# Programming Techniques for Scientific Simulations I

More on classes

# Review: Classes

- Are a way to create new data types ("extended data structure")
  - E.g. vector, matrix, genome, animal, frog, ... type
- Object oriented programming:
  - Instead of asking: "What are the subroutines?"
  - We ask:
    - What are the abstract entities?
    - What are the properties of these entities?
    - How can they be manipulated?
    - Then we implement these entities as classes
- Principles
  - High level of abstraction possible (**Abstraction**)
  - Hiding of representation dependent details (**Encapsulation**)
  - Basing one identity upon another retaining similar properties (**Inheritance**)
  - Single interface to identities of different types (**Polymorphism**)

# Review: What are classes?

- Classes are collections of "members" representing one entity
- Members can be
  - Functions
  - Data
  - Types
- These members can be split into
  - `public`: accessible interface to the outside. Should not be modified later!
  - `private`: hidden representation of the concept. Can be changed without breaking any program using the class.
  - `protected`: like private but also available to derived types. Should be used only for member functions (not data!)
- Objects are instances of a class
- Objects of this type can be modified only through these member functions -> localization of access, easier debugging

# Classes: Special member functions

- **Default constructor**
  - Creates objects

```
C(); // default ctor
```

- **Destructor**
  - Destroys objects

```
~C(); // dtor
```

- **Copy constructor**
  - Creates (new) copies of (existing) objects

```
C(C const&); // copy ctor
```

- **Copy assignment**
  - Copies (existing) objects into other (existing) objects

```
C& operator=(C const&); // copy assignment
```

- **Move constructor** (C++11)
  - Moves (temporary) objects into (new) objects

```
C(C&&); // move ctor
```

- **Move assignment** (C++11)
  - Moves (temporary) objects into (existing) objects

```
C& operator=(C&&); // move assignment
```

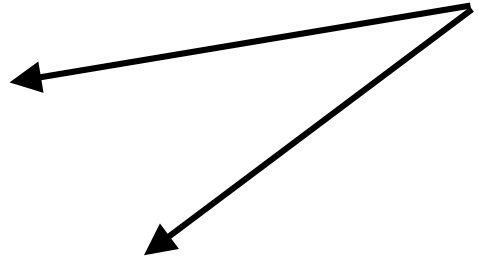
Special because they may be generated implicitly by the compiler under certain circumstances (more on that soon)

# Classes: Constructors

- Default constructor
  - Called when objects of a class are declared, but not initialized with any arguments
  - Default because it could be called with no arguments

```
class C {
  public:
    C() {/* ... */ }
  private:
    // ...
};

class Complex {
  public:
    Complex(double re = 0., double im = 0.) {/* ... */ }
  private:    // ...
};
```



Can take arguments, but has default arguments!

- Constructors can be overloaded
- If **no** constructor defined, a default one is generated by the compiler
- Declared explicit: prevents implicit type conversions (example later)

# Classes: destructors

- Destroys objects by performing the necessary cleanup (if needed!)
  - Dynamic memory, Files, Locks, …
- Signature: Tilde "~" + class name, no arguments and no return

```
~C() {
  // cleanup ...
}
```

- Called **automatically** when an object goes out of scope
- They are called in reverse order of construction (imagine like a stack of objects)
- The guaranteed destruction is ideal to handle resources (Resource Acquisition Is Initialization (RAII) idiom)

Somewhat a misnomer... It's more about destruction/cleanup

# Classes: copy constructor

- Creates a (new) object by copying an existing one
- Signature: Class name, (constant) reference to the class itself and no return

```
C(C const& c) {
  // create "this" object & copy c into "this" object
}
```

- Get's invoked on:

```
C c1(c);
C c1 = c; // that's not assignment, it's copy construction
C c1{c}; // uniform initialization form (C++11)
```

- Implicitly generated if no custom copy or move constructors/ assignment are provided
  - Member-wise: Does only so-called shallow copies of each member

