Object-Oriented Programming for Scientific Computing

Smart Pointers and Constness

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

28. April 2015

C++11 and Dynamic Memory Management

Smart Pointer

C++11 offers a number of so-called smart pointers that can help manage dynamic memory and in particular ensure the correct release of allocated memory. There are three different types of smart pointers:

- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `std::weak_ptr<T>`

The template argument `T` specifies the type of object the smart pointer points to. The C++11 smart pointers are defined in the header file `memory`.

`unique_ptr`

- In the case of `unique_ptr`s there is always exactly one smart pointer that owns the allocated data. If this pointer is destroyed (e.g. because the function in which it was defined exits or the destructor of the object to which it belongs is called), then the virtual memory is freed.
- Smart pointers and normal pointers (raw pointers) should not be mixed to avoid the risk of unauthorized access to already freed memory or double release of memory. Therefore, the allocation of memory must be placed directly in the constructor call of `unique_ptr`.
- An assignment of a normal pointer to a smart pointer is not possible (but a transfer in the constructor is).

Example for `unique_ptr`

```
#include<memory>
#include<iostream>

struct blub
{
  void doSomething()
  {}
};

int main()
{
   std::unique_ptr<int> test(new int);
   test = new int;  // not allowed: assignment from raw pointer
   int a;
   test = &a;        // not allowed: assignment from raw pointer
   std::unique_ptr<int> test5(&a); // allowed but dangerous
   *test = 2;        // normal memory access
   std::unique_ptr<int> test2(test.release()); // move to other
       pointer
   test = std::move(test2);         // assignment only using move
```

Example for `unique_ptr`

```
    test.swap(test2);                     // exchange with other pointer
    if (test == nullptr)                         // comparison
        std::cout << "test␣is␣nullptr" << std::endl;
    if (!test2)                                  // test for existence
        of object
        std::cout << "test2␣is␣nullptr" << std::endl;
    std::unique_ptr<int[]> test3(new int[32]);   // array
    test3[7] = 12;                               // access to array
    if (test3)                                   // access to raw pointer
        std::cout << "test3␣is␣" << test3.get() << std::endl;
    test3.reset();                               // release of memory
    if (!test3)
        std::cout << "test3␣is␣nullptr" << std::endl;
    std::unique_ptr<blub> test4(new blub);       // allocate object
    test4->doSomething();                        // use method of object
    std::unique_ptr<FILE, int(*)(FILE*)> filePtr(
     fopen("blub.txt", "w"), fclose);            // Create and close file
}
```

shared_ptr

- shared_ptrs point to memory that is used concurrently.
- Several shared_ptrs can point to the same memory location. The number of simultaneous shared_ptrs to the same resource is monitored with reference counting. The allocated memory is freed when the last shared_ptr pointing to it disappears.
- Apart from that the functionality of shared_ptr is the same as that of unique_ptr.
- When the first shared_ptr to an object is created, a manager object is created that manages both the allocated resources and a variable that counts how many pointers point to the resources at any given moment.
- For each copy of a shared_ptr the counter is incremented, and it is lowered each time a shared_ptr is deleted or modified to point to a different location. If the counter reaches zero, the resources are released.

`weak_ptr`

- If several objects have `shared_ptr`s pointing to each other, they can be kept alive artificially after their scope ends, because each object has at least one pointer in the circle pointing to it.
- In order to break such a circuit, the class `weak_ptr` has been created.
- A `weak_ptr` is not a real pointer. It can not be dereferenced and no methods can be invoked on it.
- A `weak_ptr` only observes a dynamically allocated resource and can be used to check if it still exists.
- If access to the resource is required, the method `lock()` of `weak_ptr` can be used to generate a `shared_ptr` to the resource. This then ensures the existence of the resource as long it is used.
- The manager object of a `shared_ptr` has another counter, the so-called weak counter, which in turn counts the generated `weak_ptr`s. While the allocated resource is released when no `shared_ptr` points on it, the manager object is released when in addition no `weak_ptr` points to it.

