8 Pointers

8.1 Introduction

A pointer is a special kind of variable which contains a memory address to for instance a number instead of directly refer to the number. That implies a detour to the value via the memory address.

This might sound unnecessarily complicated, but implies a number of advantages like for instance more efficient program code, faster execution and memory saving. Especially in object oriented programming you get these advantages when copying objects or sending objects to functions. Object oriented programming is however beyond the scope of this course.

The pointer concept is unique for C++. It is for instance not present in the programming languages Visual Basic or Java. As a consequence C++ might be felt more complicated than other languages.

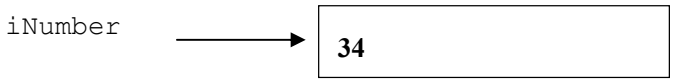
In this chapter we will aqcuire basic knowledge about pointers. We will learn how to use pointers to different data types, how to declare pointers and assign values. We will examine the anology between pointers and arrays and how to use pointers as parameters to functions.

Finally we will touch the subject dynamic memory allocation, which actually does not closely relate to pointers, but still often is used in connection with pointers.

8.2 What Is a Pointer

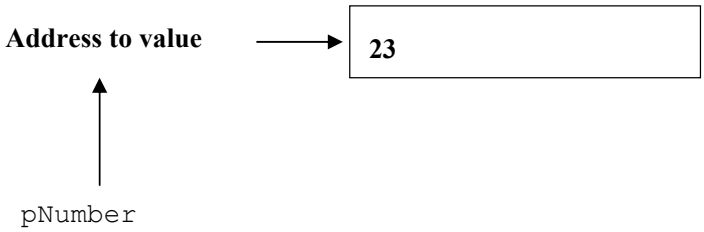
A pointer is a variable of a special kind which only can contain a memory address of the primary memory. This memory location in turn contains a value of some kind.

Le tus first study the situation for a common variable:



In the figure above we have the variable iNumber which contains the actual value of the variable, in our example 34.

Let us now focus on the corresponding pointer:



In the figure above we have a pointer named pNumber. It contains an address in the primary memory. If we go to that address, there is a number, 23 in our example.

8.3 Declaring a Pointer

Below we declare the pointer variable pNumber:

```
int* pNumber;
```

The asterisc (*) indicates that it is a pointer. `int*` means that it is a pointer to an integer value. You must always specify the data type pointed to by the pointer variable. Below we declare a pointer to a double value:

```
double* pPrice;
```

Below we declare a pointer to a char value:

```
char* pChr;
```

You can as well place the space in front of the asterisc. The declarations above could be written:

```
int *pNumber;
double *pPrice;
char *pChr;
```

You can use both variants.

8.4 Assigning Values to Pointers

An ordinary variable, say iNo, is assigned a value in the usual way:

```
int iNo = 23;
```

To get the address to the variable iNo, we use the & operator. The expression &iNo gives the address to the variable iNo.

In the declaration you can specify the memory location to be pointed at by a pointer variable:

```
int* pNumber = &iNo;
```

Here we create a pointer variable named pNumber and assign the address of the variable iNo to it. The variable iNo and the pointer variable pNumber now points to the same memory location, which means the value 23.

Note that in a pointer declaration you can't directly assign a fixed value:

```
int* pNumber = 23;   //wrong
```

since currently there is no specific memory location pointed to by pNumber. However, when pNumber has got its memory address, we can change the value in the location indicated by pNumber:

```
*pNumber = 25;
```

Here we must remember to use the asterisc together with the name of the pointer variable. The program then understands that it is the value that is to be changed.

Compare this to this erroneous statement:

```
pNumber = 25;   //wrong
```

This would mean that we updated the address pointed to by pNumber. The address 25 would be pointed to, which of course is erroneous.

We have introduced two operators in connection with pointers:

*	means 'the content of'
&	means 'the address to'

In the same way we can write:

```
//ordinary variable:
double dPrice = 34.75;
//pointer variable with the same address as
//dPrice:
double* pPrice = &dPrice;
//change the price:
*pPrice = 45.25;

// ordinary variable:
char cChr = 'x';
//pointer with the same address as cChr:
char* pChr = &cChr;
//change the character:
*pChr = 'y';
```

When printing a value pointed to by a pointer variable, you use:

```
cout << *pNumber;   //prints 25
```

This means 'print the content of pNumber'.