# Classes: copy assignment

- Copies an existing object into another object
- Signature: (constant) reference to the class itself, returns a reference to itself

```
C& operator=(C const& c) {
  // copy c into "this" object  return *this;
}
```

- Get's invoked on:

```
C c1;          // ctor
C c2(c1);      // copy ctor
C c3 = c2;     // copy ctor
c1 = c3;       // copy assignment
c3 = c2 = c1; // chain of assignments
c3 = c3;       // self assignment!
```

- Implicitly generated if no custom copy or move constructors/ assignment are provided
  - Member-wise: Does only so-called shallow copies of each member

# Classes: move constructor & assignment (C++11)

- Move constructor
  - Creates an object by moving (stealing!) a temporary object into the new object

```
C(C&& c) {
  // create "this" object and move c into "this" object
  // leave c in a "moved-from state", e.g. "empty"
}
```

- Move assignment
  - Moves (stealing!) a temporary object into another object

```
C& operator=(C&& c) {
  // move c into "this" object
  // leave c in a "moved-from state", e.g. "empty"
}
```

- "&&" (pronounced ref-ref) denotes a so-called rvalue reference, which refers to a "temporary" object

# Classes: move constructor & assignment (C++11)

- How is the following evaluated?

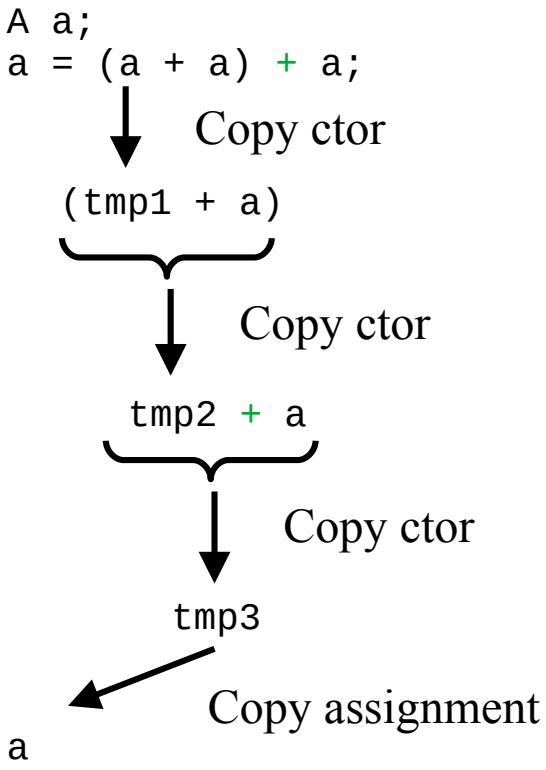
```
A a;
a = (a + a) + a;
```

```
class A {
  public:
    A() { std::cout << "A::ctor\n"; }
    ~A() { std::cout << "A::dtor\n"; }
    A(A const& a) {
      std::cout << "A::copy ctor\n";
    }
    A& operator=(A const& rhs) {
      std::cout << "A::copy assignment\n";
      return *this;
    }
};

A operator+(A a1, A const& a2) {
  std::cout << "A::operator+\n";
  return a1;
}
```

# Classes: move constructor & assignment (C++11)

- How is the following evaluated?



