PA5 Notes; Templates

ITP 435
Week 8, Lecture 1

PA5

- You will implement a “virtual machine”/“emulator” for the classic ITP-11 computer system with the Turtle Processing Unit™

- Features of this advanced system include:
  - Fifteen 32-bit integer registers
  - 1 KB of stack space
  - 3-bit color graphics (eight total colors!!!)

PA5

- Why?
  - Be the machine
  - We can use template metaprograms, exceptions, some other C++ features
  - It’s fun!

Parsing Input file

- The input files contain one or more commands, like this:

```
movi tx,110
movi ty,105
movi tc,1
pendown
mov r7,r0
movi r6,5
movi r1,100
movi r2,144
fwd r1
add tr,tr,r2
inc tc
inc r7
cmplt r7,r6
movi r5,6
jt r5
exit
```

Parsing Input file

```
movi tx,110
movi ty,105
movi tc,1
pendown
mov r7,r0
movi r6,5
movi r1,100
movi r2,144
fwd r1
add tr,tr,r2
inc tc
inc r7
cmplt r7,r6
movi r5,6
jt r5
exit
```

- Some take no parameters
- Some take a single parameter
- Some take a comma-separated list of parameters
- Some parameters are integers, some are strings (register names)

Doing this without template metaprograms

- We'd first declare a base class for all ops:

```
struct Op
{
   virtual const char* GetName() const = 0;
   virtual void Parse(const std::string& params) = 0;
   virtual void Execute(class Machine& machine) = 0;
};
```

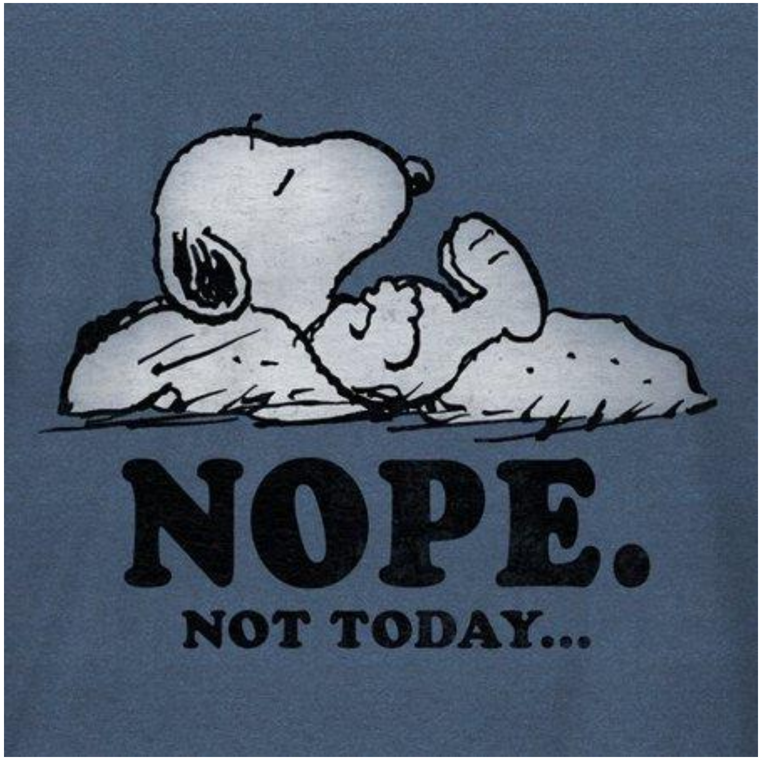
Doing this without template metaprograms

- Then maybe make a subclass for each op, like:

```
// movi reg1,int
// reg1 = int
struct MovI : Op {
   const char* GetName() const override { return "movi"; }
   void Parse(const std::string& params) override {
      // Split comma-separated list
      std::vector<std::string> paramV = Split(params);
      opReg1 = paramV[0]; // This is a string
      opInt = std::stoi(paramV[1]); // Gotta convert this to int!
   }
   void Execute(class Machine& machine) override
   { /* Do whatever */ }
   // Member data for this op!
   std::string opReg1;
   int opInt;
};
```

Doing this without template metaprograms

- Now repeat writing that parse function, and specifying member data for each of the 20+ different ops!
- I say…



We have a pattern

- Every op takes in a comma-separated list of 0 or more params
- We need member data to store each param with the correct type
- We want to convert each param to the desired type (if needed)

We have a pattern

- Every op takes in a comma-separated list of 0 or more params
- We need member data to store each param with the correct type
- We want to convert each param to the desired type (if needed)
- We want to define this pattern, and ***make the compiler generate*** the correct member data/parsing code for each op!

