BSL - Bloomberg Standard Library

Stefano Pacifico

stefano.pacifico@ieee.org

21 December 2012

Outline

Everything should be made as simple as possible, but no simpler – Albert Einstein

Introduction

- What is BSL
- What is Bloomberg L.P.

Populating A Vector

```
static const std::size_t LENGTH = 100000;
const char *array[LENGTH] = { // 100000 strings }
//...
std::vector<std::string> v;
for(std::size_t i = 0; i < LENGTH; i++) {
    v.push_back(array[i]);
}
```

Populating A Vector

```
static const std::size_t LENGTH = 100000;
const char *array[LENGTH] = { // 100000 strings }
//...
std::vector<std::string> v;

v.reserve(LENGTH);  // This improves things a bit.

for(std::size_t i = 0; i < LENGTH; i++) {
    v.push_back(array[i]);
}
```

Inserting And Removing From A `map`

```
std::map<const char *, double> askPrices;  //stock, pri

// ...
orders.insert(std::make_pair(key, value));
// ...
orders.remove(key);
```

Dynamic Memory Allocation

- General-purpose memory allocators do not offer optimal performance in all scenarios, and can be insufficient in some.

Dynamic Memory Allocation

- General-purpose memory allocators do not offer optimal performance in all scenarios, and can be insufficient in some.
- Memory allocation can be a bottleneck in high-performance environments.

Memory Allocation Concerns (In Embedded System)

Memory allocation can generate other concerns (especially for embedded systems, such robots).

Memory Allocation Concerns (In Embedded System)

Memory allocation can generate other concerns (especially for embedded systems, such robots).

- Fragmentation.

Memory Allocation Concerns (In Embedded System)

Memory allocation can generate other concerns (especially for embedded systems, such robots).

- Fragmentation.
- Availability

Memory Allocation Concerns (In Embedded System)

Memory allocation can generate other concerns (especially for embedded systems, such robots).

- Fragmentation.
- Availability
- Exceptions.

Memory Allocation Concerns (In Embedded System)

Memory allocation can generate other concerns (especially for embedded systems, such robots).

- Fragmentation.
- Availability
- Exceptions.
- Lifetime management.

Standard Allocator

STL containers can be parametrized with an `Allocator` type:

Standard Allocator

STL containers can be parametrized with an `Allocator` type:

Example (Vector With `MyAllocatorType`)

```
std::vector<int, MyAllocatorType> v;
```

Standard Allocator

```
template <class T>
class MyAllocator {
  public:
    // A bunch of typedefs
    // ...
    // A bunch of methods
    //...

    pointer  allocate(size_type n, const void * = 0);
    void     deallocate(void* p, size_type);
    void     construct(pointer p, const T& val);
    void     destroy(pointer p);

    //...
    // A bunch of other methods
};
```

Standard Allocator Problems

```
std::vector<T, A1> v1;
std::vector<T, A2> v2;
```

Standard Allocator Problems

```
std::vector<T, A1> v1;
std::vector<T, A2> v2;
```

Assignment

```
v1 = v2;
```

Standard Allocator Problems

```
std::vector<T, A1> v1;
std::vector<T, A2> v2;
```

Assignment

```
v1 = v2;
```

Standard Allocator Problems

```
std::vector<T, A1> v1;
std::vector<T, A2> v2;
```

Assignment

```
v1 = v2;
```

Comparison

```
if (v1 == v2) ...
```

Standard Allocator Problems

```
std::vector<T, A1> v1;
std::vector<T, A2> v2;
```

Assignment

```
v1 = v2;
```

Comparison

```
if (v1 == v2) ...
```

Standard Allocator Problems

```
std::vector<T, A1> v1;
std::vector<T, A2> v2;
```

Assignment

```
v1 = v2;
```

Swap

```
v1.swap(v2) ...
```

Comparison

```
if (v1 == v2) ...
```

Standard Allocator Problems

```
std::vector<T, A1> v1;
std::vector<T, A2> v2;
```

Assignment

```
v1 = v2;
```

Swap

```
v1.swap(v2) ...
```

Comparison

```
if (v1 == v2) ...
```

Even worse!

Function signatures...

```
void foo(const std::vector<T>& v);
```

Even worse!

Function signatures...

```
void foo(const std::vector<T>& v);
```

... do not play nice with standard allocators.

```
std::vector<T, A> v2;
void foo(v2);
```

Even worse!

Function signatures...

```
void foo(const std::vector<T>& v);
```

... do not play nice with standard allocators.

```
std::vector<T, A> v2;
void foo(v2);
```

Other Possible Issues

- Contention among threads.