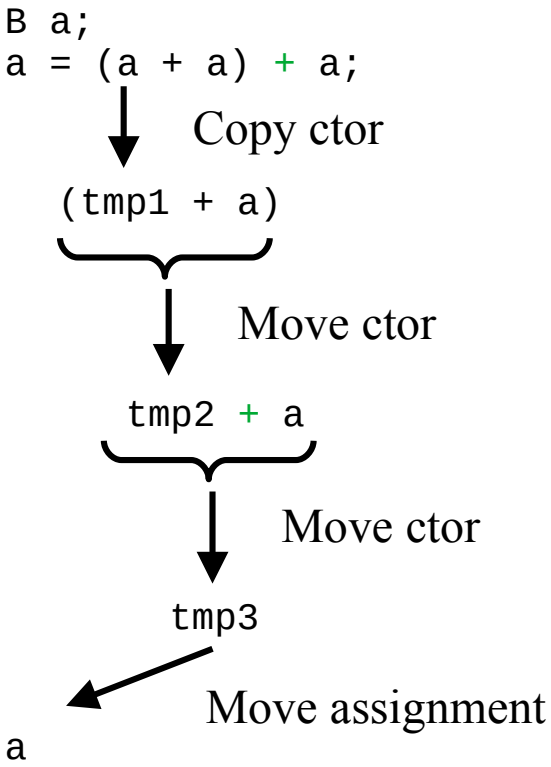

```
class A {
  public:
    A() { std::cout << "A::ctor\n"; }
    ~A() { std::cout << "A::dtor\n"; }
    A(A const& a) {
      std::cout << "A::copy ctor\n";
    }
    A& operator=(A const& rhs) {
      std::cout << "A::copy assignment\n";
      return *this;
    }
};

A operator+(A a1, A const& a2) {
  std::cout << "A::operator+\n";
  return a1;
}
```

Copying could be expensive, e.g., lots of data!

# Classes: move constructor & assignment (C++11)

- How is the following evaluated?




```
class B {
  public:
    B() { std::cout << "B::ctor\n"; }
    ~B() { std::cout << "B::dtor\n"; }
    B(B const& b) {
      std::cout << "B::copy ctor\n";
    }
    B& operator=(B const& rhs) {
      std::cout << "B::copy assignment\n";
      return *this;
    }
    B(B&& b) {
      std::cout << "B::move ctor\n";
    }
    B& operator=(B&& rhs) {
      std::cout << "B::move assignment\n";
      return *this;
    }
};

B operator+(B b1, B const& b2) {
  std::cout << "A::operator+\n";
  return b1;
}
```

# Value categories: It can be confusing…

- Rvalues, lvalues, glvalues, prvalues, xvalues, values, values, values, &, && ???
- For information on value categories see e.g.
  - https://en.cppreference.com/w/cpp/language/value_category
- A good explanation of rvalue references, see e.g.
  - http://thbecker.net/articles/rvalue_references/section_01.html

# Classes: Special member functions

- Summary

| Member function | Implicitly defined | Default definition |
|---|---|---|
| Default constructor | If no other constructors | Nothing (Default ctor memberwise) |
| Destructor | If no destructor | Nothing (Default ctor memberwise) |
| Copy constructor | If no move constructor and no move assignment | Copies memberwise |
| Copy assignment | If no move constructor and no move assignment | Copies memberwise |
| Move constructor | If no destructor, no copy constructor and no copy nor move assignment | Moves memberwise |
| Move assignment | If no destructor, no copy constructor and no copy nor move assignment | Moves memberwise |

- Keywords `default` and `delete` allow for explicit generation or non-generation (C++11)
- Deletion can also be forced by declaring the respective special member function private (C++98, C++03)

# Guidelines

- **Rule of three**
  - When you require a destructor or a copy constructor or a copy assignment operator, you most probably need to explicitly define all three
- **Rule of five** (C++11)
  - When you require a destructor or a copy constructor or a copy assignment operator or a move constructor or a move assignment operator, you most probably need to explicitly define all five
- **Rule of zero**
  - Classes that don't require explicit destructors, copy/move constructors and copy/move assignments operators are easier to handle

# Examples: Point

- Consider (again) the point class

```
class Point {
  public:
    using coord_t = double;
    Point(coord_t x = 0., coord_t y = 0.) : x_(x), y_(y) {} // ctor
    coord_t x() { return x_; }; // x coordinate
    coord_t y() { return y_; }  // y coordinate
    coord_t abs();              // distance from origin aka polar radius
    coord_t angle();            // polar angle
  private:
     coord_t x_, y_; // Cartesian coordinates
};
```

- What is?

```
Point p0;        //
Point p1(1.,2.); //
Point p2(p1);    //
Point p3;        //
p3 = p2;         //
```