Into the Template Rabbit Hole

Template Metaprogramming (TMP)

- Discovered by accident while templates were being standardized
- Is Turing-complete
- We can define and use templates such that at compile time, they will instantiate to different instances of the template that give us what we want…

Basic Template Syntax (classes)

```
template <typename T>
class List
{
      // ...
};
```

Basic Template Syntax (functions)

```
template <typename T>
T max(T a, T b)
{
      return ((a > b) ? a : b);
}
```

Compiler Instantiation

- If you use this template in code as such:

```
max(1, 2); // Type not specified, compiler attempts substitution
max<char>('a', 'b'); // Type specified (optional)
```

- Compiler will instantiate two versions of our function:

```
int max(int a, int b)
{
     return ((a > b) ? a : b);
}

char max(char a, char b)
{
     return ((a > b) ? a : b);
}
```

Template Specialization

- Suppose you want to do something specific in max when the type is std::string (kinda weird in this example)

- You can then specify a specialization:

```
template <>
std::string max<std::string>(std::string a,
  std::string b)
{
     // Code specific for this case
}
```

std::tuple

- Tuples are like std::pair, except they can have zero to infinite members

- You can use the terminology like this:
  - 0-tuple = Empty
  - 1-tuple = Has one element
  - 2-tuple = Has two elements (it’s a pair)
  - 3-tuple = Has three elements
  - …

An empty tuple

- Although it seems weird, you can declare 0-tuple if your heart desires

```
std::tuple<> empty;
```

A 3-tuple example

- This example is more useful

```
// This has three members (an int, a char, and a float)
std::tuple<int, char, float> tuple3;

// Once constructed, use std::get<idx>
// to access a member of the tuple
// (Both for setting and getting the value)
std::get<0>(tuple3) = 50; // Set int member
char c = std::get<1>(tuple3); // Get char member

// The index passed to get MUST BE CONSTANT AT COMPILE TIME!!
```

std::make_tuple

- If I want to save typing and know what values I want to construct for my tuple, I can use make_tuple:

```
auto myTuple = std::make_tuple(50, "Hello", 10.0);
// myTuple has
// 0 - An int (50)
// 1 - A const char* pointing to "Hello"
// 2 - A double (10.0)
```

std::tuple_cat

- Given 0 or more tuples, constructs a tuple concatenating the tuples together

```
auto myTuple = std::make_tuple(1337);
auto otherTuple = std::make_tuple(std::string("Yo!"));

auto cat = std::tuple_cat(myTuple, otherTuple);
// cat has
// 0 - An int (1337)
// 1 - A std::string containing "Yo!"
```

The first insight

- We want to convert the input comma-separated string into a tuple of the correct types
- This is easier said than done
- We need to know the tuple types at compile time
- So, we have to be generate types through template magics based on the expected parameters

The ground floor

- Given an input string of a single element, we want to be able to convert it to a 1-tuple of the expected type
- Here is the generic template version…note I make it uncompilable by not actually returning anything

```
// Generic version of ParseElem
// If this generic version is instantiated, it won't compile
template <typename T>
std::tuple<T> ParseElem(const std::string& elem)
{ }
```

ParseElem specializations

- Next, I make specializations that convert string to tuple<int> or to tuple<string>

```
// Specialization of ParseElem<int>
// Converts elem to a tuple{int}
template <>
inline std::tuple<int> ParseElem<int>(const std::string& elem) {
   return std::make_tuple(std::stoi(elem));
}

// Specialization of ParseElem<std::string>
// Just makes a tuple{str} from elem
template <>
inline std::tuple<std::string>
   ParseElem<std::string>(const std::string& elem) {
   return std::make_tuple(elem);
}
```

How would I use ParseElem?

```
// This will give us a tuple<int>
auto a = ParseElem<int>("11");

// This will give us a tuple<string>
auto b = ParseElem<std::string>("r1");

// This won't compile
auto c = ParseElem<double>("50.0");
```

