Tutorials

A set of tutorials is demonstrating the use of FeatStd library.

The FeatStd API documentation is added to provide full traceability between the tutorials and the used FeatStd API - refer to the sub page 'API Documentation'.

- Diagnostics
- MemoryPool

Diagnostics

Description

This tutorial describes several aspects of diagnostic support during development and post-development phases.

Assertions

For using assertions include *FeatStd/Diagnostics/Debug.h* in your program.

Compile Time Checks

Best time to catch an error is at compile time before the program starts. If the assertion is a constant expression, then the compiler is able to check it.

Use

- FEATSTD_COMPILETIME_ASSERT(assertion)

Runtime Checks

These kinds of assertions should be used to document logically impossible situations and discover programming errors.

Use

- FEATSTD_DEBUG_ASSERT(condition)
- FEATSTD_DEBUG_FAIL()
- FEATSTD_DEBUG_REENTRANCE_GUARD()
- FEATSTD_DEBUG_BREAK()

FEATSTD_DEBUG_ASSERT() and FEATSTD_DEBUG_FAIL() allow the user to check **pre-conditions** and also **post-conditions** as well as **invariants**, sometimes known as "embedded testing".

```
bool CgiApp::LoadAsset(const Char* filePath)
{
    FEATSTD_DEBUG_ASSERT(filePath != 0);

    ClearLoadedAsset();
    m_assetConfig.AddFileRepository(filePath);
```

```
                if ((*tempName >= '0') && (*tempName <= '9')) {
                    hash += (ToUInt32(*tempName) - static_cast<UInt32>('0'));
                }
                else if ((*tempName >= 'A') && (*tempName <= 'F')) {
                    hash += (ToUInt32(*tempName) - (static_cast<UInt32>('A') - 10U));
                }
                else if ((*tempName >= 'a') && (*tempName <= 'f')) {
                    hash += (ToUInt32(*tempName) - (static_cast<UInt32>('a') - 10U));
                }
                else {
                    // failure
                    FEATSTD_DEBUG_FAIL();
                }
```

FEATSTD_DEBUG_REENTRANCE_GUARD() allows to check that a part of code isn't entered twice:

```
void Appender::Append(const LogEvent& logEvent)
{
    FEATSTD_DEBUG_REENTRANCE_GUARD();
```

Be aware that all FEATSTD_**DEBUG**_... macros are expanded in Debug-Versions only. This avoids codesize and performance penalty in Release-Versions.

For a distinction between ErrorHandling and Assertions please see

http://en.wikipedia.org/wiki/Assertion_(computing)

Customizing

The behavior can be adapted by implementing

- either a platform specific FeatStd::Platform::Diagnostic class
- or a **DebugControlHook** class derived from FeatStd::Diagnostics::IDebugControlHook, activated via FeatStd::Diagnostics::DebugControl::SetHook.

See UnitTests\Diagnostics\DebugControlTest.cpp and the FeatStd::UnitTestHelper::DebugOutputChecker implementation for details.

Logging

Description

Logging is an important feature while programming (debugging), but also during the post-development phase to detect the reasons for system and software crashes.

In contrast to classical "printf" debugging, the FeatStd implementation is "two-dimensionally" configurable.

- First, the framework is structured in various **realms**.
- Second, for every realm the **level of detail** (FeatStd::Diagnostics::LogLevel::Enum, Candera::Diagnostics::LogLevel::Enum) can be chosen separately.

Free configurable **appenders** decide which channel(s) should be used to output the log messages.

See FeatStd::Diagnostics::Appender (and subclasses) for details.

Configure Log Output

Default Log Output

By default, Candera produces logging output with LogLevel "Warning" for all realms on the console. This chapter explains how to adapt those default logging configuration in an application.

To configure logging support include *Log.h* in your program.

Set LogRealm and LogLevel

To modify LogLevel for one, more, or even all realms, use

- FeatStd::Diagnostics::LogControl::SetLogLevel<LogRealm>().
- The convenience function FeatStd::Diagnostics::LogControl::SetLogLevel() sets the requested level for all realms.

For a list of all LogRealms supported by FeatStd, refer to *FeatStd/Diagnostics/LogRealm.h*:

```
FEATSTD_LOG_DECLARE_REALM(FeatStdSystem);
FEATSTD_LOG_DECLARE_REALM(FeatStdPlatform);
FEATSTD_LOG_DECLARE_REALM(FeatStdIO);
FEATSTD_LOG_DECLARE_REALM(FeatStdMemoryManagement);
FEATSTD_LOG_DECLARE_REALM(FeatStdMonitor);
FEATSTD_LOG_DECLARE_REALM(FeatStdAsync);
```

Use the macro FEATSTD_LOG_SET_REALM to activate a realm (see also chapter Custom Log Output), e.g. FeatStd class FileAppender will produce logging output under the realm "FeatStdSystem", because of the following declaration:

```
class FileAppender : public Appender
{
    typedef Appender Base;
```

```
    FEATSTD_LOG_SET_REALM(Diagnostics::LogRealm::FeatStdSystem);
```

For each realm different log levels can be set, see log levels defined in FeatStd/Diagnostics/Loglevel.h.

```
enum Enum
{
    All = 0,
    Debug = All,
    Info = 1,
    Warning = 2,
    Error = 3,
    Fatal = 4,
    Off = 5
};
```

Use FeatStd::Diagnostics::LogControl::SetLogLevel<LogRealm>() to set the desired log level.

```
    LogControl::SetAppender(&CgiAppLogAppender::GetCgiAppLogAppenderInstance());
    LogControl::SetLogLevel<CgiAppLog>(LogLevel::All);
```

> The default log level is *Warning*.

Output Channel - Log Appenders

Logging output is handled by instances of FeatStd::Diagnostics::Appender - for this purpose, all FeatStd::Diagnostics::Logger instances share a list of appenders.

FeatStd provides following appenders:

- FeatStd::Diagnostics::ConsoleAppender
- FeatStd::Diagnostics::FileAppender
- FeatStd::UnitTestHelper::MemorizeAppender

By default, a ConsoleAppender is configured, means once an application doesn't configure anything regarding Logging, log messages from Candera will be sent to the console.

The list of appenders can be modified by FeatStd::Diagnostics::LogControl::SetAppender, AppendAppender and RemoveAppender.

Implement a Custom Appender

A custom appender class has to be derived from FeatStd::Diagnostics::Appender and implement the abstract method FeatStd::Diagnostics::Appender.DoAppend().

A custom **CgiAppLogAppender** has been created for modifying the LogLevel::Info output.

```
void CgiAppLogAppender::DoAppend(const LogEvent& logEvent)
{
    // Info traces will be outputted as is, for all others keep ConsoleAppender implementation
    if (logEvent.mLogLevel <= LogLevel::Info) {
        FeatStd::Internal::Diagnostic::ConsoleOut("%s\n", logEvent.mMessage);
        FeatStd::Internal::Diagnostic::DebuggerOut("%s\n", logEvent.mMessage);
    } else {
        FeatStd::Diagnostics::ConsoleAppender::DoAppend(logEvent);
    }
}
```

This custom appender is set to LogControl and activated for all log levels.

```
    LogControl::SetAppender(&CgiAppLogAppender::GetCgiAppLogAppenderInstance());
    LogControl::SetLogLevel<CgiAppLog>(LogLevel::All);
```

Custom Log Output

Player Logging Support

This chapter explains how to create logging output from a custom application.

Include *Log.h* in your program.

Default Application Realm

FeatStd predefines a default application realm:

```
FEATSTD_LOG_DECLARE_REALM(User);
```

To activate this realm for an application, use the macro FEATSTD_LOG_SET_REALM() with the default application realm as parameter:

```
FEATSTD_LOG_SET_REALM(LogRealm::User);
```

Produce Logging Output

Output itself is done by calling level specific **FEATSTD_LOG_<level>** macros in your code:

- FEATSTD_LOG_DEBUG(...)
- FEATSTD_LOG_INFO(...)
- FEATSTD_LOG_WARN(...)
- FEATSTD_LOG_ERROR(...)
- FEATSTD_LOG_FATAL(...)

FEATSTD_LOG_<level> macros provide a **printf-like** interface, which means that a format string has to be given as first argument, followed by a variable number of arguments (referenced in the format string).

FEATSTD_LOG_INFO() example when loading a Scene.

```
bool CgiApp::LoadScene(const Char* sceneName) {
    FEATSTD_LOG_INFO(" -> Loading scene '%s'", sceneName);
```

FEATSTD_LOG_ERROR() example when loading a asset configuration fails.

```
    bool result =  LoadAssetConfig();
    if (!result) {
        FEATSTD_LOG_ERROR("\n%s\n", m_loadError);
        m_assetConfig.ClearRepositoryList();
    }
```

> The length of the assembled information **must be** less than 511 characters!

Define Custom Realms

For simple applications the usage of the predefined realm **LogRealm::User** may suffice but more sophisticated applications possibly want to structure the output by further realms.

To do so, first define a LogRealm class by using the FEATSTD_LOG_DECLARE_REALM(name_of_the_realm) macro.