To print the address pointed to by a pointer variable you write:

```
cout << pNumber;
//prints the address, e.g. 0x0066FDF0
```

The printed address is in hexadecimal format. Normally we don't have to bother about the exact address. The only thing to remember is whether we mean 'the address to' or 'the content of'.

8.5 Addresses and char Pointers

We will now take a look at how pointers work in connection with string variables, i.e. arrays of char type. We declare a string array named cName:

```
char cName[] = "John Smith";
```

We then declare a char pointer named pName which points to the same text as the content of cName.

```
char* pName = cName;
```

Why didn't we use the & operator in front of cName like in the previous example? The explanation is that an array actually is a pointer. When using the name of the array, cName, it is interpreted as a pointer to the first item of the array. So when writing the statement:

```
pName = cName;
```

it means that we let the pointer pName get the same address as the pointer (array) cName.

8.6 cout and char Pointers

The print function cout has some peculiarities you ought to know when printing strings. The statement:

```
cout << pName;
```

should actually print the address in hexadecimal format of pName. But cout performs a reinterpretation. It takes the content in the memory location pointed to by pName, i.e. the character 'J', and prints character by character until the null character is found. This means that the entire name 'John Smith' is printed. Compare the statement:

```
cout << cNamn;
```

which gives the same result, which we discussed in the Strings chapter.

The statement:

```
cout << *pName;
```

correctly prints the content of the memory location pointed to by pName, but it only takes that character. This means that only 'J' is printed.

The statement:

```
cout << &pName;
```

prints the address of the memory location in which pName is stored.

The statement

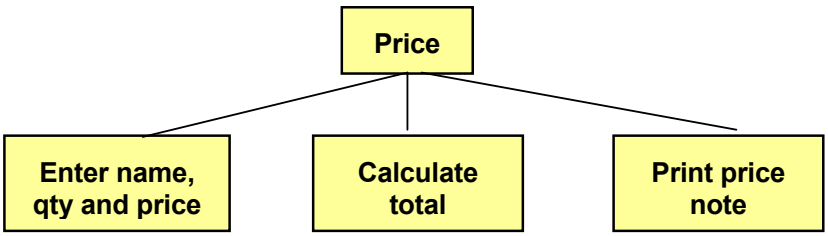
```
cout << &cName;
```

prints the address of the memory location where the name 'John Smith' is stored.

8.7 Price Program with Pointers

We will now create a program which reads quantity and unit price of a product from the user, and the name of the user. The program will then calculate the total price of the product and print a personal price note on the screen. We will use pointer variables.

The logical process is given by the following JSP graph:



Here is the code:

```
#include <iostream.h>
void main()
{
  //Declare variables and corresponding pointers
  //Set the pointers to point to the address of the
  //corresponding variable
  int iNo;
  int* pNo = &iNo;
  double dPrice, dTotal;
  double* pPrice = &dPrice;
  double* pTotal = &dTotal;
  char cName[20];
  char* pName = cName;

  //Read data and store in the pointer variables
  cout << "Enter your name: ";
  cin.getline(pName, 19);
  cout << "Enter quantity and unit price: ";
  cin >> *pNo >> *pPrice;

  //Calculate total
  *pTotal = *pNo * *pPrice;

  //Printout of personal price note
  cout << "Dear " << pName << ", your price is " <<
    *pTotal << " kr." << endl;
}
```

Let us say that we enter 'John Smith', quantity 5 and unit price 12. Then the printout will be:

```
Dear John Smith, your price is 60 kr.
```

8.8 Pointer Arithmetics

By pointer arithmetics we mean how to increment and decrement a pointer, i.e. how to make a pointer to an array move stepwise from item to item.

Let's say that we have an array of integers:

```
int iNos[] = {5, 12, 3, 24, 125, 8};
```

Here we have declared the array iNos to contain six items. Suppose an integer takes 4 bytes in the working memory. If the first integer (5) is stored in memory address 2000, then the next integer (12) will be stored in memory address 2004, the third in 2008 etc.

