More Templates

ITP 435
Week 8, Lecture 2

Vectors and emplace_back

This vector default constructor is wrong.

```
template <typename T>
class Vector {
public:
   // Default constructor allocates to capacity
   Vector()
   :mCapacity(20)
   ,mSize(0)
   ,mData(new T[mCapacity])
   { }

   void push_back(const T& e) {
      if (mSize >= mCapacity) {
         // Grow vector...
      }
      mData[mSize] = e;
      mSize++;
   }

   ~Vector() {
      delete[] mData;
      mCapacity = 0;
      mSize = 0;
   }
private:
   size_t mCapacity;
   size_t mSize;
   T* mData;
};
```

What's wrong?

- Say I have this:

```
struct Test2 {
   Test2(int i) {}
};
```

- Test2 doesn't have a default constructor, so this will not compile:

```
Vector<Test2> v; // This doesn't compile
```

- However, STL supports it!

```
std::vector<Test2> v; // This compiles
```

Specifically…

```
template <typename T>
class Vector {
public:
   // Default constructor allocates to capacity
   Vector()
   :mCapacity(20)
   ,mSize(0)
   ,mData(new T[mCapacity])
   { }

   void push_back(const T& e) {
      if (mSize >= mCapacity) {
         // Grow vector...
      }
      mData[mSize] = e;
      mSize++;
   }

   ~Vector() {
      delete[] mData;
      mCapacity = 0;
      mSize = 0;
   }
private:
   size_t mCapacity;
   size_t mSize;
   T* mData;
};
```

`new` both allocates and default-constructs!

What do we want?

```
Vector()
  :mCapacity(20)
  ,mSize(0)
  ,mData(new T[mCapacity])
  { }
```

- The vector constructor must only allocate memory up to capacity, but ***not*** construct any elements

What do we want?

```
Vector()
  :mCapacity(20)
  ,mSize(0)
  ,mData(reinterpret_cast<T*>(std::malloc(sizeof(T) * mCapacity)))
  { }
```

- Instead of using `new` we *only* want to allocate the memory so use `malloc`

What do we want?

```
Vector()
  :mCapacity(20)
  ,mSize(0)
  ,mData(reinterpret_cast<T*>(std::malloc(sizeof(T) * mCapacity)))
  { }
```

- This is the amount of memory we need

What do we want?

```
Vector()
  :mCapacity(20)
  ,mSize(0)
  ,mData(reinterpret_cast<T*>(std::malloc(sizeof(T) * mCapacity)))
  { }
```

- We must cast because `malloc` returns a `void*` and C++ does not implicitly convert from `void*` to other pointer types (unlike C)

Updating push_back

- Now when we push_back, we want to construct a single element of type T at the correct spot in the array:

```
void push_back(const T& e) {
   if (mSize >= mCapacity) {
      // Grow vector...
   }

   //Don't do it the old way!
   //mData[mSize] = e;

   // This copy constructs an instance of T into the specified memory
   new(&mData[mSize]) T(e);

   mSize++;
}
```

- This uses ***placement new***, which allows you to construct an object in already-allocated memory

push_back that move constructs

- Similar idea, just use an r-value reference and std::move:

```
void push_back(T&& e) {
    if (mSize >= mCapacity) {
        // Grow vector...
    }

    // Need to specifically say std::move here
    new(&mData[mSize]) T(std::move(e));
    mSize++;
}
```

Updating the destructor

- Rather than using `delete[]`, we need to manually invoke the destructor on constructed elements, and then `free` the memory:

```
~Vector() {
   // Manually destruct only the constructed elements
   for (size_t i = 0; i < mSize; i++) {
      mData[i].~T();
   }
   // Free the memory
   std::free(mData);
   mCapacity = 0;
   mSize = 0;
}
```

Implementing emplace_back

- The syntax is pretty weird!

```
template <typename... Args>
void emplace_back(Args&&...  args) {
   if (mSize >= mCapacity) {
      // Grow vector...
   }

   // Need to forward the arguments to T
   // (The syntax for forward is super funky)
   new(&mData[mSize]) T(std::forward<Args>(args)...);
   mSize++;
}
```

Implementing emplace_back

```
template <typename... Args>
void emplace_back(Args&&...  args) {
    if (mSize >= mCapacity) {
        // Grow vector...
    }

    // Need to forward the arguments to T
    // (The syntax for forward is super funky)
    new(&mData[mSize]) T(std::forward<Args>(args)...);
    mSize++;
}
```