- Implicitly generated copy ctor & copy assignment

# Examples: Point

- Consider (again) the point class

```
class Point {
  public:
    using coord_t = double;
    Point(coord_t x = 0., coord_t y = 0.) : x_(x), y_(y) {} // ctor
    coord_t x() { return x_; }; // x coordinate
    coord_t y() { return y_; }  // y coordinate
    coord_t abs();              // distance from origin aka polar radius
    coord_t angle();            // polar angle
  private:
     coord_t x_, y_; // Cartesian coordinates
};
```

- What is?

```
Point p0;         // default construction: p0 = (0,0)
Point p1(1.,2.); // p1 = (1,2)
Point p2(p1);     // copy construction: p2 = (1,2) (= p1)
Point p3;         // default construction: p3 = (0,0)
p3 = p2;          // copy assignment p3 = (1,2) (= p2)
```

- Implicitly generated copy ctor & copy assignment

# Examples: Point

- Overloading the compound assignment operators @=
  - Can only be implemented as member functions (see table!)
  - Should always return a reference to itself ("do as the ints")

```
a = b @= c;
```

- Plus assignment

```
Point& Point::operator+=(Point const& rhs) {
  x_ += rhs.x_;
  y_ += rhs.y_;
  return *this;
}
```

- Binary arithmetic operators a@b
  - Best implemented as non-member (i.e., free) functions

```
Point operator+(Point const& p1, Point const& p2) {
  Point result(p1.x() + p2.x(), p1.y() + p2.y());
  return result;
}
```

# Examples: Point

- Overloading the compound assignment operators @=
  - Can only be implemented as member functions (see table!)
  - Should always return a reference to itself ("do as the ints")

```
a = b @= c;
```

- Plus assignment

```
Point& Point::operator+=(Point const& rhs) {
  x_ += rhs.x_;
  y_ += rhs.y_;
  return *this;
}
```

- Binary arithmetic operators a@b
  - Best implemented as non-member (i.e., free) functions

```
Point operator+(Point p1, Point const& p2) {
  return p1 += p2;
}
```

- Best implemented in terms of their compound assignments!

# Argument Dependent Lookup (ADL)

- Consider in `demos/week05/Point/ADL`

```
namespace Geometry {
class Point { /* as before... */ };
}
int main() {
  Geometry::Point p1(1., 2.);
  std::cout << "p1 = " << p1 << '\n';
  Geometry::func(p1);
  func(p1);
}
```

- Looks in "associated classes and namespaces": namespaces of the arguments
- Applies only to unqualified calls
  - `abs(x)`, but not `std::abs(x)`
- Also known as Koenig lookup (after Andrew Koenig)
- See documentation for the (ugly?) details
  - https://en.cppreference.com/w/cpp/language/adl

# Example: Simple array

- Let's build our own simple array class

```
class SArray {
  public:
    using sz_t   = unsigned int; // size type
    using elem_t = double;       // element type
    // ctors
    SArray() : size_(0), elem_(nullptr) {}
  private:
    sz_t size_;    // size of array
    elem_t* elem_; // pointer to (dynamic) memory of array
};
```

- Public array size type and element type
- Private size and pointer to array elements
- Default ctor

# Example: Simple array

- Let's build our own simple array class

```
class SArray {
  public:
    using sz_t   = unsigned int; // size type
    using elem_t = double;       // element type
    // ctors
    SArray() : size_(0), elem_(nullptr) {}
    SArray(sz_t size) : size_(size), elem_(new elem_t[size]) {}
  private:
    sz_t size_;    // size of array
    elem_t* elem_; // pointer to (dynamic) memory of array
};
```

- Add ctor for given size array allocating the memory
- What next?

