First, typename... Rest>
auto makeComposer(First&& first, Rest&&... rest) {
  auto composed = makeComposer(std::forward<Rest>(rest)...);
  return [first, composed](auto&&... args) -> decltype(auto) {
      return first(composed(std::forward<decltype(args)>(args)...));
  };
}
```

https://godbolt.org/z/fvcev9oYv

Function composer

```
template<typename Func>
auto makeComposer(Func&& f) {
  return [f](auto&&... args) -> decltype(auto) {
      return f(std::forward<decltype(args)>(args)...);
  };
}

template<typename First, typename... Rest>
auto makeComposer(First&& first, Rest&&... rest) {
  auto composed = makeComposer(std::forward<Rest>(rest)
  return [first, composed](auto&&... args) -> decltype(auto) {
      return first(composed(std::forward<decltype(args)>(args)...));
  };
}
```

https://godbolt.org/z/fvcev9oYv

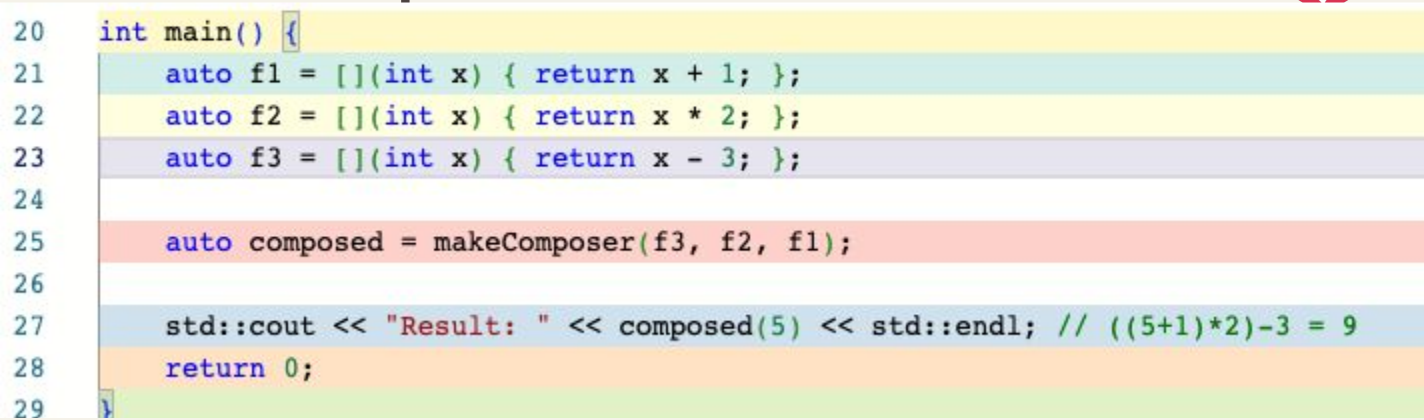
Function composer

```cpp
template<typename Func>
auto makeComposer(Func&& f) {
  return [f](auto&&... args) -> decltype(auto) {
      return f(std::forward<decltype(args)>(args)...);
  };
}

template<typename First, typename... Rest>
auto makeComposer(First&& first, Rest&&... rest) {
  auto composed = makeComposer(std::forward<Rest>(rest)...);
  return [first, composed](auto&&... args) -> decltype(auto) {
      return first(composed(std::forward<decltype(args)>(args)...));
  };
}
```

https://godbolt.org/z/fvcev9oYv

Function composer

