C++ for Ocean Modeling Branch Consideration

Robert W. Grumbine

Ocean Modeling Branch

National Centers for Environmental Prediction

June 14, 2000

Technical Note

OMB Contribution 185

1 Abstract

C++ is a modern programming language (mid 1980's invention) which provides a number of features which make it useful for some OMB tasks. The two major features are abstract data types (a late 60's to mid 70's invention) and object orientation (an 80's invention). Tasks for which these are particularly helpful are data-related processing and graphics. This document discusses these features, and how to use the OMB C++ library.

Contents

2 Introduction

Fortran, which we're all familiar with, has a long history and is very well adapted to the core modeling activities that we often are involved in. On the other hand, it does date to the 1950's, and some things have been learned in the last few decades on how to build a general purpose programming language. Two of the key elements are abstract data types, and object orientation. Fortran 90 (which we is now available on the workstations and central systems) takes some steps in the direction of abstract data types. Object orientation is still left out of Fortran (and it isn't clear to me that it is even possible to add it without breaking some of the features which make Fortran efficient for the modeling).

Abstract data types are structures which the programmer, rather than the language, defines. One we could use is 'buoy report'. A buoy report contains certain types of information, all of which should remain associated with each other. In Fortran, I've kept the association by doing things like having a bunch of arrays (latitude, longitude, temperature, wind speed, pressure, ...) and using the same index to mean a given buoy. While this works, it does leave open the possibility that I've mis-typed an entry so that I've got latitude(j) and longitude(i), where i and j turn out to be different numbers. (Murphy's law being what it is, I've done exactly this.) With an abstract data type, I would be referring to buoyreport(j). Each element is then pieced together by the computer and I don't have to remember multiple indices. Much reduces the opportunity for errors (though, of course, Murphy will have his shots anyhow). It also means that in writing the program, I can use much more intuitive description and intuitive functional units. All of my data are in nice little packages, and I can define the packing to be as close as possible to how I think of the problem.

Object orientation takes this the next steps. The first step is, we package not just data units, but the operations that we routinely perform on those units. The major step is that once we've declared some object type, say a '2d grid', we can then create a new object type, such as a 'metric 2d grid', which will *on creation* be able to do all the things that the '2d grid' was able to do (we packed the operations in to the definition)! This is inheritance. In this example, 2d grid is just a 2d array of things – they may be numbers, they could be buoy reports, they could be model fields. The metric grid adds operations to locate i,j points in terms of latitude and longitude. More on this later.

Once these objects are created, if we've done the job correctly, any later user can use the objects without having to go in to the gore of how things are being done. This goes well beyond the Fortran notion of having a library of functions and subroutines. In Fortran (even in '90) if you wanted to locate the latitude-longitude of a point in a grid, you'd still have to know the full grid specification (standard latitude, standard longitude, location of pole in x and y, resolution in x and y, ...). With C++, you would request creation of a 1/16th Bedient mastermap object (call it x). If you then wanted the location of a point in that grid, you'd get it by asking for x.locate(ijpt). At *no* time do you need to know anything about the nature of the grid (even the fact that it is polar stereographic, much less what a Bedient grid is). The grid is an object, and the behavior of the object is specified by the library.

In the rest of the document, I'll go in to some more detail on how it is that C++ fills these roles, and what it is that is useful to us. A reminder is that if the primary operation is hammering on arrays of numbers, then Fortran is still the preferred language. If the problem, as it is for satellite data, involves mostly deciding what the data *are* that we're going to work with, and doing some relatively modest mathematics on them, then C++ is probably the better tool. I'll go in to detail later, but note that BUFR and GRIB messages are themselves objects. Much of our agony in dealing with them is that we're not treating them as objects.

The following sections are: abstract data types, objects introduction, objects and operators, object templates and inheritance, using the OMB class library, and other considerations.

3 Abstract Data Types

Both C and C++ (and now Fortran 90) permit abstract data types. While I don't want to embed details of how the languages do this, it will be helpful to include illustrations, so I'll use a pseudo-language. From the example already alluded to, let's create a buoy report data type:

```
type buoyreport {
  real lat, long;
  real t_air, pressure, t_sea, u_wind, v_wind;
  integer qc_code, platform_type;
  character name[60];
  integer yy, mm, dd, hh;
}
```

In other words, we've declared that there is something called a buoy report, and that every buoy report has certain data associated with it. Inside the program, we can then declare

```
 buoyreport x[1000];
```

exactly as we might say REAL x(1000) to declare an array of 1000 real numbers. In this case, we've declared an array of 1000 buoy reports.

