1. Overview
This course tries to teach you how to program a computer. It will not succeed. Programming is as much of an art as a science, and has to be learnt through experience and science. One quarter will not be enough to make a novice into a programmer.

In this quarter, we will learn programming from the bottom up, that is, we will start with the computer, discuss how it is programmed, and then move from there to the art of programming a computer. The alternative is to take a logic-based approached, that is, worry about what computing is, and then conceptualize the general process to the writing of actual programs.

2. Data Transmission
Modern computers were developed in the late 1940s and their basic architecture has only changed by adding new components (like networking, hard drives, graphics). (Before there were computers, a computer was a person who computed. The Manhattan project used hundreds of computers (typically young women) to calculate the differential equations for the scientists of the project.)

The components of a modern computer are:

Central Processing Unit (CPU),
Memory
Storage
Input/Output devices

2.1. Example
We want to design a computer that does very little: it will check a temperature, check whether the temperature is too hot or too cold, and accordingly either do nothing, send a signal to a cooling unit, or send a signal to a heating unit.
2.2. How to Encode Data

We need to somehow measure the temperature. The temperature measurement needs to be converted into electric impulses, since we do not want to build a mechanical control. To do so, we can use a physical value such as the voltage between a signal line and ground to represent the temperature (analog), or we can encode the information digitally. Since analog computers are basically no longer used, the latter fits best into our use of a computer.

Assume that the range and accuracy of the temperature measurements are limited. Assume that the cable from the temperature gauge to the computer consists of 9 lines. We will use one line for the ground, and the other 8 for the data transfer. The temperature gauge will set a certain data line to either a value around 0 Volts or to a value around 5 Volts (this number is somewhat arbitrary.) The manufacturer of the temperature gauge promises to keep the signal levels within certain bands around these values, and the computer manufacturer promises that its device will correctly read voltage levels. We assume that voltage levels outside of the bands never happen.

It is actually quite difficult to transmit any signals through a wire without having some noticeable degradation, but since an on-off signal only needs to fall within a band, typically digital signals are transmitted very reliably. The two values for a signal are called “logic 1” and “logic 0”. We call the information transmitted in a single line a bit.

We have 8 data lines and 1 ground. In an actual situation, we would have more than a single ground. Now we have to design how to encode the temperature readings using 8 bits. We order the data lines in an arbitrary order, numbering them from 0 to 7. (You will learn why computer scientists prefer to start counting with 0.) We can write the values of the signal as \( x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 \) where symbol \( x_i \) is either 0 (indicating that data line \( i \) has a logic zero voltage level) or 1 (indicating that data line \( i \) has a logic one voltage level). For example, given the voltage levels in the following table, we receive a signal 0010 1011.
<table>
<thead>
<tr>
<th>Data Line</th>
<th>Voltage</th>
<th>Data Line</th>
<th>Voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.05V</td>
<td>4</td>
<td>4.97V</td>
</tr>
<tr>
<td>1</td>
<td>0.13V</td>
<td>5</td>
<td>0.41V</td>
</tr>
<tr>
<td>2</td>
<td>5.21V</td>
<td>6</td>
<td>4.81V</td>
</tr>
<tr>
<td>3</td>
<td>-0.02V</td>
<td>7</td>
<td>5.08V</td>
</tr>
</tbody>
</table>

### 2.3. Advantages of Analog and Digital Transmission

Let us pause and consider the respective advantages and disadvantages of analog and digital data transmission. The advantages of analog are clear: a single channel can transmit many more messages than a digital transmission using the same channel can. Often, the association between signal and message is easy. Digital’s advantage comes from the almost impossibility to transmit signals without error. In particular, if we transmit an analog signal through many way stations (transceivers), the cumulative effects of transmission errors and transceiver errors will degrade the analog signal. However, unless the digital signal is subjected to a large error, so that a 0 flips into a 1 or a 1 flips into a 0, the digital signal is exactly replicated at each transceiver and reaches the receiver with little probability of error. However, in order to do so, the digital signal usually takes up more resources to transmit the same message.

### 2.4. Binary and Hexadecimal Numbers

Our next task is to define the “meaning” of the signal values. With other words, what does reception of a signal 0010 1011 say about the temperature that we are about to control? We can of course choose an arbitrary assignment, as exemplified in the following assignment:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Message</th>
<th>Signal</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>[78.3, 79.5]</td>
<td>1000 0000</td>
<td>[-135, 10]</td>
</tr>
<tr>
<td>0000 0001</td>
<td>[23.1, 29.7]</td>
<td>1000 0001</td>
<td>[29.7, 35.6]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Obviously, this approach makes the life of the temperature gauge manufacturer difficult, just as it makes our lives as computer engineers difficult.

A more systematic approach to give meaning to strings of 8 symbols that are either zero or one is required. Fortunately, Mathematics offers such an approach. A string of eight symbols such as 0010 1011 looks very much like a decimal number, though typically we suppress leading zeroes. Indeed, we can read such a string as a number. Recall that the decimal number 4711 is nothing but an abbreviation of $4 \cdot 1000 + 7 \cdot 100 + 1 \cdot 10 + 1 \cdot 1$. Any finite string consisting of the 10 symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 can be read as a positive integer this way. However, our strings have only symbols 0 and 1, so we best interpret them as the *binary representation* of a number.

We recall from Mathematics that the $g$-adic representation ($g$ is a positive integer) of a number $x$ is the uniquely determined string $a曷a曷...a曷a曷$ with the following properties: (1)
\(a_i \neq 0; \) (2) \(a_i \in \{0, 1, \ldots, g-1\}; \) and (3) \(x = \sum_{v=0}^{i} a_v g^v.\) For example, the binary number 100 is 100 = 64 + 32 + 4 = 1*64+1*32+0*16+0*8+1*4+0*2+0*1, so that the binary representation of 100 is 1100100. Conversely, given the binary representation 11010010, we obtain the decadic representation as 1*128 + 1*64 + 0*32 + 1*16 + 0*8 + 0*4 + 1*2 + 0*1 = 210. There is a faster way to convert from decadic to binary. If you look at the power series in 2 expansion of for example 100, you see that we can divide this expression by 2 by lowering all powers of 2 by one with exception of \(2^0 = 1.\) This addend however gives the remainder of the division by 2. We can calculate the binary representation of a number using the following scheme:

\[
\begin{align*}
100 / 2 &= 50 \text{ rem } 0 \quad (\leftarrow \text{LSD}) \\
50 / 2 &= 25 \text{ rem } 0 \\
25 / 2 &= 12 \text{ rem } 1 \\
12 / 2 &= 6 \text{ rem } 0 \\
6 / 2 &= 3 \text{ rem } 0 \\
3 / 2 &= 1 \text{ rem } 1 \\
1 / 2 &= 0 \text{ rem } 1 \quad (\leftarrow \text{MSD})
\end{align*}
\]

We can read of the binary representation from the remainder values starting from the right, so that 100 = 110 0100.

Because of the easy conversion to binary and the more compact form of writing it, Computer Science uses the hexadecimal system as well. The base is 16. The 16 digits are given in the following table with their symbol, their numeric value, and the binary equivalent.

<table>
<thead>
<tr>
<th>Hexadecimal digit</th>
<th>Decadic Equivalent</th>
<th>Binary Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
To convert from binary to hexadecimal, we group the binary number starting from the right in groups of four digits. Then we replace each group of four binary digits with the hexadecimal digit equivalent. For example, given 100100011000111000001111, we group 0001 0010 0011 0000 1110 0000 1111, where we padded with leading zeroes. Then we obtain 1230E0F as the hexadecimal equivalent.

2.5. Layering

We return to our example. We can interpret the digital values of the eight signal wires as a binary number from 0000 0000 to 1111 1111, i.e. from 0 to 255 inclusive. We now attach a meaning to the numbers that we receive from the temperature gauge. To assign this meaning, we need to know more about the situation. For simplicity’s sake, assume that the range of reasonable temperatures goes from –50°C to 200°C. In this case, we can give the temperature reading \( x \), \( x \in \{0, 1, 2, \ldots, 254, 255\} \) the meaning that the room temperature is \((x-50)°C\).

We notice that we have given meaning to signals, i.e. associating signals to messages, twice while solving the same engineering problem. First, we solved a transmission problem, how to use voltage to send over numbers. Then we solved a metrics problem, associating the number with a temperature reading. Breaking up an engineering task in this manner is a common and important tool of reducing the complexity of the task.

<table>
<thead>
<tr>
<th>Metrics:</th>
<th>Message: Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal:</td>
<td>Binary Number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transmission:</th>
<th>Message: Binary Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal:</td>
<td>Voltage Differences</td>
</tr>
</tbody>
</table>

3. Microprocessor Components

A microprocessor contains the heart of a computer. It is conceptually made up of a number of components.

3.1. Clock

Clocks are ubiquitous in digital data and signal processing. They provide a signal that initiates and synchronizes the actions of many components in a microprocessor. A clock signal schematically has the following form, though in practice the periods where a clock attains a 0 or 1 value can be different.
3.2. Registers

Registers store $n$ bits, with typical values of $n = 4, 8, 16, 32$. A register has $n$ input lines and $n$ output lines. Connected to the register are two control lines, the clock and an enable line.

If the clock enters a certain phase, e.g. on the falling edge (when the clock signal goes from High to Low), and if the enable signal is activated, then the input values become available as output values. They remain valid until the clock again goes from High to Low. By not enabling the register, we can keep the output alive as long as we want.

3.3. Register Bank

At the heart of the data path within a microprocessor is a “bank” or collection of registers. In our continuing example, I will use a bank of four registers with 8 bits each. The registers are named RA, RB, RC, RD.

3.4. ALU – Arithmetic Logical Unit

A microprocessor is about computing, and the computing is done in a unit called the ALU (Arithmetic Logic Unit). This is the place where data are processed. An ALU has two data inputs, one data output, and input for an op-code (operation code) as well as for the clock and an enable line. The op-code is another binary number that tells the ALU what operation to perform. Typical operations that an ALU performs are addition,
subtraction, logic operations like bit-wise and, bit-wise or, bit-wise not, bit-wise exclusive or, comparisons, checks for equality etc.