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

namespace CgiApplication {

FEATSTD_LOG_DECLARE_REALM(CgiAppLog);
```

```
FEATSTD_LOG_DECLARE_REALM(TutorialWidgets);
```

See LightPlayer/CgiAppLogRealm.[h|cpp] for details.

> The new realm doesn't take part in any iteration functionality (e.g. FeatStd::Diagnostics::LogControl::SetLogLevel()) until the realm (Logger) is used a first time (instantiated).

So it is good practice to implement an initialization function (using FEATSTD_LOG_DEFINE_REALM(name_of_the_realm))

```
void CgiApp_LoggerInitialization()
{
    FEATSTD_LOG_DEFINE_REALM(CgiAppLog);
    FEATSTD_LOG_DEFINE_REALM(TutorialWidgets);
}
```

and to call it at an early point in program execution.

Finally, the new custom realms must be set for the application, which shall log on that realm:

```
FEATSTD_LOG_SET_REALM(CgiAppLog);
```

Use an appropriate **FEATSTD_LOG_<level>** macro to output the message.

MemoryPool

Description

FeatStd::MemoryPool is a dynamic memory management infrastructure with following high level features:

- Provide replacement for Standard C library heap
- Multi instance support
  - Dedicated MemoryPool instances for specific components
- Flexible operation modes
  - Separation of transient and permanent allocations
  - Enforce permanent allocations only (no fragmentation)
- Minimal runtime and resource overhead
- Multi threading support
- Debugging facilities
  - Memory guards to detect memory overwrites
  - Leak detection (dump currently allocated blocks with context information)
- Analysis support
  - Logging of heap operations
  - Statistics

Build Setup and MemoryPool Features

Build Setup

MemoryPool source code is located under src/FeatStd/MemoryPool. MemoryPool is an optional feature of FeatStd and can be enabled with the CMake flag FEATSTD_MEMORYPOOL_ENABLED.

MemoryPool Features

MemoryPool supports a number of diagnostic and debugging facilities. The facilities can be enabled or disabled at compile time. Following CMake variables can be used to configure MemoryPool features for specific build modes:

- FEATSTD_MEMORYPOOL_FEATURES_DEBUG
- FEATSTD_MEMORYPOOL_FEATURES_RELEASE
- FEATSTD_MEMORYPOOL_FEATURES_MINSIZEREL
- FEATSTD_MEMORYPOOL_FEATURES_RELWITHDEBINFO

Each of these variables can be set to enable one or more of the following features:

- FEATSTD_MEMORYPOOL_LEAK_DETECTION
- FEATSTD_MEMORYPOOL_STATISTICS
- FEATSTD_MEMORYPOOL_MEMGUARDS
- FEATSTD_MEMORYPOOL_ERASE_BUFFERS

A reasonable feature setup is:

- FEATSTD_MEMORYPOOL_FEATURES_DEBUG = (FEATSTD_MEMORYPOOL_LEAK_DETECTION | FEATSTD_MEMORYPOOL_STATISTICS | FEATSTD_MEMORYPOOL_MEMGUARDS | FEATSTD_MEMORYPOOL_ERASE_BUFFERS)
- FEATSTD_MEMORYPOOL_FEATURES_RELEASE = FEATSTD_MEMORYPOOL_MEMGUARDS
- FEATSTD_MEMORYPOOL_FEATURES_MINSIZEREL =
- FEATSTD_MEMORYPOOL_FEATURES_RELWITHDEBINFO = FEATSTD_MEMORYPOOL_MEMGUARDS

Using MemoryPool

Main MemoryPool Interfaces

Allocate And Deallocate Memory

The main interfaces to MemoryPool are following macros.

- FEATSTD_MEMORYPOOL_ALLOC
- FEATSTD_MEMORYPOOL_TRANSIENT_ALLOC
- FEATSTD_MEMORYPOOL_PERMANENT_ALLOC
- FEATSTD_MEMORYPOOL_REALLOC
- FEATSTD_MEMORYPOOL_GLOBAL_REALLOC
- FEATSTD_MEMORYPOOL_FREE
- FEATSTD_MEMORYPOOL_GLOBALFREE
- FEATSTD_MEMORYPOOL_NEW
- FEATSTD_MEMORYPOOL_TRANSIENT_NEW
- FEATSTD_MEMORYPOOL_PERMANENT_NEW
- FEATSTD_MEMORYPOOL_DELETE

- FEATSTD_MEMORYPOOL_ARRAY_NEW
- FEATSTD_MEMORYPOOL_ARRAY_NEW_WITH_PRESET
- FEATSTD_MEMORYPOOL_ARRAY_DELETE

The first parameter to most macros is MemRealm. As MemoryPool supports multiple instances (heaps), allocation requests always have to specify the pool the allocation should be performed from. MemRealm is the "name" of a specific MemoryPool that is defined in the system.

MemoryPool Declaration

To define a memory pool, following macros can be used:

- FEATSTD_MEMORYPOOL_DEFINE_MANAGED_MEMORYPOOL
- FEATSTD_MEMORYPOOL_DEFINE_MEMORYPOOL
- FEATSTD_DEFINE_MEMORYPOOLHEAP

MemoryPool Configuration

Following macros can be used to create a MemoryPool configuration:

- FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER
- FEATSTD_MEMORYPOOL_CONFIG
- FEATSTD_MEMORYPOOL_BIN_SPECIFICATION
- FEATSTD_MEMORYPOOL_BUFFER_PRESET

Usage Sample

Following code illustrates how to transient allocate 20 bytes of memory from a MemoryPool called MallocBackingHeapPool.

```
void *p = FEATSTD_MEMORYPOOL_ALLOC(MallocBackingHeapPool, MemAttrib::Transient, 20);
FEATSTD_MEMORYPOOL_FREE(MallocBackingHeapPool, p);
```

> please refer to later tutorial chapters (MemoryPool Sample Application) for instructions how to use these macros.

MemoryPool Sample Application

Introduction

The MemoryPool sample application shows several use cases and aspects of MemoryPool. Please refer to the sub sections below for more detail.

The sample application can be found in folder *featstd/src/MemoryPoolSample*.

Configuration of FeatStd::DefaultMemoryPool

Introduction

Once FeatStd::MemoryPool feature has been enabled in the build system, it is mandatory to initialize FeatStd::DefaultMemoryPool accordingly. Enabling FeatStd::MemoryPool feature without any further action will lead to an unresolved FeatStd::InitDefaultMemoryPool symbol error in the link step of the software build.

FeatStd::DefaultMemoryPool is the memory pool which FeatStd library components use to allocate memory from. Memory allocated with FEATSTD_NEW, FEATSTD_NEW_ARRAY, FEATSTD_NEW_PRESET_ARRAY, FEATSTD_ALLOC, and FEATSTD_REALLOC will use FeatStd::DefaultMemoryPool to allocate the memory. Same applies to the allocation interfaces in class FeatStd::MemoryManagement::PlatformHeap.

FeatStd::InitDefaultMemoryPool

The sample code shows how to implement FeatStd::InitDefaultMemoryPool and to configure FeatStd::DefaultMemoryPool. The code referenced in this tutorial can be found in FeatStd/src/MemoryPoolSample/FeatStdDefaultMemoryPool.cpp

Initializing a MemoryPool is in most cases a two stage process

- initialize the backing heap for the memory pool
- initialize the memory pool

First have a look at FeatStd::InitDefaultMemoryPool sample implementation:

```
namespace FeatStd {

    // ------------------------------------------------------------------------
    bool InitDefaultMemoryPool()
    {
        // FeatStd will invoke this function before main function is entered
        // initialize FeatStd::DefaultMemoryPool
```

```
        using namespace MemoryManagement;
        // IntrinsicBackingHeap defines a BackingHeap with an intrinsic memory buffer that
        // will be used by this heap. The size of the memory buffer is specified as template
        // parameter (here 1MB)
        typedef IntrinsicBackingHeap<DefaultMemoryPool, 1 * 1024 * 1024> Heap;

        // define the configuration for DefaultMemoryPool.
        static const MemoryPoolConfig config = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
            config,                             // name of the variable receiving the configurat
            "FeatStdDefaultMemoryPool",         // the name of the MemoryPool
            MemoryPoolFlags::PermanentOnly,     // Flags that control the memory pool behavior
            128,                                // all bins up to 128 bytes are small bins
            cPoolBinSpecs,                      // the bin configurations
            cBufferPresets,                     // preset allocations
            Heap,                               // type name of the backing heap
            &SampleApp::OnOutOfMemory,          // optional the out of memory function
            &OnDestroy                          // optional OnDestroy function
        );