- This declares a variadic template

Implementing emplace_back

```
template <typename... Args>
void emplace_back(Args&&... args) {
    if (mSize >= mCapacity) {
        // Grow vector...
    }

    // Need to forward the arguments to T
    // (The syntax for forward is super funky)
    new(&mData[mSize]) T(std::forward<Args>(args)...);
    mSize++;
}
```

- This says I want to pass all the parameters by r-value reference into the function (and call this list args)

Implementing emplace_back

```
template <typename... Args>
void emplace_back(Args&&...  args) {
    if (mSize >= mCapacity) {
        // Grow vector...
    }

    // Need to forward the arguments to T
    // (The syntax for forward is super funky)
    new(&mData[mSize]) T(std::forward<Args>(args)...);
    mSize++;
}
```

- This says I want ***perfect forwarding*** of everything in args to the constructor

Implementing emplace_back

```
template <typename... Args>
void emplace_back(Args&&...  args) {
    if (mSize >= mCapacity) {
        // Grow vector...
    }

    // Need to forward the arguments to T
    // (The syntax for forward is super funky)
    new(&mData[mSize]) T(std::forward<Args>(args)...);
    mSize++;
}
```

- ***Note***: Yes, the syntax is confusing. We don't expect you to memorize this syntax as it's something that's hard to remember!

In-class Activity

Templates

Default Parameter

- You can specify a default template parameter

```
template <typename T, int size = 20>
class A
{
};
```

- This would default to a size of 20:

```
A<int> my_A;
```

Static Assertion

- When using complex templates, we need a way at compile time to halt compilation if something is unexpected
- Ex. Suppose we have a class which is templated by an int. But we want to restrict it such that the int must be divisible by 16.

- We could just add a comment:

```
// Size MUST be divisible by 16
template <int size>
class MyClass
{
};
```

- But what if no one reads it? Or forgets about it?

static_assert

- C++11 compile time assertion (supported in Visual Studio 2010+)

```
// Size MUST be divisible by 16
template <int size>
class MyClass
{
  static_assert(size % 16 == 0, "Size must be divisible by 16");
};
```

- If you try to do this:

```
MyClass<15> test;
```

- `Error C2338: Size must be divisible by 16`

Simple TMP Example

- At *compile time*, we want to check whether or not a type is an char.

- We want this assert to get hit:

```
static_assert(is_char<int>::value, "Not a character!");
```

- But *not* this one:

```
static_assert(is_char<char>::value, "Not a character!");
```

is_char, Part 1

- First, create a “boolean” struct as a helper:

```
template <bool set>
struct boolean
{
   static const bool value = set;
};
```

- Then a couple of using statements…

```
using true_type = boolean<true>;
using false_type = boolean<false>;
```

is_char, Part 2

- Now let's declare a base template class for is_char

```
template <typename T>
struct is_char : false_type
{};
```

- And if we specialize for type char…

```
template <>
struct is_char<char> : true_type
{};
```

is_char, Part 3

- So what happens when the compiler sees this line?

```
is_char<char>::value
```

- First, it selects the specialized version of `is_char`, which inherits from `true_type`

- Since `true_type` has a `static const` value of `true`, the value of the overall expression is `true`

A slightly less trivial example…

is_unsigned

- It's the same, but more specializations…

```
template <typename T>
struct is_unsigned : false_type {};

template<>
struct is_unsigned<unsigned char> : true_type {};
template<>
struct is_unsigned<unsigned short> : true_type{};
template<>
struct is_unsigned<unsigned int> : true_type {};
template<>
struct is_unsigned<unsigned long> : true_type {};
template<>
struct is_unsigned<unsigned long long> : true_type {};
```

is_same

- What if you wanted to check if two types are the same?
- So we have a declaration with two types…

```
template <typename T1, typename T2>
struct is_same : false_type {};
```

- And a specialization where both types are the same…

```
template <typename T>
struct is_same<T, T> : true_type {};
```

But how?

- Question: How does the compiler know which one to pick?
- Answer: It tries the most restrictive specializations first.
- If it FAILS … *it's not an error*
- It tries the next one
- ***SFINAE:*** "Substitution failure is not an error" – more on this in a bit

Traits

- STL defines many helper classes for such uses in <type_traits>

- Example:

```
#include <type_traits>

template <typename T>
class MyTest
{
    static_assert(std::is_integral<T>::value, "Not an int!");
};
```

Some Traits

- Here are some of the many traits you can inquire about:

```
std::is_floating_point // Is it a float?
std::is_array // Is it an array?
std::is_function // Is it a function?
std::is_class // Is it a class?
std::is_pod // Is it plain-old data?
std::is_abstract // Is it an abstract class?
std::is_same // Are the two types the same?

std::is_default_constructible // Does it have a default constructor?
std::has_virtual_destructor // Is the destructor virtual?
```

Why use traits?

