Subject: Ptr improve
Posted by tojocky on Mon, 16 May 2011 12:07:58 GMT
View Forum Message <> Reply to Message

Hello Mirek,

I propose to improve a little Ptr class by add autodelete in PteBase::Prec and Autodelete in PteBase:

in h file:

```
template <class T> class Ptr;

class PteBase {
protected:
 struct Prec {
  PteBase *ptr;
  Atomic   n;
  bool autodelete;
 };

 volatile Prec  *prec;

 Prec          *PtrAdd();
 void Autodelete();
 static void     PtrRelease(Prec *prec);
 static Prec    *PtrAdd(const Uuid& uuid);
 static void Lock();
 static void Unlock();

 PteBase();
 ~PteBase();

 friend class PtrBase;
};

class PtrBase {
protected:
 PteBase::Prec *prec;
 void Set(PteBase *p);
 virtual void Release();
 void Assign(PteBase *p);
public:
 virtual ~PtrBase();
};

template <class T>
class Pte : public PteBase {
 friend class Ptr<T>;
```

```
};

template <class T>
class PteAuto : public PteBase {
 friend class Ptr<T>;
public:
 PteAuto(){PteBase::Autodelete();}
};

template <class T>
class Ptr : public PtrBase, Moveable< Ptr<T> > {
 T   *Get() const                    { return prec&&prec->ptr ? static_cast<T *>(prec->ptr) : NULL; }
protected:
 void Release();
 void Autodelete();
public:
 T       *operator->() const          { return Get(); }
 T       *operator~() const           { return Get(); }
 operator T*() const                  { return Get(); }

 Ptr& operator=(T *ptr)               { Assign(ptr); return *this; }
 Ptr& operator=(const Ptr& ptr)       { Assign(ptr.Get()); return *this; }

 Ptr()                              { prec = NULL; }
 Ptr(T *ptr)                        { Set(ptr); }
 Ptr(const Ptr& ptr)                { Set(ptr.Get()); }
 ~Ptr()                             { Autodelete(); }
 String ToString() const;

 friend bool operator==(const Ptr& a, const T *b)   { return a.Get() == b; }
 friend bool operator==(const T *a, const Ptr& b)   { return a == b.Get(); }
 friend bool operator==(const Ptr& a, const Ptr& b) { return a.prec == b.prec; }

 friend bool operator==(const Ptr& a, T *b)         { return a.Get() == b; }
 friend bool operator==(T *a, const Ptr& b)         { return a == b.Get(); }

 friend bool operator!=(const Ptr& a, const T *b)   { return a.Get() != b; }
 friend bool operator!=(const T *a, const Ptr& b)   { return a != b.Get(); }
 friend bool operator!=(const Ptr& a, const Ptr& b) { return a.prec != b.prec; }

 friend bool operator!=(const Ptr& a, T *b)         { return a.Get() != b; }
 friend bool operator!=(T *a, const Ptr& b)         { return a != b.Get(); }
};

template<class T>
void Ptr<T>::Release(){
 Autodelete();
 PtrBase::Release();
```

```
}

template<class T>
void Ptr<T>::Autodelete(){
 PteBase::Lock();
 if(prec&&prec->autodelete&&prec->ptr&&prec->n < 2){
  delete (T*)(prec->ptr);
  prec->ptr = NULL;
 }
 PteBase::Unlock();
}

template <class T>
String Ptr<T>::ToString() const{
 return prec&&prec->ptr ? FormatPtr(Get()) : String("0x0");
}
```

in cpp:

```
#include "Core.h"

NAMESPACE_UPP

static StaticCriticalSection sPteLock;

void PteBase::Lock(){
 sPteLock.Enter();
}

void PteBase::Unlock(){
 sPteLock.Leave();
}

PteBase::Prec *PteBase::PtrAdd(){
 sPteLock.Enter();
 if(prec) {
  ++prec->n;
  sPteLock.Leave();
 }
 else {
  sPteLock.Leave();
  prec = new Prec;
  prec->n = 1;
  prec->ptr = this;
  prec->autodelete = false;
 }
 return const_cast<Prec *>(prec);
}
```

```
void PteBase::PtrRelease(Prec *prec){
 CriticalSection::Lock __(sPteLock);
 if(prec && --prec->n == 0){
  if(prec->ptr){
   prec->ptr->prec = NULL;
  }
  delete prec;
  prec = NULL;
 }
}

void PteBase::Autodelete(){
 if(!prec){
  prec = new Prec;
  prec->n = 0;
  prec->ptr = this;
 }
 prec->autodelete = true;
}

PteBase::PteBase(){
 prec = NULL;
}

PteBase::~PteBase(){
 CriticalSection::Lock __(sPteLock);
 if(prec)
  prec->ptr = NULL;
}

void PtrBase::Release(){
 PteBase::PtrRelease(prec);
}

void PtrBase::Set(PteBase *p){
 prec = p ? p->PtrAdd() : NULL;
}

void PtrBase::Assign(PteBase *p){
 Release();
 Set(p);
}

PtrBase::~PtrBase(){
 Release();
}
```

```
END_UPP_NAMESPACE
```