        // initialize first the backing heap and then the memory pool associated with the backir
        return Heap::Init() && DefaultMemoryPool::Init(&config);
    }
}
```

The FeatStd::InitDefaultMemoryPool has to be implemented in namespace FeatStd. The function will be either invoked when FeatStd::DefaultMemoryPool receives the first allocation request or latest before main function is entered. This typically ensures that the function is executed in a single threaded environment, which in fact is a requirement for every MemoryPool initialization.

Memory allocations before main function may happen in the progress of C++ static objects construction.

Likewise FeatStd::DefaultMemoryPool will be initialized automatically, it will also be destroyed automatically at the latest possible point of time but not before main function has exit.

Back to FeatStd::InitDefaultMemoryPool sample implementation. First initialization step is to setup a Backing Heap. The sample implementation uses a

FeatStd::MemoryManagement::IntrinsicBackingHeap. This Backing Heap implementation defines its own intrinsic static buffer which will be used as heap. The heap is configured with 1MB RAM and "personalized" for FeatStd::DefaultMemoryPool.

The second step is to create the FeatStd::InitDefaultMemoryPool configuration. A FeatStd::MemoryPool is configured at runtime with a

FeatStd::MemoryManagement::MemoryPoolConfig structure. The FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER or FEATSTD_MEMORYPOOL_CONFIG macros should be used to initialize this structure correctly.

The final step of FeatStd::InitDefaultMemoryPool implementation is to execute the FeatStd::MemoryManagement::IntrinsicBackingHeap and FeatStd::DefaultMemoryPool init functions.

MemoryPool Configuration

In order to configure MemoryPools, the allocation footprint of the application needs to be analysed. Various MemoryPool diagnostic tools and statistics support in this process. A rough outline of the process is

- Instrumentation of the application with StatisticsFileDumper and PermanentBlockFileDumper
- Adaptation of StatisticsFileDumper and PermanentBlockFileDumper to the platform capabilities (e.g. if no file system is available)
- Stage 1 - permanent block analysis with a minimal configuration
- Use MemoryPoolStatisticsAnalysis.xlsm to analyse application allocation footprint
- Stage 2 - Create bin configuration according to results of the analysis

The single steps of the configuration process can also be seen from MemoryPoolSample/Configuration.cpp

See also:

How does the number of bins influence system performance?

Instrumentation of the application with StatisticsFileDumper and PermanentBlockFileDumper

MemoryPoolSample application not only shows the usage of MemoryPool, but also includes utility code that can be re-used in applications.

The classes PermanentBlockFileDumper and StatisticsFileDumper can be easily incorporated in any application

```
class PermanentBlockFileDumper {
    public:
        ~PermanentBlockFileDumper();

        template<typename MemRealm> static bool Dump(const FeatStd::Char *fileName) {
            PermanentBlockFileDumper pbd;
            FeatStd::MemoryManagement::ConcretePoolMgmtObject<MemRealm> pmo;
            return pbd.Init(fileName) &&
                   pmo.VisitBins(&BinVisitor, &pbd);
```

```
        }

        static bool Dump(const FeatStd::Char *fileName);

    private:
        FeatStd::FileHandle mFileHdl;

        PermanentBlockFileDumper();

        bool Init(const FeatStd::Char *fileName);

        static bool PoolVisitor(const FeatStd::MemoryManagement::PoolMgmtObject
 &pmo, void *userData);
        static bool BinVisitor(const FeatStd::MemoryManagement::BinMgmtObject
 &bmo, void *userData);
        static bool BlockVisitor(const FeatStd::MemoryManagement::BlockMgmtObject
 &bmo, void *userData);

        bool DoDump(const FeatStd::MemoryManagement::BlockMgmtObject &bmo);

        FEATSTD_MAKE_CLASS_UNCOPYABLE(PermanentBlockFileDumper);
};
```

```
class StatisticsFileDumper {
    public:
        ~StatisticsFileDumper();

        static bool Dump(const FeatStd::Char *poolStatsFileName, const FeatStd::Char *binStatsFi
            StatisticsFileDumper sfd;
            return sfd.Init(poolStatsFileName, binStatsFileName) &&
                   FeatStd::MemoryManagement::MemoryPoolManager::DumpStatistics
(&DumpPoolStatistics, &DumpBinStatistics, &sfd);
        }

        template<typename MemRealm> static bool Dump(const FeatStd::Char *poolStatsFileName, cor
            StatisticsFileDumper sfd;
            return sfd.Init(poolStatsFileName, binStatsFileName) &&
                   FeatStd::MemoryManagement::MemoryPoolManager::DumpStatistics<MemRealm>(&DumpF
        }

    private:
        FeatStd::FileHandle mHdlPoolStats;
        FeatStd::FileHandle mHdlBinStats;

        StatisticsFileDumper();

        bool Init
(const FeatStd::Char *poolStatsFileName, const FeatStd::Char *binStatsFileName);
```

```
        static bool DumpPoolStatistics(const FeatStd::MemoryManagement::PoolMgmtObject
 &pmo, void *userData);

        static bool DumpBinStatistics(const FeatStd::MemoryManagement::BinMgmtObject
 &bmo, void *userData);

        FEATSTD_MAKE_CLASS_UNCOPYABLE(StatisticsFileDumper);
};
```

Both classes will write diagnostic and statistic information in CSV format to text files. if the target platform does not support file system, the classes need to be adapted. It is also possible to run the analysis in host environment with certain limitations (differing platform abstraction layers, potential size differences of complex types due to different tool chains).

PermanentBlockFileDumper and StatisticsFileDumper use MemoryPool functionality that is only present when FEATSTD_MEMORYPOOL_STATISTICS_ENABLED and FEATSTD_MEMORYPOOL_LEAK_DETECTION_ENABLED is enabled.

Stage 1 - permanent block analysis with a minimal configuration

The first stage in MemoryPool configuration is to start off with a very simple configuration.

```
static const BinSpecification cStageOneBinSpecs[] = {
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 4),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1024),

    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 0)
};

// define the configuration for DefaultMemoryPool.
static const MemoryPoolConfig cStageOneConfig = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
    cStageOneConfig,                    // name of the variable receiving the configuration
    "ConfigSampleStageOneMemoryPool",   // the name of the MemoryPool
    MemoryPoolFlags::PermanentOnly,     // Flags that control the memory pool behavior
    4,                                  // all bins up to 128 bytes are small bins
    cStageOneBinSpecs,                  // the bin configurations
    0,                                  // preset allocations
    ConfigSampleHeap,                   // type name of the backing heap
    0,                                  // optional the out of memory function
    0                                   // no OnDestroy function
);
```

The goal of the permanent block analysis is to get an allocation profile of the application. Thus the configuration sets FeatStd::MemoryManagement::MemoryPoolFlags::PermanentOnly. This will enforce that allocation requests will be permanent allocations. Thus memory will never be returned to the backing heap, but blocks will be maintained in the bin free lists. MemoryPool supports to dump all permanent blocks of a memoryPool (this is exactly the main task of

PermanentBlockFileDumper).

Thus dumping the permanent blocks after running the application will give a good overview on the allocation profile of the application.

> each MemoryPool configuration must have at least one small block and one large block bin. A limit of 255 bins per MemoryPool applies.

Use MemoryPoolStatisticsAnalysis.xlsm to analyse applicaton allocation footprint

PermanentBlockFileDumper and StatisticsFileDumper dump data to three CSV formatted files.

```
    // use StatisticFileDumper to dump the statistics to files
    StatisticsFileDumper::Dump("MemoryPoolStatistics.csv", "MemoryPoolBinStatistics.csv");
```

```
    // dump permanent block list
    PermanentBlockFileDumper::Dump("PermanentBlockList.csv");
```

- MemoryPoolStatistics.csv contains the overall MemoryPool statistics
- MemoryPoolBinStatistics.csv contains the statistics of the bins
- PermanentBlockList.csv contains the list of permanent blocks

The CSV formatted files now can be imported to MemoryPoolStatisticsAnalysis.xlsm (can be found in supplement/MemoryPool folder)

Open the Excel document (enable external content to enable external data sources and macros). The Excel document contains 4 sheets

1. PoolAnalysis
2. PoolBinAnalysis
3. PermanentBlockAnalysis
4. ResourceReq

The first three sheets contain update buttons that will import data and update the Pivot tables.

- MemoryPoolStatistics.csv gets imported to sheet PoolAnalysis
- MemoryPoolBinStatistics.csv gets imported to sheet PoolBinAnalysis
- PermanentBlockList.csv gets imported to sheet PermanentBlockAnalysis

> the numbers shown in this tutorial origin from MemoryPoolSample. Numbers might vary from actual program output.

After importing the data sheet PermanentBlockAnalysis will show following Pivot table data. The Pivot table will group the permanent block list by BinIndex and BufferSize (and context, which is currently out of interest). Actually the list already represents an "optimal" configuration of bin sizes (Bin configuration).

BufferSize	BlockCount	BufferSize	BlocksSize/Diag	Overhead/Diag	BlockSize	Overhead	Allocated
4	7	28	56	28	140	112	7
12	5	60	100	40	160	100	5
16	4	64	96	32	144	80	4
20	7	140	196	56	280	140	7
24	4	96	128	32	176	80	4
28	8	224	288	64	384	160	8
32	8	256	320	64	416	160	8
36	8	288	352	64	448	160	8
40	6	240	288	48	360	120	6
44	7	308	364	56	448	140	7
48	2	96	112	16	136	40	2
52	9	468	540	72	648	180	9
56	6	336	384	48	456	120	6
60	3	180	204	24	240	60	3
64	11	704	792	88	924	220	11
68	2	136	152	16	176	40	2
88	1	88	96	8	108	20	1
108	1	108	116	8	128	20	1
116	2	232	248	16	272	40	2
120	1	120	128	8	140	20	1
124	1	124	132	8	144	20	1
200	1	200	208	8	220	20	1
244	1	244	252	8	264	20	1

276	1	276	284	8	296	20	1
284	2	568	584	16	608	40	2
296	1	296	304	8	316	20	1
300	1	300	308	8	320	20	1
312	1	312	320	8	332	20	1
340	1	340	348	8	360	20	1
344	1	344	352	8	364	20	1
352	2	704	720	16	744	40	2
356	1	356	364	8	376	20	1
380	1	380	388	8	400	20	1
392	1	392	400	8	412	20	1
396	1	396	404	8	416	20	1
404	1	404	412	8	424	20	1
420	1	420	428	8	440	20	1
472	1	472	480	8	492	20	1
496	1	496	504	8	516	20	1
504	1	504	512	8	524	20	1
508	1	508	516	8	528	20	1
1024	1	1024	1032	8	1044	20	1
1056	1	1056	1064	8	1076	20	1
1632	1	1632	1640	8	1652	20	1

Stage 2 - Create bin configuration according to results of the analysis

Transfer Excel data to source file

The first two columns of the table can be easily copied to source code with the clipboard. Visual Studio find & replace can be used to generate a Bin configuration from table of numbers.

