Object-Oriented Programming for Scientific Computing

Dynamic Memory Management

Ole Klein

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

Summer Semester 2018

Memory Layout

The amount of memory that is available to a program is divided into three main areas:

- Static memory
- Stack (automatic memory)
- Heap (free memory)

Each of these has its purpose and way of being accessed, explicitly or implicitly

Apart from these three, there are read-only areas for code and data (e.g., literals)

Static Memory

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

Memory Layout

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).
- Thread-parallel processes create one stack per thread.

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

- Variables represent memory locations 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
  - can be retrieved using the function `sizeof(variable type)`
- Each variable has a memory address that
  - can be queried with the address operator `&`
  - can't be modified

References

- References only define different names for already existing variables
- The type of the reference is the type of the variable followed by an &
- 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; //  4
  std::cout << e << std::endl; // 12
}
```

Dangling References

References are aliases that may persist when the original variable goes out of scope and is destroyed. This can be dangerous:

```
#include <iostream>

class Foo
{
     const int& k;
   public:
     Foo(const int& k_) : k(k_) {}
     int get() { return k; }
};

Foo f(bool toggle)
{
  int i = 123;
  int j = 456;
  if (toggle)
    return Foo(j);
  else
    return Foo(i);
}
```

Dangling References

```
  Foo g(int i)
  {
    return Foo(i);
  }

  int main()
  {
    Foo a = f(false);
    std::cout << "a: " << a.get() << "\n\n"; // 123?
    Foo b = f(true);
    std::cout << "a: " << a.get() << "\n";   // 123?
    std::cout << "b: " << b.get() << "\n\n"; // 456?
    Foo c = g(789);
    std::cout << "a: " << a.get() << "\n";   // 123?
    std::cout << "b: " << b.get() << "\n";   // 456?
    std::cout << "c: " << c.get() << "\n\n"; // 789?
}
```

Dangling References

Expected output:

```
a: 123

a: 123
b: 456

a: 123
b: 456
c: 789
```

Actual output:

```
a: 123

a: 123
b: -571951913

a: 789
b: -571951913
c: 32556
```

- a produces correct output because it uses an address on the stack that has not been used again since
- value in `b` is overwritten by `cout` call
- function `g` uses same space as `f` on stack and therefore overwrites value in “unrelated” `a`

- While references act like variables in most regards, the original variable determines the lifetime of the object / memory location
- Exception: `const` references to temporary values (e.g. return values)
- *Don’t store references as members if you cannot guarantee the integrity of the original data*

Pointers

- Pointers are a concept closely linked to the hardware
- A pointer can store the address of
    - a variable of a certain type
    - a function
- The type of a pointer variable is the type of the underlying variable followed by an asterisk `*`, e.g. `int*` for a pointer to `int`
- Pointers 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 to 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` (or constant `NULL`). Since C++11 the special keyword `nullptr` can be used for this.
- This makes it easy to test whether a pointer is valid.

Pointers

- There are also pointers of 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 to / 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; // e.g. 0x7ffc439b83b8
  b = b + a;   // is allowed, but risky
               // b now points to another memory cell
  std::cout << a << std::endl; // 3
  std::cout << d << std::endl; // 0
  std::cout << b << std::endl; // e.g. 0x7ffc439b83c4
}
```

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 its first element
- 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 array. 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 exact 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 are 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 affect 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.

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

Possible Problems

Complications that may arrise due to inconsistent management of dynamic memory:

1. The pointer no longer exists, but the object does
   $\Longrightarrow$ memory is lost, the program gets bigger and bigger (memory leak).
2. The object no longer exists, but the pointer does
   $\Longrightarrow$ accessing the pointer creates a `segmentation fault`. Especially dangerous when several pointers point to 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;
```

$\Longrightarrow$ 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

- Calling `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.
- This means it's exactly the dangerous situation (pointer to non-existent object) that can't be checked.

- The C memory commands `malloc` and `free` should not be used in C++ programs.
- Exception: inclusion of C libraries that require the user to manage memory (should be limited to a wrapper class around the library).

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
- Destructors 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 member.