and a little test:

```
#include <Core/Core.h>

using namespace Upp;

struct Foo : PteAuto<Foo> {
 String text;
};

Foo* factory(){
 return new Foo;
}

CONSOLE_APP_MAIN
{
 Ptr<Foo> ptr;
 {
  Ptr<Foo> ptr1 = new Foo;
  ptr1->text = "Text";
  ptr = ptr1;
  Cout() << (void*)~ptr << " -> " << ptr->text << "\n";
 }
 Cout() << "----------\n";
 Cout() << (void*)~ptr << "\n";
 ptr = factory();
 ptr = factory();
}
```

To use this PteAuto you need to know exactly that classes which use this class will automatically deleted.

Any comments are welcome!

PS: changed the realization to resolve with destructor of T.

Subject: Re: Ptr improve
Posted by mirek on Mon, 16 May 2011 13:05:55 GMT
View Forum Message <> Reply to Message

Quote:Any comments are welcome!

No. This is going against U++ principles.

Mirek

Subject: Re: Ptr improve
Posted by tojocky on Mon, 16 May 2011 13:31:51 GMT
View Forum Message <> Reply to Message

mirek wrote on Mon, 16 May 2011 16:05Quote:Any comments are welcome!

No. This is going against U++ principles.

Mirek

Exists situations when you do not know when to delete the created objects. In this case prec->n will counting and delete the unused objects. PteAuto can be use only with new constructor.

This can be as a future.

Or maybe exist something like?

Subject: Re: Ptr improve
Posted by tojocky on Mon, 16 May 2011 14:06:54 GMT
View Forum Message <> Reply to Message

mirek wrote on Mon, 16 May 2011 16:05Quote:Any comments are welcome!

No. This is going against U++ principles.

Mirek

Mirek,

As an alternative solution can you add a new virtual method OnPrecDelete() in class PteBase:

```
class PteBase {
protected:
 struct Prec {
  PteBase *ptr;
  Atomic   n;
 };

 virtual void OnPrecDelete() {};
```

```
 volatile Prec  *prec;

 Prec          *PtrAdd();
 static void    PtrRelease(Prec *prec);
 static Prec   *PtrAdd(const Uuid& uuid);

 PteBase();
 ~PteBase();

 friend class PtrBase;
};
```

and change in cpp file the method:

```
void PteBase::PtrRelease(Prec *prec){
 CriticalSection::Lock __(sPteLock);
 if(prec && --prec->n == 0){
  if(prec->ptr){
   prec->ptr->prec = NULL;
   prec->ptr->OnPrecDelete();
  }
  delete prec;
  prec = NULL;
 }
}
```

In this case I can use for my specialized class this functionality by:

```
#include <Core/Core.h>

using namespace Upp;

struct Foo : public Pte<Foo> {
 String text;
 ~Foo();
protected:
 virtual void OnPrecDelete(){delete (Foo*)this;}
};

Foo::~Foo(){
 Cout() << "deleted " << (void*)(this) << "\n";
}

Foo* factory(){
 return new Foo;
}
```

```
CONSOLE_APP_MAIN
{
 Ptr<Foo> ptr;
 {
  Ptr<Foo> ptr1 = new Foo;
  ptr1->text = "Text";
  ptr = ptr1;
  Cout() << (void*)~ptr << " -> " << ptr->text << "\n";
 }
 Cout() << "-------------\n";
 Cout() << (void*)~ptr << "\n";
 ptr = factory();
 ptr = factory();
}
```

Subject: Re: Ptr improve
Posted by mirek on Mon, 16 May 2011 21:13:36 GMT
View Forum Message <> Reply to Message

Well, if you want to go this path, I recommend boost.

