Data Analysis Using C++

or, better perhaps

Getting you started with C++ and OO

This little course is based on a little C++ introduction I gave about 2 years ago. I want to show you some concepts of object-oriented programming, and show you some examples which get you started.

The code examples are available on the Web, so you can go and run the examples on your own (highly recommended!). There are often different versions of a certain solution, so the “best” or “final” one is marked as such.

This is no substitute for two or three good books, or for the “design patterns” book.

We will assume some familiarity with C, and with a procedural language such as Fortran.

There will be examples from your everyday life as a physicist later in this course, but there will be other simple examples first to show some basic concepts. Bear with me.

Our Goals

After we are through...

• you should have some notion of the difference between the procedural (Fortran,C) and object-oriented (C++, Java) programming approach;

• you should be able to understand existing C++ programs;

• you should be be able to use existing class libraries;

• you should be able to develop simple classes of your own.

• you should be familiar enough with the concept so that you can continue to learn just from the books.

Object Oriented Programming in C++

• In our real life, we are used to deal with objects. If you want a toast, you use your toaster. For the dishes, you use your dishwasher. You use a TV, VCR, washing machine, and tools such as a drill. Do you know how your washing machine works in all detail? Think of all those applicances/devices as objects which are tools providing a service, usually without making you aware of their inner workings.

• If you transfer that tool concept to programming, you will want a file manager object if you deal with files, a display object to display your graphics, and so on. Like in your real life.

• Only years of procedural computer language training makes us perceive Fortran and C as the natural way of designing programs. It is really the way the F77 and C compilers need us to pre-arrange their work to make the job easier for them.

• After a while the natural way of thinking will come back and you will see OO as the natural way of attacking problems.

• Except for the small hurdle of learning a modest amount of new syntax constructs, C++ is a lot easier to master than Fortran (I hope that you will agree with me after this course).

Supposed benefits of OO

If you read any book about “Programming in an Object-Oriented Language”, you will see a lot of talk about the benefits you may expect… Among them

• faster code development
• reusability of existing code
• modularity of the code
• applicability to a wider range of problems
• < no end to this list... >

Is it true?

First, you pay a price. Most of the software you have and you wrote is probably not in C++.

The “faster code development” applies only if most of the existing code is already in C++, and you can use a lot of existing class libraries (very true for JAVA, by the way… different topic).

Reusability: Indeed. You have to make it re-usable, though. But you can do more in this respect than F77 or C allows you to do.

Modularity of code: it does not come by ityself, you have to make it so. The best tools don’t help you if you don’t know how to use them. But yes, if you make it modular, the language will support you in making it so.

Wider range of problems… I don’t have a strong opinion.

I personally think...

Since I joined PHENIX in 1996, I have written virtually everything in C++ (and Java). Here it really starts to pay off. The maintenance of the packages which you may have seen (PDAQ, Event libs) has become very easy, and over time I made a lot of extensions which were easy to do, and did not break any existing code.

Code development became very easy, because so far it turned out that there no major design flaws in the class model.

I did not hit any of the proverbial walls which you hit (and I used to hit) with huge Fortran programs. Side effects which one class has on others are small and much better controllable, there is no need for Common block-alikes, and so on. I routinely code also small utility programs in C++.

Yes, I believe that object-oriented programming is the future. We will all see a lot of C++, with Java a close second.

Buzzwords you may have heard

Inheritance -- Information hiding -- Polymorphism -- Overloaded functions -- Type-safety -- Overriding -- Association -- Helper Objects -- Operator overloading -- Constructors -- Destructors -- Proxy objects -- Design patterns -- Late binding -- Object relationships -- Singletons -- Object factories -- Container classes -- ...

Let’s pick the most important ones and see what’s behind and what they are good for.

Towards our first C++ example

In our example, we are dealing here with simple geometrical bodies, such as cylinders and cubes (and maybe many more)

We are -- for now -- interested in only two aspects of those bodies: their volume and their surface.

Obviously these (having a volume and having a surface) are properties of all bodies. For each body there are a well-defined numbers for both quantities.

This is what we will (in a moment) use to show off the OO implementation of cylinders and cubes.

Some technical stuff so we have something we can use

print something to the terminal (stdout):

```
cout << ”text” << endl;
```

(think of this as “drop something on stdout”)

standard error:

```
cerr << ”text” << endl;
```

“end this line, start a new one”

cout and cerr are provided by the C++ system.

you can “drop” more than one thing at a time on cout:.

```
int i = 5;
cout << ”i = ”;
cout << i;
cout << endl;
```

```
int i=5;
cout << ”i = ” << i << endl;
```

in order to use it, we must include the definition file:

```
#include <iostream.h>
```

```
#include <iostream.h>
int main()
{
  int i = 5;
  cout << ”i = ” << i << endl;
  return 0;
}
```

A word on the header (.h) files

Header files are supposed to contain definitions only, and not actual code. If you include code in a header file, this will usually lead to linker errors. Header files will be included in many places. Never put any actual code in header files.
Definitions must be made only once per compiler run; so always prevent a header file from being included more than once with a preprocessor construct:

file `pi.h`:

```
#ifndef __PI_H__
#define __PI_H__

#define PI  3.141592654

#endif /* __PI_H__ */
```

The statements will only be "seen" if the "`__PI_H_`" is not defined, that means, if we include the file again, we will be ok.

We have functions in emacs which automatically generate the "ifndef… define … endif" enclosure for header files.

A trivial "cylinder", non-OO

A cylinder is defined by its radius and it height. You are familiar with this:

```
#include <iostream.h>
#define PI 3.141592654
struct cylinder
{
  float radius;
  float height;
};
int main()
{
  float volume;
  struct cylinder c;
  c.radius = 2.;
  c.height = 1.;
  volume =  PI*c.radius*c.radius  * c.height;
  cout << "volume is " << volume << endl;
  return 0;
}
```

You use a `struct` to group several simple entities to a more complex entity.

But you still have to know how to use that information to, say, calculate the volume.

Now a “cylinder” class

This looks very similar:

```
class cylinder {

protected:
  float radius;
  float height;

public:
  float GetVolume();
  float GetSurface();

  cylinder (float,float);
};
```

Here you group not only variables, but also access methods

And this is new, a “constructor”, which will build an object of class “cylinder” for you.

Using our new class

```
class cylinder {

protected:
  float radius;
  float height;

public:
  float GetVolume();
  float GetSurface();

  cylinder (float,float);
};
```

```
float cylinder::GetVolume(){
 return ( PI*radius*radius  * height );
}

float cylinder::GetSurface() {
 return ( 2*PI*radius*height + 2*PI*radius*radius);
}

cylinder::cylinder(float r, float h){
 radius = r;
 height = h;
}
```

```
#include <iostream.h>
main()
{
  float volume;
  cylinder c( 2. , 1. );
  volume = c.GetVolume();
  cout << "volume is " << volume << endl;
}
```

Here we construct a new cylinder with radius 2 and height 1.

Then we ask the cylinder to please calculate its volume and return it to us

BTW, the output is: `volume is 12.5664`

What have we gained?

Rather than doing (and having to know) all the volume and surface calculations in our code, we ask the object questions and let it do the math.

We (as the author of the main program) need not be concerned about the inner workings of the cylinder class, nor about how the volume is actually calculated.

We are not aware how the cylinder object stores information about itself (radius, height) internally; we do not see (and cannot access) those variables, nor can we make assumptions about their meaning.

If the class programmer thinks of a better way to describe a cylinder (say, diameter, height) in the future, only the few functions in the class will need to be changed, none of our user code.

first buzzword!

This is called “information hiding”.

Example of information hiding

Imagine you ask someone from a different experiment what the momentum distribution of neutral particles is because you want to compare their findings with what you have got. What can happen is

a) some nice guy does or already did that analysis, gives you a table, and you are done.

b) somebody gives you a number of convoluted tables, numbers, and instructions, and says “now you can easily calculate the distribution… oh, one more thing, don’t forget the correction for multiplicity…”

Information hiding can make your life a lot easier.

Next Buzzwords: Inheritance and Polymorphism

Something the human brain is good at -- figure out commonalties between different things and categorize and classify things by their properties or behavior

Example: “mammals” - summarizes a lot of the commonalties between a certain class of animals (has a spine, breast-feeds the offspring, has warm blood, needs oxygen...)

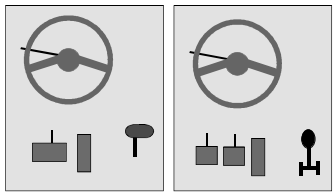
You save a lot of work by categorizing different things and dealing with the different categories most of the time.

“Can I use your Phone?”

Q. If you ask somebody whether you may use his or her phone, what makes you so sure that you know how to use it?
A.: The common interface to (virtually) all phones:

How many different cars can you drive?

There are two essential interfaces to the overwhelmingly complicated internals of modern cars:



Without a common interface, you would need to learn how to drive each new model/vendor of car -- a nightmare!

What we see: through a generic interface we can interact with a huge variety of different types/brands/models of an “object” in the same way (wouldn’t it be great if this were the case with VCR’s?)

Let’s make a cube

```
class cube {

protected:
  float length;

public:
  float GetVolume();
  float GetSurface();

  cube(float);
};
```

```
float cube::GetVolume(){
 return length*length*length;
}

float cube::GetSurface() {
 return 6.*length*length;
}

cube::cube(float x){
 length = x;
}
```

```
#include <iostream.h>
main()
{
  float volume;
  cube c( 2.);
  volume = c.GetVolume();
  cout << "volume is " << volume << endl;
}
```

Here we construct a new cube with size 2

Then we ask the cube to please calculate its volume and return it to us

Does this look familiar?

Extracting the commonalties

Obviously, having a volume and a surface is common to all bodies. In general, they share a lot of properties, in fact, all cubes, cylinders, cones, pyramids, spheres, ellipsoids, and so on, are bodies.

Let's extract the commonalties so that we treat different entities with common properties the same.

buzzword!

What we achieve is polymorphism, what we use here is called inheritance.

Commonalties checklist

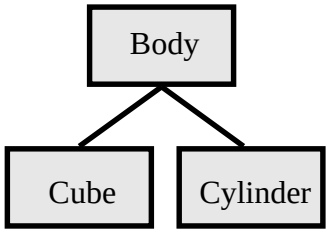
Do all bodies have a surface? Yes.

Do all bodies have a volume? Yes.

Does is take the same amount of information to describe all bodies? No.

Is the volume and surface calculated in the same way for all bodies? No.

What we will do:



The “body” class

```
class body {
public:
  virtual float GetVolume()  =0;
  virtual float GetSurface() =0;
};
```

Meaning

```
virtual float GetVolume() =0;
```

this function can be overridden by a more specific one in a subclass

function returns a float value

this virtual function must in fact be overridden; the class “body” cannot be instantiated, only a specific sub-class.

The C++ code

```
class body {
public:
  virtual float GetVolume()  =0;
  virtual float GetSurface() =0;
};
```

```
class cube : public body {

public:
  cube (float);
  virtual float GetVolume();
  virtual float GetSurface();

protected:
  float length;

};
```

The functions in the cube class override the functions in “body”

```
class cylinder : public body {

public:
  cylinder (float,float);
  virtual float GetVolume();
  virtual float GetSurface();

protected:
  float radius;
  float height;
};
```

The functions in the cylinder class override the functions in “body”

Now both “cube” and “cylinder” inherit from the class “body”, that means, they are in fact bodies (although different ones).

Ok, ok, but what is it good for?

