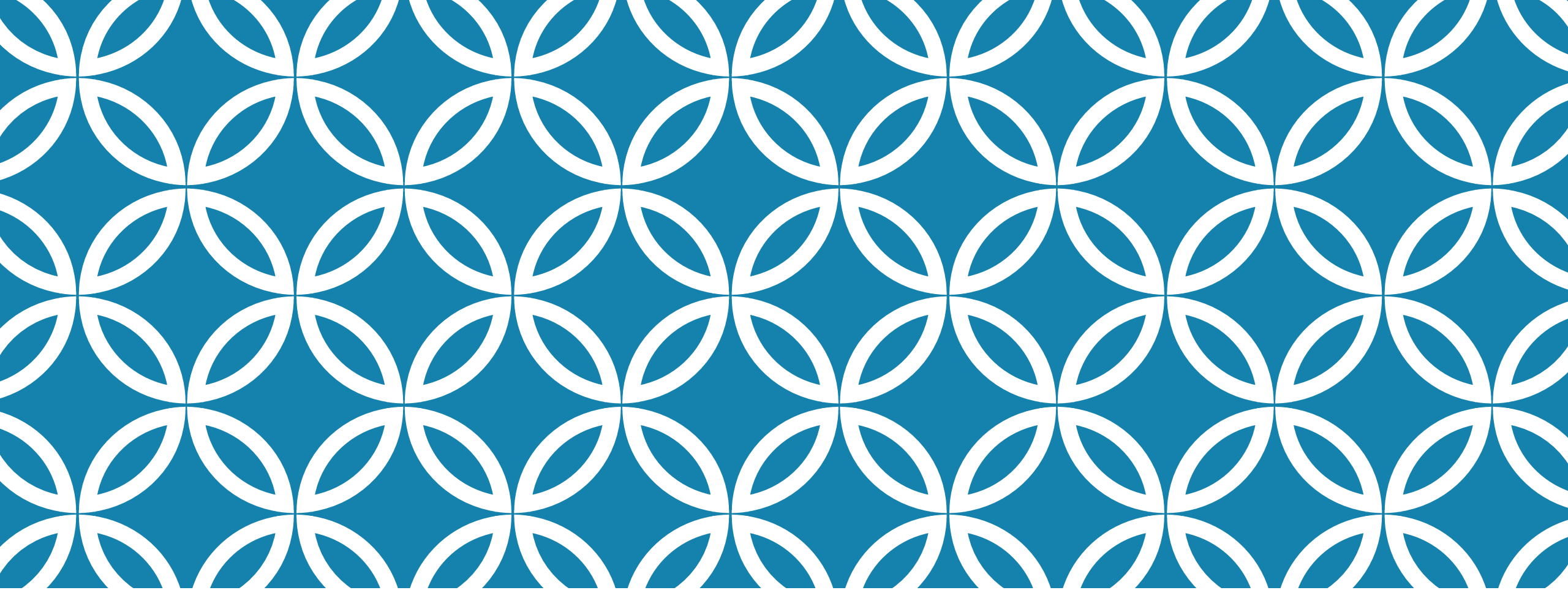


AVOIDING RUNTIME COMPUTE: PRECOMPUTING AT COMPILE TIME

Professor Ken Birman

CS4414/5416 Lecture 6

IDEA MAP FOR TODAY

Many things can be "precomputed". In C++ these include the loop bounds we saw in lecture 6, the types used in "auto" declarations, and code-inlining and refactoring

The C++ compiler is especially famous for this

Why is this technique so valuable?

How do other languages handle the same issue?

Keywords: const, constexpr, consteval

Templates are also a compile-time feature

CONNECTION TO ABSTRACTION

As we saw in prior lectures, abstract thinking isn't limited just to designing ADTs and modules implementing data structures

Dijkstra taught us to think about abstractions that might span entire layers of the operating system (like file systems).

In systems programming we often want to give names to constants used by the system and even computed expressions.

HOW DO PROGRAMS IN C OR C++ BECOME EXECUTABLES?

Languages like Python and Java are highly portable. They compile to byte code… Java does "just in time" compilation to machine code. A JIT or interpreter must be rapid.

In contrast, C++ is compiled using an optimization-driven model: *on this architecture, what is the best way to turn your code into machine instructions?* It is willing to spend a lot of time during compilation to reduce delay at runtime.

