Allocators for Shared Memory in C++03, C++11, and Boost

ACCU 2013, Bristol

Frank Birbacher
Inform GmbH, Aachen

13th April 2013, r210

C++ allocators are rarely looked at but for using shared memory as a means of interprocess communication they spring to mind. For a long time this was a good idea in theory but in practice not until C++11. Explaining allocators and their use case "shared memory" is the focus here. Shared memory is shown using tne Boost.Interprocess library.

Contents

Goals of this presentation **Slide 2**

- show functionality of allocators
- show allocator use in shared memory
- compare allocators in C++03 and C++11
- show the scoped allocator adaptor

The functionality of C++03 allocators is explained and how the idea of allocators can be leveraged for accessing shared memory. Examples will demonstrate construction of standard containers in shared memory. Afterwards the changes to allocators in C++11 will be shown. These changes influence the way containers use allocators and they enable more control over object construction. This will be shown with the scoped allocator adaptor.

1 Allocators 03

1.1 Idea

The hidden strategy **Slide 5**

- allocator is parameter of containers
- allocator is member of containers
- strategy pattern
- rarely changed

```
namespace std {
template<
    typename T,
    typename Alloc = std::allocator<T> // <==
    >
class list { ... };
```

An allocator is rarely looked at. It is a template parameter of every standard container and defaults to std::allocator. If changed this parameter will influence the way a container interacts with memory. Additionally an instance of the given type is stored as a container member. As such the allocator is a strategy object that may carry state.

Concept **Slide 6**

Influence ...

- access to objects
  - abstraction from memory model
  - for example far and near pointer (Intel x86 16bit)
- allocation scheme

    - changing allocation strategy
- object construction
    - influence initialization $\Longrightarrow$ C++11 scoped allocator

by defining …

- types for pointer and its arithmetic
    - usually T*
    - could be a pointer like class
    - could be separate pointer type provided by compiler
- replacements for "malloc" and "free"
    - called "allocate" and "deallocate"
    - they match pointer type
    - provide uninitialized memory
- wrappers for object con-/destruction
    - called "construct" and "destroy"
    - initialise given memory

An allocator has three tasks: first it defines how memory and objects are addressed and accessed, second it may change the way to allocate raw memory, and third it can influence object construction.

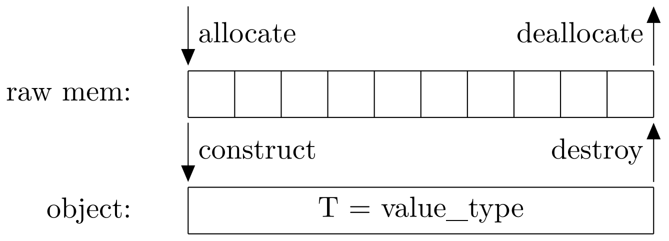
Allocators were invented in 1994 in preparation of the C++1998 standard. It was envisioned allocators could enable access to different types of memory like shared memory, persistent memory, or different memory hardware in embedded systems. Alexander Stepanov, the original author of the STL, said the following:

> A nice feature of STL is that the only place that mentions the machine-related types in STL—something that refers to real pointer, real reference—is encapsulated within roughly 16 lines of code. *(Alexander Stepanov, in Dr. Dobb's Journal, March 1995)*

1.2 Definition

Functionality of allocators Slide 8

template<typename `T>` **struct** `allocator`



- two step object construction
- allocation and construction separated

Before showing the raw definition of allocators we take a look at the functionality of allocators. That is the way they are used to create new objects in memory.

An allocator is usually a class template with a parameter. This parameter is called the value type. The allocator is prepared to provide and initialize memory for its value type.

The object construction and memory allocation are separated. Memory is allocated by calling the function allocate which returns uninitialised memory. This memory is initialised using the function construct. On the reverse the functions destroy and deallocate free the object.

A quick reference of all allocator members is given in the appendix, see page 22.

Pointer type **Slide 9**

Task 1: define pointer type

```
template<typename T> struct allocator
{
    typedef T value_type;
    typedef ... pointer; // T*
    typedef ... const_pointer; // T const*
    typedef ... reference; // T&
    typedef ... const_reference; // T const&

    typedef ... size_type; // size_t
    typedef ... difference_type; // ptrdiff_t
    ...
```

An allocator defines a bunch of types. At first it publishes its value type. But the most important one is the pointer type. It may be any pointer like type that overloads operators -> and unary * and acts like a random access iterator. The pointer type is accompanied by the respective reference type and types for pointer arithmetic. The size_type is unsigned while the difference_type is signed and is the result of pointer subtraction.

Malloc and free **Slide 10**

Task 2: change allocation scheme

```
template<typename T> struct allocator
{ ...
    pointer allocate(size_type nObjs, void* hint =0);
        // malloc(nObjs * sizeof(T))

    void deallocate(pointer p, size_type nObjs);
    // free(p)
  ...
```

Besides the pointer type the allocator provides the functions allocate and deallocate. They act similar to malloc and free in that they provide uninitialised memory. But they define the size of the request in number of objects of value type, not in number of bytes.