# Example: Simple array

- Let's build our own simple array class

```
class SArray {
  public:
    using sz_t   = unsigned int; // size type
    using elem_t = double;       // element type
    // ctors
    SArray() : size_(0), elem_(nullptr) {}
    SArray(sz_t size) : size_(size), elem_(new elem_t[size]) {}
    // dtor
    ~SArray() { delete[] elem_;  elem_ = nullptr;  size_ = 0; }
  private:
    sz_t size_;    // size of array
    elem_t* elem_; // pointer to (dynamic) memory of array
};
```

- Add ctor for given size array allocating the memory
- Dtor for cleanup!

# Example: Simple array

- Let's build our own simple array class

```
class SArray {
  public:
    using sz_t   = unsigned int; // size type
    using elem_t = double;       // element type
    // ctors
    SArray() : size_(0), elem_(nullptr) {}
    SArray(sz_t size) : size_(size), elem_(new elem_t[size]) {}
    // dtor
    ~SArray() { delete[] elem_;  elem_ = nullptr;  size_ = 0; }
  private:
    sz_t size_;    // size of array
    elem_t* elem_; // pointer to (dynamic) memory of array
};
```

- So let's try it:

```
SArray a;      // empty array OK
SArray b(10);  // array of size 10 OK
SArray c = 10; // array of size 10 PROBABLY NOT OK
```

# Example: Simple array

- Let's build our own simple array class

```
class SArray {
  public:
    using sz_t   = unsigned int; // size type
    using elem_t = double;       // element type
    // ctors
    SArray() : size_(0), elem_(nullptr) {}
    explicit SArray(sz_t size) : size_(size), elem_(new elem_t[size]) {}
    // dtor
    ~SArray() { delete[] elem_;  elem_ = nullptr;  size_ = 0; }
  private:
    sz_t size_;    // size of array
    elem_t* elem_; // pointer to (dynamic) memory of array
};
```

- So let's try it:

```
SArray a;      // empty array OK
SArray b(10);  // array of size 10 OK
SArray c = 10; // array of size 10 PROBABLY NOT OK
```



See https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c46-by-default-declare-single-argument-constructors-explicit

# Example: Simple array

- Let's build our own simple array class

```
class SArray {
  public:
    using sz_t   = unsigned int; // size type
    using elem_t = double;       // element type
    // ctors
    SArray() : size_(0), elem_(nullptr) {}
    explicit SArray(sz_t size) : size_(size), elem_(new elem_t[size]) {}
    // dtor
    ~SArray() { delete[] elem_;  elem_ = nullptr;  size_ = 0; }
  private:
    sz_t size_;    // size of array
    elem_t* elem_; // pointer to (dynamic) memory of array
};
```

- Copy ctor & copy assignment?

```
SArray a;
SArray b(a);
SArray c;
c = b;
```

# Example: Simple array

- Let’s build our own simple array class

```
class SArray {
  public:
    // ... as before ...
    // copy ctor
   SArray(SArray const& a) : size_(a.size_), elem_(new elem_t[a.size_]) {
     // copy elements
     for (sz_t i = 0; i < size_; ++i) {
       elem_[i] = a.elem_[i];
     }
   }
   // copy assignment
   SArray& operator=(SArray const& rhs) {
     delete[] elem_;
     size_ = rhs.size_;
     elem_  = new elem_t[size_];
     for (sz_t i = 0; i < size_; ++i) {
       elem_[i] = rhs.elem_[i];
     }
     return *this;
   }
};
```

- Now?

# Example: Simple array

- Let's build our own simple array class

```
class SArray {
  public:
    // ... as before ...
    // copy ctor
   SArray(SArray const& a) : size_(a.size_), elem_(new elem_t[a.size_]) {
     // copy elements
     for (sz_t i = 0; i < size_; ++i) {
       elem_[i] = a.elem_[i];
     }
   }
   // copy assignment
   SArray& operator=(SArray const& rhs) {
     delete[] elem_;
     size_ = rhs.size_;
     elem_  = new elem_t[size_];
     for (sz_t i = 0; i < size_; ++i) {
       elem_[i] = rhs.elem_[i];
     }
     return *this;
   }
};
```

- Now? What if self-assignment? What if memory allocation fails?