CONSIDER THE HUMBLE PROCEDURE CALL…

In fact, let's look at an example:

fibonacci(n) computes the n'th fibonacci integer

```
int fibonacci(int n)
{
    if(n <= 1)
        return n;
    return fibonacci(n-1)+fibonacci(n-2);
}
```

0 1 1 2 3 5 8 13 21 ….

21 = 8 + 13

… FIBONACCI IS THE MOST FAMOUS EXAMPLE OF RECURSION

When first introduced to recursion, many students are confused because

1. The method is invoking itself,
2. The variable n is being used multiple times in different ways,
3. We even call fibonacci twice in the same block!

Over time, you learn to think in terms of "scope" and to view each instance as a separate scope of execution.

WHERE IS FIBONACCI PROCESSED?

In the case of Java or Python, we would know that Fibonacci is performed at runtime, and we learn all about the costs (will review these costs in a moment).

But the bottom line is: The function is translated to a highly efficient data structure (Python) or intermediate code that maps to instructions (Java), then this logic is interpreted or executed.

WHERE IS FIBONACCI PROCESSED?

In C++ there are several possible answers.

The compiler generates any required code but with a more complete analysis: Python and Java types cannot be fully known until runtime, whereas C++ types are known at compile time.

But there are also cases where *less* code or *no* code is needed!

… DOES N NEED A MEMORY LOCATION?

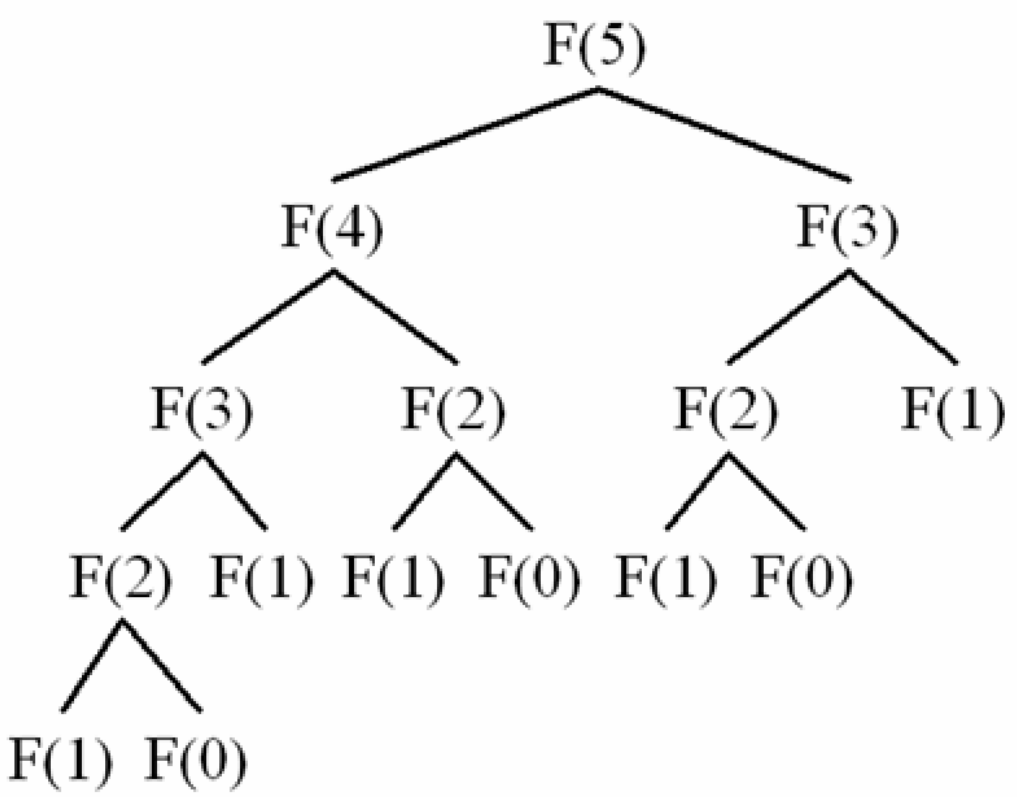
Where does the memory for argument n reside? In Java or Python, n resides on the stack. Each time fibonacci is called:

- Push any registers to the stack, including the return PC
- Push arguments (in our case, the current value of n)
- Jump to fibonacci, which allocates space on the stack for local variables (in our case there aren't any), and executes
- When finished, fibonacci pops the PC and returns to the caller
- The caller code pops data it pushed (and perhaps also the result)