A possible implementation of allocate could call malloc. It would multiply the number of objects and the size of its value type to determine the number of bytes required. The optional hint pointer can have special meaning which is solely defined by the allocator. On the slide the hint pointer is shown with type void*, but actually it has the pointer type of the void instantiation of the allocator:

`allocator<void>::const_pointer`.

The value of the hint must be a still valid—not deallocated—previously allocated memory block. For instance it could be a request to place the new allocation near the old one in order to increase data locality.

Note that the deallocate function also takes the size of the block. This is unusual, but the value is usually known and it may help the allocator implementation on custom allocation schemes.

Construction and Destruction — Slide 11

Task 3: wrap object construction

```
template<typename T> struct allocator
{ ...
    // in C++03:
    void construct(pointer, T const&);
    void destroy(pointer);
    ...
```

For actual object construction and destruction the functions construct and destroy are provided. The first one copies a given object instance into the memory that the pointer points to. That means construct will invoke the copy constructor of T. The function may have side effects.

The second function simply has to invoke the destructor of the object pointed to. It may have side effects as well. This very limited functionality of construct and destroy is enhanced in C++11.

Changing value type — Slide 12

$$\begin{tikzcd}
 & \text{A = allocator<int>} \arrow[d, "\textit{rebind+ctor}"] \\
\text{list<int, A>} \arrow[ur, "\textit{gets}"] \arrow[dr, "\textit{wants}"'] & \\
 & \text{allocator<list::node>}
\end{tikzcd}$$

```
template<typename T> struct allocator
{ ... // "template typedef":
    template<typename U> struct rebind
    { typedef allocator<U> other; };
    // constructor:
    template<typename U>
    allocator(allocator<U> const&);
```

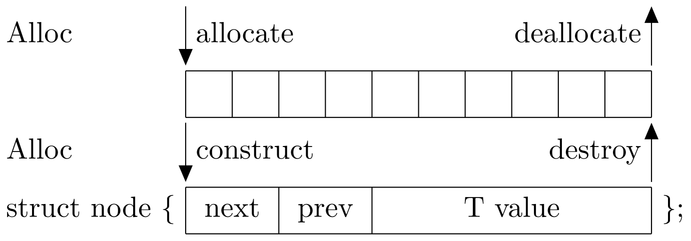
The allocator instances passed to standard containers always specify the same value type as the container. But for some containers objects of different type have to be created in memory. For instance the std::list will construct list nodes with extra pointers to store the links to next and previous nodes. This means a list<int> gets an allocator for int but requires an allocator for its node<int> type.

In order to enable construction of nodes the list needs to somehow "rebind" the given allocator type to a new value type. This is accomplished by a nested class template named "rebind" that provides a new allocator instantiation. This is like the new C++11 alias templates, but in old C++98 style.

The rebind template is accompanied by a converting constructor that converts any allocator object into its own type. Using the rebind template the list determines the rebound allocator type and using the constructor it converts any allocator object passed in to that new type. It will store and use that converted object as a member.

A quick reference of all allocator members is given in the appendix, see page 22.

std::list using an allocator — Slide 13



Continuing with the list the picture shows the steps to construct a list node. On the rebound allocator the allocate function provides the list with memory for a single node. In that memory the list constructs a node by means of the construct function. The node contains the list pointers and the actual list element type. As this demonstrates the element type T is not constructed by the allocator directly. That's why the allocator<T> is of no direct use. Everything is done by the allocator<node<T> >. Code that shows the approach can be found in the appendix, see page 23.

1.3 Problems in C++03

Problems in the definition — Slide 15

- reference type redundant

  - will always be T&, no replacement possible
  - cannot overload operator .
- lengthy definition
  - mostly according to the book
- construct copies argument
  - no in-place construction
  - only copy possible
- construct/destroy restricted
  - construct: new((void*)p) T(t)
  - destroy: (*p).~T()
- bothersome default constructor
- behaviour on container copy/assign/swap unclear

The specification of allocators in the C++03 standard comes with some restrictions and problems. The reference typedef can be freely defined by an allocator, but possible definitions are restricted to T&. No other type can be used in place of a reference since the operator . cannot be redefined. Alone compiler extensions could provide replacements. In effect the reference type is pinned to T& and thus it is redundant.

The definition of a custom allocator encompasses a bunch of typedefs and functions, most of which are identical to the std::allocator and only few are actually changed. There are no utilities to help in defining a custom allocator. This makes own definitions long.

The construct method restricts object construction. It requires a prototype object that will be copied into the given memory location. There is no way to avoid the copy or use different constructors besides the copy constructor. The destruct function is required to call the object destructor.

A serious restriction is for allocators to have default constructors. This becomes visible when an allocator has members that need to be initialised. Often there is no sensible default for the allocator state. Using such a default constructed allocator can only be disabled by runtime checks.

At last the behaviour of a container assignment is not clear. An allocator might be transferred on container copy assignment, or it may not. Same for container swap. The question seems simple, but in practise it might not.

Shortcomings in implementation — Slide 16

Standard containers may assume ...