Now we declare an integer pointer to point to the first item of the array:

```
int* pNos = iNos;
```

pNos now has the value 2000 (the address of the first item in the array).

We can now perform the following pointer arithmetic:

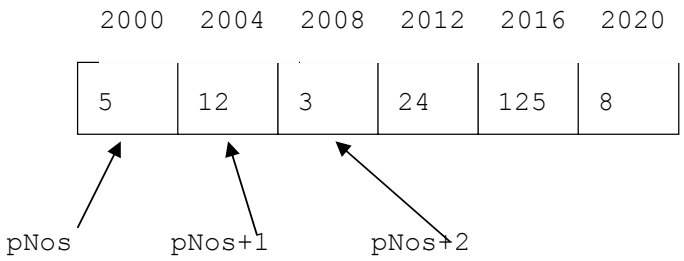
```
pNos++;
```

which means that pNos is increased by 1. You might then be fooled to beleive that pNos now has the value 2001. But that is not the case. Since we have declared pNos as a pointer to integer, the system knows that each integer requires four bytes, and pNos is consequently increased by 4 making the new value of pNos be 2004. This implies that pNos now points to the second item of the array.

If the array would have been declared as double and pNos declared as a pointer to double, the system knows that each double number takes 16 bytes, and a stepwise increase would add 16 to pNos.

As a consequence the data type pointed to by a pointer is of ultimate importance when using pointer arithmetics.

The situation with integers is summarized by the following figure:



To print the numbers you can use the statement:

cout << *pNos << *(pNos+1) << *(pNos+2) …

Note that, when printing the second, third item etc. we must enclose pNos+1, pNos+2, etc. with a parenthesis, so the address is calculated first before taking the asterisk ('the content of') into account, otherwise the content of pNos, i.e. the first item of the array, would be increased by 1, 2 etc.

We may as well use a loop for the printout:

```
for (int i=0; i<6; i++)
  cout << *pNos++ << endl;
```

Here we use the loop counter i, which goes from 0 to 5, i.e. as many turns as there are items in the array. For each turn of the loop the content of the memory address pNos is printed, and the pointer is increased by 1, i.e. it moves to the next item of the array.

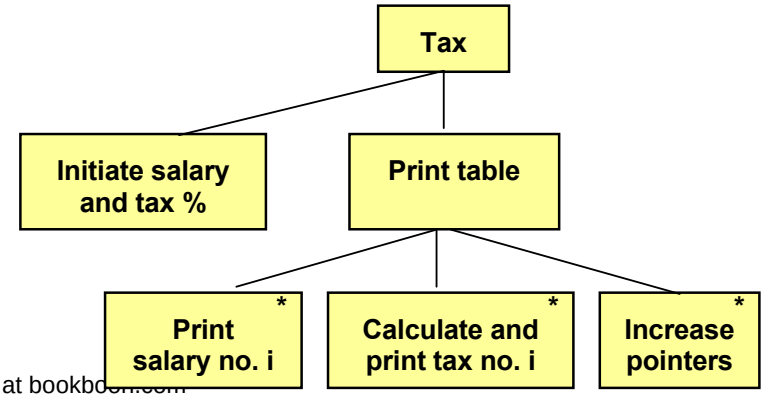
8.9 Tax Program

We will not create a full-featured tax calculation program, but only calculate the tax deduction for a number of monthly salaries based on fix tax percentages.

We store a number of monthly salaries in an array and corresponding tax percentages in another. We multiply each salary by corresponding tax percentage, which gives the tax deduction amount. Each salary and tax percent are then printed in table format.

We will use pointer arithmetics to move from item to item in the arrays.

The program logic is explained by the following JSP graph:



First we initiate the arrays with salaries and tax percentages. When printing the table we go through one item at a time of the arrays and print salary and corresponding tax, which is calculated by multiplying salary by tax percentage. Then we increase both pointers to proceed to the next salary and tax percentage.