FIBONACCI(5)

```
int fibonacci(int n)
{
    if(n <= 1)
        return n;
    return fibonacci(n-2)+fibonacci(n-1);
}
```

```
fibonacci(5) = fibonacci(3)+Fibonacci(4)
fibonacci(4) = fibonacci(2)+Fibonacci(3)
fibonacci(3) = fibonacci(1)+Fibonacci(2)
fibonacci(2) = fibonacci(0)+Fibonacci(1)
fibonacci(1) = 1
Fibonacci(0) = 1
```



Due to repeatitive pattern, requires 15 calls to Fibonacci!

COMMON OPTIMIZATION

In CS2110 we teach about caching (memoization). But a compiler would not automate this solution: it "changes the code"

```
int fibonacci(n)
{
    if(n <= 1)
        return n;
    if(!known_results.contains(n)) {
        known_results[n] = fibonacci(n-1)+fibonacci(n-2);
    }
    return known_results[n];
}
```

WITHOUT MEMOIZATION, WHERE IS TIME BEING SPENT?

How many instructions really relate to computing fibonacci?

2

We have an if statement: a comparison (call it compare “a and b”) then branch “if a >= b”.

1

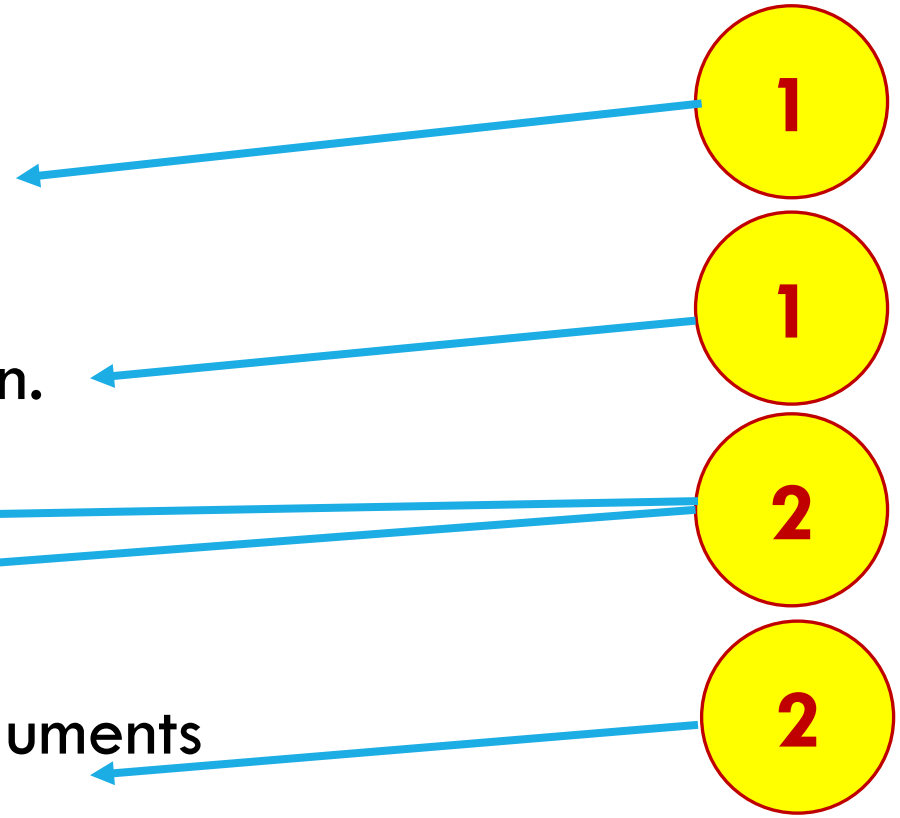
Two recursive calls, one addition, then return.

2 * ? + 1 + 1

THE COST OF THE RECURSIVE CALLS?

They each

- Push registers. Probably 1 is in use.
- Push arguments. In our case, value of n.
- Push the return PC, jump to fibonacci
- After the call, we need to pop the arguments and also pop the saved registers.



… NOW WE CAN FILL IN THE "?" WITH 6

How many instructions really relate to computing fibonacci? **2**

We have an if statement: a comparison (call it compare "a and b") then branch "if a >= b". **1**

Two recursive calls, one addition, then return. **2 * 6 + 2 + 1**

HOW MANY INSTRUCTIONS TO PUSH AND POP ARGUMENTS?

