CONSTANT EXPRESSIONS IN C++

Professor Ken Birman
CS4414 Lecture 9

IDEA MAP FOR TODAY

In Lecture 8 we learned that C++ can automatically compile to vector-parallel instructions.

… But we also saw longs lists of "suggested" coding styles intended to make it feasible for C++ to do this!

Even without those long lists of advice, this same issue arises when C++ compiles normal code for normal machine instructions! Some styles promote faster code

Today we will look at another example, unrelated to parallelism: the C++ concepts of "const" and "by reference". Const is notationally hard to get used to but valuable. "By reference" is risky to use carelessly, but important to understand!

CONNECTION TO CONCEPTUAL ABSTRACTION

Lectures 7 and 8 looked at cases in which the C++ compiler can carry out some sort of conceptual transformation or optimization if we understand the design pattern.

We saw this with control flow, and with SIMD parallelization.

Today we continue this theme by looking at compile-time expression evaluation: another powerful conceptual abstraction!

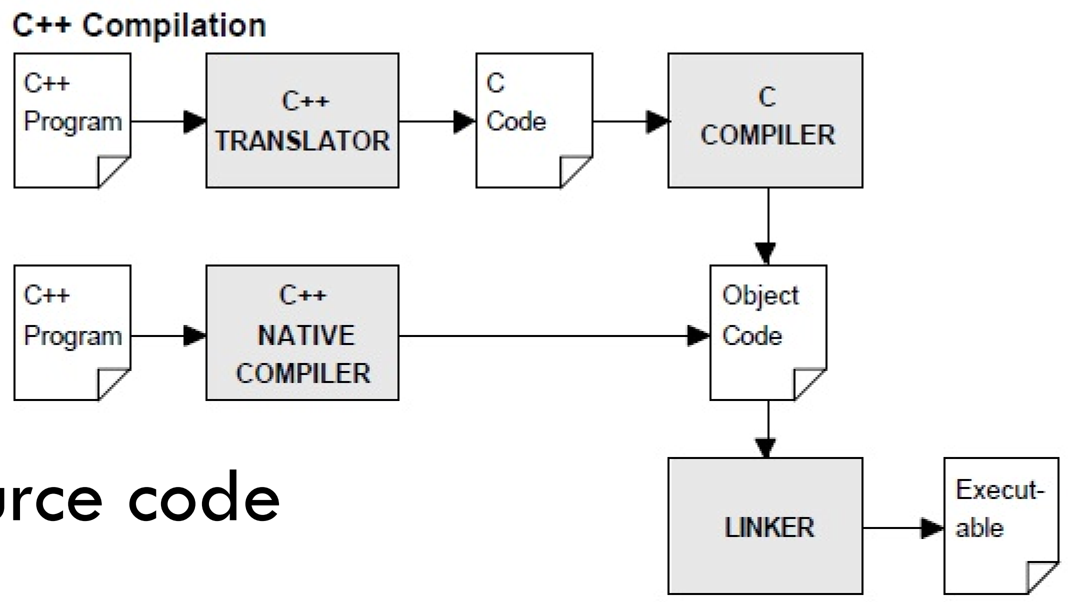
HOW DO PROGRAMS IN C OR C++ BECOME EXECUTABLES?

Languages like Python and Java are highly portable. They compile to byte code… Java does "just in time" compilation to machine code.

This is ***not*** the case for C and C++. Each distinct computer may have a different CPU and its own memory layout rules.

Thus, "find" or "cat" or "tr" or "sort" or "uniq" needs to be turned into machine-language specific for the particular machine

COMPILATION



name.c/.cpp, name.h/.hpp: source code

name.s: assembler language

name.o: “object” code: machine code plus symbol table

name.dll: “dynamically linked library”

a.out: The default name for a compiled executable

core: If enabled, a file created in the current directory (or in /var/core) if your program crashes. Use gdb to find out where and why it happened. Compiler option –g is useful in this context.