```
// these are the first two columns of the Excel sheet PermanentBlockAnalysis (copied with Clipbo
// use Visual Studio Find&Replace with Regex to convert to bin configuration.
// Search string (regex): "(\d+)\t(\d+)"
// Replace string: "FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, $1), // $2"
static const BinSpecification cStageOneBinSpecs[] = {
4    7
12    5
16    4
20    7
```

```
24    4
28    8
32    8
36    8
40    6
44    7
48    2
52    9
56    6
60    3
64    11
68    2
88    1
108    1
116    2
120    1
124    1
200    1
244    1
276    1
284    2
296    1
300    1
312    1
340    1
344    1
352    2
356    1
380    1
392    1
396    1
404    1
420    1
472    1
496    1
504    1
508    1
1024    1
1056    1
1632    1
};
```

After applying regex search & replace, following bin configuration has been created:

```
static const BinSpecification cStageOneBinSpecs[] = {
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 4), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 12), // 5
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 16), // 4
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 20), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 24), // 4
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 28), // 8
```

```
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 32), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 36), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 40), // 6
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 44), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 48), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 52), // 9
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 56), // 6
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 60), // 3
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 64), // 11
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 68), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 88), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 108), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 116), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 120), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 124), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 200), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 244), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 276), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 284), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 296), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 300), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 312), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 340), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 344), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 352), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 356), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 380), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 392), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 396), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 404), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 420), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 472), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 496), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 504), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 508), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1024), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1056), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1632), // 1
};
```

Optimize bin configuration

The bin configuration contains now more than 40 bins. As each bin has a memory penalty of 12 bytes (see Excel sheet ResourceReq), the next step is to minimize the number of bins.

For this bins with a low number of buffers are identified. This is the reason to copy the BufferCount column from the Excel sheet too. After applying the regex search & replace, the number of buffers is in the comment after each FEATSTD_MEMORYPOOL_BIN_SPECIFICATION line. The bin with buffer size 48 has two blocks. The next larger bin has buffer size 52. Eliminating the bin with buffer size 48 would implies that the buffers would be allocated from the bin with buffer size 52. The additional overhead for the two 48 bytes buffers are 8 (2 * 4) bytes. This means that eliminating the bin with buffer size 48 will save 4 bytes of overall memory consumption (12 bytes bin overhead saved minus 8 bytes overhead for the two buffers that will move to the bin with buffer size 52).

The result of bin reduction can be seen below (eliminated bins are commented out).

```
static const BinSpecification cStageTwoBinSpecs[] = {
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 4), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 12), // 5
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 16), // 4
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 20), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 24), // 4
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 28), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 32), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 36), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 40), // 6
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 44), // 7
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 48), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 52), // 9
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 56), // 6
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 60), // 3
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 64), // 11
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 68), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 88), // 1
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 108), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 116), // 2
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 120), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 124), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 200), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 244), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 276), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 284), // 2
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 296), // 1
```

```
FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 300), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 312), // 1
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 340), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 344), // 1
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 352), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 356), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 380), // 1
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 392), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 396), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 404), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 420), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 472), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 496), // 1
//    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 504), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 508), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1024), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1056), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1632), // 1

    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 0)
};

// define the configuration for DefaultMemoryPool.
static const MemoryPoolConfig cStageTwoConfig = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
    cStageTwoConfig,                    // name of the variable receiving the configuration
    "ConfigSampleStageTwoMemoryPool",   // the name of the MemoryPool
    MemoryPoolFlags::PermanentOnly,     // Flags that control the memory pool behavior
    1056,                               // all bins up to 128 bytes are small bins
    cStageTwoBinSpecs,                  // the bin configurations
    0,                                  // preset allocations
    ConfigSampleHeap,                   // type name of the backing heap
    0,                                  // optional the out of memory function
    0                                   // no OnDestroy function
);
```

Create Buffer Presets

The next step is optional and configures the buffer presets. With buffer presets permanent blocks can added to the bin free lists at MemoryPool initialization time. With buffer presets a specific buffer free list layout can be enforced.

Assume the application requires at system startup for whatever reason a 2040 byte buffer. This buffer will be freed as soon the system is setup. Later the application will allocate a 2048 byte buffer. The existing 2040 byte buffer is not suitable for the later 2048 byte allocation request and a new 2048 bytes buffer must be allocated from the BackingHeap. With buffer presets the allocation of a 2048 bytes buffer from BackingHeap can be enforced prior the allocation of the 2040 bytes

buffer. Thus the first allocation will use the preset 2048 bytes buffer, and the same buffer can be used for the succeeding later allocations.

After re-running the application with the optimized bin configuration and import of the new PermanentBlockList.csv into PermanentBlockAnalysis sheet, the first two columns (BufferSize and BufferCount) can be copied again to source code. Like in the prior steps, regex search & replace can be used to generate from this table the FeatStd::MemoryManagement::BufferPreset table. The table is then input to the final configuration.

```
static const BinSpecification cStageThreeBinSpecs[] = {
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 4), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 12), // 5
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 16), // 4
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 20), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 24), // 4
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 28), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 32), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 36), // 8
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 40), // 6
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 44), // 7
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 52), // 9
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 56), // 6
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 64), // 11
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 68), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 88), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 116), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 124), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 200), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 244), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 276), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 284), // 2
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 300), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 312), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 344), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 356), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 380), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 396), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 404), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 420), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 472), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 496), // 1
```

```
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 508), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1024), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1056), // 1
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1632), // 1

    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 0)
};

// based on re-imported PermanentBlockList.csv
// BufferSize / BufferCount columns
const BufferPreset cStageThreeBufferPresets[] = {
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(4, 7),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(12, 5),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(16, 4),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(20, 7),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(24, 4),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(28, 8),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(32, 8),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(36, 8),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(40, 6),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(44, 7),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(52, 11),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(56, 6),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(64, 14),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(68, 2),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(88, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(116, 3),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(124, 2),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(200, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(244, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(276, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(284, 2),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(300, 2),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(312, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(344, 2),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(356, 3),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(380, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(396, 2),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(404, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(420, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(472, 1),
```

```
FEATSTD_MEMORYPOOL_BUFFER_PRESET(496, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(508, 2),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(1024, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(1056, 1),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(1632, 1),

    // mandatory last entry
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(0, 0)
};

// define the configuration for DefaultMemoryPool.
static const MemoryPoolConfig cStageThreeConfig = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
    cStageThreeConfig,                  // name of the variable receiving the configuration
    "ConfigSampleStageTwoMemoryPool",   // the name of the MemoryPool
    MemoryPoolFlags::PermanentOnly,     // Flags that control the memory pool behavior
    1056,                               // all bins up to 128 bytes are small bins
    cStageThreeBinSpecs,                // the bin configurations
    cStageThreeBufferPresets,           // preset allocations
    ConfigSampleHeap,                   // type name of the backing heap
    0,                                  // optional the out of memory function
    0                                   // no OnDestroy function
);
```

Application Specific MemoryPool

MemoryPool declaration

One central feature of MemoryPool is the ability to define multiple MemoryPool instances. The instances are separated by "names" - the MemRealm.

Following statement defines a specific MemoryPool with the MemRealm "AppMemoryPool".

```
extern bool InitAppMemoryPool();

// define an application specific memory pool with realm AppMemoryPool
FEATSTD_MEMORYPOOL_DEFINE_MANAGED_MEMORYPOOL(AppMemoryPool, &InitAppMemoryPool);
```

The InitAppMemoryPool function is invoked when the first allocation on this pool is done or latest before main function will be executed.

This is the whole setup of an application specific MemoryPool. Macros as listed below can ease the use of the specific MemoryPool.