About 15 instructions per call to fibonacci. Of these, 1 is the actual addition operation, and the others are "housekeeping"

For example: fibonacci(5)=0…1…1…2…3…5

Our code needs to do the required 5 additions. However, to compute it we will do 15 recursive calls at a cost of about 15 instructions each: 255 instructions… 51x slower than ideal!

SOME QUESTIONS WE CAN ASK

When C++ creates space for us to hold n on the stack, why is it doing this?

We should have a copy of n if we will make changes, but then would want them discarded, or perhaps if the caller might be running a concurrent thread that could make changes to n "under our feet" (if the caller is spawning concurrent work).

But Fibonacci does not change n!

C++ "CONST" ANNOTATION

Expresses the promise that something will not be changed. (At least, won't be changed by this piece of code).

The compiler can then use that knowledge to produce better code, in situations where an opportunity arises.

Can only be used if you genuinely won't change the value!

FIBONACCI WITH CONST

Our code doesn't change n, so we could try:

```
int fibonacci(const int& n)
{
    if(n <= 1)
        return n;
    return fibonacci(n-1)+fibonacci(n-2);
}
```

… but n-1 and n-2 aren't guaranteed to be in memory, so C++ will need to create a temporary variable for them.

C++ CONST ANNOTATION

The easiest case:

```
const int  MAXD = 1000;       // Length of myvec
char  myvec[MAXD];            //  digits is an array 8-bit ints
```

Here, we are declaring a “compile time constant”. C++ knows that MAXD is constant and can use this in various ways.

WHAT IF I DON'T KNOW AHEAD OF TIME?

Sometimes you can be conservative and declare a constant length that really is the largest value permitted.

Another option is to pass MAXD in as a compiler argument!

```
g++ -std=c++20 –DMAXD=value myprog.cpp –o myprog
```

Many companies do this for things like photo dimensions where they have a general purpose program and want to specialize it for common photo sizes to get a better quality of code from C++

HOW DOES C++ LEVERAGE THIS CONST?

… for example, consider

myvec[MAXD-k-1] = c;

```
movb    %rbx,_myvec(999-%rax)
```

This sets the item "k" from the end to 8. **C++ can compute MAXN-1 as a constant**, and index directly to this item as an offset relative to myvec.

By having c and k in registers, only a single instruction is needed!

WHY IS THIS SO GREAT?

If C++ had not been able to anticipate that these are constants, it would have needed to c*ompute* the offset into digits.

- That would require more instructions.

Here, we are leveraging knowledge of (1) which items are constants, and also (2) that C++ puts "frequently accessed" variables in registers.

MORE EXAMPLES USING "CONST"

We can mark an argument to a method with "const".

This means "this argument will not be modified".

- C++ won't allow that argument to be used in any situation where it might be modified.
- C++ will also leverage this knowledge to generate better code.
- But the argument must correspond to a variable in memory

CONSTEVAL AND CONSTEXPR

The consteval keyword says that "this expression should be entirely constant". *The expression can even include function calls.*

C++ will complain if for some reason it can't compute the result at compile time: a constant expression turns into a "result" during the compilation stage.

If successful, it treats the result as a const.

CONSTEVAL AND CONSTEXPR

In contrast, the constexpr keyword says “try to evaluate this at compile time, but runtime code is ok if the evaluation fails.”

C++ will not complain if your constexpr is not, in fact, a constant expression. It just creates some code and evaluates at runtime.

But, if successful, it treats the result as a const.

CONSTEVAL AND CONSTEXPR

The consteval keyword says that "this expression should be e

C
a
th

```
constexpr float x = 42.0;
constexpr float z = exp(5, 3);
constexpr int i; // Not an error… but not a constant!  I isn't initialized
int j = 0;
constexpr int c = j + 1; //Legal, but will be computed at runtime
consteval int k = j + 1; //Error! j isn't const, so can't be fully evaluated
```

If successful, it treats the result as a const.

FUNCTIONS USED IN CONSTANT EXPRESSIONS

To use a function in as an initializer for a const, or in a constexpr, the function itself must be marked as a constexpr.

The compiler will complain if any aspect of the function cannot be fully computed at compile time.