shared_ptr to this

- Sometimes a pointer pointing at this is needed. As one shouldn't mix smart pointers and raw pointers, a shared_ptr to this must be used.
- If this is realized by shared_ptr<T> blub(*this), then a new manager object will be created and the memory of the object is either not released or released to early.
- Instead, one derives the class from the template class enable_shared_from_this<T>. A pointer to this is then created with the method shared_from_this:

```
shared_ptr<T> blub = shared_from_this();
```

- During the creation of such a derived object in the constructor of a shared_ptr, a weak_ptr to the object itself is stored within the object. The method shared_from_this generates a shared_ptr out of this stored weak_ptr.

Example for `shared_ptr`

```
#include<memory>
#include<iostream>

class Base : public std::enable_shared_from_this<Base>
{
    void doSomething()
    {
        std::shared_ptr<Base> myObj = shared_from_this();
    }
};

class Derived : public Base
{};

int main()
{
   std::shared_ptr<int> testPtr(new int), testPtr2;
   testPtr2 = testPtr; // increases shared count
   std::cout << testPtr.use_count() << std::endl; // number of
       shared_ptrs
   testPtr.reset();    // decreases shared count, testPtr is
       nullptr
```

Example for `shared_ptr`

```
// weak pointer example
std::weak_ptr<int> weakPtr = testPtr2; // increases weak count
testPtr = weakPtr.lock();
if (testPtr)
    std::cout << "Object␣still␣exists" << std::endl;
if (weakPtr.expired())
    std::cout << "Object␣doesn't␣exist␣any␣more" << std::endl;
std::shared_ptr<int> testPtr3(weakPtr); // throws exception if
    object has vanished
// Casting of shared pointers
std::shared_ptr<Base> basePtr(new Derived);
std::shared_ptr<Derived> derivedPtr;
derivedPtr = std::static_pointer_cast<Derived>(basePtr); //
    create cast smart pointer sharing ownership with original
    pointer


}
```

Constant Variables

- For constant variables the compiler ensures that the content is not changed during program execution.
- Constant variables must be initialized when they are defined.
- They can not be changed later on.

```
const int numElements = 100;     // initialization
numElements = 200;               // not allowed, const
```

- Compared to the macros in C, constant variables are preferred, because they allow the strict type checking of the compiler.

Constant References

- References can be also defined as constant. The value pointed to by the reference cannot be changed (using the reference).
- Constant variables only allow constant references (since otherwise they might be changed using the reference).

```
int numNodes = 100;             // variable
const int& nn = numNodes;       // variable cannot be canged using
    nn
                                // but can be using numNodes
const int numElements = 99;     // initialization
int& ne = numElements;          // not allowed, const-correctness
                                // wouldn't be guaranteed anymore
const int& numElem = numElements; // allowed
```

- Constant references are a great way to pass a variable to a function without copying.

```
MatrixClass& operator+=(const MatrixClass& b);
```

Constant Pointers

For pointers there are two different types of constness. For a pointer it may be forbidden

- to change the contents of the variable to which it points. This is expressed by writing `const` before the type of the pointer:

```
char s[17];
const char* pc = s; // pointer to constant
pc[3] = 'c';        // error, content is const
++pc;               // allowed
```

- to change the address stored in the pointer (such a pointer effectively acts as a reference). This is expressed by writing `const` between the type of the pointer and the name of the pointer:

```
char* const cp = s; // const pointer
cp[3] = 'c';        // allowed
++cp;               // error, pointer is const
```

Constant Pointers

- Of course there is also the combination of both (which corresponds to a constant reference):

```
const char* const cpc = s; // const pointer to constant
cpc[3] = 'c';              // error, content is const
++cpc;                     // error, pointer is const
```

