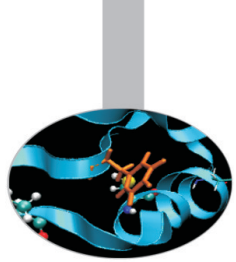
CINECA SCAI
SuperComputing Applications and Innovation

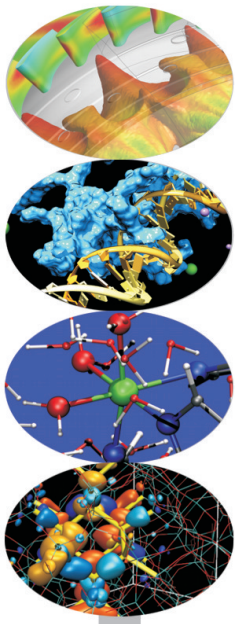


Scientific and Technical Computing in C++

Part 3 Class Templates & STL

Luca Ferraro Mario Tacconi

CINECA Roma - SCAI Department



Rome, November 2016

CINECA

Outline

Object Oriented Design

- OO languages allow for definition of new, very rich, powerful, domain specific types and operations
- Particularly C++, with all its features

- This is not as easy as it may seem from the first examples one meets
- Object Oriented programming is all about design

- A design choice in one class may have annoying consequences elsewhere
- Fundamental OOP principle:

> *The sins of the fathers are to be laid upon the children*
> *W. Shakespeare, "The Merchant of Venice", act. III, sc. V*

Positions in Space

- We want a type that represents positions in a 3D space
- And we want all arithmetic operations that make sense:
  - unary + and −
  - binary + and −
  - multiplication times a scalar and division by a scalar
  - dot product
  - and combined operator-assignment like +=

Design and Implementation Choices

- The three components of a position are well defined: let's have them public
- Let's use double precision
- Let's define two constructors:
  1. a default constructor, doing nothing
  2. a constructor that initializes the three components
- For arithmetic operators, let's use a traditional approach:
  1. define unary ones as methods (they are so simple!)
  2. define combined operator-assignment as methods (they modify the object!)
  3. define all other binary operators in terms of the previous ones
- And let's make them **`inline`**, as they are small and used very often
  - It's automatic on methods defined inside the class

position.h: Fundamentals


```
#ifndef POSITION_H
#define POSITION_H

struct position {
    double x, y, z;

    position() {}
    position(double a, double b, double c) : x(a), y(b), z(c) {}

    position operator+() const { return *this; }
    position operator-() const { return position(-x, -y, -z); }

    position& operator+= (position r) { x += r.x; y += r.y; z += r.z; return *this; }
    position& operator-= (position r) { x -= r.x; y -= r.y; z -= r.z; return *this; }
    position& operator*= (double s) { x *= s; y *= s; z *= s; return *this; }
    position& operator/= (double s) { x /= s; y /= s; z /= s; return *this; }

};

#endif
```

Arithmetic Operators Alternatives

- Implemented as **`position`** client functions
  - \+ read and write access are separated
  - – were data members private, dot product would not work
  - – looking at the class definition is not enough
- Implemented as **`position`** member functions
  - \+ all supported operations listed in class definition
  - \+ all operations have complete access and work
  - – left operand passed by reference!
- Implemented as **`position`** friend functions
  - \+ all supported operations listed in class definition
  - \+ all operations have complete access and work
  - \+ both operands passed by value
- Remember:
  - member functions are inherited on derivation
  - client and friend functions are not

position.h: Operators as Clients


```
struct position {
    double x, y, z;

    position() {}
    position(double a, double b, double c) : x(a), y(b), z(c) {}

    position operator+() const { return *this; }
    position operator-() const { return position(-x, -y, -z); }

    position& operator+= (position r) { x += r.x; y += r.y; z += r.z; return *this; }
    position& operator-= (position r) { x -= r.x; y -= r.y; z -= r.z; return *this; }
    position& operator*= (double s) { x *= s; y *= s; z *= s; return *this; }
    position& operator/= (double s) { x /= s; y /= s; z /= s; return *this; }

};

inline position operator+ (position r1, position r2) {
    r1 += r2;
    return r1;
}

inline double operator* (position r1, position r2) {
    return r1.x*r2.x + r1.y*r2.y + r1.z*r2.z;
}
```

`position.h`: Operators as Methods


```
struct position {
    double x, y, z;

    position() {}
    position(double a, double b, double c) : x(a), y(b), z(c) {}

    position operator+() const { return *this; }
    position operator-() const { return position(-x, -y, -z); }

    position& operator+= (position r) { x += r.x; y += r.y; z += r.z; return *this; }
    position& operator-= (position r) { x -= r.x; y -= r.y; z -= r.z; return *this; }
    position& operator*= (double s) { x *= s; y *= s; z *= s; return *this; }
    position& operator/= (double s) { x /= s; y /= s; z /= s; return *this; }

    position operator+ (position r) const;
    double operator* (position r) const;
};

inline position position::operator+ (position r) const {
    position temp(*this);
    temp += r;
    return temp;
}

inline double position::operator* (position r) const {
    return x*r.x + y*r.y + z*r.z;
}
```

`position.h`: Operators as Friends


```
struct position {
    double x, y, z;

    position() {}
    position(double a, double b, double c) : x(a), y(b), z(c) {}

    position operator+() const { return *this; }
    position operator-() const { return position(-x, -y, -z); }

    position& operator+= (position r) { x += r.x; y += r.y; z += r.z; return *this; }
    position& operator-= (position r) { x -= r.x; y -= r.y; z -= r.z; return *this; }
    position& operator*= (double s) { x *= s; y *= s; z *= s; return *this; }
    position& operator/= (double s) { x /= s; y /= s; z /= s; return *this; }

    friend position operator+ (position r1, position r2);
    friend double operator* (position r1, position r2);
};

inline position operator+ (position r1, position r2) {
    r1 += r2;
    return r1;
}

inline double operator* (position r1, position r2) {
    return r1.x*r2.x + r1.y*r2.y + r1.z*r2.z;
}
```

Hands-on Session #1

- Write the operator functions for:
  - binary **`position`** subtraction
  - multiplication times a scalar and division by a scalar
  - equality and inequality
- Write a program that exercises the **`position`** class:
  - testing all methods and operators
  - verifying operator associativity and precedence in complicated expressions
  - verifying illegal expressions are rejected by the compiler

Operator Arguments

- Operator arguments ar almost always defined as:
  - references, when content is changed by the operator
  - `const` references, when it isn't
- This is not a matter of style
  - Indispensable, in the first case
  - Wise, in the second
- Expression evaluation involves intermediate values
  - Which have to be properly constructed and destructed
  - I.e. suitable constructors and destructors are called
- Here, we dispensed with `const` references
  - For the sake of simplicity
  - Because constructors are so simple that the compiler can easily optimize the code
- This is not true for more complicated types: use `const` references in real codes!

Consistency

- OO Programming amounts to extending the base language into a *domain specific language*
- Consistency is key to facilitate use of new data types and code reuse
- Consistency must be ensured on two sides:
  1. the application domain
  2. the base language
- Let's analyze the issue by adding cross product and modulus to `position` class

Methods as Messages

- Let's add in the class definition two methods:

```
position cross(position r) const {
    return position(y*r.z - z*r.y,
                    z*r.x - x*r.z,
                    x*r.y - y*r.x);
}

double abs() const {
    return sqrt(*this**this);
}
```

- So that:
  - **`r1.cross(r2)`** returns the cross product
  - and **`r1.abs()`** returns the modulus
- This style is typical of programmers overexposed to other OO languages, like Smalltalk, that think of methods as messages sent to an object
- Unfortunately:
  - it's inconsistent with the application domain
  - it's inconsistent with C++ style of arithmetic
  - may be confusing and ambiguous to users

Functions Can be Our Friends

- Let's add to **`position.h`** the two functions:

```
inline position cross(position r1, position r2) {
    return position(r1.y*r2.z - r1.z*r2.y,
                    r1.z*r2.x - r1.x*r2.z,
                    r1.x*r2.y - r1.y*r2.x);
}

inline double abs(position r) {
    return sqrt(r*r);
}
```

  and make the former (or possibly both) a friend of **`position`** class

- This is:
  - + quite consistent with the application domain
  - + definitely consistent with C++ style of arithmetic
  - + and if both are friends, you'll spot them all at a glance in the class definition

A Tempting Alternative

- Let's add in **`position`** class:

```
position& operator%= (position r) {
    x = y*r.z - z*r.y,
    y = z*r.x - x*r.z,
    z = x*r.y - y*r.x);
    return *this;
}

friend position operator% (position r1, position r2);
```

and to **`position.h`** the function:

```
inline position operator% (position r1, position r2) {
    r1 %= r2;
    return r1;
}
```