A pointer to an object of class “body” can now point to either a “cube” or a “cylinder” object; so we have a common “handle” for both of them:

```
#include "cube.h"
#include "cylinder.h"
main()
{
  body *p1;
  body *p2;

  p1 = new cube(2.5);
  p2 = new cylinder (2., 5.);

  cout << "Volume:  " << p1->GetVolume() << endl;
  cout << "Surface: " << p1->GetSurface() << endl;

  cout << "Volume:  " << p2->GetVolume() << endl;
  cout << "Surface: " << p2->GetSurface() << endl;
}
```

both p1 and p2 are “pointers to an object of class body”

the are assigned pointers to an object of type cube and cylinder, respectively.

Automatically the proper member functions of cube or cylinder are called.

If it is a body, it has functions GetVolume and GetSurface which we can call.

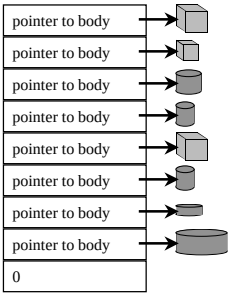
a “new” object...

```
p1 = new cube(2.5);

p2 = new cylinder(2., 5.);
```

the new operator dynamically creates (instantiates) a new object of the specified class.
It allocates the memory required to set up the new object, and initializes variables as needed.
It then calls the appropriate constructor for the new object.
It then returns a pointer to the just created object.
In our code, we can dynamically create and delete as many objects as we need (resources permitting). This is a very powerful concept.

Now watch...

0-terminated list of "pointers to body"



This routine now calculates the total volume of all bodies in the list

```
#include "body.h"
float totalVolume (body *bptr[])
{
  int i = 0;
  float volume = 0.;
  body *b = bptr[i++];
  while (b)
    {
      volume += b->GetVolume();
      b = bptr[i++];
    }
  return volume;
}
```

This routine only sees (and knows about) "bodies", it does not care whether a body is actually a cube or a cylinder. It will also work (without re-compilation!) with other classes of body which didn't exist at the time we wrote this function, such as a sphere, a pyramid, or a cone.

"list of Bodies" code example

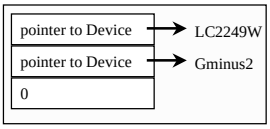
```
#include "cube.h"
#include "cylinder.h"
#include "iostream.h"

// the declaration of the totalVolume call.
float totalVolume (body *bptr[]);
main()
{
  body *bodylist[10];
  bodylist[0] =  new cube(1.25);
  bodylist[1] =  new cube(1.);
  bodylist[2] =  new cylinder(1,1);
  bodylist[3] =  new cylinder(0.5,2);
  bodylist[4] =  new cube(1.25);
  bodylist[5] =  new cylinder(1,2);
  bodylist[6] =  new cylinder(1,0.5);
  bodylist[7] =  new cylinder(3,2);
  bodylist[8] =  0;
  cout << "Total volume is: " <<  totalVolume(bodylist) << endl;
}
```

```
It yields
Total volume is: 74.0213
```

Real-life example… PDAQ

1998 Drift chamber definition file (used @ Stony Brook):

```
daq_create_readoutdevice ("LC2249W",1,1005,0,1,9,0)
daq_create_readoutdevice ("Gminus2",1,1010,0xe0000000)
```




```
class Device{
public:
  // ...stuff deleted...
  virtual int max_length(int etype)=0;
  virtual int init();
  virtual int put_data(int, int *,int) =0;
};
```

(… there are many more functions which aren't shown here)

the maximum amount of data this device can produce (worst-case, if no zero-suppression is possible)

have the device initialize "itself" (the object does whatever it takes to initialize the hardware properly)

the readout: ask the device to put its data to the address specified. We tell it how much data it is allowed to put there at most (safeguard), and it returns the amount it actually put there.

Once we have created the proper device object (which matches the hardware) with the proper parameters, for most purposes we don't need to know anymore what the hardware is in detail. We know we can ask *any* device to initialize, readout, tell us its max length, etc.

Latest addition: The packet_dcm0 class, which reads out the PHENIX data collection modules ("DCM") through the so-called "semaphore" readout.

Pdaq device objects at work...

For most purposes we don't need to know anymore what the hardware is in detail, the knowledge that it is a "Device" will suffice:

Finding out what the maximum event length is...

```
idev = 0;
while ( (device = devicelist[idev++] ) )
    size += device->max_length(i);
```

Initializing all devices...

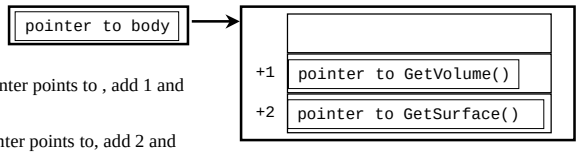
```
idev = 0;
while ( (device = devicelist[idev++] ) )
    device->init();
```

Reading out all devices...

```
idev = 0;
while ( (device = devicelist[idev++] ) )
  len += fillBuffer->addSubevent(device);
```

In each of the code snippets, we loop through all Device objects and call the appropriate member function. The while loop will stop when we reach the final entry (which is "0").
If we add a new device, we do not need to modify or recompile the code that deals with the Devices. This part of the code has essentially been unchanged since 1997.

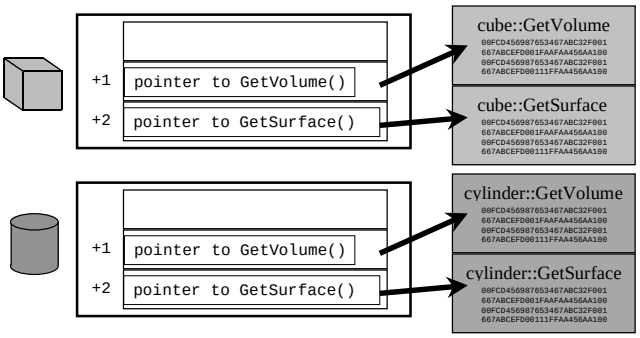
How does this object stuff work?



The recipe for the compiler:

To call GetVolume(): to where the pointer points to , add 1 and call the routine this location points to.

To call GetSurface(): to where the pointer points to, add 2 and call the routine which this location points to.



All the compiler needs to know is that at the locations pointed to it will find the appropriate routines to call. It doesn't need to know what those routines do or how they do it, only what the recipe is.

So what? you may say… I can do the same thing in C!

```
#include "body.h"
float totalVolume (struct *bptr[])
{
  int i = 0;
  float volume = 0.;
  struct body *b;
  *b = bptr[i++];
  while (b)
    {
      if (b->type == CUBE)
        volume += CubeVolume(b);
      elseif (b->type == CYLINDER)
        volume += CylinderVolume(b);

      b = bptr[i++];
    }
  return volume;
}
```

```
#include "body.h"
float totalSurface (struct *bptr[])
{
  int i = 0;
  float surface = 0.;
  struct body *b;
  *b = bptr[i++];
  while (b)
    {
      if (b->type == CUBE)
        volume += CubeSurface(b);
      elseif (b->type == CYLINDER)
        volume += CylinderSurface(b);

      b = bptr[i++];
    }
  return surface;
}
```

All right, that would work. But what if you get a new type of body, say, a Cone?

Adding a cone in C

```
#include "body.h"
float totalVolume (struct *bptr[])
{
  int i = 0;
  float volume = 0.;
  struct body *b;
  *b = bptr[i++];
  while (b)
    {
      if (b->type == CUBE)
        volume += CubeVolume(b);
      elseif (b->type == CYLINDER)
        volume += CylinderVolume(b);
      elseif (b->type == CONE)
        volume += ConeVolume(b);

      b = bptr[i++];
    }
  return volume;
}
```

```
#include "body.h"
float totalSurface (struct *bptr[])
{
  int i = 0;
  float surface = 0.;
  struct body *b;
  *b = bptr[i++];
  while (b)
    {
      if (b->type == CUBE)
        surface += CubeSurface(b);
      elseif (b->type == CYLINDER)
        surface += CylinderSurface(b);
      elseif (b->type == CONE)
        surface += ConeSurface(b);

      b = bptr[i++];
    }
  return surface;
}
```

This looks easier than it is in practice...

- For each new body type added, you must find and change all locations where a type-specific decision is made.
- In a large package, these may be hundreds of locations in a large number of different files.
- You must get them all, or you have a nice little bug.
- You can easily introduce new bugs because you are forced to change existing, debugged (and potentially certified) code.

Having to modify debugged code is bad.

This is what you wanted...

```
  while (b)
    {
      if (b->type == CUBE)
        surface += CubeSurface(b);
      elseif (b->type == CYLINDER)
        surface += CylinderSurface(b);
      elseif (b->type == CONE)
        surface += ConeSurface(b);
      b = bptr[i++];
    }
  return surface;
}
```

This is what you pasted from getVolume...

```
  while (b)
    {
      if (b->type == CUBE)
        surface += CubeSurface(b);
      elseif (b->type == CYLINDER)
        surface += CylinderSurface(b);
      elseif (b->type == CONE)
        surface += ConeVolume(b);
      b = bptr[i++];
    }
  return surface;
}
```

This is because it was 2:30 am...

```
  while (b)
    {
      if (b->type == CUBE)
        surface += CubeSurface(b);
      elseif (b->type == CYLINDER)
        surface += CylinderSurface(b);
      elseif (b->type == CONE)
        surface += CylinderSurface(b);
      b = bptr[i++];
    }
  return surface;
}
```

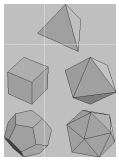
This is because you forgot about this routine...

```
  while (b)
    {
      if (b->type == CUBE)
        surface += CubeSurface(b);
      elseif (b->type == CYLINDER)
        surface += CylinderSurface(b);


        b = bptr[i++];
    }
  return surface;
}
```

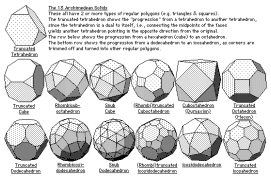
If you are forced to modify debugged code, you take a chance to introduce new bugs which are really hard to find.
The compiler will happily compile the faulty code and will not help you to find those bugs.

And then there is the speed issue

```
while (b)
  {
    if (b->type == CUBE)
      surface += CubeSurface(b);
    elseif (b->type == CYLINDER)
      surface += CylinderSurface(b);
    elseif (b->type == SPHERE)
      surface += SphereSurface(b);
    elseif (b->type == PYRAMID)
      surface += PyramidSurface(b);
    elseif (b->type == CONE)
      surface += ConeSurface(b);
    elseif (b->type == TORUS)
      surface += TorusSurface(b);
    elseif (b->type == TETRAHEDRON)
      surface += TetrahedronSurface(b);
    elseif (b->type == OCTAHEDRON)
      surface += OctahedronSurface(b);
    elseif (b->type == DODECAHEDRON)
      surface += DodecahedronSurface(b);
    elseif (b->type == ICOSAHEDRON)
      surface += IcosahedronSurface(b);
      elseif (b->type == ELLIPSOID)
        surface += EllipsoidSurface(b);
      elseif (b->type == PARABOLOID)
        surface += ParaboloidSurface(b);
      elseif (b->type == TTETRAHEDRON)
        surface += TTetrahedronSurface(b);
      elseif (b->type == TOCTAHEDRON)
        surface += TOctahedronSurface(b);
      elseif (b->type == TDODECAHEDRON)
        surface += TDodecahedronSurface(b);
      elseif (b->type == TICOSAHEDRON)
        surface += TIcosahedronSurface(b);
      elseif (b->type == TELLIPSOID)
        surface += TEllipsoidSurface(b);
      elseif (b->type == PARABOLOID)
        surface += ParaboloidSurface(b);
      b = bptr[i++];
    }
  return surface;
}
```



So for each volume calculation there is a lot of decision-making until we have the proper routine. Compare that to our simple recipe with pointers to a function.
With objects, most of the decisions about functions are made at the time the object is constructed.



Let's pause for a moment and review what we've got.