3.5. Memory

Memory can store a large number of fixed length binary numbers. Actually, what we store is up to us, as long as it is stored represented by a fixed length binary number. Each memory location has an address and a content. For the sake of our example, we use an unrealistically small memory with contents of size 8 and addresses of size 8 bits. That is, we store up to 256 8 bit binary numbers of 8 bit in length. By looking at it, we cannot tell whether an 8 bit number (a number from 0 to 255) is an address or the contents of a memory location. This difference, which is so intuitively clear, will later come back to haunt us; most beginners in C will have a hard time to distinguish between the address and the contents of the address, when they start writing programs.

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0011</td>
<td>0110 0111</td>
</tr>
<tr>
<td>0000 0111</td>
<td>0100 1111</td>
</tr>
<tr>
<td>0010 0001</td>
<td>1110 0001</td>
</tr>
<tr>
<td>0010 1001</td>
<td>0000 0111</td>
</tr>
</tbody>
</table>
While we do not need to understand how memory works in detail, we need to understand the principles of the memory interface. That is, we need to understand how the microprocessor accesses memory. To access memory, the microprocessor needs to provide the memory address. In our example, the memory address can be stored in an 8-bit register. Furthermore, the microprocessor needs to provide a register in order to buffer the data that is sent to memory or comes from memory. Finally, the microprocessor needs certain control lines to communicate with the memory. For once, the microprocessor needs to tell the memory what type of operation to perform. Second, the microprocessor needs to wait for the memory to be ready, as the memory chip might fall behind the microprocessor when many operations are performed one after the other.

In order to read from memory (Read or Fetch operation), the microprocessor places the address of memory to be read into the Memory Address Buffer. The microprocessor control places the control word into the Memory Control Buffer. The memory constantly reads this buffer, and when it finds that the microprocessor tries to communicate with it, it reads the whole control word. On finding that the READ signal has been asserted, the memory module transfers the address from the Memory Address Buffer, and then performs the operation. The result is the contents of the memory location given by the memory address buffer. This is placed into the Memory Data Buffer. By asserting a control line, the memory changes the memory control word in the Memory Control Buffer. The control of the microprocessor now knows that memory has read the
requested memory location and that the contents there are now in the Memory Data Buffer. Control now transfers the contents of the Memory Data Buffer to the desired register and the memory read operation is finished.

In order to write to memory (a Write or Store operation), the microprocessor not only loads the memory address in the Memory Address Buffer, but also loads the future contents of that memory location into Memory Data Buffer. The control then loads a different memory control word (indicating a Write) into the Memory Control Buffer. The memory checking periodically the contents of the Memory Control Buffer realizes that it is requested to perform a write. It transfers the address and the contents, and then accordingly overwrites the memory location given by the address with the new contents. The old contents are irretrievably gone. The memory indicates by overwriting a bit in the Memory Control Buffer that it is ready to accept new requests.

### 3.6. Ports

In our example, we need to get the data from the temperature sensor, and we need to write control data to the air conditioning and heating units. This is accomplished through special registers called ports. An input port allows an outside entity to write to a register that the microprocessor can read. Obviously, this is not an easy engineering task, since we need to protect the microprocessor against voltage spikes, but also we need to enable the microprocessor to read cleanly, e.g. when an input from the outside changes, we might see some spurious values. An output port performs the opposite function, the microprocessor writes to a register that can be read by an outside entity.

### 3.7. Data Paths

As we are talking about the various parts of the microprocessor, you have hopefully become aware that data needs to be gated from the various entities. For example, all the registers in the register bank need to be connected to the two inputs of the ALU. This is being accomplished by a host of electronic switches that allow one and only one register to be connected to – let’s say – the Memory Data Buffer. These switches are set by the “brains” of the microprocessor, the control unit.

### 3.8. Control

So far, we have discussed elements of the data path. However, there needs to be a unit who sets up the data paths and controls the various elements. For example, to execute an arithmetic operation, we need to gate the two inputs from the register bank to the ALU, give the ALU the right operations command, and gate the output back into a register of the register bank. All this has to be done under the control of the programmer.

The task of the control unit (the most complicated piece of the microprocessor to design) is to load an instruction from memory, interpret the instruction by setting up the correct data paths within the microprocessor and by giving the right commands to the various units, and thus execute the instruction. The control unit uses two registers, the
Instruction Register (IR) and the Instruction Counter (IC), which is also called the Program Counter (PC). The instruction register contains the instruction that is about to be executed. The instruction counter contains the memory address where the instruction can be found. Since in our microprocessor design, we allow any two of the four registers to become the operands of an arithmetic operation and since we allow storing the result in an arbitrary register, we need $4 \times 4 \times 4 = 64$ instructions for any arithmetic operation. Since there are more than four arithmetic operations, we need more than 256 different types of instruction. Therefore, we cannot possibly store all instructions in eight bits. Therefore, we assume an IR that contains 16 bits. Since our memory only stores words of eight bits, we need to consecutive memory locations to store a single instruction. The instruction needs to be loaded in two different memory accesses.

The control unit performs the Fetch-Execute Cycle:
1. **Fetch**: The IC “points” to the address in memory where the next instruction is stored. The control loads eight bits of the instruction by reading from memory. Then the IC is incremented, so that it now points to the next instruction in memory, which contains the second half of the instruction. Simultaneously, the result of the read is loaded into one half of the IR. The control then performs another read, which transfers the second half of the instruction from memory and at the same time, it increments the IC. The IC now points to the beginning of the next instruction.
2. **Execute**: The control unit now interprets the instruction in IR. Because of this interpretation, the right data paths for the execution of the instruction are set and microprocessor units such as the memory control word or the ALU receive the correct operation code.

### 4. Our First Program

We recall that our example situation was a control problem. We receive an eight bit signal from a temperature sensor and in return will set two bits that will be interpreted by the heater and the cooling units as signals when to turn on and when to turn off. Our task is to write a program that will set these signals correctly.

As a matter of course, we will use the input port to receive temperature data. Recall that the binary number we received is decremented by 50 to yield the (rounded) temperature in degrees Celsius. For the output, we need to set only two bits. Since our output port (the only one to communicate to the outside world) has eight bits, we have spare capacity. Of the eight bits, we will use the second to last in order to give a command to the heater (1 meaning that the heater should be on) and the last to give a command to the cooling unit. The remaining six bits are not going to be used. However, we need to give them some value, so we set them all to zero. We are left with three words that we could possibly write to the output port.

<table>
<thead>
<tr>
<th>Word</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>heater off, cooler off</td>
</tr>
<tr>
<td>0000 0001</td>
<td>heater off, cooler on</td>
</tr>
<tr>
<td>0000 0010</td>
<td>heater on, cooler off</td>
</tr>
</tbody>
</table>
Our program at this point is underspecified, because we have not yet determined what the desirable temperature rate should be. We set the range from 18°C to 24°C.

Our program will continually execute the following loop: We read in the temperature. Then we compare the temperature with 18°C. If the temperature is below 18°C, we send the word 0000 0010 to the output port, which will turn on the heater. If not, then we check whether the temperature is higher than 24°C. If yes, then we turn on the cooler. Otherwise, we turn both cooler and heater off.

We need to store the cut-off values of 24°C and 18°C (corresponding to the temperature codes 68 and 74) either in memory or in the register bank. Since we can get away with storing them in the registers, we will do this at the beginning of our program.

Somewhere in our program, we will have to execute a conditional instruction, that is, an instruction, which is only executed when a certain condition is true. We assume that the way the ALU calculates a condition in the form of a statement such as \((RX < RY) \rightarrow RZ\), which means that we compare RX with RY using one of the symbols \(\leq, \geq, >, <, ==\) (equals), \(!=\) (not equals) and store the result in RZ. If the comparison evaluates to true, then RZ contains the word 1111 1111, otherwise the word 0000 0000. Here, RX, RY, RZ are stand-ins for RA, RB, RC, or RD.

It is perfectly possible to design and build a microprocessor with a “guarded statement” of the form \(RA? RB+RC \rightarrow RD\), where the second part of the statement \((RB+RC \rightarrow RD)\) is only executed if RA contains the value true, i.e. the word 1111 1111. However, this approach is not taken, probably because the instruction would be too complicated and if we were to include it in the instruction mix, then it would slow the clock speed of the
computer. However, one type of guarded statement is frequently implemented, namely a *conditional jump*. We can conditionally jump by loading the IC with the value of an instruction that is going to be the next in line to be executed. For an example, assume that we have stored the following instructions in memory:

<table>
<thead>
<tr>
<th>2l:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RA&lt;RB) -&gt; RC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2l+1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>if RC then IR&lt;- 2l+6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2l+2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA + RA -&gt; RA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2l+3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA -&gt; Output Port</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2l+4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
</tr>
</tbody>
</table>

Recall that instructions take up 16 bits, therefore we use two memory locations to store a single instruction. 2l is an even number, therefore it is where a new instruction would start. Assume now that RA is indeed smaller than RB. The instruction stored in memory words 2l and 2l+1 then assigns 1111 1111 to RC. The instruction stored in location 2l+2 and 2l+3 then loads the IC with the value 2l+6. Therefore, the next instruction to be loaded is “RA -> Output Port” and then “STOP” (which stops the microprocessor). Thus, we skipped the instruction “RA + RA -> RA”. If on the other hand RA is not smaller than RB, then RC is loaded with 0000 0000. At the end of the instruction “if RC then IC<- 2l+6”, IC is not overwritten, so that it contains the value 2l+4. Therefore the instruction “RA + RA -> RA” is executed and afterwards the remaining two instructions. With the conditional jump, we have *de facto* executed the guarded statement “if RC then RA + RA -> RA”. Thus, conditional jumps give us the same programming power as guarded statements.

Simpler than conditional jumps are unconditional jumps where we just load the IC with the address of the next instruction to be executed.