- This is:
  - \+ very consistent with the application domain, even if the operator symbol differs
  - +/– slightly inconsistent with C++ style of arithmetic, as the `%` operator is the integer modulo

Hands-on Session #2

- In a 3D space, velocities have the same components and arithmetics of positions
- Let's try the following:

```
#include "position.h"

struct velocity : public position {
    velocity()
    velocity(double a, double b, double c) : position(a,b,c)
};
```

- Of course, friend functions will not be inherited
- Nonetheless:
  - try to exercise the **`velocity`** class
  - verify that a **`velocity`** value cannot be assigned to a **`position`** object or viceversa
  - check which **`position`** methods can be succesfully used with **`velocity`** values and objects

The Code Reuse Problem

Inheriting **`velocity`** from **`position`** class has some drawbacks:

- **`velocity`** values can be assigned to **`position`** variables
- Base class methods that return an object of the base class can't be immediately used on the derived class
- As we said, we got a bonus conversion from **`velocity`** to its base class, and this cannot be hidden
- Adding two velocities to get a position is not physically sound

Using a Common Base Clase

- The obvious solution: have both **`position`** and **`velocity`** inherit from a common base class **`vect`** and delegate actual computations to it
- **`position`** and **`velocity`** will be two separate branches of the inheritance tree, so no conversion between them will be possible
- To avoid using **`vect`** as the base class and convert it on its derived:
  - declare protected all its methods, so that no operations can be performed outside its derived classes
  - declare protected its constructors, so that no **`object`** can be created outside its derived classes

vect.h: Part 1 of 3


```
#ifndef VECT_H
#define VECT_H

#include <cmath>

struct vect {
    double x, y, z;
protected:
    vect() {}
    vect(double a, double b, double c) : x(a), y(b), z(c) {}

    vect operator+() const { return *this; }
    vect operator-() const { return vect(-x, -y, -z); }

    vect& operator+= (vect v) { x += v.x; y += v.y; z += v.z; return *this; }
    vect& operator-= (vect v) { x -= v.x; y -= v.y; z -= v.z; return *this; }
    vect& operator*= (double s) { x *= s; y *= s; z *= s; return *this; }
    vect& operator/= (double s) { x /= s; y /= s; z /= s; return *this; }

    vect operator+ (vect v) const;
    vect operator- (vect v) const;
    double operator* (vect v) const;
    vect operator* (double s) const;
    friend vect operator* (double s, vect v);
    vect operator/ (double s) const;
    bool operator== (vect v) const;
    bool operator!= (vect v) const;

    friend double abs(vect v);
};
```

vect.h: Part 2 of 3


```
inline vect vect::operator+ (vect v) const {
    vect temp(*this);
    temp += v;
    return temp;
}

inline vect vect::operator- (vect v) const {
    vect temp(*this);
    temp -= v;
    return temp;
}

inline double vect::operator* (vect v) const {
    return x*v.x + y*v.y + z*v.z;
}

inline vect vect::operator* (double s) const {
    vect temp(*this);
    temp *= s;
    return temp;
}

inline vect operator* (double s, vect v) {
    v *= s;
    return v;
}

inline vect vect::operator/ (double s) const {
    vect temp(*this);
    temp /= s;
    return temp;
}
```

vect.h: Part 3 of 3


```
inline bool vect::operator== (vect v) const {
    return x==v.x && y==v.y && z==v.z;
}

inline bool vect::operator!= (vect v) const {
    return !(*this==v);
}

inline double abs(vect v) {
    return sqrt(v*v);
}

#endif
```

position: Deriving from vect


```
struct position : public vect {
protected:
    position(vect r) : vect(r) {}
public:
    position() {}
    position(double a, double b, double c) : vect(a,b,c) {}

    position operator+() const { return *this; }
    position operator-() const { return this->vect::operator-(); }

    position& operator+= (position r) { this->vect::operator+=(r); return *this; }
    position& operator-= (position r) { this->vect::operator-=(r); return *this; }
    position& operator*= (double s) { this->vect::operator*=(s); return *this; }
    position& operator/= (double s) { this->vect::operator/=(s); return *this; }

    position operator+ (position r) const { return this->vect::operator+(r); }
    position operator- (position r) const { return this->vect::operator-(r); }
    double operator* (position r) const { return this->vect::operator*(r); }
    position operator* (double s) const { return this->vect::operator*(s); }
    friend position operator* (double s, position r);
    position operator/ (double s) const { return this->vect::operator/(s); }
    bool operator== (position r) const { return this->vect::operator==(r); }
    bool operator!= (position r) const { return this->vect::operator!=(r); }
    friend double abs(position r);
};

inline position operator* (double s, position r) {
    r *= s;
    return r;
}

inline double abs(position r) {
```

velocity: Deriving from vect


```
struct velocity : public vect {
protected:
    velocity(vect r) : vect(r) {}
public:
    velocity() {}
    velocity(double a, double b, double c) : vect(a,b,c) {}

    velocity operator+() const { return *this; }
    velocity operator-() const { return this->vect::operator-(); }

    velocity& operator+= (velocity v) { this->vect::operator+=(v); return *this; }
    velocity& operator-= (velocity v) { this->vect::operator-=(v); return *this; }
    velocity& operator*= (double s) { this->vect::operator*=(s); return *this; }
    velocity& operator/= (double s) { this->vect::operator/=(s); return *this; }

    velocity operator+ (velocity v) const { return this->vect::operator+(v); }
    velocity operator- (velocity v) const { return this->vect::operator-(v); }
    double operator* (velocity v) const { return this->vect::operator*(v); }
    velocity operator* (double s) const { return this->vect::operator*(s); }
    friend velocity operator* (double s, velocity v);
    velocity operator/ (double s) const { return this->vect::operator/(s); }
    bool operator== (velocity v) const { return this->vect::operator==(v); }
    bool operator!= (velocity v) const { return this->vect::operator!=(v); }
    friend double abs(velocity v);
};

inline velocity operator* (double s, velocity v) {
    v *= s;
    return v;
}

inline double abs(velocity v) {
```

The Power of Syntax Checking

- We could derive from `vect` more classes:
  - acceleration
  - momentum
  - force
  - angular velocity
  - angular momentum
  - torque
  - electric field
  - magnetic field
- Compiler will perform syntax checking, and ensure no misuses are made
- But we can do more... let's define a time class

A Class to Represent Physical Time


```
class time {
    double val;
public :
    time() {}
    time(double t) : val(t) {}
    time operator+() const { return *this; }
    time operator-() const { return time(-val); }

//  <op>= member operators

//  member arithmetic operators (sum, difference,
//                               multiplication and division by a scalar)
//  all comparison member operators

// friend << and >> operators for iostreams

    friend position operator* (velocity v, time t);
    friend position operator* (time t, velocity v);
    friend velocity operator/ (position r, time t);
};

position operator* (velocity v, time t) {
    return v*t.val;
}

position operator* (time t, velocity v) {
    return v*t.val;
}

velocity operator/ (position r, time t) {
    return r/t.val;
}
```

Dimensionally Consistent Arithmetic

- Let's also enroll the three mixed type operators in **`position`** and **`velocity`** friends

- Note: depending on include file structure this will involve forward declarations like:

```
class time;
struct velocity;
```

  or other pairs from the three classes, to be added in suitable places

- And we'll get a physical quantities arithmetic whose dimensional consistency is checked by the compiler
  - Have a look at Units or MPL in the Boost library (for braves only)

Consistent Extension and Code Reuse

- Better yet, we could:
  - define a base class for a generic physical scalar implementing the basic arithmetic and I/O
  - protecting all its members, including constructors, like we did with **`vect`**
  - derive time, mass, charge, pressure, temperature, ... classes from it, like we did with **`position`** and **`velocity`** from **`vect`**
  - define suitable mixed-type operations according to the rule of physics
- And we'd get all the benefits of code reuse and syntax checking

Outline

Class Templates

- Like functions, classes can be parameterized
- Class templates generate multiple specific classes from a single definition
- Templates are a form of overloading
  - And can be themselves overloaded
- Templates can be combined together
  - A template class can inherit from another template class
  - A template class method can be a template in itself
- Templates are one of the most powerful features of C++
  - They are the foundation of generic programming
  - They can be an alternative to inheritance

