Object-Oriented Programming for Scientific Computing

Dynamic Memory Management

Ole Klein

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

21. April 2015

Administrativa

Current situation:

- Registered students: 84 (75 BaMa / 9 PhD)
- Expected audience: 15 - 25 people
- Up to now only one tutor for the whole lecture

$\Rightarrow$ seriously understaffed

6 credit points for everybody:

- Master students: lecture + exercises + final exam
- PhD students: lecture + exercises

Administrativa

In anticipation of a second tutor:

- A second exercise group slot is created, everybody from the Master group is moved to this slot
- You may freely choose, moving you just forces you to make an active decision
- The PhD group will be moved to the time slot with fewer Master students

Capacity for correction is severely limited, therefore

- You **must** hand in the exercises in groups of three to four students (normally I would allow single hand in)
- Exercise submission only digitally via GitLab, will be explained on Thursday
- PhD students must hand in solutions, but these are only corrected if capacities allow

Organization of Memory

Static Memory

- Here global variables, variables belonging to a namespace and static variables are created.
- The memory is allocated when the program starts and is kept until the program ends.
- The addresses of variables in static memory don't change while the program is running.

Stack (Automatic Memory)

- Here local and temporary variables are created (e.g. for function calls or return values).
- The allocated memory is automatically freed when the variable goes out of scope (e.g. when leaving the function in which it is defined).
- The size of the stack is limited (e.g. in Ubuntu by default 8192kb).

Organization of Memory

Heap (Free Memory)

- Can be requested by the program with the command `new`.
- Must be released with the command `delete`.
- Is in general limited only by the size of the main memory.
- May get lost due to programming errors.

Variables

- A variable designates a memory location in which data of a certain type can be stored.
- A variable has a name and a type.
- The amount of memory required for a variable depends on its type.
- The amount of memory that is required for a particular type of variable can be retrieved using the function `sizeof(variable type)`.
- Each variable has a memory address that can be queried with the address operator `&`.
- The address of a variable cannot be modified.

References

- A reference only defines a different name for an already existing variable.
- The type of the reference is the type of the variable followed by a `&`.
- A reference is initialized the moment it is defined and cannot be changed thereafter. It therefore always points to the same variable.
- A reference can be used in exactly the same way as the original variable.
- Modifications of the reference also change the content of the original variable.
- There can be multiple references to the same variable.
- A reference can also be initialized with a reference.

Example for References

```
#include<iostream>

int main()
{
    int  a = 12;
    int& b = a;   // defines a reference
    int& c = b;   // is allowed
    float& d = a; // is not allowed, type mismatch
    int e = b;
    b = 2;
    c = a * b;
    std::cout << a << std::endl;
    std::cout << e << std::endl;
}
```

Pointers

- Pointers are a concept that is closely linked to the hardware.
- A pointer can store the address of a variable of a certain type or the address of a function.
- The type of a pointer variable is the type of the underlying variable followed by an asterisk $*$.
- Pointer contain memory addresses of variables, changing the pointer changes the memory location it points to.
- If one wants to access the value at that memory address, one places a $*$ in front of the name of the pointer.
- If a pointer points at an object and one wants to access the attributes or methods of the object, one can use the operator `->`. The expressions `*a.value` and `a->value` are equivalent.

Pointers

- If a pointer is not initialized during its definition, then it just points at a random memory location.
- If a pointer points to a memory location that wasn't assigned to the program by the operating system and reads or writes to the value at that address, then the program will terminate with an error message named `segmentation fault`.
- To clearly mark a pointer as not pointing to a variable or function one assigns the value `0`. In C++11 the special keyword `nullptr` can be used for this.
- This makes it simple to test whether a pointer is valid.

Pointers

- There are also pointers pointing to pointers, e.g.

```
int   a = 2;
int*  b = &a;
int** c = &b;
```

- The increment and decrement Operators ++/-- increase a pointer not by one byte, but by the size of the variable type to which the pointer points (the pointer then points to the “next” element) .
- If a number *i* is added/substracted from a pointer, then the memory address changes by *i* times the size of the variable to which the pointer points.