- “pointer” is always T*
- operator == always returns true
  - all allocator instances compare equal
  - thus an allocator doesn’t need member variables
- allocator without state

  - no member variables anyway
  - allocator not copied along with container

One of the major shortcomings of the C++03 allocators is their option on implementation. An implementor of the STL may assume the allocator does not use a custom pointer type and the operator == for the allocator always returns true. This means there is no sense in giving allocators any member variables that define state. Because an allocator object compares equal to any other it does not need to be copied or swapped along with containers.

This seriously limits the use of custom allocators and maybe led to late adoption of proper allocator handling in STL implementations. Some implementations fail to cope with changed pointer types.

2 Shared Memory

2.1 What is it?

Rough idea — Slide 18

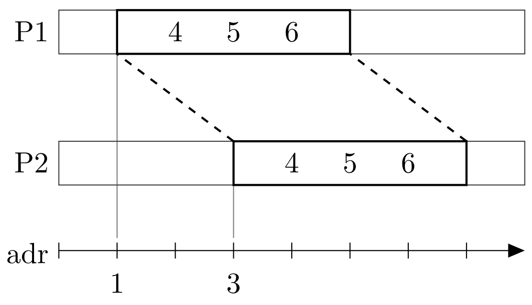
```
void* base = open_shmem("identifier");
```

- memory block with identifier
- same data for all processes
- data race issues as with threads
- persisted in file

Shared memory is enabled by hardware when processes use a virtual address space that is mapped to physical memory on the fly during program execution. It can easily be arranged for two processes to use the same physical memory through different virtual addresses in their respective address spaces. Shared memory is provided by the operating system as a named memory block. Data in this block is the same for all participating processes at all times.

The fact that two processes access the same physical memory is no different from two threads accessing the same memory. That's why we have to face the same data race issues as with threads.

Uncertainty of base address — Slide 19



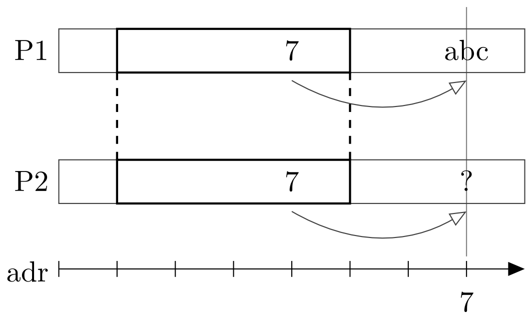
Base address different …

- for different processes
- for different runs of same exe
- for repeated calls to "open"

The memory block that represents the shared memory is accessed by a process using regular pointers. The pointer to the beginning of the memory block is called the base address. It may differ in different processes although they've access to the same shared memory. The base address may also differ for repeated execution of the same program. This can be compared to the behaviour of malloc.

2.2 Pointers in shared memory

Pointing to outside of shared memory **Slide 21**

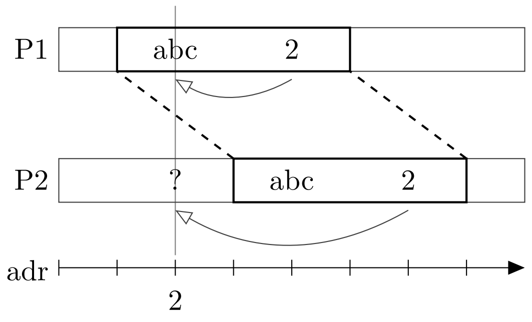


- P1 stores "7"
- "7" points to "abc" outside shmem
- "abc" not found by P2

In the picture two processes are shown that have a linear address space—shown as horizontal rectangles—with increasing addresses from left to right. They share the highlighted potions of their memory.

When using shared memory to transfer data between processes it does not suffice to store a data pointer in shared memory. Suppose the data "abc" is to be transferred from P1 to P2 and the data is located at address 7 in P1. P1 stores the 7 in the shared memory. But this pointer is of no use in P2 since the data is not accessible. The data to be transferred needs to be stored in shared memory, too.

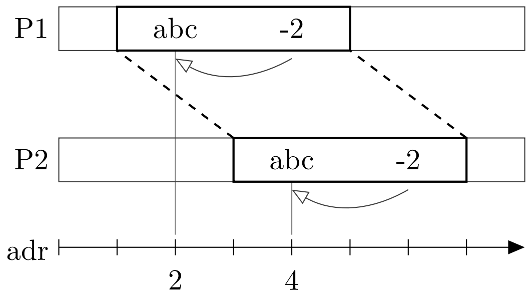
Pointing to shared memory Slide 22



- P1 stores “2”
- “2” points to “abc” inside shmem
- “abc” not found by P2

In this scenario the process P1 stores the data “abc” in shared memory and also stores the pointer to the data there. The pointer has value 2 and P2 can read it. But since the base address of shared memory blocks may be different in different processes there is no use in reading the address 2 in P2. The second process fails to access the data.

Using offset Slide 23



- offset “-2” points to “abc”
- “abc” now found by P2
- function pointers won’t work

Here the process P1 stores an offset to the data “abc”. The offset is calculated between the location where the offset is stored and the location of the data. The process P2 reads the offset and can now determine the correct address of the data in terms of its own address space. In general regular pointers in shared memory are useless. Because of this Boost uses offsets throughout its Interprocess library, at least internally.