`float` vs. `double`

- We based our vectors on **`doubles`**, but we could need a **`float`** version to use less memory
- We could duplicate the code for **`floats`**, but this is bad for code management
- Let's make **`vect`** and its descendants parametric in the coordinate type
- And let's make it safely, and consistently with C++ rules for arithmetic
- We'll have to proceed in steps

vect.h: Template Version Part 1 of 3


```
#ifndef VECT_H
#define VECT_H

#include <cmath>

template<class T> struct vect {
    T x, y, z;
protected:
    vect() {}
    vect(T a, T b, T c) : x(a), y(b), z(c) {}

    vect operator+() const { return *this; }
    vect operator-() const { return vect(-x, -y, -z); }

    vect& operator+= (vect v) { x += v.x; y += v.y; z += v.z; return *this; }
    vect& operator-= (vect v) { x -= v.x; y -= v.y; z -= v.z; return *this; }
    vect& operator*= (T s) { x *= s; y *= s; z *= s; return *this; }
    vect& operator/= (T s) { x /= s; y /= s; z /= s; return *this; }

    vect operator+ (vect v) const;
    vect operator- (vect v) const;
    T operator* (vect v) const;
    vect operator* (T s) const;
    template<class F> friend vect<F> operator* (F s, vect<F> v);
    vect operator/ (T s) const;
    bool operator== (vect v) const;
    bool operator!= (vect v) const;

    template<class F> friend F abs(vect<F> v);
};
```

vect.h: Template Version Part 2 of 3


```
template<class T> inline vect<T> vect<T>::operator+ (vect<T> v) const {
    vect<T> temp(*this);
    temp += v;
    return temp;
}

template<class T> inline vect<T> vect<T>::operator- (vect<T> v) const {
    vect<T> temp(*this);
    temp -= v;
    return temp;
}

template<class T> inline T vect<T>::operator* (vect<T> v) const {
    return x*v.x + y*v.y + z*v.z;
}

template<class T> inline vect<T> vect<T>::operator* (T s) const {
    vect<T> temp(*this);
    temp *= s;
    return temp;
}

template<class F> inline vect<F> operator* (F s, vect<F> v) {
    v *= s;
    return v;
}
```

vect.h: Template Version Part 3 of 3


```
template<class T> inline vect<T> vect<T>::operator/ (T s) const {
    vect<T> temp(*this);
    temp /= s;
    return temp;
}

template<class T> inline bool vect<T>::operator== (vect<T> v) const {
    return x==v.x && y==v.y && z==v.z;
}

template<class T> inline bool vect<T>::operator!= (vect<T> v) const {
    return !(*this==v);
}

template<class F> inline F abs(vect<F> v) {
    return std::sqrt(v*v);
}

#endif
```

`vect` Template: Remarks

- `class T` is a template type parameter
  - It doesn't need to be a class
  - It can be any type
- Method declarations do not need to be in template form, it's automatic
- Method definitions *inside* the class definition do not need to be in template form, it's automatic
- Method definitions *outside* the class definition must be in template form
- Friend function templates must be declared in template form inside the class

- Of course, we have to 'templatize' `position` and `velocity` too
- And we'll make `double` the default, using a default template argument

position: Template Version


```
template<class T=double> struct position : public vect<T> {
protected:
    position(vect<T> r) : vect<T>(r) {}
public:
    position() {}
    position(double a, double b, double c) : vect<T>(a,b,c) {}

    position operator+() const { return *this; }
    position operator-() const { return this->vect<T>::operator-(); }

    position& operator+= (position v) { this->vect<T>::operator+=(v); return *this; }
    position& operator-= (position v) { this->vect<T>::operator-=(v); return *this; }
    position& operator*= (T s) { this->vect<T>::operator*=(s); return *this; }
    position& operator/= (T s) { this->vect<T>::operator/=(s); return *this; }

    position operator+ (position v) const { return this->vect<T>::operator+(v); }
    position operator- (position v) const { return this->vect<T>::operator-(v); }
    T operator* (position v) const { return this->vect<T>::operator*(v); }
    position operator* (T s) const { return this->vect<T>::operator*(s); }
    template<class F> friend position<F> operator* (F s, position<F> v);
    position operator/ (T s) const { return this->vect<T>::operator/(s); }
    bool operator== (position v) const { return this->vect<T>::operator==(v); }
    bool operator!= (position v) const { return this->vect<T>::operator!=(v); }
    template<class F> friend F abs(position<F> v);
};

template<class F> inline position<F> operator* (F s, position<F> v) {
    v *= s;
    return v;
}

template<class F> inline F abs(position<F> v) {
```

velocity: Template Version


```
template<class T=double> struct velocity : public vect<T> {
protected:
    velocity(vect<T> r) : vect<T>(r) {}
public:
    velocity() {}
    velocity(double a, double b, double c) : vect<T>(a,b,c) {}

    velocity operator+() const { return *this; }
    velocity operator-() const { return this->vect<T>::operator-(); }

    velocity& operator+= (velocity v) { this->vect<T>::operator+=(v); return *this; }
    velocity& operator-= (velocity v) { this->vect<T>::operator-=(v); return *this; }
    velocity& operator*= (T s) { this->vect<T>::operator*=(s); return *this; }
    velocity& operator/= (T s) { this->vect<T>::operator/=(s); return *this; }

    velocity operator+ (velocity v) const { return this->vect<T>::operator+(v); }
    velocity operator- (velocity v) const { return this->vect<T>::operator-(v); }
    T operator* (velocity v) const { return this->vect<T>::operator*(v); }
    velocity operator* (T s) const { return this->vect<T>::operator*(s); }
    template<class F> friend velocity<F> operator* (F s, velocity<F> v);
    velocity operator/ (T s) const { return this->vect<T>::operator/(s); }
    bool operator== (velocity v) const { return this->vect<T>::operator==(v); }
    bool operator!= (velocity v) const { return this->vect<T>::operator!=(v); }
    template<class F> friend F abs(velocity<F> v);
};

template<class F> inline velocity<F> operator* (F s, velocity<F> v) {
    v *= s;
    return v;
}

template<class F> inline F abs(velocity<F> v) {
    return abs(vect<F>(v));
```

Hands-on Session #4

- Write a program to exercise the **`position`** class template
- Then try to mix in the same expression:
  - **`float`**s with **`position<double>`**s
  - **`double`**s with **`position<float>`**s
  - **`position<float>`**s with **`position<double>`**s
- Verify that **`position<>`** is equivalent to **`position<double>`**
- And try combinations like **`position<unsigned int>`** or **`position<char>`**

The Perils of Class Templates

- Now we can represent coordinates in single and double precision
- And, if we are not careful enough, we could represent **`unsigned`** and **`char`** coordinates
- Or, for that matter, we could happen to represent coordinates with **`velocity`**s, or whatever abstract type that supports some sort of exotic arithmetic
  - very subtle bugs are possible
  - particularly if a template is not explicitly instantiated, and members are instantiated on demand
- To make our classes more robust, let's allow only **`position<float>`** and **`position<double>`** objects
- To this purpose, we need to exploit:
  - one more feature: template specialization
  - a common C++ idiom: traits classes

Template Specialization

- It's a form of overloading
- A specialized template is a specific implementation for specific values of template arguments
- Very useful to complete the generic template with specific ones optimized for particular cases
- Specialized template and functions prevail on less specialized ones in overload resolution
- More on this later, let's use it now to avoid **`position`** template abuses

- Brute force solution:
  - defining only two specialized templates, **`vect<float>`** and **`vect<double>`**
  - but this is code replication, bad for source code management
- Clever solution: traits classes

vect.h: using Traits


```
#ifndef VECT_H
#define VECT_H

#include <cmath>

template<class T> struct vectComponentTrait {};

template<> struct vectComponentTrait<double> {
    typedef double component;
};

template<> struct vectComponentTrait<float> {
    typedef float component;
};

template<class T> struct vect {
    typename vectComponentTrait<T>::component x;
    T y, z;
protected:
//...
```

...everything else unchanged

`vect` Component Traits

- **`vectComponentTrait<>`** template defines a suitable type for a **`vect`** component
  - But it does so only for argument types we allow
  - And doesn't define anything in the general case
- **`vectComponentTrait<>::component`** is used to define **`vect<>`** first member
  - **`typename`** tells the compiler that it is a type indeed
- Thus, an instantiation of the **`vect<>`** template will:
  - succeed, if the argument type is **`float`** or **`double`**
  - fail at compile time otherwise
- Please, notice:
  - no code duplication
  - no other changes to **`vect<>`** or its descendants
  - to add support for one more type, just add one more specialization of **`vectComponentTrait<>`**