You will get all kinds of smart pointers there...

My personal opinion is that all this stuff only makes your code ineffective and hard to maintain. But it is your choice, after all

Mirek

Subject: Re: Ptr improve
Posted by kohait00 on Wed, 18 May 2011 09:40:17 GMT
View Forum Message <> Reply to Message

for this matter
http://www.cplusplus.com/reference/std/memory/auto_ptr/
  http://www.boost.org/doc/libs/1_46_1/libs/smart_ptr/smart_pt r.htm

std::auto_ptr -> UPP::One<>, sole ownership, not shared amongst others, pick semantic

Quote:
Conceptually, smart pointers are seen as owning the object pointed to, and thus responsible for deletion of the object when it is no longer needed.

The smart pointer library provides six smart pointer class templates:

scoped_ptr <boost/scoped_ptr.hpp> Simple sole ownership of single objects. Noncopyable.
scoped_array <boost/scoped_array.hpp> Simple sole ownership of arrays. Noncopyable.
shared_ptr <boost/shared_ptr.hpp> Object ownership shared among multiple pointers.
shared_array <boost/shared_array.hpp> Array ownership shared among multiple pointers.
weak_ptr <boost/weak_ptr.hpp> Non-owning observers of an object owned by shared_ptr.
intrusive_ptr <boost/intrusive_ptr.hpp> Shared ownership of objects with an embedded reference count.
These templates are designed to complement the std::auto_ptr template.

scoped_ptr is a restricted version of One<>

shared_ptr shares ownership of same object among multiple shared_ptr instances (aka ref count, or Value). for this we dont have a 'clean' leightweight implementations, sth like Shared<> would be great.

weak_ptr is a weak ref, pointing same stuff shared_ptr already points to, comparable to Ptr<>, it doesnt hold ownership, just as Ptr<> doesnt, so we actually *do* have allmost all of it.

maybe we really should consider to implement such a shared ownership container, sth like Shared<>

Subject: Re: Ptr improve
Posted by tojocky on Wed, 18 May 2011 11:12:12 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Wed, 18 May 2011 12:40
maybe we really should consider to implement such a shared ownership container, sth like Shared<>

Yes,
Shared<>(shared_ptr<>) is the container which I need. I will create this class and propose to vote.

Thank you!
Ion.

Subject: Re: Ptr improve
Posted by kohait00 on Thu, 19 May 2011 17:09:40 GMT
View Forum Message <> Reply to Message

here is a proposal..

```
//shared pointer
//idea borrowed from boost shared_ptr, an additional chunk of memory is managed
//which centrally holds the refcount of that object pointed to
//if Shared is created freshly, it AtomicInc's the ref count to 1;
//if a Shared is destroyed it AtomicDec's the refcount, and if its 0,
// it will delete both, the object and the refcount chunk
//if another instance is created as copy, the refcount is taken and incremented.
//if it is assigned, it decrements own existing counter, possibly releasing mem, and retains new
//pick semantic is not needed here anymore, it not even is possible
//since an 'operator=(const Shared<>&) is needed to aquire the source. pick is const in some
cases as well)
//thus Shared is only Moveable, without deepcopyoption, which in fact would speak agains the
idea of Shared anyway
//Attach / Detach remains

template <class T>
class Shared : Moveable< Shared<T> > {
 mutable T  *ptr;
 Atomic     *rfc;

 void Retain() const { ASSERT(rfc); AtomicInc(*rfc); }
 void Release()      { ASSERT(rfc); if(AtomicDec(*rfc) == 0) { Free(); delete rfc; rfc = NULL; } }

 void       Free()                  { if(ptr && ptr != (T*)1) delete ptr; }
 void       Chk() const               { ASSERT(ptr != (T*)1); }
 void       ChkP() const              { Chk(); ASSERT(ptr); }

public:
 void       Attach(T *data)           { Free(); ptr = data; }
 T         *Detach() pick_            { ChkP(); T *t = ptr; ptr = NULL; return t; }
 T         *operator-() pick_         { return Detach(); }
 void       Clear()                 { Free(); ptr = NULL; }

 void       operator=(T *data)        { Attach(data); }
 void       operator=(const Shared<T>& d){ Release(); ptr = d.ptr; rfc = d.rfc; Retain(); }
 void       operator=(pick_ One<T>& d) { Attach(d.Detach()); }

 const T    *operator->() const        { ChkP(); return ptr; }
 T          *operator->()              { ChkP(); return ptr; }
 const T    *operator~() const         { Chk(); return ptr; }
 T          *operator~()               { Chk(); return ptr; }
 const T&    operator*() const         { ChkP(); return *ptr; }
 T&         operator*()                { ChkP(); return *ptr; }

 template <class TT>
 TT&        Create()                  { TT *q = new TT; Attach(q); return *q; }
 T&         Create()                 { T *q = new T; Attach(q); return *q; }
```

```
 bool       IsEmpty() const           { Chk(); return !ptr; }

 operator bool() const                { return ptr; }

 Shared()                            { ptr = NULL; rfc = new Atomic(1); }
 Shared(T *newt)                        { ptr = newt; rfc = new Atomic(1); }
 Shared(const Shared<T>& p)             { ptr = p.ptr; rfc = p.rfc; Retain(); }
 ~Shared()                           { Release(); }

 Shared(pick_ One<T>& p)               { ptr = p.Detach(); rfc = new Atomic(1); }
 Shared(const One<T>& p, int)          { ptr = DeepCopyNew(*p); rfc = new Atomic(1); }
};
```

