CHAPTER 2
Modern C++

C++ was thoroughly modernized in 2011 with the addition of a plethora of features and constructs borrowed from more recent programming languages. As a result, C++ kept its identity as the language of choice for programming close to the metal, high-performance software in demanding contexts, and at the same time, adapted to the demands of modern hardware, adopted modern programming idioms borrowed from the field of functional programming, and incorporated useful constructs into its syntax and standard library that were previously available only from third party libraries.

The major innovation is the new Standard Threading Library, which is explored in the next chapter. Since we are using new C++11 constructs in the rest of the text, this chapter selectively introduces some particularly useful innovations. A more complete picture can be found online or in up-to-date C++ textbooks. Readers familiar with C++11 may easily skip this chapter.

2.1 LAMBDA EXPRESSIONS

One of the most useful features in C++11, borrowed from the field of *functional programming*, is the ability to define anonymous function objects on the fly with the *lambda* syntax like in:

```
auto myLambda = [] (const double r, const double t)
{
    double temp = r * t; return exp( -temp);
};
```

The new *auto* keyword provides automatic type deduction, which is particularly useful with lambdas that produce a different compiler-generated type for every lambda declaration. A lambda declaration starts with a *capture* clause [], followed by the list of its arguments (a lambda is after all a function, or more exactly a *callable object*), and its *body*, the sequence of instructions that are executed when the lambda is called, just like the body of functions and methods. The difference is that lambdas are declared *within* functions. Once declared, they may be called like any other function or function object, for example:

```
cout << myLambda( 0.01, 10) << endl;
```

One powerful feature of lambdas is their ability to *capture* variables from their environment *on declaration*.

[] means no capture.
[=] means capture *by value*, that is, by copy, of all variables in scope *used in the lambda's body*.
[&] means capture all variables *by reference*.

We may also capture variables selectively with the syntax:

[x] means capture only , by value with this syntax or by reference with .
[=, &x, &y] means capture and by reference, and all others by value. Obviously means capture and by value and all others by reference.
[x, &y] means capture by value, by reference and nothing else.

For instance,

```
1  double mat = 10;
2  auto myLambdaRef = [&mat] (const double r)
3  {
4      return exp( -mat * r);
5  };
6  auto myLambdaCopy = [mat] (const double r)
7  {
8      return exp( -mat * r);
9  };
10
11 cout << myLambdaRef( 0.01) << endl;
12 //  exp( -10 * 0.01)
13 cout << myLambdaCopy( 0.01) << endl;
14 //  exp( -10 * 0.01)
15
16 mat = 20;
17 cout << myLambdaRef( 0.01) << endl;
18 //  exp( -20 * 0.01)
19 cout << myLambdaCopy( 0.01) << endl;
20 //  exp( -10 * 0.01)
```

Behind the scenes, the compiler creates a *function object* when we declare a lambda, that is, an object that defines the operator () (with the arguments of the lambda) and therefore is *callable* (like a function). The captured variables are implicitly declared as data members with a value type when captured by value and a reference type when captured by reference, initialized with the captured data on declaration. Hence, the syntax:

```
void main()
{
    double a, x;
    //  ...
    auto l = [=, &x] (const double y) { return a*x*y; }
    //  ...
    double z = l(y);
}
```

is equivalent to the (much heavier):

```
class Lambda
{
    const double  myA;
    const double& myX;

public:

    Lambda(const double a, const double& x) : myA(a), myX(x) {}

    operator() (const double y) const { return myA * myX * y; }
};

void main()
{
    double a, x;
    //  ...
    Lambda l(a, x);
    //  ...
    double z = l(y);
}
```

As a function object, a lambda can be passed as an argument or returned from functions. Functions that manipulate functions are called *higher-order functions*, and the standard <algorithm> library provides a vast number of these.