WE CAN COMBINE THESE ANNOTATIONS

Here we declare that exp is a constant expression using a recursive method to compute x^n

```
constexpr float exp(const float x, const int n)
{
    if(n == 0)
        return 1;
    if(n % 2 == 0)
        return exp(x * x, n / 2);
    return exp(x * x, (n - 1) / 2) * x;
}
```

HOW DOES THIS IMPACT FIBONACCI(N)?

If n is a constant, fibonacci(n) *can actually be computed as a constant expression too.*

The C++ consteval concept focuses on this sort of optimization. If something is marked as a consteval, C++ computes it at compile time, and gives an error if this fails!

Constexpr is very similar but with no error message if it fails. It just generates normal code if needed.

WE CAN COMBINE THESE ANNOTATIONS

C++ can compute fibonacci(5) as a constexpr entirely at compile time. It will just turn this into the constant 5. Same with consteval… but only if you supply a constant argument.

```
constexpr int fibonacci(const int n)
{
    return n <= 1? n: fibonacci(n-1)+fibonacci(n-2);
};
```

INLINE

```
inline int sum(const int &a, const int& b)
{
    return a+b;
};
```

Inline tells C++ to “expand” the code of the method. For example:

```
c = sum(a, b);
```

would expand into

```
c = a + b;
```

INLINING IS AUTOMATIC… YET THE KEYWORD IS STILL COMMONLY USED

In effect, when we write "inline" we often are giving a hint both to the compiler (which probably ignores the hint and makes its own decision!) and also to other readers of the code.

We are saying "*I wrote this code as a method, but in fact I am anticipating that this is really a "code pattern" that will be expanded for me, then optimized in place*".

WHAT IF INLINING ISN'T FEASIBLE?

C++ will warn you that you requested inlining but something about your code makes it necessary to compile the method as a normal function. -Wall compiler argument ensures this.

Coding choices can have huge impact on program performance. Read those warnings!

-Wpedantic is useful too. It enforces C++ rules rigidly, which improves portability in situations where Clang or Gcc are using experimental features.

CONSTEVAL WILL SAVE 255 INSTRUCTIONS!

A big win for Fibonacci, as long as the compiler can actually compute the desired value at compile time.

If it can't, the value isn't a legitimate consteval. For the constexpr case, C++ will try inlining your code (and you can "force" it)

HOW ARE THEY USED TODAY?

Elegant, portable code needs a way to express complicated ideas, and often code that computes something is the only option

But const and constexpr allow us to give such things abstracted names and to use them in a higher-level way

This keeps our code simpler and easier to understand!

TEMPLATES

In C++, templates are a major “user” of const and constexpr

In fact, a C++ template is expanded at compile time, making them a form of constexpr!

The templating language is very elaborate and looks a bit like Haskell (a famous functional programming language)

BEWARE OF "TEMPLATES GONE WILD"!

Sounds like a bad fall break movie…

… but it is a real thing. Some people joke that C++ templates have become a virus that selectively infects PL nerds

They start to try to define everything using templates, which leads to some very complex templating features in C++

VARIADIC TEMPLATES

The idea is a bit “brain bending”!

But this feature is a form of compile-time recursion in the template language system, and it allows you to handle variable argument lists with different types for each item.

For printf: we end up with a series of printf calls, each for a single argument. The “remaining arguments” are dealt with recursively.

SAFE_PRINTF (BASE CASE ON LEFT, RECUSIVE ON RIGHT)

```
// In the .hpp file, this comes first, so that
// C++ will know how to compile the "lone" call to
// safe_printf with no arguments, when it sees it.
void safe_printf(const char *s)
{
    // We processed all the arguments, scan remainder
    while (*s) {
        if (*s == '%') {
            if (*(s + 1) == '%') {
                ++s;
            }
            else {
                throw "invalid format: missing arguments";
            }
        }
        std::cout << *s++;
    }
}
```

Base Case

```
template<typename T, typename... Args>
void safe_printf(const char *s, T& value, Args... args)
{
    while (*s) {                    // Scan up to the next format item
        if (*s == '%') {            // found it
            if (*(s + 1) == '%') {
                ++s;
            }
            else {   // Really should check that *s matches T...
                std::cout << value;
                // call even when *s == 0 to detect extra arguments
                safe_printf(s + 1, args...);
                return;
            }
        }
        std::cout << *s++;          // Output text part of the format
    }
    throw "extra arguments provided to printf";
}
```

Recursive Case

Only these lines "generate code"!