i first thought deriving from One<> but it will have problems with pick semantics
so i decided to stay with a clean separated version, but it's 80% One<> code
i added a convenience pick semantic for One<>

it's open for discussion..

```
Shared<Size> Test(Shared<Size> s)
{
 if(!s.IsEmpty())
  RLOG(*s);
 return s;
}

CONSOLE_APP_MAIN
{
 Shared<Size> p;
 {
 Shared<Size> s;

 s.Create();
 *s = Size(123,456);

 Shared<Size> q;
 q = Test(s);

 p = q;
 }
 if(!p.IsEmpty())
  RLOG(*p);
 One<Size> os;

 os.Create();
```

```
 *os = Size(1,2);
 p = os;
 RLOG(*p);

 os.Create();
 *os = Size(3,4);
 p = Shared<Size>(os);
 RLOG(*p);
}
```

Subject: Re: Ptr improve
Posted by tojocky on Fri, 20 May 2011 06:32:07 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Thu, 19 May 2011 20:09here is a proposal..

```
//shared pointer
//idea borrowed from boost shared_ptr, an additional chunk of memory is managed
//which centrally holds the refcount of that object pointed to
//if Shared is created freshly, it AtomicInc's the ref count to 1;
//if a Shared is destroyed it AtomicDec's the refcount, and if its 0,
// it will delete both, the object and the refcount chunk
//if another instance is created as copy, the refcount is taken and incremented.
//if it is assigned, it decrements own existing counter, possibly releasing mem, and retains new
//pick semantic is not needed here anymore, it not even is possible
//since an 'operator=(const Shared<>&) is needed to aquire the source. pick is const in some
cases as well)
//thus Shared is only Moveable, without deepcopyoption, which in fact would speak agains the
idea of Shared anyway
//Attach / Detach remains

template <class T>
class Shared : Moveable< Shared<T> > {
 mutable T  *ptr;
 Atomic     *rfc;

 void Retain() const { ASSERT(rfc); AtomicInc(*rfc); }
 void Release()      { ASSERT(rfc); if(AtomicDec(*rfc) == 0) { Free(); delete rfc; rfc = NULL; } }

 void       Free()                { if(ptr && ptr != (T*)1) delete ptr; }
 void       Chk() const             { ASSERT(ptr != (T*)1); }
 void       ChkP() const             { Chk(); ASSERT(ptr); }

public:
 void       Attach(T *data)          { Free(); ptr = data; }
```

```
 T         *Detach() pick_           { ChkP(); T *t = ptr; ptr = NULL; return t; }
 T         *operator-() pick_        { return Detach(); }
 void       Clear()                 { Free(); ptr = NULL; }

 void       operator=(T *data)        { Attach(data); }
 void       operator=(const Shared<T>& d){ Release(); ptr = d.ptr; rfc = d.rfc; Retain(); }
 void       operator=(pick_ One<T>& d) { Attach(d.Detach()); }

 const T    *operator->() const       { ChkP(); return ptr; }
 T         *operator->()             { ChkP(); return ptr; }
 const T    *operator~() const        { Chk(); return ptr; }
 T         *operator~()              { Chk(); return ptr; }
 const T&    operator*() const        { ChkP(); return *ptr; }
 T&         operator*()              { ChkP(); return *ptr; }

 template <class TT>
 TT&        Create()                 { TT *q = new TT; Attach(q); return *q; }
 T&         Create()                { T *q = new T; Attach(q); return *q; }

 bool       IsEmpty() const          { Chk(); return !ptr; }

 operator bool() const               { return ptr; }

 Shared()                          { ptr = NULL; rfc = new Atomic(1); }
 Shared(T *newt)                     { ptr = newt; rfc = new Atomic(1); }
 Shared(const Shared<T>& p)          { ptr = p.ptr; rfc = p.rfc; Retain(); }
 ~Shared()                         { Release(); }

 Shared(pick_ One<T>& p)             { ptr = p.Detach(); rfc = new Atomic(1); }
 Shared(const One<T>& p, int)        { ptr = DeepCopyNew(*p); rfc = new Atomic(1); }
};
```