Lambdas are also incredibly useful as *adapters* and resolve a constant annoyance C++ developers face when calling functions with signatures inconsistent with their data. The Standard Template Library (STL), for instance, includes a wealth of useful generic algorithms. But to use these algorithms we must respect their functions' signatures. Say we hold a vector of times from today:

```
vector<double> times;
```

and we want to compute an annuity given a constant rate r. We could write a hand-crafted loop, of course:

```
double ann = 0.0;
for(size_t i = 0; i < times.size(); ++i)
{
    ann += exp(-r*times[i]);
}
```

but it is considered best professional practice to apply generic algorithms instead.[1] The computation we just conducted is a *reduction*, where a collection is traversed sequentially and an *accumulator* is updated for each element. The STL algorithm for reductions is $accumulate()$, located in the <numeric> header. The version of interest to us has the following signature:

```
template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,
              BinaryOperation op );
```

The type T of the accumulator in our case is double, as is *InputIt, so the function op that updates the accumulator acc for each element x must be consistent with the form:

```
double op(const double& acc, const double& x);
```

but our instruction for the update of the accumulator is:

```
acc += exp(-r*x);
```

and prior to C++11, it would have been such an annoyance to squeeze that line of code into the required signature that we would probably have ended up with the hand-crafted loop. With the lambda syntax, it takes a line to do that right:

```
1  double ann = accumulate(times.begin(), times.end(), 0.0,
2                [r] (const double& acc, const double& x)
3                {
4                  return acc + exp(-r*x);
5                });
```

There are of course many other uses of lambdas, and we will discuss a few later, but their ability to seamlessly adapt data to signatures is the

reason why we use them every day.

C++11 also provides dedicated adapter functions $bind()$ and $mem\_fn()$ in the <functional> header (the latter turning member functions $class.func()$ into free functions $func(class)$), although lambdas can also do this in more convenient manner. The syntax for $bind()$ in particular is rather peculiar and it is easier to achieve the exact same behavior with lambdas.

We will be working with lambdas throughout the book.

2.2 FUNCTIONAL PROGRAMMING IN C++

The introduction of lambdas is part of an effort to modernize C++ with idioms borrowed from the growing and fashionable field of functional programming. Although C++ does not, and never will, support functional programming idioms the way a language like Haskell does, C++ does support some key elements of functional programming, in particular *value semantics for functions* and *higher-order functions*.

Value semantics means that functions may be manipulated just like other types and in particular they can be assigned to variables and passed as arguments or returned as results by higher-order functions. Note that lambdas are literals for functions, which means that the instruction:

```
auto f = [] (const double x) { return 2*x; };
```

assigns a *function literal* to f in the same way we assign number or string literals in:

```
double x = 2.0;
string s = "C++11";
```

C++11 defines the *function* template class in the <functional> header as a unique class for holding functions and *anything callable*. That means that a concrete type like

```
function<double(const double)>
```

can hold anything that may be called with a double to return a double: a C style function pointer, a function object, including a lambda, or a member function bound to an object. An object of that type is itself callable of course, and it has value semantics, in the sense that it can be assigned or passed as an argument, or returned as a result from a higher-order function.

It looks peculiar and at first sight impossible in C++ to define a type based on the behavior rather than the nature of the objects it holds.[2] *function* is implemented with an interesting, advanced design pattern called *type erasure*. Unfortunately, this versatility comes with a cost. Type erasure necessarily involves the storage of the underlying objects on the heap. Hence, to initialize, assigning or copying a *function* object involves an allocation.[3] For this reason, we refrain from using this class despite its convenience, and manipulate functions as template types instead.[4]

Composition

As a first example, we consider the composition of functions, and write a (higher-order) function that takes two functions as arguments and returns the *function* resulting from their composition.

```
1  template<class F1, class F2=F1>
2  auto compose(const F1& f, const F2& g)
3  {
4      return [=](const auto& x) { return f(g(x)); };
5  }
```

We use the auto keyword so that types are deduced at compile time. Note that it is a function, not a number, that is returned. For instance, the following code creates a function by composing an exponential with a square root:

```
int main()
{
    auto f = compose([](const double x) { return exp(x); },
        [](const double x) { return sqrt(x); });
    cout << f(0.5) << endl;
}
```

Lambdas are obviously unnecessary here; they wrap the functions $exp()$ and $sqrt()$ without adapting anything. However, the following does not compile on Visual Studio:

```
int main()
{
    auto f = compose(exp, sqrt);
    cout << f(0.5) << endl;
}
```

Standard mathematical functions are overloaded so they work with many different types, and the compiler doesn't know which overload to pick to instantiate the templates. For this reason, we must explicitly state the function types when we compose standard functions, as follows:

```
auto f = compose<double(double)>(exp, sqrt);
```