This is only a start on using abstract data types, but let's see what we can gain by it. First, we can do windowing more readily, e.g.

```
if (abs(x[i].lat - lat_ref) < 0.2 AND abs(x[i].lon - lon_ref)  ) then
C  process data for being in range
endif
```

Note that elements of a buoyreport are referred to by giving the name of the variable (x) dot element name. The above is more flexible and readable than having separate arrays for latitude, longitude etc. (the Fortran 77 alternative). This is not a tremendous savings.

But now consider the processing that we would do. This likely will be done in a subroutine. In F77, we would have to give an argument list to the subroutine like SBR(lat, lon, temp, etc. etc. etc.). And since the subroutine may not be in the same file as the calling routine, we get the added joy of keeping two separate codes synchronized. The joy increases when we've got multiple sorts of buoys, each with slightly different data elements. By declaring the abstract data type, we avoid all of that. The subroutine is a subroutine that takes a buoyreport as an argument. There's nothing extra to synchronize between the codes as we've defined what buoyreports are. If we later add different types of reports, say to separate cman_reports, drifter_reports, etc., we only have to define what these entities are (though in C and F-90, each of these must be declared in full separately. C++ permits us to create a new type by saying that it is just like an old type except that it also has ... whatever.)

There's an additional level of utility in the abstract data types. That is, an abstract data type can include data elements which are themselves abstract data types. In the buoyreport example, I had yy, mm, dd as data elements. This is to record the time of the observation. It would be more convenient to simply say date. We can do exactly that – declare a data type called 'date' and then make observation_time a variable in the buoy report. Now, when we write our function to do a time difference check, we can call time_diff(time1, time2, delta), where time1 and time2 are the observation time, reference time, and the time difference (we may choose to make that in hours, or perhaps to make it a 'date' type variable itself).

This nesting of abstract types permits a very natural construction of the programming. For my pre-BUFR SSMI - SDR data processing, for instance, I have a several level nesting, each level of which makes internal sense. The set up was that there's an orbit of data. Each orbit of data has an sdr header followed by a number of data records. (SDR_HEADER, and DATA_RECORD are then abstract data types we define). Each data record includes a scan_header and a data_block. The scan header includes various elementary data pieces which we look up, declare, and are finished with. The data_block includes some information on the mode of the satellite, and then 64 pieces of 'long data'. Long data turns out to be latitude-longitude of observation, surface type, position within the scan line, 7 antenna temperatures, and three pieces of 'short data'. Short data is latitude-longitude of observation, surface type, and two 85 Ghz antenna temperatures.

We can now work at whatever level we're interested in. If we want the data that characterize the orbit itself, they're available in the sdr_header. If we want to know about the scan line, it's in the scan_header. If we're looking for 19v antenna temperatures, those are in the long_data, and so on. The great virtue here is that we *don't* need all at once to know how to locate all pieces of information. Further, we aren't restricted from changing some of the pieces. That is, we could discover that long_data included a quality control flag (or we decide to add one ourselves). Rather than rewriting every piece of code we have, we merely change the definition of long_data, and *only* where we use or change this new bit, add some code.

In the buoy report example, I listed the time of observation as being year, month, day, hour. Suppose now we discovered that the buoys reported time to the minute. In the F-77 case, we have to add an argument to all our subroutines (in the right order, with the right declarations within our called routines, etc.). With abstract data types, there is *no* change to any routines except *internal* to the few routines that compute things with dates. Every other routine merely refers to a date. They don't care exactly what a 'date' is, but if it matters, they can correctly pass a date along to the routine (like time_diff) which does. Only in the interior of time_diff do we have to rewrite any programming. (Aside – note that there is still some work to be done. The newer programming capabilities don't get rid of that fact. What they do, as here, is to let us make the changes in the fewest possible places.)

Abstract data types are definitely good things, in the right cases. A bad case is when the problem really does consist of only a few things which are elementary data types. One example would be the dynamics section of a CFD model. We have arrays of temperature, pressure, velocity, We could construct an abstract data type:

```
type dynamics {
  real temperature(360, 180), salt(360, 180), u(360, 180), v(360, 180);
  real pressure(360, 180);
}

dynamics model;

  or

type dynamics {

  real temperature, salt, u, v, pressure;

}

dynamics model(360, 180);
```

but it would be hard to argue that we make the code any clearer by doing this. Abstract data types are a tool, not necessarily the solution.

