

# Introduction to C++

## Day 3

Marco Frailis

INAF – Osservatorio Astronomico di Trieste

Contributions from:  
Stefano Sartor (INAF)

# Classes

- A **class** type is a mechanism to:
  - **combine** related **data values** into a data structure, in order to treat it as a single entity
  - **hide implementation** details to the class user and
  - only **expose an interface**, i.e. a set of member functions and operators, to access objects of the class

```
vec3.h  
  
class Vec3 {  
public:  
    double& at(size_t i) { return elems[i];}  
    const double& at(size_t i) const { return elems[i];}  
  
private:  
    double elems[3];  
};
```

Methods implemented in the class definition

- Member functions (methods) are part of the class definition and must be called using the dot operator on an instance of the class
- Methods defined inside the class definition ask the compiler to expand calls to them **inline**

# Class members

- In general, a class is composed by a set of entities called members; we can identify the following type of members:
  - **Data attributes:** variables, constants, class variables or constants
    - Class variables are associated to the class and not to a single instance
  - **Member functions:** functions, class functions
    - Class functions do not depend on the state of a specific class instance
  - **Types:** type aliases, nested classes, etc.

# Protection labels

- In the previous example, we use the keyword **class** instead of **struct**.
- After that keyword, a **protection label** is used: **public**. It defines the two overloaded member functions 'at' as public, i.e. accessible by all users of the type
- Another protection label, **private**, defines the members that follow it (in this case just `elems`) as private, hence not accessible by the user:

```
Vec3 p;  
p.elems[0] = 1; // compilation error, elems is private
```

- Protection labels can occur in any order within the class and can occur multiple times
- Every member declared between the `{` and the first protection label is private by default (while for **struct**, the default is public)

# Const member functions

- Since an object of type `Vec3` could be passed as a const reference, it is necessary to declare which member functions don't change the object state
- One of the `at` member functions just reads the `Vec3` elements hence it is declared as a **const member function** by using the keyword **const** immediately after the parameter list
  - Such **const** is part of the method signature
- With the class `Vec3` we can perform the following operations:

```
Vec3 p;  
// setting the three elements of p using the non-const at  
p.at(0) = 1.0;  
p.at(1) = -3.0;  
p.at(2) = 4.0;  
  
// printing p  
cout << "x: " << p.at(0) << ", y: " << p.at(1) << ", z: "  
      << p.at(2) << endl;
```

# Constructors

- One problem of the Vec3 class is that we cannot initialize an object of type Vec3
- Initialization can be defined for a class by special member functions called **Constructors**
- Constructors have the same name as the name of the class itself and have no return type

```
class Vec3 {  
public:  
    Vec3();  
    Vec3(double x, double y, double z);  
    ...  
};
```

- In the class definition above we are now declaring two constructors
  - The one with an empty parameter list is called **default constructor**
  - The second constructor let us specify an initial value for the 3 elements of Vec3

# Constructor definition

- Class member functions defined outside a class definition must use the class name as scope:

vec3.cpp

```
Vec3::Vec3()  
{  
    elems[0] = elems[1] = elems[2] = 0;  
}  
  
Vec3::Vec3(double x, double y, double z)  
{  
    elems[0] = x; elems[1] = y; elems[2] = z;  
}
```

main()

```
Vec3 p; // default-constructor is called  
Vec3 q{0, 1.5, -2.4}; // the second constructor is called  
Vec3 r(q); // implicit copy-constructor used
```

- The third initialization initializes r as a copy of q by using an implicit constructor that performs a copy element-by-element
- The two constructors are defined in file vec3.cpp

# Additional member functions

```
class Vec3 {  
public:  
    // computing the norm  
    double norm() const;  
    ...  
}
```

- inline member functions must be defined in the header file (vec3.h)

```
inline double Vec3::norm() const  
{  
    return sqrt( elems[0]*elems[0] + elems[1]*elems[1] +  
                 elems[2]*elems[2] );  
}
```

- The const keyword must be used both in the declaration and definition since it is part of the method signature