Example for Pointers

```
#include<iostream>

int main()
{
    int a = 12;
    int* b = &a; // defines a pointer to a
    float* c;    // defines a pointer to floats (pointing
                 // to somewhere unspecified)
    double* d = nullptr;  // better this way
    float e;
    c = &e;
    *b = 3;      // modifies variable a
    b = &e;      // not allowed, wrong type
    e = 2 * *b;  // allowed, equivalent to *c = 2 * a
    std::cout << b << std::endl;
    b = b + a;   // is allowed, but risky
                 // b now points to another memory cell
    std::cout << a << std::endl;
    std::cout << d << std::endl;
    std::cout << b << std::endl;
}
```

Arrays in C (and C++)

- Arrays in C are closely related to pointers.
- The name of an array in C is also a pointer to the first element of the array.
- The use of the bracket operator `a[i]` corresponds to a pointer operation `*(a+i)`

```
#include<iostream>

int main()
{
    int numbers[27];
    for (int i = 0; i<27; ++i)
        numbers[i] = i*i;
    int* end = numbers + 26;
    for (int* current = numbers; current<=end; ++current)
        std::cout << *current << std::endl;
}
```

Risks of Pointers

While dealing with pointers and arrays in C/C++, there are two major threats:

1. A pointer (particularly in the use of arrays) will be modified (accidentally or on purpose), so that it points to memory areas which haven't been allocated. At best, this leads to closing of the program due to a `segmentation fault`. In the worst case it can be used to gain access to the operating system.
2. Data is written beyond the end of an arry. If the affected memory was allocated by the program (because other variables are stored in that location), this often leads to very strange errors, because these other variables suddenly contain wrong values. In large programs the exakt spot where this happens may be hard to find.

Call by Value

If an argument is passed to a function, then a local copy on the stack is created for this argument with each function call.

- If a normal variable is in the argument list, then a copy of this variable is generated.
- This is called *Call by Value*.
- Modification of the variables within the function does *not* change the original variable where the function was called.
- If large objects are passed this way, then generating this copy can become very expensive (running time, memory requirements).

```
double SquareCopy(double x)
{
    x = x * x;
    return x;
}
```

Call by Reference

- If a reference or a pointer is in the list of argument, then copies of the reference or of the pointer can be generated. These still point to the same variable.
- This is called *Call by Reference*.
- Changes in the contents of the reference or the memory cell to which the pointer points effect the original variable.
- This allows writing functions that return more than one value and functions with an effect but without return value (procedures).
- A constant reference, e.g. `double Square(const double &x)`, can be used to pass large objects as an argument while preventing modification of the original.

```
void Square(double &x)
{
    x = x * x;
}
```

Dynamic Memory Management

Large objects, or arrays with a size that is determined during runtime, can be allocated on the heap with the help of `new`.

```
class X
{
  public:
    X();       // constructor without arguments
    X(int n); // with an int argument
    ...
};

X* p = new X;      // constructor without arguments
X* q = new X(17); // with an int argument
...
```

Dynamic Memory Management

Objects which are produced with `new` don't have a name, only an address in memory. This has two consequences:

1. The lifetime of the object isn't fixed. The programmer must destroy it explicitly with the command `delete` :

   ```
   delete p;
   ```

   This can only be done once per reserved object.
2. In contrast, the pointer used to access this object usually has a limited lifespan.

$\Rightarrow$ Object and pointer must be managed consistently.

Possible Problems

1. The pointer no longer exists, but the object is still existing ⇒ memory is lost, the program gets bigger and bigger (memory leak).
2. The object is no longer existing, but the pointer does ⇒ accessing the pointer creates a `segmentation fault`. Especially dangerous when several pointers point at the same object.

These two issues will be addressed by smart pointers introduced later in the lecture.

