Exam info by email on Wed. (tomorrow)

General format of the exams:

1/3 short answer

1/3 code reading

1/3 code writing

I will make a "practice exam" available with key. (Wed.)

Parameter Passing Mechanisms:

C++ has two parameter passing methods: pass by value and pass by reference.

Pass by value is what we have been using so far. When you call a function, a copy of the value of the argument (actual parameter) is copied into the "in box" of the function. It's effectively assigned to the formal parameter as though to a variable.

```
double abs (double x) {
  if (x < 0) { x = -x; }
  return x;
}


….
   cout << abs(z) << endl;
…
```

The value of the actual parameter z is copied into (assigned) the formal parameter x in abs. Notice that the above definition of abs does not change the value of z.

That is often exactly what we want: the parameters are used for input to a function, not output from it.

There are times when we want a function to be able to change the values of its parameters. Suppose we wanted a function increment() defined so that increment (Z) adds one to Z.

```
// incorrect definition of increment
void increment (int x) {
  x = x+1;
  cout << x << endl;
}
```

But if I try the following:

```
int Z = 10;
increment (Z);
cout << Z << endl; /* even through x changed, Z is unchanged because I copied
the value of Z into x. */
```

So what I need to do is to give increment a reference to the variable I want to change. I do this with pass by reference. The terms reference, pointer, and address are all almost synonymous, and all refer to the name of a location in memory.

```
// correct definition of increment
void increment (int& x) { // the ampersand is all I need
  x = x+1;
}
```

The ampersand tells the compiler that when it sees an invocation like increment(Z), it shouldn't pass the value of Z, but instead it should pass a reference to Z (the address of Z). The reference is the thing that allows me to go to a memory location to read it or write it.

So you should think about whether your parameters are used only for input to the function or whether they might be used for output from the function. If they are going to be used for output, they have to be passed by reference.

Reference parameters do create a problem of aliases = having more than one name for the same memory location. This can make programs hard to read and debug. There are several paths to the same location, and it might not be obvious that they refer to the same location. This is usually not to big problem for reference parameters (but C++ does have a "reference variable" mechanism which allows explicit creation of aliases, but this is not a good idea). But beware, that if a parameter is passed by reference, the actual might get clobbered, and you might be unaware of that.

Since references (addresses) are relatively small, but some data values may be very big, many programmers pass parameters by reference to save the time of copying a large value. A very common programming practice. It should be documented, so that the reader knows that the function doesn't modify the input. (There is a way in C++ to deal with this problem: "constant reference parameters"; we will get to it later).

In summary, there are two reasons to pass a parameter by reference:
(1) the function is going to modify the actual parameter.
(2) it may be significantly more efficient (in computer time).

Structures:

In all programming languages there are simple (non-composite) data types and composite data types.
Simple data types do not have parts or components that the programmer can see.
Example of simple data types include ints, chars, doubles, bools.
Composite data types have parts that we access. Examples of composite data types are strings and vectors. The components of a string are chars, and they accessible by indexing.

```
vector<double> V;
```

V is a composite data structure whose components are doubles, and accessible by indexing.

```
vector< vector<double> > M; // 2D matrix
```

M is a vector whose components are vectors of doubles. So composite data types can have other composite types inside them.

The above (strings, vectors) are examples of homogeneous composite data types, because all their components are the same type (chars, or what ever the base type of the vector is).
This allows the compiler to calculate the memory location of any element from its index, because all the components are the same size.

Sometimes we want heterogeneous data structures, in which the components might not all the same type. So we can't calculate the positions of components, so we give them names.

The principal ways of doing this in C++ are structs and classes. They are almost identical.

Example: An Employee Database

Suppose we want a database of all a company's employees.
For employee, we want their name, emp. number, salary, and some other information, of various types.

We will define a struct "employee." And for the whole company we might have a vector or file of all the employees. I'll look at just a few.

Design approach: Think about the real world objects (employees), and what are the attributes or characteristics of them that should be in the database.

A struct is like a class of similar objects, which are called instances of the class or struct. The variables within a struct are called member variables, and each instance has its own set of member variables. A struct can also have member functions, which operate on a particular instance.

In object-oriented programming, the instances of classes or structs are called objects. So in the example, me, boss, and newHire are all objects that are instances of the struct employee. They all have their own name, empNum, dateOfHire, etc., which are their member variables (fields, attributes). And they can have member functions (behaviors, methods).