Information hiding	We learned that objects provide a service for us and we don't need to know how they do it.
Inheritance	We know how to subclass [inherit from] other classes.
Polymorphism	We know that a pointer to the parent class can point to any child class, so a routine dealing with the parent class will deal with any child class.
	We learned that we can make new objects "on the fly" with the new operator. This is one of the keys to a very flexible program.
	We saw that modifications/additions to existing code (the addition of a 'cone" object) can be kept local; no other code needs to be changed as a result of the addition.

But inheritance/polymorphism is only the smaller part...

If you want to give an object an attribute (such as color), inheritance is not the right way.

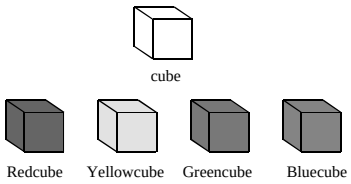
With the colored cubes on the right, this would lead to a high number of classes.

Inheritance says "This object is (a special form of) another object."

We want to look at another technique now, the association.

Association says "This object has / uses / contains another object"

Here: "our cube (or, any body for that matter) has a color."



Advantages:

If you have n different classes and m different attributes, using inheritance would yield n * m resulting classes if you want to provide all possibilities. If you have another attribute (say, flavor), with k possibilities, you get n * m * k resulting classes. This can easily lead to an extremely high number of classes.

Inheritance is static, while associations are dynamic. If color is a property of the class as in the cartoon, you cannot change the color except by deleting the object and creating a new one. But if "color" is just an attribute, you can change it easily for an existing object. The associaton technique lets you combine attributes like you shop for a car: you choose a base model and then you can decide on color, sunroof, radio, interior, tires, etc. You can also change attributes ("which radio") at any time by just replacing it.

Bodies with Color

Let’s make some colorful bodies...

The idea: Make a separate class “Color” and let the Body class have a pointer to an object of class “Color”.
The advantages:
• we can get a “standalone” Color class right off the shelves (the Web in this case);
• we can use the same Color class for other objects as well;
• we can replace the Color class in the future with a better one with very little work;
• we can now change the color of an object at any time;
• we can extend any of the classes independently.

In general, the association (“has a”-relationship) allows for much more flexibility both at compile time and at runtime, and makes the classes you write much more re-useable.
This is the toolbox approach:
Make simple classes which do one job well, and use them in many different contexts.

“The use of associations and shallow inheritance trees are the hallmarks of good OO programmers.”

Our Body has a Color

```
#ifndef __BODY_H__
#define __BODY_H__

#include "Color.h"

class body {
public:

  // the constructors for body
  body();
  body(Color *c);

  // the destuctor
  ~body();

  // the member functions we already know
  virtual float GetVolume()  =0;
  virtual float GetSurface() =0;
  // two more functions which deal with the color
  virtual void setColor (Color *c);
  virtual Color * getColor();

protected:
  Color * color;
  int our_color;
};

#endif /* __BODY_H__ */
```

```
#include "body.h"
// the standard constructor first
body::body()
{
  // since we didn’t specify a color...
  color = new Color(0.,0.,0.);
  our_color = 1;
}
// the constructor setting a color
body::body(Color *c)
{
  // we just grab the pointer to the color provided.
  color = c;
  our_color = 0;   // we did not make the color class.
}
// and the destructor
body::~body()
{
  // if we made the color object, we must delete it here.
  if (our_color) delete color;
}
// now the color-handling
void body::setColor (Color *c)
{
  if (our_color) delete color;
  our_color = 0;
  color = c;
}
Color * body::getColor()
{
  return color;
}
```

Warning: this class design works but is flawed. We will later improve the design.

Constructing with Color

cube and cylinder get a new constructor:

```
cube::cube(float x)
{
  length = x;
}


cube::cube(float x, Color *c)
  : body(c)
{
  length = x;
}
```

```
cylinder::cylinder(float r, float h)
{
  radius = r;
  height = h;
}

cylinder::cylinder(float r, float h, Color *c)
  : body(c)
{
  radius = r;
  height = h;
}
```

They pass the color on to the constructor for body which takes care of things.

The first constructor calls (by default) body() without parameters; this constructor assigns the default color (black) to the body.

What we see:

- each constructed cube and cylinder just lets the parent class deal with the color attributes; there is no need to duplicate that functionality.
- the cube & cylinder classes inherit the functionality from their parents
- the parent (body) maintains the color as an associated class.
- we can change the color attribute at run time.

Let's make sure we understand what we did

In the original example, the abstract "body" class was used just as a handle to deal with cubes and cylinders in the same way. It did not contribute functionality at this time.
Now, however, the whole business with the color is done by the body parent class. The "children" just pass on the color information in their constructor; from that point on they are no longer involved in any color-related business. *They inherit this functionality from their parent class.*
There is no proliferation of the color-handling code. It's all contained in the code for the body class.
In real life, this is the norm: Lots of aspects of the classes are dealt with at a high level in the inheritance tree, so that code does not get duplicated. Only the code which is truly class-specific (such as the getVolume and getSurface) "lives" with the actual class.

Also, for the first time, we saw a destructor. It has the name ~<classname>, ~body, ~cube, etc. We need it here to get rid of memory which we allocated. Destructors are usually found in all class definitions except in the most simple ones. More about destructors later.

Using the colorful new bodies

```
#include "cube.h"
#include "cylinder.h"
#include "iostream.h"
#include "Color.h"

main()
{
  body *b1;
  body *b2;
  body *b3;
  body *b4;
  body *b5;

  Color *yellow  = new Color(  1.0,  1.0,  0.0 ); // yellow
  Color *green   = new Color(  0.0,  1.0,  0.0 ); // green

  b1 = new cylinder (1,2);        // default-colored cylinder
  b2 = new cylinder (1,2,green);  // green cylinder
  b3 = new cylinder (1,2,yellow); // yellow cylinder
  b4 = new cube(1.25);            // default-colored cube
  b5 = new cube(1.25, green);

  Color *c =  b1->getColor();
  cout << " red: "   << c->red()
       << " green: " << c->green()
       << " blue: "  << c->blue()
       << endl;

  c =  b2->getColor();
  cout << " red: "   << c->red()
       << " green: " << c->green()
       << " blue: "  << c->blue()
       << endl;
}
```

red: 0 green: 0 blue: 0

red: 0 green: 1 blue: 0

Why is this design flawed?

If we specify a color either at construction time or through the setColor call, the object copies the pointer to the external Color class to its internal pointer.

So our objects are in fact pointing to other objects outside their scope.

We (the user) might choose to delete the Color object, leaving the internal pointer pointing to an invalid location, or we might change the Color object directly and so change the cube's or cylinder's properties behind its back.

Occasionally this is a desired functionality, but there are better and safer implementations for that.

On our way to a better solution, we will now introduce several new concepts:

- **the `const` keyword;**
- **overloaded functions and operators**
- **the object reference;**
- **the copy constructor**

We will see the concept of a client-server contract which is important for distributed computing.

`const` - a client-server contract

We have seen the definition

```
class body {
public:

  // the constructors for body
  body();
  body(Color *c);

  // the destuctor
  ~body();

  // the member functions we already know
  virtual float GetVolume()  =0;
  virtual float GetSurface() =0;
  // two more functions which deal with the color
  virtual void setColor (Color *c);
  virtual Color * getColor();
//… stuff deleted
```

when we call
`body (Color *c)`
we do not want really want to change c, or do we? No.
So we make it explicit:
`body (const Color *c)`
If we now attempt to modify the object which c points to anyway, we get a compiler error.

GetVolume and getSurface just return values, they do not actually change the object's properties. We inform the compiler:
`virtual float GetVolume() const = 0;`
tells the compiler that as a result of this call, the object is supposed to remain unchanged.

This is called a client-server contract.

The server guarantees that it won't modify an object that we let it use, and we really can rely on the contract. The const function definition will catch all attempts to "illegally" modify the object, and eases the bookkeeping work in a distributed environment.

Note: `float GetVolume()const` and `float GetVolume()` are considered two different functions!

`const` can save you a lot of trouble and time

```
class constviolate1 {
private:
  int a;
public:
  constviolate1(const int i);
  virtual int get_a() const;
};

constviolate1::constviolate1(const int i)
{
  a =i;
}

constviolate1::get_a() const
{
  a = 5;
  return a;
}
```

The compiler does not let you violate the contract!
The const says that `get_a` does not change the object in any way, so the compiler produces an error.

```
$ g++ -c const_violation.C
const_violation.C: In method 'int constviolate1::get_a() const':
const_violation.C:17: assignment of read-only member 'int
constviolate1::a'
```

Here you said that the parameter won't get modified!

```
constviolate1::constviolate1(const int i)
{
  a =i;
  i =a;
}
```

```
 g++ -c const_violation.C
const_violation.C: In method 'constviolate1::constviolate1(int)':
const_violation.C:13: assignment of read-only parameter 'int
const i'
```

Use of the `const` keyword will catch all those errors at compile time.
The largest impact, however, is when it comes to distributed computing (CORBA, ORB,…). If the compiler knows that remote object or the calling parameters won't be modified, the operation can be executed a lot faster and with less overhead.

Fixing (some of) Body's design flaws now...

Let's make a client-server contract and add the `const` keywords where they belong:

```
class body {
public:

  // the constructors for body
  body();
  body(const Color *c);

  // the destuctor
  ~body();

  // the member functions we already know
  virtual char* identify() const  =0;
  virtual float GetVolume() const =0;
  virtual float GetSurface() const =0;
  // two more functions which deal with the
color
  virtual void setColor (const Color *c);
  virtual Color * getColor() const;

protected:
  Color * color;
  int our_color;
};
```

But the compiler is not happy...

```
body::body(const Color *c)
{
  // we just grab the pointer to the color provided.
  color = c;
  our_color = 0;   // we did not make the color
class.
}
-----
body.C: In method 'body::body(const Color *)':
body.C:16: warning: assignment to 'Color *' from
'const Color *' discards const
body.C: In method 'void body::setColor(const class
Color *)':
body.C:32: warning: assignment to 'Color *' from
'const Color *' discards const
```

Why is that?

- We say that we will not modify the `Color` object which we get (and we don't in either call)
- However, we retain a pointer to it and can at least potentially modify it
- Worse, we will hand out a pointer to the `Color` object to anybody who asks for it and has no obligation to honor the "const" contract.

The solution is to make a copy of the Color object which no one else gets to see.

An ugly (but working) solution...

```
// the constructor setting a color
body::body(const Color *c)
{
  // we just grab the pointer to the color provided.
  color = new Color (c->red(), c->green(), c->blue());
  our_color = 1;   // we made the color class.
}
```

We just "copy" the color by making a new object with the same properties as the color passed as a parameter. So we do not retain a pointer to the outer Color object (which the compiler did not like because it violated the const contract).

However, this exposes our constructor unnecessarily to the innards of "Color", because a better Color object might not have those member functions.

In a few moments you will learn about a better way to just copy one object to another.

But we have to learn a few new things first...

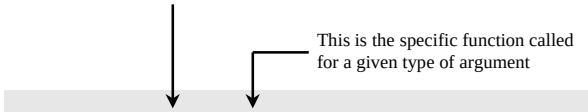
Overloaded Functions...

You may have used this without much thought:

```
      program cosine

      real x
      real*8 y

      x = 0.5
      y = 0.5

      print *, x, cos(x)
      print *, y, cos(y)

      end
------
 .5 .8775826
 .5 .8775825618903728
```

You call `cos` once with a `real` and once with a `real*8` argument. The compiler sees what you want and calls the right routine.

This is an overloaded function (in Fortran!)



Args	Generic	Specific	Argument	Result Type
1	COS	COS	REAL*4	REAL*4
		DCOS	REAL*8	REAL*8
		QCOS	REAL*16	REAL*16
		CCOS	COMPLEX*8	COMPLEX*8
		CDCOS	COMPLEX*16	COMPLEX*16
		ZCOS	COMPLEX*16	COMPLEX*16

Unfortunately, this works with the few Fortran intrinsic functions only.

