

Week 9 Notes: A Tour of C++ for Scientific Computing

Adapted from the CAAM 420 Fall 2010 Notes of **Professor Symes**

October 17, 2011

Contents

1 C++ as a "better C"	2
2 Classes and Class Hierarchies in C++	2
3 Order of construction and destruction	14
4 Concrete, virtual, and pure virtual functions	14
5 Default Arguments	16
6 Abstraction	16
7 Using C functions to define classes and class methods	23
8 Source files vs. header files	25
9 Combining attributes via multiple inheritance: avoiding cyclic inheritance using mixins	25
10 Matrices without Matrices, and dynamic casting	27
11 Const, intentional copying, and copy construction	28
12 C and C++ strings	29
13 Templates and Generic Programming	31
14 Template specialization and interface to non-generic code	37
15 Complex numbers in C++	37
16 Exceptions	38
17 The Standard Library's templated container classes	42
18 Project organization: building your own libraries	44
19 Namespaces	46

1 C++ as a "better C"

C++ enhances C in various ways - please study these examples, and read the corresponding sections in Stroustrup Chs. 4-7:

- bool type
- pass by reference
- linking to C, avoiding name-mangling by using extern "C"
- elements of i/o streams
- dynamic allocation: new and delete
- default arguments

2 Classes and Class Hierarchies in C++

Following the introduction of classes in C++, their features, and the concept of class hierarchy. An example is given here:

```
1 #include <stdio.h>
#include <stdlib.h>
3
// WARNING: NO ERROR CHECKING
5
7 // forward declaration of smart mydouble class
class mydouble;
9
// example skinny base class for a generic vector
11 class vector {
protected:
13 // single defining characteristic of a variable: length
int dim;
15 public:
17 // default constructor (zero length)
vector(){
19     dim = 0;
}
21
// construct with length (no storage)
23 vector(int in_dim){
    dim = in_dim;
25 }
27 // pure virtual unary [] operator
virtual mydouble operator[] (int id) = 0;
29
// pure virtual unary () operator
31 virtual double operator() (int id) const = 0;
33 // pure virtual set function
```

```

35     virtual void set(int id, double val) = 0;
36
37     // implemented length function
38     int length(){
39         return dim;
40     }
41
42     virtual void report() = 0;
43 };
44
45 // "smart" mydouble class object to provide active entry
46 // access to entry in vector
47 class mydouble {
48
49     private:
50         // will use generic pointer
51         // [ uses the pointer compatibility of classes derived from vector ]
52         vector *spv;
53         int index;
54     public:
55
56         // default constructor
57         mydouble(){
58             spv = NULL;
59             index = 0;
60         }
61
62         // constructor
63         mydouble(vector *in_spv, int in_index){
64             spv = in_spv;
65             index = in_index;
66         }
67
68         // type conversion operator: mydouble -> double
69         operator double () const{
70             return (*spv)(index);
71         }
72
73         // assignment operator (mydouble a = 1.2)
74         mydouble &operator = (const double val){
75
76             spv->set(index, val);
77
78             return *this;
79         }
80     };
81 };
82
83
84
85 // dense vector class derived from vector base class

```

```

87 class dvector : public vector {
private:
89     double *data;
public:
91
92     // default constructor
93     dvector() : vector() {
94         data = NULL;
95     }
96
97     // constructor to set length and data
98     dvector(int in_dim) : vector(in_dim) {
99         data = new double[dim];
100     }
101
102     void set(int index, double val){
103         data[index] = val;
104     }
105
106     // implementation of unary [] operator
107     mydouble operator [] (int id){
108         mydouble pt(this, id);
109         return pt;
110     }
111
112     // implementation of unary () operator
113     double operator() (int id) const{
114         return data[id];
115     }
116
117     // output entries
118     void report(){
119         for(int i=0;i<dim;++i)
120             printf("d[%d] = %g\n", i, data[i]);
121     }
122 };
123
124 // sparse vector class derived from vector base class
125 class spvector : vector {
126 private:
127     int nnz;
128     int *indices;
129     double *data;
130 public:
131
132     // constructor for empty sparse vector
133     spvector(int in_dim) : vector(in_dim){
134         nnz = 0;
135         indices = NULL;
136         data = NULL;
137     }
138
139

```

```

141 // implementation of set function
142 // checks for existing entry and
143 // extends storage for new entries
144 void set(int index, double val){
145     int i;
146
147     for(i=0;i<nnz;++i){
148         if(indices[i] == index){
149             data[i] = val;
150             return;
151         }
152     }
153
154     // check to see if non-zero
155     if(!val) return;
156
157     // extend list by one
158     // oops - realloc is not appropriate here
159     ++nnz;
160     double *tmpdata = new double[nnz];
161     for(i=0;i<nnz-1;++i) tmpdata[i] = data[i];
162     tmpdata[i] = val;
163
164     delete data;
165     data = tmpdata;
166
167     // oops need to realloc indices
168     int *tmpindices = new int[nnz];
169     for(i=0;i<nnz-1;++i) tmpindices[i] = indices[i];
170     tmpindices[i] = index;
171
172     delete indices;
173     indices = tmpindices;
174
175 }
176
177 // implementation of operator () const
178 double operator()(int index) const {
179
180     for(int i=0;i<nnz;++i){
181         if(indices[i] == index){
182             return data[i];
183         }
184     }
185
186     return 0;
187 }
188
189 // return smart mydouble for access to sparse entry
190 mydouble operator[] (int index) {
191
192     mydouble pt(this, index);
193     return pt;

```

```

193     }
195     // output stored entries
197     void report(){
198         for(int i=0;i<nnz;++i)
199             printf("nnz: %d, index: %d, val: %g\n", i, indices[i], data[i]);
200     }
201 };
202
203
204
205 // some demos
main(){
207     // create an empty sparse vector of length 10
    spvector sp_v(10);
208
209     // set non-zero entries
210     for(int i=0;i<10;i+=2)
211         sp_v[i] = -2.4*i ;
212
213     // read value
214     double a = sp_v[3];
215
216     // print out copied value and example read value
217     printf("a = %f \n v(4) = %f\n", a, (double)sp_v[4]);
218
219     // report non-zero entries
220     sp_v.report();
221
222     dvector d_v(10);
223     d_v[3] = 1.6;
224
225     d_v.report();
226
227     // read entry
228     double b = d_v(3);
229     printf("b = d_v(3) = %g\n", b);
230
231
232 }

```

This week we will continue this discussion, again mostly through examples and readings in Stroustrup. The key chapters are 10-12. I will touch just the highlights in my notes and examples.

The examples have already shown that classes are like structs in that they may contain both data and functions as members (function pointers may be members of C structs, though we did not use that construction in our struct examples). However there are two main differences:

- class members may have one of three access types, a restriction completely missing from C structs:
 - private: only accessible directly by instances ("objects") of the class;
 - protected: only accessible directly by instances of the class and its subclasses (and by friend classes, to be covered later);

- public: accessible by any other class or function.
- The compiler generates code to call two member functions automatically namely class constructor(s) and destructor. This feature solves the problem of assigning default or computed initial values to data members, and of getting rid of them when class objects go out of scope - it relieves the programmer of the necessity to call functions to carry out these tasks for every instance, in contrast to C structs.

Constructors are recognizable from the function name: it is the same as the class name. The function overloading mechanism in C++ (Ch. 7 in Stroustrup) permits functions of the same name to coexist, so long as their signatures differ - "signature" is the C++ version of "prototype", see pp. 149-151. Classes often have more than one constructor, all having the name of the class as the function name. They must differ in number or type of argument to be distinguishable. You can see a simple instance of this arrangement in the dense vector class.

Each class has one, and only one, destructor; its name is the class name, preceded by a tilde. The compiler will provide default constructor (with no argument) and destructor (always with no argument). These defaults simply allocate (or deallocate) any variables declared as class data, exactly as a C struct would do. The compiler-provided default constructor does not do any dynamic allocation or initialization of data members, and the compiler-provided destructor does not free or delete any dynamically allocated memory.

Thus last week's dense vector class `dvector` contains a built-in memory leak:

```
void doSomething() { ... dvector v(10); ... }
```

leaks 10 doubles. **ANY CLASS ALLOCATING MEMORY DYNAMICALLY MUST BE PROVIDED WITH ITS PROPER DESTRUCTOR** to avoid memory leaks - that destructor must delete or free any memory allocated during the lifetime of any class object, beginning with construction.

