Contents

1 The `List<T,std::size_t>` Class

The `List<T,std::size_t>` class is an object that may be heap allocated or stack allocated. Its intended to be used with template classes where the class will determine if the list should allocated its contents on the heap or on the stack. The use of `List<T,std::size_t>` must know at compile time which allocation method will be used.

1.1 The Basic Function of `List<T,std::size_t>`

Example 1: The `List<T,std::size_t>` Class

```
1
2  List<int, 0> heapList;   //Allocated on the heap, auto resizes.
3  List<int, 30> stackList; //Allocated on the stack, will not resize.
4  template<bool UseStack>
5  class Points
6  {
7  public:
8    /* ... Member functions ...*/
9  private:
10   cg::List<int, UseStack ? 100 : 0> m_points;
11 };
12 ...
13 /*Reguardless of the location of the data, the list is used in
       exactly the same way, with exactly the same code.*/
14 Points<false> m_slowPoints; //Slow heap based data
15 Points<true> m_fastPoints;  //Fast stack data
16 ...
```

The `List<T,std::size_t>` Class, when stack allocated, will keep track of how many units of its array are used. It works exactly the same way as a heap list except that it has a max capacity, which is its `std::size_t` parameter. When the max capacity is hit, it will throw a `std::runtime_error`. See the following section on how to use the `List<T,std::size_t>` interchangeably with stack and heap mode.

Why?

A developer may wish to have a single class that can be used for both stack based calculation and heap based calculations. Two classes could have been made, one for the stack, and one for the heap, but why write overhead twice?

1.2 Using List<T,std::size_t>

Member functions are provided to access the list in a way that it mey be stack or heap based without changing any operating code except for the template parameter where 0 means heap allocated and !0 means stack allocated of that size..

1.3 The Size of List<T,std::size_t>

To determine the size or max size of the list, use the following members (Example 6 on the following page).

Example 2: Determining the Size of a list.

```
1 cg::List<int, 0> heapList;   //Allocated on the heap, auto resizes.
2 cg::List<int, 30> stackList; //Allocated on the stack, will not
      resize.
3 auto hlSize = heapList.Size(); //Get the amount of elements in
      heapList.
4 auto slSize = stackList.Size();//Geth the amount of elements in the
       stackList.  May be  the size of the template paramter or less.
5 auto hlMax = heapList.MaxSize(); //Will be the max size of std::
      size_t.
6 auto slMax = stackList.MaxSize();// will be the template parameter
      (30 in this example).
```

Its important to note that the `MaxSize()` of a stack list will always be the value of the template parameter and that `Size()` need not be exactly `MaxSize()` but it may be. Consider the following Example **??** on page ??.

Example 3: Size() and MaxSize() of stackList

```
1
2 cg::List<int, 30> stackList; //Allocated on the stack, will not
      resize.
3 auto slSize = stackList.Size();//Geth the amount of elements in the
       stackList.  May be  the size of the template paramter or less.
4 auto slMax = stackList.MaxSize();// will be the template parameter
      (30 in this example).
5 int i = 0;
6 while(i < 25)
7   stackList.PushBack(i++);
8 slSize = stackList.Size(); //slSize is now 25.
9 slMax = stackList.MaxSize();// slMax is still 30.
```

1.4 Accessing the `List<T,std::size_t>`

There are multiple ways to access the list. An iterator (raw ptr), `operator[]`, and `Get()` functions.

Example 4: Accessing via iterators

```
cg::List<int, 30> stackList; //Allocated on the stack, will not
    resize.
int i = 0;
while (i < 25)
  stackList.PushBack(i++); //add 25 items to the list.
auto beg = stackList.Begin();
auto end = stackList.End();  //The end poitner is one past the end
    of the size, and not the max sie.
for (; beg != end; ++beg)    //prints 0, 1, 2, ... , 23, 24
  std::cout << *beg << ", ";
```

One might thing that the `End()` function might point to the end of the allocated space (index 31 with the previous example) but thats not the case. The `End()` function actually returns a pointer to one-past-the-end of the last element actually *placed* into the list (for the previous example, `Begin()` + `Size()` is the effective pointer). There is also a `operator[]`, and `Get()` function.

Example 5: Accessing via Get() and square bracket operator

```
cg::List<int, 30> stackList; //Allocated on the stack, will not
    resize.
int i = 0;
while (i < 25)
  stackList.PushBack(i++); //add 25 items to the list.
for (std::size_t i = 0; i < stackList.Size(); ++i)
{
  std::cout << stackList[i] << ", ";
  //std::cout << stackList.Get(i) << ", ";   //also works the same.
}
```

1.5 Altering the `List<T,std::size_t>`

There are a few ways to erase elements off the list.

Example 6: Erasing items

```
1  cg::List<int, 30> stackList; //Allocated on the stack, will not
       resize.
2  int i = 0;
3  while (i < 25)
4    stackList.PushBack(i++); //add 25 items to the list.
5  stackList.PopBack();     //Pop off the back item
6  stackList.PopFront();    //Pop off the front item
7  stackList.Erase(3);      //Pop off the item at index 3
8  stackList.Erase(5,2);    //Pop off 2 items starting at and including
       index 5
9  stackList.Pop(7);        //Pop off the item at index 7
10 stackList.Pop(7,6);      //Pop off 6 items starting at and including
       index 7
11 for (std::size_t i = 0; i < stackList.Size(); ++i)
12 {
13   std::cout << stackList[i] << ", ";
14 }
```

Note that there are functions `Pop(index, amt)` and `Erase(index, amt)` that do the same thing.