Other Possible Issues

- Contention among threads.
- Code bloat.

`bsl` Polymorphic Allocator

- `bsl` offers the support for polymorphic allocators.

`bsl` Polymorphic Allocator

- `bsl` offers the support for polymorphic allocators.
- A polymorphic allocator is an allocator class that derives from `bslma::Allocator` (pure abstract class).

`bsl` Polymorphic Allocator

- `bsl` offers the support for polymorphic allocators.
- A polymorphic allocator is an allocator class that derives from `bslma::Allocator` (pure abstract class).
- All the types that allocate memory in `bsl` accept a `bslma::Allocator *` in their constructors.

`bsl` Polymorphic Allocator

- `bsl` offers the support for polymorphic allocators.
- A polymorphic allocator is an allocator class that derives from `bslma::Allocator` (pure abstract class).
- All the types that allocate memory in `bsl` accept a `bslma::Allocator *` in their constructors.
- Allocators become part of the state of an object, no longer of its type!

bsl Containers and bslma::Allocator

```
class MyAllocator1 : public bslma::Allocator {
//...
};
class MyAllocator2 : public bslma::Allocator {
//...
};
//...
bsl::vector<int> v1(myAllocator1Ptr);
bsl::vector<int> v2(myAllocator2Ptr);
```

`bsl` Containers and `bslma::Allocator`

`bsl` Containers and `bslma::Allocator`

Assignment

```
v1 = v2;
```

`bsl` Containers and `bslma::Allocator`

Assignment

```
v1 = v2;
```

`bsl` Containers and `bslma::Allocator`

Assignment

```
v1 = v2;
```

Comparison

```
if (v1 == v2) ...
```

`bsl` Containers and `bslma::Allocator`

Assignment

```
v1 = v2;
```

Comparison

```
if (v1 == v2) ...
```

bsl Containers and bslma::Allocator

Assignment

```
v1 = v2;
```

Swap

```
v1.swap(v2) ...
```

Comparison

```
if (v1 == v2) ...
```

`bsl` Containers and `bslma::Allocator`

Assignment

```
v1 = v2;
```

Swap

```
v1.swap(v2) ...
```

Comparison

```
if (v1 == v2) ...
```

bsl Containers and bslma::Allocator

Assignment

```
v1 = v2;
```

Comparison

```
if (v1 == v2) ...
```

Swap

```
v1.swap(v2) ...
```

... and it plays nice with function signatures

```
bsl::vector<T> v2(
                myPtrAllocator2);
void foo(v2);
```

bsl Containers and bslma::Allocator

Assignment

```
v1 = v2;
```

Comparison

```
if (v1 == v2) ...
```

Swap

```
v1.swap(v2) ...
```

... and it plays nice with function signatures

```
bsl::vector<T> v2(
                myPtrAllocator2);
void foo(v2);
```

Let's Review

Valid Syntax	STD Allocator	`bsl` Allocator
`v1 = v2;`	NO	YES
`if (v1 == v2)...`	NO	YES
`v1.swap(v2);`	NO	YES
`foo(v2);`	NO	YES

Useful Custom Allocators

- Buffered sequential allocators
- Arena allocators
- Pooled allocators
- Many more!

References

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1850.pdf
- http://www.cs.umass.edu/ẽmery/pubs/berger-oopsla2002.pdf

Conventions

- A *component* is the unit of both physical and logical design (e.g., `bslma_allocator`).

Conventions

- A *component* is the unit of both physical and logical design (e.g., `bslma_allocator`).

- A component consists in a pair of `.h`/`.cpp` files and a `.t.cpp` test driver file (see Physical Code Organization).

Conventions

- A *component* is the unit of both physical and logical design (e.g., `bslma_allocator`).
- A component consists in a pair of `.h`/`.cpp` files and a `.t.cpp` test driver file (see Physical Code Organization).
- Components live in *packages*; Packages live in *package groups* (e.g., `bsl`). Code lives in its package namespace (e.g., `bslma::`).

Conventions

- A *component* is the unit of both physical and logical design (e.g., `bslma_allocator`).
- A component consists in a pair of `.h`/`.cpp` files and a `.t.cpp` test driver file (see Physical Code Organization).
- Components live in *packages*; Packages live in *package groups* (e.g., `bsl`). Code lives in its package namespace (e.g., `bslma::`).
- See Coding Standards for more information.