See a revised dense vector class for an example:

```

1 #ifndef _CAAM420_VECPPDENSE_
2 #define _CAAM420_VECPPDENSE_
3
4 #include "vecppbase.hh"
5
6 /*
7  * Author: WWS
8  *
9  * Purpose: first cut at dense vector class – essentially a C++
10  * port of struct vec.
11  */
12
13 class VecPPDense: public VecPPBase {
14
15 private:
16
17     int len;
18     double * vdata;
19
20 public:
21
22     /* default constructor
23     NOTE THE PROBLEM POSED BY THIS DEFAULT CONSTRUCTOR-
24     IT IS IMPOSSIBLE TO USE THE OBJECT PRODUCED BY IT!!!
25     */
26     VecPPDense(): VecPPBase() {

```

```

27     len=0;
        vdata=NULL;
29 }

31 /* main constructor */
VecPPDense(int _len, bool init=false): VecPPBase() {
33     // cout<<"VecPPDense constructor\n";
        len=_len;
35     vdata=new double[len];
        if (init) {
37         for (int i=0;i<len;i++) vdata[i]=0.0;
        }
39 }

41 /* destructor – note that call to delete is safe. At moment contemplate
        no subclasses, so need not be virtual.

43     NOTE proper use of delete for arrays – see Stroustrup p. 251.
45 */
~VecPPDense() {
47     // cout<<"VecPPDense destructor\n";
        if (vdata) delete [] vdata;
49 }

51 int getLength() const { return len; }

53 double & operator()(int i) {
        if (vdata && i >= 0 && i <= len-1) return vdata[i];
55     else {
        cerr<<"Error: VecPPDense::operator() (non-const)\n";
57         cerr<<"attempted access failed – either not initialized or \n";
        cerr<<"out of range\n";
59         /* Will see a better solution to this problem */
        exit(1);
61     }
}

63 double const & operator()(int i) const {
65     if (vdata && i >= 0 && i <= len-1) return vdata[i];
        else {
67         cerr<<"Error: VecPPDense::operator() (non-const)\n";
        cerr<<"attempted access failed – either not initialized or \n";
69         cerr<<"out of range\n";
        /* Will see a better solution to this problem */
71         exit(1);
        }
73 }

75 /* mandatory report function */
virtual ostream & report(ostream & str) const {
77     str<<"VecPPDense object: length="<<len<<"\n";
        for (int i=0;i<len;i++) {
79         str<<"v["<<i<<"]="<<vdata[i]<<"\n";

```

```

    }
81     return str;
    }
83 };
85 #endif

```

also a sparse vector class:

```

8 #ifndef _CAAM420_VECPPSPARSE_
2 #define _CAAM420_VECPPSPARSE_

4 // sparse vector class derived from vector base class
   class VecPPSparse : public VecPPBase {
6
   private:
8     int len;
     int nnz;
10    int *indices;
     double *data;
12    double zero;

14 public:

16    // main constructor for empty sparse vector
     // NOTE includes default constructor!!!
18    VecPPSparse(int in_dim=0) : VecPPBase() {
        len=in_dim;
20        nnz = 0;
        indices = NULL;
22        data = NULL;
        zero=0.0;
24    }

26    // destructor
     ~VecPPSparse() {
28        if (data) delete [] data;
    }

30    int getLength() const { return len; }

32
     // this returns a mutable (assignable) reference
     // references have to refer to something that
     // exists - so construct entry if not already constructed
34    // morph of implementation of set function
     // void set(int index, double val){
36    // in last week's class
     // checks for existing entry and
     // extends storage for new entries
40    double & operator()(int index) {

42        /***** should put index check here!!! */
44        int i;

```

```

46     for(i=0;i<nnz;++i){
47         if(indices[i] == index){
48             return data[i];
49         }
50     }

52     // otherwise extend list by one
53     // oops - realloc is not appropriate here
54     ++nnz;
55     double *tmpdata = new double[nnz];
56     for(i=0;i<nnz-1;++i) tmpdata[i] = data[i];

58     delete [] data;
59     data = tmpdata;

60

61     // oops need to realloc indices
62     int *tmpindices = new int[nnz];
63     for(i=0;i<nnz-1;++i) tmpindices[i] = indices[i];
64     tmpindices[i] = index;

66     delete [] indices;
67     indices = tmpindices;

68

69     return data[nnz-1];
70 }

72 // implementation of operator () const
73 double const & operator()(int index) const {
74
75     for(int i=0;i<nnz;++i){
76         if(indices[i] == index){
77             return data[i];
78         }
79     }

80

81     return zero;
82 }

84 // reimplement of zero operator - minimal storage (none!)
85 // note that there is NO MENTION of double 0.0! That's implicit
86 // in the indexing operator implementation - and these operators
87 // provide the only access to the data vector!!!
88 void zero() {
89     nnz=0;
90     if (data) delete [] data;
91     data=NULL;
92 }

94 // output stored entries
95 ostream & report(ostream & str) const {
96     for(int i=0;i<nnz;++i)
97         str<<"nnz: "<<i<<" index: "<<indices[i]<<" val: "<<data[i]<<"\n";

```

```

98     return str;
99     }
100 };
101
102 #endif

```

which also has an effective destructor. See example:

```

1 /*
2  * Author: WWS
3  *
4  * Purpose: illustrate how destructors avoid memory leak
5  *
6  */
7
8 #include "vecppdense.hh"
9
10 void foo() {
11     int dummy;
12     const int n=100000000;
13     VecPPDense v(n);
14     cout<<"in foo\n";
15     cin>>dummy;
16 }
17
18 int main(int argc, char ** argv) {
19     int dummy;
20     for (int i=0;i<10;i++) {
21         foo();
22         cout<<"in main\n";
23         cin>>dummy;
24     }
25 }

```

to see how the destructor assists in avoiding memory leaks.

These classes are both subclasses of a simple vector base class:

Listing 1: Simple Vector Base Class

```

1 #ifndef __CAAM420_VECPPBASE__
2 #define __CAAM420_VECPPBASE__
3
4 #include "cppstd.hh"
5
6 /*
7  * Author: WWS
8  *
9  * Purpose: first cut at base class for VecPP hierarchy. Adapted
10 * from Warburton's "skinny base class".
11 */
12
13 /*

```

```

14  * Common programming practice:
15  * 1. class names begin with upper case, and are nouns:
16  * MyVector, DenseMatrix,...
17  * 2. function names begin with lower case and are verbs:
18  * get(), setValue(), initializeInternalStruct(),...
19  * 3. avoid internal underscores if possible – separate name
20  * components with capitalization. Internal spaces are illegal,
21  * of course.
22  */
class VecPPBase {
24
25  /* if we insert a "dim" int as private or protected data,
26  * then that implementation of dimension must be inherited
27  * by all subclasses. Will see that this is not always convenient.
28  * however, so instead make dimension-revealing function pure
29  * virtual so that all instantiable subclasses must implement it
30  */
31
32  public:
33
34  VecPPBase() {
35  // cout<<"VecPPBase constructor\n";
36  }
37
38  /* there are no data members, so the default constructor and destructor
39  * provided by the compiler are just fine. However it is NECESSARY to
40  * declare the destructor to be virtual, so that it can be overridden by
41  * subclasses, and thus whatever actual cleanup is needed actually takes
42  * place when a VecPPBase object is destroyed (see Stroustrup p. 319).
43  */
44  virtual ~VecPPBase() {
45  // cout<<"VecPPBase destructor\n";
46  }
47
48  /* public length function – must be implemented in instantiable
49  * subclasses. Does not change any conceivable internal data, so
50  * should be declared const.
51  */
52  virtual int getLength() const = 0;
53
54  /* a solution to the element assignment problem, more "C++-ish": use
55  * call-by-reference, and provide both const and non-const
56  * references. The latter can be assigned!
57  */
58  virtual double & operator()(int i) = 0;
59  virtual double const & operator()(int i) const = 0;
60
61  /* basic linear algebra op,
62  *
63  * y ← a*x + y
64  *
65  * implemented but virtual so can be overridden in derived class
66  */

```

```

68  *
69  * very simple example of an abstract algorithm – uses only functions
70  * defined for all VecPPBase instances. no matter what concrete subclass
71  * they come from.
72  */
73  virtual void axpy(double a, VecPPBase const & x) {
74      if (x.getLength() != this->getLength()) {
75          cerr<<" Error: VecPPBase::axpy\n";
76          cerr<<" incompatible vectors:\n";
77          cerr<<" length of this = "<<this->getLength();
78          cerr<<" length of that = "<<x.getLength();
79          exit(1);
80      }
81      for (int i=0;i<getLength();i++) (*this)(i)+=a*x(i);
82  }
83  /*
84  * standard Euclidean dot product, can be overridden
85  */
86  virtual double dot(VecPPBase const & x) const {
87      if (x.getLength() != this->getLength()) {
88          cerr<<" Error: VecPPBase::dot\n";
89          cerr<<" incompatible vectors:\n";
90          cerr<<" length of this = "<<this->getLength();
91          cerr<<" length of that = "<<x.getLength();
92          exit(1);
93      }
94      double d=0.0;
95      for (int i=0;i<getLength();i++) d += ((*this)(i))*(x(i));
96      return d;
97  }
98  /*
99  * assign zero vector – the only coordinate-invariant assignment.
100  * of course this is not a const function.
101  */
102  virtual void zero() {
103      for (int i=0;i<this->getLength();i++) (*this)(i)=0.0;
104  }
105  /* mandatory report function */
106  virtual ostream & report(ostream & str) const = 0;
107  };
108  #endif

```

which adds two features to last week's example, besides the destructor:

- it uses a pair of operator() overloads returning references to doubles, one const, one non-const, to allow access to elements while not (necessarily) enlarging storage (in the sparse case). The const method is the only one which may be called on const objects - the compiler enforces this restriction (see ex49).
- it adds three linear algebra operations:

- the assignment $y \leftarrow ax+y$ for vectors y, x and scalar a ;
- the dot product;
- zero initialization (the only coordinate-invariant initialization possible).

3 Order of construction and destruction

Class hierarchies like VecPP entail a specific order of construction and destruction. Construction is top down: first all data for the base class is allocated and the commands in the body of the constructor scope executed, then all data for its subclasses and the commands in its constructor’s scope, then all data and constructor commands for their subclasses, etc. Destruction is bottom up: first the destructor for the most concrete (lowest in the class hierarchy) class is executed, then the destructor for the next lowest, etc.

I’ve added print statements to the dense vector constructor and destructor so that you can see this ordering in action, using example 50.

