BaseLib2 Tutorial Series

Lists and Iterators

André Neto

June 13, 2011

Outline

What are lists good for?

- Containers of information
  - Simple types (floats, integers,...)
  - Complex types (structures, classes)
- Good for abstraction
  - A container of mammals will know how to store Humans and Dolphins
- The lists discussed here are quite low-level
  - Later you will use easier/smarter versions of these
  - Nevertheless these are the base classes for the more sophisticated versions

Static Lists

- Container of elements with a given size
  - This size is given by the minimum number of bytes required to store the element , divided by the size of a pointer, both for the target architecture
    - e.g. 32 bit integers in a 32 bit architecture, the element size is 1
    - e.g. 64 bit integers in a 32 bit architecture, the element size is 2
- Most important operations are: **add**, **extract**, **peek**, **delete** and **find**
  - extract returns the element and removes it from the list
- All elements must have the same size and, consequently, usually are from the same data type

StaticListHolder.h

- The father of all static lists is named StaticListHolder
- Multi-thread access can be protected with built-in semaphore

Most important functions are

```
uint32 ListSize();
bool ListAdd(const intptr *element,int position = SLH_EndOfList);
bool ListExtract(intptr *element=NULL,int position = SLH_EndOfList);
bool ListPeek(intptr *element=NULL,int position = SLH_EndOfList);
bool ListDelete(const intptr *element);
int ListFind(const intptr *element);
void SetAccessTimeout();
```

StaticListHolder example (BaseLib2/Documentation/Tutorials/examples/StaticListHolderExample.cpp)

Source

```
//32 bit floating point elements to insert in the list
float f1 = 1.23;
float f2 = 3210;
float f3 = -1.23;
//The constructor receives the elements size...  All elements must have the same size.
StaticListHolder slh(1);
//Add
slh.ListAdd((const intptr *) &f1);
slh.ListAdd((const intptr *) &f2);
//Remove the second element and copy the value to retrivedValue
float retrivedValue;
if(!slh.ListExtract((intptr *) &retrivedValue, 1)){
    CStaticAssertErrorCondition(FatalError, "Failed to retrieve the second element on the
list!");
    return -1;
}
//Print the value (which I already now it was a float)
CStaticAssertErrorCondition(Information, "The value of the second element in the list is %f",
retrivedValue);
CStaticAssertErrorCondition(Information, "My list holder size is:  %d", slh.ListSize());
//Search for an element int32 elementPos = slh.ListFind((const intptr *)&f3);
CStaticAssertErrorCondition(Information, "Value %f is in position:  %d", f3, elementPos);
```

Other static lists

- The StaticQueueHolder and the StaticStackHolder are just specialisations/renames of StaticListHolder to ease the implementations of queues and stacks
- The StaticListTemplate allows the direct usage of templates in a StaticListHolder

Most important functions StaticQueueHolder

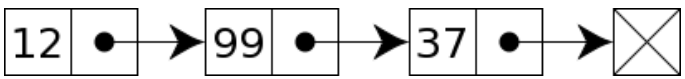
```
uint32 QueueSize();
void QueueAdd(const intptr *element);
bool QueueExtract(intptr *element);
bool QueuePeek(intptr *element, uint32 index);
```

Most important functions StaticStackHolder

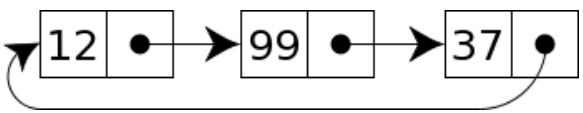
```
uint32 StackDepth();
void StackPush(const intptr *element);
bool StackPop(intptr *element);
bool StackPeek(intptr *element,int position);
```

Introduction

- Lists where each element, or node, contains a reference to the next element (in the singly case as in BaseLib2 implementation)
- Can be linear...



- ...or circular



- Doubly implementations where a reference to the previous node is also stored, are not available in BaseLib2
- Very easy and fast to add or remove a node
- Random access might require scanning all elements

Iterators and filters

- Helper objects to cycle through a given container
  - Replace the classic *while and for*...
- Greatly simplifies code and leverages on re-usage
- Filters automatically allow to sort and search for entries, without always repeating the same iterative code
- You're expected to implement a call-back with the action to be performed for each of the entries
  - **Do**, when iterating through all the elements
  - **Compare**, when sorting elements
  - **Test**, when filtering elements

LinkedListable.h

- Singly linked list
- Enables to insert and set the next node
- Extract and delete nodes
- Iterate, Sort and Filter for nodes
- Multi-thread access can be protected with built-in semaphore

Most important functions are

```
uint32 Size();
bool BSort(SortFilter *sorter);//other templates available
bool Insert(LinkedListable *p);//other templates available
bool Add(LinkedListable *p);//end of the queue
LinkedListable *Search(SearchFilter *filter);//other templates available
LinkedListable *Extract(LinkedListable *p);//other templates available
void Iterate(Iterator *it);//other templates available
```

Iterators.h

- Must create a class which inherits from one of these to provide the required functionality (template versions also available)