```
// macros to allocate and free memory from AppMemoryPool
#define APP_PERMANENT_ALLOC(nBytes) FEATSTD_MEMORYPOOL_PERMANENT_ALLOC(AppMemoryPool, nBytes)
#define APP_TRANSIENT_ALLOC(nBytes) FEATSTD_MEMORYPOOL_TRANSIENT_ALLOC(AppMemoryPool, nBytes)
#define APP_REALLOC(p, newSize) FEATSTD_MEMORYPOOL_REALLOC(AppMemoryPool, p, newSize)
#define APP_FREE(p) FEATSTD_MEMORYPOOL_GLOBALFREE(p)
```

```
// scalar new and delete macros
#define APP_PERMANENT_NEW(type) FEATSTD_MEMORYPOOL_PERMANENT_NEW(AppMemoryPool, type)
#define APP_TRANSIENT_NEW(type) FEATSTD_MEMORYPOOL_TRANSIENT_NEW(AppMemoryPool, type)
#define APP_DELETE(p) FEATSTD_MEMORYPOOL_DELETE(p)

// array new and delete macros
#define APP_PERMANENT_ARRAY_NEW(type, itemCount) \
        FEATSTD_MEMORYPOOL_ARRAY_NEW(AppMemoryPool, FeatStd::MemoryManagement::MemAttrib::Permar

#define APP_TRANSIENT_ARRAY_NEW(type, itemCount) \
        FEATSTD_MEMORYPOOL_ARRAY_NEW(AppMemoryPool, FeatStd::MemoryManagement::MemAttrib::Transi

#define APP_PERMANENT_ARRAY_INIT_NEW(type, itemCount, preset) \
        FEATSTD_MEMORYPOOL_ARRAY_NEW_WITH_PRESET(AppMemoryPool, FeatStd::MemoryManagement::MemAt

#define APP_TRANSIENT_ARRAY_INIT_NEW(type, itemCount, preset) \
    FEATSTD_MEMORYPOOL_ARRAY_NEW_WITH_PRESET(AppMemoryPool, FeatStd::MemoryManagement::MemAttrib

#define APP_ARRAY_DELETE(p) FEATSTD_MEMORYPOOL_ARRAY_DELETE(p)
```

MemoryPool Initialization

```
static const BinSpecification cPoolBinSpecs[] = {
    // PermanentOnly will force all allocations from these pools to be permanent.
    // This means that the buffer will be added to the bin free list and not
    // returned to the backing heap.
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 8),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 16),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 32),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 64),

    // An OverlowBarrier will stop bins with smaller buffer sizes to allocate
    // from the pools beyond the OverflowBarrier bin.
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::OverflowBarrier |
MemoryPoolBinFlags::PermanentOnly, 128),

    // GenuineOnly stops temporary allocations to cannibalize permanent buffers
    // in default allocation mode, a transient allocation request can be satisfied
    // by allocating a permanent buffer. With GenuineOnly, permanent buffers will
    // not be used for transient allocations.
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::GenuineOnly, 256),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::GenuineOnly, 512),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::GenuineOnly, 1024),

    // mandatory final entry
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(0, 0)
};
```

```
static const BufferPreset cBufferPresets[] = {
    // all bins up to 128 bytes are small bins. Thus buffer size
    // will always round up to bin buffer size.
    //                                  bufferSize    bufferCount
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(       8,            16),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(      16,            16),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(      32,            16),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(      64,            16),
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(     128,            16),

    // large bin buffers will have exactly the size requested
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(     300,             1),

    // mandatory final entry
    FEATSTD_MEMORYPOOL_BUFFER_PRESET(0, 0)
};



// ----------------------------------------------------------------------
bool InitAppMemoryPool()
{
    // initialize the memory pool of this sample application

    using namespace FeatStd::MemoryManagement;
    // IntrinsicBackingHeap defines a BackingHeap with an intrinsic memory buffer that
    // will be used by this heap. The size of the memory buffer is specified as template
    // parameter
    typedef IntrinsicBackingHeap<AppMemoryPool, 1 * 64 * 1024> Heap;

    // define the configuration for DefaultMemoryPool.
    static const MemoryPoolConfig config = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
        config,                             // name of the variable receiving the configuration
        "AppMemoryPool",                    // the name of the MemoryPool
        MemoryPoolFlags::None,              // Flags that control the memory pool behavior
        128,                                // all bins up to 128 bytes are small bins
        cPoolBinSpecs,                      // the bin configurations
        cBufferPresets,                     // preset allocations
        Heap,                               // type name of the backing heap
        &SampleApp::OnOutOfMemory,          // optional the out of memory function
        0                                   // no OnDestroy function
    );

    // initialize first the backing heap and then the memory pool associated with the backing he
    return Heap::Init() && AppMemoryPool::Init(&config);
}
```

Diagnostic Utilities

In MemoryPool Configuration StatisticsFileDumper and PermanentBlockFileDumper have been already briefly discussed. The two utility classes use FeatStd::MemoryManagement::MemoryPoolManager visitor facilities to dump MemoryPool statistics and permanent block list to a file.

The generated files then can be imported to Excel (CSV format) as outlined in MemoryPool Configuration.

The diagnostic utility classes can be incorporated in any application to analyse the application allocation profile.

StatisticsFileDumper

StatisticsFileDumper generates two CSV files

- one containing the overall MemoryPool statistics (equivalent to the data that can be found in FeatStd::MemoryManagement::PoolStatisticsData)
- one containing the bin statistics of all MemoryPools (equivalent to the data that can be found in FeatStd::MemoryManagement::BinStatisticsData)

> All MemoryPools means all MemoryPool instances registered at MemoryPoolManager at the time of the dump.

```
class StatisticsFileDumper {
    public:
        ~StatisticsFileDumper();

        static bool Dump(const FeatStd::Char *poolStatsFileName, const FeatStd::Char *binStatsFi
            StatisticsFileDumper sfd;
            return sfd.Init(poolStatsFileName, binStatsFileName) &&
                   FeatStd::MemoryManagement::MemoryPoolManager::DumpStatistics
(&DumpPoolStatistics, &DumpBinStatistics, &sfd);
        }

        template<typename MemRealm> static bool Dump(const FeatStd::Char *poolStatsFileName, cor
            StatisticsFileDumper sfd;
            return sfd.Init(poolStatsFileName, binStatsFileName) &&
                   FeatStd::MemoryManagement::MemoryPoolManager::DumpStatistics<MemRealm>(&DumpF
        }

    private:
        FeatStd::FileHandle mHdlPoolStats;
```

```
        FeatStd::FileHandle mHdlBinStats;

        StatisticsFileDumper();

        bool Init
(const FeatStd::Char *poolStatsFileName, const FeatStd::Char *binStatsFileName);

        static bool DumpPoolStatistics(const FeatStd::MemoryManagement::PoolMgmtObject
 &pmo, void *userData);

        static bool DumpBinStatistics(const FeatStd::MemoryManagement::BinMgmtObject
 &bmo, void *userData);

        FEATSTD_MAKE_CLASS_UNCOPYABLE(StatisticsFileDumper);
};
```

PermanentBlockFileDumper

PermanentBlockFileDumper uses

FeatStd::MemoryManagement::BinMgmtObject::VisitPermanentBlocks function to dump the permanent blocks of all MemoryPools to a CSV formatted file.

```
class PermanentBlockFileDumper {
    public:
        ~PermanentBlockFileDumper();

        template<typename MemRealm> static bool Dump(const FeatStd::Char *fileName) {
            PermanentBlockFileDumper pbd;
            FeatStd::MemoryManagement::ConcretePoolMgmtObject<MemRealm> pmo;
            return pbd.Init(fileName) &&
                   pmo.VisitBins(&BinVisitor, &pbd);
        }

        static bool Dump(const FeatStd::Char *fileName);

    private:
        FeatStd::FileHandle mFileHdl;

        PermanentBlockFileDumper();

        bool Init(const FeatStd::Char *fileName);

        static bool PoolVisitor(const FeatStd::MemoryManagement::PoolMgmtObject
 &pmo, void *userData);
        static bool BinVisitor(const FeatStd::MemoryManagement::BinMgmtObject
 &bmo, void *userData);
        static bool BlockVisitor(const FeatStd::MemoryManagement::BlockMgmtObject
 &bmo, void *userData);
```

```
        bool DoDump(const FeatStd::MemoryManagement::BlockMgmtObject &bmo);

        FEATSTD_MAKE_CLASS_UNCOPYABLE(PermanentBlockFileDumper);
};
```

> MemoryPool in normal operation on tracks free blocks.
> FEATSTD_MEMORYPOOL_LEAK_DETECTION_ENABLED must be enabled to include also allocated blocks into the permanent block dump.

MallocBackingHeap

FeatStd::MemoryManagement::MallocBackingHeap is a specific backing heap implementation that uses Standard C library malloc, realloc, and free to allocate memory.

Intended use scenarios:

- Host development
- Prototyping
- Debugging on target to utilize toolchain specific heap analysis functions (e.g. Greenhills heap analyzer)

```
// ============================================================================

bool InitMallocBackingHeapPool();

FEATSTD_MEMORYPOOL_DEFINE_MANAGED_MEMORYPOOL
(MallocBackingHeapPool, &InitMallocBackingHeapPool);

// ============================================================================


static const BinSpecification cPoolBinSpecs[] = {
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 8),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 16),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 32),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::PermanentOnly, 64),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::OverflowBarrier |
MemoryPoolBinFlags::PermanentOnly, 128),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::GenuineOnly, 256),
```

```
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::GenuineOnly, 512),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::GenuineOnly, 1024),

    // mandatory final entry
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(0, 0)
};

// ----------------------------------------------------------------------
bool InitMallocBackingHeapPool()
{
    using namespace FeatStd::MemoryManagement;

#if defined(FEATSTD_PLATFORM_OS_WIN32)
    // for host development use MallocBackingHeap. MallocBackingHeap will
    // use C Standard Library Heap (malloc, realloc, free). Thus is
    // especially usefull if compiling for SCHost to avoid.
    typedef MallocBackingHeap<MallocBackingHeapPool> Heap;
#else
    // for other builds use a fixed size heap
    typedef IntrinsicBackingHeap<MallocBackingHeapPool, 1 * 1024> Heap;
#endif

    // define the configuration for DefaultMemoryPool.
    static const MemoryPoolConfig config = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
        config,                             // name of the variable receiving the configuration
        "MallocBackingHeapPool",            // the name of the MemoryPool
        MemoryPoolFlags::None,              // Flags that control the memory pool behavior
        128,                                // all bins up to 128 bytes are small bins
        cPoolBinSpecs,                      // the bin configurations
        0,                                  // preset allocations
        Heap,                               // type name of the backing heap
        &SampleApp::OnOutOfMemory,          // optional the out of memory function
        0                                   // no OnDestroy function
    );