We are not limited to numerical functions. Any function that takes an argument of type T_1 and returns a result of type T_2 (which we denote $f : T_1 -> T_2$) may be composed with any function $g : T_2 -> T_3$ to create a function $h : T_1 -> T_3$. We can imagine a function that creates a vector $1..n$ out of an unsigned integer:

```
vector<unsigned> generateVec(const unsigned n)
{
    vector<unsigned> result(n);
    generate(result.begin(), result.end(),
        [counter = 0]() mutable { return ++counter; });
    return result;
}
```

where *generate()* is an STL algorithm from the header <algorithm> that fills a sequence by repeated calls to a function, and the lambda is marked mutable because its execution modifies its internal data *counter*. We can code a function that sums up the values in a vector:

```
template <class T>
T accumulateVec(const vector<T>& v)
{
    return accumulate(v.begin(), v.end(), T());
}
```

where the STL *accumulate()* algorithm was discussed earlier. We could define a (particularly inefficient) way to compute the sum of the first n numbers by composition:

```
int main()
{
    auto h = compose(accumulateVec<unsigned>, generateVec);

    cout << h(100) << endl;
}
```

We could even design ways to compose functions of *multiple* arguments, either by binding or currying. We have to stop here and refer interested readers to a specialized publication like [21].

Lifting

Another useful idiom borrowed from functional programming is *lifting*. To lift a function means to turn it into one that operates on compound types. For instance, we may implement a lift that turns a scalar

function into a vector function that applies the original function to all the elements of a vector:

```
template <class F>
auto lift(const F& f)
{
     return [f](const vector<double>& v)
     {
          vector<double> result(v.size());
          transform(v.begin(), v.end(), result.begin(), f);
          return result;
     };
}
```

transform() is a generic STL algorithm from header <algorithm> that applies a unary function to all the elements in a collection. What is returned from *lift()* is not a vector but a function of a vector that returns a vector. It can be used as follows (we lift the *exp()* function into a *vExp()* that computes a vector of exponentials from a vector of numbers):

```
int main()
{
    auto vExp = lift<double(double)>(exp);

    vector<double> v = { 1., 2., 3., 4., 5. };
    vector<double> r = vExp(v);

    for_each(r.begin(), r.end(),
        [](const double& x) {cout << x << endl; });
}
```

for_each() is another generic algorithm from the <algorithm> header that sequentially applies an action to all the elements in a collection. We use it to display the entries in the result vector *r*.

As a (slightly) more advanced example, suppose we have a function that implements the Black and Scholes formula from [22]:

```
double blackScholes(const double spot,
    const double strike,
    const double expiry,
    const double vol);
```

We can lift it into a function that computes a vector of option prices from a vector of spots, but we must first turn it into a function of the spot alone by binding the other arguments. That could be done with a lambda, or with the $bind()$ function from the header <functional>:

```
#include <functional>
using namespace std;
using namespace placeholders;

int main()
{
    //  Create unary pricing function out of spot alone
    //      by binding the other arguments
    auto BSfromS = bind(
        blackScholes,    //  Function to bind
        _1,              //  spot = 1st arg of bound function
        100.,            //  strike = 100
        1.,              //  maturity = 1
        .10);            //  vol = 10

    //  Lift the unary function into a vector function
    auto vBlackScholes = lift(BSfromS);

    //  Apply the lifted function to a vector of spots
    vector<double> spots = { 50., 75., 100., 125., 150. };
    vector<double> calls = vBlackScholes(spots);

    //  Display results
    for_each(calls.begin(), calls.end(),
        [](const double& x) {cout << x << endl; });
}
```

More information about $bind()$ can be found online. It takes a function, followed by its arguments in order, and returns a new function. When we pass a value for an argument, the argument is bound to this value. When we pass a placeholder $\_n$, the argument is bound to the nth argument of the resulting function. In our example, we created a new function out of $blackScholes()$, by binding its first argument (the spot) to the first argument of the new function, and all other arguments to fixed values.

