Exam II will be postponed for 1 week.

Functional vs. Object-oriented Ways of Manipulating Data:

In the functional approach, we pass data to functions or operators, which do something with the data, and may return some data as a result. Within in the function, we may have to get access to the "parts" (member variables) of the data. Think of the data as an envelope that the function has to open to do its job.

In the functional way, we tend to think of data as static or passive, and the functions or operators as the active parts of the program, which do something with the data.

In the object-oriented approach we think of the data as being active, that is, as objects that can do things, rather than have things done to them. For example, we ask the robots to turn, stop, return their name, change their name, return a sensor value, etc.

To do this in C++, we make the functions part of the object; we put them inside the object.

Function-oriented:

```
struct Time {
    int hour, minute;
    double second;
};

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

Notice that the printTime function is outside the object, so we have to tell it which Time to print (with a parameter).

In the object-oriented (OO) approach, the function is inside the object:

```
struct Time {
    int hour, minute;
    double second;

    void printTime () {
        cout << hour << ":" << minute << ":" << second << endl;
    }
};
```

Then I can ask a Time to print itself:

```
Time T = {9, 23, 0};
T.printTime();
```

Terminology:

In the OO case the printTime function is called a member function (or method) because it's a member or part of the Time, just like the member variables hour, minute, second.

A member function has implicit access to the member variables and other member functions of the structure of which it's a member.

In the functional case, where the function is not part of the structure, we speak of a standalone or free-standing function, or non-member function.

For binary (two argument functions), we have to decide which is more readable and understandable, to treat the function as standalone or as a member function.

Standalone functions:

```
    after (T1, T2)
    addTime (T1, T2)
```

Member functions:

```
    T1.after(T2)
    T1.addTime(T2)
```

Information Hiding Principles:

(1) The user of a module should have all the information needed to use the module correctly, and nothing more.

(2) The implementer of a module should have all the information needed to implement the module correctly, and nothing more.

One implication is that the user of a module should not need to know how it's implemented.
The other implication is that the implementer of a module should not need to know how it's going to be used.

A system designed according to the Information Hiding Principles is well-modularized, and therefore easier to understand and maintain.

The interface defines in effect a contract between the user and the implementer. It's the specifications that the implementation must meet and that the user can depend on.

In C++ the interface is called a declaration, and the implementation is called a definition. We can separate the declaration of a function from its definition.

// declaration of after:

```
bool after (const Time& time2) const;
```

This tells you the inputs and outputs (the interface).

//definition of after:

```
bool after (const Time& time2) const {
    if (hour > time2.hour) return true;
    if (hour < time2.hour) return false;
    // hours are the same
    if (minute > time2.minute) return true;
    if (minute < time2.minute) return false;
    // minutes are the same
    return (second > time2.second);
}
```

The above implements the after function (tells you how to do it).

Header Files and Separate Compilation:

In C++ we can separate the declaration of a structure from the definition of its member functions:

```
// declaration of Time
struct Time {
	int hour, minute;
	double second;

	void print (); // declaration of print()
};

// definition of print()
void Time::print () {
	cout << hour << ":" << minute << ":" << second << endl;
};
```

The Time:: qualifier says that the print() function I am defining is the print() that is a member of Time (and not some other print() function).

The above division facilitates separate compilation:
(1) Put the <u>declaration</u> of Time in Time.h. This file is the <u>interface</u> for Time.
(2) Put the <u>definitions</u> of the Time member functions in Time.cpp, but make sure to #include "Time.h" so that the compiler knows what Time is, along with its member variables. This file is the <u>implementation</u> of Time.
(3) In the main program (T11-files.cpp), #include "Time.h" so that you can use the Time structure and its member functions and variables (i.e., use the interface).

Time.cpp and T11-files.cpp can be compiled separately and linked together. If only one is modified, the other does not have to be recompiled.