4 Concrete, virtual, and pure virtual functions

The functions defined in the dense vector class:

Listing 2: dense vector class

```

1 #ifndef __CAAM420_VECPPDENSE__
2 #define __CAAM420_VECPPDENSE__
3
4 #include "vecppbase.hh"
5
6 /*
7  * Author: WWS
8  *
9  * Purpose: first cut at dense vector class – essentially a C++
10  * port of struct vec.
11  */
12
13 class VecPPDense: public VecPPBase {
14
15 private:
16
17     int len;
18     double * vdata;
19
20 public:
21
22     /* default constructor
23     NOTE THE PROBLEM POSED BY THIS DEFAULT CONSTRUCTOR-
24     IT IS IMPOSSIBLE TO USE THE OBJECT PRODUCED BY IT!!!
25     */
26     VecPPDense(): VecPPBase() {
27         len=0;
28         vdata=NULL;
29     }
30
31     /* main constructor */

```

```

33 VecPPDense(int _len, bool init=false): VecPPBase() {
    // cout<<"VecPPDense constructor\n";
    len=_len;
35 vdata=new double[len];
    if (init) {
37     for (int i=0;i<len;i++) vdata[i]=0.0;
    }
39 }

41 /* destructor – note that call to delete is safe. At moment contemplate
    no subclasses, so need not be virtual.

43     NOTE proper use of delete for arrays – see Stroustrup p. 251.
45 */
~VecPPDense() {
47     // cout<<"VecPPDense destructor\n";
    if (vdata) delete [] vdata;
49 }

51 int getLength() const { return len; }

53 double & operator()(int i) {
    if (vdata && i >= 0 && i <= len-1) return vdata[i];
55     else {
        cerr<<" Error: VecPPDense::operator() (non-const)\n";
57         cerr<<" attempted access failed – either not initialized or \n";
        cerr<<" out of range\n";
59         /* Will see a better solution to this problem */
        exit(1);
61     }
}

63 double const & operator()(int i) const {
65     if (vdata && i >= 0 && i <= len-1) return vdata[i];
    else {
67         cerr<<" Error: VecPPDense::operator() (non-const)\n";
        cerr<<" attempted access failed – either not initialized or \n";
69         cerr<<" out of range\n";
        /* Will see a better solution to this problem */
71         exit(1);
    }
73 }

75 /* mandatory report function */
virtual ostream & report(ostream & str) const {
77     str<<" VecPPDense object: length="<<len<<"\n";
    for (int i=0;i<len;i++) {
79         str<<" v["<<i<<"]="<<vdata[i]<<"\n";
    }
81     return str;
}

83 };

```

```
#endif
```

all examples of concrete member functions. Each one is implemented (defined), and no provision is made for alternate implementation in subclasses.

The linear algebra functions `axpy`, `dot`, and `zero` are virtual functions: the keyword `virtual` at the beginning of their declarations tells the compiler that subclasses can override the definitions, that is, can supply alternative implementations. Note that subclasses may use the parent class implementations, or override them - the dense vector class uses the base class definition of `zero`, whereas the sparse vector class overrides it (provides an alternative that accomplishes the intended goal), to avoid uselessly increasing storage.

The indexing and report functions are pure virtual - they are not defined at all. Their presence means that an object of class `VecPPBase` cannot be allocated - of course not, as some of the functions lack definitions! For a subclass to be instantiable, i.e. allow allocation of class objects, it must supply implementations of all pure virtual base class functions. The dense and sparse vector classes do this, so you can instantiate (allocate) objects of their types - but not objects of type `VecPPBase`. Of course, if you create a subclass object, it is a priori a member of the parent (abstract base) class, and may be treated as such. BUT you can't allocate a member of the base class, in itself.

5 Default Arguments

C++ permits functions to supply default values for their arguments - see Stroustrup p. 153. I will point out uses of this very handy device as I use it in examples. The main rule is that arguments with default values must all appear after any argument without when any function is called - all arguments up to the last non-default argument must be explicitly passed.

6 Abstraction

The linear algebra methods in the vector base class:

```
#ifndef __CAAM420_VECPPBASE__
2 #define __CAAM420_VECPPBASE__

4 #include "cppstd.hh"

6 /*
  * Author: WWS
8  *
  * Purpose: first cut at base class for VecPP hierarchy. Adapted
10 from Warburton's "skinny base class".
  */

12
14 /*
  * Common programming practice:
  * 1. class names begin with upper case, and are nouns:
16 MyVector, DenseMatrix,...
  * 2. function names begin with lower case and are verbs:
18 get(), setValue(), initializeInternalStruct(),...
  * 3. avoid internal underscores if possible - separate name
20 components with capitalization. Internal spaces are illegal,
  * of course.
22 /*
class VecPPBase {
```

```

24  /* if we insert a "dim" int as private or protected data,
26  * then that implementation of dimension must be inherited
28  * by all subclasses. Will see that this is not always convenient.
30  * however, so instead make dimension-revealing function pure
32  * virtual so that all instantiable subclasses must implement it
34  */
36  public:
38  VecPPBase() {
40  // cout<<"VecPPBase constructor\n";
42  }
44  /* there are no data members, so the default constructor and destructor
46  * provided by the compiler are just fine. However it is NECESSARY to
48  * declare the destructor to be virtual, so that it can be overridden by
50  * subclasses, and thus whatever actual cleanup is needed actually takes
52  * place when a VecPPBase object is destroyed (see Stroustrup p. 319).
54  */
56  virtual ~VecPPBase() {
58  // cout<<"VecPPBase destructor\n";
60  }
62  /* public length function – must be implemented in instantiable
64  * subclasses. Does not change any conceivable internal data, so
66  * should be declared const.
68  */
70  virtual int getLength() const = 0;
72  /* a solution to the element assignment problem, more "C++-ish": use
74  * call-by-reference, and provide both const and non-const
76  * references. The latter can be assigned!
78  */
80  virtual double & operator()(int i) = 0;
82  virtual double const & operator()(int i) const = 0;
84  /* basic linear algebra op,
86  *
88  *  $y \leftarrow ax + y$ 
90  *
92  * implemented but virtual so can be overridden in derived class
94  *
96  * very simple example of an abstract algorithm – uses only functions
98  * defined for all VecPPBase instances. no matter what concrete subclass
100  * they come from.
102  */
104  virtual void axpy(double a, VecPPBase const & x) {
106  if (x.getLength() != this->getLength()) {
108  cerr<<" Error: VecPPBase::axpy\n";
110  cerr<<" incompatible vectors:\n";
112  cerr<<" length of this = "<<this->getLength();

```

```

    cerr<<"length of that = "<<x.getLength();
78     exit(1);
    }
80     for (int i=0;i<getLength();i++) (*this)(i)+=a*x(i);
    }
82
83     /*
84     * standard Euclidean dot product, can be overridden
85     */
86     virtual double dot(VecPPBase const & x) const {
87         if (x.getLength() != this->getLength()) {
88             cerr<<" Error: VecPPBase::dot\n";
89             cerr<<" incompatible vectors:\n";
90             cerr<<" length of this = "<<this->getLength();
91             cerr<<" length of that = "<<x.getLength();
92             exit(1);
93         }
94         double d=0.0;
95         for (int i=0;i<getLength();i++) d += ((*this)(i))*(x(i));
96         return d;
97     }
98
99     /*
100    * assign zero vector - the only coordinate-invariant assignment.
101    * of course this is not a const function.
102    */
103    virtual void zero() {
104        for (int i=0;i<this->getLength();i++) (*this)(i)=0.0;
105    }
106
107    /* mandatory report function */
108    virtual ostream & report(ostream & str) const = 0;
109 };
112 #endif

```

are very simple examples of abstract algorithm implementation in C++. These functions are implemented, but use only methods of the base class, some of which are pure virtual (definition deferred to subclasses). This means that any definition (implementation) of the missing (indexing) operations may be used, so long as the mathematical meaning is the same.

This is an astonishingly powerful device, allowing the programmer to implement algorithms at their natural level of abstraction, deferring irrelevant details to other software layers. Note for example that the definition of vector addition (the `axpy` method of `VecPPBase`) requires only that the index operator work as it should - it depends only on WHAT the index operator does, not on HOW it works.

An important and less trivial example of abstraction is an abstract implementation of the Conjugate Gradient method for solution of linear systems (Golub & van Loan, *Matrix Computations*, 3rd edn., pp. 520 ff). This algorithm solves a linear system $Ax = b$ for a special category of matrices (symmetric positive definite) which occur often in applications. It does not actually use the matrix entries directly, unlike Gaussian elimination - it only uses the matrix-vector product.

I've written a simple matrix base class:

Listing 3: matrix base class

```

/*
2  * Author: WWS
  * Purpose: simple linear operator base class
4  */

6  #ifndef __CAAM420_MATBASE__
  #define __CAAM420_MATBASE__
8
  #include "vecppbase.hh"
10
  /** defines basic interface for linear operators = implicit
12      matrices - chief attribute is ability to multiply a vector.
  */
14  class MatBase {

16  public:

18      virtual ~MatBase() {}

20      /** abstract nrows, ncols functions */
  virtual int getNumRows() const = 0;
22      virtual int getNumCols() const = 0;

24      /** abstract indexing pair */
  virtual double & operator()(int, int) = 0;
26      virtual double const & operator()(int, int) const = 0;

28      /** main operator - given a default implementation using
  the abstract interfaces of this and VecPPBase. Very much
30      same as old matmult, except for dim check
  */
32      virtual void matmult(VecPPBase const & vecin,
                          VecPPBase & vecout) const;

34
  /** reporting function - shows C++ stream equivalents of format
36      conversion - "%n.me" in fprintf has same effect as
  str<<setw(n)<<setprecision(m)<<setiosflags(ios::scientific)
38      */

40      virtual ostream & report(ostream & str) const;

42  };

44  #endif

```

analogous to (and using) the simple vector base class described above. It defines access to entries, and in terms of it a matrix-vector multiplication method (member function) - basically, good old `matmult.c`, rewritten as a class method, with an overridable (virtual) implementation that uses only the pure virtual access and dimension methods. Exactly how this is done is discussed below - the header file contains only declarations, not definitions.