# Overloading operators

- If we want to be able to add two Vec3 objects using the following notation:  $p += q$  and  $r = p + q$ . We need to overload two operators,  $+=$  and  $+$
- I could define the operation  $p + q$  as  $p.operator+(q)$  or as  $operator+(p, q)$
- The  $+=$  modifies the left operand. Hence, it must be defined as a member function
- The  $+$  doesn't modify the two operands, so it should be implemented as a non-member function

```
Vec3& Vec3::operator+=(const Vec3& s) {  
    elems[0] += s.elems[0];  
    elems[1] += s.elems[1];  
    elems[2] += s.elems[2];  
    return *this;  
}
```

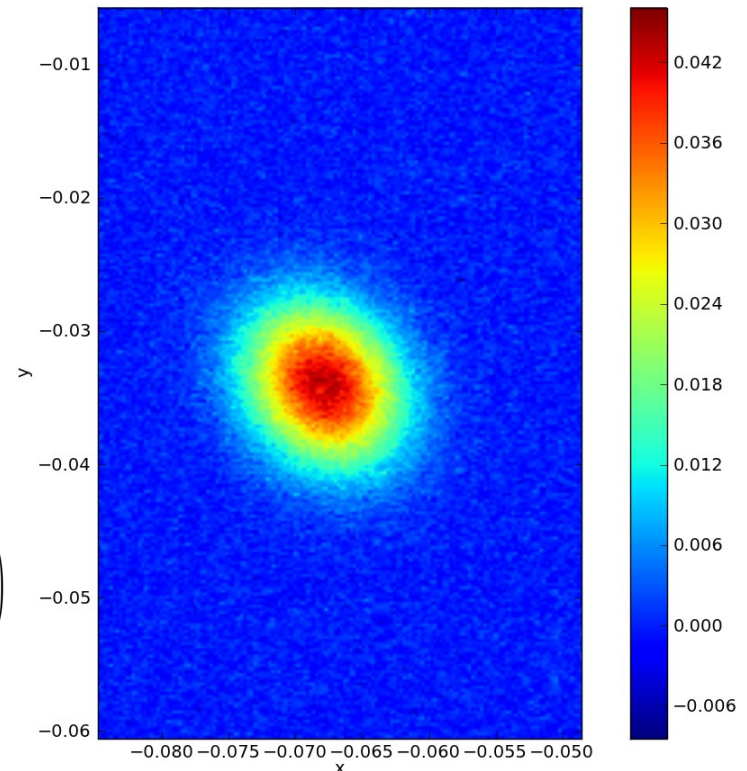
```
Vec3 operator+(const Vec3& lhs, const Vec3& rhs) {  
    Vec3 r = lhs; // implicit copy constructor used  
    r += rhs;  
    return r;  
}
```

- The **this** keyword is valid only inside a member function, where it denotes a pointer to the object on which the member function is operating

# Call operator

- The call operator `()` can be overloaded in the same way as other operators can
- One use of the operator `()` is to provide the usual function call syntax for objects that in some way behave like functions. They are called functors or function objects
- Example: Jupiter is used by the Planck-LFI mission for the in-flight reconstruction of the antenna beam shape.
- The elliptical beam can be defined as a bivariate Gaussian:

$$beam(x, y) = A \cdot e^{-\frac{1}{2} \left( \frac{(\Delta x \cos \alpha + \Delta y \sin \alpha)^2}{\sigma_x^2} + \frac{(\Delta y \cos \alpha - \Delta x \sin \alpha)^2}{\sigma_y^2} \right)}$$



# Bivariate Gaussian functor

```
class GaussianBeam
{
public:
    // Constructor
    GaussianBeam(double center_x, double center_y, double sigma_x, double sigma_y,
                 double amplitude, double orientation):
        x0(center_x), y0(center_y), sigmax(sigma_x),
        sigmay(sigma_y), amp(amplitude), alpha(orientation) {}

    // Call operator
    double operator() (double xpos, double ypos);

private:
    GaussianBeam(); // prevent use of default constructor

    // Beam center
    double x0, y0;
    // Beam dispersions
    double sigmax, sigmay;
    // Beam amplitude (kelvin)
    double amp;
    // Beam orientation (radians)
    double alpha;
};
```