```
template
auto mak
  return
      re
  };
}
```



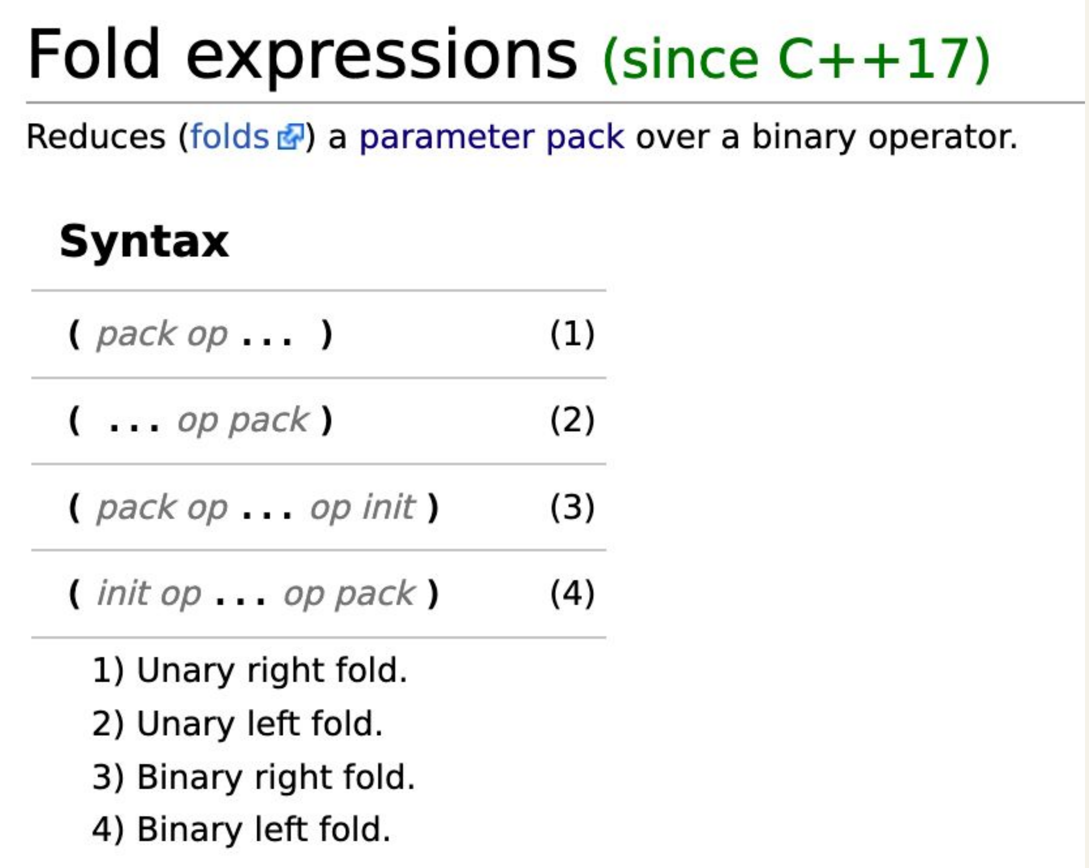
```
int main() {
    auto f1 = [](int x) { return x + 1; };
    auto f2 = [](int x) { return x * 2; };
    auto f3 = [](int x) { return x - 3; };

    auto composed = makeComposer(f3, f2, f1);

    std::cout << "Result: " << composed(5) << std::endl; // ((5+1)*2)-3 = 9
    return 0;
}
```

```
template<typename First, typename... Rest>
auto makeComposer(First&& first, Rest&&... rest) {
  auto composed = makeComposer(std::forward<Rest>(rest)...);
  return [first, composed](auto&&... args) -> decltype(auto) {
      return first(composed(std::forward<decltype(args)>(args)...));
  };
}
```

https://godbolt.org/z/fvcev9oYv

Expressing like Math

Fold expression



Fold expressions (since C++17)

Reduces (folds) a parameter pack over a binary operator.

Syntax

(*pack op* **. . .**)	(1)
(**. . .** *op pack*)	(2)
(*pack op* **. . .** *op init*)	(3)
(*init op* **. . .** *op pack*)	(4)

1) Unary right fold.

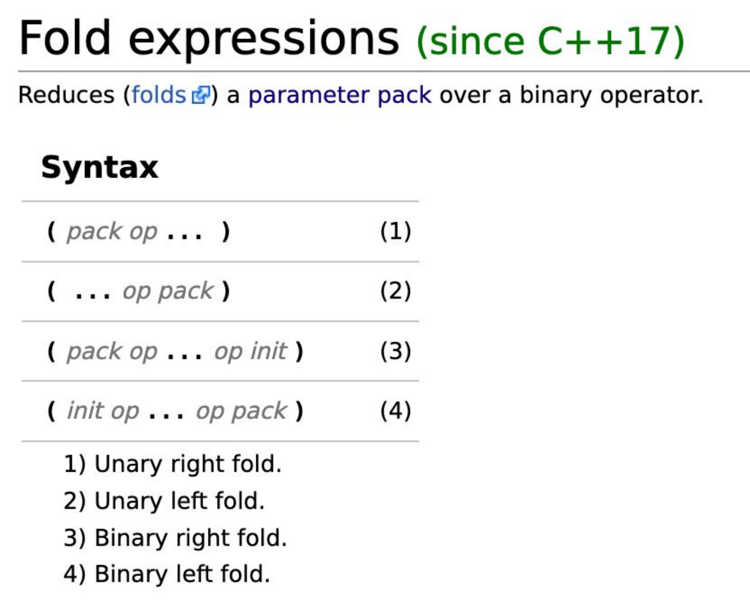
2) Unary left fold.

3) Binary right fold.

4) Binary left fold.

Fold expression