i first thought deriving from One<> but it will have problems with pick semantics
so i decided to stay with a clean separated version, but it's 80% One<> code
i added a convenience pick semantic for One<>

it's open for discussion..

```
Shared<Size> Test(Shared<Size> s)
{
 if(!s.IsEmpty())
  RLOG(*s);
 return s;
}

CONSOLE_APP_MAIN
```

```
{
 Shared<Size> p;
 {
 Shared<Size> s;

 s.Create();
 *s = Size(123,456);

 Shared<Size> q;
 q = Test(s);

 p = q;
 }
 if(!p.IsEmpty())
  RLOG(*p);
 One<Size> os;

 os.Create();
 *os = Size(1,2);
 p = os;
 RLOG(*p);

 os.Create();
 *os = Size(3,4);
 p = Shared<Size>(os);
 RLOG(*p);
}
```

Very nice,

I thought deriving from PtrBase, but it conflicts with Ptr<>.
Your proposal seems to be very clear.

Subject: Re: Ptr improve
Posted by kohait00 on Fri, 20 May 2011 09:04:45 GMT
View Forum Message <> Reply to Message

i am thinking of an addition

class A;

class B : A

think of a Shared<B> that can point to same instance, as some Shared<A> do.. so they need to

share the refcount.. so which ever of the Shared instances dies last, will destroy the object properly. provided virtual dtor..

thats a current use case for me now..
in C# this is done automatically, since 'object' has got a ref count internally. with this U++ would come very close to that..

Subject: Re: Ptr improve
Posted by cbpporter on Fri, 20 May 2011 09:51:26 GMT
View Forum Message <> Reply to Message

AFAIK, C# uses generational garbage collection, not reference counting.

And what we have now with U++ is a lot better that reference counting (except the few cases where we actually use reference counting).

If I understood Mirek correctly, he suggested to directly use one of the smart pointer variants from boost instead of reinventing them in U++ .

Subject: Re: Ptr improve
Posted by kohait00 on Fri, 20 May 2011 11:03:26 GMT
View Forum Message <> Reply to Message

that's what mirek said

nevertheless, the proposal is simple, so we dont need to pack out boost canon for this simple usecase. generational GC is fine, we dont have it and can live with it very well.. ref count is in most cases more than enough.

Subject: Re: Ptr improve
Posted by mirek on Fri, 20 May 2011 11:19:07 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Fri, 20 May 2011 07:03that's what mirek said

nevertheless, the proposal is simple, so we dont need to pack out boost canon for this simple usecase.

There is no usecase, as long as you design the code U++ way.

Well, perhaps you can dismiss following as anecdotal evidence, but I have produced quite a lot of software in the past 10 years (say about half million lines of C++). I was addressing pretty wide

area of problems, from theide to website backends. There are no shared ownership pointers in my code anywhere. So what is the usecase?

Mirek

Subject: Re: Ptr improve
Posted by kohait00 on Fri, 20 May 2011 12:43:43 GMT
View Forum Message <> Reply to Message

i totally agree with you, thinking the shared or semi-GC way to the end, if it's implemented halfway (like this would be in this case) would bring one in serious trouble sooner or later. you end up breaking your head on how to prevent this damn thing from beeing destroyed. and u keep a Shared<T> instance around to do just that. and you could keep the whole instance just as well.

if at all used, it needs to be veeery consequent. so if one decides to handle an object in shared way, it needs to go completely like that. no mixes of API's (half is using Shared<T> and half T*).. and it makes trouble when handling a polymorph type, here you want a Shared<Base>, there a Shared<Derived> and they'd need to have the same ref (which i adressed as well, but it's not posted yet)

nevertheless, as a helper, i will put it in my Gen package, as a matter of discussion, just in case one happens to need it some time again. is that ok?

