C DEFINES AND C++ TEMPLATES

Professor Ken Birman
CS4414 Lecture 10

COMPILE TIME "COMPUTING"

In lecture 9 we learned about const, constexpr and saw that C++ really depends heavily on these

- Ken's solution to homework 2 runs about 10% faster with extensive use of these annotations
- Constexpr underlies the "auto" keyword and can sometimes eliminate entire functions by precomputing their results at compile time.
- Parallel C++ code would look ugly without normal code structuring. Const and constexpr allow the compiler to see "beyond" that and recognize parallelizable code paths.

... BUT HOW FAR CAN WE TAKE THIS IDEA?

Today we will look at the concept of programming the compiler using the templating layer of C++

We will see that it is a powerful tool!

There are also programmable aspects of Linux, and of the modern hardware we use. By controlling the whole system, we gain speed and predictability while writing elegant, clean code.

IDEA MAP FOR TODAY

History of generics: #define in C

We have seen a number of parameterized types in C++, like std::vector and std::map

These are examples of “templates”. They are like generics in Java

Templates are easy to create, if you stick to basics

The big benefit compared to Java is that a template is a compile-time construct, whereas in Java a generic is a run-time construct.

The template language is Turing-complete, but computes only on types, not data from the program (even when constants are provided).

CONCEPT OF A GENERIC OR TEMPLATE

A class that can be specialized by providing element types as class arguments.

For example, "a list of pets" or "a map from strings to counters"

This separates the abstraction the class implements from the specific types of objects it manages.

EARLY HISTORY OF GENERICS

Many trace their roots to C, the original language introduced with Unix (which was the original Linux)

C++ still has C as a subset… C++ will compile a C program.

But C lacked classes, so object oriented coding was infeasible

C INTRODUCED A SIMPLE FORM OF MACRO SUBSTITUTION

The most basic option just defines a replacement rule:

```
#define SOMETHING  something-else
```

But this wasn't enough, and people added parameters

```
#define SOMETHING(x)  something-else that uses x
```

EXAMPLES

Examples using #define

```
#define OPT1    0x00001

#define DEBUGMODE  1

#define SIGN(x)  (x < 0: -1: 1)

#define ERRMSG(s) { if(DEBUGMODE) printf(s); }
```

ONE USE OF #DEFINE(T) WAS FOR TYPES

Allows a library method to be specialized for a single type. But C code gets confusing if #define(T) is “respecialized” for multiple uses in different places.

For example, if you use #define(T) Pet to specify the type in a list of pets, you wouldn’t also be able to have a list of desserts.

IN FACT, THE C "PREPROCESSOR" DOES MORE

#if, #else/#elif, #endif offer some limited compile-time control.

The "dead code" vanishes, and the compiler never even sees it.

Typical uses: compile one version of the code if the computer has a GPU, a different version if it lacks a GPU. Or have debugging logic that vanishes when we aren't testing.

… BUT THEY ARE TOO LIMITED

As noted, we couldn't create a type that can be parameterized with types of objects using it, or that it holds.

And we can't ***reason about types*** in #if statements, which have very limited conditional power.

All of these limitations frustrated C users. C++ *to the rescue!*

WHILE ALL OF THIS WAS PLAYING OUT, LANGUAGES BECAME OBJECT ORIENTED

Java was the first widely successful OO language, but in fact there were many prior attempts. Java used pragmas and annotations for some roles "similar" to what C did with #define, #if, etc.

A very large community began to use objects… but early decisions resulted in runtime reflection overheads (discussed previously)!

JAVA POLYMORPHISM

Allows a single method with a single name to have varying numbers of arguments, and varying argument types.

The compiler had the task of matching each invocation to one of the polymorphic variants.

JAVA GENERICS

In the early versions of Java, a class such as list or a hash map would just treat the values as having "object type".

This, though, is impossible to type check: ***"is this a list that only includes animals that can bark, or might it also have other kinds of animals on it, or even non-animals?"***

Java generics solved that problem, but Java retained the older form of unresolved object types as a form of legacy.

THE POWER OF GENERICS

In fact Java's generics are amazingly powerful.

You can literally load a Java JAR file, see if it implements class List with objects that all support operations Bark, Sit, LieDown, etc, and if so, call them.