I've provided a simple implementation of CG:

Listing 4: Simple Conjugate Gradient Solver

```

#include "matbase.hh"
2
/*
4  * implementation of simplest version of Conjugate Gradient
  * iteration, using only base class matrix and vector methods
6  */
void simplecg(MatBase const & A,      // matrix
8          VecPPBase const & b,      // right-hand side
          VecPPBase & x,              // estimated solution
10         VecPPBase & r,              // residual  $b-A*x$  - workspace
          VecPPBase & p,              // search direction - workspace
12         VecPPBase & ap,             //  $A*p$  - workspace
          int nit) {                  // number of iterations (?)
14
    /* workspace */
16    double rtr;
    double ptap;
18    double alpha;
    double beta;
20    int it;

22    /* compute the initial residual - note that this
      amounts to a copy, maybe we should add a copy method to
24    the VecPP base class
    */
26    r.zero();
    r.axy(1.0,b);
28
    /* rtr is dot product of r with itself */
30    rtr = r.dot(r);
    cout<<"initial squared residual = "<<rtr<<"\n";
32
    /* initial search direction is same as initial residual -
34    another argument for copy method
    */
36    p.zero();
    p.axy(1.0,r);
38
    /* initial guess at solution: zero */
40    x.zero();

42    /* CG loop */
    for (it=0;it<nit;it++) {
44
        A.matmult(p,ap);
46        ptap=p.dot(ap);
        // this is dangerous!!!
48        alpha=rtr/ptap;
        x.axy(alpha,p);
50        // r = b-A*x always!!
        r.axy(-alpha,ap);
52        beta=rtr;

```

```

    rtr=r.dot(r);
54 cout<<"squared residual at step "<<it+1<<" = "<<rtr<<"\n";
    // this is dangerous too!!!
56 beta=rtr/beta;
    // a convoluted way of writing p = r + beta*p
58 p.axy(beta-1.0,p); // p = beta*p
    p.axy(1.0,r); // p = r + p
60
62 }
}

```

which uses only the methods of the matrix and vector base classes. Note that the only matrix attribute used is its ability to multiply a vector by the matrix. ANY implementation of matrix-vector multiply could be used.

Disclaimer: this implementation of CG is not industrial-strength. We will improve it substantially over the next several weeks.

To exercise this thing, we need an instantiable (concrete) matrix class, so we can actually allocate matrix objects. For a first pass, here is a simple dense matrix class, declarations only:

Listing 5: prototypes for dense matrix class

```

/*
2 * Author: WWS
  * Purpose: simple dense matrix class, after struct mat
4 */

6 #ifndef __CAAM420_MATDENSE__
  #define __CAAM420_MATDENSE__
8
  #include "matbase.hh"
10
  class MatDense: public MatBase {
12
  private:
14
    int nrows;
16    int ncols;
    double * data;
18
    /* here is a way to make sure that the useless
20     and irritating default constructor does not
     get called
22
     The practical effect: any successfully constructed
24     MatDense must have allocated memory to the double *
     data. Therefore it can safely be deleted in the
26     destructor.
    */
28    MatDense();

30 public:

32    /** the real constructor */
    MatDense(int n, int m);

```

```

34  /** the real destructor */
36  ~MatDense() { delete [] data; }

38  /** abstract nrows, ncols functions */
39  int getNumRows() const { return nrows; }
40  int getNumCols() const { return ncols; }

42  /** indexing op – non-const*/
43  double & operator()(int i, int j);
44  /** indexing op – const */
45  virtual double const & operator()(int i, int j) const;
46
47  };
48  #endif

```

and here is an example driver:

```

1  #include "matdense.hh"
2  #include "vecppdense.hh"
3
4  void simplecg(MatBase const &,
5              VecPPBase const &,
6              VecPPBase &,
7              VecPPBase &,
8              VecPPBase &,
9              VecPPBase &,
10             int);
11
12 int main(int argc, char ** argv) {
13
14     int dim = 4;
15
16     // create matrix – must be symmetric positive definite
17     MatDense A(dim, dim);
18
19     A(0,0)=4.0;
20     A(0,1)=-1.0;
21     A(0,2)=0.0;
22     A(0,3)=0.0;
23
24     A(1,0)=-1.0;
25     A(1,1)=4.0;
26     A(1,2)=-1.0;
27     A(1,3)=0.0;
28
29     A(2,0)=0.0;
30     A(2,1)=-1.0;
31     A(2,2)=4.0;
32     A(2,3)=-1.0;
33
34     A(3,0)=0.0;
35     A(3,1)=0.0;

```

```

37  A(3,2)=-1.0;
    A(3,3)=4.0;

39  A.report(cout);

41  // create RHS
    VecPPDense b(dim);
43  b(0)=3.0;
    b(1)=2.0;
45  b(2)=2.0;
    b(3)=3.0;

47

49  // create workspace
    VecPPDense r(dim);
    VecPPDense p(dim);
51  VecPPDense ap(dim);

53  // estimated solution
    VecPPDense x(dim);

55

57  // two its of CG
    simplecg(A,b,x,r,p,ap,2);

59  // what did we get
    x.report(cout);

61  }

```

7 Using C functions to define classes and class methods

C is a subset of C++ - main issue to avoid is name-mangling. Example: MatFile adds file i/o capability to the my matrix struct example:

```

1  /*
   * Author: WWS
3  * Purpose: subclass MatDense to add file i/o capability
   */

5  #ifndef __CAAM420_MATDENSEF__
7  #define __CAAM420_MATDENSEF__

9  #include "matdense.hh"

11 extern "C" {
    #include "mio.h"
13 }

15 class MatFile: public MatDense {

17     /* here is a way to make sure that the useless
        and irritating default constructor does not
19     get called

```

```

21  */
22  MatFile ();
23  public:
24
25  /** construct dense matrix by reading file */
26  MatFile(int n, int m): MatDense(n,m) {}
27
28  /** read matrix from minfo file pair */
29  void MatFile::readMat(char const * fname);
30
31  /** write the matrix out as an minfo file pair -
32      minfo file name = fname, data file name = dname
33      (note that the latter has to be non-const, not because
34      it will be changed - it won't - but because it's being
35      passed to a non-const arg of minfo_write).
36  */
37  void writeMat(char const * fname, char * dname) const;
38
39 };
40
41 #endif

```

These use, without adaptation, the functions `minfo_read` and `minfo_write` developed to access the minfo file structure. This collection of functions solved a real problem, so why not use them rather than rewrite them? They are stand-ins for the many C library functions you might use to develop class methods.

It would be best not to recompile `mio.c` with a C++ compiler, but rather to simply use the compiled object file `mio.o` produced by a C compiler (indeed, if this really were library code you would have no alternative). However C++ name mangling gets in the way. Let's assume that `mio.h` is `#included` in a C++ driver or class definition, like so:

```
#include "mio.h"
```

just as you would in a C function definition using `minfo_read`. However the C++ compiler will produce a symbol in the driver or class object file that looks like this (output of `nm`):

```
__Z10minfo_readPiS_PcP7__sFILE
```

However the corresponding symbol in `mio.o`, produced by a C compiler, is

```
_minfo_read
```

So linkage is impossible. The elaborate C++ symbol is an example of name mangling - the C++ compiler must add a lot of information to the symbol, to reflect possible class membership and other attributes important in C++ linkage.

The solution is to turn off name mangling, as follows:

```
extern "C" {
#include "mio.h"
}
```

(see Stroustrup sec. 9.2.4, pp. 205-6 - `extern` is also a C construct, see K&R, but is mostly redundant in the C context- means "definitions are stored externally").

This example:

```

#include "matdensef.hh"
2
int main() {
4
    int nrows=3;
6    int ncols=3;
    MatFile m(nrows, ncols);
8    m.readMat("a.minfo");
    m.report(cout);
10   m.writeMat("b.minfo", "b.dat");
    };

```

uses the MatFile class methods to read a matrix from an minfo file pair, and write it out to another (both names hardwired) - in effect a file copy.

8 Source files vs. header files

Stroustrup says that putting function definitions in header files is in general to be avoided, with certain definite exceptions (Ch. 9 - a short chapter, which I recommend that you read in its entirety). He discusses the reasons for this recommendation, and what happens to various types of code included in header files.

The matrix and vector class have so far been completely implemented in headers. I've now separated all of the matrix classes into header files (.hh) containing only declarations of data and function members, and source (implementation) files containing function definitions:

- matbase.hh, matbase.cc
- matdense.hh, matdense.cc
- matdensef.hh, matdensef.cc

A general rule is: if the entire function definition fits on a line (or maybe two), then it's a wash whether it appears in the header file or in a source file. If it's longer than that, put it in a source file, if possible, to avoid having the function compiled everywhere it appears and the resulting code bloat.

In order to put a member function or class method definition, as opposed to an ordinary (stand-alone, not class-member) function definition, in a source file, you have to indicate the class membership, as the source file is outside of the class declaration scope. This is done by indicating function membership in a class by double colons: for example, in matdensef.cc

```
void MatFile::readMat(const char * fname) {
```

Note that the class membership tag MatFile:: goes next to the function name, not elsewhere. The other matrix class definition files show other examples of this construction.

9 Combining attributes via multiple inheritance: avoiding cyclic inheritance using mixins

Begin this discussion by imagining a sampled function as a vector. The function is, say, $f(x), a \leq x \leq b$. Suppose you sample $f(x)$ at point $xi = a + i * dx$, where $dx = (b - a)/N$ and $i = 0, \dots, N$. That's an $N+1$ -length vector of samples, and you could treat it as simply a vector of that length. HOWEVER, then

you would lose the ability to work with it as a sampled function, for example by calculating approximations of the derivative $f'(x)$ by divided differences. For example, you could use the approximation

$$f'(x + dx/2) = (f(x + dx) - f(x))/dx.$$

From the samples of f at $x_i, i = 0, \dots, N$, you could approximate the samples of f' at $y_i = a + (i + 1/2)dx, i = 0, \dots, N - 1$ using this formula.

