4 Arrays

4.1 Introduction

In this chapter you will learn what an array is, namely a method of storing many values under a single variable name, instead of using a specific variable for each value. We will begin by declaring an array and assign values to it.

In connection with arrays you will have great use for loops, by means of which you can efficiently search for a value in the array and sort the values.

Arrays is a fundamental concept within programming which will frequently be used in the future.

4.2 Why Arrays

An array is, as already mentioned, a method of storing many values of the same data type and usage under a single variable name. Suppose you want to store temperatures measured per day during a month:

```
12.5
10.7
13.1
11.4
12.1
...
```

If you didn't know about arrays, you would need 30 different variable names, for instance:

```
tempa = 12.5
tempb = 10.7
tempc = 13.1
tempd = 11.4
tempe = 12.1
...
```

This is a bad option, especially if you want to calculate the average temperature or anything else. Then you would need to write a huge program statement for the sum of the 30 variables.

Instead, we use an array, i.e. one single variable name followed by an **index** within square brackets that defines which of the temperatures in the array that is meant:

```
temp[1] = 12.5
temp[2] = 10.7
temp[3] = 13.1
temp[4] = 11.4
temp[5] = 12.1
...
```

The name of the array is temp. The different values in the array are called **elements**.

In this way we can use a loop, where the loop variable represents the index, and do a repeated calculation on each of the temperatures:

```
for (i=1; i<=30; i++)
{
   //Do something with temp[i];
}
```

The loop variable i goes from 1 to 30. In the first turn of the loop i has the value 1, which means that temp[i] represents temp[1], i.e. the first temperature. In the second turn of the loop i has the value 2 and temp[i] represents the second temperature.

By using a loop the amount of code required will not increase with the number of temperatures to handle. The only thing to be modified is the number of turns that the for loop must execute.

In the code below we calculate the average of all the temperatures:

```
iSum = 0;
for (i=1; i<=30; i++)
{
   iSum += temp[i];
}
dAvg = iSum / 30;
cout << dAvg;
```

The variable iSum is set to 0 since it later on will be increased by one temperature at a time. The loop goes from 1 to 30, i.e. equal to the number of elements in the array. In the loop body the variable iSum is increased by one temperature at a time. When the loop has completed, all temperatures have been accumulated in iSum. Finally we divide by 30 to get the average, which is printed.

4.3 Declaring an Array

Like for all variables, an array must be declared. Below we declare the array temp:

```
double temp[31];
```

The number within square brackets indicates how many items the array can hold, 31 in our example. 31 positions will be created in the primary memory each of which can store a double value. The indeces will automatically be counted from 0. This means that the last index is 30. If you need temperatures for the month April, which has 30 days, you have two options:

1. Declare temp[30], which means that the indeces goes from 0 to 29. 1st of April will correspond to index 0, 2nd of April to index 1 etc. 30th of April will correspond to index 29. The index lies consequently one step after the actual date.
2. Declare temp[31]. Then 1st of April can correspond to index 1, 2nd of April to index 2 etc. 30th of April will correspond to index 30. The date and index are here equal all the time. This means that the item temp[0] is created "in vain" and will never be used.

It is no big deal which of the methods you use, but you will have to be conscious about the method selected, because it affects the code you write. We will show examples of both methods.

Note that, in the declaration:

```
double temp[31];
```

all elements are of the same data type, namely double. For arrays *all elements all items always have the same data type.*

4.4 Initiating an Array

You can assign values to an array already at the declaration, e.g.:

```
int iNo[5] = {23, 12, 15, 19, 21};
```

Here the array iNo will hold 5 items, where the first item with index 0 gets the value 23, the second item with index 1 the value 12 etc.

The enumeration of the values must be within curly brackets, separated by commas.

As a matter of fact it is redundant information to specify the number of items to 5 in the declaration above, since the number of enumerated values is 5. Therefore you could as well write:

```
int iNo[] = {23, 12, 15, 19, 21};
```

An enumeration within curly brackets can only be written in the declaration of an array. For instance, the following is erroneous:

```
double dTemp[4];
dTemp = {12.3, 14.1, 11.7, 13.8};
```

In the following code section we declare an array of integers and assign values to the array in the loop:

```
int iSquare[11];
for (int i=0; i<=10; i++)
{
   iSquare[i] = i*i;
}
```

The array iSquare is declared to hold 11 items of integer type. The loop then goes from 0 to 10. In the first turn of the loop i is =0 and the item iSquare[0] gets the value 0*0, i.e. 0. In the second turn of the loop i is =1 and iSquare[1] gets the value 1*1, i.e. 1. In the third turn of the loop the item iSquare[2] gets the value 2*2, i.e. 4. Each item will contain a value equal to the square of the index.

4.4.1 Index outside the Interval

As a C++ programmer you must yourself keep track of the valid index interval. The result could be disastrous if you wrote:

```
temp[35] = 23.5;
```

This means that we store the value 23.5 in the primary memory at an adress that does not belong to the array, but might belong to a memory area used for other data or program code. If you run such a code the system might in worst case break down and you will have to restart the computer.