Allocating Arrays

- Arrays are allocated by writing the number of elements in brackets behind the type of variable.
- Arrays can only be allocated if the class has a constructor without arguments.
- Arrays are deleted with `delete []`. The implementation of `new []` and `delete []` may by incompatible with that of `new` and `delete`, e.g. in some implementations the length of the array is stored before the data and a pointer pointing to the actual data is returned.

```
int n;
std::cin >> n;    // user enters desired length of array
X* pa = new X[n];
...
delete [] pa;
```

⇒ One must not mix the different forms of `new` and `delete`. For individual variables `new` and `delete` are used, and for arrays `new []` and `delete []`.

Releasing Dynamically Allocated Memory

- Caling `delete` or `delete []` for a pointer that points to a location that has already been freed or wasn't reserved results in a `segmentation fault`.
- Passing a null pointer to `delete` and `delete []` is harmless.
- The C memory commands `malloc` and `free` should not be used in C++ programs.

Classes with Dynamically Allocated Members

Wrapping the dynamic memory management with a class definition

- Can hide the details of dynamic memory usage from the users
- Fixes (if correctly programmed) some of the major disadvantages of dynamically allocated memory in C

Issues of raw pointers that are addressed:

- Call by value becomes possible
- Objects are able to know their size
- If an object is destroyed, the destructor can automatically release dynamically allocated memory

Example: Matrix Class with Dynamic Memory

- The data is stored in a two-dimensional dynamically allocated array.
- Instead of the vector of vectors, the matrix class receives a pointer to a pointer of `double` as private menber.

```
double **a_;
int numRows_;
int numCols_;
```

- Methods to implement: constructor(s), destructor, copy constructor, assignment operator

Constructors

```
MatrixClass() : a_(0), numRows_(0), numCols_(0)
{};

MatrixClass(int dim) : a_(0)
{
    Resize(dim,dim);
};

MatrixClass(int numRows, int numCols) : a_(0)
{
    Resize(numRows,numCols);
};

MatrixClass(int numRows, int numCols, double value) : a_(0)
{
    Resize(numRows,numCols,value);
};
```

Resize Methods

```
void MatrixClass::Resize(int numRows, int numCols)
{
    Deallocate();
    a_= new double*[numRows];
    a_[0] = new double[numRows*numCols];
    for (int i=1;i<numRows;++i)
        a_[i]=a_[i-1]+numCols;
    numCols_=numCols;
    numRows_=numRows;
}

void MatrixClass::Resize(int numRows, int numCols, double value)
{
    Resize(numRows,numCols);
    for (int i=0;i<numRows;++i)
        for (int j=0;j<numCols;++j)
        a_[i][j]=value;
}
```

Destructor

```
~MatrixClass()
{
    Deallocate();
};

private:
inline void Deallocate()
{
    if (a_!=0)
    {
        if (a_[0]!=0)
            delete [] a_[0];
        delete [] a_;
    }
}
```

Copy Constructor and Assignment Operator

The default versions of copy constructor and assignment operator create a direct copy of all the variables. This would mean that now two pointers point to the same dynamically allocated data.

```
MatrixClass(const MatrixClass &b) : a_(0)
{
    Resize(b.numRows_,b.numCols_);
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]=b.a_[i][j];
}

MatrixClass &operator=(const MatrixClass &b)
{
    Resize(b.numRows_,b.numCols_);
    for (int i=0;i<numRows_;++i)
        for (int j=0;j<numCols_;++j)
            a_[i][j]=b.a_[i][j];
    return *this;
}
```

Further Adjustments

The bracket operators still need to be adapted (actually this only affects the return type). The parenthesis operators require no changes:

```
double *operator[](int i);
const double *operator[](int i) const;
```

The implementation of matrix-vector product and Gauss algorithm for this variant of the matrix class is omitted.