beam.h

beam.cpp

```
double GaussianBeam::operator() (double xpos, double ypos)
{
    double sinalpha = sin(alpha);
    double cosalpha = cos(alpha);
    double dx = xpos - x0;
    double dy = ypos - y0;
    double partx = (dx*cosalpha + dy*sinalpha)/sigmax;
    double party = (-dx*sinalpha + dy*cosalpha)/sigmay;

    return amp*exp(-0.5*(partx*partx + party*party));
}
```

```
GaussianBeam beam = {-0.0678, -0.034, 0.0036, 0.00446, 0.044, 0.70};
```

main()

```
double x = -0.070, y = -0.035;
double measure = beam(x, y);
```

# Constructor initializers

- Let's consider the following template class definition, for image data manipulation:

```
template <class PixelT>
class Image {
public:
    using size_type = std::size_t;

    Image(size_type width, size_type height)
        : _width(width), _height(height) {_pix_array = new PixelT[_width * _height];}

    PixelT& operator()(size_type i, size_type j) {return _pix_array[i*_width + j];}
    const PixelT& operator()(size_type i, size_type j) const
        {return _pix_array[i*_width + j];}

private:
    size_type _width, _height;
    PixelT* _pix_array;
};
```

- It defines a Image data structure, with dynamic allocation on the heap memory
- The constructor uses some new syntax: between the : and the { we can have a list of constructor initializers
  - the given members are **initialized** by the compiler with the values appearing between parenthesis, before entering the constructor body

# Copy constructor

- With the previous Image class definition, if we write:

```
Image<double> a(15, 10);  
Image<double> b(a);
```

a default copy constructor is created by the compiler to copy a in b

- It copies each data element from an existing object into the new object
  - AND, when copying a pointer data element, it just copies the memory address it contains. So, a and b point to the same pixel buffer
- To avoid this behavior we should define our own **copy constructor**:

```
template<class PixelT>  
Image<PixelT>::Image(const Image& img)  
: _width(img._width), _height(img._height)  
{  
    size_type size = _width*_height;  
    _pix_array = new PixelT[size];  
    std::copy(img._pix_array, img._pix_array + size, _pix_array);  
}
```

# Improving the Image class definition 1/2

- We need to extend the Image class interface and improve its modularity

image.h

```
template <class PixelT>
class Image {
public:
    // Definition of type aliases used by the class
    using size_type = std::size_t;
    using const_iterator = const PixelT*;
    using iterator = PixelT*;

    // Default constructor
    Image(): _width(0), _height(0), _pix_array(nullptr) {}
    // Constructor requiring the number of rows and columns
    Image(size_type width, size_type height)
        : _width(width), _height(height) {_pix_array = new PixelT[_width * _height];}
    // Copy constructor
    Image(const Image& img)
        : _width(img._width), _height(img._height) {create(img.begin(), img.end());}

    // Destructor
    ~Image() {uncreate();}

    // Call operator, to access each pixel
    PixelT& operator()(size_type i, size_type j) {return _pix_array[i*_width + j];}
    const PixelT& operator()(size_type i, size_type j) const
        {return _pix_array[i*_width + j];}

    // Total number of pixels and image dimensions
    size_type size() const {return _width*_height;}
    size_type width() const {return _width;}
    size_type height() const {return _height;}
```

# Improving the Image class definition 2/2

```
// Iterators
iterator begin() {return _pix_array;}
const_iterator begin() const {return _pix_array;}

iterator end() {return _pix_array + size();}
const_iterator end() const {return _pix_array + size();}

protected:
// Helper functions to create and destroy class instances
void create(const_iterator istart, const_iterator iend);
void uncreate();

private:
// Private attributes
size_type _width, _height;
PixelT* _pix_array;
};
```

image.h

image.cpp

```
template <class PixelT>
void Image<PixelT>::create(const_iterator istart, const_iterator iend)
{
    _pix_array = new PixelT[iend - istart];
    std::copy(istart, iend, _pix_array);
}

template <class PixelT>
void Image<PixelT>::uncreate()
{
    delete[] _pix_array;
    _pix_array = nullptr;
    _width = _height = 0;
}
```