    // initialize first the backing heap and then the memory pool associated with the backing he
    return Heap::Init() && MallocBackingHeapPool::Init(&config);
}

// ----------------------------------------------------------------------
void MallocBackingHeapSample()
{
    void *p = FEATSTD_MEMORYPOOL_ALLOC(MallocBackingHeapPool, MemAttrib::Transient, 20);
    FEATSTD_MEMORYPOOL_FREE(MallocBackingHeapPool, p);
}
```

Memory Leak Analysis

MemoryPool has a highly capable memory leak analysis facility. At any time, the application can request to visit the allocated block list. An allocated block list is maintained for each MemoryPool bin.

> FEATSTD_MEMORYPOOL_LEAK_DETECTION_ENABLED must be enabled to visit the allocated blocks list

Typically the allocated block list will be dumped in FeatStd::MemoryManagement::MemoryPoolConfig::mOnDestroy callback function. This ensures that all non-leaked memory has been de-allocated and the remaining allocated blocks may indicate memory leaks.

> It might be a design decision not to de-allocate all memory at system shutdown. Technically a memory leak is given when no references to the allocated buffer exist any more.

Central classes for MemoryPool analysis are:

- FeatStd::MemoryManagement::MemoryPoolManager
- FeatStd::MemoryManagement::BlockMgmtObject
- FeatStd::MemoryManagement::BinMgmtObject
- FeatStd::MemoryManagement::PoolMgmtObject
- FeatStd::MemoryManagement::ConcretePoolMgmtObject

> Visiting the allocated block list requires locking of the lists. Thus no memory allocations can be done during visit.

The following code shows how to trigger an allocated block dump.

```
// ----------------------------------------------------------------------
void MemoryLeakAnalysisSample()
{
#if defined(FEATSTD_MEMORYPOOL_LEAK_DETECTION_ENABLED)
    using FeatStd::Internal::Diagnostic;

    // allocate some memory and fill with data
    Char *str = static_cast<Char*>(APP_TRANSIENT_ALLOC(30));
```

```
    if (str == 0) {
        FEATSTD_LOG_FATAL("out of memory");
    }

    FeatStd::Internal::String::Copy(str, "this is a memory leak");

    // now dump the allocated blocks of AppMemoryPool - output will
    // go to Diagnostics::DebuggerOut.  this is the default implementation
    // provided by MemoryPool.
    FeatStd::MemoryManagement::MemoryPoolManager::DumpAllocatedBlocks<AppMemoryPool>();

    // VisualStudio Output pane should have now following output
    // <...>\MemLeakAnalysis.cpp(23): 0018AC28[32+16] AppMemoryPool::2
    //      data < 74 68 69 73 20 69 73 20 61 20 6d 65 6d 6f 72 79 20 6c 65 61 6b 00 cd cd > thi

    // repeat dump of allocated blocks, this time with a user defined visitor function
    LogAppMemoryPoolAllocatedBlocks();

    // BlockMgmtObject can be used to get information on each MemoryPool allocated buffer
    FeatStd::MemoryManagement::BlockMgmtObject bmo(str);
    FeatStd::Internal::Diagnostic::DebuggerOut("str has been allocated from pool %s at: %s\n", b

    // release memory
    APP_FREE(str);
#endif
}
```

Executing the code above in MS Visual Studio will result in following output in Debug Output Pane:

```
s:\featstd\src\MemoryPoolSample\MemLeakAnalysis.cpp(121): 00443DB0[32+16] AppMemoryPool::2
    data < 74 68 69 73 20 69 73 20 61 20 6d 65 6d 6f 72 79 20 6c 65 61 6b 00 cd cd > this.is.a.m
```

A customized version of an allocated blocks visitor can be seen below.

```
// ------------------------------------------------------------------------
static void LogBlockManagementObject(const FeatStd::MemoryManagement::BlockMgmtObject &bmo)
{
    using FeatStd::Internal::Diagnostic;

    // dump information to Diagnostic::DebuggerOut
    Diagnostic::DebuggerOut("%s: %p[%p] %s::%u %s %s [%u+%u]\n",
                            bmo.GetAllocContext(),
                            bmo.GetBlockAddress(),
                            bmo.GetBuffer(),
                            bmo.GetAssociatedPoolName(),
                            bmo.GetAssociatedBinIndex(),
                            (bmo.IsLargeBlock() ? "large" : "small"),
                            (bmo.IsPermanentBlock() ? "permanent" : "transient"),
                            bmo.GetBufferSize(),
```

```
                            bmo.GetBlockSize() - bmo.GetBufferSize());
}

namespace {
    // Accumulation of information abount allocated blocks
    struct AllocatedBlockData {
        AllocatedBlockData() {
            mBlockCount = 0;
            mAccumulatedBlockSize = 0;
            mAccumulatedBufferSize = 0;
        }

        UInt32 mBlockCount;
        UInt32 mAccumulatedBlockSize;
        UInt32 mAccumulatedBufferSize;
    };
}

#if defined(FEATSTD_MEMORYPOOL_LEAK_DETECTION_ENABLED)
// ---------------------------------------------------------------------
static bool VisitAllocatedBlock(const FeatStd::MemoryManagement::BlockMgmtObject
 &bmo, void *userData)
{
    FEATSTD_DEBUG_ASSERT(userData != 0);

    // userData refers to an AllocatedBlockData
    // cast and update AllocatedBlockData with the current block data
    AllocatedBlockData *abd = static_cast<AllocatedBlockData*>(userData);
    ++abd->mBlockCount;
    abd->mAccumulatedBlockSize += bmo.GetBlockSize();
    abd->mAccumulatedBufferSize += bmo.GetBufferSize();

    // finally log the block info
    LogBlockManagementObject(bmo);

    // always return true (returning false abandons the visit)
    return true;
}
#endif

// ---------------------------------------------------------------------
void LogAppMemoryPoolAllocatedBlocks()
{
    using FeatStd::Internal::Diagnostic;

#if defined(FEATSTD_MEMORYPOOL_LEAK_DETECTION_ENABLED)
    // VisitAllocatedBlock expects a AllocatedBlockData passed with userData
    AllocatedBlockData abd;
    FeatStd::MemoryManagement::MemoryPoolManager::DumpAllocatedBlocks<AppMemoryPool>(&VisitAlloc

    if (abd.mBlockCount > 0) {
        // if allocated blocks have been found, dump summary information
```

```
        Diagnostic::DebuggerOut("%u allocated block(s) found with %u+%u bytes\n",
                                abd.mBlockCount,
                                abd.mAccumulatedBufferSize,
                                abd.mAccumulatedBlockSize - abd.mAccumulatedBufferSize);
    }
    else {
        Diagnostic::DebuggerOut("no allocated blocks found\n");
    }
#endif
}
```

Executing the code above in MS Visual Studio will result in following output in Debug Output Pane:

```
S:\featstd\src\MemoryPoolSample\MemLeakAnalysis.cpp(121): 00443DA0[00443DB0] AppMemoryPool::2 sm
1 allocated block(s) found with 32+16 bytes
```

STL (Standard Template Library) Sample

STL sample shows how to use MemoryPool with Standard Template Library container classes. STL container classes accept an allocator template argument. The allocator type has standardized interface to allocate and free memory.

The StdAllocator class shows how to implement a STL allocator with MemoryPool:

```
template<typename T, typename Pool, UInt8 attrib = FeatStd::MemoryManagement::MemAttrib::Transie
struct StdAllocator {
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef const T* const_pointer;
    typedef const T& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    inline StdAllocator() throw() { }
    inline StdAllocator(const StdAllocator&) throw() { }
    template<class U> inline StdAllocator(const StdAllocator<U, Pool, attrib> &) throw() { }
    ~StdAllocator() { }

    inline pointer address ( reference x ) const { return &x; }
    inline const_pointer address ( const_reference x ) const { return &x; }

    inline pointer allocate(size_type n, const void *hint = 0) const {
        FEATSTD_UNUSED(hint);
        return static_cast<pointer>(Pool::Alloc(FEATSTD_MEMORYPOOL_CALL_CONTEXT(sizeof(T) * n),
    }

    inline void deallocate(pointer p, size_type n) const {
        FEATSTD_UNUSED(n);
        Pool::Free(p);
    }
```

```
    inline size_type max_size() const throw() {
        return UInt32(0xffffffff) / sizeof(T);
    }

    inline void construct(pointer p, const_reference val) const {
        (void) FeatStd::MemoryManagement::ConstructObject<T>(p, val);
    }

    inline void destroy(pointer p) const {
        FeatStd::MemoryManagement::DestructObject<T>(p);
    }

    // MS specific (not mentioned in C++ standard)
    template<typename U> struct rebind {
       typedef StdAllocator<U, Pool, attrib> other;
    };
};
```

> The StdAllocator::rebind type is not part of C++ standard but required by MS Visual Studio flavor STL implementation.

The following code shows how to use StdAllocator with STL vector class.