The Boost offset_ptr Slide 24

```
template<typename T> struct offset_ptr
{
   ptrdiff_t offset;

   offset_ptr( T* p )
      : offset( p - this ) // in bytes
   {}
   ...
```

- stores offset from self
- retrieve pointer on dereference
- can be placed in shmem

To ease the use of offsets Boost defines an offset_ptr template. It stores a pointer by calculating the offset of the pointee to its own location and storing that difference. Upon access the offset_ptr can retrieve the original pointer by using the stored offset and its own this pointer.

2.3 Boost.Interprocess: The Big Picture

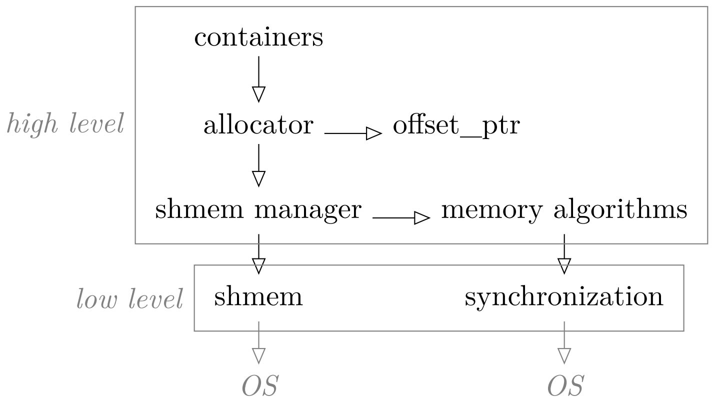
Managed shared memory Slide 26

shmem manager provides:

- uniform scheme to start ipc
- memory algorithm
- named objects

When seriously using shared memory for interprocess communication one usually creates some interface that is stored at the very beginning of the shared memory block. This interface enables further communication of object locations in the block and somehow provides an memory allocation scheme in case two processes want to place a new object simultaneously. There is need for an memory algorithm and synchronisation methods. Both of which are provided by the Boost managed shared memory. Additionally new objects can be given a string identifier and can be located by using this string. This makes utilisation of the shared memory easy in the beginning.

Parts in Boost.Interprocess Slide 27



The Boost.Interprocess library provides lots of services. The most fundamental ones are the shared memory and the interprocess synchronisation methods. Both interact directly with the operation system and are show in the lower level of the diagram.

The higher level entities build upon the lower level. The shared memory manager combines memory algorithms with object naming. An allocator uses the manager to provided containers with memory allocated from a given shared memory block. It utilises the offset pointer to make containers suitable for shared memory.

Starting an example Slide 28

```
// get ”handle” to shared memory:
boost::interprocess::managed_shared_memory
    // creation / name / size
    segment(open_or_create, ”identifier”, 65536);

// no fiddling with base address:
MyType *instance = segment.construct<MyType>
    (”MyType instance”) //name of the object
    (0.0, 0); //ctor arguments
```

To actually use a managed shared memory segment we construct a managed_shared_ memory object and pass it the identifier for our segment and an initial size. If a segment has already been created with that identifier we just start to participate. Otherwise we create it with the specified size.

The segment object has methods that create named objects and construct them in place. We create an instance of MyType that is named “MyType instance” by calling construct. It’s constructed with two constructor arguments. By using the identifier another process can retrieve a pointer to the instance without explicitly fiddling with offsets or the base address of the segment.

The above example will serve as a base for the next couple of slides.

2.4 String in shared memory

Boost Interprocess Allocator Slide 30

allocator is class template on:

- value type
- segment manager

allocator function:

- pointer type is offset_ptr
- references shmem manager
- asks manager for raw memory

Our goal is to place a string object into shared memory. To accomplish this we will use the Boost.Interprocess allocator and containers. The allocator is a class template on its value type and also on the segment manager. We will just use a single type of segment manager, the one used by the managed shared memory.

The allocator does two important things: it defines the offset_ptr as its pointer type and calls the segment manager for memory. Therefore it stores a reference to the manager and has state.

Differing from a standard allocator the Boost allocator defines no default constructor. The interprocess library brings its own set of containers as a drop in replacement for standard containers. These are prepared to use an allocator without a default constructor. We will use the Boost containers exclusively.

Defining the string type — Slide 31

```
using namespace boost::interprocess;

// (1st) allocator using shmem:
typedef allocator<
    char,
    managed_shared_memory::segment_manager
    > CharAlloc;

// (2nd) parameterise basic_string:
typedef basic_string<
    char,
    std::char_traits<char>,
    CharAlloc //allocator here
    > IPCString;
```

For using a string in shared memory we first define an appropriate allocator type. The string is a container of char and thus requires a char allocator. We instantiate the interprocess allocator with char and the segment manager type.

This allocator is used to instantiate the basic_string. The basic_string template is modeled exactly like the standard string and as such it takes the same template parameters. We specify the character type and traits and the own allocator type as template arguments. We name this string type IPCString. The raw code is shown in the appendix, see page 26.