Constant Objects

- Objects can also be defined as constant.
- The user assumes that the content of a constant object doesn't change. This must be guaranteed by the implementation.
- Therefore, it isn't allowed to call methods that could change the object.
- Functions which will not violate the constness are marked by the addition of the keyword `const` after the argument list.
- The keyword is part of the name. There can be a `const` and a non-`const` variant with the same argument list.
- Important: the `const` must also be added to the method definition outside of the class.
- For constant objects only `const` methods can be called.

```
#include<iostream>
class X
{
  public:
    int blub() const
    {
        return 3;
    }
    int blub()
    {
        return 2;
    }
};

int main()
{
    X a;
    const X& b = a;
    std::cout << a.blub() << "␣" << b.blub() << std::endl;
    // produces the output "2 3"
}
```

Of course the behavior used here for illustrative purposes is misleading and should not be used.

Example: Matrix Class

```
double* MatrixClass::operator[](int i)
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}

const double* MatrixClass::operator[](int i) const
{
    if ((i<0)||(i>=numRows_))
    {
        std::cerr << "Illegal␣row␣index␣" << i;
        std::cerr << "␣valid␣range␣is␣(0:" << numRows_ << ")";
        std::cerr << std::endl;
        exit(EXIT_FAILURE);
    }
    return a_[i];
}
```

Using this we may write:

```
MatrixClass A(4,6,0.0);
for (int i=0;i<A.Rows();++i)
    A[i][i] = 2.0;
const MatrixClass E(5,5,1.0);
for (int i=0;i<E.Rows();++i)
    std::cout << E[i][i] << std::endl;
```

Returning a pointer to a constant will prevent the object being implicitly modified by the return value:

```
A[2][3] = -1.0;   // ok, no constant
E[1][1] = 0.0;    // compiler error
```

Physical and Logical Constness

When is a method `const`?

1. The object remains bitwise unchanged. That's how the compiler sees it (that's all it can check) and what it tries to ensure e.g. by treating all data members of a `const` object also as constants. This is also known as physical constness.
2. The object remains conceptually unchanged for the user of the class. This is referred to as a logical constness. But the compiler is unable to check the semantics.

Physical Constness and Pointers

- In our matrix class example with dynamic memory management, we have used a pointer of type `double **` to store the matrix.
- Making this pointer constant we obtain a pointer of type `double ** const`. This way it's only forbidden to change the memory address which is stored in the pointer but not the entries in the matrix.
- The compiler doesn't complain about the definition:

  ```
  double& MatrixClass::operator()(int i, int j) const;
  ```

  This therefore allows changing a constant object:

  ```
  const MatrixClass E(5,5,1.0);
  E(1,1)=0.0;
  ```

- It is even allowed to change the entries within the class itself:

  ```
  double& MatrixClass::operator()(int i,int j) const
  {
      a_[0][0]=1.0;
      return a_[i][j];
  }
  ```

Alternatives

- Using an STL container as in the first variant of the matrix class:

```
std::vector<std::vector<double> >
```

- In a `const` object this becomes a `const std::vector<std::vector<double> >`.
- Defining the access function

```
double& MatrixClass::operator()(int i, int j) const;
```

results in an error message from the compiler:

```
matrix.cc: In member function 'double&␣
    MatrixClass::operator()(int,␣int)␣const':
matrix.cc:63: error: invalid initialization of reference of
    type 'double&' from expression of type 'const␣double'
```

Alternatives (II)

- Returning entire vectors with:

  ```
  std::vector<double>& MatrixClass::operator[](int i) const;
  ```

  fails as well:

  ```
  matrix.cc: In member function 'std::vector<double,␣
      std::allocator<double>␣>&␣MatrixClass::operator[](int)␣
      const':
  matrix.cc:87: error: invalid initialization of reference of
      type 'std::vector<double,␣std::allocator<double>␣>&' from
      expression of type 'const␣std::vector<double,␣
      std::allocator<double>␣>'
  ```

Note: Using pointers it is easy to circumvent the compiler functionality for monitoring physical constness. Therefore it is appropriate to exercise caution when using `const` methods for objects that use dynamically allocated memory.