# Assignment operator

- The class definition also controls the behavior of the assignment operator (=)
  - Several overloaded assignment operators can be defined
  - However, the version that takes a const reference to the class itself is special and is considered “the assignment operator”

image.cpp

```
template <class PixelT>
Image<PixelT>& Image<PixelT>::operator=(const Image& rhs)
{
    // check for self-assignment
    if (&rhs != this) {
        // free the array in the left-hand side
        uncreate();
        // copy pixels from rhs
        create(rhs.begin(), rhs.end());
        // copy dimensions
        _width = rhs._width;
        _height = rhs._height;
    }
    return *this;
}
```

Image.cpp

```
template <class PixelT>
Image<PixelT>& Image<PixelT>::operator=(const PixelT val)
{
    std::fill(begin(), end(), val);
    return *this;
}
```



# Using the Image class

- A quick example showing the usage of the Image class:

```
int main(int argc, char* argv[])
{
    // Creates an image of double values
    // with 15 rows and 10 columns
    Image<double> a(15, 10);
    // All pixels are set to 5.0
    a = 5;
    // Setting pixel (4,5) to 10
    a(4, 5) = 10;
    // b is a copy of a
    Image<double> b(a);
    // Retrieving image dimensions and size
    auto size = b.size();
    auto width = b.width();
    auto height = b.height();
    // Resetting all pixels in a range-for loop
    for (auto& x : a)
        x = 0;
}
```

# Destructor

- Like constructors, which say how to create objects, there is a special member function, called a **destructor**, that controls what happens when objects of the type are destroyed
- An object is destroyed when it goes out of scope or, for dynamically allocated objects, when calling **delete** on the object
- For the Image class defined in the previous example, it is necessary to define a destructor, in order to correctly free the memory allocated for the `_pix_array` member
- Destructors have the same name as the name of the class prefixed by a tilde ( `~` )

```
// Destructor  
~Image() { uncreate(); }
```

# The rule of three

- Classes that **allocate resources** (in the heap) in their constructors, require that every copy deal correctly with those resources
  - Such classes almost surely need a **destructor** to free the resources
  - If a class needs a destructor, it almost surely need a **copy constructor** to control how the dynamically allocated resources are copied
  - If a class needs a copy constructor, it also needs an **assignment operator** since copying or assigning allocates those resources in the same way
- To control how every object of class T deals with its resources, you need:

```
T::T()           // one or more constructors, perhaps with arguments
T::~~T()        // the destructor
T::T(const T&)  // The copy constructor
T::operator=(const T&) // the assignment operator
```

# Reuse inheritance

- Together with an image, we also need to define a mask, i.e. an array of bitsets that provides information on the quality of each pixel in the image (e.g. Bad, Saturated, Cosmic-ray hit, etc.)
- We would like to reuse the Image definition, since its basic interface is also suitable for a Mask data structure
- We can define a Mask class as a class **derived** from the Image class
  - then Image is a **base class** of Mask
  - Mask inherits from Image “all” its attributes and its member functions, with the exception of:
    - Constructors
    - Assignment operators
    - Destructor

# The Mask derived class

```
template<class MaskPixT>
class Mask: public Image<MaskPixT> {
public:
    using Super = Image<MaskPixT>;
    using typename Super::size_type;

    // Default constructor
    Mask() {}
    // Constructor requiring image dimensions and maximum bit plane
    Mask(size_type width, size_type height,
         size_type maxBitPlane = std::numeric_limits<MaskPixT>::digits)
        : Super(width, height), _maxBitPlane(maxBitPlane) {}
    // Copy constructor
    Mask(const Mask& rhs): Super(rhs), _maxBitPlane(rhs._maxBitPlane) {}

    // Assignment operators
    Mask& operator=(const Mask& rhs);
    Mask& operator=(const MaskPixT val);

    // Method to get a specific bit of the mask
    bool getBitAt(size_type i, size_type j, size_type bitPlane)
        {return this->operator()(i,j) & getBitMask(bitPlane);}
    // Operator performing a bitwise-or with another mask
    Mask& operator|=(const Mask& rhs);

private:
    // Helper function to compute bit plane mask
    MaskPixT getBitMask(size_type bitPlane)
        {return (bitPlane >= 0 && bitPlane < _maxBitPlane) ? 1 << bitPlane : 0; }

    size_type _maxBitPlane;
};
```