Subject: Re: Ptr improve
Posted by kohait00 on Mon, 23 May 2011 11:34:31 GMT
View Forum Message <> Reply to Message

Shared<> is in bazaar/Gen package..
the stuff above i put into a src.doc section there

Subject: Re: Ptr improve
Posted by cbpporter on Mon, 23 May 2011 11:52:07 GMT
View Forum Message <> Reply to Message

Here is an idea that has been going around in my head for a while: what if we combine the U++ way with GC.

GC is great, but the cost of mark & sweep can be to much for some cases.

Traditional memory management is problematic, and we have a relatively big cost of allocation/deallocation.

With the U++ we have everything belongs somewhere. But what if we kept the principle intact for non heap allocated objects, but for heap allocated one, the destructor would only mark the object for deletion, and it would actually get deleted latter. So basically mark&sweep, but without mark, its role handled by destructors. It would even make GC more deterministic, but not by much.

What do you think?

Subject: Re: Ptr improve
Posted by kohait00 on Mon, 23 May 2011 13:23:09 GMT
View Forum Message <> Reply to Message

i think this is not easy to accomplish in upp. it would make the use of the UPP mem manager obligatory, USE_MALLOC is past then.

Subject: Re: Ptr improve
Posted by dolik.rce on Mon, 23 May 2011 16:26:46 GMT
View Forum Message <> Reply to Message

The GC in U++ idea is interesting, but I think it will never be the main memory management strategy for U++ It could be optional using e.g. USE_GC flag, which would be (most probably) mutually exclusive with USE_MALLOC. I just wonder if it would really bring enough speed up to justify all the code to be written to make it work

---

Subject: Re: Ptr improve
Posted by mirek on Mon, 23 May 2011 20:37:25 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Mon, 23 May 2011 07:52Here is an idea that has been going around in my head for a while: what if we combine the U++ way with GC.

GC is great, but the cost of mark & sweep can be to much for some cases.

Traditional memory management is problematic, and we have a relatively big cost of allocation/deallocation.

Well, that is relative, average allocation/deallocation is about pretty fast in U++.... (about the same as link/unlink in double-linked list + a couple of simple loads)

Quote:
With the U++ we have everything belongs somewhere.

GC is quite incompatible with the concept of destructors.

Quote:
But what if we kept the principle intact for non heap allocated objects, but for heap allocated one, the destructor would only mark the object for deletion, and it would actually get deleted latter.

IMO results in unmaintainable mess.

Besides, GC AFAIK is still not a part of C++. And you cannot realistically add it by library - the best you can do is stochastic approach (Boehm), which works fine, unless you are processing white noise

Still, what is unclear to my is why to complicate your code with heap if you do not have to? The whole mission of U++ is to eliminate universal shared heap as much as possible.

Mirek

Subject: Re: Ptr improve
Posted by kohait00 on Mon, 23 May 2011 20:41:53 GMT
View Forum Message <> Reply to Message

i have to admit that from time to time one is really tempted to 'enrich upp with this and that'. but its beauty is also its simplicity. i'm kinda thankfull that mirek is somehow conservative to my (countless) attempts also

---

Subject: Re: Ptr improve
Posted by cbpporter on Tue, 24 May 2011 07:25:34 GMT
View Forum Message <> Reply to Message

mirek wrote on Mon, 23 May 2011 23:37cbpporter wrote on Mon, 23 May 2011 07:52Here is an idea that has been going around in my head for a while: what if we combine the U++ way with GC.

GC is great, but the cost of mark & sweep can be to much for some cases.

Traditional memory management is problematic, and we have a relatively big cost of allocation/deallocation.

Well, that is relative, average allocation/deallocation is about pretty fast in U++.... (about the same as link/unlink in double-linked list + a couple of simple loads)

Quote:
With the U++ we have everything belongs somewhere.

GC is quite incompatible with the concept of destructors.

Quote:
But what if we kept the principle intact for non heap allocated objects, but for heap allocated one, the destructor would only mark the object for deletion, and it would actually get deleted latter.

IMO results in unmaintainable mess.

