- Examples

The following examples explain the common usage of smart pointers in embedded programming.

WE'LL COVER THE FOLLOWING ^

- Example 1
  - Explanation
- Example 2
  - Explanation

Example 1

To get a visual idea of the life cycle of the resource, there is a short message in the constructor and destructor of `MyInt` (line 8 - 16).

```
// sharedPtr.cpp

#include <iostream>
#include <memory>

using std::shared_ptr;

struct MyInt{
  MyInt(int v):val(v){
    std::cout << "  Hello: " << val << std::endl;
  }
  ~MyInt(){
    std::cout << "  Good Bye: " << val << std::endl;
  }
  int val;
};

int main(){

  std::cout << std::endl;

  shared_ptr<MyInt> sharPtr(new MyInt(1998));
  std::cout << "    My value: " << sharPtr->val << std::endl;
  std::cout << "sharedPtr.use_count(): " << sharPtr.use_count() << std::endl;


  {
```

```
  {
    shared_ptr<MyInt> locSharPtr(sharPtr);
    std::cout << "locSharPtr.use_count(): " << locSharPtr.use_count() << std::endl;
  }
  std::cout << "sharPtr.use_count(): "<<  sharPtr.use_count() << std::endl;

  shared_ptr<MyInt> globSharPtr= sharPtr;
  std::cout << "sharPtr.use_count(): "<<  sharPtr.use_count() << std::endl;
  globSharPtr.reset();
  std::cout << "sharPtr.use_count(): "<<  sharPtr.use_count() << std::endl;

  sharPtr= shared_ptr<MyInt>(new MyInt(2011));

  std::cout << std::endl;

}
```

Explanation

- In line 22, we create `MyInt(1998)`, which is the resource that the smart pointer should address. By using `sharPtr->val`, we have direct access to the resource (line 23).

- The output of the program shows the numbers of the reference counter. It starts in line 24 with 1. It then has a local copy `shartPtr` in line 28 and goes to 2. The program then returns to 1 after the block (lines 27-30).

- The copy assignment call in line 33 modifies the reference counter. The expression `sharPtr= shared_ptr<MyInt>(new MyInt(2011))` in line 38 is more interesting.

- Firstly, the resource `MyInt(2011)` is created in line 38 and assigned to `sharPtr`. Consequently, the destructor of `sharPtr` is invoked. `sharedPtr` was the exclusive owner of the resource new `MyInt(1998)` (line 22).

- The last resource new `MyInt(2011)` will be destroyed at the end of main.

Example 2

You can quite easily push `std::shared_ptr` onto a `std::vector<std::shared_ptr<int>>` with different deleters. The special deleter will be stored in the control block.

In this example, we create a special `std::shared_ptr` that logs how much memory has already been released.

```
// sharedPtrDeleter.cpp

#include <iostream>
#include <memory>
#include <random>
#include <typeinfo>

template <typename T>
class Deleter{
public:
  void operator()(T *ptr){
    ++Deleter::count;
    delete ptr;
  }
  void getInfo(){
    std::string typeId{typeid(T).name()};
    size_t sz= Deleter::count * sizeof(T);
    std::cout << "Deleted " << Deleter::count << " objects of type: " << typeId << std::endl;
    std::cout <<"Freed size in bytes: "  << sz << "." <<  std::endl;
    std::cout << std::endl;
  }
private:
  static int count;
};

template <typename T>
int Deleter<T>::count=0;

typedef Deleter<int> IntDeleter;
typedef Deleter<double> DoubleDeleter;

void createRandomNumbers(){

  std::random_device seed;

  std::mt19937 engine(seed());

  std::uniform_int_distribution<int> thousand(1,1000);
  int ranNumber= thousand(engine);
  for ( int i=0 ; i <= ranNumber; ++i) std::shared_ptr<int>(new int(i),IntDeleter());

}

int main(){

  std::cout << std::endl;

  {
    std::shared_ptr<int> sharedPtr1( new int,IntDeleter() );
    std::shared_ptr<int> sharedPtr2( new int,IntDeleter() );
    auto intDeleter= std::get_deleter<IntDeleter>(sharedPtr1);
    intDeleter->getInfo();
    sharedPtr2.reset();
    intDeleter->getInfo();

  }
  createRandomNumbers();
  IntDeleter().getInfo();
```

```
  {
    std::unique_ptr<double,DoubleDeleter > uniquePtr( new double, DoubleDeleter() );

    std::unique_ptr<double,DoubleDeleter > uniquePtr1( new double, DoubleDeleter() );
    std::shared_ptr<double> sharedPtr( new double, DoubleDeleter() );

    std::shared_ptr<double> sharedPtr4(std::move(uniquePtr));
    std::shared_ptr<double> sharedPtr5= std::move(uniquePtr1);
    DoubleDeleter().getInfo();
  }

  DoubleDeleter().getInfo();

}
```

Explanation

- In lines 8 - 24, `Deleter` is the special deleter. The deleter is parametrized by the type `T`. It counts, alongside, the static variable `count` (line 23), how often the call operator (lines 11 - 14) was used.
- `Deleter` returns all the information with `getInfo` (lines 15-21).
- The function `createRandomNumbers` (lines 32-42) creates between 1 to 1000 `std::shared_ptr` (line 40) parametrized by the special deleter `intDeleter()`.
- The first usage of `intDeleter->getInfo()` shows that no resource has been released. This changes with the call `sharedPtr2.reset()` in line 53.
- An `int` variable with 4 bytes has been released. The call `createRandomNumbers()`, in line 57, creates 74 `std::shared_ptr<int>`.
- Of course, you can use the deleter for an `std::unique_ptr` (line 60 - 68).
- The memory for the double objects will be released after the end of the block, in line 68.

The classic issue of smart pointers using a reference count is to have cyclic references. Therefore, std::weak_ptr solves our problems. In the next lesson, we will take a closer look at `std::weak_ptr` and show you how to break cyclic references.