We are now ready to write our first program:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000, 0000 0001: 0100 0100 -&gt; RC //load 68 into RC</td>
<td></td>
</tr>
<tr>
<td>0000 0010, 0000 0011: 0100 1010 -&gt;RD //load 74 into RD</td>
<td></td>
</tr>
<tr>
<td>0000 0100, 0000 0101: I-Port -&gt; RA //load temp. from port into RA</td>
<td></td>
</tr>
<tr>
<td>0000 0110, 0000 0111: (RA≥RC)-&gt;RB //compare RA and RC, //and load the result into RB</td>
<td></td>
</tr>
<tr>
<td>0000 1000, 0000 1001: if RB then 0000 1110 -&gt; IC</td>
<td></td>
</tr>
<tr>
<td>0000 1010, 0000 1011: 0000 0010 -&gt; Oport //turn heater on</td>
<td></td>
</tr>
</tbody>
</table>
Our program is simple, we only have 10 statements, but the control flow is difficult to understand. Here is one schematic way to understand it better:

Load parameters into registers RC and RD

Load temperature into RA

Is RA ≥ RC?

Yes

Is RA ≤ RD?

Yes

Turn cooler on.

No

Turn heater on.

Turn cooler on.

Turn cooler on.
This “flow charts” explains the decision process that our little program implements. The boxes indicate a series of one or more statements that implement the description in the box. The diamond shaped boxes indicate a decision described in the text of the box. The decision is binary that is a yes or no question. Depending on the answer, the program “flows” to either one of two alternative statements. Arrows indicate these flows as well as the flow between groups of statements.

In the actual program, we move along an arrow either by moving to the next statement or by loading the IC with the address of the next statement to be executed. Clearly, we save a jump instruction if the instructions corresponding to the next box are located in memory just behind the instructions belonging to the previous box.

In our example, decisions are implemented by loading a register with the result of the decision and then making a conditional jump depending on the value in the register. All microprocessors use conditional jumps to implement decisions, but many have slightly different techniques. For example, in the INTEL family of chips, a condition is evaluated by loading a “results” register. Depending on the bits in this result register, various conditional jumps are executed.

Before we leave the example, a word of warning: If the temperature measurement changes, e.g. from 0001 1000 to 0001 0111, then the input port can pick up intermediate temperatures, because the bits do not flip at exactly the same time. Thus, the change could be 0001 1000 -> 0001 1010 -> 0001 1011 -> 0001 1111 -> 0001 0111, and we could pick up any of the intermediate values. Around the cut-off values, this phenomenon could lead us to spuriously turn on the heater or the cooler. This is not a good idea, but with the means at our hands, it would be a difficult task to change the program to be safe.

### 5. Flow Control

We have already seen that a program needs to make decisions, that is, that the flow of the program depends on internal values and ultimately upon the input. All microprocessors implement flow control with “jump” statements. A jump statement loads the IC with the address of the next instruction to be executed. Jumps can be conditional, depending on the value of a certain register, or they can be unconditional.

### 6. Program Access To Memory

Most programs need more storage for intermediate results than they can find in the register bank. Thus, they need to read from memory and write to memory. The read instruction has the form

\[
\text{Mem}[\text{address}] \rightarrow \text{RX}.
\]

The instruction has two parameters, the address of the memory and the target register to which the memory contents are copied. To execute this instruction, the microprocessor has to accomplish the following:
(a) Load the address into the Memory Address Buffer. The address is part of the instruction and can hence be found in IR. For example, if the instruction is Mem[01010101] -> RA, then the instruction has the form xxxxxxxx01010101. The first eight “x” stand for eight bits that tell the control that this is a memory read. Two of the eight “x” are an encoding for the internal number of register A. Thus, control moves the contents of the right half of IR (which at this moment contains the full instruction) into Memory Address Buffer.

(b) After the address in the Memory Address Buffer is valid, the control sets a bit or bits in the Memory Control Word to indicate a read operation. This sends a signal to the memory unit.

(c) The memory unit upon receipt of the signal performs the operation. It goes to memory location 01010101 and retrieves the contents – let’s say 0000 1111. The memory unit moves these contents to the Memory Data Buffer. After the data 0000 1111 in the Memory Data Buffer is valid, the memory unit resets bits in the Memory Control Buffer to alert the microprocessor that the read operation has terminated.

(d) The microprocessor receives the “Data Valid” signal from the memory unit and transfers the contents of the Memory Data Buffer to the register indicated in the instruction.

A write to memory instruction has the form

RX -> Mem[address]

with two parameters, the register that contains the data to be stored and the address of memory, to which data is being transferred. To execute a write instruction, let’s say RA -> Mem[01010101] the microprocessor does the following:

(a) The microprocessor loads the address 0101 0101 into the Memory Address Buffer. The address is part of the instruction, so that it is in the right half of IR.

(b) Simultaneously, the microprocessor copies the contents of register RA into the Memory Data Buffer.

(c) After both transfers are completed, the microprocessor sets a bit or bits in the Memory Control Word to indicate a write operation. This sends a signal to the memory unit.

(d) The memory unit copies the contents of the Memory Data Buffer into the memory location 0101 0101 given by the Memory Address Buffer.

There are other read and write instructions as well. For example, the memory address can be contained in another register. Thus, the instruction

RA->Mem[RB]

loads the contents of register A into the memory location, which address in RB.


The microprocessor reads an instruction from memory every fetch-execute cycle. The instruction is in machine readable form, that is, it is a binary number. For a human being, it is very hard to keep different instructions apart and to accurately load a machine language program to memory. For this reason, we used mnemonics, which are stand ins
for the real machine language instruction. For example, the machine language instruction 1001 0100 0101 0101 might be the machine language instruction for the mnemonic Mem[0101 0101] -> RA. In the very beginning, a computer programmer would write in machine language. This became very quickly to error prone to be sustainable, and a computer program was written – the Assembler – that made the task more feasible. The assembler takes an Assembly Program and translates it into a program in machine language instructions. An assembly program consists almost exclusively of instruction in the mnemonic format. However, two further improvements lighten the load on the programmer: First, we recall that we need to jump to instructions in order to implement program flow control. Finding the jump address means carefully counting all instructions to find the number of the instruction to which we want to jump. The assembler allows the programmer to label statements. Instead of giving the address of the instruction in a jump statement, the programmer gives the label. For example, the absolute jump 1100 0010 -> IC statement becomes the “goto” statement

goto PointA

where PointA is the label. The instruction at memory location 1100 0010 (assuming it for the example’s sake to be RA->Mem[RB]) is labeled as

PointA: RA -> Mem[RB].

A further easement on the programmer’s burden is given by giving memory locations different names and referring to them by name. For example, an assembly program can declare a memory location to be called parama (for parameter a). Writing to this location is then done by the assembly language statement

RA -> parama

The assembler will replace the name “parama” with the memory location. To do all this, the assembler keeps an internal list of all the labels and names. Processing the labels is easy. Whenever the assembler encounters a label, it makes an entry into a table, associating the name with the number of the instruction. To help the assembler with the names for memory location, many assemblers require the programmer to give a list of names to be used with the size of the memory requested.

Because computers were very expensive in the past, stopping a computer, loading a program (e.g. from a deck of punch cards), executing the program, gathering the output is a time-consuming burden involving human help (the operator). Therefore, general purpose computers started to run an Operating System. In its primitive incarnation, an operating system is a single program that continuously runs. The operating system interacts with the user through keyboard and console. Through user commands, the OS can run other programs. To do so it would first link and load. This entails essentially (I am leaving out some important details for the sake of clarity) the following steps:
The program to be executed is in quasi-machine language. The OS decides where in memory to place the machine code of the program. The linker goes through and “resolves” all memory addresses, so that they reflect the intentions of the program. For example, instead of jumping to address 10, the linker will have the program jump to address 10+X if X is the start of the program. At the end, a machine language program (a binary) is loaded in memory.
The operating system now jumps to the first instruction of the newly loaded program. Thus, the program starts executing. When the program is done, instead of stopping the computer, the program jumps back to an instruction in the OS, so that the OS now again controls the computer. The OS waits for input from the console telling it what program to execute next.

Programming in Assembly is difficult. Furthermore, an assembly program will only run on one type of machine, because most assembly language instructions refer directly to machine language instructions that are machine dependent, assembly programs run on only that type of machine.

In the 50s, 60s, and 70s, money in the computer industry was made by selling hardware. Customers (companies and government agencies mostly) would be responsible for generating their own, tailor-made software. These customers could not be talked into upgrading their hardware if it meant rewriting all their programs. New customers could not be found if programming continued to be difficult. Therefore an easier way to program needed to be found. IBM introduced the first two, which are still highly successful, namely Cobol for business applications and Fortran (formula translator) for scientific applications. These hll (higher level languages) ease the burden of programming and programs written in them can be transformed into machine languages for all machines that have a Compiler, i.e. a tool that translates an hll-program into an assembly.
The C Programming Language

8. Elements of a C Program

We distinguish a number of elements in a C program:

1. **Compiler directives.** These statements tell the compiler what to do. Often, they are prefaced with a sharp sign #, in which case they invoke the C preprocessor, which is a tool that is invoked before the actual compilation takes place. The C preprocessor will change the text of the C program based on the information given to it in the directives.

2. **Comments.** Comments are notes by the programmer or the IDE to the human reader of the C-program. C comments are enclosed in a pair /* */ of symbols. Because “commenting out” parts of the program effectively removes the part from the view of the compiler, commenting can also be a tool in hunting down errors. This practice can yield difficulties if one comments out a comment. Therefore C++ introduced an alternative form of commenting, in which a comment begins with a double forward slash // and extends to the end of the line.

3. **Declarations.** A declaration allocates storage. Since memory sizes and processing powers differed at the time of the inception of the programming language (a process which will repeat itself as we move to 64 bit processing) different machines would work best with different sizes for common variable types. For example, integers could be best stored in 16 bit or 32 bit. C has native types, but also allows the declaration of new types.