ParseStr – The secret sauce begins

- Now we declare a function called ParseStr that:
  - Has 0 or more template parameter types
  - Returns a tuple corresponding to the 0 or more template types
  - Takes in a vector of strings (the different parameter strings)

```
template <typename... Args>
std::tuple<Args...> ParseStr(std::vector<std::string>& paramV);
```

ParseStr – The secret sauce begins

- Now we declare a function called ParseStr that:
  - Has 0 or more template parameter types
  - Returns a tuple corresponding to the 0 or more template types
  - Takes in a vector of strings (the different parameter strings)

```
template <typename... Args>
std::tuple<Args...> ParseStr(std::vector<std::string>& paramV);
```

- The … syntax is a ***variadic template***, a template that takes in zero or more types
- Notice how we can directly pass that list of types into the template parameter of std::tuple

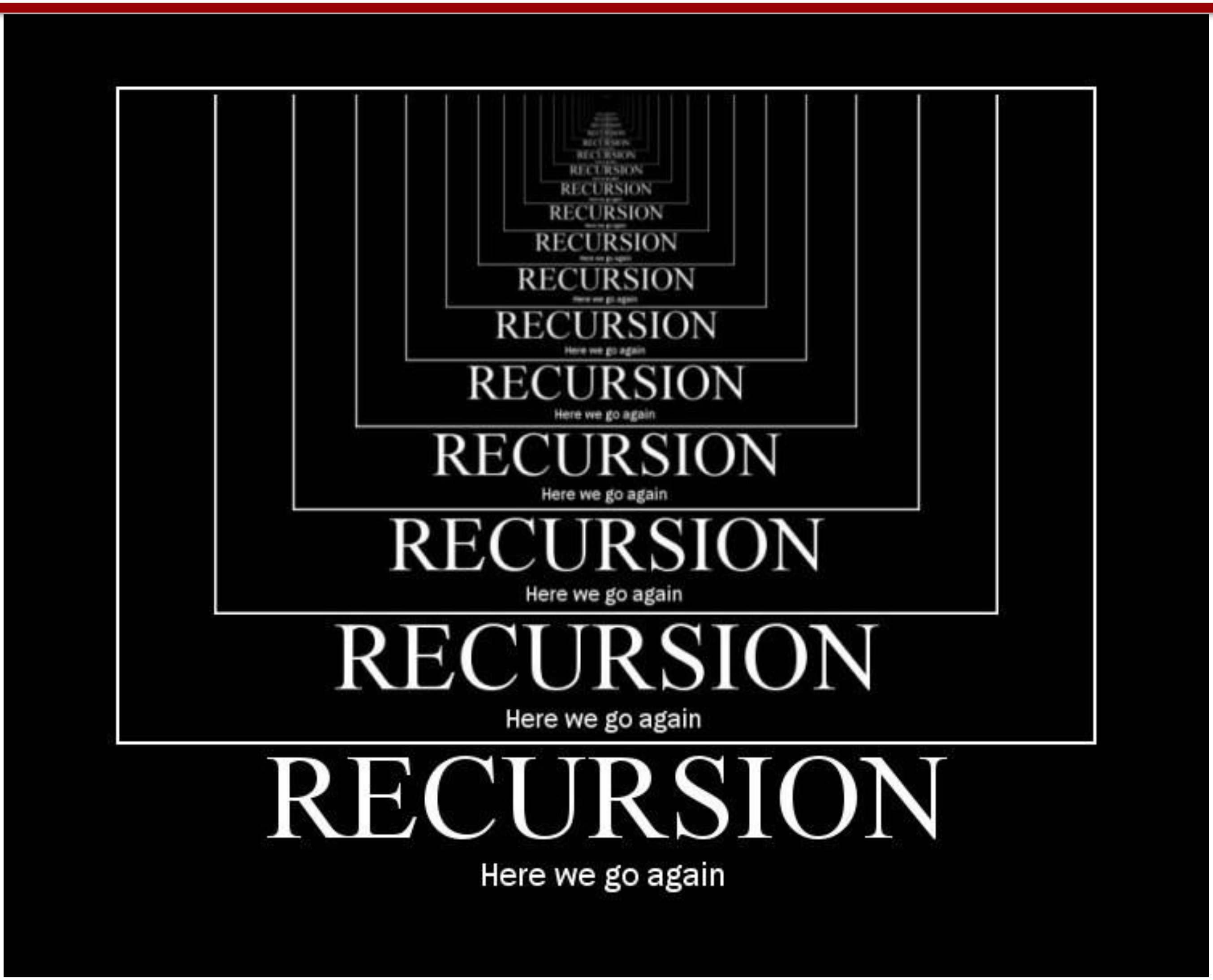
ParseStr – Specialization for zero types