- One example is that you want to have a template function that only works for certain types…and you want to require specializations for other types

```
template<typename T>
void Write(T inData, uint32_t inBitCount = sizeof(T) * 8)
{
   static_assert(std::is_arithmetic<T>::value ||
      std::is_enum<T>::value,
      "Generic Write only supports primitive data types" );

   WriteBits(&inData, inBitCount);
}
```

SuperPrint

SFINAE – Substitution failure is not an error

- Now it gets a bit more complicated…
- Suppose we want to have a templated `superPrint` function
  - `superPrint` on a type with an iterator should do a range-based for
  - `superPrint` on a type without an iterator should just output it
- Only way to do this is with TMP

How to check if a type has an iterator

```
template <typename T>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<T>(nullptr))::value;
};
```

How to check if a type has an iterator, cont'd

```
template <typename T>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<T>(nullptr))::value;
};
```

```
// Test 1:
has_iterator<int>::value;
```

How to check if a type has an iterator, cont'd

```
template <int>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
         decltype(check<int>(nullptr))::value;
};
```

Step 1: T is replaced with int

```
// Test 1:
has_iterator<int>::value;
```

How to check if a type has an iterator, cont'd

```
template <int>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<int>(nullptr))::value;
};
```

Step 1: T is replaced with int

Step 2: Try substituting for the first check function … SFINAE because `int::iterator` is not a valid type

```
// Test 1:
has_iterator<int>::value;
```

How to check if a type has an iterator, cont'd

```
template <int>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<int>(nullptr))::value;
};
```

Step 1: T is replaced with int

Step 2: Failed substituting for the first check function

Step 3: Try substituting for the second check function … SUCCESS!

```
// Test 1:
has_iterator<int>::value;
```

How to check if a type has an iterator, cont'd

```
template <int>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<int>(nullptr))::value;
};
```

```
// Test 1:
has_iterator<int>::value;
```

Step 1: T is replaced with int

Step 2: Failed substituting for the first check function

Step 3: Second check was a success

Step 4: What is the declared return type of this function?

How to check if a type has an iterator, cont'd

```
template <int>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<int>(nullptr))::value;
};


// Test 1:
has_iterator<int>::value;
```

Step 1: T is replaced with int

Step 2: Failed substituting for the first check function

Step 3: Second check was a success

Step 4: What is the declared return type of this function?

Step 5: false_type::value is false

How to check if a type has an iterator, cont'd

```
template <typename T>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<T>(nullptr))::value;
};


// Test 2:
has_iterator<std::list<int>>::value;
```

How to check if a type has an iterator, cont'd

```
template <std::list<int>>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<std::list<int>>(nullptr))::value;
};
```

Step 1: T is replaced with std::list<int>

```
// Test 2:
has_iterator<std::list<int>>::value;
```

How to check if a type has an iterator, cont'd

```
template <std::list<int>>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
         decltype(check<std::list<int>>(nullptr))::value;

};
```

Step 1: T is replaced with int

Step 2: Try substituting for the first check function … SUCCESS! Because `std::list<int>::iterator` is a valid type!

```
// Test 2:
has_iterator<std::list<int>>::value;
```

How to check if a type has an iterator, cont'd

```
template <std::list<int>>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
          decltype(check<std::list<int>>(nullptr))::value;
};
```

Step 1: T is replaced with int

Step 2: First check was a success

Step 3: What is the declared return type of this function?

```
// Test 2:
has_iterator<std::list<int>>::value;
```

How to check if a type has an iterator, cont'd

```
template <std::list<int>>
struct has_iterator
{
   template <typename U>
   static true_type check(typename U::iterator*);

   template <typename U>
   static false_type check(...);

   static const bool value =
         decltype(check<std::list<int>>(nullptr))::value;
};


// Test 1:
has_iterator<std::list<int>>::value;
```

Step 1: T is replaced with int

Step 2: First check was a success

Step 3: What is the declared return type of this function?