Public methods in Image  
are also public in Mask

Helper type aliases

**typename** tells the compiler that `size_type` is the name of a type even if `MaskPixT` is not yet instantiated

Need to call base class constructor to initialize base class attributes

# Mask class operators

```
template <class MaskPixT>
Mask<MaskPixT>& Mask<MaskPixT>::operator=(const Mask& rhs)
{
    Super::operator=(rhs);
    _maxBitPlane = rhs._maxBitPlane;
    return *this;
}
```

```
template <class MaskPixT>
Mask<MaskPixT>& Mask<MaskPixT>::operator=(const MaskPixT val)
{
    Super::operator=(val);
    return *this;
}
```

```
template <class MaskPixT>
Mask<MaskPixT>& Mask<MaskPixT>::operator|=(const Mask& rhs)
{
    if (Super::width() != rhs.width() || Super::height() != rhs.height())
        throw std::length_error("Images are of different size");

    std::transform(rhs.begin(), rhs.end(), Super::begin(), Super::begin(),
                  std::bit_or<MaskPixT>());
    return *this;
}
```

Calling the base class  
equivalent operator  
(reuse)

# Inheritance

- When a function or method takes, as parameter, a **reference** or **pointer** to the base class, then we can pass as actual value also a reference or pointer to a derived class
  - Supposing that a function requires, as parameter, a reference or pointer to an instance of type `Image<PixelT>`
  - We can call it passing an instance of type `Mask<MaskPixT>`
  - BUT, due to the `Image` and `Mask` definitions, the function will take only the `Image` portion of the `Mask` instance
- A derived class can redefine methods of the base-class (same method name, parameters and returned type). In this manner it **overrides** (hides) the same methods from the base-class, i.e.:
  - If class `B` inherits from class `A` and both have a method `name()` defined, then:  
    `B x; // x instance has type B`  
    `x.name()` refers to the `name()` method defined in `B` and not in `A`

# Abstract class

- A file in FITS standard format usually stores an image or table data together with the associated metadata.
- Metadata is provided as a set of keywords. Each keyword has a unique name and a value. The value can be a boolean, an integer, a floating point value, a string or a complex number
- When we read the set of keywords from a FITS file, it would be useful to keep them in a `std::map`, associating the keyword name with the corresponding value
- But a `std::map` requires all values to have the same type
- Then, we need to group all possible keyword types under a common, **abstract**, parent type, that defines a set of **virtual** common operations for each keyword type
  - Virtual means that each concrete subtype will implement such set of operations, based on its own structure