- We make a specialization for where there are no types passed to the template:

```
template <>
inline std::tuple<> ParseStr<>(std::vector<std::string>& paramV)
{
    return std::make_tuple();
}
```

- This is going to serve as a "base case"

Uh oh


RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again

Recursive Templates

```
// Calculate a Fibonacci number at COMPILE TIME
template <unsigned int i>
struct Fibonacci {
   static const int value = Fibonacci<i - 2>::value + Fibonacci<i - 1>::value;
};
// The "base cases" are i == 0 or i == 1
// So use specialization...
template <>
struct Fibonacci<0> {
   static const int value = 0;
};
template <>
struct Fibonacci<1> {
   static const int value = 1;
};
// This would be 610 (at COMPILE TIME)
Fibonacci<15>::value
```

ParseStrHelper – Declaration

- Declare ParseStrHelper, which has these template params:
  - A single type T
  - Zero or more additional types Args

```
template <typename T, typename... Args>
std::tuple<T, Args...> ParseStrHelper(std::vector<std::string>& paramV)
```

- Notice how again, we forward the types in that order to the tuple

ParseStrHelper – Declaration

```
template <typename T, typename... Args>
std::tuple<T, Args...> ParseStrHelper(std::vector<std::string>& paramV)
```

- Using this pattern allows us to separate the first type in the template parameters from the remainder
- We want to convert the first type with ParseElem and forward the remaining to a recursive call of ParseStr!

ParseStrHelper – Implementation

```
template <typename T, typename... Args>
std::tuple<T, Args...> ParseStrHelper(std::vector<std::string>& paramV)
{
   // Get the last string from the vector
   // This assumes paramV is in reverse
   std::string elem = paramV.back();
   paramV.pop_back();

   // ParseElem<T>(elem) takes elem and converts it into a tuple{T},
   //
   // Then, take the remaining elements in paramV, and pass
   // it to ParseStr for the remaining variadic types (Args...)
   // (This is the recursive step)
   ///
   // Concatenate these tuples with tuple_cat
   return std::tuple_cat(ParseElem<T>(elem),
      ParseStr<Args...>(paramV));
}
```

ParseStr – Generic implementation

- All ParseStr does is forwards the arguments to ParseStrHelper

- You need this extra indirection so ParseStrHelper can separate the first type from the rest

```
template <typename... Args>
std::tuple<Args...> ParseStr(std::vector<std::string>& paramV)
{
    return ParseStrHelper<Args...>(paramV);
}
```

It's complicated.

- Yes, it is
- Luckily, I just showed you all the code…you just have to type it into your program

ParseStr in action

- Suppose I have this code:

```
std::vector<std::string> paramV = {"5", "r1"};
auto t = ParseStr<std::string, int>(paramV);
```

ParseStr in action

```
std::vector<std::string> paramV = {"5", "r1"};
auto t = ParseStr<std::string, int>(paramV);
```

At compile, time, the compiler has to “instantiate” all the templates we just declared, starting from the top:

- ParseStr<Args={std::string, int}>
  - ParseStrHelper<T=std::string, Args={int}>
    - ParseElem<T=std::string>
    - ParseStr<Args={int}>
      - ParseStrHelper<T=int, Args={}>
        - ParseElem<T=int>
        - ParseStr<Args={}>

ParseStr in action

```
std::vector<std::string> paramV = {"5", "r1"};
auto t = ParseStr<std::string, int>(paramV);
```

This means the type of t is:

std::tuple<std::string, int>

ParseStr in action

```
std::vector<std::string> paramV = {"5", "r1"};
auto t = ParseStr<std::string, int>(paramV);
```

After executing this code:

- std::get<0> gives me a string with “r1”
- std::get<1> gives me an int with 5

It’s in reverse, but that’s intentional

Back to Ops, Back to Reality

- So how do we leverage this for the ops?

```
template <typename... Args>
struct OpBase : Op
{
    void Parse(const std::string& params) override
    {
        // Split
        std::vector<std::string> paramV = Split(params);
        // Reverse vector for simplicitly of template metaprogram
        std::reverse(paramV.begin(), paramV.end());
        // Generate the tuple
        mParameters = ParseStr<Args...>(paramV);
    }

    // Tuple to hold op arguments
    std::tuple<Args...> mParameters;
};
```

MovI

- Now declare the op as a subclass of OpBase with the correct template parameters:

```
// movi reg1,int
// reg1 = int
struct MovI : OpBase<std::string, int>
{
   const char* GetName() const override { return "movi"; }
   void Execute(class Machine& machine) override
   { /* Still have to implement this/ }
};
```

MovI

```
struct MovI : OpBase<std::string, int>
```

This means that…

- MovI will have an mParameters tuple with:
  - 0 – String name of register
  - 1 – Integer value to store in register

- The correct Parse function is automatically generated for you by the template metaprogram!

In-class activity

One More Thing…

Constructing the Correct Op Subclass

```
// Open the file
while (/* Not eof */) {
   // Get the string for op name and params
   std::string opName = /*blah*/;
   std::string params = /*blahblah*/;
   std::shared_ptr<Op> ptr;
   if (opName == "movi") {
      ptr = std::make_shared<MovI>();
   } else if (opName == "exit") {
      ptr = std::make_shared<Exit>();
   } else if (opName == "add") {
      ptr = std::make_shared<Add>();
   }
   // Yikes.........
   // .........
   ptr->Parse(params);
}
```

We have a pattern

- Given a string corresponding to the op, we want to construct a shared pointer to the correct type

- We want a map where:
  - Key is string name of op
  - Value is a “factory method”/“creator function” that knows how to construct said type

The CreateOp template function

- All it does is calls make_shared on the desired op subclass T

```
template <typename T>
std::shared_ptr<Op> CreateOp()
{
    return std::make_shared<T>();
}
```

- (This has to be a standalone or static function)

Now let's declare our map

```
std::map<std::string,
    std::function<std::shared_ptr<Op>()>>
    opMap;
```

- The key type is std::string (name of op)
- The value type is a function that takes in no parameters and returns a std::shared_ptr<Op>

Add Elements to the Map

```
opMap.emplace("movi", &CreateOp<MovI>);
```

- This says that for “movi” call CreateOp<MovI>

Now with the map

- Look how simple our code is

```
while (/* Not eof */0) {
   // Get the string for op name and params
   std::string opName = /*blah*/;
   std::string params = /*blahblah*/;

   // Look up the opName in our map, and call the
   // correct CreateOp function! (note the extra
   // parenthesis at the end, that's the function call)
   std::shared_ptr<Op> ptr = opMap.at(opName)();
   ptr->Parse(params);
}
```

And Just For Fun…

For more fun…

- We can meta-metaprogram via the power of X-macros, you could also generate those map entries
- You could even use X-macros to auto-generate all the OpBase subclass declarations!

If we want to X-Macro…

- Make a Ops.def file, like:

```
// Define all the ops our VM supports, along with parameters
// Syntax: OP(textName, className, args...)
OP(exit,Exit)
OP(movi,MovI,std::string,int)
OP(add,Add,std::string,std::string,std::string)
OP(mov,Mov,std::string,std::string)
//...
```

To declare the Op subclasses

```
// Define version of OP macro that creates declaration
// __VA_ARGS__ is an expansion of variadic arguments
#define OP(textName,className,...) \
struct className : OpBase<__VA_ARGS__> \
{ \
   const char* GetName() const override { return #textName; } \
   void Execute(class Machine& machine) override; \
};

// This will take definitions and apply OP macro to each
#include "Ops.def"

// Undefine the OP macro
#undef OP
```

To declare the opMap

```
// This macro converts a single OP definition to the appropriate
// {key,value} pair entry in the map
#define OP(textName,className,...) { #textName, &CreateOp<className> },

std::map<std::string, std::function<std::shared_ptr<Op>()>> opMap =
{
#include "Ops.def"
};

#undef OP
```

Should we do this?

- We ***can*** do this, but ***should*** we?

- It may make our code more difficult to follow, debug, and our fellow developers may hate us

- But it still is a cool trick to have up your sleeve for special occasions!