In the article comparing CPU and GPU implementations of the median filter, the class was designed to receive image buffers with the AddSample method and return a pointer to the image buffer with the GetMedianImage method:

```
class TM_CPURefImpl : public TM_BaseClass { public: TM_CPURefImpl(const int
_NumSamples,
const
int
_BufferSize);
virtual
~TM_CPURefImpl(
void
);
virtual
void
AddSample(
void
*NewBuffer);
virtual
void
*GetMedianImage();
protected
:
unsigned
char
*Buffers;
unsigned
char
*MedianBuffer; std::vector<
unsigned
char
> SortBuffer;
unsigned
char
*MedianValuePtr;
void
BuildMedianImage(
unsigned
char
*NewBuffer); };
```

However, this is a poor design (granted, OOP was not the aim of the article) and should be improved. In fact, it assumes that the programmer using this class never:

1. deletes the class instance and keeps using the returned buffer
2. deletes the returned buffer

Let us start with case number 1. Given that the temporal median filter is a running estimate of a state, it is unlikely that the programmer would instance such a class, get a result then delete the instance and keep using the obtained pointer:

```
  TM_CPURefImpl * NewImpl = new TM_CPURefImpl(3, 1024);
NewImpl->AddSample(myBuffer);                                    unsigned char
*myResultBuffer = (
unsigned
char
*)NewImpl->GetMedianImage();
delete
NewImpl;  DoFurtherProcessing(myResultBuffer);
```

But he may assume that GetMedianImage returns a copy of the buffer and so that it remains valid even after the deletion of the class instance, as there is no clear hint that this buffer is handled exclusively by the class instance.

Case number 2 is even worse, as the programmer could delete the returned image buffer, and causa a catastrophic failure in the code of the class instance as an internal buffer used as output for the computation would become invalid:

```
  TM_CPURefImpl * NewImpl = new TM_CPURefImpl(3, 1024);
NewImpl->AddSample(myBuffer);                                    unsigned char
*myResultBuffer = (
unsigned
char
*)NewImpl->GetMedianImage();
delete
[] myResultBuffer;  NewImpl->AddSample(myNewBuffer);
```

The clear solution for this poor design is wrapping the pointer to the median buffer inside a shared_ptr that manages the count of references, so that the median buffer gets automatically deleted when both the class instance is freed and the returned pointer is not used anymore by the calling code. Still, using shared_ptr with an array is not feasible, as on destruction it calls delete instead of delete[] so it would leak memory. There are at least three possible solutions for this problem:

1. use a std::vector instead of an array and wrap it inside a shared_ptr
2. declare a custom destructor for this shared_ptr
3. use Boost shared_array

We will use solution number 3. Boost's shared_array behaves like shared_ptr but is optimized for wrapping an array, so it properly deallocates memory. The declaration of the class becomes:

```
#include <vector> #include <algorithm> #include <iostream> #include <boostsmart_ptr.hpp>
using namespace std; class TM_CPURefImpl { public: TM_CPURefImpl(const int
_NumSamples,
const
int
_BufferSize);
virtual
~TM_CPURefImpl(
void
);
virtual
void
AddSample(
unsigned
char
*NewBuffer);
virtual
boost::shared_array<
unsigned
char
> GetMedianImage();
protected
:
int
NumSamples;
int
BufferSize;
int
BufferIndex; vector<
unsigned
char
> Buffers; boost::shared_array<
unsigned
char
> MedianBuffer; vector<
unsigned
```

```
char
> SortBuffer;
unsigned
char
*MedianValuePtr;
int
GetIndexOfMedian();
void
BuildMedianImage(
unsigned
char
*NewBuffer);  };
```

The relevant changes are:

- GetMedianImage now returns a boost::shared_array<unsigned char> instead of a void *
- the type of MedianBuffer is now boost::shared_array<unsigned char> instead of unsigned char *

The setup of the MedianBuffer is now in the initializers list of the class constructor:

```
  TM_CPURefImpl::TM_CPURefImpl(const int _NumSamples, const int _BufferSize) :
MedianBuffer(                                                                 new unsi
gned
char
[_BufferSize])  {      _ASSERT(_NumSamples >= 3);        NumSamples = _NumSamples;
BufferSize = _BufferSize;      BufferIndex = 0;        Buffers.resize(NumSamples * BufferSize);
//< stores NumSamples buffers of the given BufferSize, interleaving data from different buffers
   SortBuffer.resize(NumSamples);      MedianValuePtr = &SortBuffer[GetIndexOfMedian()];  }
```