4. **Statements.** Statements correspond to assembly language statements, though C is so expressive that a typical C statement needs to be translated into several assembly language statements.

5. **Function declarations.** Every statement in C is encapsulated in a function. The function that is called when the program is invoked is called main. You can define main in several different ways, but you have to have a function called main:

   ```c
   int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
   {
      // your code here
      return 0;
   }
   ```

   is the form that Visual C++ generates. In a UNIX system, you would use

   ```c
   int main()
   {
      // your code here
   }
   ```

9. Types and Variables

A variable is the name for a storage location. We *think* of a variable often in the way that a mathematician thinks about the object represented by the variable. C contains some “native” types. At this point, we use the following types:
int An integer.
long An integer, too, but one that has typically more storage allocated to it.
float A floating point number.
double A double precision floating point number.

The ANSI C requirements on the actual storage being allocated to each type are intentionally vague. This way, your C program will still be running on machines that are not yet designed, but which will abound (by present standards) with memory and register space. C does **type checking**. That is, C will not allow you to assign variables that should not be assigned, as for example in moving a string to a floating-point number. This presents typically a programmer error. C will allow the operation eventually, but only after the programmer makes her/his intentions clear. We will not see this facility immediately, because C does allow **type casting**. This means, one variable can be transformed into another variable. For example, the expression 1/ix, where ix is declared to be an integer variable, is evaluated to 0 unless ix is – absolutely speaking – at least 1. We often would prefer to make ix into a floating point number first. We can do so in two ways, first we can say
1 / (float) ix
which changes ix into a floating point number. This is not quite what happens, but we can think about it this way. The compiler will create a new, temporary variable, and then load this variable with a floating point number corresponding to ix. We can make this explicit in our program, e.g.

```c
int ix;
float sum, fx;
...
fx = ix;
sum = sum + 1/fx;
...
```

is a program fragment that defines a variable ix and a variable fx. When we assign ix to fx, then fx gets the value of ix. This works the other way around, too. In

```c
ix = fx;
```
we assign the floating point value of fx to ix. “Mathematically”, these are two different values, because ix will contain only the integer portion of fx.

### 10. Control Flow

C contains a number of high-powered statements for program flow. These statements usually use a boolean expression, that is, an expression that is either true or false. Examples of boolean expressions are:

```c
(fx == fy), (fx<fy),
```

e.g. The **while** statement has the following form:

```c
while(boolean statement) {
    //body of while loop
}
```

The program examines the boolean expression. If the expression is true, the program flow enters the body of the while loop. If the expression is false, then it jumps to the
A much more specialized version of the while loop is the for loop. It has the following syntax:

```c
for( initialization ; loop condition ; iteration statement)
```

as in the following example

```c
for(i=0; i < 10; i=i+1);
```

The initialization statement is a statement that is only executed once, namely when the for loop is first encountered. The initialization statement can also be used to declare the variable. For example, if the variable `i` is not declared before, we can do so within the for statement:

```c
for(int i = 0; i<10; i++)
```

The variable is only defined inside the loop, though compilers will differ on this point.

The exit condition is a boolean expression that is evaluated to determine whether the loop is entered or not. The iteration statement is a statement that is executed at the very end of the for loop. The first and the third statement in a for loop can be compound statements separated by commas as in

```c
for(i=0, j=0; i<10; i=i+1, j=j+1) {
  // body of for loop
  i=i+1; j=j+1;
}
```

This last example is given by the following flow chart:

```
flow chart:
```

```plaintext
i=0; j=0;

i<10

// body of for loop

i=i+1; j=j+1;

// next statement
```

The most elementary control flow statement is the if then else statement, which has the following structure:

```c
if(boolean expression) {
  // code executed if boolean expression is true
}
else {
  // code executed if boolean expression is false
}
```
//code executed if boolean expression is not true
}

The last else clause is optional. The construct is fairly simple, but there is a possible ambiguity if we have nested statements like in the following example:

```c
a = 3;
if(count == 0)
  if(sum >= 0)
    a = 5;
else
  a = 4;
```

Here we have a “dangling else” that could be the alternative to either the first or the second if condition. C somewhat arbitrarily associates the dangling else to the closest if statement. Thus, the code reads in full bracketing:

```c
a = 3;
if(count == 0) {
  if(sum >= 0) {
    a = 5;
  }
  else {
    a = 4;
  }
}
```

It is usually a good idea to use full bracketing anyway. You can get your intent across with indentation but indentation is ignored by the compiler. Can you spot the differences from this code?

```c
a = 3;
if(count == 0) {
  if(sum >= 0) {
    a = 5;
  }
}
else {
  a = 4;
}
```

If count is 1 and sum is –1, then the variable a stays 3 in the first fragment, but becomes 4 in the second.

### 11. More C Types

#### 11.1. Fundamental Types: int, float, char, bool

C has loose minimum standards for the amount of storage given to primitive types. For example, an int has to have at least 16 bits, allowing us to represent numbers from $-2^{15}$ to $2^{15}-1$, or –32000 to 32000 approximately. On many platforms (Windows, Linux on a PC, Mac) an integer is normally 32 bits long, ranging from $-2^{31}$ to $2^{31}-1$ or from –2,000,000,
000 to 2,000,000,000 approximately. C recognizes the need for higher range and more precision for floating point numbers. Thus C has a family of number types: int, short int, long int, float, double, long double. Not all integers need to be signed. If you do not want signs, then the range doubles, you indicate this by using the qualifier unsigned as in “unsigned long int”. Instead of using “long int” or “short int”, you can also use just “long” and “short”.

In a statement “int x = 5;” the number 5 is a constant. If you precede a constant with a leading 0, then the number is read as an octal number (base is 8). If you precede a constant with a leading 0x or 0X (the first character is a 0), then the constant is read as a hexadecimal number. Thus,

```c
int x = 012; // octal, x has dec. value 10
int x = 0x12; // hex, x has dec. value 18
int x = 12;   // decimal.
```

Another important type is a character. A character is an encoding for a character as in the Latin characters in which we write. Original character encoding used 7 bits per character, but this was expanded to the 8 bit long ASCII character set. The ASCII character set supports most Latin based alphabets, i.e. it has accents, suffixes, and umlauts, but it is too small to encode other alphabets. This has been accomplished in the 16 bit long Unicode characters, which aim at representing basically every written language on earth. To define a character ch, use this statement:

```c
char ch = 'A';
```

The final important fundamental type is a boolean. A boolean is a true, false value. It does not exist as a type in ANSI C, but is being introduced in newer versions of the language. In standard C, boolean results (e.g. of an expression using the equality operator) are encoded as integers.

### 11.2. Type Conversion

C knows of two types of conversions. One is programmer controlled, and we will talk about it later. The other is automatic or implicit conversion. If you assign a floating point number to an integer, the floating point number will be truncated (not rounded), that is, the fractional part is thrown away. Since the range of floating point numbers is larger than the integer range, the conversion is not always accurate. For example, if a floating point variable contains a large number as its value and if we assign it to an integer variable, then the result of the conversion is technically not defined. But the C compiler will do its best to provide the best approximate value for the conversion.

### 11.3. Casting

In certain expressions the type of the variable matters. We can change the type of a value (whether variable or constant) by putting the new type in parentheses in front of it. For example, in the following code fragment, we convert the integer constant 2 to the floating point constant 2.0. As a result, the fraction will consists of an integer divided by a
floating point number, thus the integer gets silently converted into a floating point number, and the division is executed as a floating point division.

cout << 1/(float)2 << endl;

12. Arithmetic Expressions

12.1. Operators

The list of C operators is quite long. Here, we only talk about the indispensable operators for the beginning programmer.

“=” The equal sign does not exactly mean equal. To compare, we use the double equal sign, ==. Rather, the equal sign is the assignment operator. Speaking about C, we call a data object some region of storage. A variable is an example for a data object. C uses the term lvalue to denote the name or expression that identifies a particular data object. Something that can be stored in a data object is called a rvalue. For example, in the statement

iSum = 200;

the variable iSum is a lvalue and the 200 is a rvalue. The assignment operator can be repeated. For example

iSum = iCount = iValue = 1;

assigns the rvalue 1 to all the variables iSum, iCount, iValue.

Arithmetic Operators “+”, “-”, “*”, “/”: These binary operators express the addition, subtraction, multiplication, and division of number types. These operations work differently for integer types and floating point types. Integer operands are converted silently into floating point operands if another operand is of floating point type.

The unary negation “-” operator: By putting a minus sign in front of a signed number variable, we can change the sign of this variable.

The modulus operator “%”: The expression iVal1 % iVal2 gives the remainder of the division of the first value by the second value. It is only defined between integer types. Be careful when using negative numbers.

Increment and Decrement Operators ++ and --: Since incrementing an integer is a frequent operation, C has a very concise way of writing them. In the following fragment, we increase the size of the value of iSize in the while statement before we check for the inequality

```c
int iSize = 0;
while(++iSize < 20) {
    cout << iSize << endl;
}
```

If we want to increase it after the check, we write

```c
int iSize = 0;
while(iSize++ < 20) {
```
cout << iSize << endl;
}

The difference between the behavior of both fragments is in the number of times that the while loop is executed.

Relational Operators: >  >=  <  <=
Equality Operators:  ==  !=

These operators are used to form boolean conditions. To form more complicated conditions, we can use the logical functions AND and OR, written in C as && (and) and || (or). A condition composed with these operators is evaluated from the left to the right and evaluation stops if the rest of the formula can now longer change the value of the expression. For example, in 
(i<0) || (i>=0) || (j<5)
the evaluation stops after either evaluating the first condition to true or the second condition to true. The third condition is never evaluated.