More on Traits Classes

- Traits classes are widely used in Standard C++ Library and in many other libraries
- They usually don't have data members, only:
  - type members
  - static methods
- Traits classes static methods are useful to abstract a unified interface from a bunch of heterogeneous classes
- Traits classes with more than one type member help to manage mixed precision computations
- Do you remember the issues we had with `gcd()` and `lcm()` on mixed precision types?
  - To avoid troubles, we had to hide the template
  - And use it explicitly in (too) many wrapper functions
- Now we know more, let's write an elegant solution

numbertheory.h 3.0 - Part 1 of 2


```
#ifndef NUMBERTHEORY_H
#define NUMBERTHEORY_H

template<class T, class F> struct numth_traits {};

template<> struct numth_traits<int, int> {
    typedef int narrow_t;
    typedef int wide_t;
};

template<> struct numth_traits<int, long> {
    typedef int narrow_t;
    typedef long wide_t;
};

template<> struct numth_traits<long, int> {
    typedef int narrow_t;
    typedef long wide_t;
};

template<> struct numth_traits<long, long> {
    typedef long narrow_t;
    typedef long wide_t;
};
```

numbertheory.h 3.0 - Part 2 of 2


```
// Greatest Common Divisor
template <class T, class F>
typename numth_traits<T,F>::narrow_t gcd(T aa, F bb) {

  typename numth_traits<T,F>::wide_t a = abs(aa);
  typename numth_traits<T,F>::wide_t b = abs(bb);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    typename numth_traits<T,F>::wide_t t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}


// Least Common Multiple
template <class T, class F>
typename numth_traits<T,F>::wide_t lcm(T a, F b) {
  if (a == 0 || b == 0)
    return 0;
  return a*(b/gcd(a,b));
}

#endif
```

Mixed Precision Arithmetic and Consistency

- Excluding mixed precision computations is very annoying
- We have three options

1. Defining implicit conversions and rely on them
   - Inconsistent with C++ rules!
   - Arithmetic operations on a **`float`** and a **`double`** shall be performed in **`double`**
   - Automatic down-conversion shall only happen on assignment
2. Do as in C++ does with **`complex<>`** values
   - Mixed precision only allowed on `=`, `+=`, `-=`, `*=`, `/=`
   - Explicit conversions needed otherwise
3. Implement mixed precision arithmetic for all binary operators

- Let's go for option #2

vect: Mixed Precision Arithmetic


```
template<class T> struct vect {
    typename vectComponentTrait<T>::component x;
    T y, z;
protected:
    vect() {}
    vect(T a, T b, T c) : x(a), y(b), z(c) {}
    template<class F> vect(const vect<F> &v) : x(v.x), y(v.y), z(v.z) {}

    vect operator+() const { return *this; }
    vect operator-() const { return vect(-x, -y, -z); }

    template<class F>
        vect& operator= (vect<F> v) { x = v.x; y = v.y; z = v.z; return *this; }

    template <class F>
        vect& operator+= (vect<F> v) { x += v.x; y += v.y; z += v.z; return *this; }
    template <class F>
        vect& operator-= (vect<F> v) { x -= v.x; y -= v.y; z -= v.z; return *this; }
    template <class F>
        vect& operator*= (F s) { x *= s; y *= s; z *= s; return *this; }
    template <class F>
        vect& operator/= (F s) { x /= s; y /= s; z /= s; return *this; }

    vect operator+ (vect v) const;
    vect operator- (vect v) const;
    T operator* (vect v) const;
    vect operator* (T s) const;
    template<class F> friend vect<F> operator* (F s, vect<F> v);
    vect operator/ (T s) const;
    bool operator== (vect v) const;
    bool operator!= (vect v) const;

    template<class F> friend F abs(vect<F> v);
```

Remarks

- A template copy constructor has been added
- Thus, instantiating **`vect<double>`** would automatically generate, as needed:

```
vect<double> vect<double>(const vect<float> &v);
vect<double> vect<double>(const vect<double> &v);
```

- And instantiating **`vect<float>`** would automatically generate, as needed:

```
vect<float> vect<float>(const vect<float> &v);
vect<float> vect<float>(const vect<double> &v);
```

- Ditto for **`=`**, **`+=`**, **`-=`**, **`*=`**, **`/=`**
- Now we need to change **`position`** and **`velocity`** definitions accordingly
- And make their copy constructor **`explicit`** to avoid unintended ‘automagic’ behavior

position: Mixed Precision Arithmetic


```
template<class T=double> struct position : public vect<T> {
protected:
    position(vect<T> r) : vect<T>(r) {}
public:
    position() {}
    position(double a, double b, double c) : vect<T>(a,b,c) {}
    template<class F> explicit position(const position<F> &v) : vect<T>(v) {}

    position operator+() const { return *this; }
    position operator-() const { return this->vect<T>::operator-(); }

    template<class F> position& operator= (position<F> v)
                          { this->vect<T>::operator=(v); return *this; }

    template <class F> position& operator+= (position<F> v)
                          { this->vect<T>::operator+=(v); return *this; }
    template <class F> position& operator-= (position<F> v)
                          { this->vect<T>::operator-=(v); return *this; }
    template <class F>
        position& operator*= (F s) { this->vect<T>::operator*=(s); return *this; }
    template <class F>
        position& operator/= (F s) { this->vect<T>::operator/=(s); return *this; }

    position operator+ (position v) const { return this->vect<T>::operator+(v); }
    position operator- (position v) const { return this->vect<T>::operator-(v); }
    T operator* (position v) const { return this->vect<T>::operator*(v); }
    position operator* (T s) const { return this->vect<T>::operator*(s); }
    template<class F> friend position<F> operator* (F s, position<F> v);
    position operator/ (T s) const { return this->vect<T>::operator/(s); }
    bool operator== (position v) const { return this->vect<T>::operator==(v); }
    bool operator!= (position v) const { return this->vect<T>::operator!=(v); }
    template<class F> friend F abs(position<F> v);
};
```

`velocity`: Mixed Precision Arithmetic


```
template<class T=double> struct velocity : public vect<T> {
protected:
    velocity(vect<T> r) : vect<T>(r) {}
public:
    velocity() {}
    velocity(double a, double b, double c) : vect<T>(a,b,c) {}
    template<class F> explicit velocity(const velocity<F> &v) : vect<T>(v) {}

    velocity operator+() const { return *this; }
    velocity operator-() const { return this->vect<T>::operator-(); }

    template<class F> velocity& operator= (velocity<F> v)
                          { this->vect<T>::operator=(v); return *this; }

    template <class F> velocity& operator+= (velocity<F> v)
                          { this->vect<T>::operator+=(v); return *this; }
    template <class F> velocity& operator-= (velocity<F> v)
                          { this->vect<T>::operator-=(v); return *this; }
    template <class F>
        velocity& operator*= (F s) { this->vect<T>::operator*=(s); return *this; }
    template <class F>
        velocity& operator/= (F s) { this->vect<T>::operator/=(s); return *this; }

    velocity operator+ (velocity v) const { return this->vect<T>::operator+(v); }
    velocity operator- (velocity v) const { return this->vect<T>::operator-(v); }
    T operator* (velocity v) const { return this->vect<T>::operator*(v); }
    velocity operator* (T s) const { return this->vect<T>::operator*(s); }
    template<class F> friend velocity<F> operator* (F s, velocity<F> v);
    velocity operator/ (T s) const { return this->vect<T>::operator/(s); }
    bool operator== (velocity v) const { return this->vect<T>::operator==(v); }
    bool operator!= (velocity v) const { return this->vect<T>::operator!=(v); }
    template<class F> friend F abs(velocity<F> v);
};
```

Enough for Now

- Implementing full mixed precision arithmetic is the next logical step
- Quite similar to `gcd()` and `lcm()` mixed precision arguments issue
- Unsurprisingly, traits classes come to rescue

- Homework assignment:
  - test that the traits based templates we implemented work as intended
  - test that the simplified mixed precision arithmetic version we implemented works as intended
  - implement full mixed precision aritmetic using traits classe as we did in `gcd()` and `lcm()`

Template Parameters


```
template<int N, class T> class vertex : public position<T> {
protected:
    position<T> *edge[N];
public:
    vertex() : position<T>() {
        for(int i=0; i<N; ++i)
          edge[i] = NULL;
    }

    int edgesno() { return N; }

    // more methods, friends...
}
```

- Templates are not restricted to a single parameter
- And template parameters do not need to be types
- They can also be constant expressions of **`int`** type, to:
  - size internal data structures
  - bound a constant into methods of an object at creation
  - ...
- Or constant expressions of pointer or reference type
- Actually, non-type template parameters turn templates into a powerful, Turing complete, declarative language

Non-Type Parameters and Specialization

- Non-type parameters too support overloading and specialization
- Let's imagine we want to write a generic code to do plasma physics simulations in 2D and 3D spaces
- In two dimensions, we only have the *x* and *y* coordinates
- We could:
  - add an `int D` parameter to `vect` and its descendants
  - write two specialized versions with 2 or 3 components respectively
  - adapt all operators and functions accordingly (the cross product is now a scalar!)
  - and create objects of `position<2,double>` or `position<3,float>` type
- But now for something completely different...

