Printout

Thursday, May 7, 2015 6:10 PM

EE380L.5 Spring 2015

Midterm

NAME: Solution

1. (20 pts) For this question, you can draw upon your understanding of either std::vector<T> or our own epl::vector<T>. Recall that vector<T> has a push_back function that allows us to append a new value to the end of an existing vector. I want you to write a new push_back function that allows me to append an entire data structure to the vector. A large part of the grading for this question is how you choose to design the function (what the parameters are, the return value, etc. etc.), so I'll leave those very important details up to you. Here are the principles I want you to achieve.

  a. I should be able to append the elements from any type of data structure. For example, if I have a vector<T> x and I want to append all the elements from a linked_list<T> y, I should be able to use your push_back function to do that.
  b. You should follow the general conventions of the C++ standard library (i.e,, use the STL conventions to the extent that it makes sense for this problem).
  c. Your push_back should be a member function for class vector<T>
  d. You can call any function from vector<T> that you wish (including other versions of push_back).

means the params must be iterators.

```
template <typename T>   // for Vector<T>
template <typename It>  // member template

void Vector<T>::push_back (It b, It e) {
    while (b != e) {
        this-> push_back (*b);
        ++ b;
    }
}
```

That's it

The idea here is to create an "iterator adapter" – a new class that modifies the existing iterator

2. (10 pts) When you wrote Project 1 phase C, you wrote a constructor that accepted an iterator pair. I could use that constructor to create a new vector x from an existing data structure y as follows:

    vector<T> x{ begin(y), end(y) };

When invoked in that manner, your constructor would copy-construct values from data structure y into the new vector x. Consider the challenge of designing a solution that would permit us to optionally move-construct the values from y into x.

    vector<T> x{ mbegin(y), mend(y) };

When invoked in this manner, the elements from y should be moved into x. Write template functions mbegin and mend. Include whatever structs or classes you need to implement mbegin and mend. Your design should not involve making any changes to vector<T> (i.e,. don't write any new constructors or new methods for vector).

```
template <typename It>
class madapt : public It {
    using T = typename iterator_traits<It>::value_type;
    T&& operator*(void) const {
        It const & super = *this;
        return std::move(*super);
    }
    madapt (It const& base) : It(base) {}
};

template <typename T>
auto mbegin (T& ds) -> madapt<decltype(begin(ds))> {
    using It = decltype(begin(ds));
    return madapt<It> { begin(ds) };
}
```

mend is same as mbegin except for calling end here

Need specialization
easiest to use bool true/false sizeof(T) < 10
to specialize on

3. (10 pts) I'm working on a generic algorithm, and I expect to have to move objects frequently (e.g., a sorting algorithm that swaps elements). I'm worried that some larger objects will be inefficient to move. For this problem, assume I have an object of type T. I want to create a struct that can store objects of type T – i.e., a storage container Cell<T>. I've started the solution by writing two Cell templates, SmallCell and LargeCell shown on the next page

Create a template type alias for me with the following properties

a. The template type alias should be called Cell and should have one template argument: i.e, write Cell<T>
b. If sizeof(T) is less than 10, then Cell<T> should be SmallCell<T>
c. If sizeof(T) is greater than or equal to 10, then Cell<T> should be LargeCell<T>

Recall a template type alias can be created with the following syntax. Write the Cell template alias and include any other structs or functions you need in your solution.

```
template <typename T>
using new_alias = old_temmplate<T>; // creates new_alias<T>
```

```
template<bool P, typename T> struct Cell_Select {
    using type = LargeCell<T>;
};
template<typename T> struct cell_select<true, T> {
    using type = SmallCell<T>;
};

template<typename T>
using Cell = typename Cell_select<(sizeof(T)<10), T>::type;
```

4. (15 pts) The implementations below work well enough that I can create a vector<SmallCell<int>> and a vector<LargeCell<int>> and sort them with std::sort. The only problem is that my implementation of LargeCell has a nasty memory leak. Rewrite LargeCell so that there are no memory leaks. Do not change the basic design – recall that LargeCell is used to avoid copying objects of type T, hence the T object is on the heap.

```
template <typename T> struct SmallCell {
	T data;
	SmallCell(T const& d) : data(d) {}
	operator T& (void) { return data; }
};

template <typename T> struct LargeCell {
	T* data;
	LargeCell(T const& d) : data(new T{d}) {}
	operator T& (void) { return *data; }
};

int main(void) {
	std::vector<LargeCell<int>> v; // SmallCell works too
	v = { 8, 7, 2, 1, 5, 6, 3, 9, 4 };
	v[4] = 42; // assignment works as if v is a vector of int

	std::sort(begin(v), end(v)); // correctly sorts v
}
```

The key here is that we need not only a <u>destructor</u> but also a <u>move</u> constructor. Copy constructor would make LargeCell too slow.