...and overloaded operators

```
      program plus
      integer i,j
      real x,y
      i=3
      j=4
      x=3.
      y=4.

      print*, i + j
      print*, x + y

      end
```

Here you expect the " + " operator to add the 2 numbers, but of course "adding two integers" is quite different from "adding two real numbers".

This is a trivial overloaded operator.

Wouldn't it be nice to define some of the following:

- Matrix + Matrix, and int + Matrix
- color + color
- vector + vector

and also

- Matrix * Matrix, and int * Matrix, as well as Matrix * int
- vector * vector

however

- color * color might not be a good metric - but who knows.

In C++, you can actually do that!

Let's do overloaded functions first...

Remember this?

```
call hfill(10,x,0,1.)
call hfill(20,x,y,1.)
call hfill(10,x,0,1)
```

It's a bit annoying:
- you always need 4 parameters
- the parameters have to be of the right type

Wouldn't it be nice if
- hfill(10,x) meant "hfill(10,x,0,1.)"
- hfill(10,x,y) meant "hfill(10,x,0,y)" for one-dim histograms
- hfill(10,x,y) meant "hfill(10,x,y,1.)" for two-dim histograms

but since there is only one "hfill" function, you cannot do that in Fortran or C.

```
TH1F *h1 = new TH1F("Name","Title",128,0.,400.);
TH2F *h2 = new TH2F("Name","Title",16,0.,40.,16,0.,40.);

h1->Fill(x);
h1->Fill(x,weight);

h2->Fill(x,y);
h2->Fill(x,y,weight);
```

In C++, you can! These are ROOT TH1F and TH2F histogram, with overloaded fill methods

A fictitious example

We have seen: as long as the function name and the parameter list and the return type for a function is unique, the C++ compiler can distinguish between the functions.

All the declarations on the left are ok -- all the "function" routines can be distinguished by their parameter list and return type.

```
class example {
public:
virtual void function (const int);
virtual void function (const int, const float);
virtual void function (const int, const *float);
virtual void function (const int, const double);
virtual void function (const int, const *double);

virtual void function (const float);
virtual void function (const float, const float);
virtual void function (const float, const *float);
virtual void function (const float, const double);
virtual void function (const float, const *double);

virtual int function (const int);
virtual int function (const int, const float);
virtual int function (const int, const *float);
virtual int function (const int, const double);
virtual int function (const int, const *double);

virtual int function (const float);
virtual int function (const float, const float);
virtual int function (const float, const *float);
virtual int function (const float, const double);
virtual int function (const float, const *double);

};
```

```
example *e = new example();
int i = 0;
int j;
float x = 3.1415;
double d = 3.141592654;

e->function(i);

e->function(i,x);

e->function(i,d);

j = e->function(i);

j = e->function(i,x);

j = e->function(i,d);

j = e->function(x,&d);
```

Now we better understand this...

```
class body {
public:
  // the constructors for body
  body();
  body(Color *c);
  // the destuctor
  ~body();
//… stuff deleted
}
```

We have more than one constructor for our “body” class; this is an overloaded function like any other.

A convenient shorthand notation for a default value

```
virtual void Function(const float x);
virtual void Function(const float x, const float y);
```

We want the `y` parameter, if omitted, to default to `y=0`. For this case there is a shorthand notation:

```
#ifndef SOMECLASS_H
#define SOMECLASS_H

class SomeClass {
public:
  virtual void Function(const float x, const float y =0.);
};
#endif
```

This means:
“if this parameter is not given, set it to 0”

You can give more than one default parameter:

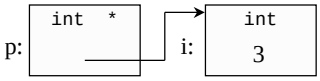
```
#ifndef SOMECLASS_H
#define SOMECLASS_H

class SomeClass {
public:
  virtual void Function(const float x = 3.1415, const float y =0.);
};
#endif
```

```
SomeClass *sc;
sc->Function();
sc->Function(3.1415);
sc->Function(3.1415,0.);
```

Next tool: Object References

If you program in C, you know what a pointer is.

```
int i =3;
int  *p;
p = &i;
```



The value of the variable `p` is the memory address of the variable `i`.
`i` is the same as `*p` (dereference pointer).
`*p` means "take the value of the memory location where `p` points to".

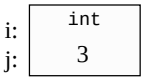
```
#include <iostream.h>
main()
{
  int i=3;
  int *p;
  p = &i;
  cout << "i = " << i  << endl;
  cout << "*p= " << *p << endl;
}
```

```
i = 3
*p= 3
```

A reference is something only available in C++

```
int i =3;
int  &j = i;
```

now you can call the variable by either name.



This is a bit like an `equivalence` statement in Fortran:

```
integer i
integer j
equivalence (i,j)
i =5
print*, j
end
```

This can be confusing. Read as "give the variable i a second name".
Object references are important because they are a very efficient way to pass objects as parameters.

Reference example

```
#include <iostream.h>
void negate ( int & j)
{
  j = -1 * j;
}
main()
{
  int i=3;
  negate(i);
  cout << "i = " << i  << endl;
}
```

```
i = -3
```

The parameter `i` gets passed on to the function as a reference, so in the `negate()` routine the variable j means the same value as the i in `main()`. We can assign values to the reference and change the variable in `main()`.

```
#include <iostream.h>
void negate ( int j)
{
  j = -1 * j;
}
main()
{
  int i=3;
  negate(i);
  cout << "i = " << i  << endl;
}
```

```
i = 3
```

If we define the function like the one left, `i` gets passed on to the function by value, that is, a copy of the variable i is made and given to `negate()`. We can assign values to the copy, but that will not change the variable in `main()`.

Because passing parameters via a reference is very efficient (no copying of values), the C++ compiler uses this technique in many places (so it's important to understand the concept).

Couldn't this be done with a pointer as well? To some extent, yes, but not all. Code using references is more powerful and more elegant.

C++ uses references in many places

… that's why we did this detour. Consider this

```
int i = 3;
int j;
j = i;
```

is conceptually the same as

```
SomeObject obj1;
Someobject obj2;
obj2 = obj1;
```

and results in a "call"

```
obj2.operator= (const SomeObject &obj1);
```

the compiler does this with a reference

Some constructs:

```
obj2 = obj1;
SomeObject obj2 = obj1;
obj3 = obj1 + obj2;
```

```
obj2.operator= (const SomeObject &obj1);
obj2 (const &obj1);    // "copy constructor"
obj3 = SomeObject.operator+ ( const &obj1, const &obj2);
```

If you want to deal with operators, you have to understand the object references.

End of detour - let's now get back to our operators.

Let's now overload some Operators

You know the operators `+ - * / < > += -= /= *=` , …
We have already seen `>>` and `<<` , which are probably the most often used ones.
You cannot define new operators, only overload existing ones… sorry, no `**` for exp()!
As we have seen, operators are a very intuitive notation for what is really a function call.

For our own classes, we can define what those operators mean.

The operator overloading is probably the most often abused feature of all. If it is not entirely obvious what "+" or "*" between two object means, do not implement it! It is your responsibility that it makes sense.

Remember our cube and cylinder classes from the previous examples? We will now define some operators for those classes. Let's start with the assignment "=". We want to support the following:

```
cube c1(1.);
cube c2(2.);
c2 = c1;
```

→

```
class cube : public body {
public:
  cube (const float);
  cube (const float, const Color *c);
  virtual char* identify() const;
  virtual float GetVolume() const;
  virtual float GetSurface() const;
  // the assignment operator
  cube& operator= ( const cube & );

protected:
  float length;
};
```

```
// the assignment operator
cube& cube::operator= (const cube& c)
{
  if ( this == &c) return (*this) ;
  length = c.length;
  return (*this);
}
```

Complicated? Let's go through it step by step.

Detour 1: Introducing the `this` pointer

Inside any member function, the this pointer points to the object whose member function this is. In this (made-up) example we need the this pointer to distinguish the private member `value` from the parameter `value`

```
class simple
{
  int value;
public:
  void setvalue(const int);
}

void simple::setvalue (const int value)
{
  this->value = value;
}
```

```
int main()
{
  simple s1;
  simple s2;

  s1.setvalue(1);

  s2.setvalue(15);

  return 0;
}
```

Inside this call, the `this` pointer used by `setvalue` refers to `s1`.

Inside this call, the `this` pointer used by `setvalue` refers to `s2`.

End of detour...

`c2=c1`: A simple "=" operation?

First, keep in mind that you are calling a member function of the cube "c2" with "c1" as the argument.

```
// the assignment operator
cube& cube::operator= (const cube& c)
{
  if ( this == &c) return (*this) ;
    length = c.length;
    return (*this);
}
```

So "length" refers to c1.length, the length of cube c1

c.length refers to the length value of cube c2.

Here we assign the length value of `c2` to `c1`, as it should be.

We return a reference to `c1` (the `this` pointer points to the object whose member function this is).

But before we check if the objects happen to be the same, that is, we write something like `c1 = c1`;

Why are we returning a reference to "us" (the `this` pointer)? After we assign the length, we are done, aren't we?

Yes. But we must support constructs like

```
c3 = c2 = c1;
```

which means "assign c1 to c2. Then assign c2 to c3", or

```
c3 = (c2 = c1);
```

Here's what happens:

- we invoke a member function of `c2` with the parameter "reference to `c1`".
- we assign `c1`'s length to `c2`'s length, and return a reference to `c2`.
- then we invoke a member function of c3 with what's right of the assignment (=), that is, the reference to `c2` returned from the function.
- we assign the length value of `c2` (through the reference) to `c3`'s length.

Note: We haven't taken care of the Color yet! Wait.

Taking care of the Color

As it happens (because we took a ready-made, thoroughly designed class), the Color class already supports the assignment (=) operator. This makes it easy to take care of the Color in the "operator=" member function:

```
// the assignment operator
cube& cube::operator= (const cube& c)
{
  if ( this == &c) return (*this) ;
  length = c.length;
  *color = *c.color;
  return (*this);
}
```

Beware: `*color = *c.color` is different from `color = c.color` !

Here we invoke the `operator=` member function function of the Color object. (We make an actual copy of the object).

here we just copy the pointer, something we don't want.

"cube c2 = c1" is different from "c2 = c1"!

The effect is the same, but what happens is quite different:

```
cube c1(1.);
cube c2(2.);
c2 = c1;
```

≠

```
cube c1(1.);
cube c2 = c1;
```

On the left, we saw that we are calling the `operator=` member function of object `c2`.

On the right, we construct the object c2 using c1. We are calling a special constructor, the so-called "copy constructor", of c2, which gets a reference to the object to be copied (c1 in this case).

```
class cube : public body {
public:
  cube (const float);
  cube (const float, const Color *c);
  cube (const cube &);
  virtual char* identify() const;
  virtual float GetVolume() const;
  virtual float GetSurface() const;
  // the assignment operator
  cube& operator= ( const cube & );

protected:
  float length;
};
```

```
// the copy constructor
cube::cube (const cube& c)
{
 length = c.length;
 *color = *c.color;
}
```

Different from the operator member functions, the compiler will automatically generate a copy constructor whether you want it or not, and this is almost never what you really want.
The best is to always supply a copy constructor just to be safe.

Why not specifying the copy constructor is usually bad

The compiler has only limited knowledge of what you really want to do.
All it can do is to make a member-wise copy of all the data members in your class.
If you have a pointer, it will copy the pointer -- but then your new object has a pointer pointing to somewhere outside the object's scope.

Instead of this

```
// the copy constructor
cube::cube (const cube& c)
{
 length = c.length;
 *color = *c.color;
}
```

you will get this

```
// the copy constructor
cube::cube (const cube& c)
{
 length = c.length;
 color = c.color;
}
```

and it can lead to a lot of problems.

Once more: *Always provide a copy constructor*.

Now we can finally come back to the colored body...