This is done using *runtime reflection* in which a program can take a reference to a class (even one loaded from a JAR file) and enumerate over the variables, types and methods it defines

THE ISSUE WITH JAVA GENERICS

The language never eliminated the universal “object” class, which is the common supertype for all the more specific Java classes.

As a result, Java needed an **instanceof** test, as well as other features, so that the runtime can figure out what types of objects it is looking at (for runtime type checking) and also which method to call (for polymorphic method invocations)

C++ TEMPLATE GOALS

When C++ was designed, the owners of its template architecture were aware of the C and Java limitations.

They wanted to find a "pure" way to express the same concepts while also programming in an elegant, self-explanatory way, and they wanted to do this without loss of performance.

TEMPLATES AND POLYMORPHISM IN C++

Polymorphic method calls, but resolved at compile time. Extensible classes, but flexible and able to look at object types and generate different logic for different types.

C++ lacks the equivalent of the Java **runtime** "instanceof".

- It does have a **compile** time instanceof.
- In C++ all types are fully resolved at compile time.
- Every C++ object has a single and very specific type

TEMPLATES AND POLYMORPHISM IN C++

… in fact, even <u>polymorphism</u> in C++ is resolved at compile time!

C++ is always able to identify the specific method instance to call.

C++ even dynamically loads libraries without worrying that somehow the library methods won't be what it expects.

TEMPLATES AND POLYMORPHISM IN C++

… but there is one powerful feature that is very much "like" runtime polymorphism: inheritance of "fully virtual classes"

In C++ we often define a virtual class that describes a standard set of methods shared across some set of different classes. So for example, IBark could be an interface shared by "animals that know how to bark", with a method "bark".

INTERFACE CLASSES IN C++

For example:

```
class shape   // An interface class
{
  public:
    virtual ~shape() {};
    virtual void move_x(int x) = 0;
    virtual void move_y(int y) = 0;
    virtual void draw() = 0;
//...
};
```

```
class line : public shape
{
  public:
    virtual ~line();
    virtual void move_x(int x); // implements move_x
    virtual void move_y(int y); // implements move_y
    virtual void draw(); // implements draw
  private:
    point end_point_1, end_point_2;
//...
};
```

INTERFAC

Some developers prefer names like IShape but in fact, there is no rule

Says that any class inheriting the shape interface must define this method.

Tells C++ that this is "supposed to match" a virtual method inherited from some other class (in our case, from "shape")

For exampl

```
class shape   // An interface class
{
  public:
    virtual ~shape() {};
    virtual void move_x(int x) = 0;
    virtual void r
    virtual void c
//...
};
```

```
class line : publ
{
  public:
    virtual ~line();
    virtual void move_x(int x); // implements move_x
    virtual void move_y(int y); // implements move_y
    w(); // implements draw
    _1, end_point_2;
//...
};
```

We are looking at line.hpp, which has the type signature but not the implementation. Line.cpp is required to implement line::move_x(int x), etc.

THESE FULLY VIRTUAL CLASSES ARE INHERITED BY CONCRETE CLASSES

A class like Dog would inherit a fully virtual class like IBark.

Dog is required to provide implementations (code bodies) for the virtual IBark methods that had = 0.

YOU CAN TEMPLATE AN INTERFACE!

The syntax is just like a template for any other class.

This allows a very powerful form of runtime polymorphism

TEMPLATES ALSO HAVE A FORM OF COMPILE-TIME "INSTANCEOF" FEATURE

You can check to see if a type has some specific characteristic and generate code conditional on that.

For example, a template could check to see if the given type supports IBark and if so, call the bark method. But then if not, it could check for IPurr. And then for IChirp…

This all occurs when the template is "instantiated" at compile time

C++ TEMPLATES

Botton line: they can do everything Java generics can do, but at compile time, and also cover defines, varargs, etc.

We will start with simpler cases that you might often want to use, then will just “skim” the fancier things seen in C++ libraries, but that normal mortals don’t normally need to actually do.

SUMMARY OF TEMPLATE GOALS

Compile time type checking and type-based specialization.

A way to create classes that are specialized for different types

Conditional compilation, with dead code automatically removed

Code polymorphism and varargs without runtime polymorphism

C++ ADVANTAGE?

It centers on the compile-time type resolution. Impact? The resulting code is blazingly fast.

In fact, C++ wizards talk about the idea that at runtime, all the fancy features are gone, and we are left with "plain old data" and logic that touches that data mapped to a form of C.