4.5 Copying an Array

Suppose we want to copy the temperatures from the array with April's values to an array with June's values. You cannot copy an entire array in this way:

```
dblTempJune = dblTempApr;
```

You will have to copy the values item by item by means of a loop:

```
for (int i=1; i<=30; i++)
{
   dblTempJune[i] = dblTempApr[i];
}
```

Here the loop goes from 1 to 30 and we copy item by item for each turn of the loop.

4.6 Comparing Arrays

What is meant by comparing whether two arrays are equal? They must contain item values that are equal in pairs. In the following code section we compare the two arrays with April's and June's temperatures:

```
int eq = 1;
for (int i=1; i<=30; i++)
```

```
{
   if (dblTempJune[i] != dblTempApr[i])
      eq = 0;
}
```

Here we let the variable eq reflect whether the two arrays are equal, where the value 1 corresponds to ”equal” and 0 ”not equal”. From the beginning we assign eq the value 1, i.e. we presume the arrays to be equal. Then in the loop we go through item by item in the two arrays and checks if they are equal in pairs. The if statement checks if they are different. If so, the variable eq is set to 0, otherwise nothing is changed. If two items happen to be different, the variable eq will have the value 0 after the loop has completed. If however all pairs of items are equal, the statement:

```
eq = 0;
```

will never be executed, and the variable eq will remain =1. We could then complete our program with output about the result:

```
if (eq == 1)
   cout << "The arrays are equal";
   else
   cout << "The arrays are different";
```

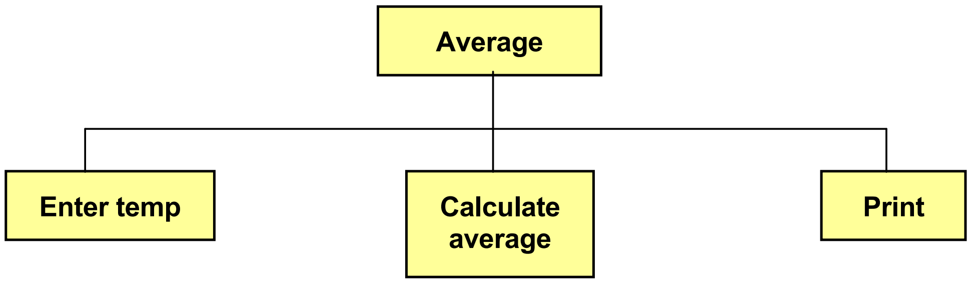
It is *not* possible to in one single statement check whether the arrays are equal:

```
if (dblTempJune == dblTempApr)
```

You must compare item by item like in the code above.

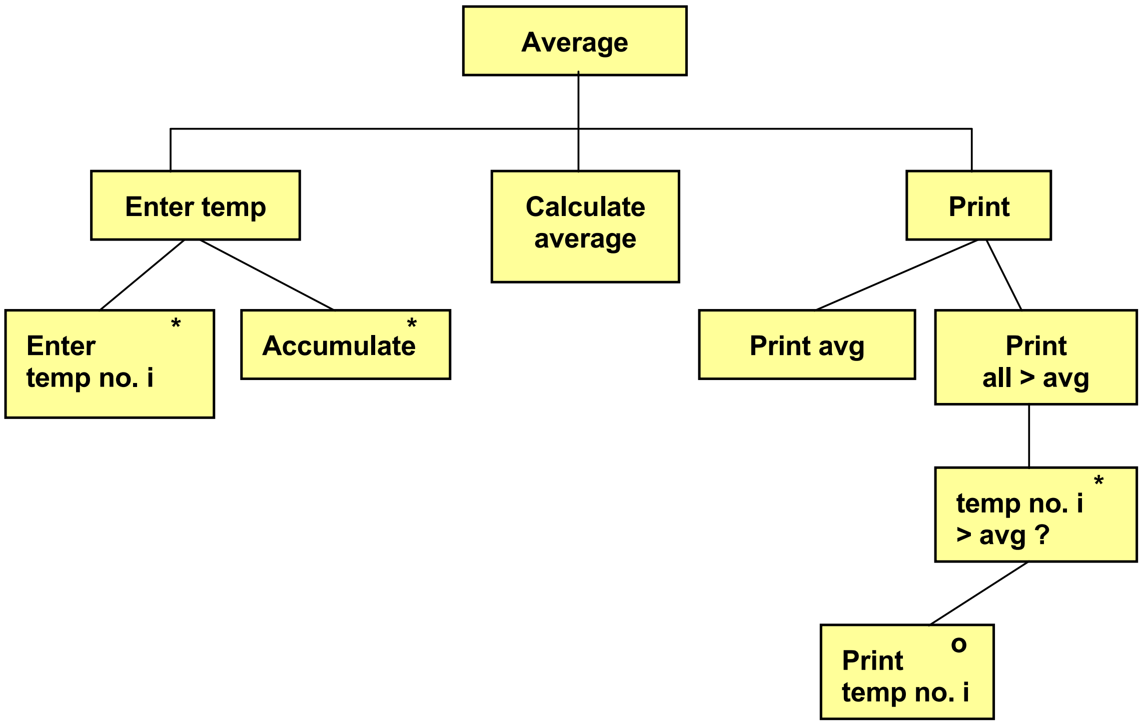
4.7 Average

We will now write a program that reads temperatures to an array from the user and then calculates the average of all temperatures. The program should then print the average and all temperatures exceeding the average. We begin with a JSP graph:



We have made an overview JSP that mainly describes the procedure.