polynomial.h: Part 1 of 2


```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#include <cstring>
#include <stdexcept>

template<class T=double> class polynomial {
    int order;
    T *coeff;

    T horner(T x) const;
public:
    polynomial() : order(-1), coeff(NULL) { }
    polynomial(int n) : order(n), coeff(new T[n+1]) {
        for (int i = 0; i<n+1; ++i)
            coeff[i] = 0.0;
    }
    polynomial(int n, const T c[]) : order(n), coeff(new T[n+1]) {
        memcpy(coeff, c, (n+1)*sizeof(T));
    }

    ~polynomial() { delete[] coeff; }

    polynomial(const polynomial& p);
    polynomial& operator= (const polynomial& p);

    int degree() const { return order; }
    T& operator[] (int i) { return coeff[i]; }

    T operator() (T x) const { return horner(x); }
};
```

polynomial.h: Part 2 of 2


```
// deep copy constructor

template<class T> polynomial<T>::polynomial(const polynomial<T>& p)
                                                : order(p.order), coeff(NULL) {
    if (order >= 0) {
        coeff = new T[order+1];
        memcpy(coeff, p.coeff, (order+1)*sizeof(T));
    }
}

// deep assignment

template<class T> polynomial<T>& polynomial<T>::operator= (const polynomial<T>& p) {
    if (this == &p) return *this;
    order = p.order;
    delete[] coeff; coeff = NULL;
    if (order >= 0) {
        coeff = new T[order+1];
        memcpy(coeff, p.coeff, (order+1)*sizeof(T));
    }
    return *this;
}

// polynomial evaluation with horner algorithm

template<class T> T polynomial<T>::horner(T x) const {
    if (!coeff) throw std::domain_error("uninitialized polynomial");
    T p = coeff[order];
    for (int i=order-1; i>=0; --i)
        p = p*x + coeff[i];
    return p;
}
```

`polynomial`: Miscellaneous Remarks

- All constructors put the pointer to coefficients storage in a consistent state
- Copy constructor and assignment must be explicitly defined because of deep copy
- `delete` on a null pointer does not cause errors
- The `[]` operator is overloaded to allow accessing coefficients like it were an array
- Yes, all operators can be overloaded
  - arithmetic (`+`, `-`, `*`, `/`, `%`, `~`, `|`, `&`, `^`, `<<`, `>>`)
  - increment and decrement (`++`, `--`)
  - assignment (`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`)
  - comparison (`==`, `!=`, `<`, `>`, `<=`, `>=`)
  - logical (`!`, `||`, `&&`, `^^`)
  - address of, dereferencing, and access (`&`, `*`, `->`, `[]`)
  - function call (`()`) and more

Let's Get Real

- We'll never evaluate a 10 thousands degree polynomial with this class
- Most polynomials in use are of low degree
- And high degree ones require special numeric care
- Or have so many zero coefficients to make this approach inefficient

- Moreover, when iteration count is known at compile time, compilers generate better code for loops

- Let's add an integer parameter to the template, to fix the degree at compile time
- We'll loose some runtime flexibility, but the class will become very simple

polynomials of Fixed Degree


```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#include <cstring>

template<int N, class T=double> class polynomial {
    T coeff[N+1];

    T horner(T x) const {
        T p = coeff[N];
        for (int i=N-1; i>=0; --i)
            p = p*x + coeff[i];
        return p;
    }

public:
    polynomial() {
        for (int i = 0; i<N+1; ++i)
            coeff[i] = 0.0;
    }

    polynomial(const T c[]) {
        memcpy(coeff, c, (N+1)*sizeof(T));
    }

    T& operator[] (int i) { return coeff[i]; }

    T operator() (T x) const { return horner(x); }
};

#endif
```

Remarks and New Ideas

- Coefficients are now stored inside the object: no need for deep copies
- No dynamic memory allocation:
  - default destructor is OK
  - no need to throw exceptions
- Too simple? Wait...

- Most polynomial approximations are of *very* low degree
- And we could suspect that the function call and the loop cost more than the calculations
- Should we build specialized template of the class for low degrees to make them more efficient?
  - NO! duplicating code is bad
  - Let's use *template metaprogramming*

Template Metaprogramming

- We said templates are a declarative language
- In declarative languages, loops are performed by recursion
  - A final step is defined, for the first or the last iteration
  - All other iterations are defined in terms of the next or previous one

- The basic idea is:
  1. Define a function template with an integer parameter
  2. Having it perform one iteration and invoke itself recursively incrementing or decrementing the parameter
  3. Making a specialized version for the ending interations
  4. Make all these templates expand inline

- Unfortunately, we need a special kind of template specialization not allowed in function templates
- We'll have to use a helper class template

polynomial.h: Expanding the Loop


```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#include <cstring>

template<int N, class T=double> class polynomial {
    T coeff[N+1];

    // U is the polinomial type, M is max degree, C is current iteration
    template<class U, int M, int C> struct horner {  // private template struct
        static U eval(const U *c, U x) {             // with a static method
            return horner<U,M,C-1>::eval(c, x)*x + c[M-C];  // using recursion
        }
    };

    template<class U, int M> struct horner<U,M,0> { // partial specialization
        static U eval(const U *c, U x) { return c[M]; }
    };

public:
    polynomial() {
        for (int i = 0; i<N+1; ++i)
            coeff[i] = 0.0;
    }
    polynomial(const T c[]) {
        memcpy(coeff, c, (N+1)*sizeof(T));
    }

    T& operator[] (int i) { return coeff[i]; }

    T operator() (T x) const { return horner<T,N,N>::eval(coeff, x); }
};
```

A Few Details

- The helper class templates are private members of the class to avoid namespace pollution
- `coeff` must be passed in because its address is unknown at compile time and `eval()` is a static method
- As methods are defined inside a class, they will be expanded inline
- A template specializing only part of its parameters has a special syntax
- And is termed a *partial specialization*

Hands-on Session #5

- Write a program to exercise the **`polynomial`** class template
- Test that all functionalities work as desired
- Test that template metaprogramming works
  - By compiling with **`-c`** option
  - And looking at symbols in the object file (using **`nm`** command)
  - Then compiling with **`-c -O2`** or **`-c -O3`**
  - And looking again at symbols in the object file
- Beware: if you don't invoke the functor, no template will be instantiated
  - And no symbols of interest will show up in the object
- Then test what happens if out of bounds indexes are passed to the `[]` operator

Want to Know More?

- Template metaprogramming is incredibly powerful
  - Likewise, you can define if/then/else, **`switch`**-like structures, ...
  - Gives way much more flexibility than the macro preprocessor
  - It's abundantly used in libraries (notably, the STL)
  - You may encounter it in very complex codes
- Template metaprogramming is at least as difficult as powerful
  - It stresses compilers and programmers
  - C++ standard and bibles are silent or very obscure
  - Search for info and help on the web

Outline

Standard Template Library

- Research effort started in 1979, to explore generic implementations abstracting widely used data organizations and manipulations
- Developed in different languages, in the same decade C++ was developed
- Until both efforts joined in 1993
- STL revolutionized OO programming
- It sports:
  - *container* data structures hosting other objects
  - *iterators* to access their contents
  - generic *algorithms* operating on data in a container
  - *functors* to be applied by the latter

Containers

- A panoply of options:
  - `vector<>`
  - `list<>`
  - `deque<>`
  - `set<>`
  - `map<>`
  - ...
  - `#include <container>` and start using it
- STL containers shine at managing objects whose number changes dynamically
- Far easier to use and flexible than C++ built-in arrays:
  - manage memory for you
  - keep track of how many objects they hold
  - and more...
- May appear slower than C++ built-in arrays
  - But they are not, if wisely used
  - Their methods implement code you should anyway write

Containers Memory Organization

- *contiguous-memory* containers like **`vector<>`**, **`string<>`**, **`deque<>`**:
  - store all *N* elements consecutively in one or more chunks of memory
  - on insertion/deletion of an element, only elements in the same chunk have to be shifted
  - thus sequential access is very fast ($O(1)$), while insertions and deletions can be slow ($O(N)$)
- *node-based* containers like **`list<>`** or **`map<>`**:
  - store each element in an independent chunk of memory
  - on insertion/deletion of an element, only pointers in neighboring ones are affected
  - they trade sequential access speed ($O(N)$) for fast insertions and deletions ($O(1)$)