Recall that we had problems with the const keywords because we were violating the contract. All the problems came from the pointer to the "color" object, which we can eliminate now.
We also implement the copy constructor, the assignment operator, all const keywords, and are basically set.

```
#ifndef __BODY_H__
#define __BODY_H__

#include "Color.h"

class body {
public:
  // the constructors for body
  body();
  body(const Color &c);

  // the destuctor
  ~body();

  // the member functions we already know
  virtual float GetVolume() const =0;
  virtual float GetSurface() const =0;
  // two more functions which deal with the color
  virtual void setColor (const Color &c);
  virtual Color * getColor() const;

protected:
  Color color;
};
```

We now pass the Color object as a reference and no longer as a pointer.
By way of the assignment operator, we make a copy of the color and are no longer in danger of violating the contract (we cannot inadvertently modify the constant color object we got).

```
// the constructor setting a color
body::body(const Color &c)
{
  // we assign our color the color provided.
  color = c;
}

// and the destructor
body::~body()
{
}

// now the color-handling
void body::setColor (const Color &c)
{
  color = c;
}
```

…and cubes and cylinders

```
#include "body.h"

class cube : public body {

public:
  cube (const float);
  cube (const float, const Color &c);
  cube (const cube &);
  virtual float GetVolume() const;
  virtual float GetSurface() const;
  // the assignment operator
  cube& operator= ( const cube & );

protected:
  float length;

};
```

```
#include "body.h"
#define PI 3.141592654

class cylinder : public body {

public:
  cylinder (const float,const float);
  cylinder (const float,const float,const Color &c);
  cylinder (const cylinder &);
  virtual float GetVolume() const;
  virtual float GetSurface() const;
  // the assignment operator
  cylinder& operator= ( const cylinder & );


protected:
  float radius;
  float height;
};
```

Take a look at the code and see how it is done. It is in the “body_prefinal” directory. All is there, but we have one more little thing to take care of... Stay tuned for the “final” body class.

Let’s summarize what we have learned so far

- We know when and how to use inheritance, and when and how to use associations;
- We know when and how to provide operators for our classes;
- We know about pointers and references;
- We know about overloaded functions;
- We know what a constructor is, that there can be overloaded constructors just like with any other member function;
- We know about the copy constructor, and that we should always provide one;
- We have seen a destructor, and what it’s good for (although the important stuff is still to come).

Next we will have a close look at the “cout <<“ operators and then show a few real-life things in actual use in PHENIX.

`cout <<` -- an operator not like the others.

And you thought you would understand the operator business. After all, we did implement the assignment (`operator=`) -- what is different if we write `cout << c1 << endl;` ??

```
cube c1(1.);
cube c2(2.);
c2 = c1;
```

Remember that with the construct on the left, we are calling a member function of `c2` with a "reference to object `c1`" as its parameter.

```
cube c1(1.);
cout << c1 << endl;
```

In the same way, we would call a member function of the object `cout` with a parameter of "reference to object `c1`", which is of class "cube".
Wait a minute -- does `cout` have a member function "`operator<< (const cube &)`"?
No, of course not. We made the cube class, there is no way how `cout` could know about our cube class.
Also, this would not be what we want. We want the knowledge how to output our cube class to live with the cube class, not with cout. What do we do?

```
cube c1(1.);
c1 << cout;
endl << cout;
```

?

Ok, now we are actually calling the cube member function with a parameter of type "reference to the object cout". But this doesn't look right to the eye.
And we would have to write `endl << cout` in a separate statement. Not good.

The solution is to have a friend.

What is a `friend`?

We didn't pay much attention to it yet, but we already saw the "public" and "protected" keywords in our class definitions. There is also the "private" keyword.

```
#include "body.h"

class cube : public body {

public:
  cube (const float);
  cube (const float, const Color &c);
  cube (const cube &);
  virtual float GetVolume() const;
  virtual float GetSurface() const;
  // the assignment operator
  cube& operator= ( const cube & );

protected:
  float length;

private:
  int private_member; // just to get the idea

};
```

All the green stuff following the "public" keyword is accessible for everybody.

All the red stuff following the "protected" keyword is accessible to member functions of this class, and to member functions of all derived classes.

All private members are accessible only to member functions of this class, and not even to member functions of derived classes (they have to go through public or protected member functions of this class to access the private members).

With the friend keyword, you can grant other classes or functions limited access to protected or private members or functions. ***"Only friends can get at your private parts."***

Declaring who our friend is

```
class body {

friend class antibody;

friend void massage (body &);

public:

 // stuff deleted
};
```

Here we declare the (fictitious) class “antibody” to be a friend of our class “body”, so any member function of the antibody class has access to our private and protected stuff.

Also, the (again fictitious) non-member function “massage” can access all of our body class. A non-member function is not a member function of any class (just as any function in F77 or C used to be).

```
#include "body.h"

class cube : public body {

friend ostream& operator<< (ostream&, cube &);

public:
  cube (const float);
  cube (const float, const Color &c);
  cube (const cube &);

  // stuff deleted...
};
```

Here we define the “non-member operator” `operator<<` (which is a function just like the “massage” from above) to be a friend of our class cube. This function has now access to all of our protected and private stuff. By the way, the cout object is of class “ostream”.

The `ostream<<` non-member function

```
cube c1(1.);
cout << c1 << endl;
```

Our non-member function operator<< gets called here.

Note that we do not write “`cube::operator<<`“ here; then it would be a member function, which we don’t want.

```
ostream& operator<< (ostream& os, cube &c)
{
  os << "cube of length " << c.length;
  return os;
}
```

We can do this only because this function is a friend of cube; otherwise we wouldn’t get access to the private member “length”.

We return the “ostream” after we are done with it because we want to be able to “chain” the << operators (cout << c1 << c2 << endl;).

We may want to also say something about the color of our cube. As it happens (because it’s a well-designed class), the Color class already defines the << operator for itself. However, the color attribute is not with the cube class but with the “body” parent and so we need to go through publicly accessible member functions to obtain a copy of the color object.

```
ostream& operator<< (ostream& os, cube &c)
{
  Color *cl = c.getColor();
  os << "cube of length " << c.length
     << " " << *cl;
  delete cl;
  return os;
}
```

In the end, it's overloading a function...

Apart from the "friend" business, it's just another overloaded "operator<<" function:

```
ostream& operator<< (ostream& os, int &);
ostream& operator<< (ostream& os, float &);
ostream& operator<< (ostream& os, double &);
ostream& operator<< (ostream& os, char &);
ostream& operator<< (ostream& os, long &);

//… all the system-defined ones …

ostream& operator<< (ostream& os, cube &c);
```

We add one more, just as we saw before (ok, so it's a non-member function, it's not a function but an operator, but the principle holds).

and this is how it looks like:

```
#include "cube.h"
#include "cylinder.h"
#include "iostream.h"
#include "Color.h"

main()
{
  Color *yellow = new Color(  1.0,  1.0,  0.0 ); // yellow
  Color *green   = new Color(  0.0,  1.0,  0.0 ); // green

  cylinder *c1 = new cylinder (1.,2.);        // default-colored cylinder
  cylinder *c2 = new cylinder (1.,2.,*green);  // green cylinder
  cube     *c3 = new cube (1.,*yellow); // yellow cube
  cout << *c1 << endl;
  cout << *c2 << endl;
  cout << *c3 << endl;
}
```

```
$ main
cylinder of radius 1 height 2 Color ( 0, 0, 0 )
cylinder of radius 1 height 2 Color ( 0, 1, 0 )
cube of length 1 Color ( 1, 1, 0 )
$
```

Close, very close, but not yet quite ok. It is ok in the above example because c1,c2,and c3 are pointers to cube and cylinder, respectively. Were they pointers to the parent class "body", it would not work. We will of course define a operator<< for body, but this function wouldn't know whether to access cube's or a cylinder's "private parts" (even if it were allowed to, which it isn't). We will need to do this through member functions again...

The real final, ultimate body class

We make “traditional” member functions `identify(ostream&)` for our classes (which have nothing to do with the << operators)

```
void cube::identify(ostream &os)
{
  os << "cube of length " << length
     << " " << color;
}
```

```
void cylinder::identify(ostream &os)
{
    os << "cylinder of radius " << radius
     << " height " << height
     << " " << color;
}
```

and we let the non-member operator<< functions call the (public) member functions; no problem here:

```
ostream& operator<< (ostream& os, cube &c)
{
  c.identify(os);
  return os;
}
```

```
ostream& operator<< (ostream& os, cylinder &c)
{
  c.identify(os);
  return os;
}
```

but now also body can easily have a operator<< non-member function which calls the proper member function and so gives the proper output.

```
ostream& operator<< (ostream& os, body &b)
{
  b.identify(os);
  return os;
}
```

Now we see it at work...

Here we had ‘cylinder” and “cube” before →

```
#include "cube.h"
#include "cylinder.h"
#include "iostream.h"
#include "Color.h"

main()
{
  Color *yellow = new Color(  1.0,  1.0,  0.0 ); // yellow
  Color *green   = new Color(  0.0,  1.0,  0.0 ); // green

  body *c1 = new cylinder (1.,2.);        // default-colored cylinder
  body *c2 = new cylinder (1.,2.,*green);  // green cylinder
  body *c3 = new cube (1.,*yellow); // yellow cube
  cout << *c1 << endl;
  cout << *c2 << endl;
  cout << *c3 << endl;
}
```

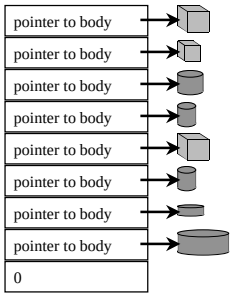
```
$ main
cylinder of radius 1 height 2 Color ( 0, 0, 0 )
cylinder of radius 1 height 2 Color ( 0, 1, 0 )
cube of length 1 Color ( 1, 1, 0 )
$
```

That’s it! The fully functional, fully by-the-rules body class and its children. Find the code in the `9_body_final/` directory.

Some templates, some STL

We have already mentioned the reusability of well-designed, well-implemented C++ code. There is a mechanism which we can only touch (this would be a course by itself), the *templates*.

We have seen this...



- So we have something which sort of manages a collection of “Bodies” for us.
- The next day someone asks if you can make the class deal with “Shapes”. You fire up emacs, replace all “Body” with “Shape”, and you are set.
- The day after someone asks for a vector of “Colors”. You fire up emacs, replace all “Shape” with “Color”, and so on.
- The other week someone asks you to support his own class for “complex” values. At that point you might think that there is a better way of doing that, and indeed there is.

Simple templated class (not for actual use)

```
template<class T> class sList {
public:
  sList( const int n );
  ~sList();
  virtual int addEntry(T*);
  inline int getNumberofSlots() const
       { return number; };
  inline int getNumberofEntries() const
       { return current+1; };
  inline T* getEntry( const int n) const;
protected:
  T** list;
  int number;
  int current;
};

template<class T> inline sList<T>::sList( const int n)
{
  list = new T*[n];
  number = n;
  current = -1;
}
template<class T> inline sList<T>::~sList()
{
  delete [] list;
}
// stuff deleted
//
};
```

```
#include <iostream.h>
#include "sList.h"
int main()
{
  int i = 37;
  int j = 41;
  int k = 49;
  float x = 3.14;
  float y=2.718;

  sList<int> intList(15);
  sList<float> floatList(12);

  intList.addEntry(&i);
  intList.addEntry(&j);
  intList.addEntry(&k);

  floatList.addEntry(&x);
  floatList.addEntry(&y);

// stuff deleted

}
```

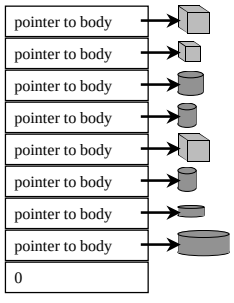
STL (Standard Template Library)

Templates provide a way to make a generic implementation which would deal with any class (double, int, float, and also other user-written classes).