```
#include <vector>
#include "SampleApp.h"
#include "StdAllocator.h"

FEATSTD_LOG_SET_REALM(SampleAppLogRealm);

// ---------------------------------------------------------------------
void StlSample()
{
    using namespace FeatStd::MemoryManagement;

    SampleApp::EnableMemoryPoolLogging();

    {
        // define an allocator that uses AppMemoryPool and does permanent allocations
        typedef StdAllocator<UInt32, AppMemoryPool, MemAttrib::Permanent> AppMemPoolAllocator;

        // std::vector using AppMemoryPool for allocations
        std::vector<UInt32, AppMemPoolAllocator> v;
        // reserve space for 10 items
        v.reserve(10);
        // fill the items with data
        for (UInt32 i = 0; i < v.capacity(); ++i) {
```

```
            v.push_back(i);
        }
    }

    SampleApp::EnableMemoryPoolLogging(false);
}
```

The code above will trigger following log output:

```
S:\featstd\src\FeatStd\MemoryPool\MemoryPoolState.cpp(113)
0000000.250 [0x00001fcc] DEBUG FeatStdMemoryManagement S:\featstd\src\memorypoolsample\StdAlloca
 {LogAllocSuccess}
S:\featstd\src\FeatStd\MemoryPool\MemoryPoolState.cpp(113)
0000000.250 [0x00001fcc] DEBUG FeatStdMemoryManagement S:\featstd\src\memorypoolsample\StdAlloca
 {LogAllocSuccess}
S:\featstd\src\FeatStd\MemoryPool\MemoryPoolState.cpp(140)
0000000.250 [0x00001fcc] DEBUG FeatStdMemoryManagement S:\featstd\src\memorypoolsample\StdAlloca
 {LogFree}
S:\featstd\src\FeatStd\MemoryPool\MemoryPoolState.cpp(140)
0000000.250 [0x00001fcc] DEBUG FeatStdMemoryManagement S:\featstd\src\memorypoolsample\StdAlloca
 {LogFree}
```

Unmanaged MemoryPool

The lifetime of managed MemoryPools is handled by FeatStd library facilities. Unmanaged MemoryPool lifetime is completely in control of the application.

Managed MemoryPools

- Initialization
  - latest before main function will be invoked
  - at event of first allocation request
- Destruction
  - at the latest possible time, but earliest before main function has been executed.

Unmanaged MemoryPools

Initialization and destruction of the pool is controlled by the application.

Following code shows how to declare an Unmanaged MemoryPool:

```
// define an application specific memory pool with realm UnmanagedPool
// in contradiction to a managed memory pool, the application needs to take
// care to initialize the pool correctly before first use
// The pool can be - as any other MemoryPool - explicitly destroyed, or
// MemoryPoolManager will destroy the pool after main function has exit.
FEATSTD_MEMORYPOOL_DEFINE_MEMORYPOOL(UnmanagedPool);
```

Following code shows some sample use cases for Unmanaged MemoryPools

```
// ============================================================================

static const BinSpecification cPoolBinSpecs[] = {
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 8),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 16),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 32),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 64),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::OverflowBarrier, 128),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 256),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(0, 0)
};

namespace {

    // define the heap to be used for UnmanagedPool
    typedef BackingHeap<UnmanagedPool> Heap;

    // define the configuration for DefaultMemoryPool.
    static const MemoryPoolConfig cUnmanagedPoolConfig = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
        cUnmanagedPoolConfig,               // name of the variable receiving the configuration
        "UnmanagedPool",                    // the name of the MemoryPool
        MemoryPoolFlags::None,              // Flags that control the memory pool behavior
        128,                                // all bins up to 128 bytes are small bins
        cPoolBinSpecs,                      // the bin configurations
        0,                                  // no presets
        Heap,                               // type name of the backing heap
        &SampleApp::OnOutOfMemory,          // optional the out of memory function
        0                                   // no OnDestroy function
    );

}

// ----------------------------------------------------------------------------
static bool InitUnmanagedPool()
{
    using namespace FeatStd::MemoryManagement;

    // the size of the backing heap buffer (buffer managed by BackingHeap)
    static const UInt32 cHeapSize = 2048;
    // define the heap buffer (never use UInt8 to define this buffers as
    // the linker might locate a UInt8 to an unaligned address)
    static UInt32 sHeapBuffer[(cHeapSize + 3) / 4];

    // initialize first the backing heap and then the memory pool associated with the backing he
    // The BackingHeap::Init expects a pointer to a buffer and the buffer size
    // The buffer can be - like in this example - a static declared memory buffer,
    // but also a fixed address (e.g. video ram area of a GPU).
    return Heap::Init(sHeapBuffer, cHeapSize) && UnmanagedPool::Init(&cUnmanagedPoolConfig);
```

```
}

static void *sUnmanagedHeapBuffer;

// ------------------------------------------------------------------------
static bool InitDynamicUnmanagedPool(UInt32 heapSize)
{
    using namespace FeatStd::MemoryManagement;
    using FeatStd::Internal::Diagnostic;

    if (sUnmanagedHeapBuffer != 0) {
        return true;
    }

    // in this sample the heap buffer is allocated from the application heap
    // thus a MemoryPool within a MemeoryPool is created (nested MemoryPool)
    sUnmanagedHeapBuffer = APP_TRANSIENT_ALLOC(heapSize);

    // if heap buffer allocation succeeded, initialize the Heap with the
    // allocated buffer
    if (sUnmanagedHeapBuffer == 0 || !Heap::Init(sUnmanagedHeapBuffer, heapSize)) {
        return false;
    }

    Diagnostic::DebuggerOut("allocated %u bytes for backing heap, %u bytes available for allocat
                            heapSize,
                            Heap::GetFreeAreaSize());

    // log the buffer attributes
    LogBufferAttributes(sUnmanagedHeapBuffer);

    // now initialize UnmanagedMemoryPool
    return UnmanagedPool::Init(&cUnmanagedPoolConfig);
}

// ------------------------------------------------------------------------
static void DestroyDynamicUnmanagedPool()
{
    using namespace FeatStd::MemoryManagement;

    if (sUnmanagedHeapBuffer != 0 && UnmanagedPool::IsInitialized()) {
        // if sanity checks passed, destroy UnmanagedPool, destroy Heap, and free
        // heap memory buffer to AppMemoryPool
        UnmanagedPool::Destroy();
        Heap::Destroy();
        APP_FREE(sUnmanagedHeapBuffer);
        sUnmanagedHeapBuffer = 0;
    }

}

// ------------------------------------------------------------------------
void UnmanagedPoolSample()
```

```
{
    if (!InitUnmanagedPool()) {
        FEATSTD_LOG_FATAL("could not initialize UnmanagedPool");
    }

    void *p = UnmanagedPool::Alloc(FEATSTD_MEMORYPOOL_CALL_CONTEXT(32), FeatStd::MemoryManagemer
    if (p != 0) {
        UnmanagedPool::Free(p);
    }

    // explicit destruction of the pool && Heap
    UnmanagedPool::Destroy();
    Heap::Destroy();

    // create UnmanagedPool with a 1024 bytes buffer
    if (InitDynamicUnmanagedPool(1024)) {
        p = UnmanagedPool::Alloc(FEATSTD_MEMORYPOOL_CALL_CONTEXT(32), FeatStd::MemoryManagement:
        if (p != 0) {
            UnmanagedPool::Free(p);
        }
        DestroyDynamicUnmanagedPool();
    }
}
```

Creating A Fragment Free Memory Pool Confuration

MemoryPool allows a broad range of operation modes. MemoryPool implementation and design itself already tries to reduce the risk of memory fragmentation by separating transient and permanent allocations. Nevertheless fragmentation still can happen as long transient allocations are done.

> To be more precise - also a system with transient allocations can be fragment free. The application "just" has to ensure that functions are side effect free in terms of dynamic memory allocations. E.g. each function will free all dynamic memory it has allocated before returning to the calling function.

Permanent allocations do not cause memory fragmentation, thus the easiest and most straight forward way to achieve a fragment free system. With MemoryPool this can be easily achieved by defining FeatStd::MemoryManagement::MemoryPoolFlags::PermanentOnly.

Once set in the MemoryPool configuration, this flag will overrule the application requests for transient memory and the pool will only perform permanent memory allocations.

Following code sample shows how to configure a MemoryPool for permanent allocations only:

```
static const BinSpecification cStageOneBinSpecs[] = {
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 4),
    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 1024),

    FEATSTD_MEMORYPOOL_BIN_SPECIFICATION(MemoryPoolBinFlags::None, 0)
};