4 Objects Introduction

As mentioned in the introduction in addition to bundling bits of data together we bundle the operations which can be done on those data into our objects. Using only abstract data types, for example, we have to have a subroutine time_diff(t1, t2, delta) that returns a time difference given two different times. One source of errors is that we have to put the arguments in the right order. Another is that delta may be a different type than t1 and t2 (delta might be an integer, number of hours, rather than a 'date' type of variable). Worse, if we need to add some capabilities to the time_diff subroutine, we may need to change (add to, say) the argument lists and pass additional data (for instance, a change to dates in seconds from dates in minutes or changing delta to be seconds difference rather than a date difference).

By bundling together the operations with the data, we can avoid those opportunities for error. delta = t1 - t2 becomes a legitimate expression. No argument order to remember.

What needs to be done is for one person to define how to subtract two dates. We may want the delta to be an integer (number of seconds), in which case by operator overloading (section NN) we'll get the conversion from whatever the - operation returns to an integer.

An illustration of an object (I'll use something like C++ code here) is:

```
class date {
  public:
    int year, month, day, hour, minute, second;

    operator+(date);
    operator-(date);
    operator=(date);
    operator>(date);
    operator<(date);

    near(date, window);
    int seconds(date);
    int()(date);
}
```

What we've done here is to say that dates are given by the year, month, day, hour, minute, second, same as for an abstract data type. We also declare that it will be possible to add, subtract, and equate two dates (the business with (date) is saying that we're declaring a function which takes a date as its argument). More interesting is that we are also declaring that it is meaningful to ask if one date is greater than or less than another date. Now we can use proper logical tests! Much, much, nicer than either explicitly calling a function or inlining the series of tests that would otherwise be needed.

Before proceeding in to the details of objects, I'll list some things that we work with that can easily be considered/ treated as objects.

Dates

Locations

Buoy reports

Soundings (both atmospheric and oceanic)

Satellite scans

2d grids

3d grids

Metric grids

GRIB messages

BUFR messages

HDF

SDR, TDR, EDR files/messages

CEOS data (Alaska SAR facility Radarsat, some ERS data)

The list also serves to introduce the notion that we can nest types of objects. A metric grid, for instance, is a particular type of 2d grid. As I've done it for the COFS interpolator, for instance, a metric grid is a 2d grid which also has a latitude-longitude < - > ij mapping. Each type of object also can have further descended objects, for instance, the class (type of object) metric grid, can have subclasses:

Polar stereographic

Latitude-longitude

Mercator

Lambert Conformal

COFS

ETA native

Gaussian

Spectral

EASE

And, to continue a bit, Polar stereographic can have subclasses

North ice analysis

South ice analysis

North ice model

South ice model

North Bedient

Nouth Bedient

North NASA

Nouth NASA

The line of descent is then 2d grid -> metric grid -> polar stereographic grid -> North ice analysis (for example). The 2d grid is a nice general object, with quite a few operations that can be specified and typically used. This class has a few thousand lines supporting it. The metric grid adds some capabilities, namely location translation (but, since we don't know what the mapping is, we can't really invoke this class. This is a virtual class, which will be defined in section 6) for only a few lines. Creating a subclass of the metric class requires that we add a couple dozen lines per grid type (namely, describing the mappings between latitude-longitude and i-j) , but once this is done, we can do anything that we can do to any ancestor class – anything that can be done with a 2d grid can be done with a polar stereographic grid. And, here's the important part, *nobody* needs to write an extra line of code to do this. We make the compiler figure out those parts. The last bit of descent, declaring a north ice analysis grid, takes only the few lines required to specify what is different between the north ice analysis grid and the generic polar stereographic grids. Only a handful of lines, and we make the grid's creator write those.

Suppose, now, someone wanted to use a data file that contained a north ice analysis grid. The form for doing so would be:

#include "metric.h" (include the metric grid class)

main program here

north_analysis x; (declare that x is a north_analysis grid)

latpt llocation; (a location in latitude-longitude space)

ijpt ijlocation; (a location in ij space)

int val; (an integer value)

(various set up done here)

x.read("b3north.970618"); (read in the north_analysis grid from some file)

x.print("output"); (print it out in plain text)

ylocation = x.locate(ijlocation); (find the latitude-longitude of an ij point)

val = x.get(ylocation); (find the value corresponding to a latitude-longitude location)

val = x.get(ijlocation); (find the value corresponding to an ij location)

...

Note that our programmer here never needed to know anything at all about the nature of the north_analysis grid, except that there was such a thing and some descriptions of what things can be done to 2d grids. We've gone beyond Fortran libraries because we never need to know those details. Further, we'd write the same program if we were working with a mercator grid, an ETA grid, etc. No arguments change, and no function names change. At some later date, the north_analysis grid could become higher resolution, or change to Lambert Conformal (say), and our programmer would *still* not need to change a single line of his program.