Iterate

```
class Iterator{
    virtual void Do (LinkedListable *data)=0;
```

Search

```
class SearchFilter{
    virtual bool Test (LinkedListable *data)=0;
```

Sort

```
class SortFilter{
    virtual int32 Compare(LinkedListable *data1,LinkedListable *data2)=0;
```

The simple Human
(BaseLib2/Documentation/Tutorials/examples/LinkedListableExample.cpp)

- LinkedListable is a Human defined by a name and an height

Human class source

```
class Human :  public LinkedListable{
    ...
    public:  Human(const char *n, float h){
        height = h;
        if(n != NULL){
            name = (char *)malloc(strlen(n) + 1);
            strcpy(name, n);
        }
        else{
            name = NULL;
        }
    }
    virtual ~Human(){
        if(name != NULL){
            free((void *&)name);
        }
    }
    const char *Name(){
        return name;
    }
    float Height(){
        return height;
    }
};
```

Human iterator
(BaseLib2/Documentation/Tutorials/examples/LinkedListableExample.cpp)

- This function will be called for every element in the list (to the right of the calling node)
- For each Human, print the name and the height

Iterator class source

```
class HumanIterator :  public Iterator{
    virtual void Do(LinkedListable *data){
        Human *h = dynamic_cast<Human *>(data);
        if(h == NULL){
            return;
        }
        CStaticAssertErrorCondition(Information, "My name is %s and my height is %f", h->Name(),
h->Height());
    }
};
```

Human sorter
(BaseLib2/Documentation/Tutorials/examples/LinkedListableExample.cpp)

- Compare two LinkedListables and return a value less than, equal to, or greater than zero if data1 is found, respectively, to be less than, to match, or be greater than data2
- Order humans by name
  - Notice the direct usage of strcmp

Sorter class source

```
class HumanSorter :  public SortFilter{
    virtual int32 Compare(LinkedListable *data1,LinkedListable *data2){
        Human *h1 = dynamic_cast<Human *>(data1);
        Human *h2 = dynamic_cast<Human *>(data2);
        if((h1 == NULL) || (h2 == NULL)){
            return 0;
        }
        return strcmp(h1->Name(), h2->Name());
    }
};
```

Human search
(BaseLib2/Documentation/Tutorials/examples/LinkedListableExample.cpp)

- Returns True if the test data satisfies the search condition
  - Search for humans whose name start by a configurable character

Search class source

```
class HumanSearchFilter :  public SearchFilter{
private:
    char firstChar;
public:
    HumanSearchFilter(char fc){
        firstChar = fc;
    }
    bool Test(LinkedListable *data){
        Human *h = dynamic_cast<Human *>(data);
        if(h == NULL){
            return False;
        }
        if(h->Name() == NULL){
            return False;
        }
        return h->Name()[0] == firstChar;
    }
};
```

Node creation
(BaseLib2/Documentation/Tutorials/examples/LinkedListableExample.cpp)

Create the nodes

```
...
//Create a root node
LinkedListable root;
//Create some humans and link them
Human tim("Tim", 1.75);
root.Insert(&tim);
Human tom("Tom", 1.85);
tim.Insert(&tom);
Human john("John", 1.66);
tom.Insert(&john);
Human sue("Sue", 1.78);
john.Insert(&sue);
Human sam("Sam", 1.88);
sue.Insert(&sam);
...
```

Classic iteration vs iterator
(BaseLib2/Documentation/Tutorials/examples/LinkedListableExample.cpp)

Classic iteration

```
//Cycle through all the humans and print their name and height
LinkedListable *list = &root;
int32 i=0;
while(list != NULL){
    Human *h = dynamic_cast<Human *>(list);
    if(h != NULL){
        CStaticAssertErrorCondition(Information, "[%d]:My name is %s and my height is %f", i,
h->Name(), h->Height());
    }
    list = list->Next();
    i++;
}
```

Using the iterator

```
...
//Repeat the same exercise but using an iterator
//reset the list to the starting point
list = &root;
//Create the iterator and iterate
HumanIterator hi;
CStaticAssertErrorCondition(Information, "Printing from iterator");
list->Iterate(&hi);
```

Sorting and searching
(BaseLib2/Documentation/Tutorials/examples/LinkedListableExample.cpp)

Sorting

```
//Sort by name
//reset the list to the starting point
list = &root;
//Create the sorter
HumanSorter sorter;
list->BSort(&sorter);
```

Searching

```
//Create a search filter for all the names starting with a T
HumanSearchFilter humanSearchT('T');
CStaticAssertErrorCondition(Information, "Printing only names starting with a T");
//reset the list to the starting point
list = &root;
LinkedListable *found = list;
while(found != NULL && list != NULL){
    found = list->Search(&humanSearchT);
    if(found != NULL){
        Human *h = dynamic_cast<Human *>(found);
        CStaticAssertErrorCondition(Information, "My name is %s and my height is %f", h->Name(),
h->Height());
        list = found->Next();
    }
}
```

Training ideas

1. Replicate the StaticListHolderExample but using the StaticListTemplate
2. Add the possibility of sorting by height to the LinkedListableExample