Constructing the string Slide 32

```
// (1st) make allocator instance:
CharAlloc achar(segment.get_segment_manager());

// (2nd) construct string, pass allocator:
IPCString *str = segment.construct<IPCString>
    ("app title") //name of the object
    ("hello", achar); //ctor arguments

std::cout << *str; //access string
*str = "world"; //access string
```

After we've defined the types for allocator and string we now need to construct the respective objects. First we create an instance of the CharAlloc and pass it the segment manager. This way the allocator knows where to get memory from.

Second we use the manager to construct an IPCString. This string is initialised with the literal "hello". In order for the string to place its string data in the correct shared memory it will receive the allocator object. This in turn knows the segment. The string constructor will use the given allocator to get memory from the manager. The string will copy its data into this memory.

This example just shows a string in shared memory, but in fact other container types are equally well brought into shared memory.

List of strings Slide 33

```
using namespace boost::interprocess;

// (1st) allocator for whole strings:
typedef allocator<
    IPCString,
    managed_shared_memory::segment_manager
    > StringAlloc;

// (2nd) list of strings:
typedef list<
    IPCString,
    StringAlloc //allocator here
    > IPCStringList;
```

The next step is to nest containers. We'll put a string into a list. In order to do this we define the appropriate allocator type first. Compared to the CharAlloc here the first template argument has changed to IPCString. Notice how this type itself already depends on the interprocess allocator template.

Second we use the fresh StringAlloc type to instantiate the list. The list shall contain strings so we give it the IPCString as the first argument. The second argument is the according allocator type.

Constructing the list **Slide 34**

```
// (1st) make allocator instance:
StringAlloc astr(achar);

// (2nd) construct list, pass allocator:
IPCStringList *list = segment.construct<IPCStringList>
    ("title list") //name of the object
    (astr); //ctor arguments

list->push_back(*str); //access list
list->push_back("abc"); // failure?!?
```

And again: for creating a string list in shared memory we first instantiate the allocator. We give it the other allocator to copy from. The conversion constructor will allow this. It has the same meaning as if we had taken the segment manager again. But the conversion clearly shows that the allocator is from the same template.

In a second step we construct the list. It will receive the allocator object in its constructor. Since this is the only argument the list will be start up empty.

We push the string from the previous slides onto the list. This works quite well: the string will be properly copied—twice of course. But pushing a string literal will produce a compile time error. We'll look into this on the next slide.

Mind the allocator **Slide 35**

```
list->push_back("abc"); //failure?!?

// mind the allocator, cumbersome:
list->push_back(IPCString("abc", achar)); //ok

// since Boost 1.37:
// better, still needs getting used to:
list->emplace_back("abc", achar); //ok
```

The list in shared memory seems friendly at first. But when we try to insert a string literal it fails. The reason is the signature of the push_back method. It does not take a string literal but an instance of its element type, namely IPCString. Thus an IPCString is constructed temporarily just to call push_back, but this string cannot be constructed without an allocator instance. The conversion from the literal to the IPCString calls for a default constructed allocator which is not allowed. This is the point where a default constructible allocator would quietly break the code.

To fix the call to push_back we need to explicitly construct the IPCString with the correct allocator object. This works but it's somewhat cumbersome. In C++11 we could call emplace_back instead and at least save the explicit construction of the string. Boost also offers emplace_back and it does so since version 1.37. The emplacement still requires us to specify the allocator. C++11 helps here with the scoped allocator which we will discuss just after presenting the C++11 allocators.

3 Allocators 11

3.1 What's new in C++11?

Separation of definition Slide 37

- std::pointer_traits<T>
  - now contains most typedefs
- std::allocator_traits<A>
  - provides defaults for most operations and types
- a minimum definition contains:
  - value_type
  - converting constructor
  - allocate / deallocate
  - operator == / !=

Implementing a custom allocator has become much easier with C++11. With the allocator_traits class providing defaults for nearly every aspect of an allocator we now just need to override the defaults where needed. The least we need to provide is the value_type typedef, the converting constructor, a pair of allocate and deallocate as well as the operators == and !=.

The allocator_traits uses the new pointer_traits which contains all the typedefs for the pointer type and its arithmetic. This allows to simply reuse the pointer_traits for other allocators or by themselves.

Extension of definition Slide 38

- more general "construct"
  - see next slide
- == becomes equivalence relation
- allocator determines copy semantics
  - allocator can be copied/assigned/swapped/moved with container
- default constructor is optional

Besides making the definition of allocators easier the new standard also modifies and extends the functionality of allocators. The function construct has had a major change which we will show on the next slide. The equality operators are now simply defined as describing an equivalence relation. It hasn't been as clear in the old standard. Allocators now may also influence container copy and swap semantics. Refer to the appendix for details, see page 25. And last the need for a default constructor was removed.