Logical Constness and Caches

- Sometimes it is useful to store calculated values with high computational cost in order to save computing time when they are needed several times.
- We add the private variables `double norm_` and `bool normIsValid_` to the matrix class and make sure that `normIsValid_` will always be initialized with `false` in the constructor.

Logical Constness and Caches

- Then it is possible to implement an infinity norm as follows:

```
double MatrixClass::InfinityNorm()
{
  if (!normIsValid_)
  {
    norm_ = 0.;
    for (int j = 0; j < numCols_; ++j)
    {
      double sum = 0.;
      for (int i = 0; i < numRows_; ++i)
        sum += fabs(a_[i][j]);
      if (sum > norm_)
        norm_=sum;
    }
    normIsValid_ = true;
  }
  return norm_;
}
```

- This function also makes sense for a constant matrix and doesn't semantically violate the constness.
- But the compiler doesn't allow it.

Solution

- One defines both variables as `mutable`.

```
mutable bool normIsValid_;
mutable double norm_;
```

- Members that are `mutable` can also be modified in `const` objects.
- This should only be applied when it's absolutely necessary and doesn't change the logical constness of the object.

Friend

In some cases it may be necessary for other classes or functions to access the protected members of a class.

Example: Simply linked list

- `Node` contains the data.
- `List` should be able to change the data of `Node`.
- The data of `Node` should be private.
- `List` is `friend` of `Node` and may therefore access private data.

Friend II

- Classes and free functions can be `friend` of another class.
- Such a `friend` may access the private data of the class.

Example `friend` class:

```
class List;

class Node
{
private:
    Node* next;
public:
    int value;
    friend class List;
};
```

Friend II

- Classes and free functions can be `friend` of another class.
- Such a `friend` may access the private data of the class.

Example `friend` function:

```
class MatrixClass
{
    friend MatrixClass invert(const MatrixClass&);
    // ...
};

...
MatrixClass A(10);
...
MatrixClass inv = invert(A);
```

Friend III

- Almost everything that can be written as a class method can also be programmed as a free `friend` function.
- All classes and functions which are `friend` logically belong to the class, as they build on its internal structure.
- Avoid `friend` declarations. They open up encapsulation and raise the cost of maintenance.

Build Systems

- Complex projects consist of various programs and libraries.
- Each program / library consists of many files (header and source files).
- Build systems are created to make things easier when compiling.

Goal:

- A build system knows how to create the programs and libraries from the files.
- If a file is modified, the project should be updated.
- Typically not all files must be recompiled.
- Recompile as many files as necessary . . . and as few as possible.

Build Systems

- Complex projects consist of various programs and libraries.
- Each program / library consists of many files (header and source files).
- Build systems are created to make things easier when compiling.

Goal:

- A build system knows how to create the programs and libraries from the files.
- If a file is modified, the project should be updated.
- Typically not all files must be recompiled.
- Recompile as many files as necessary . . . and as few as possible.

Choice of Build Systems

There are several different systems with different ranges of functionality:

- make
- mk
- SCons
- ant
- jam
- Rant
- built-ins or plugins of the IDE
- ...

Moreover, there are meta-systems which generate input files for other systems:

- automake/autoconf
- cmake
- qmake
- mkmf
- ...

Choice of Build Systems

There are several different systems with different ranges of functionality:

- make
- mk
- SCons
- ant
- jam
- Rant
- built-ins or plugins of the IDE
- ...

Moreover, there are meta-systems which generate input files for other systems:

- automake/autoconf
- cmake
- qmake
- mkmf
- ...