```
double** entries;
int numRows;
int numCols;
```

- This is actually a step back from the `vector` implementation, but serves as an illustration
- Methods to implement: constructor (s), destructor, copy constructor, assignment operator
- Since C++11: move constructor, move assignment operator (later)

Constructors

```
Matrix() : entries(nullptr), numRows(0), numCols(0)
{};

Matrix(int dim) : entries(nullptr)
{
    resize(dim,dim);
};

Matrix(int numRows_, int numCols_) : entries(nullptr)
{
    resize(numRows_,numCols_);
};

Matrix(int numRows_, int numCols_, double value)
    : entries(nullptr)
{
    resize(numRows_,numCols_,value);
};
```

Resize Methods

```
void Matrix::resize(int numRows_, int numCols_)
{
    deallocate();
    entries= new double*[numRows_];
    entries[0] = new double[numRows_*numCols_];
    for (int i = 1; i < numRows_; ++i)
        entries[i] = entries[i-1] + numCols_;
    numCols = numCols_;
    numRows = numRows_;
}

void Matrix::resize(int numRows_, int numCols_, double value)
{
    resize(numRows_,numCols_);
    for (int i = 0; i < numRows; ++i)
        for (int j = 0; j < numCols; ++j)
            entries[i][j] = value;
}
```

Destructor

```
~Matrix()
{
    deallocate();
};

private:
void deallocate()
{
    if (entries != nullptr)
    {
        if (entries[0] != nullptr)
            delete [] entries[0];

        delete [] entries;
        entries = nullptr;
    }
}
```

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 two pointers pointing to the same dynamically allocated data.

```
Matrix(const Matrix& b) : entries(nullptr)
{
    resize(b.numRows,b.numCols);
    for (int i = 0; i < numRows; ++i)
        for (int j = 0; j < numCols; ++j)
            entries[i][j] = b.entries[i][j];
}

Matrix& operator=(const Matrix& b)
{
    resize(b.numRows,b.numCols);
    for (int i = 0; i < numRows; ++i)
        for (int j = 0; j < numCols; ++j)
            entries[i][j] = b.entries[i][j];
    return *this;
}
```

Further Adjustments

- The bracket operators still need to be adapted (which actually only affects the return type):

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

- The parenthesis operators require no changes, since they don't reference implementation details that differ from `vector`.
- The implementation of matrix-vector product and Gauss algorithm for this variant of the matrix class is omitted. They are virtually identical — except for the local copy of the matrix entries in the Gauss algorithm.

Static Variables

- Sometimes classes have members which exist only once for all objects.
- 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 per object), and memory for this 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 up to C++17, where inline non-const static members were introduced:

```
class X
{
  public:
    inline static int max = 10;
};

...
int i = X::max;
```

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 on the outside.

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

```
Matrix a(10,10,1.);
Matrixs b = std::move(a);     // now b is a 10x10 Matrix
std::vector<double> x(10,1.);
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(Matrix& a, Matrix& b)
{
  Matrix 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(Matrix& a, Matrix& b)
{
  Matrix tmp(std::move(a)); // uses memory of a
  a = std::move(b);         // uses memory of b
  b = std::move(tmp);       // uses memory of tmp
}
```

- Move constructors (and move-assignment operators) are automatically created in C++11 if there are no user-declared copy constructors, copy-assignment operators and destructors.
- 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.:

  ```
  Matrix(Matrix&&) = default;
  ```

  (`Matrix&&` 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)
- 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.

Example: Matrix Class

Copy constructor (has to create deep copy):

```
Matrix(const Matrix& b) : entries(0)
{
    resize(b.numRows,b.numCols);
    for (int i = 0; i < numRows; ++i)
        for (int j = 0; j < numCols; ++j)
            entries[i][j] = b.entries[i][j];
}
```

Move constructor (steals resources from other matrix):

```
Matrix(Matrix&& b) : entries(b.entries),
    numRows(b.numRows), numCols(b.numCols)
{
    // ensure that b remains valid matrix object
    b.entries = nullptr;
    b.numRows = 0;
    b.numCols = 0;
}
```

Rule of Three / Five

Rule of Three:

If any of the three methods

- Copy constructor
- copy-assignment operator
- Destructor

is user-defined, it is usually a good idea to implement all three (due to shared responsibility for memory management)

Rule of Two:

If RAII (e.g. smart pointers, next lecture) is used, the destructor can be omitted

Rule of Five:

Since C++11, the above rule is extended to move constructors and move-assignment operators, i.e. if one of the five is implemented it often a good idea to implement them all

Leaving out move constructors / operators when they are not automatically created can lead to decreased performance.

Summary

- Memory is divided into three parts:
  - static (global variables)
  - automatic (Stack, local variables)
  - dynamic (Heap, user-controlled)
- 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