"construct" made more general — Slide 39

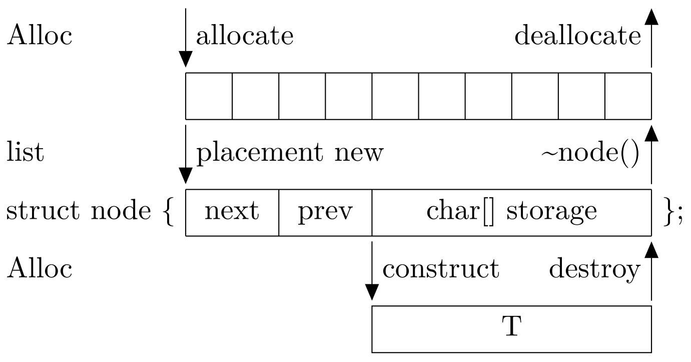
```
template<typename U, typename... Args>
void construct(U* const p, Args&&... args)
```

- in-place construction
  - copy replaced by forwarded arguments
  - used in "emplace_back" of containers
- arguments may be changed
- allocator<T> constructs any U
  - "U" is independent template parameter
  - see "changed functionality" on next slide

Some major changes have been made on the construct function. With variadic templates and perfect forwarding the construct function has become a template on its own and is able to construct objects in-place using any accessible constructor. Additionally the construct method now takes a pointer to any type, not just to the value type of the allocator. This enables users to construct any type from a single allocator. This seems counterintuitive but serves a specific purpose. The impact of this change is shown on the next slide.

The fact that the constructor arguments are no longer limited permits the allocator to actually change them at will. The C++11 scoped_allocator_adaptor will do just that. It is presented in the next section.

Changed functionality with std::list — Slide 40



Let's go back to the linked list and see what the changed allocator does to it. The rebinding of the allocator type is as before: the list creates an allocator for its nodes. The list uses it to allocate memory for a node, as before, too. But then there's the change: the list will no longer ask the allocator to construct the node. Instead it will do so itself via placement new.

The containers are no longer permitted to construct their private helper types via the allocator. Only the element type may and must be constructed by the allocator. That's why the node now only contains a POD[1] in the size of the element instead of directly having the element as a member. The allocator is asked to construct the element within the POD. This is the point where the node allocator is used to construct the element type. It's the point where the new freedom of the construct function is required.

This adds a third step to the list node construction and makes using the new allocators more complex compared to C++03. And this third step also has to be made during deconstruction: destroy the element via the allocator, destroy the node object directly, and then free the memory via the allocator. The code for the list example is shown in the appendix, see page 24.

Overview of changes Slide 41

- allocators no longer "optional"
- simpler to write own allocator
- "construct" was improved
- defined container copy/assign/swap/move semantics
- usage more complex

The most important change is that allocators are now required by the standard. There is no option to leave them out. This requires all vendors to adjust their library implementation accordingly and allows custom allocators to be used more often. Implementing own allocators has become quite easy, too.

One major change is the construct method and its usage. Containers need to adapt to the new protocol: only their element type may be constructed using the allocator. Additionally you gain the flexibility of in-place construction by perfectly forwarding any number of arguments.

In addition the allocator can now define its "propagation" semantics on container operations such as copy, swap, and move. The default is not to propagate the allocator—just as it's done in C++03.

The new standard has refurbished the allocator concept and made some major improvements. We will wait for compilers and library implementations to adapt. The most advanced one are probably GCC 4.8 using libstdc++ and Clang 3.3 using libc++. You can use it to experiment on your own.

[1] A char array in the diagram, but there is also the C++11 std::aligned_storage.

3.2 Scoped allocator

Recap: nested containers problem Slide 43

```
ipclist->push_back(”abc”); //failure
ipclist->emplace_back(”abc”, alloc); //ok

ipclist->emplace_back(”abc”, ipclist->get_allocator());
```

suppose IPCStringListMapVector:

- push_back takes constructed element
- must pass allocator
- cumbersome

Let’s recap on the nested containers. In a list of strings we had a compile time error when using push_back with a string literal. This is because of the implicit construction of a string object without an allocator given. The family of emplace functions forwards any arguments to the string constructor. Here we can easily pass the correct allocator instance.

The allocator to pass to the string can be converted from the allocator that the list already has. If we are in a distant function and we are passed a list of strings we can just take the allocator instance from the list and construct a string with it. Passing in the allocator for all elements becomes quite cumbersome. Suppose filling a nested data structure of type vector of maps from X to list of strings. To help here the C++11 is equipped with an allocator adaptor.

New allocator adaptor Slide 44

```
#include <scoped_allocator>

template<typename OuterA, typename... InnerAllocs>
class scoped_allocator_adaptor
    : public OuterA
{ ... };
```

- adaptor: wrap and replace “outer allocator”
- pass through most functions
- augment construct function

The scoped allocator adaptor takes an existing allocator and augments it. The wrapped allocator is given as the first template argument and is called the outer allocator. It will become a public base class of the adaptor and therefore can be used to substitute the outer allocator in many places.

The adaptor will mostly behave just like the wrapped class except for the construct function. The construct function is still calling the outer allocator but it may modify the constructor arguments as is shown on the next slides.

List of strings, how it should work — Slide 45

```
using namespace boost::interprocess;

// (1st) WOULD wrap StringAlloc:
typedef std::scoped_allocator_adapter<
    StringAlloc //Boost allocator here
    > ScopedAlloc;

// (2nd) WOULD define new list type:
typedef std::list< //use std here
    IPCString,
    ScopedAlloc //allocator here
    > IPCScopedList;
```

The adaptor is meant to wrap and replace an existing allocator. Suppose the Boost.Interprocess allocator could already be used with a conforming C++11 compiler[2]. You would instantiate the adaptor with the IPC allocator for strings to produce the type ScopedAlloc. This would go into the instantiation of a standard list template to produce the type IPCScopedList. On this list we can now examine the construction of a new element.