Static Variables

- Sometimes classes have members which exist only once for all objects of the class.
- These variables are of type `static`, e.g. `static int max`.
- In a program there is exactly one version of a static member (not one version per object), and memory for the member is only occupied once.
- Methods that don't work with the data of a specific object (i.e. use at most static variables) can also be defined as static member functions.
- Prefixing the name of the class followed by two colons, one can access the static attributes and methods without creating a temporary object.
- (Non-constant) static attributes must be initialised outside of the class.

Static Variables

```
#include<iostream>
class NumericalSolver
{
    static double tolerance;
  public:
    static double GetTolerance()
    {
        return tolerance;
    }
    static void SetTolerance(double tol)
    {
        tolerance=tol;
    }
};

double NumericalSolver::tolerance = 1e-8;

int main()
{
    std::cout << NumericalSolver::GetTolerance() << std::endl;
    NumericalSolver::SetTolerance(1e-12);
    std::cout << NumericalSolver::GetTolerance() << std::endl;
}
```

C++11 and Dynamic Memory Management

Temporary Objects

Problem: If a value is e.g. returned by a function, temporary objects may be created. The following function may create up to two temporary objects when it returns:

```
double SquareCopy(double x)
{
    return x*x;
}
```

- A temporary object stores the result of `x*x`.
- Since this object is created inside the function and will be deleted when the function exits, a copy of the return value is generated.

Copying large amounts of data can be quite time consuming. This is for the most part optimized by C++ compilers (return value optimisation, RVO).

C++11 and Dynamic Memory Management

Move Constructors

Idea: Since the temporary objects are directly destroyed after use, it isn't necessary to copy the data. It can be "recycled" by other objects. (There are also other applications). In C++11, there are explicit constructs for this:

- Move constructors and move-assignment operators reuse the contents of another (usually temporary) object. The members of this other object are replaced with default values (which are cheap to produce).
- This is applicable during initialization of objects, for the transfer of function arguments and for return values of functions.
- If the object is not temporary, the compiler needs to be explicitly informed that resources can be acquired. This is done with the keyword `std::move()`, e.g.

```
MatrixClass a(10,10,1.0);
MatrixClass b = std::move(a); // now b is a 10x10 Matrix
std::vector<double> x(10,1.0);
x=b.Solve(std::move(x));     // call of the function
```

Efficient Swap Using Move Semantics

The following code snippet copies a presumably large matrix three times:

```
void swap(MatrixClass& a, MatrixClass& b)
{
  MatrixClass tmp(a); // creates complete copy
  a = b;              // as above
  b = tmp;            // as above
}
```

All three lines copy from a location that is overwritten or discarded later on. Move semantics can be used to avoid the expensive copies:

```
void swap(MatrixClass& a, MatrixClass& b)
{
  MatrixClass tmp(std::move(a)); // uses the memory of a
  a = std::move(b);              // uses the memory of b
  b = std::move(tmp);            // uses the memory of tmp
}
```

- Move constructors (and move-assignment operators) are automatically created in C++11 if for a user-defined class no constructor, move constructor, assignment operator or destructor has been defined and if it is trivial to generate a move constructor.
- In other cases, the generation of a default move constructor or assignment operator follows the same rules as for normal constructors with the keyword `default`, e.g.:

  ```
  MatrixClass(MatrixClass &&) = default;
  ```

  (`MatrixClass &&` is a so-called r-value reference, which can only refer to temporary objects or objects marked with `std::move` and which was first introduced in C++11)
- A move constructor is trivial when:
  - The class pocesses neither virtual functions nor virtual base classes.
  - The move constructor for each direct base class of the class is trivial.
  - The move constructor of all non-static attributes is trivial.
- All standard data types which are compatible with C are trivially movable.
- The move concept works not only for memory but also for other resources, such as files or communicators.

Summary

- Memory is divided into three parts, static, automatic (Stack) and dynamic (Heap)
- References and pointers are two different ways of indirection when dealing with variables
- Pointers are more flexible but also much more dangerous
- Hiding dynamic memory management inside classes avoids pitfalls and reduces complexity
- C++11 introduces move semantics that reduce the number of unnecessarily created temporary variables