The GetMedianImage method returns a copy of the shared_array to the caller, increasing the reference count of the median buffer:

```
  boost::shared_array<unsigned char> TM_CPURefImpl::GetMedianImage()  {      return
MedianBuffer;  }
```

Now let's see if this design solves the problems we have highlighted before. In this code fragment we delete the class instance and then access again the returned buffer to verify that it

is valid. The code works as follows:

1. instantiating the class initializes the shared_array in the class constructor, with reference count set to 1
2. when the code calls RefImpl->GetMedianImage(), a copy of the shared_array is returned and the reference count goes to 2
3. when the class instance is freed, the shared_array inside the class instance is deleted, the reference count falls to 1 but since it is not zero, the median buffer is not deallocated
4. at the end of Test1(), the ResultBuffer shared_array is destroyed, so the reference count finally drops to 0 and the median buffer is deallocated

```
void Test1() {      cout << "Test1: verifies that deleting the TM_CPURefImpl does not delete the returned median image" << endl;              const int NUM_SAMPLES = 3;      const int SAMPLES_SIZE = 1024;      TM_CPURefImpl *RefImpl =
new
TM_CPURefImpl(NUM_SAMPLES, SAMPLES_SIZE);      boost::shared_array<
unsigned
char
> InputBuffer(
new
unsigned
char
[SAMPLES_SIZE]);
for
(
int
i = 0; i < SAMPLES_SIZE; i++)          InputBuffer[i] = (
unsigned
char
)(i & 0xFF);
for
(
int
j = 0; j < NUM_SAMPLES; j++)          RefImpl->AddSample(InputBuffer.get());
boost::shared_array<
unsigned
char
> ResultBuffer = RefImpl->GetMedianImage();
if
(memcmp(InputBuffer.get(), ResultBuffer.get(), SAMPLES_SIZE) != 0)
//< verify result of computation
      cout << " Error! wrong computation" << endl;
delete
RefImpl;
//< delete instance
```

```
if
(memcmp(InputBuffer.get(), ResultBuffer.get(), SAMPLES_SIZE) != 0)
//< verify that returned median image is still valid
      cout << " Error! invalidated buffer" << endl;
else
      cout << " buffer is ok" << endl;  }
```

Given that the programmer now receives a shared_array instead of a raw pointer from the GetMedianImage() method, he cannot directly delete the buffer, but he can reset the returned shared_array, that is discarding the result. However, resetting the shared_pointer has no effect on the median buffer inside the class instance. This is what happens when this code fragment runs:

1. instantiating the class initializes the shared_array in the class constructor, with reference count set to 1
2. when the code calls RefImpl->GetMedianImage(), a copy of the shared_array is returned and the reference count goes to 2
3. the returned shared_array is reset by the caller as he is not interested in the result anymore, so the reference count drops to 1 (the same happens when the returned shared_array goes out of scope and is freed)
4. when the class instance is deleted, the reference count falls to 0 and the median buffer is deleted

```
 void Test2()  {      cout << "Test2: verifies that deleting the returned shared_ptr does not delete
the internal median buffer" << endl;              const int NUM_SAMPLES = 3;      const int
SAMPLES_SIZE = 1024;      TM_CPURefImpl *RefImpl =
new
TM_CPURefImpl(NUM_SAMPLES, SAMPLES_SIZE);      boost::shared_array<
unsigned
char
> InputBuffer(
new
unsigned
char
[SAMPLES_SIZE]);
for
(
int
i = 0; i < SAMPLES_SIZE; i++)          InputBuffer[i] = (
unsigned
char
)(i & 0xFF);
for
```

```
(
int
j = 0; j < NUM_SAMPLES; j++)          RefImpl->AddSample(InputBuffer.get());
boost::shared_array<
unsigned
char
> ResultBuffer = RefImpl->GetMedianImage();
if
(memcmp(InputBuffer.get(), ResultBuffer.get(), SAMPLES_SIZE) != 0)          cout << "Error!
wrong computation" << endl;      ResultBuffer.reset();
//< reset returned median buffer
  boost::shared_array<
unsigned
char
> NewResultBuffer = RefImpl->GetMedianImage();
//< get another median buffer

if
(memcmp(InputBuffer.get(), NewResultBuffer.get(), SAMPLES_SIZE) != 0)
//< verify that new buffer is still correct
      cout << " Error! corrupted buffer" << endl;
else
      cout << " buffer is ok" << endl;
delete
RefImpl;  }
```

Summing up, using the Boost shared_array allows us to easily design a class that properly handles different usage patterns, while protecting the programmer from crashes and memory leaks.