Besides, GC AFAIK is still not a part of C++. And you cannot realistically add it by library - the best you can do is stochastic approach (Boehm), which works fine, unless you are processing white noise

Still, what is unclear to my is why to complicate your code with heap if you do not have to? The whole mission of U++ is to eliminate universal shared heap as much as possible.

Mirek

Destructors are not generally incompatible with GC, just the normal GC scenarios and implementation we have now. What I am proposing is reversing the order of the GC flow.

Another additional advantage would be that it would eliminate one of the big disadvantages of conservative GC: false positives and inability to deal with extremely large heaps or data that looks like pointers.

With what I am proposing, only objects whose destructors have been called will be collected, or those that were marked by the API. It is not GC, it is the U++ memory management flow, with one single difference: data is not deleted immediately. It is delayed.

This would apply to all containers that allocate memory, so it is not about eliminating universal shared heap or not so it wouldn't contrast with our mission statement.

Subject: Re: Ptr improve
Posted by mirek on Tue, 24 May 2011 15:46:17 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 24 May 2011 03:25mirek wrote on Mon, 23 May 2011 23:37cbpporter wrote on Mon, 23 May 2011 07:52Here is an idea that has been going around in my head for a while: what if we combine the U++ way with GC.

GC is great, but the cost of mark & sweep can be to much for some cases.

Traditional memory management is problematic, and we have a relatively big cost of allocation/deallocation.

Well, that is relative, average allocation/deallocation is about pretty fast in U++.... (about the same as link/unlink in double-linked list + a couple of simple loads)

Quote:
With the U++ we have everything belongs somewhere.

GC is quite incompatible with the concept of destructors.

Quote:
But what if we kept the principle intact for non heap allocated objects, but for heap allocated one, the destructor would only mark the object for deletion, and it would actually get deleted latter.

IMO results in unmaintainable mess.

Besides, GC AFAIK is still not a part of C++. And you cannot realistically add it by library - the best you can do is stochastic approach (Boehm), which works fine, unless you are processing white noise

Still, what is unclear to my is why to complicate your code with heap if you do not have to? The whole mission of U++ is to eliminate universal shared heap as much as possible.

Mirek

Destructors are not generally incompatible with GC, just the normal GC scenarios and implementation we have now. What I am proposing is reversing the order of the GC flow.

Another additional advantage would be that it would eliminate one of the big disadvantages of conservative GC: false positives and inability to deal with extremely large heaps or data that looks like pointers.

With what I am proposing, only objects whose destructors have been called will be collected, or those that were marked by the API. It is not GC, it is the U++ memory management flow, with one single difference: data is not deleted immediately. It is delayed.

This would apply to all containers that allocate memory, so it is not about eliminating universal shared heap or not so it wouldn't contrast with our mission statement.

Well, very unlikely to happen, but for the fun of theoretical debate:

How do you plan to implement GC? are you going to write your C++ compiler? How would it handle typical C++ inconsistencies like pointer casts

In the past, I was experimenting with such library based GC approach, where only specific objects were subjects of GC. I was able to get it working, and it was quite optimal, however the whole thing collapsed the moment you have started mixing such objects and regular ones. I have not found a way out. Since then I consider the whole point of GC in C++ moot - perhaps Boehm or is

good for fixing leaky code, but my conclusion is that you can either have reliable destructors or GC. There is nothing in between.

Mirek

Subject: Re: Ptr improve
Posted by mr_ped on Tue, 24 May 2011 16:28:12 GMT
View Forum Message <> Reply to Message

Once you have reliable destructor call, what's the point to not free also memory? QED what Mirek said.

Subject: Re: Ptr improve
Posted by cbpporter on Tue, 24 May 2011 20:44:51 GMT
View Forum Message <> Reply to Message

Well, I guess you don't like my idea . It has been on my mind for some time. I am wondering if we could get a best of both worlds scenario out of this.

There is no need for writing compilers or libraries. Allocating memory without a container or a smart pointer and not freeing it would still be a memory leek. What I want is to delay the free operation. It does not have anything to do with pointer casting.

GC proponents have been hyping at least three things: no memory leaks, the advantage on parallel computing caused by a functional style combined with GC enabled more frequent allocations done to enable immutable data structures and and the responsiveness of allocation and deallocation. I was wondering if we could get some of that final point with what we have in U++ and test if it does bring an advantage or not. Seems like a fun experiment. We would need a very fast allocator, even at the price of a very slow deallocator.