// define the configuration for DefaultMemoryPool.
static const MemoryPoolConfig cStageOneConfig = FEATSTD_MEMORYPOOL_CONFIG_INITIALIZER(
    cStageOneConfig,                    // name of the variable receiving the configuration
    "ConfigSampleStageOneMemoryPool",   // the name of the MemoryPool
    MemoryPoolFlags::PermanentOnly,     // Flags that control the memory pool behavior
    4,                                  // all bins up to 128 bytes are small bins
    cStageOneBinSpecs,                  // the bin configurations
    0,                                  // preset allocations
    ConfigSampleHeap,                   // type name of the backing heap
    0,                                  // optional the out of memory function
    0                                   // no OnDestroy function
);
```

Of course this may lead to accessive memory use if the application has a very high buffer size variance in its allocations. MemoryPool Configuration shows how to optimize the system towards specific application memory allocation footprints.

The optimization of MemoryPool configuration can be an exhaustive process. Using the MemoryPool overflow mechanism strategically to re-use buffers in different use cases can save significant amount of memory, but requires in-depth analysis of allocations over time and ties application dynamics tightly to the MemoryPool configuration - something definitely to avoid in application development phase.

Frequently Asked Questions and Tips

- What is the difference between small and large blocks?
- Is it possible to determine from which MemoryPool a buffer has been allocated from?
- My application reports memory leaks, but the allocation happens in a generic class used in multiple places
- What is the sequence of block allocations?
- How to log all allocations and deallocations?
- How does the number of bins influence system performance?

What is the difference between small and large blocks?

Small blocks bear less overhead - both in RAM and runtime - then large blocks. As all blocks in a small block bin have the same size, the size of the block must not be stored with the block. Thus a small block has only 4 bytes overhead. In contrary large blocks differ in size and therefore the size must be stored with the block. This results in 8 bytes administrative overhead for a large block.

Small blocks have also less runtime overhead. As all blocks in small block bin have the same size, the free list does not require sorting. Adding a block to the free list - which is a simple single linked list - has O(1) run time.

Large block bin free list is a balanced binary tree (Red Black Tree). Thus inserting and removing has O(log n) run time.

Is it possible to determine from which MemoryPool a buffer has been allocated from?

Yes, the class FeatStd::MemoryManagement::BlockMgmtObject does the trick. An instance of FeatStd::MemoryManagement::BlockMgmtObject can be created with a valid pointer to a buffer allocated from a MemoryPool.

In the following code FeatStd::MemoryManagement::BlockMgmtObject will be used to determine the name of the MemoryPool a certain buffer has been allocated from.

```
// ------------------------------------------------------------------------
void MemoryLeakAnalysisSample()
{
#if defined(FEATSTD_MEMORYPOOL_LEAK_DETECTION_ENABLED)
    using FeatStd::Internal::Diagnostic;

    // allocate some memory and fill with data
    Char *str = static_cast<Char*>(APP_TRANSIENT_ALLOC(30));
    if (str == 0) {
        FEATSTD_LOG_FATAL("out of memory");
    }

    FeatStd::Internal::String::Copy(str, "this is a memory leak");

    // now dump the allocated blocks of AppMemoryPool - output will
    // go to Diagnostics::DebuggerOut.  this is the default implementation
    // provided by MemoryPool.
    FeatStd::MemoryManagement::MemoryPoolManager::DumpAllocatedBlocks<AppMemoryPool>();

    // VisualStudio Output pane should have now following output
    // <...>\MemLeakAnalysis.cpp(23): 0018AC28[32+16] AppMemoryPool::2
    //      data < 74 68 69 73 20 69 73 20 61 20 6d 65 6d 6f 72 79 20 6c 65 61 6b 00 cd cd > thi

    // repeat dump of allocated blocks, this time with a user defined visitor function
    LogAppMemoryPoolAllocatedBlocks();
```

```
    // BlockMgmtObject can be used to get information on each MemoryPool allocated buffer
    FeatStd::MemoryManagement::BlockMgmtObject bmo(str);
    FeatStd::Internal::Diagnostic::DebuggerOut("str has been allocated from pool %s at: %s\n", b
GetAssociatedPoolName(), bmo.GetAllocContext());

    // release memory
    APP_FREE(str);
#endif
}
```

My application reports memory leaks, but the allocation happens in a generic class used in multiple places

Generic classes like FeatStd::Internal::Vector that allocate memory are used in various places. Thus the allocation context of the leaked block will tell only that the allocation happened in FeatStd::Internal::Vector class, but will not tell from which of the many places FeatStd::Internal::Vector class is used the leak stems from. If the leak is reproducible in MS Visual Studio host environment, the origin of the leak can be easily determined.

- Create a breakpoint the line after the allocation context of the leaked block is referring to. In this example this will be inside FeatStd::Internal::Vector::Reserve method.
- Once created, open the breakpoint context menu and select "When hit ...". In the "When Breakpoint Is Hit" dialog select print a message.
- Now add the name of the variable that received the address of the allocated block in curly braces (in case of FeatStd::Internal::Vector::Reserve add {newMemory}). This will print the value of the variable to the Debug output pane of Visual Studio when the application is executed.
- Now add also $CALLER (or $CALLSTACK) the the message. This will print the calling function / the complete callstack too.
- Run the program in VS debug mode
- Go to to the end of the debug output pane and get the address of the leaked block from the memory leak report.
- Search the address from bottom to top in the debug output pane.
- This will bring you to the output generated from the breakpoint. It will contain the address of the leaked block and the caller / callstack that invoked the function.

What is the sequence of block allocations?

The default behavior of MemoryPool for allocations is:

1. Determine the bin to allocate from according the the requested buffer size
2. Check the free list of the identified bin for suitable blocks
    - Trivial for small block bins

    - Best fit search for large block bins
3. If no suitable block could be found in the free list, try to allocate an appropriate buffer from the backing heap.
4. If the allocation from the backing heap fails, try to allocate from the next larger bins (overflow).

The default behavior can be influenced with flags FeatStd::MemoryManagement::MemoryPoolFlags::Enum and FeatStd::MemoryManagement::MemoryPoolBinFlags::Enum.

- If the MemoryPool configuration specifies FeatStd::MemoryManagement::MemoryPoolFlags::Enum::NoOverflow, the MemoryPool will not execute step 4.
- If FeatStd::MemoryManagement::MemoryPoolFlags::Enum::GenuineOnly is specified, transient blocks will not be allocated from the bin free list (the MemoryPool will skip step 2).
- If the bin defines FeatStd::MemoryManagement::MemoryPoolBinFlags::Enum::OverflowPreferred, the MemoryPool will exchange step 3 and 4 (first try to overflow and then allocate from the backing heap).

How to log all allocations and deallocations?

MemoryPool logs all requests with FeatStd logging facilities. To see the allocation logs, logging facilities need to be configured accordingly:

```
        static inline void EnableMemoryPoolLogging(bool enable = true) {
            using namespace FeatStd::Diagnostics;

            // get the logger and set log level to all (MemoryPool logs allocs and free at log
            // log level Info
            if (enable) {
                LogControl::GetLogger<LogRealm::FeatStdMemoryManagement>().SetLevel(
LogLevel::All);
            }
            else {
                LogControl::GetLogger<LogRealm::FeatStdMemoryManagement>().SetLevel(LogControl::
            }

        }
```

How does the number of bins influence system performance?

The number of bins have both impact on RAM and system performance. Typically 12 bytes must be accounted for each bin object.

RAM

The Excel file MemoryPoolStatisticsAnalysis.xlsm contains a sheet labeled ResourceReq. MemoryPool configuration can be specified and the overall RAM consumption of the MemoryPool will be estimated.

Runtime Performance

The number of bins have two impacts on runtime performance.

- bin lookup
- locking (in terms of multithreading)

Bin Lookup

Bin lookup is done with binary search - thus it has a runtime complexity of O(log n). Therefore bin count has marginal impact on bin lookup time.

Locking

MemoryPool implementation is thread safe, therefore shared resource access must be synchronized. MemoryPool locking is done on bin level. This means that two threads allocating memory from two different bins of the same MemoryPool will not lock each other. Thus the number of bins can have positive impact on the number of thread context switches.

MemoryPool uses FeatStd::Internal::Spinlock for thread synchronization.

Summary

	Low Bin Count	High Bin Count
Runtime	increased number of locking conflicts may decrease overall performance	reduced risk of locking conflicts
RAM	marginal (12 bytes per bin)	marginal (12 bytes per bin)
Overhead per allocation	low bin count may be an indicator of higher overhead per allocation	high bin count is an indicator of well configured bins and lower overhead per allocation

> The number of bins to configure is heavily related to variance of buffer sizes actually allocated. Many different buffer sizes lead to a high number of bins (otherwise the overhead per allocation increases), low buffer size variance typically results also in a low bin count.

MemoryPool Excel Analysis

To ease the analysis of MemoryPool statistics and diagnostics output, the Excel file supplement/MemoryPool/MemoryPoolStatisticsAnalysis.xltm is supplied. The Excel can import data generated during runtime of an application using MemoryPool (CSV files). The data import format is defined by re-useable code snippets in MemoryPoolSample application (Diagnostic Utilities)

Incorporate and adapt the code snippets from MemoryPoolSample in your application. Once the application has been run, a list of CSV files will be generated:

- MemoryPoolStatistics.csv contains the overall MemoryPool statistics
- MemoryPoolBinStatistics.csv contains the Bin statistics of each MemoryPool
- PermanentBlockList.csv contains the information on all permanent blocks in all MemoryPool instances

Note that features FEATSTD_MEMORYPOOL_LEAK_DETECTION and FEATSTD_MEMORYPOOL_STATISTICS must be enabled to generate the CSV files.

PoolAnalysis Sheet

The pool analysis sheet can be used to visualize the overall MemoryPool statistics of each MemoryPool in the system. Press the update button and select the file MemoryPoolStatistics.csv that has been generated by the application. The data from the file will be read and the tables will be updated.

PoolBinAnalysis

The PoolBinAnalysis sheet can be used to analyse the bin statistics of each MemoryPool. Press the update button and select the file MemoryPoolBinStatistics.csv which has been generated by the application.

In the field PoolName the MemoryPool to analyse can be selected.

PermanentBlockAnalysis

Can be used to analyse all permanent blocks MemoryPool instances. MemoryPool Configuration how to use the permanent block list.

ResourceReq

Can be used to estimate the RAM resource requirements of MemoryPool.