Subject: Quick bi-array
Posted by Mindtraveller on Wed, 21 Jan 2009 13:38:36 GMT
View Forum Message <> Reply to Message

I need a deque-like container where I may reserve space for a number of elements, and the main requirement is that removing element doesn`t make actual memory deallocation, as well as adding element by reference doesn`t actually make any memory allocation or constructor call. I want add/delete mechanism to be as quick as possible: all the actions are to be done within reserved set of elements. Adding is just calling operator= to internal reserved container element, removing is just marking it unused. Something like that.
Is there any appropriate container in U++?

Looking into sources, it looks like BiVector and BiArray use new/delete and don`t match requirements.

P.S. Sorry, it`s really a U++ Core topic.

Subject: Re: Quick bi-array
Posted by mirek on Wed, 21 Jan 2009 13:53:55 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Wed, 21 January 2009 08:38I need a deque-like container where I may reserve space for a number of elements, and the main requirement is that removing element doesn`t make actual memory deallocation, as well as adding element by reference doesn`t actually make any memory allocation or constructor call. I want add/delete mechanism to be as quick as possible: all the actions are to be done within reserved set of elements. Adding is just calling operator= to internal reserved container element, removing is just marking it unused. Something like that.
Is there any appropriate container in U++?

Looking into sources, it looks like BiVector and BiArray use new/delete and don`t match requirements.

BiVector would call only single new at Reserve, then nothing else.

BiArray behaves as Array, of course (each element is newed/deleted).

It is not easy to decipher your requirements, but if you mean by 'reference' what I understand, I guess something like

BiVector<Element *>

should be fine and would never call new/delete after Reserve, as long as you do not exceed the reserved size.

Would you be more specific, I would try to find more detailed answer.

Mirek

Subject: Re: Quick bi-array
Posted by Mindtraveller on Wed, 21 Jan 2009 13:59:47 GMT
View Forum Message <> Reply to Message

Thank you very much for quick answering. It looks like you have successfully deciphered my requirements and BiVector<Element *> is really what I need for now.

Subject: Re: Quick bi-array
Posted by Mindtraveller on Wed, 21 Jan 2009 17:55:36 GMT
View Forum Message <> Reply to Message

I`ve tried to test BiVector<Element *> and it looks like it was not the thing I wanted.
What did I want? I wanted a deque-like container which is extremely fast on adding and removing it`s records. This means no allocation/deallocation is accepted (just calling Element::operator= only). The idea of such a container is as quick as possible container access while program runs. Contrary, possible allocation/deallocation on program start/finish is acceptable.
So the thing I wanted is such code

```
 struct Element
 {
  Element()       :a(0)  {Cout()<<"*";}
  Element(int _a) :a(_a) {Cout()<<"+";}
  ~Element()             {Cout()<<".";}
  Element & operator= (const Element&op) {a=op.a; Cout()<<"="; return *this;}
  int a;
 };
 QuickDeque<Element> vec;
 static Element elm(0);
 for (int i=0;i<10;++i)
 {
  elm.a = i;
  vec.AddTail(elm);
 }
```

should give output:
Quote:==========
without any constructors/desctructors after program started.
And it looks like I managed to do something like it (just a second quick try after BiVector test):

```
template<class T> class QuickDeque
{
public:
 QuickDeque(int cap = 10) :capacity(0), data(NULL), start(0), count(0) {ASSERT(cap > 0);
AddAlloc(cap);}
 ~QuickDeque() {DeAlloc(); }
```

```
 void AddTail(const T&t) {if (count >= capacity) AddAlloc(count); int offs=start+count; if (offs >=
capacity) offs-=capacity; *((T *) &data[offs*sizeof(T)]) = t; ++count;}
 void DropHead(T &t) {ASSERT(count > 0); t = *((T *) &data[start*sizeof(T)]); if (++start ==
capacity) start=0; --count;}
 T &operator[] (int n) {ASSERT(n>=0 && n<count); int offs=start+n; if (offs >= capacity)
offs-=capacity; return *((T *) &data[offs*sizeof(T)]);}
 int GetCount() {return count;}

private:
 void AddAlloc(int capacityAdd)
 {
  ASSERT(capacityAdd >= 0);
  int capacityNew = capacity+capacityAdd;
  byte *newData = new byte[capacityNew*sizeof(T)];
  memcpy(&newData[0], &data[start*sizeof(T)], (capacity-start)*sizeof(T));
  memcpy(&newData[(capacity-start)*sizeof(T)], &data[0], start*sizeof(T));
  for (int i=count; i<capacityNew; ++i)
   new (&newData[i*sizeof(T)]) T();

  if (data)
   delete[] data;
  start    = 0;
  data     = newData;
  capacity = capacityNew;
 }

 void DeAlloc()
 {
  for (int i=0; i<capacity; ++i)
   ((T *) &data[i*sizeof(T)])->T::~T();
  if (data)
   delete[] data;
 }

 int   capacity;
 byte *data;
 int   start,
       count;
};
```

I wonder if it works with polymorphic elements...

Subject: Re: Quick bi-array
Posted by mirek on Wed, 21 Jan 2009 21:03:55 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Wed, 21 January 2009 12:55

I wonder if it works with polymorphic elements...

Nope.

Anyway, maybe you can look at BiArray::

```
T&      AddHead(T *newt)             { bv.AddHead(newt); return *newt; }
T&      AddTail(T *newt)             { bv.AddTail(newt); return *newt; }
template <class TT> TT& CreateHead()   { TT *q = new TT; bv.AddHead(q); return *q; }
template <class TT> TT& CreateTail()   { TT *q = new TT; bv.AddTail(q); return *q; }
T       *DetachHead()                { T *q = (T*) bv.Head(); bv.DropHead(); return q; }
T       *DetachTail()                { T *q = (T*) bv.Tail(); bv.DropTail(); return q; }
```

might provide the kind of operations you need.

Note that basically, polymorphy requires new... You can move that "outside", but it is hard to avoid it in generic case, because sizeof(T) varies.

Polypmorhy in general also excludes operator=.

Mirek