And yes, my approach would need the use of Shared or a similar "rich pointer". Just using normal C pointer would be as bad of an idea as using them in normal U+ code to manage memory.

Subject: Re: Ptr improve
Posted by mr_ped on Wed, 25 May 2011 07:34:37 GMT
View Forum Message <> Reply to Message

"I am wondering if we could get a best of both worlds"

The problem with me is, that I don't see anything good about GC, except that you can be careless about where are your data.

As most of the SW task is to manipulate data, I don't find that idea attractive at all, I prefer to know

exactly what my data do and where they sit and why. From there I never really wished for GC, gives me goosebumps to never know exactly when the memory is freed.

edit:
"What I want is to delay the free operation"
and why?? What's the advantage? I see plenty of problems with it, but no single advantage.

Subject: Re: Ptr improve
Posted by kohait00 on Wed, 25 May 2011 08:27:43 GMT
View Forum Message <> Reply to Message

the strong typed nature of C++ and especially U++ leaves you with a lot of control and responsibility. imho constructing object without care (like it is in C#) is both easy and wasting performance, inviting for careless handling with ressources. knowing what's up and needed is best practice to keep apps reactive. beeing that said, i go along with mr_pen & mirek.

this does not mean that Shared<> is just bad per se. it is merley a helper. but seeing it as an invitation to go GC is a bit too far i think. as mirek said, GC is compiler support. either have it or dont. there is no safe way in between.

some more info about shared_ptr usage, which clearly shows that it brings a lot of hassle as well. hassle one can spare when investing the thinking power in a correct model. a lot of work to save work is cumbersome.
http://www.codeproject.com/KB/stl/boostsmartptr.aspx#Example : Using shared_ptr in containers

---

Subject: Re: Ptr improve
Posted by cbpporter on Wed, 25 May 2011 10:09:58 GMT
View Forum Message <> Reply to Message

mr_ped wrote on Wed, 25 May 2011 10:34
edit:
"What I want is to delay the free operation"
and why?? What's the advantage? I see plenty of problems with it, but no single advantage.

Well the advantage is that you could have normal program flow without allocating and deallocating breaking up you execution. Deallocation would happen as a bulk operation only once in a while, either naturally or under the control of the programmer, i.e. before opening up a dialog and waiting for user input.

While I can routinely prove that C++ is considerably faster than a GC language, there are a few scenarios where C++ looses, and almost all of them involve allocating and freeing a huge number of objects.

I do not agree that we need compiler support, but here is the thing: with what I am proposing we do have compiler support. The destructors.

But I'll drop the subject now since nobody thinks this is a good idea.

Subject: Re: Ptr improve
Posted by kohait00 on Wed, 25 May 2011 10:19:47 GMT
View Forum Message <> Reply to Message

Quote:
But I'll drop the subject now since nobody thinks this is a good idea.

not nesseccary. why not provide a proof-of-concept?

Subject: Re: Ptr improve
Posted by mirek on Sat, 28 May 2011 18:57:17 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 24 May 2011 16:44
GC proponents have been hyping at least three things: no memory leaks, the advantage on parallel computing caused by a functional style combined with GC enabled more frequent allocations done to enable immutable data structures and and the responsiveness of allocation and deallocation.

I would say that the partial problem here is that there was huge development invested in GC, but much less in traditional malloc/free.

That said, I believe that the current iteration of U++ allocator is close to optimal and is able to beat any GC easily...

BTW, cross-thread free's are already deferred in U++ allocator - but that simply happens "behind the scene"... In-thread deallocation is usually about as 'complex' as one simple test, one load from memory and linking single element to the the double-linked list...

Mirek

Subject: Re: Ptr improve
Posted by mirek on Sat, 28 May 2011 19:10:04 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 25 May 2011 06:09mr_ped wrote on Wed, 25 May 2011 10:34
edit:

"What I want is to delay the free operation"
and why?? What's the advantage? I see plenty of problems with it, but no single advantage.

While I can routinely prove that C++ is considerably faster than a GC language, there are a few scenarios where C++ looses, and almost all of them involve allocating and freeing a huge number of objects.

I have only experienced this in scenario where there are only allocations and garbage collection is never invoked.

That said, in C++ you would need to keep the track of freed blocks (for bulk free) and that would likely be as expensive as current U++ allocator free...

Mirek