# Example: Simple array

- Solution: Copy & swap technique!
- Introducing new member function swap

```
class SArray {
  public:
    // ... as before ...
    void swap(SArray& other) {
      using std::swap; // let C++ determine which swap to call!
      swap(size_, other.size_);
      swap(elem_, other.elem_);}
  }
};
```

See https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c165-use-using-for-customization-points

- Copy assignment reduces to

```
class SArray {
  public:
    // ... as before ...
    SArray& operator=(SArray const& rhs) {
      SArray tmp(rhs); // copy
      swap(tmp); // swap
      return *this;
  }
};
```



See https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c62-make-copy-assignment-safe-for-self-assignment

# Example: Simple array

- Compound arithmetic operators (have to be member functions)

```
class SArray {
  public:
    // ... as before ...
    // compound arithmetic operators
    SArray& operator+=(SArray const& rhs) {
      assert(size_ == rhs.size_); // check if compatible sizes!
      for (sz_t i = 0; i < size_; ++i) {
        elem_[i] += rhs.elem_[i];
      }
      return *this;
    }
};
```

- Binary arithmetic operators (non-member)

```
SArray operator+(SArray a, SArray const& b) {
  return a += b;
}
```

- Implement the others…

# Example: Simple array

- We also want to access individual elements with the subscript operator
- Typically needs two variants: const and non-const (why?)
- Overload over const-ness!

```
class SArray {
  public:
    // ... as before ...
    // subscript operator
    SArray::elem_t& operator[](sz_t index){
      assert(index < size());
      return elem_[index];
    }
    // const subscript operator
    SArray::elem_t const& operator[](sz_t index) const {
      assert(index < size());
      return elem_[index];
    }
};
```

# Class templates

- Allow classes to have members that use template parameters as types
- Syntax:

```
template <typename T>
class Point {
  public:
    using coord_t = T;
    Point(coord_t x = 0., coord_t y = 0.) : x_(x), y_(y) {} // ctor
    coord_t x() { return x_; }; // x coordinate
    coord_t y() { return y_; }  // y coordinate
    coord_t abs();              // distance from origin aka polar radius
    coord_t angle();            // polar angle
  private:
     coord_t x_, y_; // Cartesian coordinates
};
// ...
Point<float> pf;
Point<double> pd;
```

# Class templates

- Allow classes to have members that use template parameters as types
- Syntax:

```
template <typename T>
class Point {
  public:
    using coord_t = T;
    // ...
    coord_t abs();              // distance from origin aka polar radius
    // ...
};

// out of class definition of abs
template <typename T>
T Point<T>::abs() {
  return std::sqrt(x_*x_ + y_*y_);
}
```

# Class templates

- Allow classes to have members that use template parameters as types
- Syntax:

```
template <typename T>
class Point {
  public:
    using coord_t = T;
    // ...
    coord_t abs();              // distance from origin aka polar radius
    // ...
};

// out of class definition of abs
template <typename T>
typename Point<T>::coord_t Point<T>::abs() {
  return std::sqrt(x_*x_ + y_*y_);
}
```

# Class templates

- Class templates
  - Type parameters: `typename` or `class` (most common case)
  - Non-type parameters (values, mostly integers)
  - Template-template parameters (passing a template as a template parameter)
- Class templates specialization
  - Concrete/special implementation for specific template parameters

```
template <typename T> class SArrayT { ... };
template <> class SArrayT <char> { ... };
```

Note that all members have to be specialized (even those identical to the generic template class!), because there is no "inheritance" of members from the generic template

  - Partial specialization possible (remember: for function templates only full specialization!)

# Example: Simple array class template

- See `demos/week05/SArrayT`

```
template <typename T>
class SArray {
  public:
   using size_type = std::size_t;    // size type
    using value_type = T;             // value/element type
    using reference = T&;             // reference type
    using const_reference = T const&; // const reference type
  // ...
};
```

- Add type aliases similar to STL conventions
  - E.g. https://en.cppreference.com/w/cpp/container/vector