12.2. Precedence
We can always spell out the precedence of the operators in a complicated formula using parentheses:
result = ((iSize++)*5)+iAddend;

However, this makes code hard to read. C uses an extended version of the precedence rules that we use in Mathematical formulae (points before lines). The precedence table for the operators we know is this:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --, unary +, unary -</td>
<td>right to left</td>
</tr>
<tr>
<td>*, /, %</td>
<td>left to right</td>
</tr>
<tr>
<td>binary +, binary -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>==, !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=, +=, -=, *=, /=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

The associativity rules spell out how we parenthesize among operators of equal precedence. For example, multiplication and division associate from the left to the right. Thus, the expression
i*j/2*k
is the fully parenthesized expression
(((i*j)/2)*k

Knowing the precedence and associativity rules helps you to write compact code and understand other people’s code.

13. Arrays
Frequently, we use larger chunks of storage space then one allocated to just a single variable. Strings form the most important example. To store the string “Hello World”,

25
we need to store the characters ‘H’, ‘e’, ‘l’, ‘l’, etc. or at least 11 characters. Actually, for reasons that will become clear towards the end of the course, we store an additional last character, which has the ASCII value 0. Thus, the string “Hello World” is stored as 12 consecutive characters. The C construct "Array" does this for you. First, we declare 12 characters in a row by saying:

```c
char greeting[12];
```

where the number of characters is given in the bracket. We access individual characters by for example setting

```c
greeting[0] = 'H';
greeting[1] = 'e';
```

and so on. Notice that the first index into the string is the index 0. C allows us to use a bulk assignment such as

```c
char greeting[12] = “Hello World”.
```

### 14. Functions

C programs are composed of functions, typically distributed over many files. For the time being, we will however place functions into the same file as the main function. A function has variables, from 0 to as many as you want, and returns a value or not. You want to assign to a function a particular task to be accomplished. Breaking up a complicated task into a good set of functions is an essential part of writing good C programs and will only come easy with practice.

Let’s start out with some easy examples. The Fahrenheit to Celsius conversion table selects a Fahrenheit value, then converts it to a Celsius value, and then prints out the result. Arguably, the conversion is a particular task and should be done in its own function. We accomplish this by writing the following program

```c
float celsius(float fah) {
    float fahren = (5.0/9.0)*fah;
    return fahren;
}
```

```c
int main() {
    int i;
    for(i=0; i<200; i+=10) {
        cout << i << "degrees Fahrenheit are " << celsius(i) << "degrees Celsius" << endl;
    }
    return 1;
}
```

This program uses a function celsius. We declare this function in the first line, because the compiler wants to know that this is a function before it accepts it. The implementation of the function makes up the last three lines. The main function is now a
simple loop changing only the variable i and printing out pairs of i and celsius(i). In the cout line, the program calls the function celsius. It copies the value of the variable i into a temporary storage location. Since celsius asks for a floating point number, this variable is first cast as a floating point number. Then the main program hands over control to the function. This is accomplished by setting the instruction pointer to the beginning of the celsius portion of the program. The beginning of the celsius portion of the program then reads the copied value of the argument i. The celsius portion does it work. Afterwards, because it returns a value, it saves the return value in another temporary storage location. Control is then returned to the main function (by jumping to the appropriate place in the main program). Now, main proceeds using the return value it finds in this temporary place.

main gets to the machine instruction that calls celsius.

main stores the address of the next instruction (the return address) in a temporary variable.

main copies all the variables in temporary variables, doing some casting if necessary.

main jumps to the entry point (beginning of function in the binary)

function starts running. It uses the temporary storage space to access

When done, the function places the return result into temporary storage. The copy of the variables can be destroyed.

Function finishes by jumping back to the return address.

main continues where it stopped using the return value in temp. storage.
15. Pointers and Arrays

15.1. Pointers and Memory Locations

For reasons that will become obvious, C provides direct access to memory locations. Recall that there is a fundamental (an ontological difference) between memory contents and memory locations. However, memory locations are numbered, so that an integer can be both a memory content and a memory location.

C does not number memory locations directly. This would be impossible without a greater standardization of computers than there is. Most computers are byte addressable, which means that at some level the programmer has access to individual bytes, and these bytes are individually labeled by numbers representing the memory addresses. But this property is not necessarily universal and would involve programmers in writing machine dependent code, which would not be portable to other machines.

The solution that the designers of C took is to give names to the location of a declared variable. These names are called pointers. A pointer points to the variable of which it is the address. We declare pointers by putting an asterisk in front of the name. For example

```
int data, *pdata, i=0;
```

declares three variables; variables data and i are of type integer and variable pdata is a pointer to an integer location. If we want to give our integer pointer the address of integer data as content, then we can use the address operator:

```
pdata = &data;
```

This statement assigns the address of the variable data to pdata. In order to access the contents of the storage location to which a pointer points, we again use the asterisk. For example, to assign to i the contents of the storage location pdata, we can say:

```
i = *pdata;
```

15.2. Pointers as Arguments to Functions

First, an almost primitive programming problem. Given two storage locations a and b of type character, write a code sequence that will swap their values. The naive solution

```
char a='a', b='b';
a=b;
b=a;
```

fails miserably, because by the time we come to the last statement, character a already contains the value previously stored in b, so that both characters contain ‘b’. We solve this problem by using a temporary variable:

```
char a='a', b='b', temp;
temp=a;
a=b;
b=a;
```

Assume that we want to write a function swap that will swap the values of to character variables. The naive approach

```
void swap(char x, char y){
```
char temp;
temp = x;
x = y;
y = temp;
return;

fails, because when we call `swap(a, b)` the contents of the storage locations `a` and `b` are copied into the arguments of the swap function. The swap function changes the contents of its arguments correctly, but then returns, whereupon the arguments to swap are discarded. To set things right, we need to give the memory addresses of the characters to be swapped to the function swap:

```c
void swap(char* pcX, char * pcY) {
    char temp;
    temp = *pcX;
    *pcX = *pcY;
    *pcY = temp;
}
```

can now be called with the addresses of characters `a` and `b`. Then we call them by

```
swap(&a, &b);
```

### 15.3. Pointers and Arrays

Pointers and arrays are intimately related in C. If we define an array, the compiler insures that storage for all elements in the array are allocated. For example

```c
char ar[5];
```

allocates 5 characters, that are referred to as

```
ar[0], ar[1], ar[2], ar[3], ar[4]
```

If we define a character pointer

```c
char *pc;
```

Then the assignment

```c
pc = &(ar[0]);
```

is legal. For example, we can now change the first element of the character array by

```c
*pc = ‘H’;
```

We can use pointer arithmetic to access the following elements of the array, since the compiler ensures that array elements are stored one after the other.

```c
pc++;
*pc = ‘u’;
pc++;
*pc = ‘b’;
pc++;
*pc = ‘s’;
pc++;
*pc = 0;
```

We can shorten this code by recalling that the increment operator is performed after the access to the variable:

```c
pc++;
*(pc++) = ‘u’;
```
This text loads the array with the string “Hubs” followed by the string terminating zero character. As we see, in lieu of saying
\[
\text{ar[1]} = 'a';
\]
we can just say
\[
*(\text{pc}+1) = 'a';
\]

Things are even easier in C, because the name of an array is a synonym for the location of the initial argument. Thus, an array name in many instances can be used as a pointer. The only difference is that an array name is not a variable, and so cannot appear on the right of an assignment. Thus, in C
\[
\text{ar[i]} = 5;
\]
means exactly the same as
\[
*(\text{ar}+i) = 5;
\]
and this is how C translates the first statement immediately. One immediate consequence is that we can use an array name as an argument for a pointer variable in a function. Here is an example, that calculates the length of a string:

\[
\begin{align*}
\text{int } \text{strlen}(\text{char } *s) \\
&\quad /* \text{returns the length of the string s} */ \\
&\quad \{ \\
&\quad \quad \text{int } n; \\
&\quad \quad \text{for}(n=0; *s != '\0'; s++) \quad \{ \\
&\quad \quad \quad n++; \\
&\quad \quad \} \\
&\quad \quad \text{return } n; \\
&\quad \}
\end{align*}
\]

The following calls are all valid:
\[
\begin{align*}
\text{id } i &= \text{strlen}(\text{“Hello”}); \\
\text{char } \text{ar}[10] &= \text{“Haha”}; \\
i &= \text{strlen}(\text{ar}); \\
\text{char } *\text{pc} &= &\text{ar}; \\
i &= \text{strlen}(\text{pc});
\end{align*}
\]

If we make any of these calls to the function strlen, the address of the beginning of the string (whether constant string or character array) or the storage location contained in the pointer are copied into *s. strlen then sets variable n equal to 0. The for loop condition tests whether the current character value pointed to by s is zero. If not, then n is incremented and finally s is incremented so that it now points to the next character.

\section*{15.4. Examples: Input and Output from Files}
C provides a crude but powerful interface to access files. First a warning: the backslash character in a file name needs to be double back-slashed, as in
char * filename = "c:\temp\data.txt";
If you forget to do this, then the filename will not be read correctly. This type of subtle
and not well-documented mistake can cost you hours. File access is provided by
accessing a file pointer:
    FILE * pfile; //don’t forget to include <stdio.h>
We then open a file using the fopen systems call:
    pfile = fopen("c:\temp\dat.txt","r");
which opens the file located in c:\temp called dat.txt for reading (the “r” in the second
argument). If the directory does not exist, the file pointer will be the null pointer, and no
file will have been created. We check with:
    if(!pfile) {
        cout << "Error, file could not be opened" << endl;
        return;
    }
The second argument to the fopen call gives the file mode. The modes are “r” for read,
“w” for write, and “a” for append. These and some other modes are explained on page
242 of K&R.

When we are done with a file, we should close it, using fclose as in
    fclose(pfile);
If a program exists, then the OS will return all resources used by the process to the
system, this includes closing files opened. However, it is a good discipline to close files
as soon as they are no longer needed.

A file is implemented as a stream, which means in first approximation that there is a
cursor pointing to the current position in the file. When we use any read or write
operations, these are executed at the position of the cursor. There are some functions
such as fseek, ftell, and rewind (p. 248 K&R) that allow you to change the position of the
cursor, but do not expect them to always behave as advertised, especially on a Windows
platform.

