STL List

The Standard Template Library (STL) has several container object like list , vector. List containers are implemented as doubly-linked lists; The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

Example -8: Exploring the list container of STL (ListHandler.cpp)

- **Create two list of numbers and print them using iterator**
- **Transfers elements from one list to another using splice operation.**
- **Sort and merge the lists**

ListHandler.h

```
#ifndef LISTHANDLER_H
#define LISTHANDLER_H

#include <iostream>        #include <list>
#include <algorithm>       #include <iterator>

class ListHandler {
public:
   ListHandler();
   ListHandler(const ListHandler& orig);
   virtual ~ListHandler();
   void procList();
private:
   void printLists (const list<int>& l1, const list<int>& l2);
};
#endif /* LISTHANDLER_H */
```

Unable to resolve identifier error can be resolved

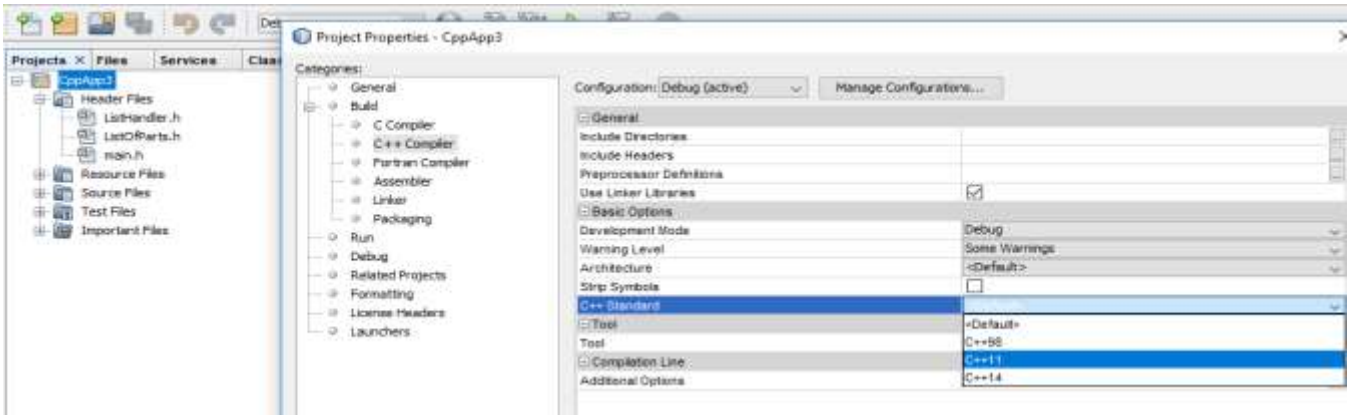
```
class ListHandler {
public:
    ListHandler();
    ListHandler(const ListHandler& orig);
    virtual ~ListHandler();
    void procList();
private:
    void printLists ( const std::list<int>& l1, const std::list<int>& l2);
};
```

Unable to resolve identifier list.

(Alt-Enter shows hints)

using namespace std;
should not be included in .h file

- To resolve the unable to resolve identifier error select project property and set C++Standard = C++11 and C Standard = C11



ListHandler.cpp

```
#include "ListHandler.h"

ListHandler::ListHandler() {   }
ListHandler::ListHandler(const ListHandler& orig) {   }
ListHandler::~ListHandler() {    }

//  Print two List using iterator
void ListHandler::printLists (const list<int>& l1, const list<int>& l2)
{
   //  An Iterator is an object that can traverse (iterate over) a container class without the user
       having to know how the container is implemented. Iterators are the primary way elements of
       container objects of STL (list, vector) are accessed.

   //  Iterators are generated by STL container member functions, such as begin() and end().
       • begin() returns an iterator representing the beginning of the elements in the container.
       • end() returns an iterator representing the element just past the end of the elements.

   //  An iterator is best visualized as a pointer to a given element in the container, with a set of
       overloaded operators to provide a set of well-defined functions:
       • Operator* - Dereferencing the iterator returns the element that the iterator is currently
                     pointing at.
       • Operator++ - Moves the iterator to the next element in the container.
                      Most iterators also provide Operator- to move to the previous element.
       • Operator== and Operator!=  - Basic comparison operators to determine if two iterators point
                      to the same element.
       • Operator=  - Assign the iterator to a new position (typically the start or end of the container's
                      elements).
//     To assign the value of the element the iterator is point at, deference the iterator first, then use
       the assign operator.

//     Ostream iterators are output iterators that write sequentially to an output stream (such as cout).
//     std::ostream_iterator is a single-pass OutputIterator that writes successive objects of type T
       into the std::basic_ostream object for which it was constructed, using operator<<.
       Optional delimiter string is written to the output stream after every write operation.

   //  copy(v.begin(), v.end(), l.begin());

   //  copy() takes three arguments, all iterators:
       ▪  an iterator pointing to the first location to copy from
       ▪  an iterator pointing one element past to the last location to copy from
       ▪  an iterator pointing to the first location to copy into
```

```
    cout << "list1: ";

    //  Use std::copy algorithm to copy a portion of list to cout using ostream_iterator iterator
    //  It will copy the portion of list to cout each element will be delimited by ' '
    copy (l1.begin(), l1.end(), ostream_iterator<int>(cout," "));
    cout << endl << "list2: ";
    copy (l2.begin(), l2.end(), ostream_iterator<int>(cout," "));
    cout << endl << endl;
}

//  Examine different features of List collection
void ListHandler::procList()
{
    // create two empty lists
    list<int> list1, list2;

    // fill both lists with elements
    for (int i=0; i<6; ++i) {
        list1.push_back(i);                 list2.push_front(i);
    }
    printLists(list1, list2);    //   Output   list1: 0 1 2 3 4 5          list2: 5 4 3 2 1 0

    //  Transfers elements from one list to another using splice operation.
    //  No elements are copied or moved, only the internal pointers of the list nodes are re-pointed.

    //  All element from list1 will be transfered before the first element with value 3 of list2
    //  find() returns an iterator to the first element with value 3 in list2

    list2.splice(find(list2.begin(),list2.end(), 3), list1);                    // source list
    printLists(list1, list2);    //  Output   list1:     list2: 5 4 0 1 2 3 4 5 3 2 1 0

    // Move first element of list2 to the end
    list2.splice(list2.end(),        // destination position
                 list2,                  // source list
                 list2.begin());        // source position
    printLists(list1, list2);       //  Output   list1:     list2: 4 0 1 2 3 4 5 3 2 1 0 5

    // Sort second list, assign to list1 and remove duplicates
    list2.sort();
    list1 = list2;
    list2.unique();
    printLists(list1, list2);    //  Output   list1: 0 0 1 1 2 2 3 3 4 4 5 5     list2: 0 1 2 3 4 5

    // Merge both sorted lists into the first list
```