Alternatively, we could bind the spot and create a function that values a call out of volatility alone, and then lift it so it returns a vector of calls out of a vector of volatilities:

```
int main()
{
    //  Create unary pricing function out of vol alone
    auto BSfromSigma = bind(
        blackScholes,    //  Function to bind
        100.,            //  spot = 100
        120.,            //  strike = 120
        1.,              //  maturity = 1
        _1);             //  vol = (first) argument to created function

    //  Lift the unary into a vector function
    auto vBlackScholes = lift(BSfromSigma);

    //  Apply the lifted function to a vector of vols
    vector<double> vols = { .05, .10, .15, .20, .25, .50 };
    vector<double> calls = vBlackScholes(vols);

    //  Display results
    for_each(calls.begin(), calls.end(),
        [](const double& x) {cout << x << endl; });
}
```

Note that our lifting function is specialized for functions $double -> double$, lifting into a function $vector < double > -> vector < double >$. It is possible, with template magic, to produce a generic lifting function for functions $T_1 -> T_2$, lifting into $C < T_1 > -> C < T_2 >$ where C is an arbitrary collection, not necessarily a vector. This exercise is out of scope here, and we refer to specialized publications.

Functional programming idioms are exciting and fashionable. For an excellent introduction to functional programming in its natural habitat Haskell, we refer to [23].

We barely scratched the surface of functional programming in C++11, but hopefully gave a sense of how functions may be created and manipulated like any other type of data. It would take a dedicated publication to cover that subject in full, and, indeed, one such publication exists, [21], where interested readers will find a much more complete discussion of the implementation of functional programming idioms in C++.

2.3 MOVE SEMANTICS

Moving onto a different topic, the following pattern is valid but inefficient in traditional C++:

```
vector<double> f(/*...*/)
{
    vector<double> inner;
    //  ...
    return result;
}

int main()
{
    vector<double> outer = f(/*...*/);
}
```

The $inner$ vector is destroyed when f returns, but before that, the $outer$ vector is allocated and the contents of $inner$ are copied. This is of course very inefficient: memory is allocated twice, and an unnecessary duplication of data is conducted from a container that is destroyed immediately afterwards. This inefficiency *might* be caught by the compiler's RVO (Return Value Optimization), whereby the compiler would directly instantiate $outer$ inside f. RVO is not guaranteed[5] so programmers settled for a less natural syntax where result vectors are passed by reference as arguments rather than returned from functions.

C++11 *move semantics* permanently resolved this situation.

Conventional C++ allows class developers to implement their own *copy constructors* and *copy assignment operators*:

```
struct MyClass
{
    MyClass() { cout << "ctor" << endl; }
    ~MyClass() { cout << "dtor" << endl; }

    MyClass(const MyClass& rhs)
    {
        cout << "copy ctor" << endl;
    }

    MyClass& operator=( const MyClass& rhs)
    {
        if( this == &rhs) return *this;
        cout << "copy =" << endl;
        return *this;
    }
};
```

The code in the body of the copy constructor and assignment is automatically executed whenever an object of that type is initialized or assigned from another object of the same type. The code doesn't have to conduct a copy (our example does not) but it is expected that this is the case, and that such code should result in the duplication of the right-hand side (rhs) into the left-hand side (lhs).

When the developer does not supply a copy constructor or a copy assignment, the compiler provides default ones that perform copies of all data members (by calling their own copy constructors or assignment operators when these members are themselves classes).

C++11 introduced additional *move* constructors and assignments, with the perhaps unusual “&&” syntax:

```
MyClass(MyClass&& rhs)
{
    cout << "move ctor" << endl;
}

MyClass& operator=(MyClass&& rhs)
{
    if( this == &rhs) return *this;
    cout << "move =" << endl;
    return *this;
}
```

These are automatically invoked whenever the *rhs* is a *temporary object*, like a result returned from a function, as opposed to a *named* object. They can also be explicitly invoked with the *move()* keyword (which is actually a function):

```
MyClass someFunc(const MyClass& x)
{
    MyClass temp;
    //...
    return temp;
}

void myFunc()
{
    MyClass x, y, r;

    x = y;
    //  copy assign

    myClass z(y);
    //  copy construct

    myClass t(move(z));
    //  move construct

    r = move(t);
    //  move assign

    MyClass s = someFunc(r);
    //  temporary detected: auto-move!
}
```

We can (and, in the example, did) code whatever we want in the move constructor and assignment. What is expected is a quick transfer of the ownership of the *rhs* object's resources to the *lhs* object, without modification of the managed resources themselves, leaving the *rhs* empty.