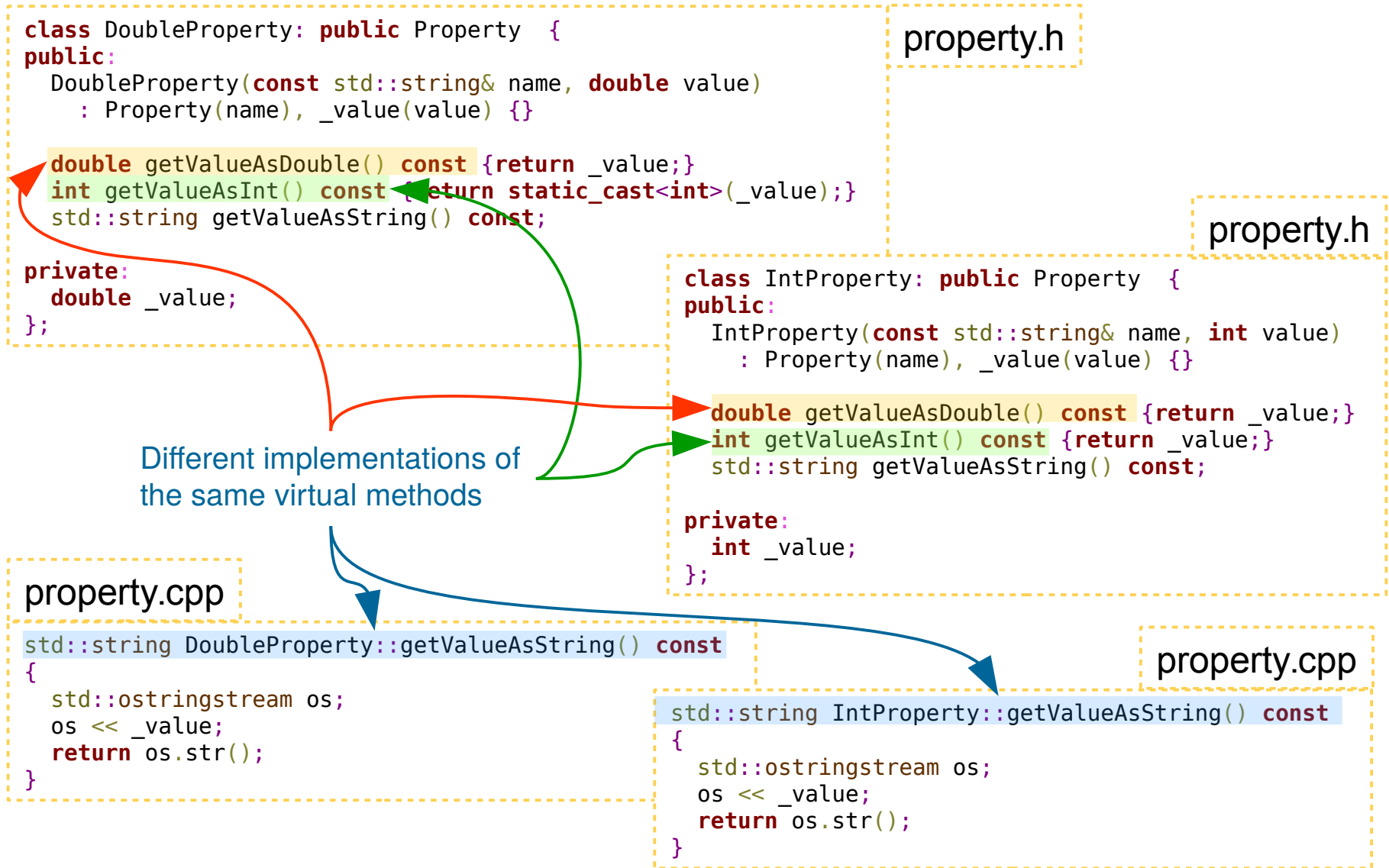
# Virtual methods example: the Property class

- Let's define a class, named Property, that represents all keyword types we can read from a FITS file (simplified version)

```
class Property {  
public:  
    Property(const std::string& name): _name(name) {}  
    const std::string& name() const {return _name;}  
  
    virtual ~Property() {} // virtual destructor  
  
    virtual double getValueAsDouble() const = 0;  
    virtual int getValueAsInt() const = 0;  
    virtual std::string getValueAsString() const = 0;  
private:  
    std::string _name;  
};
```

- The Property class has only the name (keyword name) attribute, while the concrete value will be provided by derived classes
- It defines the common set of operations expected in each derived class, declared as virtual, but it does not provide an implementation that will depend on the concrete value provided by the derived classes
- We cannot instantiate objects of type Property, since it is a pure abstract class, i.e. some virtual methods are not defined (= 0)

# Virtual methods example: two concrete derived classes



# Virtual methods example: dynamic binding

property.cpp

```
Property* createProperty(const std::string& name, double value)
{
    return new DoubleProperty(name, value);
}

Property* createProperty(const std::string& name, int value)
{
    return new IntProperty(name, value);
}

Property* createProperty(const std::string& name, const std::string& value)
{
    return new StringProperty(name, value);
}
```

```
int main(int argc, char* argv[])
{
    std::map<string, Property*> pmap;

    pmap["APID"] = createProperty("APID", 1536);
    pmap["TELESCOP"] = createProperty("TELESCOP", "PLANCK");
    pmap["MEAN_VAL"] = createProperty("MEAN_VAL", 4.434234);

    for (const auto& x : pmap){
        cout << x.first << ": " << x.second->getValueAsString() << endl;
    }
}
```