Let us look at some simple programs. First, a program that counts the total number of
characters in a file:

#include <stdio.h>
#include <stdlib.h> // contains the exit() call
#include <iostream> // contains cout, cin

// This is the entry point for this application
int main(void)
{
    char pszFileName[80], c;
    int counter=0;
    printf("Enter file name:\n");
    scanf("%s",pszFileName);
    FILE *fp = fopen((const char *)pszFileName,"r");
    if(fp==0) {
        cout << "File could not be opened, exiting" << endl;
        exit(1);
    }
}
while ((c=getc(fp))!=EOF) {
    counter++;
} 
printf("The file contained %d characters\n",counter); 
fclose(fp); 
return 0; }

This simple program opens a file that the user inputs. For the input we use the scanf function, one of the major headaches in C programming. Consult page 245 K&R for the definition of scanf and its cousin fscanf. Notice, that scanf needs a pointer to the location of the input. Giving scanf the wrong type of variable is source of uncountable misery, broken marriages and suicide could result from it!!! For example, if you try to read an integer in using scanf, you might be tempted to write:

    int i; 
    scanf("%d",i); 

and the program will crash (or maybe not, or sometimes). Instead, write

    scanf("%d",&i); 

providing scanf with the location of the integer variable i. We then open up a file. Notice that we interact with the file by means of a pointer to an object called FILE. The FILE object contains a lot of information that allows the application to access the file. Once we have opened the file, we use a while statement to continually read in characters from the file until the character is equal to the end of file sentinel which is produced when the file cursor reached the end of the file. At the end, we close the file using fclose.

#include <stdio.h>
#include <stdlib.h> // contains the exit() call
#include <iostream> // contains cout, cin

int main(void) {
    char* pszInputFileName[80],pszOutputFileName[80], c;
    printf("Enter input file name;\n");
    scanf("%s",pszInputFileName);
    FILE *fpi = fopen((const char *)pszInputFileName,"r");
    if(fpi==0)  {
            cout << "Input File could not be opened, exiting" << endl;
            exit(1);
    }
    printf("Enter output file name;\n");
    scanf("%s",pszOutputFileName);
    FILE *fpo = fopen((const char *)pszOutputFileName,"w");
    if(fpo==0)  {
            cout << "Output File could not be opened, exiting" << endl;
            exit(1);
    }
    while ((c=getc(fpi))!=EOF) { 
       putc(c,fpo);
    }
    printf("File %s has been copied to file %s\n", pszInputFileName, pszOutputFileName);
    fclose(fpi);
    fclose(fpo);
A little bit more sophisticated way of reading in a file is `fscanf`, which is like `scanf`, but has an additional first parameter, that is the `FILE*`. Here is a short example, where we read in a file of numbers, each located in a single line.

```c
void main() {
    char * pszFileName = "e:\temp\temp.txt";
    int i;
    FILE * fp;
    if( !(fp = fopen((const char*) pszFileName,"r"))) { 
        cerr << "File could not be opened" << endl;
        exit(1);
    }
    while(fscanf(fp,"%d\n",&i)==1) {
        cout << i << endl;  // or printf("%d\n",i);
    }
    fclose(fp);
}
```

Here we use the return value of the `fscanf` function in order to stop at the end of file.

### 15.5. Example: String Processing

Assume that we want to compare two strings. If the first string comes alphabetically before the other, we return a value of –1, if the other comes first, we return 1, and if they are equal, we return a 0. Our first function uses array notation:

```c
#include <stdio.h>

int stringComparison(char *, char *);

void main() {
    char str1[20]="Hellob";
    char str2[20]="Hello";
    cout << stringComparison(str1,str2);
}

int stringComparison(char *s1, char *s2) {
    int i,j;
    for(i=0,j=0; s1[i]!="\0" && s2[j]!="\0"; i++,j++) {
        if(s1[i]<s2[j]) return -1;
        else if(s1[i]>s2[j]) return 1;
    }
    if(s1[i]=="\0" && s2[j]=="\0") { //strings are equal
        return 0;
    }
    else if(s1[i]=="\0") { //string s1 is shorter
        return -1;
    }
    else { return 1;
    }
}
```
The string comparison function is quite elaborate, making many different distinctions. Since in this implementation i always equals j, we can avoid the double indices. However, they make it easier to translate the function to pointer arithmetic:

```c
int stringComparison(char *s1, char *s2) {
    for( ; *s1!='\0' && *s2!='\0'; s1++,s2++) {
        if(*s1<*s2) return -1;
        else if(*s1>*s2) return 1;
    }
    if(*s1=='\0' && *s2=='\0') { //strings are equal
        return 0;
    }
    else if(*s1=='\0') { //string s1 is shorter
        return -1;
    }
    else {
        return 1;
    }
}
```

The strcmp function explained on p. 106 of K&R uses a different and preferable logic. It also speeds up because it returns a negative value, zero, or a positive value depending on the result of the comparison, which saves us some case distinctions.

### 15.6. Multidimensional Arrays

C supports multi-dimensional arrays. Assume we want to store the sales figures for the last ten years of a dot.bomb product. We could simply store them in an array of size 120, but that would mean quite a bit of work if we want to compare with last year’s sales. Thus, we want to access the data with two indices, one for the year, and the other for the month. We can do this by declaring

```c
double sales[10][12];
```

Since 10 and 12 are numbers that are hard to interpret, we can use preprocessor commands that explain these numbers once for all:

```c
#define YEARS 10
#define MONTHS 12
```

//place this at the beginning of the relevant file

```c
double sales[YEARS][MONTHS]  = {
    { 0.04, 0.1, 0.08, 0.05, 0.10, 0.1, 0.5, 0.9, 2.10, 0.12, 0.4, 0.8, 0.74},
    { 3.4, 0.1, 1.8, 0.4, 0.05, 0.10, 1.1, 0.5, 0.9, 0.5, 0.12, 0.12, 0.7, 0.14, 0.74},
    { 0.72, 0.5, 0.08, 0.24, 0.51, 0.4, 0.9, 0.5, 0.92, 0.10, 0.12, 0.4, 0.5, 0.4, 0.7, 0.47},
    //some more lines like these
    { ... },
};
```

This gives us ten lines of monthly sales figures.

If a two-dimensional array is passed to a function, the parameter declaration in the function must include the number of columns, because otherwise the compiler cannot
find the location of the array element that needs updating. This is because the two-
dimensional array is arranged sequentially in memory, and two get to an element
sales[5][10], the compiler uses the equivalent pointer notation sales+5*MONTHS+10.
Thus, the following function definitions are appropriate:

```c
double average(double sales[YEARS][MONTHS]);
double average(double sales[][MONTHS]);
double average(double (*sales)[MONTHS]);
```

In general, the first dimension of a multi-dimensional array does not need to be given, all
the others have to be specified.

16. The C Preprocessor

Before a file is submitted to the compiler, the C preprocessor changes the file. We need
to learn the most basic commands of the Preprocessor.

16.1. Include

The command
```
#include <stdio.h>
```
includes a header file in lieu of the include command. We need two types of header files,
those defining library functions and values (such as EOF, FILE) and header files that we
ourselves produced. The pointy brackets around stdio.h indicate that the preprocessor
should look for this files in a directory with all the other library header files. Were we to
say
```
#include “myheader.h”
```
then the preprocessor would look for the file myheader.h in the current (or working)
directory. It is important to remember that the preprocessor places the whole header file
in lieu of the include statement. If there is a problem with one of your header files, the
compiler will report this problem in the file that included the header file.

16.2. Define

The command
```
#define MONTHS 12
```
looks for all occurrences of the first string “MONTHS” and replaces it with the rest of the
line “12” wherever the first string occurs. Use of this command allows you to adapt your
program very quickly, e.g. if you were to change your program to a different calendar
(e.g. 13 lunar months per year), you can do that changing only this one line in the code.
Also, your program becomes more readable, since a reader knows why you are using the
constant 12. Thus, the define command is highly recommended for two reasons: you can
change constants quickly and you are not using magic numbers. You can congregate all
your definitions in a single header file and then include this header file in all your
implementation files.
16.3. Conditional Compilation

If the compiler sees the same header file twice, the compiler will complain that you are redefining all the functions and refuse to proceed. To deal with this problem, you need to conditionally compile, by enclosing the whole header file with these commands

```c
//file myfile.h

#ifndef __myfile.h.15Nov2002
#define __myfile.h.15Nov2002

// put myfile.h here
#endif
```

If the compiler encounters the file myfile.h the first time, the constant “__myfile.h.15.Nov2002” has not been defined. The result of the first line (the if not defined line) is therefore true, and the compilation proceeds. The first thing to be done is to define the token “__myfile.h.15.Nov2002” to be true (i.e. 1). The second time the compiler sees myfile.h, the token is set to 1, and the compiler skips over everything between the if not defined line and the end if line. The header is thus ignored and there are no more errors.

16.4. Other Preprocessor Commands

We have by no means exhausted the capabilities of the preprocessor. Sooner or later, but not right now, you need to learn about macros and you need to use conditional compilation more exhaustively. See K&R, p.88 – p.93.

17. Storage Classes, Linkage, Memory Management

17.1. Scope

Recall that scope determines the visibility of a variable definition. With other words, the scope of a definition is the area in the code in which the name of the variable defined can be used. The following are the scope rules:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-level Identifiers</td>
<td>Extends from the point of declaration to the end of the source program file</td>
</tr>
<tr>
<td>Formal Parameters in Functions</td>
<td>Extends from the point of declaration to the end of the function.</td>
</tr>
<tr>
<td>Formal parameters in a function prototype</td>
<td>Extends from the point of declaration to the end of the prototype</td>
</tr>
<tr>
<td>Block (Local) Identifiers</td>
<td>Extends from the point of declaration in a</td>
</tr>
</tbody>
</table>
17.2. Local Variables

Most variables are local variables, that is, they are only defined within a function (or within a block within a function), and their scope does not extend beyond the function (or the block).

```c
int super;

void main() {  
  int n;
  if(n>5) {  
    int i = 3;
    i++;
  }  
  else { 
    int i = 4;
    i--;
  }
  return;
}
```

This code does nothing useful but illustrates some of the scope rules. The scope of super is the entire program starting with its declaration. There are two i variables, both defined within the blocks set up by the if statement. Their scope are the two branches of the if statement. The scope of n is the body of the function main. Variable super is not a local variable, since it is defined outside of any function. Its scope extends to the whole code.