The job of C++ templates is to be as expressive as possible without ever requiring any form of runtime reflection.

THE BASIC IDEA IS EXTREMELY SIMPLE

As a concept, a template could not be easier to understand.

Suppose we have an array of objects of type int:

```
int     myArray[10];
```

With a template, the user supplies a type by coding something like Things<long>. Internally, the class might say something like:

```
T       myArray[10];
```

THE BASIC IDEA IS EXTREMELY SIMPLE

As a concept, a template could not be easier to understand.

Suppose we have an array of objects of type int:

```
int     myArray[10];
```

With a templat nd we express this by just coding:

T behaves like a variable, but the "value" is some type, like int or Bignum

```
T     myArray[10];
```

TO ACCESS THIS FUNCTIONALITY, YOU CREATE A TEMPLATE FOR A CLASS

```
template<typename T>
class Things {
    T       myArray[10];
    T       getElement(int);
    void  setElement(int,T);
}
```

YOU CAN ALSO SUPPLY A CONSTANT

```
template<class T, const int N>
class Things {
    T       myArray[N];
    T       getElement(int);
     void  setElement(int,T);
}
```

TEMPLATED FUNCTIONS

Templates can also be associated with individual functions. The entire class can have a type parameter, but a function can have its own (perhaps additional) type parameters

```
Template<typename T>
T max(T a, T b)
{
    return a>b? a : b;                    // T must support a > b
}
```

FUNCTION TEMPLATES

Nothing special has to be done to use a function template

```
int main(int argc, char* argv[]) {
    int     a = 3, b = 7;
    double  x = 3.14, y = 2.71;

    cout << max(a, b) << endl;   // Instantiated with type int
    cout << max(x, y) << endl;   // Instantiated with type double
    cout << max(a, x) << endl;   // ERROR: types do not match
}
```

Note that in these examples, the type is automatically inferred by C++

CLASS TEMPLATES

```
template <class T>
class myarray {
private:
   T* v;
   int sz;
public:
   myarray(int s)  { v = new T [sz = s]; }          // Constructor
   myarray(const myarray& b) { v = b.v; sz = b.sz; }// Copy constructor
   ~myarray()  { delete[] v; }                      // Destructor
   T& operator[] (int i)  { return v[i]; }
   size_t size()  { return sz; }
};
```

You can instantiate the same templated class with different types

```
myarray<int> intArray(10);
myarray<shape> shapeArray(10);
```

CLASS TEMPLATES

Developer of this class wanted T to be a class type (not a base type like int, double, etc)

```
template <class T>
class myarray {
private:
   T* v;
   int sz;
public:
   myarray(int s)  { v = new T [sz = s]; }          // Constructor
   myarray(const myarray& b) { v = b.v; sz = b.sz; }// Copy constructor
   ~myarray()  { delete[] v; }
   T& operator[] (int i)  { return v[i]; }
   size_t size()  { return sz; }
};
```

Syntax is fine, but gives a compilation error: int is a type, but it is not a class type (not a C++ object)

You can instantiate the same templated class with different types

```
myarray<int> intArray(10);
myarray<shape> shapeArray(10);
```

TYPEDEF

Typedef allows you to give a short name to a type that might otherwise require a long name.

Examples:

```
typedef vector<int> VecInt;
typedef map<string, tuple<double, int, float, MyClass> > MyTuple;
```

USING IS SIMILAR TO TYPEDEF

Syntax is “using name = type;”

Can appear in a template definition

(1) using name = type ;

(2) template < template-parameter-list >
using name = type ;

TEMPLATE TYPES CAN BE “CONSTRAINED”

Suppose that we want to build a template for a class with a method “speak()” that calls “bark()”.

Dogs and seals bark. Cats do not. So we might want to restrict our template type:

template<typename T> requires T instanceof(Ibark)

TEMPLATE TYPES CAN BE "CONSTRAINED"

We might even want to implement a given function in a different way for different types of objects. You could do this using instanceof "inline" in your code, but it is handled at compile time.

This rule introduces some complications.

C++ has many options; we will just look at one of them. See **https://en.cppreference.com/w/cpp/language/constraints**

REQUIRES

This clause allows you to say that the template type must implement some interface.