Choosing the Proper Container

- *Do you need to insert or delete elements in arbitrary positions?*
  - Containers such as lists, queues or deques will perform better
- *Do you only add elements at the end?*
  - A vector could be ok
- *Do you need constant time, fast access?*
  - A vector, no doubts
- *Has internal data to be layout-compatible with C?*
  - Only vectors will do
- *Is it crucial that on insertion or deletion other elements do not move in memory?*
  - Vectors and contiguous-memory containers will not do

Collecting Atoms

- We want to collect in a container representations of atoms in a box we are simulating
  - An `Atom` class is used to represent each atom
- We want direct access to any atom, using an index
- Number of atoms in our box is constant during the simulation
- But only known after reading a complex input file

- Our (quite obvious) choice: a `vector<Atom>`

A Sketch to Read Atoms


```
// prepare an empty vector container for atoms
vector<Atom> system;


// read atomic positions and properties
// and store them in system vector
while (!in.eof() || in.good()) {
    Atom current_atom();
    in >> current_ atom;
    system.push_back(current_atom);
}

// print out a report for DEBUGGING
cerr << "System has " << system.size() << "atoms" << endl;
```

Automatic Incremental Growth

- STL containers automatically grow to accommodate additional items
- `push_back()` method inserts an element at the end
- This operation might involve a preliminary resize, i.e.:
  1. *allocation* of a new, bigger memory block
  2. *copy* of all elements from old to new block
  3. *destruction* of objects in old block
  4. *deallocation* of old memory block
- Beware!
  - Steps 1 to 3 can be very expensive
  - Any pointer/reference to a container's element might be invalidated!

`reserve()` in Advance


```
// prepare an empty vector container for atoms
vector<Atom> system;
system.reserve(educated_guess);

// read atomic positions and properties
// and store them in system vector
while (!in.eof() || in.good()) {
    Atom current_atom();
    in >> current_ atom;
    system.push_back(current_atom);
}

// print out a report for DEBUGGING
cerr << "System has " << system.size() << "atoms" << endl;
```

Reserving Space in Advance

- `reserve(size_t n)` method reserves space in advance for `n` items of element type
- If we reserve enough space:
  - no reallocation will be needed on `push_back()` calls
  - a lot of memory reallocations and data copies will be spared
  - a lot of constructor and destructor calls will be spared too
- Related methods:
  - `capacity()` returns how many elements fit in memory presently allocated by the container
  - `size()` tells how many elements are in the container
  - `empty()` ... guess it

An Experiment


```
vector<Atom> system;
cerr << "Size: " << system.size() << "Capacity: " << system.capacity() << endl;

system.reserve(n_atoms);        // run, then comment this out and run again

while (!in.eof() || in.good()) {
    Atom current_atom();
    in >> current_ atom;
    atom.id = system.size();    // vectors indexes are zero-based

    // beware: the following horribly slows down the code
    cerr << "Size: " << system.size() << "Capacity: " << system.capacity() << endl;
}
cerr << "Size: " << system.size() << "Capacity: " << system.capacity() << endl;
```

Trimming Memory Usage

- Many STL implementations will *double* a **`vector<>`** memory on automatic resize
  - No problem going from 100KB to 200KB
  - An issue if going from 1GB to 2GB
- You can free unused space with:
  `vector<Atom>(system).swap(system);`
  - **`vector<Atom>(system)`** creates an unamed temporary copy of **`system`**
  - Copy constructor allocates just enough memory for existing elements
  - **`swap(system)`** method call swaps memory blocks between the two
  - At `;` the temporary is destructed and memory freed
- Beware: next **`push_back()`** will result in a resize

Hands-on Session #1

- `pop_back()` method removes last element from a vector

- Take the bingo class inheriting from `rng`
- Use a vector for the list of numbers to draw
- Use another vector for the list of already drawn numbers
- Dispense with `m` field and rewrite the class relying on vectors

- What do you think of the new code?

Inter-Atomic Interactions

- $atom_i$ interacts with $atom_j$ if their distance is less than R
- Atomic positions evolve in time
  - Some atoms will depart, some will come closer
  - Interacting pairs will change
- For each atom, at each simulation time step, we need to list all atoms it interacts with
- This is a costly process
  - Comparing distances for each pair has $O(N^2)$ complexity
  - R might depend on interacting atomic species and force field
  - Unbearable for medium to large systems
  - Newton's Third Law can halve it, but is not enough

Linked-Cells

- A widely used technique
- Simulation domain is split into regular cells
- Interaction lists for each atom are computed considering:
  1. atoms belonging to the same cell
  2. atoms belonging to 26 (in 3D) neighboring cells
- Using lists of atoms located in each cell, complexity reduces to $O(N)$
- Still, atom moves, diffusing from cell to cell
- We need suitable data structures to represent:
  - atoms in a cell
  - the cell itself
  - the collection of cells composing our domain
  - interaction lists

Our Choices

- The cell itself
  - A very simple class
- Collection of cells
  - Direct access to neighbouring cells is needed
  - Let's use vectors
- Atoms in a cell
  - We'll scan them sequentially for interactions
  - At each time step, some of them will change cell
  - Let's use a list
- Interaction lists
  - Rebuilt from scratch at each time step
  - Lists or vectors?
  - Let's opt for vectors, more cache friendly and quick to access

Linked-Cells


```
vector< vector<Atom *> > interactions(system.size());

struct cell {
  unsigned n_atoms;
  list<Atom *> atom_list; // atoms belonging to current cell

  cell() : n_atoms(0) {}

  unsigned size() const { return n_atoms; }
};

//  compute cell size and n_x, n_y, n_z from R and box size
vector< vector< vector<cell> > > linkedCells(n_z); // resize() at construction

// linkedCells build up
for (int k=0; k < n_z; ++k) {
    linkedCells[k].resize(n_y);
    for (int j=0; j < n_y; ++j) {
        linkedCells[k][j].resize(n_x)
        for (int i=0; i < n_x; ++i)
            linkedCells[k][j][i] = cell();   // invokes member constructors
    }
}
```

Containers are Composable

- We got:
  - a `vector<>`
  - of `vector<>`s
  - of `vector<>`s
  - of `structs`
- Beware!
  - This looks like a built-in array:
    `cell linkedCells[n_z][n_y][n_z]`
  - But is very different!
  - Each `vector<>` in a `vector< vector <> >` may have different size
  - If you forget, you might insert bugs in your code

Copies or Pointers?

- Containers store copies of the template parameter type
- Easy answer: use less memory
  - Use pointers to `system` elements for atoms in cells and interaction lists
  - Use values for container of cells
- Object-smart answer: copies of objects might be costly
  - Copy constructor or copy assignment must be called
- Inheritance-smart answer: copy leads to ‘slicing’
  - If you put a derived class object in a container for its base class and try to copy it back
  - Unless you make copy assignments and constructors virtual, which adds to costs
  - Pointers are safer in this respect
- Beware: destruction of pointer containers does not destruct pointees
  - Which is what we need, with atoms

Lists

- Lists are better for frequent insertions and deletions
  - Each element is an independent chunk of memory
  - No $O(N)$ resize costs, insertion/deletion is $O(1)$
  - Pointers to unaffected elements still valid afterwards
- They provide many insertion and deletion methods
  - **`push_front()`** / **`push_back()`** insert a new item at beginning/end of list ($O(1)$)
  - **`pop_front()`** / **`pop_back()`** delete an item at beginning/end of list ($O(1)$)
  - **`insert(`*`pos`*`)`** / **`erase(`*`pos`*`)`** insert/delete the item at position *pos* of list ($O(N)$!)
  - **`remove(`*`val`*`)`** deletes all items with value ***`val`***
- Beware of **`size()`** in node-based containers!
  - It's ($O(N)$)
  - That's why we cache size in **`n_atoms`** member of **`cell`**
  - Always use $O(1)$ **`my_list.empty()`** instead of **`(my_list.size() == 0)`**

Putting Atoms in Cells


```
const double invDimCell_x = n_x/BoxSide_x;
const double invDimCell_y = n_y/BoxSide_y;
const double invDimCell_z = n_z/BoxSide_z;

for (int idx=0; idx<system.size(); idx++) {
    Atom& atom = system[idx];

    int i = invDimCell_x * atom.pos.x;
    int j = invDimCell_y * atom.pos.y;
    int k = invDimCell_z * atom.pos.z;

    linkedCells[i][j][k].atom_list.push_back(&atom);
    linkedCells[i][j][k].n_atoms++;
}
```

`lists` Aren't `vectors`

- Lists don't provide **`capacity()`** method
  - They don't need it!
- Ditto for **`reserve()`** method
- Lists provide special member functions for moving elements
  - Generally faster since they only change pointers
- Lists don't support subscript operator **`[i]`** nor **`at(i)`**
  - To avoid performance noxious abuses
- So, how to run through a list or list portion?