Let us discuss a relevant example. If we wanted to code our own vector class wrapping a dumb pointer, we could proceed as follows (we simplify to the extreme and only show code for the core functionality). We start with the skeleton of a custom Vector class, including the copy constructor and copy assignment:

```
template<class T>
class Vector
{
    T*      myPtr;
    size_t  mySize;

public:

    Vector(size_t size = 0)
        : mySize(size), myPtr(size > 0? new T[size]: nullptr)
    {}

    ~Vector() { delete[] myPtr; }

    Vector(const Vector& rhs)
        :   mySize(rhs.mySize),
            myPtr(rhs.mySize > 0 ? new T[rhs.mySize] : nullptr)
    {
            copy(rhs.myPtr, rhs.myPtr + rhs.mySize, myPtr);
    }

    void swap(Vector& rhs)
    {
        std::swap(myPtr, rhs.myPtr);
        std::swap(mySize, rhs.mySize);
    }

    Vector& operator=(const Vector& rhs)
    {
        if (this != &rhs)
        {
            Vector<T> temp(rhs);
            swap(temp);
        }
        return *this;
    }

    void resize(size_t newSize)
    {
        if (mySize < newSize)
        {
            Vector<T> temp(newSize);
            copy(myPtr, myPtr + mySize, temp.myPtr);
            swap(temp);
        }
        mySize = newSize;
    }

    T& operator[] (const size_t i) { return myPtr[i]; }
    const T& operator[] (const size_t i) const { return myPtr[i]; }
    const size_t size() const { return mySize; }
    T* begin() { return myPtr; }
    const T* begin() const { return myPtr; }
    T* end() { return myPtr + mySize; }
    const T* end() const { return myPtr + mySize; }
};
```

The copy constructor clones the *rhs* Vector into *this*, implementing a memory allocation followed by a copy of the data.[6] The copy assignment operator does the same thing, but it must also release the data

previously managed by $this$ (for the copy constructor, $this$ is not yet constructed so it doesn't manage any data).

To avoid duplicating the copy constructor code into the copy assignment operator, we applied the well known “copy and swap” idiom. The method $swap()$ swaps the pointers (and sizes) of $*this$ and rhs Vectors, effectively swapping the *ownership* of data, without modifying the data itself in any way: after the swap, $this$ owns rhs's previous data, and rhs owns the data previously managed by $this$. In the assignment operator, we construct a temporary vector $temp$ by copy of rhs, and swap $this$ with $temp$. As a result, $this$ holds a copy of the previous contents of rhs and $temp$ manages the previous data of $this$. When $temp$ goes out of scope, its destructor is invoked and the data previously managed by $this$ is destroyed and its memory is released. A similar process is implemented in the resizer.

Copy semantics make a copy of rhs into $this$, without modification to rhs, at the cost of an expensive allocation and an expensive copy of the data. We now implement move semantics. A move is not supposed to copy any data, but transfer the ownership of rhs's resources to $this$ and leave rhs in an empty state. It follows that rhs is *not* a const argument to the move constructor:

```
//  ...
Vector(Vector&& rhs)
{
    swap(rhs);
    rhs.myPtr = nullptr;
}
//  ...
```

After the swap, $this->myPtr$ points on rhs's former data, $rhs.myPtr$ points on $this$'s former, uninitialized memory, since $this$ is not yet constructed. No data was copied, no memory allocated, and the ownership of rhs's data was swiftly and efficiently transferred to $this$, in exchange for some uninitialized memory. Therefore, rhs is empty after execution and $this$ effectively owns its former contents

and resources. For avoidance of doubt, we set $rhs.myPtr$ to $nullptr$ so its destructor would not attempt to deallocate it.