This calculation requires not just the samples, but the sample rate or step dx , also the first sample abscissa a . You could group these things together in a struct and build a vector type (either C-style, like struct `vec`, or C++ style like `VecPPDense`, around it. However you can preserve "your" investment of effort in developing `VecPPDense`, for example, by creating a subclass which adds the attributes required of a grid sample vector. This is very simple, as there are only two - the initial sample a and the step dx .

The obvious way to do this is by adding these attributes, like I did for the file reading attributes in `MatFile`. We could call the class so generated `VecPPGrid0`. However, that approach has a drawback. Suppose you also want to add a feature to `VecPPDense` to read a file to initialize a vector - just like we did with `MatFile`. Call the class obtained by adding this attribute `VecPPFile` (the "0"s at the end of these two classnames indicate they are not keepers - we are going to replace them with something better). The source example directory contains implementation files with the missing function definitions. Now suppose you want to combine these attributes - read a file to create a grid of samples - you might try combining the two types by multiple inheritance, i.e. inheriting methods from both `VecPPGrid0` and `VecPPFile`. The class `VecPPGridFile0` shows how this is done. HOWEVER there is a big problem, as trying to compile Example 54 shows: the compiler claims that the request for operator() is ambiguous.

The reason for this complaint is that there are two routes to resolving this function:

- `VecPPGridFile` → `VecPPGrid0` → `VecPPDense`, and
- `VecPPGridFile0` → `VecPPFile` → `VecPPDense`.

This type of cyclic dependency presents unresolvable difficulties for the compiler, and must be outlawed.

There are two remaining solutions. The obvious one is to add the grid code to `VecPPFile0`, or to add the file code to `VecPPGrid0`. Neither is appealing: you wind up duplicating code from one of these two classes in the derived class. Having the same code in two different places is an invitation to trouble: if you find a better design for the duplicate function, you are very likely to implement the improvement in one place but forget to put it in the other.

A much better solution is to add the new attributes via a different, non-cyclic use of multiple inheritance, using so-called mix-in classes. The class `Grid1D` isolates the additional features of the grid, namely origin and step, without inheritance from `VecPPDense`. The class `VecPPGridFile` multiply inherits from `Grid1D` and from `VecPPFile`. Since `Grid1D` does not inherit from `VecPPDense`, there is now only one route for the compiler to find the `VecPPDense` functions, so no ambiguity and everything works, as Example 55 illustrates.

Note that `Grid1D` can be used in this mode (mix-in) because its information is independent of that contained in `VecPPDense`. The same cannot be said of `VecPPFile`, however - it must use the length attribute of `VecPPDense`, so cannot be independent, and is not a candidate for mix-in use.

Stroustrup's discussion of multiple inheritance (section 15.2) is one of his most involved. The first couple of pages, however, make good reading. The examples I have discussed here are typical of safe, simple use of multiple inheritance via mix-ins (`VecPPGridFile`) and intrinsically ambiguous constructions which don't work (`VecPPGridFile0`). The language supports much more elaborate multiple inheritance use patterns, as Stroustrup explains.

10 Matrices without Matrices, and dynamic casting

Now consider the relation between the the vector of samples $f = (f(xi, i = 0, N))$ and the vector of (approximate) derivative samples $f' = (f(yi, i0, , N - 1))$ computed via the centered difference formula above. The relation between these two vectors is $f' = Df$, in which D is an $(N - 1) \times N$ matrix. You can figure out what the matrix entries are - most are zero. However it does not seem a particularly good way to represent this relation. Just as there are sparse representations for vectors, there are sparse representations for matrices (indeed, they are a good deal more important). However we already have a perfectly good way to represent this operation, just it's not via a matrix. The class `Deriv1D` shows one way to do this.

The ultimate aim of this project is to solve linear systems defined by this and related "implicit" matrices, using the Conjugate Gradient and similar algorithms - these do not explicitly access matrix entries, so should be good candidates. These algorithms use `VecPPBase` objects, which raises an interesting issue: if we pass `Deriv1D::matmult` a `VecPPBase` object, how can it access the step (a `Grid1D` attribute)?

The answer is casting: the `VecPPBase` object passed to `Deriv1D::matmult` will play a role very similar to the `void*` argument in the GSL ODE solver functions. However C++ provides a superior way to avoid stupid errors in casts, via an enhanced set of cast functions. Of the several types of cast defined in C++, I am only going to discuss one (the most useful), the dynamic cast, and show only one of its two major use modes at this point (will get to the other one in the section on exception handling).

The here is how you cast an object of some type or another to an object of type `T` in C: you cast the pointers, of course, and it looks like

```
T * p = (T *)q;
```

There is no guarantee that this really succeeds, even if it appears to do so; if `q` is a `void*`, then this cast always succeeds, whether `q` points to memory holding a `T` or not. Dynamic casts actually check the type, on the other hand, and return a `NULL` pointer if the pointer being cast does not actually point to memory holding an object of the desired type. Thus you can check whether you really had a `T` or not and do something sensible in either case.

The syntax is

```
T * p = dynamic_cast < T * > (q);
if (p) {
    (do something)
}
else {
    (do something else)
}
```

The implementation file for `Deriv1D` shows how this is done. The `matmult` function dynamic-casts the input vectors to make sure that they are really `VecPPGridFile` objects, then computes the approximate derivative using the step.

Thus `Deriv1D` acts like a matrix - in fact really implements matrix multiplication - without anywhere in its data holding the entries of a matrix. Such a thing has come to be called a (linear) operator. In order to write a version of the conjugate gradient algorithm that applies to all such things, introduce an `Op` base class - exactly like `MatBase`, but without the `Mat`. `Deriv1D` derives from `Op`, and a slight rewrite of conjugate gradient iteration uses `Op` instead of `MatBase` as its base class for "matrices".

That's all very well, but now we have TWO conjugate gradient implementations, one for matrices, one for operators, that do exactly the same thing - the difference between `cg.cc` and `cgop.cc` is exactly one line!!! So that is another invitation to code chaos, where you make improvements in one version and neglect to duplicate them elsewhere.

We can solve that problem by making matrices into operators, however there are at least two ways to do that. The first would be to make `MatBase` a subclass of `Op`. This has the advantage of using inheritance in a straightforward way, but the disadvantage of forcing you to edit the code for `MatBase`. Not much of an issue for code that's just class examples, but a very big deal if the code is part of a library. For example, you might want to use the extensive capabilities of the Trilinos project libraries, and there are many other examples (beginning with `GSL!`).

Inheritance implements a base class interface via the "is" relation: a `MatDense` is a `MatBase`. I propose an implementation of the `Op` base interface by the "has" relation instead: the class `OpMat` (definition file `opmat.cc`) implements an `Op` by using a `MatBase` object, rather than being a `MatBase`. This implementation uses the facilities of `MatBase` to implement an `Op` without requiring any change in the `MatBase` code.

Since an `OpMat` is an `Op`, you can solve linear systems involving its matrix (`MatBase`) data member via the `Op`-based CG implementation, and we are back to having only one of these to worry about.

11 Const, intentional copying, and copy construction

The class example implementation of a `VecPPBase` using `gsl_vector` (class declaration file) has been augmented with a member function

```
void test() const;
```

the intent of which is simply to create another vector with the same content as the data member `gsl_vector` * (i.e. a copy), and then alter the copy.

The first attempt (commented out) simply copies the pointer data member. This doesn't compile, because the method is `const`. Therefore it must resolve the `getGSLVec` method ambiguity in favor of the `const` version, and returns a pointer to a `const gsl_vector`. This is not the same type as a pointer to a non-`const gsl_vector`, so you cannot make the assignment.

Note that this outcome is exactly correct: you have tried to access internal data in a way that would allow you to alter it, inside the body of a `const` member function. "Const" in the context of member functions means: does not alter any internal data of the object, so this access is and should be forbidden.

The second solution is the one not commented out: copy the `gsl_vector` data to a new, independent `gsl_vector`, by an assignment statement. This works - how? Answer, which should have been part of the discussion on structs: C (and C++) both define assignment of structs as bitwise copy: that is, every data member of the new `gsl_vector` `k` in

```
gsl_vector k = getGSLVec();
```

is a copy of the data members in the `gsl_vector` stored (by pointer) in the calling object.

This has a slightly unnerving consequence. The length is copied - that's fine. But the double * is also copied - that is, the data member denoted

```
double * data;
```

in `k` has the same value as the data member in the object's `gsl_vector`. Since this member is a pointer, it points to the same data! Thus if you change one of the entries in `k`, you have changed the entries in the vector data of the calling object!

This example:

```
1 #include "vecppgsl.hh"
3 int main() {
```

```

5 VecPPGSL v(10);
7 for (int i=0;i<10;i++) v(i)=0.0;
9 v.report(cout);
11 v.test();
13 v.report(cout);
15 }

```

illustrates this phenomenon, by instantiating a VecPPGSL and running the test method on it, reporting before and after on the content of the object.

Now this outcome is also proper. Altering the data by copying the gsl_vector (whose address is private data of your object) and assigning to array members of the copy, indeed respects the const nature of the test function, because the actual data member (gsl_vector *) is not altered! This may not have been what you thought "const" signified, however, and reveals one limitation of const: it means literal, rather than logical, data constancy on call of const member functions.

To achieve logical const-ness, that is, avoid altering the data "belonging to" the object (but really not - only the pointer "belongs" to the object), you must code this example differently. The last part of the test function shows one of several possible approaches. The main device is defining the copy constructor to produce a logical, rather than literal, copy. It is also possible to overload the assignment operator to achieve the same effect - this is discussed in Stroustrup.