17.3. Storage Duration

Storage duration refers to the lifespan of a variable. An automatic variable is created every time that the program flow enters its scope. A static variable is created only once and destroyed when the program exits.

Example:

```c
#include <stdio.h>

void tryStatic();
```
int main() {  
  int iteration;
  for(iteration = 0; iteration < 4; iteration++) {  
    printf("Here comes iteration %d:\n", iteration);
    tryStatic();
  }
  return 0;
}

void tryStatic() {  
  int automaticVariable = 0;
  static int staticVariable = 0;
  printf("The automatic variable is %d and the static variable is %d\n",
         automaticVariable++, staticVariable++);
  return;
}

The code is quite simple. Main calls the function tryStatic four times. tryStatic contains two variables, one of automatic and the other of static type. The automatic variable is created every time that the function tryStatic is invoked. Therefore, the effect of the increment is lost between function calls. However, the static variable is only created once, and then with value 0. However, since the variable is not destroyed and not reinitialized, the increments stick. Here is the output:

Here comes iteration 0:
The automatic variable is 0 and the static variable is 0
Here comes iteration 1:
The automatic variable is 0 and the static variable is 1
Here comes iteration 2:
The automatic variable is 0 and the static variable is 2
Here comes iteration 3:
The automatic variable is 0 and the static variable is 3

There is some use for static variables, but mostly, you will use automatic variables.

17.4. External Variables

Automatic variables are defined within a function and are not available outside the function. They also do not retain a value between function invocations. A static variable within a function retains a value, but it does not live outside the function. An external variable is defined outside of any function. It is visible to any function, but the compiler needs to be informed about its existence. The compiler assumes the existence for functions in the same file, but for each new file in which we use the variable, we need to re-"declare" it. Thus, an external variable is defined exactly once, but declared in each file (or function) that needs it. To declare an external variable, we use the keyword external. Here is an example. We use a large character array to store a terminal display. This will be our external variable. It would be better practice to pass the character array as a function parameter, but then it is difficult to come up with small programs that should use external variables.

File main.c:
#include <stdio.h>
#include "display.h"
#include "parameters.h"

char terminal[ROWS][COLS]; // definition of external variable

int main() {
    processInput();
    displayInput();
}

File display.h:
void displayInput();

File display.c:
#include <stdio.h>
#include "display.h"
#include "parameters.h"

void displayInput() {
    external char terminal[ROWS][COLS]; // declaration
    int i, j;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%c", terminal[i][j];
        }
        printf("\n");
    }
    return;
}

File parameters.h

#ifndef parameters
#define COLS 80
#define ROWS 24

#define parameters
#endif

This is not the complete code, but it gives you the idea. File main.c defines the external variable. File display.c uses it in its function displayInput(), therefore it is declared there with the keyword external. It would be much better programming style to pass the character array as a variable.

18. Structures

User defined structures allow users to create their own types. This is a feature heavily used in larger programming projects. As an example, assume that you want to create a
library database. A library database needs to work with the concept of “books”. As things develop in the implementation of the project, the concept of the book will change. Assume for the moment, that the library’s view of a “book” is simply this: A book has a set of authors, a title, a catalog number, a publisher, and a publication year. Since we have not yet broached the issues of variable length lists, we simplify by assuming that a book has as most one author. We define a “book” by cobbling together all the separate data we have for a book:

```c
struct book {
    char author[MAXSTRLENGTH];
    char title[3*MAXSTRLENGTH];
    char catalogNumber[CATALOGENTRYLENGTH];
    char publisher[MAXSTRLENGTH];
    int publicationDate;
}
```

The string sizes are defined in a parameters file. We can define / declare books just like other variables, but the name of the type book is

```c
struct book.
```

Thus, we define

```c
struct book classy = {
    "Gaius Julius Caesar",
    "De Bello Gallico",
    "HA.3.156.C1a",
    "self-published",
    -30
};
```

The components of a book are called fields. Thus, to print out the author of struct book classy, we can say:

```c
cout << classy.author << endl;
```

We can pass a structure as an argument to a function as in

```c
void screenDump(struct book aBook);
```

where struct book is the type of the variable aBook.

### 18.1. Type Definition

C allows you to give names to your own types. By writing

```c
typedef struct book Book
```

you make Book a synonym for struct book. I recommend as a matter of style to capitalize the types that you have defined. Typedef works also with types that C provides and there are sometimes good uses for this facility. Thus

```c
typedef int Length
```

makes Length into a synonym for integers. You can integrate the typedef statement and the definition of the type in a single statement as in:

```c
typedef struct book { 
    char author[MAXSTRLENGTH];
    char title[3*MAXSTRLENGTH];
    char catalogNumber[CATALOGENTRYLENGTH];
    char publisher[MAXSTRLENGTH];
    int publicationDate;
} Book;
```
18.2. Pointers to Structures

Most C programmers define structures, but manipulate them using pointers. They will also tend to organize substructures via pointers.

Let us define yet another simple data structure that describes the personal banking account at a bank. Each customer has a checking and a saving account. We say

```c
typedef struct _account {
    float checkingBalance;
    float savingsBalance;
    int accountNumber;
} Account;
```

This creates a struct _account and gives it the name Account. By saying

```c
Account acc1 = {0.0,0.0, 1001}
```

you create an account with balances 0 and account number 1001. If we define a pointer to an Account structure and let it point to the account

```c
Account *pAcc1 = &acc1;
```

we can access it fields by saying

```c
(*pAcc1).checkingBalance = 500.00;
```

to reflect a $500.00 initial deposit into the account. This notation is a little awkward, and C programmers prefer the shortcut

```c
pAcc1->savingBalance = 100.00;
```

which reflects a deposit of $100.00 into savings. The arrow is composed of the minus sign and the greater sign.

18.3. Dynamic Memory Allocation

So far, we have allocated memory by declaring (and possibly initializing a variable). For reasons entirely opaque to you now this is called allocation “on the stack”. Advanced C programming offers another way of allocating storage, called dynamic memory allocation, namely “on the heap”. Memory allocation on the heap is under programmer control, so that the programmer needs to allocate and deallocate memory within the code. Memory allocation on the stack is entirely automatic. If an automatic variable gets out of scope, then the memory is automatically freed. The programmer who uses dynamic memory allocation is responsible for freeing the storage. If this does not happen, then the programmer has created a memory leak. A memory leak is not important for small programs, since memory is reclaimed when the program exits. However, a long running program can request more and more memory until the system needs to satisfy the memory requests by allocating space on the disk instead of main memory. As a consequence, the execution of this and any other program running is slowed by a factor of roughly 10,000.

We allocate memory using the malloc function and free it using the free function, both residing in the library file stdlib.h. malloc returns a void pointer, that needs to be cast to the right type.

First, a simple example allocating an array:
```c
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h> //

int main()
{
    int * piArr, iSizeOfArray, iSum=0, i;
    printf("How many integers in the array?\n");
    scanf("%d",&iSizeOfArray);
    piArr = (int *) malloc(iSizeOfArray * sizeof(int));
    if(!piArr) {
        printf("Memory Allocation Failed.\n");
        exit(0);
    }
    for(i=0;i<iSizeOfArray;i++) piArr[i] = i*i;
    for(i=0;i<iSizeOfArray;i++) iSum += piArr[i];
    printf("The sum of the array elements is %d\n",iSum);
    free(piArr);
    return 0;
}
```

This program uses a dynamically allocated integer array. The array is allocated with a call to malloc that needs the size of the array (in bytes) as a parameter. To calculate this value, we multiply the number of integers in the array (iSizeOfArray) with the size of an integer. Since the sizes of types vary from platform to platform, we use the sizeof function. If malloc cannot allocate memory, it returns the null pointer. This does not happen often, but we need to check for it in the following if statement. The next lines show how to use the dynamic array. As you see, once we have allocated the memory, we access the dynamically allocated array as we would an array allocated on the stack. At the end of the program, we free the memory we used. This is strictly speaking not necessary, since all memory is freed once a process is terminated, but memory leaks are such a curse on C (and C++) programmers, that it is morally abhorrent to not free memory.

The next program shows what happens when you do not free up memory. Don’t run this program on a machine that you do not want to reboot.

```c
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int * piArr, i, iSum=0, iCount=0;
    while(piArr=(int*)malloc(10000*sizeof(int))) {
        // do something with the array so that the compiler does not optimize this away
        for(i=0; i<10000; i++) piArr[i]=i;
        for(i=0; i<10000; i++) iSum+=piArr[i]*i;
        iCount++;
        printf("Allocation %d succeeded.\n",iCount);
    }
    return 0;
}
```
This program will run very fast, but then suddenly come to a crawl, where bursts of allocation messages are displayed. If you run other programs at the same time, you should see a performance degradation. The performance degradation is not very bad, because the virtual memory in a modern OS works well, but even after the program has finished, the virtual memory tables are filled up and it takes some time for the computer to become its typical perky self. Each while loop in the program allocates 40KB of memory and without virtual memory the RAM is quickly eaten up.