Through the base class pointer (Property\*), the specific `getValueAsString()` implementation is selected at run-time, based on the type of the object the pointer is **bound** to

# Virtual methods summary

- We can use a derived type where a pointer or reference to the base class is expected
- **Polymorphism** refers to the ability of one type to stand in for many types
- C++ supports polymorphism through the **dynamic binding** properties of the virtual methods
  - When we call a **virtual method** through a pointer or reference to a base class, we can potentially call one of many functions, as many as the number of derived types
- In the previous example, the base class Property also defines a virtual destructor
  - A **virtual destructor** is needed any time it is possible that an object of the derived type is destroyed through a pointer to base
  - The derived class inherits the virtual property of its base-class destructor

# Class methods and static attributes

- The C++ keyword `static` can be used in several contexts
- For classes, we can use the `static` keyword both when defining data attributes and member functions

```
class Exposure {  
public:  
    // Read an exposure from a fits file  
    static Exposure readFits(std::string const & filename);  
  
private:  
    // next id to use  
    static int _id;  
  
    Image<unsigned short> ccd_array;  
    Mask<unsigned short> mask;  
}
```

- A static attribute is shared by all class instances
  - If an object of the class changes the static attribute value, all the other class instances can read the updated value
- A static method can only access class static attributes and call other class methods
- Usually, class methods are called using the class scope and not through a class instance

```
Exposure e = Exposure::readFits("sample.fits");
```

# Unique and shared pointers

- Smart pointers manage the memory held by a built-in pointer
- They provide the operator “->” to access the object and the operator “\*” to retrieve the pointed object
- **unique\_ptr** allows only a single instance to hold the built-in pointer
  - When the unique\_ptr is deleted, if it is still holding a built-in pointer, it deletes it as well
  - Using unique\_ptr instead of built-in pointer does not cause overhead neither in the execution time nor in memory consumption
- **shared\_ptr** allows the built-in pointer to be shared between multiple copies
  - When all the shared\_ptr holding the same built-in pointer are deleted, the built-in pointer is deleted as well
  - They keep a reference counting of all the shared\_pointers wrapping the same built-in pointer. Increments and decrements of the counter are thread safe
  - A shared\_ptr is at least twice the size of a built-in pointer

# Using shared pointers: vector or galaxies revisited

```
#include "../ex14/galaxy.h"
#include <vector>
#include <algorithm>
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    // filling a vector of galaxy pointers with dynamically allocated instances
    vector<shared_ptr<Galaxy>> vg;
    vg.push_back(shared_ptr<Galaxy>(new Galaxy{1, {349.18372, -0.070794291}, 0.527313}));
    vg.push_back(shared_ptr<Galaxy>(new Galaxy{2, {348.3452, 0.0653423}, 0.5135289}));
    vg.push_back(shared_ptr<Galaxy>(new Galaxy{3, {346.29340, 0.034823}, 0.5126848}));

    // lambda expression to sort galaxy shared pointers by RA
    auto sort_by_ra = [] (shared_ptr<const Galaxy> x, shared_ptr<const Galaxy> y)
        {return x->coord.ra < y->coord.ra;};

    sort(vg.begin(), vg.end(), sort_by_ra);

    cout << "galaxies ordering by RA: ";
    for (const auto x : vg)
        cout << x->id << " ";
    cout << endl;

    // deleting all the dynamically allocated galaxies within the vector
    for (auto x : vg) {
        delete x;
        x = nullptr;
    }
    return 0;
}
```

shared\_ptr and unique\_ptr are declared in the <memory> header

Defining a vector of shared\_ptr to Galaxy

Galaxy built-in pointers are immediately passed to a shared\_ptr constructor

# Linear algebra with Eigen

- Eigen is a fast and elegant header only library for the linear algebra

```
#include <iostream>
#include <eigen3/Eigen/Dense>

using namespace std;
namespace E = Eigen;
int main()
{
    double a[] = {
        1, 2, 3,
        2, 1, 2,
        2, 5, 2};

    // create a 3x3 matrix of doubles using a built-in array as buffer
    E::Map<E::Matrix<double,3,3, E::RowMajor>> A(a);

    // create a vector and initialize using the initialization list
    E::Vector3d b {1,2,1};

    // solve the linear system using the LU factorization
    E::Vector3d x = A.fullPivLu().solve(b);

    // use overloaded arithmetic operators with matrices and vectors
    double relative_error = (A*x - b).norm() / b.norm(); // norm() is L2 norm

    cout << "A: \n" << A << endl;
    cout << "Solution: \n" << x << endl;
    cout << "The relative error is: " << relative_error << endl;
}
```