The copy constructor of a type T has the signature

```
T(T const &);
```

If you don't define it, the compiler will - up to this point, we have not defined copy constructors, so have (whenever needed) used the compiler's definition, which (as for assignment) is literal copy of every data member. I have implemented an explicit copy constructor, which replaces the one generated by the compiler whenever supplied. This copy constructor produces a logical copy, with a new gsl_vector * data member, not a copy of the old, which then has all of its internal data copied.

Writing logical copy constructors can be tedious, as every bit of data referenced in any way by the object has to be copied - but it provides a way to avoid little code time-bombs like the one that I've described in this section. For this reason, good coding practice includes definition (or disabling) of copy constructors, in every case.

12 C and C++ strings

Recall that a C string is simply a char array with a null char ('\0') inserted somewhere; the null functions as the end of the string.

The C++ standard library provides a string class with a good deal of improved capability over the C string class. This example:

```

1 /*
2  * Author: WWS
3  * Purpose: illustrate a few properties of C++ standard strings
4  */
5

```

```

7  #include "cppstd.hh"
9
11 int main() {
13     // initialization using a literal string constant
15     string f = "fruitcake";
17
19     // write to stream – note standard library end-of-line
21     // struct
23     cout<<f<<endl;
25
27     // add another string
29     string g = " cook";
31     string h=f+g;
33     cout<<h<<endl;
35
37     // copy a string
39     string k = g;
41     cout<<k<<endl;
43
45     // add a string literal
47     h+="ed";
49     cout<<h<<endl;
51
53     // index into the string
55     cout<<"char 6 = "<<h[6]<<endl;
57
59     // print the size
61     cout<<"size of h = "<<h.size()<<endl;
63
65     // extract the C string hiding inside the C++ string –
67     // note that c_str() returns a const char *.
69     printf("%s\n",h.c_str());
71
73     // use some analogous C functions
75     char * cs = new char[h.size()+1];
77     strcpy(cs,h.c_str());
79     printf("%s\n",cs);
81     printf("size of h = %d\n",strlen(cs));
83
85     // add another string using C string library
87     char m[1000]; // surely enough
89     strcpy(m,cs);
91     strcat(m," to perfection");
93     printf("%s\n",m);
95
97 }

```

shows some of the most useful features of this class:

- string construction from literal strings ("...")
- building up strings from other strings via the overloaded "+" operator

- indexing into strings
- extracting a (const) C string via the "c_str()" function

The example also shows how to do most of the same things with C strings. There is clearly nothing you can do with the one that you can't do with the other, but the C++ string class allows you to perform string manipulations with simpler and easier-to-understand code.

Stroustrup devotes an entire chapter (20) to the string class, from which one could well conclude that I have shown you just a few features of this type.

13 Templates and Generic Programming

So far I have focused on abstraction via inheritance, in which you "hide" the characteristics of a type behind the virtual interface of a base or parent class. This type of programming goes under the name "object-oriented", and there has been a lot of discussion about whether C++ is really object-oriented. In fact, it is not - as it contains C, which is not. The programming style supported by C is called "procedural".

C++ supports a third programming style, called "generic", which combines well with the other two to allow very effective application construction. Generic programming permits type information to be passed directly through function or class interfaces, and allows object behaviour implicitly defined by these types. C++ implements this concept via function and class templates. A simple function template looks like this:

```
template < typename T > void foo(T t) {      (do something with t that makes sense for a T); }
```

This example:

```
#include "cppstd.hh"
2
3 template<typename T>
4 T addme(T a, T b) {
5     return a+b;
6 }

8 int main() {
9     double da=1.0;
10    double db=2.0;
11    cout<<"sum = "<<addme(da,db)<<endl;

12
13    float fa=1.0;
14    float fb=2.0;
15    cout<<"sum = "<<addme(fa,fb)<<endl;

16
17    int ia=1;
18    int ib=2;
19    cout<<"sum = "<<addme(ia,ib)<<endl;

20
21    /*
22    char a[10];
23    char b[10];
24    strcpy(a,"fruitcake");
25    strcpy(b,"fruitcake");
26    cout<<"sum = "<<addme(a,b)<<endl;
27    */
```

is an extremely simple example. It defines a function template, which adds two "things" - this works as long as the "things" can be added. Uncomment the last block to see what happens when the addition does not make sense.

Explanation: the types are resolved at compilation. That is, when the compiler has information on the types used in the template, it compiles the template code, and not until. So the code does not always have to "make sense": it need only be legal when the types actually used in the program are supplied.

A side-effect of this design is that templates themselves cannot be independently compiled: they lack essential type information. Therefore they must have their definitions included in another code fragment in which the type ambiguities inherent in their construction are resolved. This is new: a template is neither a declaration nor a definition, but something in-between: a definition with missing type info.

For this reason, it is conventional to include template code in header files.

You have already seen templates in action: `dynamic_cast` is a function template.

Class templates are similar to function templates. A natural example of a class template is the class `vectbase.hh`, which frees up the scalar field type in the vector base class - it need no longer be double, yet the same code is used throughout, even in the linear algebra methods.

I've implemented templated versions of the dense and sparse vector classes discussed earlier:

Listing 6: Dense Templated Vector

```

1 #ifndef __CAAM420_VECTDENSE__
2 #define __CAAM420_VECTDENSE__

4 #include "vectbase.hh"

6 /*
7  * Author: WWS
8  *
9  * Purpose: templated dense vector class
10  */

12 // note: outside of class declaration, must refer to class by
13 // full name - includes template!
14 template<typename Scalar>
15 class VecTDense: public VecTBase<Scalar> {
16
17 private:
18
19     int len;
20     Scalar * vdata;

22     // disabled default and copy constructors
23     VecTDense();
24     VecTDense(VecTDense<Scalar> const &);

26 public:

28     /* main constructor - note use of default arg */
29     VecTDense(int _len, bool init=false): VecTBase<Scalar>() {

```

```

30     len=_len;
        vdata=new Scalar [len];
32     if (init) {
        for (int i=0;i<len;i++) vdata[i]=0.0;
34     }
    }
36
    /* destructor – note that call to delete is safe. At moment contemplate
38     no subclasses, so need not be virtual.
    */
40     ~VecTDense() {
        if (vdata) delete [] vdata;
42     }

44     int getLength() const { return len; }

46     Scalar & operator()(int i) {
        if (vdata && i >= 0 && i <= len-1) return vdata[i];
48         else {
            cerr<<" Error: VecTDense::operator() (non-const)\n";
50             cerr<<" attempted access failed – either not initialized or \n";
            cerr<<" out of range\n";
52             /* Will see a better solution to this problem */
            exit(1);
54         }
    }
56
    Scalar const & operator()(int i) const {
58         if (vdata && i >= 0 && i <= len-1) return vdata[i];
        else {
60             cerr<<" Error: VecTDense::operator() (non-const)\n";
            cerr<<" attempted access failed – either not initialized or \n";
62             cerr<<" out of range\n";
            /* Will see a better solution to this problem */
64             exit(1);
        }
66     }

68     /* mandatory report function */
    virtual ostream & report(ostream & str) const {
70         str<<" VecTDense object: length="<<len<<" \n";
        for (int i=0;i<len;i++) {
72             str<<" v["<<i<<"]="<<vdata[i]<<" \n";
        }
74         return str;
    }
76
};
78
#endif

```

Listing 7: Sparse Templated Vector

```
#ifndef __CAAM420_VECTSPARSE__
```

```

2 #define __CAAM420_VECTSPARSE__
4 #include "vectbase.hh"
6 template<typename Scalar>
  class VecTSparse : public VecTBase<Scalar> {
8
  private:
10   int len;
11   int nnz;
12   int *indices;
13   Scalar *data;
14   Scalar zip;
16   VecTSparse();
17   VecTSparse(VecTSparse<Scalar> const &);
18
  public:
20
  // main constructor for empty sparse vector
21   VecTSparse(int in_dim) : VecTBase<Scalar>() {
22     len=in_dim;
23     nnz = 0;
24     indices = NULL;
25     data = NULL;
26     zip=0.0;
27   }
28
  // destructor
29   ~VecTSparse() {
30     if (data) delete [] data;
31   }
32
  int getLength() const { return len; }
34
  // this returns a mutable (assignable) reference
35   // references have to refer to something that
36   // exists – so construct entry if not already constructed
37   // morph of implementation of set function
38   // void set(int index, Scalar val){
39   // in last week's class
40   // checks for existing entry and
41   // extends storage for new entries
42   Scalar & operator()(int index) {
43
44     /****** should put index check here!!! */
45     int i;
46
47     for(i=0;i<nnz;++i){
48       if(indices[i] == index){
49         return data[i];
50       }
51     }
52   }
53 }
54

```

```

56 // otherwise extend list by one
57 // oops - realloc is not appropriate here
58 ++nnz;
59 Scalar *tmpdata = new Scalar [nnz];
60 for(i=0;i<nnz-1;++i) tmpdata[i] = data[i];
61
62 delete [] data;
63 data = tmpdata;
64
65 // oops need to realloc indices
66 int *tmpindices = new int [nnz];
67 for(i=0;i<nnz-1;++i) tmpindices[i] = indices[i];
68 tmpindices[i] = index;
69
70 delete [] indices;
71 indices = tmpindices;
72
73 return data [nnz-1];
74 }
75
76 // implementation of operator () const
77 Scalar const & operator()(int index) const {
78
79     for(int i=0;i<nnz;++i){
80         if(indices[i] == index){
81             return data[i];
82         }
83     }
84
85     return zip;
86 }
87
88 // reimplement of zero operator - minimal storage (none!)
89 // note that there is NO MENTION of Scalar 0.0! That's implicit
90 // in the indexing operator implementation - and these operators
91 // provide the only access to the data vector!!!
92 void zero() {
93     nnz=0;
94     if (data) delete [] data;
95     data=NULL;
96 }
97
98 // output stored entries
99 ostream & report(ostream & str) const {
100     for(int i=0;i<nnz;++i)
101         str<<"nnz: "<<i<<" index: "<<indices[i]<<" val: "<<data[i]<<"\n";
102     return str;
103 }
104 };
105
106 #endif