Influence object construction — Slide 46

```
     IPCScopedList::emplace_back(”abc”)
                     ⇩
       ScopedAlloc::construct(p, ”abc”)
                     ⇩
  StringAlloc::construct(p, ”abc”, *this)
                     ⇩
        new(p)IPCString(”abc”, *this)
```

- list uses wrapped allocator

[2]Clang 3.3 with libc++ supports the scoped allocator adaptor, but with the Boost allocator it wouldn’t compile. Boost brings its own scoped allocator as part of Boost.Containers.

- injection of self
- just requires correct container types
- infinite nesting

Suppose you have created a scoped list instance and provided it with a shared memory allocator instance. The adaptor would easily construct from the given instance and just initialise its base class with it. When you now emplace a new string at the back of the list you call emplace_back and pass it a string literal. The emplace_back function will invoke the allocator construct method.

Here the magic happens: the allocator is the adaptor in this case and it will pass construction on to its outer allocator, but append itself to the end of the arguments. This way the outer (original) allocator is instructed to construct the string from a literal and an allocator instance. Because the allocator instance is derived from the original allocator class the string will happily take it to initialise its own allocator. And voilà, every thing works out.

When you create all containers with the scoped allocator you can arbitrarily nest them and gain the convenience on every nesting level. Only the top level container needs to be explicitly be constructed with an allocator instance. The nested containers will automatically be provided the allocator.

4 Appendix

If there is time left or if there are any questions:

4.1 Allocator 03—details

Allocators 03 Slide 51

template<**typename** `T`> **struct** `allocator` with:

- 7 typedefs
- 1 nested template
- 2+1 constructors
- 7 functions
- 1 specialisation
- 2 operators

Even if Stepanov says the machine dependent part of the STL fits into 16 lines of code a custom allocator will contain lots of stuff. An allocator contains several typedefs and functions which are accompanied by a few constructors and operators, a template specialisation, and a nested class template. All the individual parts are listed on the next slide.

Overview Slide 52

```
value_type, size_type, difference_type // types
(const_)pointer, (const_)reference
template rebind, allocator<void>

address, max_size // functions
allocate / deallocate
construct / destroy

allocator(); // constructors
allocator(allocator const&);
allocator(allocator<U> const&);

operator ==, operator != // operators
```

In this overview the many parts of an allocator in C++03 become apparent. Most of this is boilerplate code as it won't differ from the std::allocator. Depending on the task you'll typically change the pointer type, the allocate/deallocate, or the construct/destroy function pairs.

The address function takes a "reference" and gives you a "pointer" to the object. Here "reference" and "pointer" are the types of the allocator. The max_size function returns a size_type and gives you an upper limit for the number of objects that could reasonably be allocated at once using allocate. It only gives the information that for larger numbers the allocation will fail. It will not guarantee that for lower values the allocation will succeed.

4.2 std::list 03 using an allocator

Initialise std::list 03 with allocator Slide 54

```
template<typename T, typename Alloc>
struct list {
  struct node { ... };

  typedef Alloc::rebind<node>::other
    allocator_type;
  allocator_type m_Allocator;
  ...
  list::list(Alloc const& a)
    : m_Allocator(a) // converts allocator
  { ... }
};
```

The list takes an allocator type as a template argument. It will rebind the allocator to its own node type and store an instance of the rebound allocator as a member. This member will be initialised by the list constructors. Lots of them take an allocator object and initialise the member with it. This will invoke the conversion constructor of the allocator.

Note on the slide some keywords are missing. The correct syntax for rebinding the allocator type is:

```
typedef typename Alloc::template rebind<node>::other allocator_type;
```

Elements in std::list 03 with allocator Slide 55

```
node& list::createNode(T const& t) {
  //malloc:
  pointer p = m_Allocator.allocate(1);

  //construct node from t:
  m_Allocator.construct(p, node(t));

  //done
  return *p;
}
```

Suppose the list had a private helper function called createNode to construct a list node using the allocator. It could be implemented as shown. First it would allocate storage for a single node object. The allocate function returns the type “pointer” defined by the allocator. This pointer can be passed to the construct method which would then copy construct a node. Dereferencing the pointer afterwards yields a reference to the new node.

4.3 std::list 11 using an allocator

Initialise std::list 11 with allocator Slide 57

```
template<typename T, typename Alloc>
struct list {
   struct node { ... };

   typedef std::allocator_traits<Alloc>
      ::rebind_traits<node> atraits;

   atraits::allocator_type m_Allocator;
   ...
   list(Alloc const& a)
      : m_Allocator(a) // converts allocator
   { ... }
};
```

In C++11 the use of allocators has changed. The list accesses all functionality of the allocator via the allocator_traits. This is why it will first get hold of the correct traits type: traits for the allocator rebound to list nodes. Rebinding will happen via template aliases now. The rest stays the same: the list stores an allocator object and initialises it in its constructors.

