Next time: TCS 13.

We've talked about modules as self-contained parts of a system with clear interfaces and well-defined purposes or functions. Modules can contain other modules, until you get to the basic building blocks (C++ statements).

So we want to be clear about the difference between the interfaces (what goes in and what comes out) and the implementation (what goes on inside).

One way this distinction is made in C++ is the difference between declarations and definitions. To declare the printTime function (from 11.2), we write:

```
void printTime (const Time& time);
```

This tells us the inputs expected by printTime and the type of thing it returns. To define the printTime function, we have to say how it is implemented:

```
void printTime (const Time& time) {
  cout << time.hour << ":" << time.minute << ":" << time.second << endl;
}
```

Aside from improving readability and comprehensibility, separating the declarations (interface) and definitions (implementation) allows separate compilation. So for example, the definition of the Time structure and the definition of the main function can be compiled separately. For large modules this can save a lot of time. In particular, if I've changed only the main function, I don't have to recompile Time. And if I've changed only Time, I don't have to recompile main.

So to avoid recompiling Time every time I change main, what I do is put the definitions in another file (Time.cpp, the implementation file). But the main program still needs to know the interface to Time, so I could include the declaration of Time in the main program. More commonly, however, I would put the declarations (interface) in a separate header file (Time.h).

When a program has several separately compiled modules, the final step is to link the compiled versions (the object code files) together. So normally what you do is just recompile what has been changed, and then link all the compiled parts together.

A make file tells how to recompile a program by specifying all the parts it needs.