```

This example shows how these can be used:

```
1 /*
2  * Author: WWS
3  *
4  * Purpose: extremely simple run-around-the-block for
5  * very simple templated array storage class – after ex40.c
6  *
7  */
8
9 #include "vectdense.hh"
10 #include "vectsparse.hh"
11
12 int main(int argc, char ** argv) {
13     int n=10;           // vector size
14
15     cout<<" create a VecT<float>\n";
16     VecTDense<float> v(n);
17     v.report(cout);
18
19     cout<<" fill it with floats\n";
20     for (int i=0;i<v.getLength();i++)
21         v(i)=(float)i;
22
23     cout<<"Dense Vector Report:\n";
24     v.report(cout);
25     cout<<"\n";
26
27     cout<<" instantiate VecTSparse<double object\n";
28     VecTSparse<double> w(n);
29
30     cout<<" assign element 5, uses non-const index op\n";
31     w(5)=1.0;
32
33     cout<<" Sparse Vector Report:\n";
34     w.report(cout);
35
36     cout<<" get element 6 = "<<w(6)<<" – also invokes non-const op(),
37     since object was not declared const\n";
38
39     cout<<" report again\n";
40
41     w.report(cout);
42
43     cout<<" declare const reference to w\n";
44     VecTSparse<double> const & z = w;
45
46     /*
47     cout<<"attempt to assign, thus invoking non-const operator()\n";
48     z(1)=2.0;
49     */
50
51     cout<<" access element of w through const reference – no change\n";
```

```

53  cout<<"in w's storage\n";
    cout<<"w(7)="<<z(7)<<"\n";
55  w.report(cout);

57  // this does not work - discuss
    // cout<<"dot product = "<<v.dot(w)<<" only invokes const op()\n";
59
    exit(0);
61 }

```

- note that both float vectors and double vectors appear in this program. However the template parameter is part of the class name, so you cannot mix them!

As we shall see, with a little tinkering this class template even accommodates complex vectors.

14 Template specialization and interface to non-generic code

Suppose you want, for some reason, to print various real numbers, represented as floats or doubles, using C stdio, with a single function. The C++ stream library does all type conversion for you, but you also might want to use varying levels of precision, reflecting the amount of bits in the different types. How would you do this via printf?

You could try to write a templated function, but C does not know about templates. You really need a way to use the template type as a switch, to turn on and off the various options for field width, precision, etc.

C++ provides the ability to use types as switches by template specialization. The idea is to provide a general implementation for any type T

```
template < typename T > void foo(T & t) {... do something}
```

supplemented with instructions for specific types:

```
template < > void foo < S > (S & s) {... do something special for s}
```

The compiler chooses the most special implementation. So if it sees

```
foo < S >(s);
```

it is obligated to insert the second implementation of foo (for S), rather than the general implementation.

I've used this idea to write a function template that prints doubles and floats in different formats. The driver shows how it works.

Note that at times there may be no sensible implementation for a general template type - I've decided that my `print_with_C` function has this spec, i.e. it is supposed to work only for real types. It's possible to make this work - since the compiler inserts the template code only when the template types are resolved, you can simply generate a piece of code that won't compile and use it for the implementation of the general case.

15 Complex numbers in C++

The C++ standard library (note - not part of the language per se, any more than the C standard library is part of C!) includes a complex type. The discussion on pp. 679-681 is comprehensive; the gist is this:

- You can template the complex type on any scalar type supporting the usual arithmetic operations, although the usual choices are float and double;

- complex `ı T ı` has all of the standard arithmetic operations of type `T` overloaded, also functions `real`, `imag`, and `abs`;
- it's very handy, but incompatible with the complex types in Fortran and C libraries like `GSL`, which complicates using it in conjunction with outside code.

16 Exceptions

Reading: Stroustrup Ch. 14. As usual, way more than you want to know, but it's all there, along with a great quote from Churchill.

Exceptions are C++'s method for systematic handling of program errors, a method basically adopted wholesale by the designers of Java. Look at the dot product method in `VecTBase`:

```
virtual Scalar dot(VecTBase < Scalar > const & x) const {
    if (x.getLength() != this->getLength()) {
        (blah blah blah)
        exit(1);
    }
    Scalar d=0.0;
    for (int i=0;i < getLength();i++) d += ((*this)(i))*(x(i));
    return d;
}
```

Basically, there is nothing wrong with this - if you try to take the dot product of two vectors of different lengths, you probably have made a fatal error. However - how can the dot product function itself make that decision? It really has no access to the context, and cannot possibly "know" whether stopping the program is the only possible response. Even worse, YOU have no access to the context: you will know what error occurred, but not why, nor where - that is, what part of a program called the dot product with the wrong inputs? The classic remedy is to return an int which may encode both the error state (zero for normal return, nonzero for an error condition) and encode the actual error in the value of the int returned. This would solve our problem, but is impossible in this context, as the return value is the dot product, and there is no way to replace it with an error flag. You could replace the function signature with one that allowed an int return, but that could be inconvenient for other reasons.

The exception handling mechanism gets around all of these issues by providing an alternate flow control route. The mechanism looks like this:

```
virtual Scalar dot(VecTBase < Scalar > const & x) const {
    if (x.getLength() != this->getLength()) {
        MyDotException e;
        throw e;
    }
    ...
}
```

An exception class is simply a class used for passing exception information - there is a standard library exception class with several subclasses, reviewed below, but it's usually necessary to create your own. I'll review how this is done below.

The `throw` function (which is separated from its argument by a space, notice) invokes an alternate return route to the calling function. Control continues to return to calling functions until the exception is caught by a `catch` command inside a `try-catch` block (or "try block"). For example,

```

void dispdot(VecTBase < float > const & x, VecTBase < float > const & y) {
    try {
        cout<<"dot of two vectors = "<<
    }
    catch (MyDotException & e) {
        cout<<"could not compute dot product - vectors incompatible\n";
    }
}

```

This example illustrates a situation in which `exit(1)` is not necessarily the right response to an error - perhaps it isn't really an error.

If an exception is not caught at the calling function, then control continues to return - if no catch statements are encountered, all the way to `main()`, where an exception causes an abort statement to be printed and `exit` to be called (what else can `main()` do?).

Note that `catch` takes a reference argument, which can be of any type - exception types are really defined by their use, not by their structure. That said, some uses are typical. One particular use is to record some information about the place where the exception occurred, or was caught, or anything else that might be useful. I provide a CAAM 420 exception class which has this "recording" capability:

Listing 8: CAAM420 exception class

```

1 /*****
3 Copyright Rice University, 2004–2010.
4 All rights reserved.
5
6 Permission is hereby granted, free of charge, to any person obtaining a
7 copy of this software and associated documentation files (the "Software"),
8 to deal in the Software without restriction, including without limitation
9 the rights to use, copy, modify, merge, publish, distribute, and/or sell
10 copies of the Software, and to permit persons to whom the Software is
11 furnished to do so, provided that the above copyright notice(s) and this
12 permission notice appear in all copies of the Software and that both the
13 above copyright notice(s) and this permission notice appear in supporting
14 documentation.
15
16 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY
19 RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS
20 NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL
21 DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
22 PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
23 ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
24 THIS SOFTWARE.
25
26 Except as contained in this notice, the name of a copyright holder shall
27 not be used in advertising or otherwise to promote the sale, use or other
28 dealings in this Software without prior written authorization of the
29 copyright holder.
30
31 *****/

```

```

33 /*
   * Author: WWS
35 * CAAM 420 exception class – mild morph of RVL::RVLEException
   *
37 * Uses std::string to build up exception messages.
   */
39
40 #ifndef _CAAM420_EXCEPT
41 #define _CAAM420_EXCEPT
42
43 #define BUFLLEN 100
44
45 #include "cppstd.hh"
46
47 /** An implementation of the std::exception interface, with additional
   * methods so it can be used more like a output stream.
49 */
50
51 class CAAM420Exception: public std::exception {
52
53 private:
54     string msg;
55
56 public:
57
58     CAAM420Exception(): msg("") {}
59     CAAM420Exception(const CAAM420Exception & s): msg("") { msg += s.msg; }
60     virtual ~CAAM420Exception() throw() {}
61
62     // report function – from std::exception
63     const char* what() const throw() { return msg.c_str(); }
64
65     CAAM420Exception & operator<< ( string str ) {
66         msg += str;
67         return *this;
68     }
69     CAAM420Exception & operator<< ( const char* str ) {
70         msg += str;
71         return *this;
72     }
73     CAAM420Exception & operator<< ( int i ) {
74         char buf[ BUFLLEN ];
75         sprintf( buf, "%d", i );
76         msg += buf;
77         return *this;
78     }
79     CAAM420Exception & operator<< ( unsigned int i ) {
80         char buf[ BUFLLEN ];
81         sprintf( buf, "%u", i );
82         msg += buf;
83         return *this;
84     }
85     CAAM420Exception & operator<< ( long i ) {