Iterators

- Elements of a list are accessed through *iterators*
- A generic technique
  - Usable with any container kind (also for vectors)
  - They mimic pointers
  - In fact, pointers are good iterators for contiguous-memory containers
  - Trickier ones are needed for node-based containers
- Basic syntax
  - **`begin()`** method returns iterator 'pointing' to first container element
  - **`end()`** method returns iterator 'pointing' right 'after' last container element
  - use **`end()`** for comparisons only
  - `j++` / `--j` advances/steps back the iterator
  - `*j` returns the element it 'points' to
- **`#include <iterator>`** for more iterator flavors

Building Interaction Lists


```
for(int i=0; i < system.size(); i++) {
    Atom& atom_i = system[i];

    int iC = invDimCell_x * atom_i.pos.x;
    int jC = invDimCell_y * atom_i.pos.y;
    int kC = invDimCell_z * atom_i.pos.z;
    cell& here = linkedCells[kC][jC][iC];

    interactions[i].clear() // destroy all element from the vector, capacity not affec
    // cell side slightly larger than maxInteractionRadius
    interactions[i].reserve(ceil(here.n_atoms*acos(-1)/6.0)*8);

    // define an iterator to explore the interaction lists
    list<Atom *>::iterator j;

    for (j = here.atoms.begin(); j != here.atoms.end(); j++)
        if ( *j != &atom_i && myShortRangeField.interact(atom_i, *j) )
            interactions[i].push_back(*j);

    // loops on neighboring cells atoms ...
}
```

More STL Containers

- We already met **`valarray<>`**s
- **`deque<>`**
  - Double ended queue
  - Similar to **`vector<>`**, can easily add/remove elements on both ends
- **`map<>`** and **`multimap<>`**
  - Associative containers good for *(key,value)* pairs
  - Keep elements sorted according to some criterion
- **`set<>`** and **`multiset<>`**
  - Associative containers mimicking logical sets
  - Elements search has $O(log(N))$ complexity
- And more...

`valarray<>` vs. `vector<>`

- Do not mistake one for the other
- Both are composable
  - Different elements may be containers of different size
- `valarray<>` supports elementwise arithmetic
  - `vector<>` *does not*
- `vector<>` supports automatic resize
  - `valarray<>` *does not*
- Both support manual resize using `resize()` method
  - But a `valarray<>` loses its contents!
  - While a `vector<>` *does not*

The Illusion of Container-Independent Code

- Sequence containers provide **`push_front()`** and/or **`push_back()`**
  - Associative containers *do not*
- Contiguous-memory containers offer random-access iterators and subscripting
  - Node-based containers *do not*
- Many methods are defined for one category of containers only
- Even basics as insert or erase have different signatures and semantics
- And apparently identical methods have wildly different performances
- Different containers *are different*:
  - they have strengths and weaknesses
  - and were not designed to be interchangeable

Some Container-Independent Code

- However, if:
  - you only use methods supported by all containers
  - and do not modify the container content
- Then you can write some container-independent template function, like this:

```
template<class T> void putto(ostream& s, const T& v) {
    if (v.empty())
        s << "Empty container!";
    else {
        typename T::const_iterator i;
        for(i=v.begin(); i != v.end(); ++i)
            s << *i << ' ';
    }
    s << endl;
}
```

- But you'll have to use the **`typename`** keyword to instance container-specific iterators or nested types

STL Algorithms

- Containers by themselves aren't that much appealing
- Real STL power lies in generic algorithms that serve most fundamental programmer's needs
  - *traversal, sorting, searching, inserting, removing, etc*
- Commonalities:
  - implemented as template functions
  - operating through iterators
  - element types inferred from iterator types
- Able to operate on containers portions
  - Beginning iterator 'points' to first element to operate upon
  - End iterator 'points' right 'after' the last one
- To exploit them **`#include`**:
  - **`algorithm`** for general ones
  - **`numeric`** for the few numerically specialized ones
  - **`functional`** for function objects (a.k.a. functors)

Initializing Containers

- A common task consist in assigning values to the container's elements
  - Elements can be set to a constant value using **`fill()`**
  - Or using a more specialized functor passed to **`generate()`**
- Both relay on **`operator=`** of container's elements
  - A natural task for POD, mind for user defined types
  - Container's elements must be already initialized
- Another common task is to print the values of elements to stdout
  - A standard **`for()`** loop can do the job
  - or you can combine **`copy`** with **`ostream_iterator`**

Assignment and Print


```
#include <iostream>
#include <vector>
#include <limits>
#include <cmath>
#include <algorithm>
#include <iterator>

vector<double> pi(100);
fill(pi.begin(), pi.end(), acos(-1.0) );

template<class T>
class RandClass {
    T maxv;
  public:
    explicit RandClass(const T &maxvalue = numeric_limits<T>::max() ) : maxv(maxvalue)
    T operator() (void) const { return (maxv/(RAND_MAX + 1.0))*rand(); }
};

vector<int> v(90);
const int max_value = 90;
generate(v.begin(), v.end(), RandClass(max_value) );


copy(pi.begin(), pi.end(), ostream_iterator<double>(cout," ")); cout « endl;
copy(v.begin(), v.end(), ostream_iterator<int>(cout," ")); cout « endl;
```

Sort Algorithm

- Sort is among most known and frequently used algorithms
  - Requires random-access iterators (works best for vectors)
  - For lists, don't use the algorithm, use `sort()` method
- `sort()` reorders container elements:
  - using comparison operators for the element type
  - or an optional comparison function argument
  - or an optional *compare* functor object
- `sort()` has $N \log(N)$ complexity on average
  - But its worst case is $O(N^2)$
- `stable_sort()` variant is $2N \log(N)$
- Partial sorts also available

Sorting


```
vector<int> v(90);
  ... // initialize v with different numbers
sort(v.begin(), v.end()) // after sort, v is modified with its elements sorted


int builtin[90];
  ... // initialize builtin with different numbers
sort(&builtin[0], &builtin[90])   // good also for built-in arrays


vector<Atom> momenta(system); // Note: costly copy for illustration purposes only

class compareAtomMomenta {  // order by decreasing momentum
  public:
    bool operator() (const Atom &a, const Atom &b) const
      { return a.mass*abs(a.vel) > b.mass*abs(b.vel); }
};

sort(momenta.begin(), momenta.end(), compareAtomMomenta());
```

Search Algorithms

- Searches among elements are also very common
- `find()` returns first occurrence of an element matching the search
  - Match performed using `(==)` operator
  - Returns container `end()` if no match
- `find_if()` accepts a predicate functor to specify complex matching criteria
  - A predicate must return a `bool`

Searching


```
int match = 90; // la paura!
vector<int>::iterator matchIterator =
  find(bingoExtractions.begin(), bingoExtractions.end(), match);
```

```
class less_than_4_neighbors {
  public:
    bool operator() (const vector<Atom *> &v) const
      { return v.size() < 4;}
}

vector<vector<Atom *> >::iterator firstUnder4 =
  find_if(interactions.begin(), interactions.end(), less_than_4_neighbors());
```

```
class atomIsCarbon {
  public:
    bool operator() (const Atom &a) const
      { return a.symbol == "C";}
}

vector<Atom>::iterator firstCarbonAtom =
  find_if(system.begin(), system.end(), atomIsCarbon());
```

Specializing and Extending Predicates

- `less_than_4_neighbors` is ugly code
  - And we'd probably need to compare against a different number of neighbours
- `atomIsCarbon` is not generic
  - And we'd like to pass the species to be searched for as argument
- STL provides some helpers
  - Functional template predicates: *equal_to, greater, greater_equal, less, less_equal, etc*
  - Binders and template predicates: *bind2nd(y), bind1st(x), unary_function, binary_function*
- Can be combined to extend predicates and define new operations
- To access them, `#include <functional>`

Specializing Predicates


```
vector<int> v(90);
  generate(v.begin(), v.end(), Rand<int>(maxv) );
  sort(v.begin(), v.end()); // after sort, v is modified with its elements sorted
  copy(v.begin(), v.end(), ostream_iterator<double>(cout," ")); cout « endl;

  1 1 3 5 5 6 7 9 11 12 12 14 14 17 17 19 20 21 21 23 24 25 25 26 26 29 30
  31 31 31 31 32 33 35 36 36 39 39 41 42 44 46 46 46 47 47 47 47 48 49 54
  56 57 57 57 59 60 61 61 64 64 66 68 69 69 69 70 71 72 72 74 75 75 76 79
  80 80 80 81 82 82 82 83 83 85 85 87 87 89

  counts = count_if(v1.begin(), v1.end(), bind2nd(equal_to<int>(), 5) );
  // result counts = 2

  // 10 <= x
  counts = count_if(v.begin(), v.end(), bind1st(less_equal<int>(), 10) );
  // result counts = 82

  // x <= 10
  counts = count_if(v.begin(), v.end(), bind2nd(less_equal<int>(), 10) );
  // result counts = 8

  vector<double> v2;
  ... // build a signal in v2
  double clamp = 0.5;
  replace_if(v1.begin(), v1.end, clamp, bind1st( less_equal<double>(), clamp));
```