Since we will calculate an average, we need the sum of all temperatures. We choose to sum the temperatures at the time of entry, which is made in a loop:



The average calculation is simple. We don't have to detail it. However the output is a little more complicated. First we print the calculated average. Then we write a loop which in turn checks each item of the array against the calculated average. If temperature number i is greater than the average, we print temp no. i.

"I studied English for 16 years but... ...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

Here is the code:

```
#include <iostream.h>
void main()
{
  // Deklarations
  const int iNoOfDays = 30;
  double dAvg, dSum = 0;
  double dblTempApr[iNoOfDays + 1];
  int i;
  // Entry and calculation
  for (i=1; i<= iNoOfDays; i++)
  {
    cout << "Temperature day " << i;
    cin >> dblTempApr[i];
    dSum += dblTempApr[i];
  }
  dAvg = dSum / iNoOfDays;
  // Printout
  cout << "Average temperature: " << dAvg << endl;
  cout << "Temperatures exceeding average: " << endl;
  for (i=1; i<= iNoOfDays; i++)
  {
    if (dblTempApr[i] > dAvg)
      cout << "Day no.: "<<i<<" temp:
        "<<dblTempApr[i]<<endl;
  }
}
```

First we declare a constant iNoOfDays which is set to 30 and is used later in loops and average calculation. The variable dAvg is used for storing of the calculated average. The variable dSum is initiated to 0 since it will be increased by the value of each entered temperature. The array dblTempApr is declared to hold 31 items, which means that we can let the index values correspond to the day numbers of the month. The item with index 0 will consequently not be used. Finalyy we declare the variable i, which is used as loop counter.

The first loop takes care of entry of the temperatures. The loop counter goes from 1 to 30 and each entered temperature is stored in the array. The variable dSum is increased by the entered temperature.

At loop completion the variable dSum contains the accumulated total of all temperatures, which is divided by the number of days, which gives the average.

The printout starts with the average. Then comes the last loop which goes from 1 to 30. For each turn of the loop we check whether temperature number i exceeds the average. If so, the day number is printed, which is equal to the index value, together with the corresponding temperature.

4.8 Sales Statistics

We will now give an example that shows how to use arrays and conditional input in a while statement. The situation is this:

A company has a number of salesmen, each with a salesman number in the interval 1-100. When a salesman has sold for a specific amount, he enters his salesman number and the sales amount. This goes on until you terminate the entry with Ctrl-Z. Then a summary should be printed with one line per salesman showing total sales amount.

Furthermore, a fee per salesman should be calculated. If the sales amount is below 50000:- the fee is 10% of the sales amount. If the amount is greater, the salesman will get a fee which is 10% of the first 50000:- plus 15% of the amount exceeding 50000:-. If for instance the sales amount is 70000:- the fee is 10% of 50000:- which gives 5000:- plus 15% of the exceeding 20000:- which is 3000:-. The total fee will in this case be 8000:-.

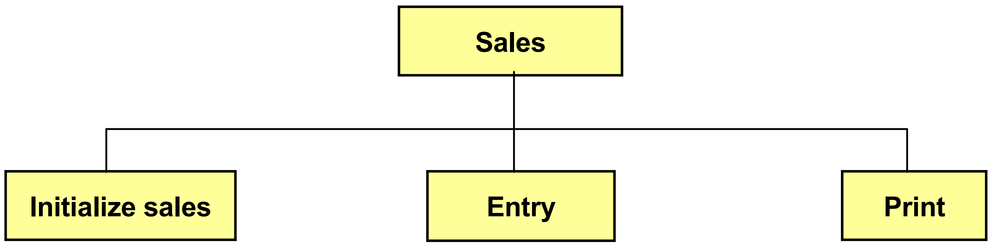
An entry from different salesmen could look like this:

```
78 10000
32 500
2 12000
100 25000
78 60000
2 1000
5 60000
```

```
The printout will then be:

Number     Amount      Fee
======     ======     =====
     2      13000      1300
     5      60000      6500
    32        500        50
    78      70000      8000
   100      25000      2500
```