SAFE_PRINTF (BASE CASE ON LEFT, RECUSIVE ON RIGHT)

```
// In the .hpp file, this comes first, so that
// C++ will know how to compile the "lone" call to
// safe_printf with no arguments, when it sees it.
void safe_printf(const char *s)
{
    // We processed all the arguments, scan remainder
    while (*s) {
        if (*s == '%') {
            if (*(s + 1) == '%') {
                ++s;
            }
            else {
                throw "invalid format: missing arguments";
            }
        }
        std::cout << *s++;
    }
}
```

Base Case

```
template<typename T, typename... Args>
void safe_printf(const char *s, T& value, Args... args)
{
    while (*s) {                    // Scan up to the next format item
        if (*s == '%') {            // found it
            if (*(s + 1) == '%') {
                ++s;
            }
            else {   // Really should check that *s matches T...
                std::cout << value;
                // call even when *s == 0 to detect extra arguments
                safe_printf(s + 1, args...);
                return;
            }
        }
        std::cout << *s++;          // Output text part of the format
    }
    throw "extra arguments provided to printf";
}
```

Recursive Case

Only these lines "generate code"!

KEY TO UNDERSTANDING THIS TEMPLATE

It creates a whole series of "safe_printf" calls, each calling the next one, for use with this specific sequence of types

```
template<typename T, typename... Args>
void safe_printf(const char *s, T& value, Args... args)
{
    …
   std::cout << value;              // At this point C++ "knows" value is of type T!
   safe_printf(s + 1, args...);   //  We've removed one argument
   …
}
```

KEY TO UNDERSTANDING THIS TEMPLATE

It creates a whole series of "safe_printf" calls, each calling the next one, for use with this specific sequence of types

```
template<typename T, typename... Args>
void safe_printf(const char *s, T& value, Args... args)
{
    …
   std::cout << value;            // At this point C++ "knows" value is of type T!
   safe_printf(s + 1, args...);   //  We've removed one argument
   …
}
```

Template expansion will replace these with a series of properly typed parameters, each with an automatically generated name

KEY TO UNDERSTANDING THIS TEMPLATE

It creates a whole series of "printf" methods, each calling the next one, for use with this specific sequence of types

```
template<typename T, typename... Args>
void safe_printf(const char *s, T& value, Args... args)
{
    …
   std::cout << value;           // At this point C++ "knows" value is of type T!
   safe_printf(s + 1, args...);  // We've removed one argument
   …
}
```

Template expansion will replace these a list of those automatically generated variable names

HOW DOES THIS EXPAND?

A call to safe_printf("%d,%s,%f", n, s, f):

safe_printf(char* format, int __a0, char* __a1, float __a2)

std::cout to print __a0 (format %d), then calls safe_printf(",%s,%f", __a1, __a2)

safe_printf(char* format, char* __a1 , float __a2)

prints __a1 (format %d), then calls safe_printf(",%f", __a2)

safe_printf(char* format, float __a2)

prints __a2 (format %f), then calls safe_printf("")

safe_printf(char* format)

HOW DOES THIS RECURSION TERMINATE?

The very last call to safe_printf will match a second safe_printf template: one with a format string but no parameters.

- We need to place this base case first in the .hpp file. C++ looks for a match in sequential order and will stop when it sees one
- This second won't be recursive… so the template expansion ends.
- Variadic lists can match an empty sequence, so if we omit it, we really would get an infinite recursion!

EVEN STD::COUT IS A TEMPLATE!

It expands to something like this:

```
outbuf[optr++] = c;
if (c == '\n') {
        write(stdout, outbuf, optr);
        optr = 0;
}
```

… and this "if" statement can be constexpr evaluated too

PRINTF("%D,%F,%D,%S\N", 2, 3.0, 4, "5.7");

... will be transformed to

```
outbuf[optr++] = '2';
outbuf[optr++] = ',';
outbuf[optr++] = '3';
        ...
outbuf[optr++] = '\n';
write(stdout, outbuf, optr);
optr = 0;
```

VISUALIZE HOW CONSTEVAL HANDLES THIS

First, the recursive variadic logic is expanded.