This says that the template is only valid for classes that define equality testing, or for types that are “aliases” of the void type.

```
template<typename T> requires EqualityComparable<T> || Same<T, void>
```

https://en.cppreference.com/w/cpp/language/constraints

VARIABLE ARGUMENT LISTS

Methods with variable numbers of arguments are also a traditional source of "confusion" in strongly typed languages.

In C, there are many methods like printf:

```
printf("In zipcode %5d, today's high will be %3.1f\n",
       local_zipcode, local_temperature);
```

… notice that the format expects specific types of arguments!

VARARGS ARE HARD TO TYPE CHECK

In Java, varargs can easily be supported using object type, and there are standard ways to iterate over the arguments supplied

But this means we are forced to do runtime type checking later, when trying to "do something" to those objects, like convert to a string for printing.

C++ wanted this same power with strong compile-time typing

… IN C, THIS CAN SIMPLY RESULT IN BUGS

If the temperature passed to this printf, in C, is of type int or some form of low-precision float type, printf will just print a nonsense output.

The C++ designers wanted generics to *also* address this issue, and they came up with an insane concept (that works): one version of printf (or whatever) for every sequence of types actually used in the code. Polymorphism to the max!

WHAT???

Consider this case:

```
printf("%d,%f,%d,%s\n", 2, 3.0, 4, "5.7");
```

… C has many other methods like this, including ones that arise in totally different situations (for example to handle networking addresses, which come in many flavors, like IPv4 versus IPv6).

WHAT???

printf("%d,%f,%d,%s\n", 2, 3.0, 4, "5.7");

The idea in C++ was to allow such things, but "translate" them to runtime code that has one version of the method (printf, in our example) for each type actually used:

printf(char *format, int i0, float f0, int i1, char* s0) { … }

printf(char *format, float f0, int i1, char* s0) { … }

printf(char *format, int i1, char* s0) { … }

WASTE OF SPACE?

Computers have a lot of memory, and you aren't likely to really use a million permutations of types. Code is fairly compact.

So they concluded that no, this won't waste space. And it does allow for very effective type checking, at compile time!

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.

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

Only these lines "generate code"!

```
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

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

```
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 "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)

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;
```

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

… Or even (if the compiler can infer that optr was initially 0):

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

WHAT ABOUT CHECKING THE FORMAT AGAINST THE ARGUMENT TYPES?

This template actually has extra code to type-check the arguments

I didn't show that code because it made the slide a bit bloated

In GNU C++, -Wformat does printf type checking as a separate compiler feature, because C++ 03 couldn't do this type checking. But starting with C++ 17, constexpr will let the template check!

CONDITIONAL COMPILATION

C++ offers several ways to get the behavior of #if…#endif

(1) With a constant variable, the compiler will do constant expression evaluation of if(HAS_GPU) { … } and can trim any "dead" code paths

(2) The templating mechanism has a way to test types at compile time, and can output different code blocks for different types (type traits, concepts)

EXAMPLE OF A C++ “CONCEPT”

A concept is a **compile-time type test**, part of the templating “language” in C++. Useful in “requires” clauses.

For type T, this example defines ‘EqualityComparable to mean “implements the operators == and !=“.

```
template<typename T>
concept EqualityComparable = requires(T a, T b) {
    { a == b } -> std::boolean;
    { a != b } -> std::boolean;
};
```

THE C++ TEMPLATE LANGUAGE IS TURING COMPLETE!

In theory, any program you could write and run on any computer can be "recoded" as a template and executed at compile time!

In practice… that might not work very well! For one thing, the C++ template processor is a very <u>slow</u> Turing machine!

MOST COMMON COMPLAINT?

Template programming is challenging to learn

- This recursive compile-time language doesn't resemble C++ (it looks more like Haskell)
- C++ compile-time error messages are bizarre because fully expanded types can be really hard to make sense of
- Compilation of a templated C++ program requires many passes and helper files, to avoid creating multiple instances of the same procedure with the same argument types.

SUMMARY

Generic programming allows for the abstraction of types

C++ templates are an instantiation of generic programming

C++ has function templates and class templates

Templates have many uses and allow for very interesting code design. An entire "compile time language", similar in style to Haskell (a functional language), extremely elaborate.

SUMMARY

More broadly, templates and const/constexpr tie to the idea of conceptual abstractions.

These tools let us control elements of the environment: *the way our code will be transformed into executable logic*. Linux has many other programmable components, and this idea is pervasive.

As a systems programmer, this idea of programmable control is a central concept you will use again and again.