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);
}
```

$1\ 2\ 3\ 5\ \underbrace{8\ 13}_{21\ =\ 8\ +\ 13}\ 21\ \ldots.$

WHERE IS FIBONACCI PROCESSED?

As we will see, in C++ there are several possible answers.

The most obvious case is when actual code will be created. Here the compiler itself generates that code.

Later we will see other cases where *no* code is generated!

… 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.

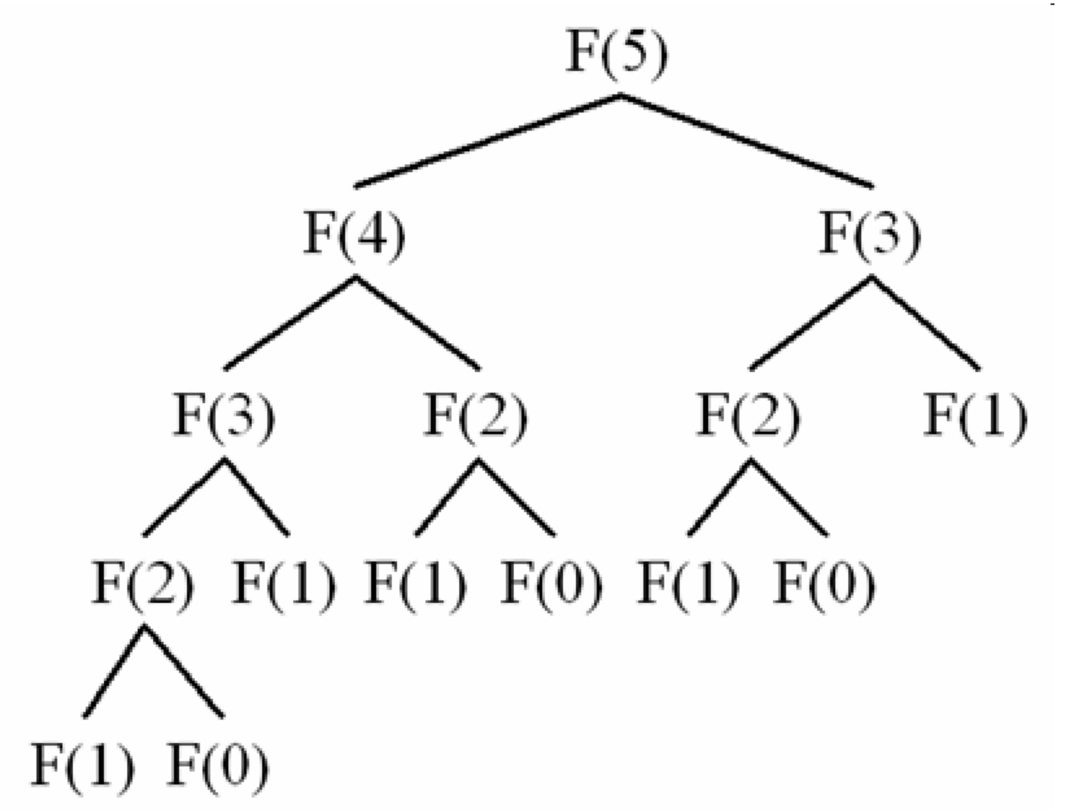
… BUT N DOES NEED A MEMORY LOCATION?

Where does the memory for n reside? … on the stack. Each time fibonacci is called, C++:

- Pushes any registers to the stack, including the return PC
- Pushes arguments (in our case, the current value of n)
- Jumps 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 pops the things it pushed

FIBONACCI(5)

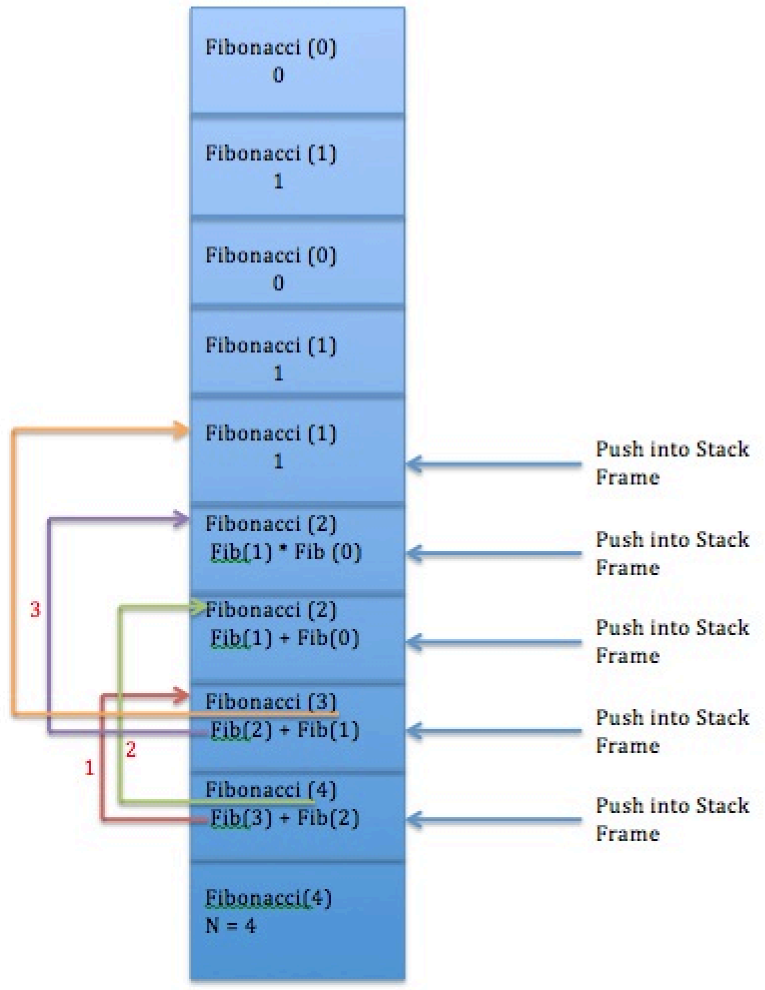
```
int fibonacci(n)
{
    if(n <= 1)
        return n;
    return fibonacci(n-1)+fibonacci(n-2);
}
```



… 15 calls to fibonacci occur, in total

FIBONACCI(5)