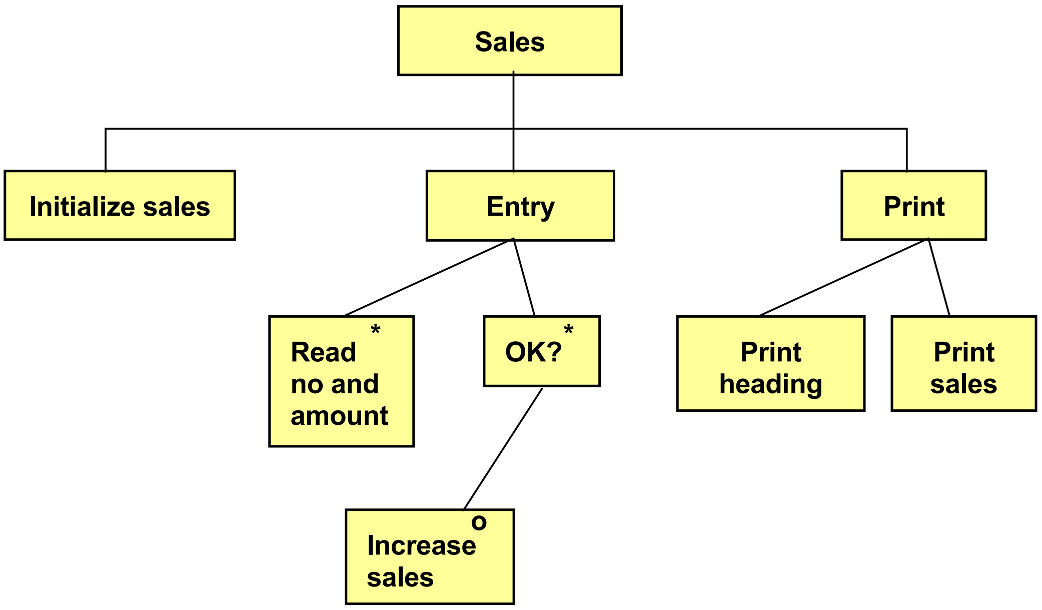
We begin with a JSP graph:



We will use an array called sales with 100 items, where each item corresponds to a certain salesman. For each entered sales amount the array item corresponding to the salesman number should be increased by the entered amount. Therefore we must initialize the entire array, i.e. set all its items = 0 so each salesman's accumulated amount starts with 0. Note that in C++ the items of an array are not automatically set to 0 at the declaration. The declaration only allocates memory space.

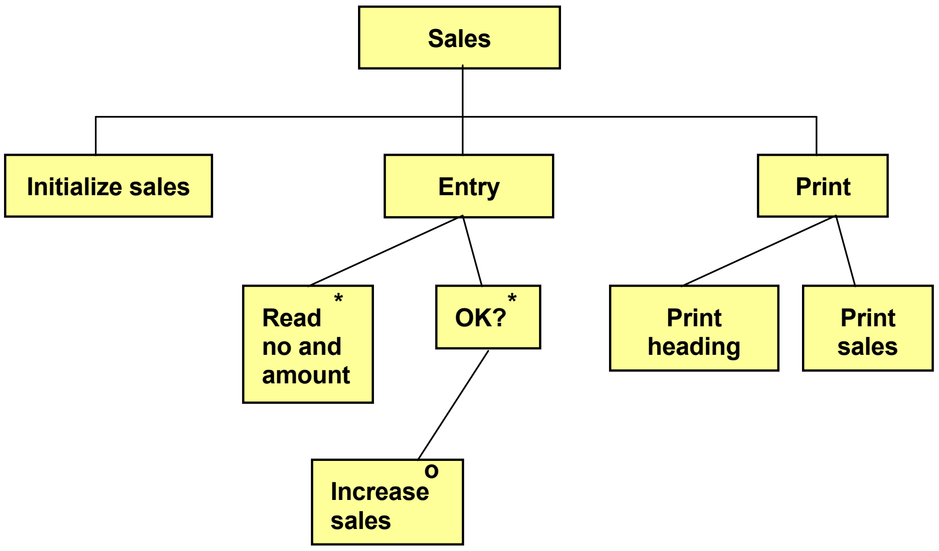
Any unpredictable values present in these memory addresses will be retained until you initialize them.

Then we read salesman number and sales amount. This is made in a loop so that we can go on with entry of values as long as we want. We break down the "Entry" box:

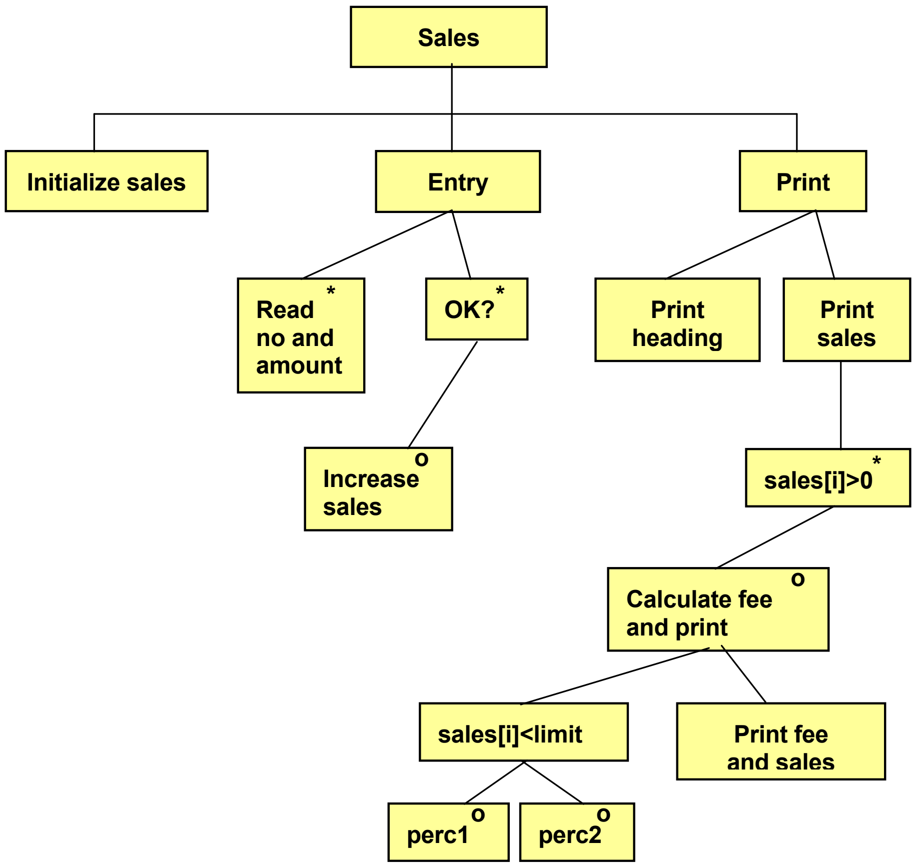


We read a salesman number and sales amount, one at a time. In the box "OK?" we check that the salesman number is between 1 and 100 and that the sales amount is not negative. If OK, we increase the corresponding item in the sales array.

The box ”Print” has been detailed by first printing the heading and then the sales values. In connection with the printing we check that the sales amount is not 0. Salesmen having sold nothing should not be included in the printed summary. We break down the box “Print sales” further:



If the sales amount exceeds 0, we calculate the fee and print one line in the summary. At fee calculation we will now pay attention to whether the amount exceeds the limit 50000:-, which gives still another detailed level:



If the sales amount is below the limit 50000:- the lower percent 10% will be applied. Otherwise the greater percent 15% will be applied to the exceeding amount. When the fee calculation is finished, the fee and sales amount are printed.

Here is the code:

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
  const int iMaxNo=100;
  const double dLimit=50000, perc1=0.1, perc2=0.15;
  double sales[iMaxNo], dAmount, dFee;
  int i, nr;
  //Initialize array
  for (i=0;i<iMaxNo; i++)
    sales[i] = 0;
  //Enter salesman info
  while (cin>>nr && cin>>dAmount)
  {
    if (nr<1 || nr>iMaxNo || dAmount<0)
      cout << "Input error" << endl;
    else
      sales[nr-1] += dAmount;
  }
  //Print summary
  cout << endl
    << "Number      Amount         Fee" << endl
    << "======      ======      =====" << endl;
  for (i=0; i<iMaxNo; i++)
  {
    if (sales[i] > 0)
    {
      if (sales[i] <= dLimit)
        dFee = perc1 * sales[i];
      else
        dFee = perc1*dLimit + perc2*(sales[i]-dLimit);
      cout << setw(4) << (i+1) <<
        setprecision(0) << setiosflags(ios::fixed) <<
        setw(13) << sales[i] << setw(10) <<
        dFee << endl;
    } // end if
  } // end for-loop
} // end main
```

The constant iMaxNo = 100 represents the number of salesmen and is used for loop control. The constand dLimit = 50000 is used for the fee calculation. The constants perc1 and perc2 are the two different percentages used for the fee.

The array is declared to contain 100 items with the index 0-99. Here, salesman no.1 will correspond to index 0, salesman no. 2 index 1 etc.

The variable dAmount is used for entry of sales amounts and the variable dFee for fee calculation. The variable i is used as loop counter and the variable nr to entry of salesman numbers.

Then the sales array is initialized, where we set all items to 0.

Entry of salesman numbers and amounts is done in a while statement. If entry is successful, the loop continues. If you however you press Ctrl-Z, the while condition is false and the entry loop is interrupted, enabling the program to continue with the next statement.

Inside the loop the program checks if the salesman number is less than 1 or greater than 100. This is for safety reason to guarantee that we don't go outside the index interval of the array, since the salesman number gives the index value of the array. In addition, the program also checks if the sales amount is less than 0. If any of these conditions are true, the text "Input error" is printed and the user can enter new values. If everything is OK, the sales item is increased by the entered amount. Note that we decrease the salesman number by 1 to get the correct item in the array.

At loop completion (Ctrl-Z), the program goes on with printing the summary heading.

Then the last loop will calculate the fee and print one line per salesman. First in the loop, we check the sales total to be greater than 0, otherwise no line for that salesman is printed. Then we check if the sales total is less than the limit 50000. If so, the fee is calculated as the lower percent multiplied by the sales total. Otherwise the fee is calculated as the lower percent times the limit 50000 plus the greater percent multiplied by the difference between the sales total and the limit 50000.

When the inner if statement has completed the fee calculation is complete and the program writes a line with salesman number (i+1), sales total (sales[i]) and fee. Note that, when we print the salesman number, we must use the index value increased by 1, since the index value all the time is 1 less than the salesman number.

We have also used the formatting functions from the header file iomanip.h to get a nice layout with straight columns.

4.9 Product File, Search

We will now examine a situation where we use several arrays in parallel. We will build a simple product file, where we use an array for the product id:s and another array for the product prices. We will organize it so that a product in the product id array with for instance index 73 has its price in the price array at the same index position, i.e. 73:

```
Prodid     Price
2304      152,50
2415       75,40
3126       26,80
...
```

The array with product id:s is called iProdid and the price array dPrice.

Suppose we want to be able to enter a product id and get the corresponding price. Then we must search the iProdid array. Look at the following code section:

```
while (cin >> iProd)
{
   for (i=1; i<=100; i++)
   {
      if (iProd == iProdid[i])
         cout << "The price is: " << dPrice[i] <<
            endl;
   }
}
```

In the while condition we read a product id to the variable iProd. This allows for repeated entry of product id:s until you interrupt with Ctrl-Z.

The inner loop goes from 1 to 100 and checks one item at a time in the product id array to equal the entered product id. The loop counter i going from 1 to 100 is used as index in the product id array and represents the different product id:s in the array. Note that we use the same i-value in the price array as in the product id array. If for instance we encounter equality for the 23rd product, also the 23rd price should be printed, since the variable i then has the value 23.

4.10 Two-Dimensional Array

In many business systems on the market customer discounts are based on the customer group that the customer belongs to, and the product group for the bought product. Different customer segments will then get different discount profiles.

Example:

		Product group		
		1	**2**	**3**
Customer	**1**	10	12	13
group	**2**	13	14	15
	3	14	16	17

If for example a customer of customer group 3 orders a product from product group 2, he will get 16% discount.

To store a discount matrix in this way in a program, you will need a two-dimensional array. Such an array has two indeces, where the first index could be thought of as representing the lines in the matrix, and the second index the columns:

```
double dDiscount[5][8];
```

Here we have declared a two-dimensional array named dDiscount with 5 lines (index 0-4) and 8 columns (index 0-7).

To assign values to the different array items, we can write:

```
dDiscount[1][1] = 10;
dDiscount[1][2] = 12;
...
```

Note that we all the time must use two indeces for dDiscount.

Suppose that we want a program section where the user can enter customer group and product group and the program should respond with the corresponding discount percent:

```
cout << "Enter customer group ";
cin >> cgrp;
cout << "Enter product group ";
cin >> pgrp;
cout << "Discount: " << dDiscount[cgrp][pgrp];
```

Suppose that, when this program section is run, the user enters 3 and 2. The variable cgrp will get the value 3 and pgrp the value 2. These two values are used as indeces in the two-dimensional array. If we use the values from the discount matrix above, we will get the printout:

```
Discount: 16
```

Let us now turn the problem the other way so that the user enters a discount percent and that the program responds with corresponding customer group and product group. A prerequisite to this is that each percent only occurs once in the matrix, which is not very likely, but it shows how to search a two-dimensional array. The code will be:

```
cout << "Enter percent: ";
cin >> dPerc;
for (i=1; i<=5; i++)
{
  for (j=1; j<=8; j++)
  {
    if (dDiscount[i][j] == dPerc)
      cout << "Product group " << i <<
             " and customer group " << j;
  }
}
```

First the user is prompted for a percent which is stored in the variable dPerc. A double loop then performs the search for the entered percent, where the outer loop goes through the lines of the matrix and the inner loop through the columns of the matrx. The loop counter i thus corresponds to line index and j to column index. The inner loop goes through all its values before the outer loop changes its value, which means that the matrix is searched one line at a time, where all items in the line are checked. The if statement checks if the matrix item equals the entered percent (the variable dPerc). If equal, the corresponding loop counters i and j are printed, which correspond to customer group and product group.

4.11 Sorting

Many times it is easier to work with an array if the items are sorted, especially when searching for a specific value. For instance, in the product id array in an earlier section, if the product id:s are sorted by size, the process of finding a certain product, and consequently also its price, is much faster than for an unsorted array. We will therefore discuss array sorting.

We will as example use an array with 6 items named iNos:

```
int iNos[6] = {5,3,9,8,2,7};
```

The items of the array are not yet sorted. We want to write a program that sorts them by size. The problem is that a computer program is not capable of, like the human eye, scan the values and instantly sort them. We have to write code that systematically compares two values in turn and interchange their positions in the array.

We will use two variables, l and r, which are indeces in the array and points to two items. l means “left” and r “right”. These items are compared in pairs, and if the right item is less than the left, they will interchange their positions in the array:

0	1	2	3	4	5
5	3	9	8	2	7
l	r				

The indeces of the array have the interval 0-5. The variable l has from the beginning the value 0 and r the value 1, i.e. they point on the two first items of the array. Since the right item is less than the left (3 is less than 5), they are interchanged:

0	1	2	3	4	5
3	5	9	8	2	7
l		r			

We then increase r by 1, so that it points to the value 9, while l remains. 9 is not less than 3, so no interchange is made. r is again increased by 1:

0	1	2	3	4	5
3	5	9	8	2	7
l			r		

Neither this time there is no interchange, since 8 is greater than 3. We increase r by 1 again. Then r points to the value 2, which is less than 3, so the items are interchanged:

0	1	2	3	4	5
2	5	9	8	3	7
l				r	

We increase r by 1 again. 7 is greater than 2, so the items remain:

0	1	2	3	4	5
2	5	9	8	3	7
l					r

Now r has gone through all values, and as result we have got the least item on index position 0 in the array. We have performed *a series of comparisons*.

Now we increase l by 1 and perform a new series of comparisons, where r goes from index position 2 to 5:

0	1	2	3	4	5
2	5	9	8	3	7
	l	r			

Here 9 is not less than 5, so we increase r by 1 and so forth. When r arrives at index position 4, the right item (3) is less than the lef t (5), so we interchange them.

When two series of comparisons have been completed, we have got the two least items on the two first positions:

0	1	2	3	4	5
2	3	9	8	5	7

Once again we increase l by 1 and let r go from the item immediately to the right of l to the last item of the array. This is repeated until we compare the two last items of the array:

0	1	2	3	4	5
2	3	5	7	8	9
				l	r

After completion of the last comparison, the entire array is sorted.

To summarize, l goes from 0 to the next last position of the array, while r goes from the position to the right of l to the last position of the array. We use an outer loop for the l-values and an inner for the r-values:

```
for (l=0; l<=4; l++)
{
   for (r=l+1; r<=5; r++)
   {
      //Check if right is less than left
      //and in that case interchange
   }
}
```

l goes from 0 (first position of the array) to 4 (next last position), while r goes from l+1 (the position to the right of l) to 5 (last position).

The check whether the right is less than the left is made by an if statement:

```
if (iNos[r] < iNos[l])
```

The interchange is a little tricky. We cannot directly let two variables change values. We must use an intermediary storage, a temporary variable that temporary stores one of the values:

```
temp = iNos[l];
iNos[l] = iNos[r];
iNos[r] = temp;
```

Here we let the variable temp get the value of the left item, then we let the left item get the value of the right item, and finally we let the right item get the value of the temporary variable, i.e. the old left value. This triangular exchange has the effect that the two array items interchange their values. After the triangular exchange the value of temp is of no concern.

Here is the complete code:

```
for (l=0; l<=4; l++)
{
   for (r=l+1; r<=5; r++)
   {

      if (iNos[h] < iNos[v])

      {
         temp = iNos[l];
         iNos[l] = iNos[r];
         iNos[r] = temp;

      }
   }
}
```

After completion of the double loop the array items are sorted.

4.12 Searching a Sorted Array

For a sorted array, when searching for a particular item, we don't need to scan the entire array from the first to the last position and check each single value. For a small array with only 6 items like in the previous example, there is no big deal. But what if we have a product array with thousands of product id:s. Then the search time would be considerable and our program would be regarded as having bad performance.

We will use a more refined method, namely to halve the index interval repeatedly. We go in to the middle item of the array and check if the searched value is to left or right. When having selected which half to continue with, we halve that part again. This is repeated until we find the searched value. The execution time will be reduced considerably.

Suppose we have a product array with 31 items (index 0-30)

```
   0     1     2   ...              15   ...                        30

  2314  2345  3123                 4526                            6745
```

The index values are shown above the product id:s.

Suppose we are searching for product id 5321. We begin with checking whether 5231 is less than the middle item with index position 15, namely 4526. If so we go on with the left interval, otherwise the right. In our case we use the right interval, which we halve and get index position 22 (index must always be an integer). We check whether the searched product id 5231 is greater or less than the product id on position 22, etc.

When having divided the interval enough number of times, we will have found the searched item, or otherwise it does not exist in the array.

We will now discuss the code for this. First we declare some variables:

```
int l=0, r=30, iFound=0, iPos, iSrch;
```

The variables l and r are index positions of the array. l is the left end point of the interval, which from the beginning is 0. r is the right which from the beginning is 30.

The variable iFound is used to indicate whether or not the searched product id has been found. The value 0 means not yet found, and the value 1 means that it has been found.

The variable iPos is the index for the found product id. The variable iSrch is the searched product id, which is read from the keyboard:

```
cout << "Enter the searched product id: ";
cin >> iSrch;
```

We then perform some introductory checks to figure out if the searched product id is first or last in the array:

```
if (iSrch == iProdid[0])
{
  iPos = 0;
  iFound = 1;
}
if (iSrch == iProdid[30])
{
  iPos = 30;
  iFound = 1;
}
```

As long as the product id has not been found, we will divide the interval:

```
while (!iFound)
{
```

First we calculate the middle of the interval:

```
int iMid = l + (int)((r-l)/2);
```

From the beginning r is =30 och l=0. (r-l)/2 then makes 15. Since this division might give a decimal number, we perform a type cast with (int) within parenthesis in front of the division. In that way we ensure that the index always is an integer. This half interval is added to the value of l. Since l by the time not necessarily equals 0 all the time, this means that we take the left endpoint of the interval and add half the interval, i.e. we calculate the middle point of the current interval, which is stored in the variable iMid.

Then we check if there is a match to the middle item of the interval:

```
if (iSrch == iProdid[iMid])
{
  iFound = 1;
  iPos = iMid;
}
```

If the searched product id equals the product id at the position given by the variable iMid in the product array iProdid, there is a match, and the variable iFound is set =1 and the found position is stored in the variable iPos.

In case of no match, we check if the searched product id is to the left or to the right of the middle point:

```
if (iSrch > iProdid[iMid])
  l=iMid;
  else
  r=iMid;
}
```

If the searched product id is greater than the product id at position iMid, we set the left endpoint (the variable l) to the value of iMid, which means that we move the left endpoint to the new middle value, and we have a new interval which is the right half of the previous interval. Otherwise we focus on the left half of the interval and we let the right endpoint (the variable r) get the value of iMid. In both cases the loop performs another turn.

By the time the loop has divided the interval so many times that we certainly get a match in the statement:

```
if (iSrch == iProdid[iMid])
```

provided that the user has entered a product id that is present in the array.

Here is the entire program:

```
#include <iostream.h>
void main()
```

```
int l=0, r=30, iFound=0, iPos, iSrch;
int iProdid[31] = {2314, 2345, 3123, ... 6745};
cout << "Enter the searched product id: ";
cin >> iSrch;
if (iSrch == iProdid[0])
{
   iPos = 0;
   iFound = 1;
}
if (iSrch == iProdid[30])
{
   iPos = 30;
   iFound = 1;
}
while (!iFound)
{
   int iMid = l + (int)((r-l)/2);
   if (iSrch == iProdid[iMid])
   {
      iFound = 1;
      iPos = iMid;
   }
```

```
      if (iSrch > iProdid[iMid])
         l=iMid;
         else
         r=iMid;
   }
}
```

4.13 Summary

In this chapter we have learnt about arrays. We have learnt to declare arrays and assign values to them. We have also seen the advantage with using loops in connection with arrays.

We have also studied an algorithm for sorting the items of an array. You should try to really understand the algorithm. You should also remember how to write the sorting code in C++.

A sorted array is very efficient when searching for a particular value. We have studied how to do a smart search in an array. The search method presented here is often called binary search.

4.14 Exercises

1. Write a program where you declare an array with 10 integers and then read both positive and negative values to the array. The program should then:
   a) print the first, the fifth and the tenth item.
   b) print the sum of all items.
   c) print the numbers in reversed order.
   d) change sign of all negative numbers to positive, and then print them.
   e) ask for a number and then print all numbers less than that number.
   f) ask for an index and print the corresponding item.
   g) ask for a number and print the index of that number. We assume that the user enters a number that exists in the array.
   h) move the first item to the last position of the array.
2. Write a program that prompts the user for a month number and prints the number of days of that month. Use an array like this:
   `const int iDaysInM[] = {31, 28, 31, 30, …};`
3. Write a program that creates random temperatures between 15 and 25 degrees, one temperature per day of the month July. The temperatures are stored in an array. The program should then create a new array for August and copy the July values (see the section 'Copying an Array'). Finally the August values should be printed.
4. Expand the previous program to compare the values of July and August and check if the arrays are equal.
5. Complete the previous program so that one of the August temperatures is changed before the comparison.
6. Complete the previous program with calculation of the average temperature during August. The program should also print all temperatures greater than the average.
7. Declare an array that contains the following nine densities for metal alloys:
   1.5 2.8 4.6

5.7	7.9	8.3
8.6	8.8	8.9

Write a program that prompts the user for a density and prints the one closest below the entered density.

8. Write a program that fills an array with 25 integers between 0-9. The program should then ask the user for a number and print the number of occurrences of that number in the array.
9. Start with the Sales Statistics program earlier in this chapter and add logic so that if a salesman has sold for more than 100000, he will get a fee also including 20% of the portion exceeding 100000.
10. Expand the previous program to also print the number of sales per salesman.
11. Expand the previous program to also print the average sales amount per salesman.
12. Declare an array containing some product id:s, and a price array with unit prices per product. Write a program that prompts the user for a product id and prints the corresponding price. If the product id is not found, a suitable message should be printed.
13. Complete the previous program so that the user also can enter a quantity of the product and get the total price for the purchase.
14. Complete the previous program with a discount matrix according to the section 'Two-Dimensional Array'. The user should be able to enter a product group and a customer group, and the corresponding discount should be deducted.
15. Write a program that creates 10 random rolls of a dice and stores them in an array. The array should then be sorted and printed.
16. Start with the program that searches a sorted array for a product id. Complete the program with initialization of the array with as many product id:s as required, and run it.
17. Go on with the previous program and make it print the index position of the found product id.
18. Improve the previous program so that if you enter a product id not existing in the array, a suitable message should be printed.
19. Expand the previous program with a price array that contains the prices of each product, and with a printout of the price of the found product id.