Writing templates is not easy if they are supposed to be general enough. Luckily, some smart people have collected the most wanted and most useful templates and bundled them into what's known as the **Standard Template Library**.

One caveat: STL is a "standard", but with the implementation, your mileage may vary. In a few months, hopefully all compilers will be smart enough to handle all of it.

Just a STL teaser...



This is the most simple and least sophisticated implementation of what's known as "container classes". Those are classes which manage (contain) other objects and let you manipulate groups of them.

What this thing does not do:

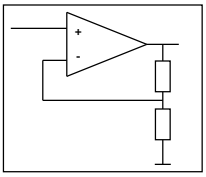
- you cannot easily insert other objects in the middle, or delete objects from it.
- you have basically one way of accessing the objects inside in a sequential fashion
- you don't have control over the order in which you retrieve the objects.
- It is very inefficient if you would want to sort the objects.
- It is also very inefficient if you want to search for a particular object.

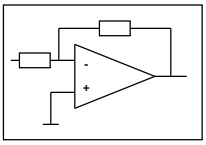
In the STL, there is a rich variety of well-designed container template classes such as "linked list" , "vector", "associative containers", and many more; you can go and look for the one which best suits your needs.

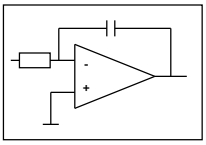
Let's move on to design patterns

Who knows what those guys are?
And where do you know it from?

Answer: These are *standard designs* to be found in any electronics book. You look them up and use them in your application.







The advantages to use those ready-made designs:

- they usually work;
- they save a lot of time;
- you don't need to develop and design those lower-level building blocks any more; but can use them directly to build a larger product;
- since they are well known, it's easy to gauge the relative strengths, weaknesses, and applicability of the designs;
- standard names make it easy to unambiguously communicate with your colleagues.

By using standard design patterns you gain access to a extremely valuable knowledge base.

What are those "design patterns"?

They are standard designs developed by good designers over time. They are not code libraries; you have to write your code according to the design. It can give you lots of ideas how to best attack a particular problem.
Most designs have standardized names, such as
factory
singleton
decorator
composite object
<many more...>
If you talk about a "factory", everybody will know what you are talking about, much like "give me the hammer out of the toolbox" will give you the desired result.

We will look at two of them and briefly go over some others.

Some design patterns (mentioned, not explained)

Decorator

we have seen that we can give objects attributes through associations (as we did with the color). Your object may have a fixed number of attributes, such as color, flavor, material, texture, and so on. For each attribute there is a place where it is stored.

But what if there is a variable number of attributes, which in a way augment each other? That's where a Decorator pattern comes in. It allows to deal with a variable number of attributes which add their behavior, and also with attributes which were not yet known when the class was designed.

Example: Text with attributes

plain text **bold text** ***bold+italic text***
underlined text **bold underlined** ***bold underlined and italic***
shadow underlined text **bold underlined** ***bold underlined and italic***

This is a good example where a decorator is useful, because we might have attributes which we did not think of before (such as a "small caps" attribute).

Composite Objects

They come in handy if a composition of single objects is conceptually of the same class again. You can then build hierarchies of collections which are very easy to deal with.

Anti-example: a group of persons is not again a person (but a group) and has different properties.

But if you glue several of our bodies (cubes, cylinders, etc) together, you will end up with a new body (albeit a more complex one). The new entity is a body in its own right, and has all the required properties (volume, surface, color, etc).

The "Composite objects" design gives you a way of dealing with the collection of objects in the same way as with a single object.

If you read the Objectivity manuals, you will see a long treatment of Composite Objects there.

A Factory

Probably the most frequently used pattern. In your real life, you use it very often. Imagine that you are buying a PC for your office. Usually you do not shop around yourself, but there is a purchase office which takes care of bids, best prices, and also takes care that all PC's bought are compatible with each other.

But in software, you have seen this…

```
int calibrate()
{
int first =1;
float x1,y1,x2,y2,x3,y3;
if (first){
    first = 0;
    hbook1_ (101,"calibration factor 1", 128, 0., 200., 0.);
    hbook1_ (102,"calibration factor 2", 128, 0., 200., 0.);
    hbook1_ (103,"calibration factor 3", 128, 0., 200., 0.);
    }
// code deleted
hfill_ (101, x1,y1,1.);
hfill_ (102, x2,y2,1.);
hfill_ (103, x3,y3,1.);
}
```

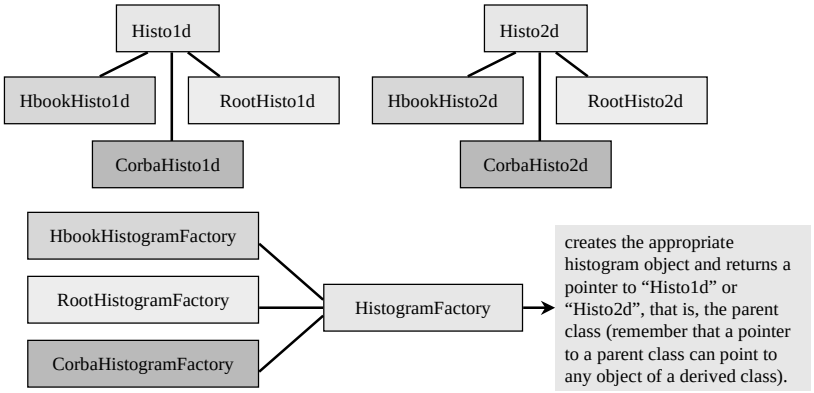
What you do here is that you make it explicit that you are dealing with Hbook histograms. If you would want to change the type of histogramming package to, say, Root histograms, you would need to change all your code and implement the new calls for Root histograms.

How can we avoid changing a lot of code?

- The need for source code changes is bad. Even having to recompile an existing object library is bad. The need for it should be avoided at all cost (how often did you have problems when someone rebuilt the libraries and some things changed?).
- In our little functions and routines we use only a tiny bit of the functionality of the histograms. Usually we just want to create them, fill them, maybe reset them, and store them. This is where a factory comes in.
- We will not directly book the histograms, but we ask the factory to please make a histogram object with the desired properties for us (the "purchase department"). We will get a histogram object which we can fill, reset, store, and so on.
- However, our routines are not aware of the fact that they happen to deal with Hbook histograms today, Root histograms tomorrow, Corba histograms on a server filled by many distributed clients, and who-knows-what in 5 years. The code libraries will happily work with any histogram implementation *without recompilation*.
- The decision which histograms to use is made outside our functions; there is only one entity in one place (such as the main program) which decides this, and no need to track down all the occurrences of histograms.
- To be very clear: This is only for the "fill" code in your software; if you deal with the histograms interactively, look at them, plot them, fit them, etc, you will need to be aware of the histogram implementation (whether to use PAW or ROOT, for example). But we will shield perhaps 95% of our code from changes of the histogramming package.

Abstract Histograms

We design an abstract “Histo1d” and “Histo2d” class, where the “real” histograms inherit from. The parent provides the interfaces “Fill”, “Reset”, and so on.



Remember that our code will be able to deal with other histogram classes which did not exist at the time when our code was compiled (as long as they inherit from “Histo1d” or “Histo2d”).

Phenix users beware: This here is the predecessor of the PhHistogramFactory; many things have changed (no Hbook, for example) .

Do Hbook and Root histos have a common parent class?

Of course not. I made the classes Histo1d, 2d, and so on, which are the interface to the real histogram (in case of Hbook, it is not even really an object). So they inherit from a common parent, and they delegate our actions to the real histogram.
This is association at work.

```
class RootHisto1d : public Histo1d {
public:
  RootHisto1d(const char* ,  const int ,  const float,  const float);
  RootHisto1d(const int id , const char* , const int , const float, const float);
  ~RootHisto1d();
  virtual void Fill( const float,  const float weight =1.);
  virtual void Reset();
  virtual void print() const;
  virtual TH1F * getHistoObject() {return h1d;} ;

protected:
  TH1F *h1d;
};
```

Our RootHisto1d object has a TH1F object associated and will delegate the actions to it.

```
RootHisto1d::RootHisto1d(const char* title , const int bins, const float xl, const float xh)
{
  h1d = new TH1F ( title, title, bins, xl, xh);
}
void RootHisto1d::Fill (const float x, const float weight)
{
  h1d->Fill (x,weight);
}
```

Visit our Factories

HbookHistogramFactory

```
Histo1d*  create1dHistogram (const int id, const char * title,
     const int bins, const float xl, const float xh)
 { return new HbookHisto1d(id, title, bins,xl,xh); }

Histo2d*  create2dHistogram (const int id, const char *title,
          const int xbins, const float xl, const float xh,
          const int ybins, const float yl, const float yh)
 { return new HbookHisto2d(id, title, xbins,xl,xh,ybins, yl, yh ); }
```

RootHistogramFactory

```
Histo1d*  create1dHistogram (const int id, const char * title,
        const int bins, const float xl, const float xh)
 { return new RootHisto1d(id,title, bins,xl,xh); }

Histo2d*  create2dHistogram (const int id, const char *title,
     const int xbins, const float xl, const float xh,
     const int ybins, const float yl, const float yh)
 { return new RootHisto2d(id,title, xbins,xl,xh,ybins, yl, yh ); }
```

Those member functions of Root /HbookHistogramFactories override the respective virtual functions in the “HistogramFactory” parent class.

usercode.cc / usercode.o

usercode.cc, the file representing user code sitting in object libraries. It is compiled once, and the same usercode.o file is used with Hbook and with Root histograms, without recompilation.

```
#include <HistogramFactory.h>

HistogramFactory * getHF();

int usercode()
{
  HistogramFactory *hf = getHF();
  Histo1d *hpx = hf->create1dHistogram(100,
                    "Simple Distribution"
                    ,100,-0.5,99.5);
  for ( int i=0; i<100; i++)
    {
      hpx->Fill(i,i);
    }
  hf->save_with_extension("file");
  return 0;
}
```

The call to a “higher entity” to get hold of the factory. This routine is provided by the main program which so decides whether to use Hbook or Root histograms.

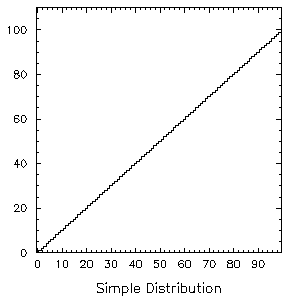
Here we ask the factory object to please make us a 1d-histogram with the desired properties. “hpx” now points to the Histogram object.

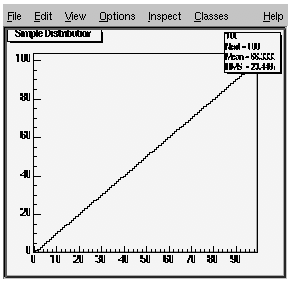
We now work with the histogram; here we fill it 100 times.

In the end we save the histogram to a file. The “save_with_extension” call adds an appropriate extension to the file name, “.root” in case of Root histograms, “.hst” in case of Hbook histograms.

Please convince yourself that this function works with [both currently implemented] kinds of histograms without the need for recompilation.

hbook and root histograms





Now, the same compiled usercode.o (representing the compiled object libraries here) will once produce an Hbook and once a Root histogram.

```
#include <HbookHistogramFactory.h>
HbookHistogramFactory *hbf =
                HbookHistogramFactory::buildFactory();
HistogramFactory * getHF(){ return hbf;}
int usercode();
int main()
{
 usercode();
}
```

```
#include <RootHistogramFactory.h>
RootHistogramFactory *rbf =
                RootHistogramFactory::buildFactory();
HistogramFactory * getHF() { return rbf;}
int usercode();
int main()
{
 usercode();
}
```

A note on the abstract histograms

You have seen this class definition

```
class RootHisto1d : public Histo1d {
public:
  RootHisto1d(const char* ,  const int ,  const float,  const float);
  RootHisto1d(const int id , const char* , const int , const float, const float);
  ~RootHisto1d();
  virtual void Fill( const float,  const float weight =1.);
  virtual void Reset();
  virtual void print() const;
  virtual TH1F * getHistoObject() {return h1d;} ;

protected:
  TH1F *h1d;
};
```

If you need functionality from a histogram which is not provided through the abstract interface (fitting the histogram is the best example), you can ask the abstract histogram object for a pointer to its real internal histogram object and do your work.