Step 4: true_type::value is true

std::enable_if

- Also declared in type_traits

- Syntax:

```
std::enable_if<bool test, typename T = void>::type
```

- If test is false, ::type is invalid (SFINAE)
- If test is true, ::type is a typedef equivalent to T

Basic superPrint

- We have a version of superPrint that we want to enable in the case that the type does not have an iterator.

- So for the "test" we can use has_iterator.
- For "type" superPrint won't return anything so we won't pass in a type:

```
template <typename T>
typename std::enable_if<!has_iterator<T>::value>::type
superPrint(const T& t)
{
    std::cout << t << std::endl;
}
```

Basic superPrint, cont'd

```
template <T>
typename std::enable_if<!has_iterator<T>::value>::type
superPrint(const T& t)
{
   std::cout << t << std::endl;
}

// Test 1
superPrint(5);
```

Basic superPrint, cont'd

```
template <int>
typename std::enable_if<!has_iterator<int>::value>::type
superPrint(const int& t)
{
   std::cout << t << std::endl;
}
```

Step 1: T is replaced with int, since 5 is an int

```
// Test 1
superPrint(5);
```

Basic superPrint, cont'd

```
template <int>
typename std::enable_if<!has_iterator<int>::value>::type
superPrint(const int& t)
{
   std::cout << t << std::endl;
}

// Test 1
superPrint(5);
```

Step 1: T is replaced with int, since 5 is an int

Step 2: value == false, so the enable_if expression is true
… This means substitution SUCCEEDS
(type is void)

Basic superPrint, cont'd

```
template <int>
typename std::enable_if<!has_iterator<int>::value>::type
superPrint(const int& t)
{
   std::cout << t << std::endl;
}

// Test 1
superPrint(5);
```

Step 1: T is replaced with int, since 5 is an int

Step 2: value == false, so the enable_if expression is true

Step 3: This version of superPrint will be called at runtime and outputs 5.

superPrint for types that have iterators

```cpp
template <typename T>
typename std::enable_if<has_iterator<T>::value>::type
superPrint(const T& t)
{
   bool printComma = false;
   for (auto i : t)
   {
      if (!printComma)
      {
         printComma = true;
      }
      else
      {
         std::cout << ',';
      }
      std::cout << i;
   }
   std::cout << std::endl;
}
```

If we ignore the bodies…

```
// Version 1 (no iterator)
template <typename T>
typename
std::enable_if<!has_iterator<T>::value>::type
superPrint(const T& t)

// Version 2 (with iterator)
template <typename T>
typename
std::enable_if<has_iterator<T>::value>::type
superPrint(const T& t)
```

superPrint, Test 2

```
// Version 1 (no iterator)
template <typename T>
typename std::enable_if<!has_iterator<T>::value>::type
superPrint(const T& t)

// Version 2 (with iterator)
template <typename T>
typename std::enable_if<has_iterator<T>::value>::type
superPrint(const T& t)
```

```
// Test 2
std::list<int> myList{1, 2, 3};
superPrint(myList);
```

superPrint, Test 2

```
// Version 1 (no iterator)
template <typename T>
typename std::enable_if<!has_iterator<T>::value>::type
superPrint(const T& t)

// Version 2 (with iterator)
template <typename T>
typename std::enable_if<has_iterator<T>::value>::type
superPrint(const T& t)
```

```
// Test 2
std::list<int> myList{1, 2, 3};
superPrint(myList);
```

Step 1: Try to replace T with std::list<int> in version 1

... SFINAE

superPrint, Test 2

```
// Version 1 (no iterator)
template <typename T>
typename std::enable_if<!has_iterator<T>::value>::type
superPrint(const T& t)

// Version 2 (with iterator)
template <typename T>
typename std::enable_if<has_iterator<T>::value>::type
superPrint(const T& t)
```

```
// Test 2
std::list<int> myList{1, 2, 3};
superPrint(myList);
```

Step 1: Version 1 failed

Step 2: Try to replace T with std::list<int> in version 2
...SUCCESS

superPrint, Test 2

```
// Version 1 (no iterator)
template <typename T>
typename std::enable_if<!has_iterator<T>::value>::type
superPrint(const T& t)

// Version 2 (with iterator)
template <typename T>
typename std::enable_if<has_iterator<T>::value>::type
superPrint(const T& t)
```

```
// Test 2
std::list<int> myList{1, 2, 3};
superPrint(myList);
```

Step 1: Version 1 failed

Step 2: Version 2 succeeded!

Step 3: At runtime, the second version will be called, outputting:
1, 2, 3