5 Objects: Operators, and Overloading

In constructing our object, we can declare how some special operators – including +, -, = work for our object. If the object is primarily a mathematical one (as most of ours tend to be) these operations are a good idea. This also lets us use standard math in writing operations in cases where we might not be able to in Fortran. For instance, in one part of my SSMI processing, I add the brightness temperatures for later averaging. At the moment, I do this in C with abstract data type syntax, like sum.t19v = sum.t19v + new.t19v; where I then repeat this for each data entry. In C++, I'd write it as sum = sum + new; and avoid the extra opportunity for typos.

In between all of this is the fact that since we declare operations (and functions) with respect to a specific class, we have the opportunity to use the same function name in different contexts. For instance, having defined a version of + to work on SSMI points, we can write x + y where x and y are ssmi points, or where they're integers. We make the compiler figure out which one is appropriate in the context. This is called overloading. Because of overloading, we are able to focus more on what we're trying to do than on the details of what the computer knows about.

Overloading also permits y = x.locate(ij); and z = x.locate(ll); to be interpreted correctly. In the first case, I'm requesting the latitude-longitude location of a given i,j point in a grid. In the second, I'm requesting the i,j location of a given latitude-longitude point in the grid. In both cases, we want a location so we use the same name. What sort of location we want is determined by the type of information we pass the function. The compiler sorts out which locate function to use, rather than making us look up (and remember) two different names.

It goes beyond this, even. Since x is some type of object (maybe a polar stereographic grid, maybe a mercator grid), not only does the compiler pull out whether I want the locate that gives a latitude-longitude versus an i,j, it also determines what grid system's conversion I'm going to need. Since we have a number of different map projections, and a number of specializations of each projection, there would be something like 500 function names to remember in a Fortran code, as compared to the one that we have via C++. One alternate method, used in the w3ft32 routine, is instead of having dozens of subroutines, to pass the grid numbers that correspond to the grids you want. The code is well-written, but instead of memorizing dozens of subroutine names, you have to memorize dozens of grid code numbers. Overloading avoids both.

6 Object Templates and Inheritance

Templates and inheritance different things, but they derive from the same principle. The principle is that objects should behave in the same way as similar objects. By inheritance, we say that the new object is just like the old one, except for the following, (whatever it is that is different.) Templates are different in that we don't create a new class, but we do say that we have the same operations being performed in the same way. To illustrate:

```
template <class T>

class grid2 {

  int nx, ny;

  T grid(nx, ny);

  etc.

}
```

We've declared that there is a grid2 class, that it has integers nx and ny (i.e. the dimensions of our grid), and that it has a grid of data nx by ny of type T. What is this T? T is our templated type. T is *any* type of thing that the system knows about. It could be an integer, it could be a floating point number, it could be a buoyreport. Logically, we can have grids of all kinds of things. Rather than defining separate classes for each kind of thing that we might grid (a phenomenally tiresome task), and write separate sets of subroutines to operate on each one of them (even worse), we say that we don't care what kind of thing it is that gets gridded. Whatever it is that we've got a grid2 of, we expect to be doing certain things (that's what's in the etc. of the declaration) such as adding two grids of these 'T' things.

Again, this gives us a great savings over Fortran and C. Instead of writing separate routines to find the average of a grid of integers and for a grid of real numbers, we write one routine. The logic is the same after all. Once we write the one routine, we are done. For example, the averaging function for a templated grid is:

```
template <class T>
T grid2<T>::average() {
  // Note that this says a grid2 of things of type T can be averaged.
  // There are no arguments to this routine.
  // The result of the averaging operation is another variable of type T
  double sum;  // Make our sum double precision to ensure against overflow
               // in the summation process.
  ijpt loc;    // We'll move over all points in the original grid.

  sum = 0.0;
  for (loc.j = 0; loc.j < this->ypoints() ; loc.j++) {
  for (loc.i = 0; loc.i < this->xpoints() ; loc.i++) {
     sum += this->operator[](loc);
  }
  }
  sum /= (this->ypoints()*this->xpoints() );
  return (T) sum;
}
```

The actual implementation is somewhat easier to read than this as we made use of some internal features. This code is robust against changes to the internal representation of grid2's.