```
int fibonacci(3)
{
    if(n <= 1)
        return n;
    return fibonacci(2)+fibonacci(1);
}
```



… 15 calls to fibonacci occur, in total

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. 1
- Push arguments. In our case, n. 1
- Push the return PC, jump to fibonacci 2
- After the call, we need to pop the arguments and also pop the saved registers. 2

… 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 + 1 + 1

HOW MANY INSTRUCTIONS TO PUSH AND POP ARGUMENTS?

About 17 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 17 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

In C++ we have a way to express that something will not be changed.

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

C++ CONST ANNOTATION

The easiest case:

```
const int  MAXD = 1000;        // Limit on number of Bignum digits
char  digits[MAXD];            //  digits is an array of 1000 8-bit ints
```

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

AN EXAMPLE

… for example, consider

```
digits[MAXD-k-1] = c;
```

```
movq    %rbx,_digits(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 *compute* 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.

MORE EXAMPLES USING “CONST”

We can mark an argument to a method with “const”.

This means “

- C++ won’ … it might be n
- C++ will

```
// constant_values1.cpp
int main(const int argc, const char**argv) {
   const int i = 5;
   i = 10;   // C3892
   i++;      // C2105
}
```

MORE EXAMPLES USING "CONST"

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

This means '

- C++ won' ... it might be r
- C++ will

```
// constant_values3.cpp
int main(const int argc, const char**argv) ) {
   char *mybuf = 0, *yourbuf;
   char *const aptr = mybuf;  // Initializes aptr…
   *aptr = 'a';              // OK
   aptr = yourbuf;     // C3892
}
```

MO

We

This

- C
  m
- C

```
// constant_member_function.cpp
class Date
{
public:
   Date( int mn, int dy, int yr );
   Date(const& Date);              // A "copy constructor"
   int getMonth() const;           // A read-only function
   void setMonth( int mn );        // A write function; can't be const
private:
   int month;
};
```

ASIDE

The “const” suffix for a read-only method like getMonth *can only appear inside a method declared as a member of a class.* It means “read only property” of the object the class defines.

If you used this same notation on a global method, it will be rejected with an error message.

ANOTHER ASIDE

A constant lives in the compiler… not in program memory, unless the compiler "needs" to save a copy for some reason.

As a result, you cannot access a constant by reference: when you take the address of an object, or pass it using the "&" notation for a parameter to a method (like sum(int& x, int& y)), you are treating the constant as if it has a location in memory.

… BUT CONST CAN ALSO MEAN "I DON'T CHANGE THIS ARGUMENT"

In this sum function, we are saying "sum will treat **a** and **b** as constants (it won't change them). It accesses them by reference, so you cannot pass a constant to it

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

… BUT CONST CAN ALSO MEAN "I DON'T CHANGE THIS ARGUMENT"

In this sum function, we are saying "sum will treat **a** and **b** as constants (it won't change them). It accesses them by reference, so you cannot pass a constant to it

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

The const at the end says that this method will not change member variables in the class that defined it.

CONSTEXPR

This 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.

CONSTEXPR

This annotation says that "this expression should be entirely

c

```
constexpr float x = 42.0;
constexpr float y{108};
constexpr float z = exp(5, 3);
constexpr int i; // Error! Not initialized
int j = 0;
constexpr int k = j + 1; //Error! j not a constant expression
```

C

a

th

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;
}
```

WHAT ABOUT FIBONACCI(N)?

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

The C++ constexpr concept focuses on this sort of optimization. If something is marked as a constexpr, C++ computes it at compile time.

In principle, it could compute fibonacci(7)…. In practice, however, it might not realize it can pull this off and could give an error.

BY-REFERENCE ARGUMENTS

A method can also ask for a "reference" to its argument, instead of the actual value being pushed on the stack.

This feature only works if C++ can be sure that the caller has an actual object (or reference to one, or a constant) to pass in.

But assuming you do, the method ends up with a second name for the argument passed in: a form of "alias"

BY-REFERENCE ARGUMENTS

Notation:

```
fibonacci(int &n)
{
        ….
}
```

n doesn't have a memory address of its own.

In fact it is a second name (an alias) for the argument passed to fibonacci

N IS “USED” JUST AS IT WAS EARLIER

We can still write things like

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

But now the compiled code is accessing the memory location the caller was using for n. Our method no longer has any local storage for n.

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.

… but it can only be used with a constant argument.

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

INLINE ANNOTATION

This tells C++ that you want it to expand any calls to the method, producing a single "straight line" block of code. In C++ 17 is it considered redundant because the compiler does it automatically.

Thus:

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

would expand into

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

WHAT IF WE <u>INLINE</u> FIBONACCI?

Suppose we had done it this way:

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

Now the expression "expands" if C++ is able to do so.

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*”.

CONSTEXPR OR INLINING WILL SAVE 255 INSTRUCTIONS!

C++ sometimes works very hard at compile time, but by doing so, it can eliminate unneeded work at runtime.

In our example, we completely eliminated any actual runtime code for fibonacci… but only for calls with a constant argument.

If a constexpr function is called with a non-constant argument, it will simply be evaluated as much as possible at compile time but the computation will still need to be be finalized by calling it at runtime.

A CONCRETE PUZZLE

Consider this program:

Why doesn't inline cause an infinite recursion in the compiler?

```
#include <iostream>
using namespace std;

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

int main(int argc, char**argv)
{
        for(int n = 1; n < 10; n++)
        {
                cout << "fibonacci(" << n << ") is " << fibonacci(n) << endl;
        }
        return 0;
}
```

RECURSIVE INLINING?

In principle, if we call this version of fibonacci with a constant, it "should" expand it fully, then collapse the expression by realizing that constant arithmetic suffices.

But this centers on the compiler realizing that if it starts with n=1..10, then n-1-1…-1 eventually reaches 0, hence the right hand side of the return statement becomes dead code!

The compiler limits how much recursion it is willing to do at compile time. If it "gives up" it simply produces a call to normal code.

WHAT ABOUT THE FOR LOOP?

In fact, C++ can "tell" that the for loop iterates over 10 constant values: 1, 2, … 10

In principle, it should be able to do a constexpr evaluation for each of the values, but it is hard to know whether it will get this right. It may depend on the "optimization level" you pick.

For something that fancy C++ can be a bit unpredictable.

HOW DO THESE FEATURES "INTERPLAY" WITH VECTORIZATION?

To write code that will vectorize nicely, it is very important that the compiler can determine:

- Sizes of your vectors and matrices
- Loop "stride" values: The increment in a for loop
- Expressions used to access matrix or vector elements
- Values used to "map" from some input x to mapped[x]

For such purposes, constexpr arithmetic can be incredibly useful!

SOME “ODDITIES” TO KNOW ABOUT

Suppose that MAXN is a const int. What does

```
const int MAXN = 10000;
const int* maxnptr = &MAXN;
```

mean? … Is maxnptr…

(1) a normal pointer to a const int, or

(2) is the pointer maxnptr itself a constant?

SOME "ODDITIES" TO KNOW ABOUT

Suppose that MAXN is a const int. What does

    const int MAXN = 10000;

    cons

mean?   .

const int is like a type

so., const int* is like a "pointer to a const int"

Compiler will reject this as illegal (a const int has a value, but no associated memory)

(1) a normal pointer to a const int, or

(2) is the pointer maxnptr itself a constant?

"CONST" STATEMENTS ARE PROMISES <u>YOU</u> MUST KEEP

Suppose I do this:

```
const int MAXN = 10000;      // MAXN is a constant (10000)
int *mptr = (int*)&MAXN;     // MAXN is really a const int
*mptr = 5000;                // What would this do?
```

In C++ this code sequence is illegal: it modifies a constant.

"CONST" STATEMENTS ARE PROMISES YOU MUST KEEP

Suppose I do this:

```
const int MAXN = 10000;     // MAXN is a constant (10000)
int *mptr = (int*)&MAXN;    // MAXN is really a const int
```

The compiler should complain that you are not permitted to take the address of a constant.

The error message will probably say that &MAXN is not a legal "rval"

do?

In C++ this code sequence is illegal: it modifies a constant.

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?

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:

auto requires a form of constexpr computation

```
std::map<std::string, Bignum> the_map;
…
f  auto  m: the_map) {
       << “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)

EXAMPLE OF AN AUTO-DISCOVERED TYPE

When creating my “Bignum” solution, I once ran into this:

```
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

WE SAW CONSTANT EXPRESSION MATH IN LECTURE 8, TOO!

C++ depends upon it to recognize parallelizable logic.

In fact, even code rearrangement can be understood as a form of constant expression evaluation: the code is like an expression, and all the variant forms of it are “equivalent” representations

This conceptual insight is key to modern compilation…

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

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 / 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.