In some parts of your code, you may want to discard the abstract interface in this way.

Still, you can protect maybe 95% or the code from changes in the histogram implementation.

More good stuff: a Singleton

Somehow there should be one and only one factory. If we allow two different factories to coexist, we are in trouble. We want to make it so that the first instance of a HistogramFactory object prevents all other instances from being created. This is called a ***Singleton***.
Let's look at a part of the RootHistogramFactory class declaration:

```
class RootHistogramFactory : public HistogramFactory {
public:

// stuff deleted...

  static RootHistogramFactory *buildFactory();
  static int deleteFactory();

protected:
  RootHistogramFactory::RootHistogramFactory();
  RootHistogramFactory::~RootHistogramFactory();

private:
  static RootHistogramFactory *single;
  static int access_count;
};
```

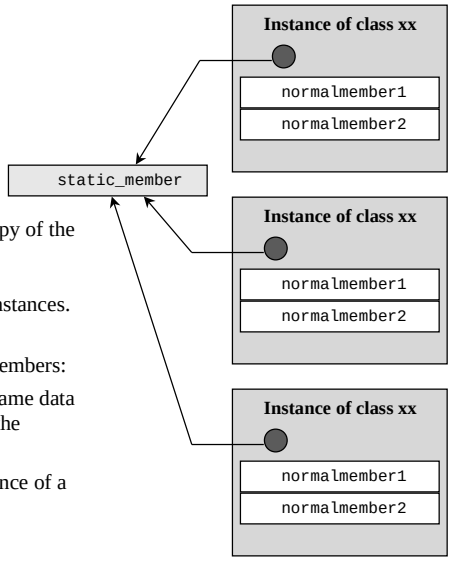
A protected constructor?!?

and a new keyword...

One piece at a time...

static data members and member functions

```
class xx {
public:
// stuff deleted...
private:
  static int static_member;
  int normalmember1;
  int normalmember2;
};
```



Each instance of the class xx has its own copy of the `normalmember1` and `normalmember2` variable, but there is only one copy of the `static_member` variable shared by all instances.

Two important applications of static data members:

- because all instances of a class "see" the same data member, you can use it to globally modify the behavior of all instances of the class;
- a static variable can be set even if no instance of a class exists yet.

A silly example...

```
#include <iostream.h>
class greeting {
public:
  greeting();
  void greet_me();
  static int language;
};
```

```
#include "greeting.h"
int greeting::language = 0;
greeting::greeting(){}
void greeting::greet_me()
{
  switch (language)
    {
    case 1:  //german
      cout << "Guten Morgen, Martin!" << endl;
      break;
    case 2:  //francais
      cout << "Bonjour, Martin!" << endl;
      break;
    default:
      cout << "Good morning, Martin" << endl;
    }
}
```

```
#include "greeting.h"

int main()
{
  greeting g1,g2,g3;
  g1.greet_me();
  g2.greet_me();
  g3.greet_me();

  greeting::language = 1;
  g1.greet_me();
  g2.greet_me();
  g3.greet_me();

  greeting::language = 2;
  g1.greet_me();
  g2.greet_me();
  g3.greet_me();
}
```



```
$ greeting
Good morning, Martin
Good morning, Martin
Good morning, Martin
Guten Morgen, Martin!
Guten Morgen, Martin!
Guten Morgen, Martin!
Bonjour, Martin!
Bonjour, Martin!
Bonjour, Martin!
```

In this way we can globally modify the behavior of all instances of class “greeting”.

Now lets see our RootHistogramFactory Singleton

```
class RootHistogramFactory : public HistogramFactory {
public:

// stuff deleted...

  static RootHistogramFactory *buildFactory();
  static int deleteFactory();

protected:
  RootHistogramFactory::RootHistogramFactory();
  RootHistogramFactory::~RootHistogramFactory();

private:
  static RootHistogramFactory *single;
  static int access_count;
};
```

- Nobody can construct an instance of the object because the constructor is protected.
- The only way to get an instance is the static “buildFactory” member function (this function can call the protected constructor)
- it will create at most one instance and subsequent requests will receive a pointer to the only existing instance.
- We keep track of how may copies of the pointer were given out
- Only if the access count is 0 we allow deleting the object again (not shown).

```
RootHistogramFactory* RootHistogramFactory::buildFactory()
{
  if (single == 0) single = new RootHistogramFactory();
  access_count++;
  return single;
}
```

Let's review what we did with the designs

We made an abstract class HistogramFactory which are subclassed by a HbookHistogramFactory and a RootHistogramFactory. One will create Hbook histogram objects, the other one Root histogram objects. The objects returned have a common parent, and our code will deal with the parent class only, that is, with any implementation of histograms which inherit from the parent, without recompilation.

We made each of the factories a singleton. It will not be possible to have more than one instance of a given object. In this way we prevent conflicts which we would have if there were more than one instance of a factory.

Design patterns can give you a lot of help in designing your object model and save a lot of time.

On to some analysis examples

This is meant to be a C++ course, not a tutorial how to use any particular Phenix analysis framework. I want show that we can achieve a lot of flexibility in our code by using a well-designed object model. We saw that we achieved polymorphism through inheritance and associations, and that we used two design patterns (the factory and the singleton pattern).

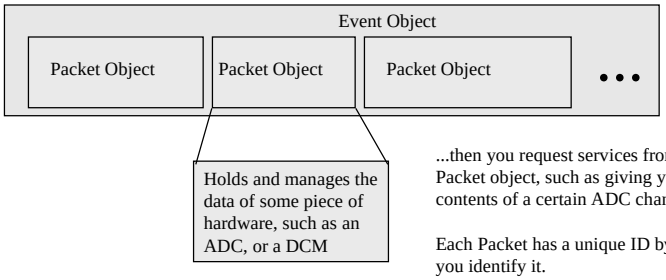
The point that I want to drive home is the following: The core designers and programmers still have a lot of work making all the class libraries (it is ***not*** more difficult than in the Fortran days, though). But for the end user of the class libraries the job becomes a lot easier because of the "service provider" concept we introduced in the beginning. Where you were exposed to a lot of the "mechanics" of the software before, you can now just request services and a lot of the internal stuff is hidden from you. You can better concentrate on the problem at hand.

Now it comes down to actually see what services can be provided by a certain object, rather than understanding what in which order has to be called in order to accomplish a given task.

Event objects

At the core of the analysis chain is the Event object, which is the manager of a given event's raw data. It provides services which let you access chunks of data, see the properties of the event, copy/move/access the event raw data, and so on.

The Event object manages a collection of Packet objects, which in turn let you access single channels of your detector. This hierarchy is necessary because of the hardware which delivers data in such chunks (a FEM or DCM worth).

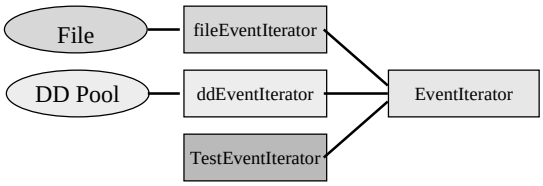


...then you request services from the Packet object, such as giving you the contents of a certain ADC channel.

Each Packet has a unique ID by which you identify it.

Getting an Event object

An Iterator gives you one Event object at a time, until it becomes empty.



There are different Eventiterators, one reading from a file, another from an online data distributor (DD) pool, another just generates fake Events (with known properties) on the fly so you can test certain aspects of your code with them.

The Coke-Can Iterator in the Physics Department

The Iterator paradigm: Hand over the next item until the iterator is exhausted.

Here: It will give you the next can of soft drink until it is empty.

Trivial PHENIX analysis example

I used a utility which creates you a data file with 100 or so Events. Each event holds exactly (only) one packet with Id 1001.

```
# include <fileEventiterator.h>

int main(int argc, char *argv[])
{
  if (argc < 2)
    {
       cout << "Usage: " << argv[0] << " filename" << endl;
       exit (1);
    }
  int status;

  Eventiterator * it =
   new fileEventiterator (argv[1], status);
  if (status)
    {
      cout << "Couldn't open input file " << argv[1] << endl;
      delete it;
      exit(1);
    }
  Event *evt;
  while ( evt = it->getNextEvent() )
    {
      evt->identify();
      delete evt;
    }
  delete it;
  return 0;
}
```

And that's a scan utility which browses through the files and displays the events in there.
The highlighted lines are the essential ones which get you event objects from the file. You are not exposed to all the mechanics of that process, you just use services provided by objects.

And that's the output...

```
$ ./fileread file.evt | more
 -- Event     1 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event     2 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event     3 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event     4 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event     5 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event     6 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event     7 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event     8 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event     9 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
 -- Event    10 Run:   198 length:   222 frames:   1 type:  1 (Data Event)
```

reading from 2 or more files

something which takes dyed-in-the-wool F77 programmers some getting-used to. You may be used to analysis code which opens THE data file, then opens the next one, and so on, typically on a certain F77 logical unit.
With objects, there can be as many objects as you wish...

```
#include <iostream.h>
#include <fileEventiterator.h>

int main(int argc, char *argv[])
{
  int status;
  // see if we can open the file
Eventiterator *it1 = new fileEventiterator(argv[1]);
Eventiterator *it2 = new fileEventiterator(argv[2]);
// stuff deleted

Event *evt1 = it1->getNextEvent();
Event *evt2 = it2->getNextEvent();

// and so on
```

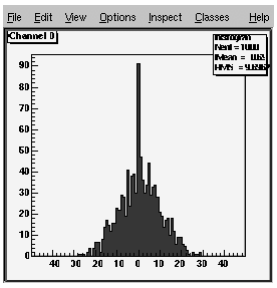
So you can make and use many Eventiterators simultaneously, and open as may files or other sources of Events at the same time. (Think of a mixed-event background analysis, for example). Also, you can open the same data file with more than one Iterator at the same time.

and now some Packets...

```
{
// load the shared library
gSystem->Load("$EVT_LIB/libEvent.so");

// make a event iterator which reads from a testEventiterator
Eventiterator *it = new testEventiterator();

// e is  pointer to an object of type "Event"
Event *e;
// now we create a histogram and set some display properties
here.
TH1F *h = new TH1F("histogram","Channel 0", 101,-49.5,49.5);
// now we read 1000 events or until the iterator is empty,
whatever comes
// first. For each event we have it identify itself and then
we delete it again.
int i = 1000;
while (i-- && (e = it->getNextEvent()) )
  {
     Packet *p = e->getPacket(1003);
     if (p)
       {
           h->Fill (p->iValue(0));
           delete p;
       }
    delete e;
  }
  h->Draw();
  delete xx;
}
```

Event objects are nice, but we want to get at the stuff inside (the Packets and their contents).
This example is actually running inside ROOT.

And that's the result.



Maybe you didn't catch it...

… but we put the polymorphism of the Eventiterator class to work here:

```
Eventiterator *it =
       new fileEventiterator (argv[1], status);
```

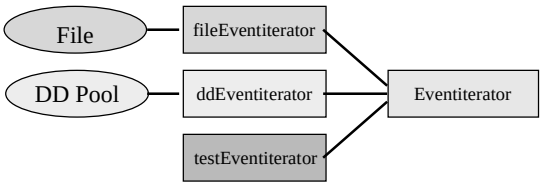
Previous example

```
Eventiterator *it = new testEventiterator();
```

Last example

We remember that a pointer to a parent class can point to an object of any of its child classes. We can also get a ddEventiterator, which reads from the DD pool.

```
Eventiterator *it = new ddEventiterator("ONLINE",status);
```



The testEventiterator is nice because you can easily get event objects with known properties for testing. The Packet 1003 gives you values with some gaussian-like distribution.