Extending Predicates with User Defined Types

- Specializing predicates on PODs is easy
- We need more control when dealing with user-defined types
- STL provides common base classes to help users build their own predicates

```
template <class Arg, class Res> struct unary_function {
  typedef Arg argument_type;
  typedef Res result_type;
};

template <class Arg, class Arg2, class Res> struct binary_function {
  typedef Arg  first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Res result_type;
};
```

Extending Predicates to User Defined Types


```
class less_neighbors_than : public unary_function< vector<Atom*>, bool> {
    int arg2;
  public:
    explicit less_neighbors_than (const int &x) : arg2(x) { }
    bool operator() (const vector<Atom *> &v) const
      { return v.size() < arg2;}
}

vector<vector<Atom *> >::iterator firstUnder4 =
  find_if(c.begin(), c.end(), less_neighbors_than(4));


class atomSpecieIs : public unary_function<Atom,bool> {
    string specie;
  public:
    explicit atomSpecieIs (const string &x) : specie(x) { }
    bool operator() (const Atom &a) const
      { return a.chemsymbol == specie;}
}

vector<Atom>::iterator firstCarbonAtom =
  find_if(system.begin(), system.end(), atomSpecieIs("C"));
```

Member Function and Pointer Function Adapters

- Most algorithms invoke built-in or user-defined operators
- We may want to invoke a method or any other function on each element in a sequence
- STL provides function adapters
  - **`mem_fun()`** call a method on each element pointer
  - **`mem_fun_ref()`** call a method on each element reference
  - **`ptr_fun()`** takes a pointer to a non-member function
- A set of algorithms map function objects or adapters on sequences
  - **`for_each()`** applies an adapter on each element
  - same for **`transform()`**, that can modify elements or generate a sequence of results
  - arithmetic function objects *plus, minus, multiplies, etc...*
- Again, **`#include <functional>`**

Applying Operations to Elements


```
// call method with no argument
for_each(system.begin(), system.end(), mem_fun_ref(&Atom::reset) );


// call method with argument timestep
for_each(system.begin(), system.end(),
          bind2nd(mem_fun_ref(&Atom::evolve), timestep) );

// compute the center of mass of the system
class centerOfMass {
    position c;
    double tot_m;
  public:
    centerOfMass() : c(0.0,0.0,0.0), tot_m(0.0) { }        // initialize
    void operator () (Atom &a) { c += a.mass * a.pos;
                                 tot_m += a.mass; }        // accumulate
    position result() const { return c/tot_m; }            // return sum
}

centerOfMass CoM;
for_each(system.begin(), system.end(), CoM);
cout << "Center of mass is " << CoM.result() << endl;
```

Composing and Transforming Elements


```
// transform A based on operand, result in B
transform(A.begin(), A.end(), B.begin(), bind1st(plus(), 100) );


// combine A and B with operand, result in C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus() );


// C = A + B with insertion (each result is pushed_back)
transform(A.begin(), A.end(), B.begin(), back_inserter(C.begin()), plus() );


// runs through a container of pointers deleting pointees
template<class T> struct Delete_ptr {
    T* operator() (T *p) {
      delete p; return NULL;
    };
}
void purgePointerContainer (vector<myClass *> &somePointerContainer) {
    transform(somePointerContainer.begin(), somePointerContainer.end(),
        somePointerContainer.begin(), Delete_ptr<myClass>());
}
```

`remove()` vs. `erase()`

- Want to remove all elements with value *val*?
- For lists, the best way is using **`remove()`** method

```
mylist.remove(88);
```

- For associative containers, use **`erase()`** method

```
associative.erase(88);
```

- For contiguous-memory containers, you must combine **`erase()`** methods with **`remove()`** algorithm

```
v.erase( remove(v.begin(), v.end(), 88) , v.end());
```

- **`remove()`** algorithm doesn't really removes elements
  - Shifts back (by copy!) following elements
  - And returns an iterator 'pointing' to new *logical end*
  - This behavior makes it compatible with built-in arrays
  - Erase will do the rest

Predicated Removal

- Want to remove all elements satisfying **`predicate`**?

```
template<class T>
bool predicate(T x); // returns wheter x is "bad" for us
```

- For lists, the best way is using **`remove_if()`** method

```
mylist.remove_if(predicate);
```

- For contiguous-memory containers, you must combine **`erase()`** method with **`remove_if()`** algorithm

```
v.erase(v.remove_if(v.begin(), v.end(), predicate), v.end());
```

- Again, **`remove_if()`** doesn't really remove elements
  - Simply shifts back (by copy!) following elements
  - And returns an iterator 'pointing' to new *logical end*

And More Algorithms

- Too many to cover
- Please find them in books and reference manuals

Outline

What We Left Out

- Much more C++ practice
  - That's your job
- Much more C++
  - Much more of **`new`** and **`delete`**
  - Much more of streams
  - Much more of STL
  - Run-Time Type Information (RTTI) support
  - Much more of everything
- Much more OO Programming
  - Much more on design
  - Much more on implementation
  - Much more on structured exception handling
  - Much more on template metaprogramming

The Present of C++

- C++11 is the official name of next C++ Standard (not covered in this course)
- The overall aims for the C++11 effort were:
  - make C++ a better language for systems programming and library building
  - make cncurrent systems programming type-safe and portable
  - make C++ a easier to teach and learn language
- C++ new features includes:
  - several additions to the core language
  - extensions to the C++ standard library (STL)

C++ 11 Core Language Improvements

- Type `long long int`
- Changes to plain old data definitions
- Generalized constant expressions
- User defined literals
- Rvalue references and move semantic
- Range based `for` loops
- Lambda functions
- Explicit conversion operators
- Extern templates
- Template aliases and variadic
- Memory model for multithreading
- ...

C++ 11 Standard Library Improvements

- Type traits for metaprogramming
- Smart pointers
- Tuples
- Hashes
- Regular expressions
- Extensible random number facilities
- Multithreading facilities
- ...

C++ Scientific Libraries

- Boost
  - Heterogeneous collection covering scientific and programming problems
  - **`http://www.boost.org`**
- Blitz++
  - Array arithmetic with Fortran performance (thanks to template metaprogramming)
  - **`http://www.oonumerics.org/blitz/`**
- Lapack++
  - Lapack C++ wrapper
  - **`http://lapackpp.sourceforge.net/`**
- Trilinos
  - Many packages for large-scale, complex multiphysics problems
  - **`http://trilinos.sandia.gov/`**
- Many more on **`http://www.oonumerics.org/oon/`**
  - Pick up those actively maintained and with live users' forums

C++


ANSI WG21
The C++ Standard Committee
`http://www.open-std.org/jtc1/sc22/wg21/`


`comp.lang.c++`
comp.lang.c++ Frequently Asked Questions
`http://www.faqs.org/faqs/by-newsgroup/comp/comp.lang.c++.htm`


B. Stroustrup
Stroustrup's Home Page
`http://www2.research.att.com/ bs/`


B. Stroustrup
The C++ Programming Language
Addison-Wesley, 3rd ed., 1997
Addison-Wesley, 4th ed., 2013 (C++11)


cplusplus.com
The C++ Resources Network
`http://www.cplusplus.com`


N. Josuttis
The C++ Standard Library: A Tutorial and Reference
Addison-Wesley, 1999


R. Lischner
STL Pocket Reference

C++ Programming


B. Eckel
Thinking in C++
`http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`


S. Meyers
Effective C++: 55 Specific Ways to Improve Your Programs and Designs
Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library
More Effective C++: 35 New Ways to Improve Your Programs and Designs
Addison-Wesley


E. Gamma, R. Helm, R. Johnson, J. Vlissides
Design Patterns: Elements of Reusable Object-Oriented Software
Addison-Wesley, 1994


Y. Shapira
Solving PDEs in C++ - SIAM, 2006
Mathematical Objects in C++ - CRC Press, 2009


J. Lakos
Large-Scale C++ Software Design
Addison-Wesley, 1996

Rights & Credits

Slides and examples were authored by:

- Michela Botti
- Federico Massaioli
- Luca Ferraro
- Stefano Tagliaventi