Makefiles

- make is a program that allows to translate only the files that have changed since the last compilation.
- A Makefile describes the files which belong to a project and how they are compiled and linked.
- Makefile rules are written in a functional language.
    - One describes *targets*, which depend on *prerequisites*.
    - *targets* and *prerequisites* normally correspond to files.
    - For individual *targets* one creates rules that define how they are generated using the *rerequisites*.
- Makefile rules are of the form

```
target-name: prerequisites-list
        build-rule
```

with the actual rule being indented with a TAB.

Easy Makefile Example

```
all: test_rational farey

farey: farey.o rational.o
        g++ farey.o rational.o -o farey

test_rational: rational_test.o rational.o
        g++ rational_test.o rational.o -o test_rational

rational_test.o: rational_test.cc rational.h
        g++ -c rational_test.cc

rational.o: rational.cc rational.h
        g++ -c rational.cc

farey.o: farey.cc rational.h
        g++ -c rational.cc
```

Advanced Makefile Example

```
# the compiler we want to use
CXX=g++

# some more variables
CPPSRC=$(wildcard *.cc)
OBJS=$(CPPSRC:.cc=.o)
APPS=test_rational farey

### build all apps
all: $(APPS)

# how to build the apps
farey: farey.o rational.o
test_rational: rational_test.o rational.o
# how to compile apps
%: %.o
        $(CXX) $? -o $@
# how to compile object files
%.o: %.cc
        $(CXX) $(CXXFLAGS) -c -o $@ $<
```

- make supports variables.
- Rules can be formulated generically for different targets,
- ... using several automatic variables.
- GNU make has several special extensions (e.g. wildcards).
- GNU make has several rules already built in.
- make can also directly clean up the created files
- and automatically check the dependencies with the help of the compiler.

Advanced Makefile Example

```
# the compiler we want to use
CXX=g++
CC=$(CXX)
# some more variables
CPPSRC=$(wildcard *.cc)
OBJS=$(CPPSRC:.cc=.o)
APPS=test_rational farey

### build all apps
all: $(APPS)

# how to build the apps
farey: farey.o rational.o
test_rational: rational_test.o rational.o
# we use implicit compilation rules
```

- `make` supports variables.
- Rules can be formulated generically for different `targets`,
- . . . using several automatic variables.
- `GNU make` has several special extensions (e.g. wildcards).
- `GNU make` has several rules already built in.
- `make` can also directly clean up the created files
- and automatically check the dependencies with the help of the compiler.

Advanced Makefile Example

```
# the compiler we want to use
CXX=g++
CC=$(CXX)
# some more variables
CPPSRC=$(wildcard *.cc)
OBJS=$(CPPSRC:.cc=.o)
APPS=test_rational farey

### build all apps
all: $(APPS)

# how to build the apps
farey: farey.o rational.o
test_rational: rational_test.o rational.o
# we use implicit compilation rules

### Dependencies
dep: .depends
# include .depends if file exists
-include .depends
# how to create .depends
.depends: $(SRC)
        $(CXX) -MM $? > .depends

### cleanup
clean:
        rm -f $(APPS) $(OBJS)
```

- `make` supports variables.
- Rules can be formulated generically for different targets,
- . . . using several automatic variables.
- `GNU make` has several special extensions (e.g. wildcards).
- `GNU make` has several rules already built in.
- `make` can also directly clean up the created files
- and automatically check the dependencies with the help of the compiler.

Alternative: IDEs

- Integrated Development Environments (IDEs) combine the properties of an editor, a build system and a debugger
- e.g. Eclipse C/C++ Development Environment (`http://www.eclipse.org/cdt`)
- Eclipse is powerful and open source, but also very complex

Multiple Header Inclusion Prevention

- One can use macros to prevent the repeated inclusion of a header file.
- The content of the header file is put inside a conditional block:

```
#ifndef _MYSPECIALHEADERFILE_
#define _MYSPECIALHEADERFILE_
// content of header file
#endif
```

- The first inclusion of the header results in the definition of the macro and parsing of the content of the header file.
- Subsequent inclusions skip the content, since the macro is already defined.