Next, all the consteval " work" is performed

Last, the optimizing compiler does everything it can to simplify and cleanup the code…

PRINTF("%D,%F,%D,%S\N", 2, 3.0, 4, "5.7");

… printf ends up looking like this

```
outbuf[0] = '2';
outbuf[1] = ',';
outbuf[2] = '3';
    …
outbuf[16] = '\n';
write(stdout, outbuf, 15);
```

C++ can statically combine these…

memcpy(outbuf, "2, 3.0, 4, 5.7\n", 15);

… and then might even eliminate outbuf (but only if the outbuf variable isn't reused and hence the value left in it is unimportant)

CONCEPT: STATIC ANALYSIS

Modern computing environments often include tools that do some form of analysis of programs or other objects before the execution actually occurs.

For the C++ compiler, constexpr and inline illustrate forms of static analysis that benefit the compilation stage.

HOW STATIC ANALYSIS IS DONE

Focusing on the C++ compiler, it first scans your program and forms a parsed code representation based on applying the syntax rules.

Next, it can study this graph structure to learn things.

What sorts of things can static analysis discover?

MORE STATIC ANALYSIS OPPORTUNITIES

We saw constants, arguments by reference and inlining

Static analysis might also discover loop bounds, “dead” code (an if statement that is never true, or always true), variables that do or do not need space allocated, etc.

Static analysis is also at the core of type checking.

CONSIDER THE "AUTO" DECLARATION

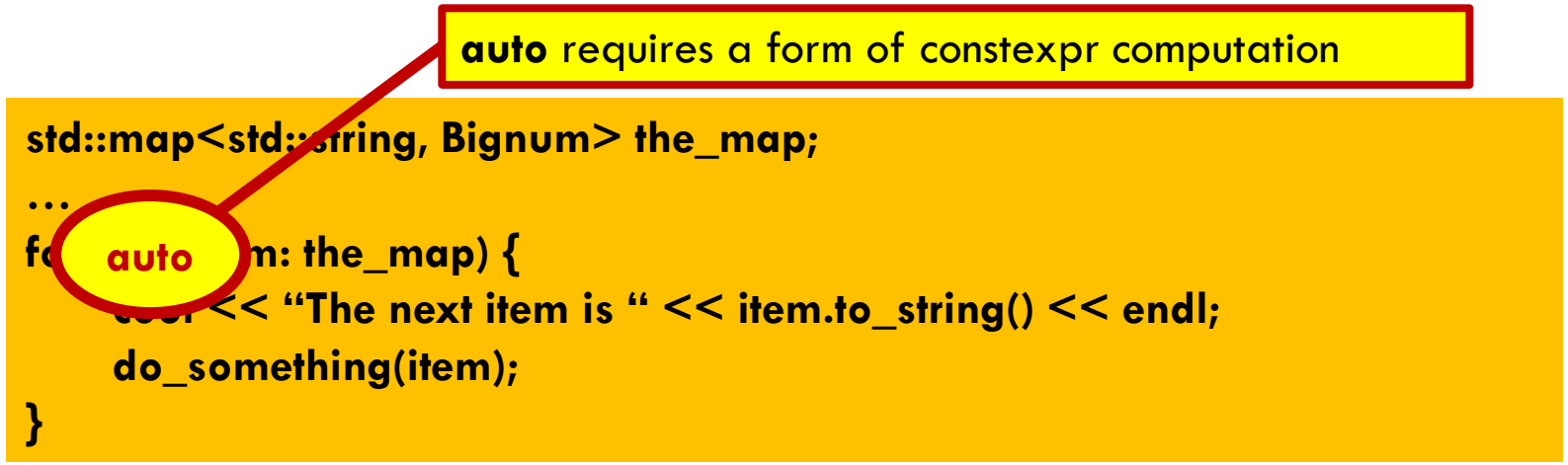
In C++ we often ask the compiler to figure out types:

```
std::map<std::string, Bignum> the_map;
…
for(auto item: the_map) {
        cout << "The next item is " << item.to_string() << endl;
        do_something(item);
}
```

Here we created a map from string "names" to Bignum objects, then iterate through the map (item will be a sequence of std::pair objects)

CONSIDER THE "AUTO" DECLARATION

In C++ we often ask the compiler to figure out types:



Here we created a map from string "names" to Bignum objects, then iterate through the map (item will be a sequence of std::pair objects)