- For a quick-reference, see [http://eigen.tuxfamily.org/dox/group\\_\\_QuickRefPage.html](http://eigen.tuxfamily.org/dox/group__QuickRefPage.html)



# Example with CCfits and Eigen libraries

```
#include <CCfits/CCfits>
#include <iostream>
#include <valarray>
#define EIGEN_DEFAULT_TO_ROW_MAJOR
#include <eigen3/Eigen/Dense>

using namespace CCfits;
using namespace std;

int main(){

    bool readAllInMemory = true; //read data at Fits object construction
    unique_ptr<FITS> pInFile(new FITS("sample.fits", Read, readAllInMemory));

    ExtHDU& image = pInFile->extension(1);

    valarray<double> v;

    // copy the image in the container
    image.read(v);

    // create an Eigen array of doubles
    Eigen::ArrayXXd m(image.axis(0), image.axis(1));

    //copy the elements in the array
    size_t i = 0;
    for(auto x : v)
        m(i++) = x;

    cout << m << endl;
}
```

# Using C libraries from C++

- C++ type system is more restrictive than the C one
- linkage specification allows the compiler to:
  - use C type system ( only in the linkage specification scope )
  - link object files compiled with C compilers

```
#include <iostream>
#include <exception>
#include <memory>

extern "C" {
#include <fitsio.h>
}
```

// C++ header

// C header

Put C headers inclusion within the extern "C" linkage specification

# Example: cfitsio and templates 1/2

```
#include <iostream>
#include <exception>
#include <memory>
#include <sstream>
#include <cstdio>
#define EIGEN_DEFAULT_TO_ROW_MAJOR
#include <Eigen/Dense>
```

Partial template specialization  
provided as an alias

```
extern "C" {
#include <fitsio.h>
}
```

```
using namespace std;
template<class T> using DMatrix = Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>;
```

```
template<typename T>
class PixelType{
public:
    static const int type;
    static const int fits_type;
};
```

Template specialization

```
template<> const int PixelType<unsigned char>::type = BYTE_IMG;
template<> const int PixelType<unsigned char>::fits_type = TBYTE;
```

```
template<> const int PixelType<float>::type = FLOAT_IMG;
template<> const int PixelType<float>::fits_type = TFLOAT;
```

```
template<> const int PixelType<double>::type = DOUBLE_IMG;
template<> const int PixelType<double>::fits_type = TDOUBLE;
```

```
...
```

# Example: cfitsio and templates 2/2

```
template<typename T>
DMatrix<T> read_image(const string& file_name, int extension=0){
    fitsfile *fptr;
    int bitpix, naxis, an;
    int status = 0;
    long naxes[2];
    stringstream ss;
    ss << file_name << "[" << extension << "];

    fits_open_file(&fptr,ss.str().c_str() , READONLY, &status);
    if(status) throw FitsException(status);

    fits_get_img_param(fptr, 2, &bitpix, &naxis, naxes, &status);
    if(status) close_and_throw(fptr, FitsException(status));

    if(PixelType<T>::type != bitpix) close_and_throw(fptr, domain_error("wrong pixel type"));
    if(naxis != 2) close_and_throw(fptr, domain_error("non 2D image"));

    DMatrix<T> m(naxes[0],naxes[1]);
    long fpixel[] = {1,1};
    fits_read_pix(fptr,PixelType<T>::fits_type,fpixel,naxes[0]*naxes[1],NULL,m.data(),&an,&status);

    if(status) close_and_throw(fptr, FitsException(status));

    fits_close_file(fptr, &status);
    return m;
}

int main(int argc, char *argv[]){
    DMatrix<float> m = read_image<float>("sample.fits", 1);
}
```