The move assignment is implemented in a similar way, the difference being, $this$ may own resources prior to the assignment, in which case they must be released. We move rhs into a temporary Vector, and swap $this$ with $temp$, so that $this$ ends up with the ownership of rhs's previous resources, rhs ends up empty after it was moved, and the previous resources of $this$, transferred to $temp$ in the swap, are released when $temp$ exits scope.

```
//  ...
Vector& operator=(Vector&& rhs)
{
    if (this != &rhs)
    {
        Vector<T> temp(move( rhs));
        swap(temp);
    }
    return *this;
}
//  ...
```

A move transfer is an order of magnitude faster than a copy. All it does is swap pointers, something called a *shallow copy*, without allocation of memory or copy of data. But it renders the moved object unusable after the transfer.

When the rhs object is *unnamed*, for example, returned from a function, then it couldn't possibly be reused in any way. The compiler knows this, so it always invokes move semantics in place of slower copy semantics in these situations. When the rhs is a named object, the compiler cannot safely move it, and it would normally make a copy. When *we* know that we don't reuse a named rhs object after the transfer, we can explicitly invoke its move semantics with $move()$.

In those situations where move semantics are invoked, either automatically or explicitly, on an object that doesn't implement them, the compiler falls back on to more expensive copy semantics. Copy seman-

tics are always implemented: when a copy constructor and assignment operator are not explicitly declared on a class, the compiler generates default ones, by copy of all data members. On the contrary, the compiler doesn't generally produce a default move constructor or move assignment operator. This only happens in restrictive cases. It follows that we must always declare move semantics explicitly in classes that manage memory or other resources, or expensive copies will be executed in situations where a faster move transfer could have been performed safely instead.

Move semantics are implemented in all STL containers and standard library classes out of the box. It is our responsibility to implement them in our own container classes and other classes that manage resources. For example, we update our simple matrix code with move semantics. We will use it in Parts II and III. The code below is in the file matrix.h on our repository.

```cpp
#include <vector>
using namespace std;

template <class T>
class matrix
{
    //  Dimensions
    size_t      myRows;
    size_t      myCols;

    //  Data
    vector<T>   myVector;

public:

    //  Constructors
    matrix() : myRows(0), myCols(0) {}
    matrix(const size_t rows, const size_t cols) :
        myRows(rows), myCols(cols), myVector(rows*cols) {}

    //  Copy, assign
    matrix(const matrix& rhs) :
        myRows(rhs.myRows), myCols(rhs.myCols), myVector(rhs.myVector) {}
    matrix& operator=(const matrix& rhs)
    {
        if (this == &rhs) return *this;
        matrix<T> temp(rhs);
        swap(temp);
        return *this;
    }

    //  Copy, assign from different (convertible) type
    template <class U>
    matrix(const matrix<U>& rhs)
        : myRows(rhs.rows()), myCols(rhs.cols())
    {
        myVector.resize(rhs.rows() * rhs.cols());
        copy(rhs.begin(), rhs.end(), myVector.begin());
    }
    template <class U>
    matrix& operator=(const matrix<U>& rhs)
    {
        if (this == &rhs) return *this;
        matrix<T> temp(rhs);
        swap(temp);
        return *this;
    }

    //  Move, move assign
    matrix(matrix&& rhs) :
        myRows(rhs.myRows),
        myCols(rhs.myCols),
        myVector(move(rhs.myVector)) {}

    matrix& operator=(matrix&& rhs)
    {
        if (this == &rhs) return *this;
        matrix<T> temp(move(rhs));
        swap(temp);
        return *this;
    }

```

```
    //  Swapper
    void swap(matrix& rhs)
    {
        //  Call std::vector::swap()
        myVector.swap(rhs.myVector);
        //  Call free function std:swap()
        ::swap(myRows, rhs.myRows);
        ::swap(myCols, rhs.myCols);
    }

    //  Resizer
    void resize(const size_t rows, const size_t cols)
    {
        myRows = rows;
        myCols = cols;
        if (myVector.size() < rows*cols) myVector = vector<T>(rows*cols);
    }

    //  Access
    size_t rows() const { return myRows; }
    size_t cols() const { return myCols; }
    //  So we can call matrix [i][j]
    T* operator[] (const size_t row) { return &myVector[row*myCols]; }
    const T* operator[] (const size_t row) const
        { return &myVector[row*myCols]; }
    bool empty() const { return myVector.empty(); }

    //  Iterators
    using iterator = typename vector<T>::iterator;
    using const_iterator = typename vector<T>::const_iterator;

    iterator begin() { return myVector.begin(); }
    iterator end() { return myVector.end(); }
    const_iterator begin() const { return myVector.begin(); }
    const_iterator end() const { return myVector.end(); }
};

//  Free function to transpose a matrix
template <class T>
inline matrix<T> transpose(const matrix<T>& mat)
{
    matrix<T> res(mat.cols(), mat.rows());
    for (size_t i = 0; i < res.rows(); ++i)
    {
        for (size_t j = 0; j < res.cols(); ++j)
        {
            res[i][j] = mat[j][i];
        }
    }

    return res;
}
```