EXAMPLE OF AN AUTO-DISCOVERED TYPE

This is from a C++ “bignum” class:

```
std::pair<typename std::_Rb_tree<_Key, std::pair<const _Key, _Tp>, std::_Select1st<std::pair<const _Key, _Tp> >,
_Compare, typename __gnu_cxx::__alloc_traits<_Allocator>::rebind<std::pair<const _Key, _Tp> >::other>::iterator,
bool> std::map<_Key, _Tp, _Compare, _Alloc>::insert(const value_type&) [with _Key =
std::__cxx11::basic_string<char>; _Tp = Bignum; _Compare = std::less<std::__cxx11::basic_string<char> >; _Alloc =
std::allocator<std::pair<const std::__cxx11::basic_string<char>, Bignum> >; typename std::_Rb_tree<_Key,
std::pair<const _Key, _Tp>, std::_Select1st<std::pair<const _Key, _Tp> >, _Compare, typename
__gnu_cxx::__alloc_traits<_Allocator>::rebind<std::pair<const _Key, _Tp> >::other>::iterator =
std::_Rb_tree_iterator<std::pair<const std::__cxx11::basic_string<char>, Bignum> >; std::map<_Key, _Tp, _Compare,
_Alloc>::value_type = std::pair<const std::__cxx11::basic_string<char>, Bignum>]
```

WHAT IN THE WORLD WAS THAT???

The first thing to know is that C++ often generates its own variables. To avoid name conflicts, it puts underscore characters (_) at the front.

The second thing to know is that a std::map has a “comparison” function and an iterator, which (in my case) were defaults.

And so… this was the complete type for std::map<std::string,Bignum>.

IN FACT, C++ WOULDN'T BE USEFUL WITHOUT TYPE INFERENCE!

Const and constexpr are "natural fits" for C++ because the compiler is already doing so much automatic inference.

These annotations simply advise the compiler to do what it wanted to do in any case!

… just a glimpse of the true complexity of modern languages

BUT BEWARE: NOT EVERY STATIC ANALYSIS PROBLEM CAN BE SOLVED!

We already saw this with constexpr and inlining: recursion can exceed the limitations of the compiler.

In fact, static analysis can even run into "unsolvable" problems!

- Type inference (auto) is potentially undecidable. Even the decidable versions have high complexity. Auto normally is successful.
- But experts can construct cases in which C++ may not be sure what the type of a variable is… and will give an error

IMPLICIT CONTROL

We've seen that a language like C++ lets you ensure that your logic will be precomputed by the compiler

These features are explicit, yet the involve *implicit* control.

- The choice to code in this way was governed by the expectation that the resulting logic would match capabilities of the **compiler.**
- Coded in some other way, the same logic would be executed but evaluation would occur at **runtime:** the program would be slower

SUMMARY FROM TODAY

C++ has advanced features that permit compile-time code analysis, compile-time type inference, and compile-time expression evaluation. This even includes recursive functions!

When we use const / consteval / constexpr, we "control" the compiler, which lets us ensure that the optimized code will use specialized instructions or achieve other kinds of efficiencies.

We code in an elegant, high-level way yet can control the compilation process down to ensuring that C++ will make the choices we want.

MORE SUMMARY

Today's lecture had one big idea: *shifting computation from runtime to compile time is a very good tactic that can benefit us in many situations.*

Then it had a dozen illustrations of that one big idea.

In your future career you may encounter other opportunities to apply this same idea in settings we did not include today.

SELF-TEST

Which of the following can't be defined as a constexpr?

- A test to decide if a float has 8 bits or more accuracy
- A test to decide if a value computed by a process is less then or equal to zero
- The data width a particular kind of SIMD instruction assumes
- A test to see if an array size is a multiple of the SIMD data width.

SELF-TEST

Consider some computation that can be done ahead of time, like a constexpr, but can also be performed at runtime, like when an LLM receives a query.

Is it always preferable to precompute answers the way that constexpr does?

A DEBATE QUESTION

We saw that printf can be rebuilt as a template that maps to std::cout

Why does printf as a function still exist? In fact, it actually is more popular than the templated approach!

SELF-TEST

Suppose we can spend time C compiling a complex program, and doing this reduces runtime by an amount R.

Under what conditions is it worthwhile to spend C units of time for this purpose. It is tempting to say "if C < R" but is this a correct answer? Could there be reasons to spend time C to save runtime R even if C >> R?