

Homework Four Solutions

(1)

```
template <class T, class FilterObj>
void filter(list<T>& l, FilterObj f) {
    list<T>::iterator l_i = l.begin(); //to move through list
    list<T>::iterator del; // point to item to delete
    while (l_i != l.end()) {
        del = l_i++; //del lags behind
        if (f(*del)) l.erase(del); //remove if predicate true
    }
}
```

(2) Instead of giving the specific answers, here's the working code for the problem. The answers for (a) and (b) are marked with comments. For part (c), the key is that you are able to convert an ItemBase pointer to a Item<T> pointer, and then that you can extract the data from the Item<T> object. The conversion is most easily done with a dynamic cast. The only decision you have is how to deal with the situation where the type you are asking for is not the right type; here, I throw an exception. The extraction from an Item<T> object requires that the Item<T> class has an accessor. This accessor cannot be a function of ItemBase, because the base class doesn't know the type of the objects being held. With these figured out you can write another templated function for the extraction. The only trick being that you have to recognize the limitations of the compiler in figuring out the type for T, and make sure that the type occurs in the parameter list. So, I used "void extract(T)" when I would rather have used "T extract()".

```
#include <iostream>
#include <string>
#include <list>
```

```
/* Working program for hw#4 question 2.
```

```
This program compiles and runs using the egcs-1.1.2 compiler under
linux. This version of the compiler doesn't require namespace
business.
```

```
*/
```

```
/*
```

```
class ItemBase
```

```
Used in polyList construction as base class for arbitrary
objects. An abstract class.
```

```
*/
```

```
class ItemBase {
public:
    virtual void print() = 0;
    virtual ~ItemBase() {}
};
```

```

/*
  class Item<T>

  Templated class used to hold arbitrary data objects.
  Simply holds a T object.  The only derived class of ItemBase.
  Implements the print() method, and also provides an accessor
  that is not available in the ItemBase class.
*/
template <class T>
class Item : public ItemBase {
public:
  Item () {}
  Item (const T& t) : info(t) {}
  Item (const Item<T>& i) : info(i.info) {}
  void print() { cout << info; }
  T getData() { return info; }
private:
  T info;
};

/*
  class polyList

  Heterogeneous list.  Implemented as STL list of ItemBase pointers.
  Use of pointers necessary to avoid loss of data when each Item<T> is
  cast to ItemBase.  Includes templated functions for inserting and
  extracting value from the end of the list.

*/
class polyList {
public:
  template <class T> void insert(const T&); //part b
  template <class T> void extract(T&);
  bool isEmpty();
  void print();
  ~polyList();
  class WrongType {};
private:
  list<ItemBase*> lst; //part a
};

/*
  polyList::insert<T>

  Templated function to allow insertions of arbitrary types into
  lst.  Creates an Item<T> object and pushes its pointer at the end of
  lst.

*/
template<class T>
void polyList::insert(const T& t) {
  lst.push_back(static_cast<ItemBase*>(new Item<T>(t)));
}

```

```

/*
polyList::extract<T>

Templatized function to allow extractions of values from end of
list. Gets last element in list, attempts to cast it to Item<T>
pointer, and if cast worked assigns value in Item<T> object to the
parameter t. Only removes data item from list if type matches T. If
the type is incorrect then an exception is thrown.

Note: this function would be nicer if it had the prototype
T extract(). But the compiler cannot infer the use of the template
function from this prototype, because there are no parameters.
*/
template<class T>
void polyList::extract(T& t) {
    ItemBase* tmpbase = lst.back(); //get last entry
    Item<T>* tmpitem = dynamic_cast<Item<T>*>(tmpbase);
    if (tmpitem != NULL) { //type matches
        lst.pop_back(); //remove last entry
        t = tmpitem->getData(); //extract data
    }
    else
        throw WrongType();
}

/*
polyList::isEmpty

Checks if STL list is empty.

*/
bool polyList::isEmpty() { return lst.empty(); }

/*
polyList::print

Calls print for all objects in list.
*/
void polyList::print() {
    list<ItemBase*>::iterator l_i = lst.begin();
    while (l_i != lst.end()) {
        (*l_i)->print(); cout << endl; l_i++;
    }
}

```

```

/*
  polyList::~~polyList

  Destructor for polyList class. Iterates through list to delete
  objects. Necessary because list holds pointers. Requires
  virtual destructor in ItemBase.
*/
polyList::~~polyList() {
  list<ItemBase*>::iterator l_i = lst.begin();
  ItemBase* d;
  while (l_i != lst.end()) {
    d = *l_i++;
    delete d;
  }
}

void main() {
  polyList l;
  l.insert(1);
  l.insert(string("two")); //this is a difference from example code
  l.print();
  string s;
  try {
    if (!l.isEmpty()) l.extract(s);
  }
  catch (polyList::WrongType) {
    cout << "type error" << endl;
  }
  l.print();
  int i;
  try {
    if (!l.isEmpty()) l.extract(i);
    cout << "i: " << i << endl;
  }
  catch (polyList::WrongType) {
    cout << "type error" << endl;
  }
  l.print();
}

```