2.4 SMART POINTERS

Smart pointers wrap standard (or *dumb*) pointers and implement the RAII (Resource Acquisition Is Initialization) idiom. RAII is a rather verbal name for an idiom that implements the release of resources in destructors, so that when an object exits scope, for whatever reason (the function returned or an exception was thrown), resources are always

automatically released. Smart pointers relieve developers from the concern of explicitly releasing allocated memory, and protect against memory leaks.

Smart pointers are otherwise manipulated just like dumb pointers; in particular they can be dereferenced with operators $*$ and $->$ to read and write the managed object.

C++ developers have been using smart pointers for decades, either hand-crafted or from third-party libraries like Boost. They are part of the standard C++ library since C++11.

A simplistic smart pointer could be coded as follows. This smart pointer cannot be copied, since the memory is owned and released on destruction by a single object. But it can be moved, in which case the rhs pointer loses ownership of memory when the lhs pointer acquires it.

```cpp
template <class T>
class SmartPointer
{
    //  Wrap a dumb pointer
    T*  myPtr;

public:

    //  Adopt ownership of memory
    SmartPointer(T* ptr = nullptr) : myPtr(ptr) {}

    //  This is the smart bit: release memory on destruction
    ~SmartPointer() { delete myPtr; }

    //  Forbid copy
    SmartPointer(const SmartPointer& rhs) = delete;
    SmartPointer& operator=(const SmartPointer& rhs) = delete;

    //  Implement move
    SmartPointer(SmartPointer&& rhs) : myPtr(rhs.myPtr)
    {
        rhs.myPtr = nullptr;
    }

    SmartPointer& operator=(SmartPointer&& rhs)
    {
        if (this != &rhs)
        {
            delete myPtr;
            myPtr = rhs.myPtr;
            rhs.myPtr = nullptr;
        }
        return *this;
    }

    //  Swapper
    void swap(SmartPointer& rhs)
    {
        std::swap(myPtr, rhs.myPtr);
    }

    //  Adopt dumb pointer
    void reset(T* ptr)
    {
        delete myPtr;
        myPtr = ptr;
    }

    //  Release ownership
    //      memory is not released and the managed object is not destroyed
    //      ownership is transferred to the returned dumb pointer
    T* release()
    {
        T* temp = myPtr;
        myPtr = nullptr;
        return temp;
    }

    //  Access managed object
    T* get() const
    {
        return myPtr;
    }

    T& operator*() const
    {
        return *myPtr;
    }

```

```
70      T* operator->() const
71      {
72          return myPtr;
73      }
74  };
```

The standard library smart pointer *unique_ptr* located in the <memory> header is implemented along these lines.

Importantly, the standard guarantees that *unique_ptr* s are manipulated without overhead compared to dumb pointers. Dynamic memory management with dumb pointers is inconvenient, prone to memory leaks, and without benefit compared to *unique_ptr* s. RAII management is a free benefit, and it is considered best practice to always manage heap memory with smart pointers.

We can create an object on the heap with the traditional operator *new* and assign it to a smart pointer for RAII management:

```
Object* object = new Object;
unique_ptr<Object> ptr(object);
//  RAII:
//      object is destroyed and its memory released
//      on destruction of ptr when ptr exists scope
```

or more simply:

```
unique_ptr<Object> ptr(new Object);
```

Since C++14, the free function *make_unique()* offers a terser, potentially more efficient syntax for the creation of objects in dynamic memory managed with a *unique_ptr* :

```
unique_ptr<Object> ptr = make_unique<Object>();
```

or more simply:

```
auto ptr = make_unique<Object>();
```

make_unique() also forwards its parameters to the managed object's contructor:

```
class Object
{
    string myName;

    // ...

public:
    private Object(const string& name) : myName(name) {}

    // ...
};

auto ptr = make_unique<Object>("object");
```

The second breed of standard smart pointers, *shared_ptr* s, also located in the <memory> header, offer further benefits, but not for free. Shared pointers are *reference counted*. They *can* be copied, in which case all copies *share* the ownership of the managed memory, and it is only when the *last* owner exits scope that the managed memory is released and its contents destroyed.