Memory leaks, the curse of dynamic memory allocation

Next to crashing programs, memory leaks are a nightmare to deal with. If a program runs 24/7, the smallest leak will eventually lead to a program or system crash because the resources are exhausted.

There are different ways to get a memory leak:

- it may be outside your control, because you use a user or system library which has a memory leak;
- it may be because the designer of a class which you use made a mistake;
- it may be because you fail to delete an object or other memory which you allocated.

The result of a failure to delete objects properly is often (only) a memory leak, and often the leak does not show because the program is a short utility, or because not many objects are instantiated. Keep in mind, however, that objects can have other side effects which will not go away when the program terminates, such as temporary files left behind, allocated shared memory, message queues, semaphores, even network sockets to other systems. Do not assume that objects perform only benign operations which are undone when the program ends. Always delete objects which you created.

A stupid, obvious user memory leak

```
#ifndef __GOODCLASS_H__
#define __GOODCLASS_H__

class good
{
public:
  good();
  virtual ~good();
protected:
  int * pointer;
};

#endif /* __GOODCLASS_H__ */
```

```
#include "goodclass.h"

good::good()
{
  pointer = new int[1024]; // 4kb memory
}
good::~good()
{
  // delete the allocated memory
  delete [] pointer;
}
```

```
#include "goodclass.h"
#include <iostream.h>

int main()
{
  good *g;
  int i = 0;

  // we leak 1000 * 4kb memory.
  while ( i++ < 1000)
    {
      cout << i << endl;
      g = new good();
    }
}
```

With each new object of class “good” created and not deleted, we leak at least 4kb of memory. Each time we overwrite the pointer with the pointer to the new object, we leave the previous object behind. There is no way to get back at the old one.

Pretty obvious, but you see it from time to time.

A class designer’s stupid mistake

```
#ifndef __BADCLASS_H__
#define __BADCLASS_H__

class bad
{
public:
  bad();
  virtual ~bad();
protected:
  int * pointer;
};

#endif /* __BADCLASS_H__ */
```

```
#include "badclass.h"

bad::bad()
{
  pointer = new int[1024]; // 4kb memory
}
bad::~bad() {} // we don’t clean up
```

```
#include "badclass.h"
#include <iostream.h>

int main()
{
  bad *b;
  int i = 0;

  // we leak 1000 * 4kb memory.
  while ( i++ < 1000)
    {
      b = new bad();
      cout << i << endl;
      delete b;
    }
}
```

Not so much we can do here -- since we usually get the object libraries only, we have no way of telling that the class programmer made the mistake.

A very common mistake

```
#ifndef __BADPARENT_H__
#define __BADPARENT_H__

class badparent
{
public:
  badparent(){};

};
#endif /* __BADPARENT_H__ */
```

```
#ifndef __CHILD_H__
#define __CHILD_H__

#include "badparent.h"

class child : public badparent {
public:
  child();
  ~child();
protected:
  int *pointer;
};
#endif /* __CHILD_H__ */
```

```
#include "child.h"
#include <iostream.h>

child::child()
{
  pointer = new int [1024];
  cout << " in child's constructor" << endl;
}
child::~child()
{
  delete [] pointer;
  cout << " in child's destructor" << endl;
};
```

"child" is doing all by the books: all memory allocated is freed again in the destructor. We print messages so that we can see the constructor and destructor being called.

```
#include "child.h"
#include <iostream.h>

int main()
{
  child *c;
  int i = 0;

  while ( i++ < 5)
    {
      c = new child();
      delete c;
    }
}
```

→

```
in child's constructor
in child's destructor
in child's constructor
in child's destructor
in child's constructor
in child's destructor
in child's constructor
in child's destructor
in child's constructor
in child's destructor
```

looks ok (and is ok) -- for each constructor called, we call the destructor, all fine.
So why is the parent "bad"?

So what's the problem?

before we had a pointer to "child" here, now we have a pointer to the parent class →

```
#include "child.h"
#include <iostream.h>

int main()
{
  badparent *c;
  int i = 0;

  while ( i++ < 5)
    {
      c = new child();
      delete c;
    }
}
```

→

```
in child's constructor
in child's constructor
in child's constructor
in child's constructor
in child's constructor
```

?

So what happened to the destructor? How can that be?

Answer: A flaw in the "badparent" class implementation -- it does not define a destructor, so the compiler, who refers to the "child" by the pointer to the parent, never knew that there is a destructor to call.

```
#ifndef __BADPARENT_H__
#define __BADPARENT_H__

class badparent
{
public:
  badparent(){};

};
#endif /* __BADPARENT_H__ */
```

Adding this line

```
virtual ~badparent(){};
```

will convert the bad into a good parent class.

A side note: When I had job interviews with several software companies in Germany, on two occasions I was given this problem to prove that I knew my C++.

Another common mistake

We have seen something like this...

```
int process_event (Event * evt)
{
  Packet *p = evt->getPacket(1001);
  if (s)
    {
      cout << setw(6) << s->iValue(0)
                << setw(6) << s->iValue(4)
                << setw(6) << s->iValue(5) << endl;
      delete s;
    }
  return 0;
}
```

No problem here.

```
int process_event (Event * evt)
{
Packet *s1 = evt->getPacket(1001);
Packet *s2 = evt->getPacket(1002);
if (s1 && s2)
    {
      cout << setw(6) << s1->iValue(0)
           << setw(6) << s2->iValue(0)
           << endl;
      delete s1;
      delete s2;
    }
  return 0;
}
```

A problem here?

Indeed. On the right, the “if” becomes true only if both the s1 and s2 objects are created. They usually are, so usually there is no problem. However, we might have a corrupt event and only one object gets created, not the other one. Then the “delete” is never reached, and as we go on, we leave the object behind. This is a “bad” memory leak. It is bad because

- Those memory leaks occur only very infrequently, and are extremely hard to recognize (not to mention to find) in the development phase of a project.
- It takes a corrupt or otherwise non-standard events to occur, and in the test phase you often work with simulated data which are less likely to show this problem;
- the problem will show up only when there is a significant amount of data to process, and that’s usually when the production has already started.

```
int process_event (Event * evt)
{
Packet *s1 = evt->getPacket(1001);
Packet *s2 = evt->getPacket(1002);
if (s1 && s2)
    {
      cout << setw(6) << s1->iValue(0)
           << setw(6) << s2->iValue(0)
           << endl;
    }
 if (s1) delete s1;
 if (s2) delete s2;
 return 0;
}
```

the solution

In which order are constructors and destructors called?

```
#ifndef __ORDER_H__
#define __ORDER_H__

class parent
{
public:
  parent();
  ~parent();
};

class child : public parent
{
public:
  child();
  ~child();
};

class grandchild : public child
{
public:
  grandchild();
  ~grandchild();
};
#endif /* __ORDER_H__ */
```

```
#include "order.h"
#include <iostream.h>
parent::parent()
{
  cout << "in parent’s constructor" << endl;
}
parent::~parent()
{
  cout << "in parent’s destructor" << endl;
}
child::child()
{
  cout << "   in child’s constructor" << endl;
}
child::~child()
{
  cout << "   in child’s destructor" << endl;
}
grandchild::grandchild()
{
  cout << "      in grandchild’s constructor" << endl;
}
grandchild::~grandchild()
{
  cout << "      in grandchild’s destructor" << endl;
}
```

```
#include "order.h"

int main()
{
  grandchild * gc = new grandchild();
  delete gc;
}
```

```
$ main
in parent’s constructor
   in child’s constructor
      in grandchild’s constructor
      in grandchild’s destructor
   in child’s destructor
in parent’s destructor
```

Answer: constructors top-down, destructors bottom-up.

Some more unrelated bits and pieces...

```
class child : public parent
{
public:
  child();
  ~child();
};
```

What does “public” inheritance mean?
What if we write “protected” or “private”?

“public” here says says public members of the parent class will become public members of the subclass, protected members will be protected, private will be private.
“protected” will upgrade public members of the parent class to protected members of the child class, protected and private will be unchanged
“private” says that both public and protected members of the parent will be private members of the subclass, that is, all is private. The only thing which can be public are members and functions newly declared in the subclass.

```
class child : virtual public parent
{
public:
  child();
  ~child();
};
```

This exists, but is beyond the scope of this course...

vectors and arrays

```
int main(){
int matrix[20];
int i;
for (i=0; i<20; i++) matrix[i] = i;
}
```

Remember: Array indices start at 0! So the “matrix[20]” will have elements 0 through 19.

```
int main(){
int matrix[] = {0,1,78,5};
}
```

You can initialize arrays this way...

```
int main(){
int *matrix;
int i;
matrix = new int[20];
for (i=0; i<20; i++) matrix[i] = i;
delete [] matrix;
}
```

You can use [] and pointer (*) interchangably. You must delete arrays with the “delete []” operator.

```
int main(){
char *string;
int i;
string = new char[strlen(”hello”) +1];
strcpy (string,”hello”);
delete [] string
}
```

remember to allocate one extra element for the invisible “0” character in C-strings.
On the left is a veryfrequently found piece of code.

more-dimensional arrays

```
program dim
integer matrix (20,30)
```

=

```
int main(){
int matrix[20][30];
```

?

Let's say it's very similar when you use it. However, conceptually this is different. The C++ code declares "a vector of length 20 of pointers to a vector of int of length 30". So "matrix" is a pointer to a pointer to int. There is no length/dimension checking done anywhere.

In Fortran, you can only get a "rectangular blob" of memory, that is, the above will always have 20*30 elements. In C++ you can (dynamically) construct arrays with "ragged" boundaries, where each row has a different number of columns.

```
int main(){
int **matrix;
matrix = new int *[20]; // a vector of length 20 of pointers to int
for (int j=0, j<20; j++) *matrix[j] = new int [30];
// do something… this is the same as matrix[20][30];
for (int j=0, j<20; j++) delete [] *matrix[j];
delete [] matrix;
}
```

```
int main(){
int **matrix;
matrix = new int *[20]; // a vector of length 20 of pointers to int
for (int j=0, j<20; j++) matrix[j] = new int [j+10];
// do something… this is the same as matrix[20][30];
for (int j=0, j<20; j++) delete [] matrix[j];
delete [] matrix;
}
```

You may have seen the message system...

not for the faint-of-heart, but very instructive: A 3-dimensional, dynamically allocated array. The dimensions are state (type , source , severity). We allocate the array dynamically in the class constructor because we don't want to hard-wire the respective numbers.

```
class filter_msg_buffer : public   msg_buffer {
  streambuf * original_streambuf;
  int ***state;
// stuff deleted
```

```
 // allocate new memory for the matrix
  state = new int **[msg_type_max];
  for (i=0; i< msg_type_max; i++)
    {
      state[i] = new int* [msg_source_max];
      for ( j =0; j<msg_source_max; j++)
      state[i][j] = new int [msg_sev_max];
    }

  for (i=0; i < msg_type_max; i++)
    for (j=0; j < msg_source_max ; j++)
      for (k=0; k < msg_sev_max ; k++)
        state[i][j][k] = ON;
}
```

```
 // free memory from the matrix
  int i,j;
  for (i=0; i< msg_type_max; i++)
    {
      for ( j =0; j<msg_source_max; j++)
          delete [] state[i][j];
      delete [] state[i] ;
    }
  delete [] state;
}
```

please convince yourself that we end up with an appropriately-sized array, and that we free the memory again correctly.

That’s it!

Thank you for your patience!