Inheritance, I've already mentioned some regarding. Here we'll fill out the picture. We start with our base class, grid2:

```
template <class T>
class grid2 {
  int nx, ny;
  T grid2(nx, ny);

  T average();
  T maximum();
  T minimum();
  grid2<T> laplace();

  writeout(FILE *);
  readin(FILE *);


};
```

The grid2 class I've constructed has quite a few more capabilities than this, but this is illustrative. We can to arithmetic on grid2's, we can read and write them, and we can do a few operations on them like finding the average or taking the laplacean. This is our basic 2d array of things class. It is a templated class, so that the 'things' can be more than the usual integer and floating point as long as we've defined how the operations that grid2's are supposed to be able to do – +,-,*, in this case – work.

Handy as this is, we may want to do more. Often we wind up caring about what point on the earth a grid point corresponds to. In order to do this, we need to know what the map projection is, in full detail. We could, and I did this initially, simply start up a set of classes and say that polars tereographic was descended from the grid2, and a latitude-longitude grids was also descended (separately) from a grid2, and on for each type of map projection. This is legal, but it obscures the fact that each projection has some things in common. Regardless of what the map projection is, we will need two functions: one to convert from ij coordinates to latitude-longitude, and one to convert back. We will also find it useful to import a grid2 to a projection grid. Here we have a case where we know that there are some operations which are required for a proper member of the group, but we don't know ahead of time how to write a function for all of them. This is the case where we want a 'pure virtual' class. The metricgrid is:

```
template <class T>
class metricgrid : public grid2<T> {
  operator=(grid2 );
  latlonpt locate(ijpt)     = 0 ;
  ijpt     locate(latlonpt) = 0;
}
```

We declare that there will be a class called metricgrid, and this is a templated class which inherits from grid2. The first line says that we're going to define how to import a grid2 in to a metricgrid. (We can do this since we can say that it simply means to let the nx, ny, and grid values of the metricgrid be the same as the grid2.) The next two lines are peculiar. The first part is the expectable business of saying that I have two 'locate' functions, one which takes an ijpt and returns a latlonpt, and one which takes a latlonpt and returns the corresponding ijpt. We can't actually write these functions, because we don't know what the map projection is. We put the = 0 in the declaration in order to tell the compiler this. By doing so, we have made metricgrid a virtual class. You can't actually declare an object which is a metricgrid.

What you *can* do is declare a class which is descended from the metricgrid. In order to construct this class, however, the compiler is going to require that you define how those two locate functions work. For free you get the thousand (and rising) lines of support that exist in the grid2 class, and get to use all functions which know how to use a metricgrid. In repayment, you have to add a couple dozen lines (or whatever) it takes to make the locate functions work. (Well, *someone* has to write those lines. Preferably we make, say, the ETA people write the ETA locate functions.)

By proper use of these pure virtual classes and some related matters, we also can construct robust libraries and enforce standards. The libraries become robust because we know that certain functions are required of the classes, and can require that they be specified even before knowing who is going to make a new class or for what reason. We enforce standards by virtue of the fact that either: 1) the function or datum is inherited from a standards-conforming base class or 2) the function is required to exist by the base class. We have the added utility in standards enforcement that it becomes much easier to find the right routine to perform the action you want. First, fewer names are required. And second, since the actions are tied to the classes, we know to look in the declaration of the class.

7 Using the OMB Class Library

The class library itself is defined in tech note 186, online at http://polar.wwb.noaa.gov/omb/papers/tn186/. The online documentation should be taken as authoratative rather than comments in this illustrative note. A more detailed examination of a particular usage of the class library – the COFS interpolator – is given in Grumbine [2000b].

To compile a program in C++, you need to name the compiler, and add 'define' the platform you're on, and give directions to the include and library directories. The specifics for each local platform are given below. The command line will look like:

```
g++ program.C -DLINUX -I /usr/local/include /usr/local/lib/cpplib
```

platform	compiler	include directory
IBM SP - operational	xlC	/nwprod/omblib90/omblib.source/include
IBM SP - developmental	xlC	/nfsuser/g01/marine/local/include
sgi100	CC	/migr/data/wd21rg/includes
polar	g++	/usr/local/include

platform	library	D
IBM SP - operational	/nwprod/omblib90/omblibc_4_604	IBM
	/nwprod/omblib90/omblibf_4_604	
IBM SP - developmental	/nfsuser/g01/marine/local/lib/cpplib	IBM
sgi100	/migr/data/wd21rg/lib/	SGI
polar	/usr/local/lib/cpplib	LINUX

8 References

Grumbine, R. W., 2000a, OMB C++ Class Library Description, Technical Note 186, http://polar.wwb.noaa.gov/papers/tn186/.

Grumbine, R. W., 2000b, OMB C++ Class Library Demonstrations. In Preparation.

Stroustroup, B., 1997, The C++ Programming Language, 3rd edition, Addison-Wesley, 911 pp.