**18.4. Dynamic Memory Allocation and Structures**

We can use malloc to allocate storage for a user defined typed variable as well. The following code gives an example

```c
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>

typedef struct _account {
    float checkingBalance;
    float savingsBalance;
    int accountNumber;
} Account;

void dump(Account a) {
    printf("Account Number %6d:\n", a.accountNumber);
    printf("Checking %5.2f\n",a.checkingBalance);
    printf("Savings %5.2f\n",a.savingsBalance);
}

Account * makeAccount(float checking, float saving, int number) {
    Account * pAccount = (Account *) malloc(sizeof(Account));
    pAccount->accountNumber = number;
    pAccount->checkingBalance = checking;
    pAccount->savingsBalance = saving;
    return pAccount;
}

int main() {
    Account acc = {100.00, 2000.00, 123345},
    *paSample=makeAccount(400, 100, 123476);
    dump(acc);
    dump(*paSample);
    free(paSample);
    return 0;
}
```

In main, we have two allocation of the account structure. The one on the stack is easy to understand, the other one is hidden in the makeAccount function. There, we allocate memory asking malloc to give us sizeof(Account) bytes.
Dynamic memory allocation also allows us to solve some memory problems that we had with names. Recall that using fixed size fields tended to allocate too much storage without removing completely the possibility of running out of space for an unusual name. We now give an example how to implement a name structure such that only storage actually used is given:

-----main.cpp-----
#include "stdafx.h"
#include "name.h"

int main() {
    Name *myName = getName("Dr.", "Hannibal", "the Cannibal", "Lector", "Jr.");
    writeOut(myName);
    printf("\n");
    deallocate(myName);
    free(myName);
}
-----main.cpp-----

-----names.h-----
#ifndef name_h_Nov_2002
#define name_h_Nov_2002

typedef struct _name {
    char * prefix;
    char * firstName;
    char * middleName;
    char * lastName;
    char * suffix;
} Name;

Name *getName(char * prefix, char * firstName, char * middleName, char * lastName, char * suffix);
void writeOut(Name * pName);
void deallocate(Name * pName);
#endif
-----names.h------

------names.cpp-----
#include <string.h>  // for strcpy, strlen
#include "stdafx.h"
#include "name.h"

Name *getName(char * pre, char * fN, char * mN, char * lN, char * suf) {
    Name *retVal = (Name*) malloc(sizeof(Name));
    retVal->prefix = (char *) malloc(sizeof(char)*strlen(pre)+1);
    strcpy(retVal->prefix, pre);
    retVal->firstName = (char *) malloc(sizeof(char)*strlen(fN)+1);
    strcpy(retVal->firstName, fN);
    retVal->middleName = (char *) malloc(sizeof(char)*strlen(mN)+1);
    strcpy(retVal->middleName, mN);
    retVal->lastName = (char *) malloc(sizeof(char)*strlen(lN)+1);
    strcpy(retVal->lastName, lN);
    retVal->suffix = (char *) malloc(sizeof(char)*strlen(suf)+1);
    strcpy(retVal->suffix, suf);
    return retVal;
}
------names.cpp-----
We define the Name datastructure in name.h. All five fields are now pointers to characters. The function getNam e creates a new Name data structure on the heap. It does so by allocating memory for all five fields and for itself. It first allocates storage space for the name data structure in the first call to malloc. If we assume that pointers use 4B of storage, the name data structure uses 20B of storage space. Afterwards, we allocate the space for the fields. Notice that we use one more byte in each field to store the end of the string symbol `\0`. If you forget this, your program will work most of the time, but might suddenly crash. But what we allocate, we need to deallocate at the end. If we just call the free function on the name data structure, this call will release the 20B or so memory of the name data structure, but the storage for the fields remain allocated. This is a classical memory leak. The deallocate function called before freeing a name data structure prevents this by explicitly freeing up the memory used by the fields. If you forget to call deallocate, your program will still run. But if your program allocates and deallocates many name data structures, then it will claim more and more storage. Possibly, the calls to malloc do not succeed (something our sample program does not test for) or performance of the memory unit slows down dramatically affecting all programs running. (If you want to test for malloc not succeeding, test whether the pointer returned to by malloc is zero. If this would happen to our program, it would crash because we would try to access memory at location 0. For a simple program like this, this is acceptable. For a real program, probably not.)

18.5. Example: The Point Structure

To show you how you might want to put things together, we are going to define a point data structure. A point is a point in a two-dimensional plane. We are going to implement the point data structure as if we were to use it to represents pixels in a graphics environment. Our implementation is (very loosely) inspired by the Microsoft Foundation Class CPoint. We need to be able to create points from their coordinates or from another point or points, we need to calculate the distance between points, and we need to move points. As a matter of programming style, we manipulate points by pointers to them. Listing Point.h gives the definition of the structure point, which I have
called “Point”. I then give three methods that create a new point. Afterwards, I calculate the distance between points and a screen dump method that allows me to display a point on the screen. Unfortunately, printf is not flexible enough for me to mix it with pointer output. If I want to do that, then I would have to access the components of Point from printf, which is dangerous if I ever wanted to change the structure. As you can see, main.cpp merely tests my structure.

-----Point.h-----

#ifndef point_h_Nov_2002
#define point_h_Nov_2002

typedef struct _point {
  int xc;
  int yc;
} Point;

Point * makePoint(int x, int y);
/* creates a new point with coordinates (x,y) */
Point * copyPoint(Point * ppPt);
/* makes a copy of the argument */
Point * midPoint(Point * ppPt1, Point * ppPt2);
/* creates a new point between the two arguments */
void move(Point * ppPt, int xOffset, int yOffset);
/* changes the point by adding xOffset to the x-coordinate and yOffset to the y-coordinate */
float distance(Point *ppPt1, Point *ppPt2);
/* calculates the distance between the two points */
void writePoint(Point *ppPt);
/* writes the point to standard output */

#endif
-----Point.h-----

-----Point.cpp-----

#include "stdafx.h"
#include "Point.h"
#include <math.h>  // for sqrt

Point * makePoint(int x, int y) {
  Point *retVal = (Point*) malloc(sizeof(Point));
  retVal->xc = x;
  retVal->yc = y;
  return retVal;
}

Point * copyPoint(Point * ppPt) {
  Point *retVal = (Point*) malloc(sizeof(Point));
  }
```c
Point * midPoint(Point * ppPt1, Point * ppPt2) {
    Point *retVal = (Point*) malloc(sizeof(Point));
    retVal->xc = (ppPt1->xc + ppPt2->xc)/2;
    retVal->yc = (ppPt1->yc + ppPt2->yc)/2;
    return retVal;
}

void move(Point * ppPt, int xOffset, int yOffset) {
    ppPt->xc += xOffset;
    ppPt->yc += yOffset;
    return;
}

float distance(Point *ppPt1, Point *ppPt2) {  
    return sqrt((ppPt1->xc-ppPt2->xc)*(ppPt1->xc-ppPt2->xc)+(ppPt1->yc-ppPt2->yc)*(ppPt1->yc-ppPt2->yc));
}

void writePoint(Point *ppPt) {
    printf("(%d,%d)\n",ppPt->xc,ppPt->yc);
}
```

```c
#include "stdafx.h"
#include "Point.h"

int main() {  
    Point * ppt1 = makePoint(8,7), * ppt2 = copyPoint(ppt1), *ppt3;
    move(ppt2,15,9);
    ppt3 = midPoint(ppt1, ppt2);
    printf("The distance between ");
    writePoint(ppt1);
    printf(" and ");
    writePoint(ppt2);
    printf(" is %f\nand the midpoint is ",distance(ppt1,ppt2));
    writePoint(ppt3);
    printf(".\n");
    free(ppt1);
    free(ppt2);
    free(ppt3);
    return 0;
}
```

### 19. Function Pointers

C allows you to pass pointers to functions. The storage location of a function is the beginning of the code that implements the function. The syntax of function pointers is complicated and counter-intuitive, to say the least, but the ability to pass a pointer to a
function in C is a tool that comes in sometimes handy. Posting parentheses around the name of the function indicates a pointer to a function. Thus

```c
float (*func)(float); /* Pointer to a function returning float with * one argument of type float */
long (*var)(long, long); /* Pointer to function returning long with * two long arguments */
int* (*foo)(float *); /* Pointer to a function that returns a * pointer to an integer and that takes * a pointer to a float as argument */
```

Declaration can become quite complicated, since you might have arrays of pointers to pointers of functions of pointers to arrays etc. but these declarations strike me as suspicious. If you cannot easily read it, then use typedef statements to make it readable.

### 19.1. Example

We write functions that calculate approximately the maximum and the minimum of a function within an interval. The functions are declared in grapher.h and implemented in grapher.cpp. Basically, we evaluate the function at STEPS points equidistributed in the interval given by the second and third parameter. The results are only exact, if the maximum or minimum fall on one of the evaluation points, but most functions (of interest) are monotonic, so that this is the case, and the others don’t vary too much so that the results give us reasonably accurate information. The way the maximum is calculated is a frequent pattern and you do well to remember it. Also, notice the way that the function argument is called. To evaluate the function at point \(x\), we use \((\ast\text{func})(x)\).

```c
-----grapher.h-----
#ifndef _grapher_h_Nov_2002
#define _grapher_h_Nov_2002
double maximum(double (*func)(double), double minArg, double maxArg);
double minimum(double (*func)(double), double minArg, double maxArg);
/* calculates the maximum and the minimum of the function within minArg
and maxArg*/
#endif
-----grapher.h-----

-----grapher.cpp-----
#include "stdafx.h"
#include "parameters.h"
#include "grapher.h"
#include "math.h"

double maximum(double (*func)(double), double minArg, double maxArg)
{
    int i;
double maxi = (*func)(minArg), delta = (maxArg-minArg)/(STEPS-1);
    for(i=1; i<STEPS; i++) {
        if(((*func)(minArg+i*delta) > maxi) maxi = (*func)(minArg+i*delta);
    }
    return maxi;

```
double minimum(double (*func)(double), double minArg, double maxArg) {
    int i;
    double mini = (*func)(minArg), delta = (maxArg - minArg) / (STEPS - 1);
    for (i = 1; i < STEPS; i++) {
        if ((*func)(minArg + i * delta) < mini) mini = (*func)(minArg + i * delta);
    }
    return mini;
}

--- grapher.cpp ---
--- main.cpp ---
#include "stdafx.h"
#include "parameters.h"
#include "grapher.h"

double func(double x) { return 4; }
int main() {
    printf("exp has range [%f,%f] in [-1,10] \n",
           minimum(exp, -1, 10),
           maximum(exp, -1, 10));
}
--- main.cpp ---