```
template<typename... Args>
void printer(Args&&... args) {
    (std::cout << ... << args) << '\n';
}
```



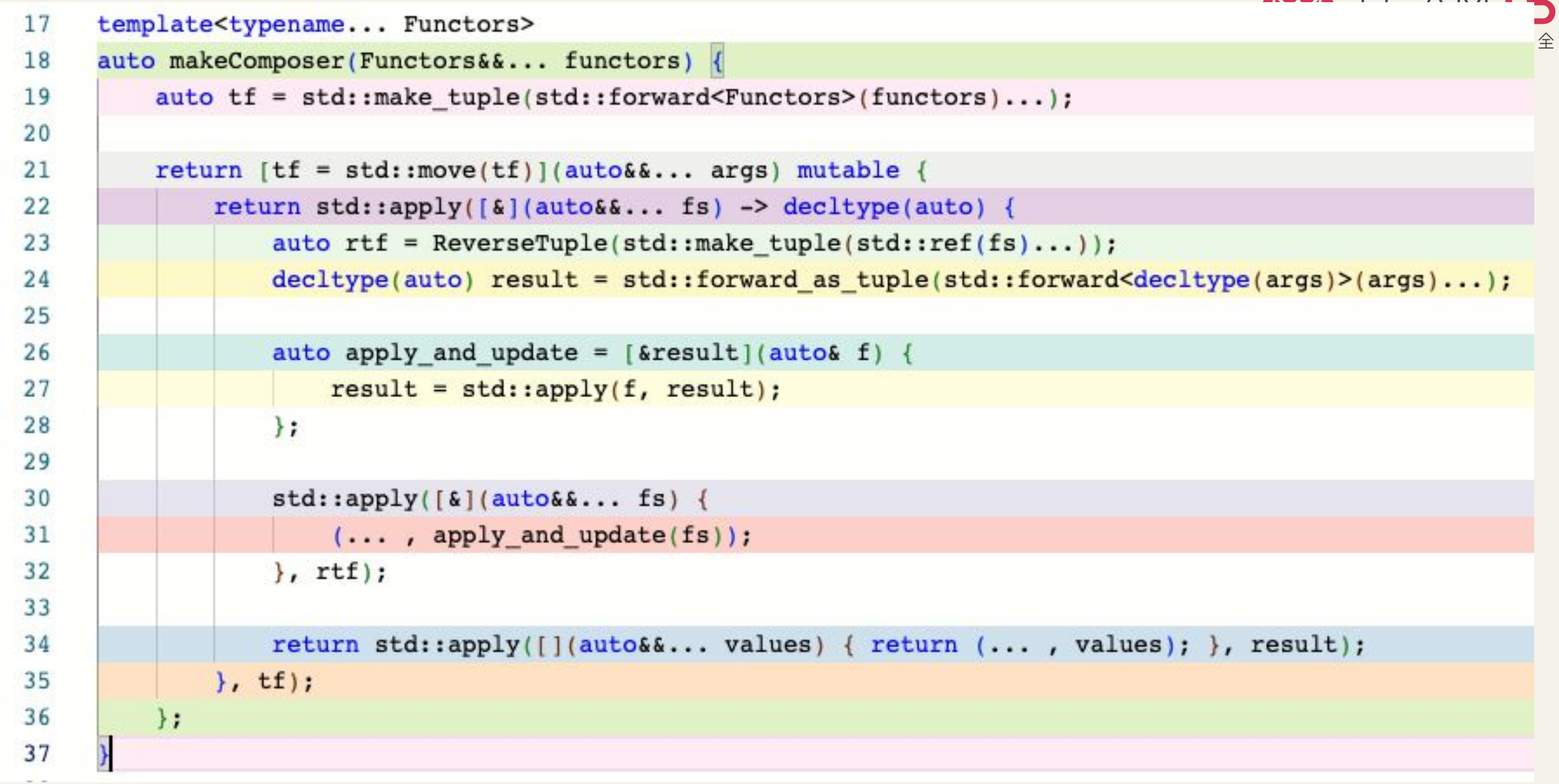
Fold expressions (since C++17)

Reduces (folds) a parameter pack over a binary operator.

Syntax

(*pack op* **...**)	(1)
(**...** *op pack*)	(2)
(*pack op* **...** *op init*)	(3)
(*init op* **...** *op pack*)	(4)

1) Unary right fold.
2) Unary left fold.
3) Binary right fold.
4) Binary left fold.



```cpp
template<typename... Functors>
auto makeComposer(Functors&&... functors) {
    auto tf = std::make_tuple(std::forward<Functors>(functors)...);

    return [tf = std::move(tf)](auto&&... args) mutable {
        return std::apply([&](auto&&... fs) -> decltype(auto) {
            auto rtf = ReverseTuple(std::make_tuple(std::ref(fs)...));
            decltype(auto) result = std::forward_as_tuple(std::forward<decltype(args)>(args)...);

            auto apply_and_update = [&result](auto& f) {
                result = std::apply(f, result);
            };

            std::apply([&](auto&&... fs) {
                (... , apply_and_update(fs));
            }, rtf);

            return std::apply([](auto&&... values) { return (... , values); }, result);
        }, tf);
    };
}
```

https://godbolt.org/z/s4ofKv1z7

Take away

- Variadic template comes from va_list
- Perfect forwarding helps the variadic template
- We need base case to make it recursion
- Fold expression needs the “operator”

Thank you

scc@teamt5.org

TEAMT5

杜浦數位安全

Persistent Cyber Threat Hunters