Fibonacci Function

Classic recursive implementation:

```
int fibonacci(int n)
{
    if (n == 0 || n == 1) {
        return 1;
    }

    return fibonacci(n - 2) + fibonacci(n - 1);
}
```

Fibonacci Function

What does this do?

```
unsigned int result = fibonacci(-1);
```

Fibonacci Function

What does this do?

```
unsigned int result = fibonacci(-1);
```

- What are the pre-conditions and post-conditions of this function?

Fibonacci Function

What does this do?

```
unsigned int result = fibonacci(-1);
```

- What are the pre-conditions and post-conditions of this function?
- How can we **know**?

Contracts

- Contracts tell us what the preconditions are.

Contracts

- Contracts tell us what the preconditions are.
- Contracts tell us more!

Contracts

```
virtual void *allocate(size_type size);
    // Return a newly allocated block of memory of (at
    // least) the specified positive 'size' (in
    // bytes).  If 'size' is 0, a null pointer is
    // returned with no other effect.  If this
    // allocator cannot return the requested number of
    // bytes, then it will throw a 'std::bad_alloc'
    // exception in an exception-enabled build, or
    // else will abort the program in a non-exception
    // build.  The behavior is undefined unless '0 <=
    // size'.  Note that the alignment of the address
    // returned conforms to the platform requirement
    // for any object of the specified 'size'.
```

Out Of Contract Function Calls

- What should a function do when invoked out of contract?

Out Of Contract Function Calls

- What should a function do when invoked out of contract?
- Do nothing?

Out Of Contract Function Calls

- What should a function do when invoked out of contract?
- Do nothing?
- Print to standard output?

Out Of Contract Function Calls

- What should a function do when invoked out of contract?
- Do nothing?
- Print to standard output?
- Throw an exception?

Out Of Contract Function Calls

- What should a function do when invoked out of contract?
- Do nothing?
- Print to standard output?
- Throw an exception?
- Abort?

Out Of Contract Function Calls

- The caller should be informed that something went wrong. This is *different than error handling*.

Out Of Contract Function Calls

- The caller should be informed that something went wrong. This is *different than error handling*.
- Contracts should be written specifying undefined behavior as the outcome of pre-conditions violations.

Out Of Contract Function Calls

- `BSLS_ASSERT` can be used, when possible, to detect undefined and report it.

Out Of Contract Function Calls

- `BSLS_ASSERT` can be used, when possible, to detect undefined and report it.

- `bslma::Assert::failAbort` is the default *assert handler*.

Out Of Contract Function Calls

- `BSLS_ASSERT` can be used, when possible, to detect undefined and report it.
- `bslma::Assert::failAbort` is the default *assert handler*.
- `bslma::Assert::failThrow` and `bslma::Assert::failSleep` are available in the library, but you can write your own!

What Does BSL Offer?

- runtime polymorphic memory allocators

What Does BSL Offer?

- runtime polymorphic memory allocators
- enhanced type trait support and trait-aware container facilities

What Does BSL Offer?

- runtime polymorphic memory allocators
- enhanced type trait support and trait-aware container facilities
- defensive programming through a configurable assert facility

What Does BSL Offer?

- runtime polymorphic memory allocators
- enhanced type trait support and trait-aware container facilities
- defensive programming through a configurable assert facility
- a facility for testing generic container

What Does BSL Offer?

- runtime polymorphic memory allocators
- enhanced type trait support and trait-aware container facilities
- defensive programming through a configurable assert facility
- a facility for testing generic container
- implementations for basic atomic operations

What Does BSL Offer?

- runtime polymorphic memory allocators
- enhanced type trait support and trait-aware container facilities
- defensive programming through a configurable assert facility
- a facility for testing generic container
- implementations for basic atomic operations
- a methodology for writing high-quality, reusable software

What Does BSL Offer?

- runtime polymorphic memory allocators
- defensive programming through a configurable assert facility

What Does BSL Offer?

- runtime polymorphic memory allocators
- defensive programming through a configurable assert facility

What Does BSL Offer?

- runtime polymorphic memory allocators

What Else Is Available Today In `bsl`

- Meta-functions for template programming (see `bslmf`)
- `-DBSL_OVERRIDES_STD` build flag (see BSL and STL)
- More! Explore the source and documentation on Github.

What Will Be Available Soon

- Date and calendars
- Timezones
- Concurrency

How To Get Involved

- `git clone` the libraries from Github
- Read https://github.com/bloomberg/bsl/wiki/Contributing-to-BSL and submit your *Individual Contributor License Agreement*.
- Start hacking away and submit your pull requests!
- www.spacifico.org - `stefano.pacifico at ieee.org`

Questions?