push_back() - Adds a new element at the end of the list container, after its current last element. The content of val is copied (or moved) to the new element.

```
   // This function requires that the list containers have their elements already ordered by value
   // before the call. list1 and list2 already sorted

   list1.merge(list2);
   printLists(list1, list2);   //  Output list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5

}
```

Example -9: Create a Link List of Car Parts using pointer (ListOfParts.cpp)

ListOfParts.h

```
#ifndef LISTOFPARTS_H
#define LISTOFPARTS_H

#include <iostream>
#include <string.h>

using namespace std;

struct CarPart
{
   long     PartNumber;
   char     PartName[40];
   double   UnitPrice;
   CarPart* Next;
};

//  This class would be responsible for all operations that can be performed on the list. When a list
    starts, it is empty. To specify this, we should declare a primary member variable, usually called
    Head. Although this member is declared as a pointer and it marks the beginning of the list, you
    should not allocate memory for it in the constructor.  ts memory would be managed when it is
     accessed. Therefore, you can simply initialize it as NULL.

class ListOfParts {
public:
   ListOfParts();
   ListOfParts(const ListOfParts& orig);
   virtual ~ListOfParts();
   int Count();
   CarPart* Head;
// List Operations
public:
   int Add(CarPart* Item);
   CarPart *Retrieve(int pos);
```

```
   bool Delete();
   bool Delete(int pos);
   bool Find(CarPart* Item);
private:
   int size;
};

#endif /* LISTOFPARTS_H */
```

ListOfParts.cpp

```
#include "ListOfParts.h"

ListOfParts::ListOfParts() : size(0), Head(NULL) {}

ListOfParts::ListOfParts(const ListOfParts& orig) {}

ListOfParts::~ListOfParts() {}

int ListOfParts::Count() {    return size;  }
int ListOfParts::Add(CarPart *NewItem)
{
   CarPart *Sample = new CarPart;
   Sample = NewItem;
   Sample->Next    = Head;
   Head = Sample;
   return size++;
}
CarPart *ListOfParts::Retrieve(int Position)
{
   CarPart *Current = Head;
   for(int i = 0; i < Position && Current != NULL; i++)
   {
         Current = Current->Next;
   }
   return Current;
}
bool ListOfParts::Delete()
{
   if( Head == NULL )
   {
         std::cout << "The list is empty\n";
         return false;
   }
   else
```

```
   {
         CarPart *Current;
         Current = Head->Next;
         Head->Next = Current->Next;
         size--;
         return true;
   }
}
bool ListOfParts::Delete(int Position)
{
   if( Retrieve(Position) == NULL )
      return false;
   else
   {
         Retrieve(Position - 1)->Next = Retrieve(Position+1);
         size--;
         return true;
   }
}
bool ListOfParts::Find(CarPart* ItemToFind)
{
   CarPart *Current;
   if( ItemToFind == NULL )    return false;
   for(Current = Head; Current != NULL; Current = Current->Next)
   {
         if( (Current->PartNumber == ItemToFind->PartNumber) &&
         (strcmp(Current->PartName, ItemToFind->PartName) == 0) &&
                  (Current->UnitPrice == ItemToFind->UnitPrice) )
         return true;
         else
         Current->Next;
   }
   return false;
}
```