```

```

87     char buf[ BUFLLEN ];
    sprintf( buf, "%ld", i );
    msg += buf;
89     return *this;
}
91 CAAM420Exception & operator<< ( unsigned long i ) {
    char buf[ BUFLLEN ];
93     sprintf( buf, "%lu", i );
    msg += buf;
95     return *this;
}
97 CAAM420Exception & operator<< ( short i ) {
    char buf[ BUFLLEN ];
99     sprintf( buf, "%d", i );
    msg += buf;
101    return *this;
}
103 CAAM420Exception & operator<< ( unsigned short i ) {
    char buf[ BUFLLEN ];
105     sprintf( buf, "%d", i );
    msg += buf;
107    return *this;
}
109 CAAM420Exception & operator<< ( double d ) {
    char buf[ BUFLLEN ];
111     sprintf( buf, "%g", d );
    msg += buf;
113    return *this;
}
115 CAAM420Exception & operator<< ( float d ) {
    char buf[ BUFLLEN ];
117     sprintf( buf, "%g", d );
    msg += buf;
119    return *this;
}
121 template<class T>
CAAM420Exception & operator<< ( complex<T> d ) {
123     char buf[ BUFLLEN ];
    sprintf( buf, "(%g,%g)", d.real(),d.imag() );
125     msg += buf;
    return *this;
127 }

129 CAAM420Exception & operator<< ( char c ) {
    char buf[ BUFLLEN ];
131     buf[ 0 ] = c;
    buf[ 1 ] = '\\0';
133     msg += buf;
    return *this;
135 }

137 };

```

139

141 *#endif*

Exceptions are also the basis to the second major usage of dynamic cast:

```

try {
    foo & x = dynamic_cast < foo & > ( y );
}
catch (bad_cast) {
    (respond somehow, maybe throw another exception)
}

```

In this code snippet, if y actually is a reference to a foo object, then the cast succeeds, and the code in the try {} block gets executed. If not, then the bad_cast standard exception gets thrown, and control passes to the catch {} block. This form of dynamic casting is often more convenient than the pointer form - the same goal can be accomplished with by casting addresses and a conditional branch, but this is more concise.

17 The Standard Library’s templated container classes

C++, like C, comes supplied with a Standard Library. The header files listed in cppstd.hh:

Listing 9: cppstd.hh

```

1 /*****
3 Copyright Rice University, 2004, 2005, 2006.
4 All rights reserved.
5
6 Permission is hereby granted, free of charge, to any person obtaining a
7 copy of this software and associated documentation files (the "Software"),
8 to deal in the Software without restriction, including without limitation
9 the rights to use, copy, modify, merge, publish, distribute, and/or sell
10 copies of the Software, and to permit persons to whom the Software is
11 furnished to do so, provided that the above copyright notice(s) and this
12 permission notice appear in all copies of the Software and that both the
13 above copyright notice(s) and this permission notice appear in supporting
14 documentation.
15
16 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY
19 RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS
20 NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL
21 DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
22 PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
23 ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
24 THIS SOFTWARE.
25
26 Except as contained in this notice, the name of a copyright holder shall
27 not be used in advertising or otherwise to promote the sale, use or other
28 dealings in this Software without prior written authorization of the
29 copyright holder.

```

```

31 *****/
   #ifndef __RVL_STD_CPP_INCLUDES_H_
33 #define __RVL_STD_CPP_INCLUDES_H_

35 // C includes

37 #include "cstd.h"

39 // C++ includes

41 #include <iostream>
   #include <fstream>
43 #include <sstream>
   #include <iomanip>
45 #include <limits>
   #include <exception>
47 #include <stdexcept>
   #include <typeinfo>
49 #include <string>
   #include <map>
51 #include <list>
   #include <vector>
53 #include <valarray>
   #include <complex>

55   using namespace std;
57   using std::cout;
   using std::cerr;
59   using std::ifstream;
   using std::map;
61   using std::vector;
   using std::string;
63   using std::ios_base;

65 #endif

```

include those needed to access the library; no additional information need be supplied to the linker, as the installation of the compiler includes the locations of the required object libraries.

The library includes (since the late 90's) a collection of templated container classes, described in chapters 16 and 17 of Stroustrup's book: vectors, lists, trees, and so on. The string class introduced earlier is actually part of this collection. The intent of these classes is to provide very rapid access to homogeneous collections of objects, in a more flexible form than the intrinsic array. To enable traversing these various types of containers in a uniform way, the library provides an abstraction for loop, the iterator, which gives an implementation of "start at the beginning, go until the end" that works in the same way across all types of containers.

The standard container classes originated in the earlier standalone Standard Template Library, a project by a group of programmers at HP. You'll still see the standard containers called "stl containers" for this reason.

I'm not going to go deeply into the standard container collection - in fact, we will look just at the vector container (Stroustrup, 16.3). This is a class template providing access to arrays of anything, with these features:

- You can build a vector $\text{vector} < T >$ provided that T has a working assignment operator with the usual behaviour (every builtin type does - classes must define it) and a "deep" copy constructor, that is, one

that creates an independent copy of the object being copied - as you know, the default copy constructor provided by the compiler often does not do that).

- You can access elements by either an overload of the standard index operator [int i], or by the at operator:

```
std::vector < T > x;  
(initialize x somehow)  
T y = x.at(10);
```

The at operator is bounds-checked, the [] operator is not.

You can use either operator as an l-value, i.e. as the left-hand side in an assignment:

```
y[i]=10;
```

- You instantiate a vector with a given length by providing it to the constructor:

```
vector < T > x(10);
```

But vectors are extensible: you can add elements by pushing them to the back - that is, vectors have stack attributes (and in fact contain an implementation of the stack data structure - see Stroustrup ch. 16).

```
vector < int > x; // vector with length 0  
for (int i=0;i < 10; i++) {  
    x.push_back(i);  
}
```

Here is a very good concise summary of this class:

CPlusPlus site

It's irresistible to implement a VectBase subclass using std::vector.

18 Project organization: building your own libraries

By this point in the class you should have the notion firmly in mind that object libraries are useful - in fact they are the main mechanism for making reusable code available at the object level. However all of the libraries we have encountered so far have arrived in a finished state. It is also useful to build your own libraries, for the same reasons that make GSL, LAPACK, and many other scientific computing libraries useful.

Libraries come in two flavors: statically linked and dynamically linked. I'm going to ignore the second type - see van der Linden's book for an excellent discussion, also earlier class notes.

A statically linked library ("library" from here on) is technically an archive, produced by the Unix archiving command ar. Archives are named with the suffix a: thus mylib.a. The ar command takes a number of arguments:

```
ar [options] [library path] [list of files to be added or deleted, if appropriate]
```

The man page is informative - look at it. The key options are

- -t - list the contents of the archive (no file list)
- -r - add or replace files in list
- -d - delete files in list
- -c - do it silently
- -v - don't do it silently

It is mandatory to select some option - the command is meaningless with out at least one. For library building, the r command is the useful one - in a makefile, you would usually want the library build to proceed quietly, so you would add the c option. Note that the dash is redundant - "ar cr" is equivalent to "ar -c -r". So a typical library building command found in a makefile might read

```
ar cr mylib.a a.o b.o c.o
```

This command will create mylib.a if it does not exist, and add the object files a.o, b.o, c.o,... to it.

One good use of a library is to organize all of the object files in a project so that all of the commands built in the project can easily link to them. The linker will take only those objects from the library that are needed in each command, so there is no penalty in linking the entire library every time - therefore you won't have to worry about which commands need which object files!

Once built, the library is not yet ready for use - it needs a table of contents, in a form that the linker can recognize. The ranlib command creates the table of contents - it takes a single argument, the library pathname.

This makefile shows a typical use of ar and ranlib to build a library (in this case containing just one object).

This example makefile is part of a small project directory:

```
LIB = ../lib/mylib.a

INCS = -I../include

OBJS = dispDot.o

default: lib

%.o: %.cc
g++ -c $(INCS) $<

lib: $(OBJS)
ar cr $(LIB) $(OBJS); ranlib $(LIB)

clean:
rm -f *.o
```

This directory has the typical structure of a Unix project, with standard subdirectory names:

- include - header files
- src - source (implementation) files for non-command objects (no main(s))
- main - source for main program files (with main() functions), also location of executable commands
- lib - location of library (or libraries) produced by project

- doc - documentation files

This is far from the only possible organization. The directory names are quite commonplace, but many software packages separate "configure", "build" and "install" processes and their outputs, whereas the simple structure I've modeled here does not. This simple organization will simplify the organization of your own project. More sophisticated software maintenance methods are appropriate if you become deeply involved in producing software for use by others.

I've written makefiles for src (builds library) and main (builds command). We will talk about an overall package makefile in class.

19 Namespaces

Namespaces solve the following problem: many implementations of the same concept are possible, and would naturally carry the same names - so how is the compiler to distinguish them?

For example, the vector concept has many implementations in the examples produced for this class. The C++ standard library vector is one of these. However you might also produce a vector class and want to call it "vector". The compiler, faced with ambiguous definitions, will simply give up.

Namespaces are simply a name extension device which resolves ambiguities like these. I've written a very simple example (header and source) to illustrate the basic use of namespaces:

Listing 10: namespace header

```
1 #ifndef __CAAM420_EX67__
2 #define __CAAM420_EX67__
3
4
5 #include "cppstd.hh"
6
7 namespace A {
8
9     class foo {
10     public:
11         void what() {
12             cout<<"foo"<<endl;
13         }
14     };
15 }
16
17 namespace B {
18
19     template<typename T> class foo {
20     public:
21         void what(T t) {
22             cout<<"bar="<<t<<endl;
23         }
24     };
25 }
26
27 }
28
29 #endif
```

Listing 11: namespace source

```
1 #include "ex67.hh"
3 namespace C {
5   class bar {
7     public:
9       void what() {
10        //      using namespace A;
11        using A::foo;
12        cout<<"foobar with a ";
13        foo a;
14        a.what();
15      }
16    };
17 }
19 int main() {
21   using A::foo;
22   using B::foo;
23   foo a;
24   foo<int> b;
25   C::bar c;
27   a.what();
28   b.what(1);
29   c.what();
31 }
```

The standard library is entirely contained in namespace `std`. You will notice that `cppstd.hh` includes using statements: these make namespace contents available to the files in which they are included. This explains why you have been able to write `"cout"` rather than `"std::cout"`.