Write your corrected LargeCell template below

```
/* don't add any data or change any of the code below, you can add any functions you wish */
template <typename T> struct LargeCell {
	T* data;
	LargeCell(T const& d) : data(new T{d}) {}
	operator T& (void) { return *data; }
~LargeCell(void) { delete data; }
LargeCell(LargeCell<T>&& that) {
	data = that.data;
	that.data = nullptr;
}
LargeCell<T>& operator=(LargeCell<T>&& that) {
	T* t = data;
	data = that.data;
	that.data = t;
	return *this;
}
```

5. (10 pts) I've decided that the C++ assignment operator really doesn't make sense anymore, and am changing my programming style to not write "x = y" anymore. Instead, if I have a variable, x, and I want to give that variable a new value, I "reboot" the variable, constructing a new value for the variable in place. Write me a generic reboot function. The function should have the following characteristics

a. The first argument to reboot will be a variable of type T&. This argument will be a reference to the variable that is being rebooted (e.g., x from my example above)
b. reboot can have zero or more additional arguments. If there are additional arguments, then all arguments are passed to the constructor T::T and used to re-initialize the object begin rebooted.
c. Before re-initializing x, the reboot function must invoke the destructor for x.

For your convenience, the construct method below can be used to run the constructor for an object of type T at address p using zero or more arguments args. You can call construct in your reboot function, or you can use similar syntax to construct your object directly.

```
template <typename T, typename... Args>
void construct(T* p, Args... args) { new (p) T{args...}; }
```

```
template <typename T, typename... Args>
void reboot (T& x, Args... args) {
    new (&x) T{args...};
}
```

6. (25 pts total) In C++, output to a stream can be defined for any type. By default, however, types are not printable. For example the empty struct Foo {}; is not printable – cout << Foo{}; fails with an error message. I want you to create a "protected stream wrapper". The wrapper is a class called pstream that can be constructed from an ostream (write one constructor that accepts a parameter of type ostream&). The pstream class has one function, an operator<< that takes an arbitrary parameter of type T. If type T can be printed, then the pstream::operator<< function prints its argument to the ostream. If type T is a type that cannot be printed, then pstream::operator<< prints the text "<UNPRINTABLE>". See the example below.

```
struct Foo {};
int main(void) {
      pstream pout{cout};
      pout << "Hello World\n"; // prints hello world
      pout << 42 << "\n"; // prints 42
      pout << Foo{}; // prints <UNPRINTABLE>
}
```

Substitution failure

a. (10 pts) To complete this solution, you must first write a function isPrintable<T>() that returns a constexpr bool. isPrintable<T>() will return true if the expression cout << T is legal (i.e. if T can be printed), and will return false if the expression cout << T is a compile-time error (i.e., T cannot be printed).

```
template <typename T, typename R= decltype(cout << declval<T>())>
constexpr bool printable (int x) { return true; }

template <typename T>
constexpr bool printable (...) { return false; }

template <typename T>
constexpr bool isPrintable (void) {
   return printable<T>(42);
}
```

pstream is not a template class (see main) so it must have a member template

Question 6, continued.

b. (15 pts) Using the function isPrintable<T>() design and implement the pstream template class

```
template <bool p, typename T> struct printer {
    static void print (ostream& out, T const & x) {
        out << "<UNPRINTABLE>";
    }
};
template <typename T> struct printer<true, T> {
    static void print(ostream& out, T const& x){
        out << x;
    }
};
class pstream {
    ostream & out;
public:
    pstream (ostream & o) : out {o}  {}
    template <typename T>
    pstream& operator<< (T const & x) {
        printer<T>:: print (out , x);
        return * this;
    }
};
```

7. (10 pts) In Project 1, Phase B we added move semantics to our vector data structure. As a result, we'd hoped that our push_back method would be faster. Sadly, when I tested my solution, it didn't seem to be any faster. For reference, I've included the important loop below. I tested my program using vector<int> (i.e., T is **int**) using versions of the loop with support for move semantics (like the one below) and versions of the loop that were identical except did not use std::move. I did not see any performance difference resulting from std::move. In my experiments I used **int** for T, what type should I have used?

Explain, or better yet… give me an example type that I can use for T – make sure your type is "realistic", don't make it do anything silly just to force a performance advantage.

```
template <typename T>
T* expand(T* data, uint64_t old_size, uint64_t new_size) {
        T* ndata = (T*) operator new(new_size * sizeof(T));
        for (uint64_t k = 0; k < old_size; k += 1) {
                new (ndata + k) T{std::move(data[k])};
                data[k].~T();
        }
        operator delete(data);
        return ndata;
}
```

The problem is that copy + move are equivalent for int
To see a performance advantage, we need to test with an element type that is much slower to copy than it is to move. There are many good examples

e.g. std::string