Elements in std::list 11 using allocator Slide 58

```
node* list::createNode(T t) {
  //malloc:
  pointer raw = atraits::allocate(m_Allocator, 1);
  node* pNode = &*raw;

  //construct node:
  new(pNode) node;

  //construct T in node:
  T* pT = reinterpret_cast<T*>(pNode->storage);
  atraits::construct(m_Allocator, pT, std::move(t));

  return pNode;
}
```

The use of the new allocators is more complex as in C++03. The list still needs to allocate memory for one node object. It will receive a "pointer" as defined by the allocator. The odd looking "&*raw" will give you a plain node pointer which the lists uses to construct its node in place.

The node will not immediately construct the element type, but instead provide storage for it. It casts the pointer to the storage into the pointer to the element type so it can call construct. The construct function is from the traits and takes the allocator as the first argument. The traits will just pass everything on to the allocator.

Note how the list will not need to create a temporary node object. The node and the actual element are constructed independently of each other.

4.4 Propagating allocators 11

Propagation — Slide 60

Allocator defines ...

- propagate_on_container_copy_assignment
- propagate_on_container_move_assignment
- propagate_on_container_swap
  - typedef for std::""false_type or true_type
  - all false by default
- select_on_container_copy_construction()
  - function to determine new allocator object
  - usually this is the identity function

The propagation semantics determine how a container shall propagate its allocator instance in case the container is copied, swapped, moved or assigned. In the case of copy construction the allocator can provide any new allocator instance. The function select_on_container_copy_construction can deliver any instance, but usually it just gives a copy of itself (identity). This function is just a hook method.

For the other cases it's sufficient to specify whether to propagate or not. That's defined by compile time boolean values. A container evaluates them and acts accordingly. The standard offers the classes std::false_type and std::true_type for this purpose.

```
template<typename T> struct allocator {
    ...
    typedef std::true_type propagate_on_container_copy_assignment;
    allocator
        select_on_container_copy_construction() const;
};
```

Implementing propagation — Slide 61

```
template<typename T, typename Alloc>
list<T, Alloc>&
list<T, Alloc>::operator = (list&& other) //move assign
{ ...
    if(atraits::propagate_on_container_move_assignment
            ::value)
        m_Allocator = other.get_allocator();
    ...
    return *this;
};
```

The container has the responsibility to evaluate the propagation description of an allocator and act accordingly. For a move assignment operator in a list this could look like shown (incomplete code though). A simple "if" statement can be used to decide on the compile time constant. When to propagate the list copies over the allocator. Here no move semantics is possible because get_allocator returns by value. There is no way to directly change the allocator of an existing container.

4.5 Source code: strings in shared memory

```
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/containers/string.hpp>
#include <boost/interprocess/containers/list.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/container/scoped_allocator.hpp>

#include <iostream>
#include <ostream>

//#include <scoped_allocator>
//#include <list>

using namespace boost::interprocess;

int main()
{
    managed_shared_memory segment(open_or_create, "ipcsample", 65536u);

    typedef allocator<
        char,
        managed_shared_memory::segment_manager
        > CharAlloc;

    typedef basic_string<
```

```
        char,
        std::char_traits<char>,
        CharAlloc //allocator here
        > IPCString;

    //make allocator instance:
    CharAlloc achar(segment.get_segment_manager());

    //construct, pass allocator:
    IPCString *str = segment.construct<IPCString>
        ("app title") //name of the object
        ("hello", achar); //ctor arguments

    std::cout << *str;
    *str = " world"; //access string
    std::cout << *str;

    typedef allocator<
        IPCString,
        managed_shared_memory::segment_manager
        > StringAlloc;

    typedef list<
        IPCString,
        StringAlloc //allocator here
        > IPCStringList;

    //make allocator instance:
    StringAlloc astr(achar);

    //construct, pass allocator:
    IPCStringList *strlist = segment.construct<IPCStringList>
        ("title list") //name of the object
        (astr); //ctor arguments

    strlist->push_back(*str); //access list

    //list->push_back("abc"); //failure?!?
    strlist->push_back(IPCString("abc", achar));
    strlist->emplace_back("abc", achar); //ok

    // using scoped allocator:
    typedef boost::container::scoped_allocator_adaptor<StringAlloc>
        ScopedStringAlloc;
    typedef list<IPCString, ScopedStringAlloc> ScopedStringList;

    //make allocator instance:
    ScopedStringAlloc ascoped(astr);

    //construct, pass allocator:
    ScopedStringList* scopedlist = segment.construct<ScopedStringList>
        ("scoped") //name of the object
        (ascoped); //ctor arguments

    scopedlist->emplace_back(*str);
    scopedlist->emplace_back("world"); // works now

/* For the future:
    typedef std::scoped_allocator_adaptor<StringAlloc> ScopedStringAlloc;
    typedef std::list<IPCString, ScopedStringAlloc> ScopedStringList;

    //make allocator instance:
    ScopedStringAlloc ssa(astr);

    //construct, pass allocator:
    ScopedStringList* ssl = segment.construct<ScopedStringList>
        ("scoped") //name of the object
        (ssa); //ctor arguments

    ssl.emplace_back("abc"); //works
*/
}
```