Shared pointers are very powerful because we never worry about memory being released too early or too late. A resource remains alive as long as there are pointers referencing it, and it is released automatically when this is no longer the case. But they are slower than dumb and unique pointers. Reference counting is not free, especially in a concurrent environment. For this reason, we must use *shared_ptr* s parsimoniously, always pass them by reference (to avoid unnecessary reference counting), and do nothing else than dereference them[7] at low level and especially in repeated code. Obviously, we only use *shared_ptr* s when we effectively need reference counting, and *unique_ptr* s otherwise.

Like *unique_ptr* s, *shared_ptr* s can adopt dumb pointers for RAII management, although this is implemented in a more convenient and more efficient manner in the factory function *make_shared()* which syntax is identical to *make_unique()* .

The following example demonstrates how the two types of standard library smart pointers work.

```cpp
#include <memory>
using namespace std;

struct MyClass
{

    ~MyClass() { cout << "dtor called" << endl; }

    void read() const
        { cout << "i am being read" << endl; }

    void write()
        { cout << "i am being written" << endl; }
};

void myFunc()
{
    {
        unique_ptr<MyClass> up2;

        {
            //  Create an object of type MyClass in dynamic memory,
            //      get a unique_ptr on the object
            unique_ptr<MyClass> up = make_unique<MyClass>();

            //  Manipulate the managed object through the unique_ptr
            up->read();
            up->write();
            //  Get a direct reference on the managed object
            MyClass& obj = *up;

            // up2 = up;    ERROR: not copyable
            up2 = move(up); //      OK: move

            // myClass& obj2 = *up; // ERROR:
            //      ownership no longer belongs to up

            //  up destroyed here,
            //  yet nothing happens: ownership was moved to up2
        }
        cout << "out of inner block" << endl;
        //   up2 destroyed here, object is destroyed and memory is released
    }
    {
        shared_ptr<MyClass> sp2;

        {
            //  Create an object of type MyClass in dynamic memory,
            //      get a shared_ptr on the object
            shared_ptr<MyClass> sp = make_shared<MyClass>();
            //  sp is the only pointer on the managed object
            //      reference count = 1

            //  Manipulate the managed object through the shared_ptr
            sp->read();
            sp->write();
            //  Get a direct reference on the managed object
            MyClass& obj = *sp;

            sp2 = sp;           // OK, resource now shared
            //  reference count = 2

            MyClass& obj2 = *sp; // OK, sp and sp2 reference the same object

            //  sp destroyed here, sp2 still alive
            //  reference count = 1
            //  managed object remains alive
        }
        cout << "out of inner block" << endl;
```

```
70          // sp2 destroyed here, reference count goes to 0,
71          //      object is destroyed and memory is released
72      }
73  }
```

C++11 comes with many, many more new features, the most useful probably being hash tables and variadic templates. They are covered in many textbooks and online resources. Readers wishing to investigate these matters in further detail are referred to Scott Meyers' [24]. An outstanding innovation is undoubtedly the Standard Threading Library investigated in the next chapter.

NOTES

1 For reasons explained for instance in Scott Meyers [18].

2 This is sometimes called “duck-typing”: if it walks like a duck and quacks like a duck, then it is a duck.

3 Visual Studio implements a small object optimization (SMO) whereby a small buffer is allocated for every object on the stack to minimize heap allocations.

4 The implementation of type erasure is outside of our subject; interested readers can find information online, in particular on Stack Overflow.

5 Although C++ rules are complex and in flux in that area, see for instance [24].

6 *copy()* is another generic algorithm from <algorithm>; its purpose and interface are self-explanatory from the code.

7 Dereference does not involve reference count: only the creation, copy, and destruction of *shared_ptr* s are expensive.