Here is the code

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
      int iSal[] = {14000, 15000, 16000, 17000, 18000, 19000};
      int* pSal = iSal;
      double dTax[] = {0.32, 0.34, 0.35, 0.36, 0.365, 0.37};
      double* pTax = dTax;
      cout << setiosflags(ios::fixed) << setprecision(2);
      cout << "  Salary        Tax" << endl;
      for (int i=0; i<6; i++)
      {
            cout << setw(8) << *pSal << setw(10) << (*pSal *
                  *pTax) << endl;
            pTax++;
            pSal++;
      }
}
```

The include files are iostream.h for input and output, and iomanip.h to be able to format the printout into a nice table.

In main() we initiate the array iSal with a number of salaries and the array dTax with a number of tax percentages. We also declare a pointer pSal which is set to point to the first item of the array iSal, and a pointer pTax for the dTax array.

The first cout statement fixes the decimal point and states two decimals. The second cout statement prints the heading of the table.

The for-loop goes from 0 to 5, i.e. as many turns as there are items in the arrays. The cout statement sets the width of the printed numbers with the setw() function, prints the salary by means of the pointer, and multiplies the salary by the tax percentage and prints the result. Note that we use asterixes in front of the pointers to get 'the content of'.

After the cout statement we increase the two pointers by 1, i.e. we move the pointers to the next item of the arrays. The program automatically remembers that one of the pointers is an integer pointer and the other a double, and moves them the corresponding number of bytes in the primary memory.

8.10 Functions and Pointers

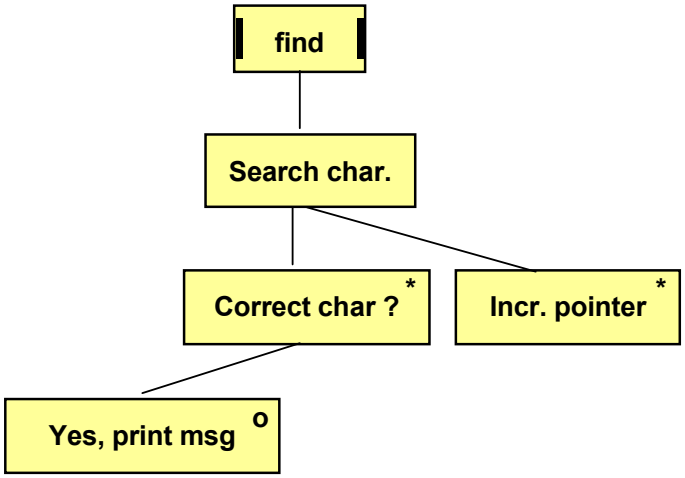
Many times you use pointers as parameters to functions. That means that you send the memory address to the function instead of sending all data. Especially in object oriented programming when you want to send an object to a function which is several Mbytes big, it is a great advantage to only send a memory address instead. It saves both memory and time.

You may remember from the Functions chapter that it was possible to send reference parameters to a function, which means that you send the address to the value instead of the value itself. It is very similar to sending pointers. An advantage with pointers is however that it is possible to use pointer arithmetics, which gives more compact code.

You should however keep in mind that if you send a pointer to a function, and the function updates the value, the value will be updated also after completion of the function. Unintentional update of a value can however be prevented by means of the keyword const. More about this later.

We will create a function which searches for a particular character, for instance @, in a string to check if it is an email address. The string is sent as pointer to the function find(). A for-loop in the function goes through the string, character by character, and checks if it is an @. If @ is found, a suitable text is printed on the screen.

We begin with a JSP graph:



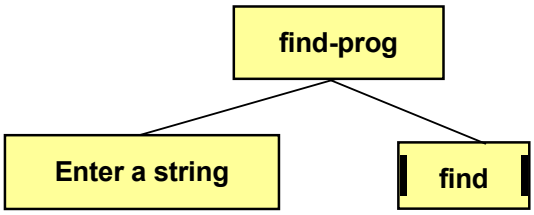
Here is the code:

```
void find(char* str)
{
   for (int p=0; p<8; p++)
   {
      if (*str == '@')
         cout << "It is an email address";
      str++;
   }
}
```

The function takes a parameter str which is a pointer to a char variable. That means that we don't send the entire string but only the address to the first character of the string. The for-loop goes from 0 to 7, which is a limitation, but we have done it as simple as possible to illustrate the use of pointers.

The if statement checks if the content of the address str is the @ character. If so, the suitable text is printed. At the end of the loop the pointer is increased by 1, i.e. it is set to point to the next character.

We will now write an entire program, where the user is prompted for a text that is sent to the function. First a JSP graph:



Here is the code:

```
#include <iostream.h>
void find(char* str);
void main()
{
  char cString[9];
  char* pString=cString;
  cout << "Enter a text: ";
  cin.getline(cString, 8);
  find(pString);
}
void find(char* str)
{
  for (int p=0; p<8; p++)
  {
    if (*str == '@')
      cout << "It is an email address";
    str++;
  }
}
```

The program first declares a string array named cString. The pointer pString is set to point to the first character of the string. When the user has entered a text (maximum 8 characters), the function find() is run.

We will now illustrate the risk of having the value outside the function be changed. We will modify the function find() so that it replaces @ by a blank:

```
void find(char* str)
{
  for (int p=0; p<18; p++)
  {
    if (*str == '@')
      *str = ' ';
    str++;
  }
}
```

Thus, the function chages the value of the string. If we would print the email address in main() after completion of the function find() with the purpose of having the original email address printed, then we would get a wrong result. The statement

```
cout << "The email address is " << pString << endl;
```

will give a printout with a blank instead of @.

One way of preventing modification of a value in the function is to use the keyword const in front of the parameter in the function header:

```
void find(const char* str)
```

If we still would write a statement in the function that tries to modify the value, we will get a compilation error. Many programmers use const in this way to clearly indicate that the value is not changed in the function.

8.11 Dynamic Memory

When declaring an array in Visual C++ we have until now been forced to specify the number of items of the array to allocate the correct memory space. That is called static memory.

Many times we cannot in advance predict the number of items needed for the array, for instance when reading a number of product id:s to an array from a file, where the number of products is unknown.

The solution to this problem is to use dynamic memory. The dynamic memory area is capable of assigning space during the execution of the program and not at the compilation. Different amounts of memory might be needed at different execution occasions. A disadvantage is however that the program cannot guarantee that the requested amount of memory is available. Therefore, you must in the code insert a check that the requested memory could be allocated.

To allocate dynamic memory you use the keyword new. Look at the following statements:

```
int* pNo;
int iNumber;
```

```
cout << "How many products will be entered? ";
cin >> iNumber;
pNo = new int[iNumber];
```

Here we declare a pointer to int, pNo, and a common int variable iNumber. The user enters an integer which is stored in iNumber. This value is used as the number of items of the array declared in the last statement.

This is however a dangerous way of coding since there might be a lack of memory, and the program will crash. Therefore, we insert a check by means of the following:

```
if ( (pNo = new int[iNumber]) == 0)
{
   cerr << "Not sufficient memory. The program
              will exit!";
   exit(1);
}
//program continues
```

Here we put the declaration of the array as a condition in an if statement. If there is not enough memory, the result of the declaration will be equal to 0. Then the warning text is printed and the program is terminated with exit(1).

If there is enough memory the value of the condition is not 0, and the program continues with subsequent statements.

You can run the code above and enter different numbers to figure out how the code behaves in different situations. Don't forget to include the header file stdlib.h, which is needed for the exit(1) function.

After the if statement we can now continue with product entry to the array:

```
for (int i=0; i<iNumber; i++)
{
   cin >> *pNo ;
   pNo++;
}
```

The loop performs as many turns as the number of items of the array. For each turn a product id is read from the user, which is stored in the memory address given by pNo. pNo is increased by 1, i.e. the pointer moves on to the next item of the array.

If we want to print the products, we must reset the pointer to the original position, i.e. to the first item of the array, and then perform a new loop which prints the products:

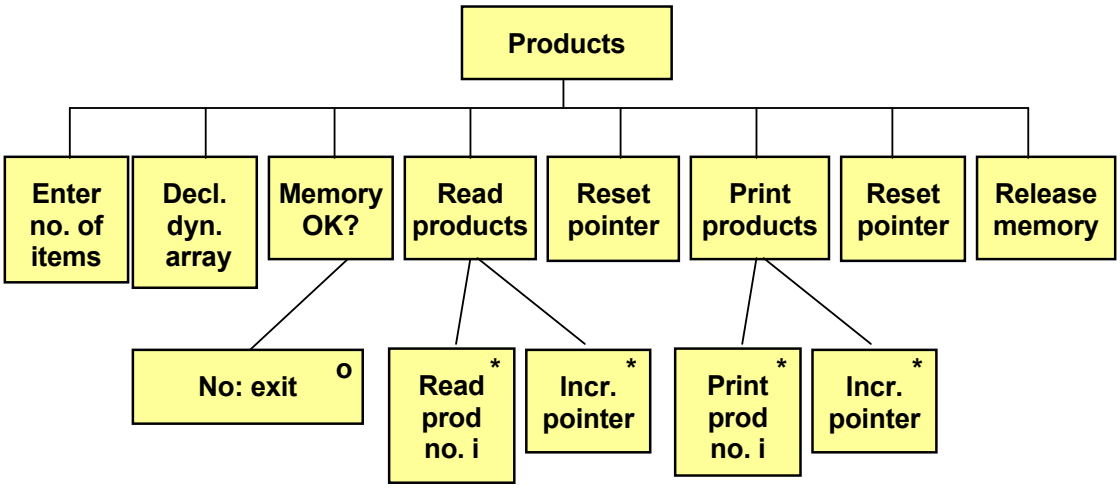
```
pNo = pNo - iNumber;
for (i=0; i<iNumber; i++)
{
   cout << *pNo << endl;
   pNo++;
}
```

When having used dynamic memory it is common by programmers to release the memory when it is not needed any more, thus freeing memory for other tasks of the program. We release the memory for the array with the statements:

```
pNo=pNo-iNumber;
delete[] pNo;
```

First we reset the pointer to its original position and then we use the delete statement to release the dynamic memory.

We now give a JSP graph that shows the process:



Here is the entire program:

```
#include <iostream.h>
#include <stdlib.h>
void main()
{
   int* pNo;
   int iNumber;
   cout << "How many products will be entered? ";
   cin >> iNumber;
   if ( (pNo = new int[iNumber]) == 0)
   {
      cerr << "Not sufficient memory. The program will exit!";
      exit(1);
   }
   for (int i=0; i<iNumber; i++)
   {
      cin >> *pNo ;
      pNo++;
   }
   pNo = pNo - iNumber;
   for (i=0; i<iNumber; i++)
   {
```

```
      cout << *pNo << endl;
      pNo++;
   }
   pNo = pNo - iNumber;
   delete[] pNo;
}
```

We will now create still another program where we instead of product id:s enter a number of names of char type and store the names in arrays in the dynamic memory. You should then remember that each name is itself an array of char type. We will then create a new char array with the new keyword for each name. Each char array will be exactly of the length required by the entered name, with the purpose of saving memory space. Here is the code:

```
#include <iostream.h>
#include <string.h>
void main()
{
   const int iNo = 5;   // Number of strings to be stored
   char temp[30];   // Tempory storage of entered name
   char *cNames[iNo]; // Space for 5 string pointers with
                      // arbitrary number of characters
   cout << "Enter the names of 5 course mates" << endl;
```

```
  for ( int i = 0; i < iNo; i++)
  {
    cout << "Mate no. " << i + 1 << " ";
    cin.getline(temp, 30);  // Temporary storage in temp
    // Don't waiste memory! strlen() gives exactly the
    // number of characters required plus 1 for null:
    cNames[i] = new char[strlen(temp) + 1];
    // Copy the name to the pointer array:
    strcpy(cNames[i], temp);
  }
  // Print the names
  for ( int j = 0; j < iNo; j++)
    //cNames is an array of pointers:
    cout << cNames[j] << endl;
  // Release memory for each separate string array:
  for ( int k = 0; k < iNo; k++)
    delete [] cNames[k];
}
```

8.12 Summary

In this chapter we have learnt the basics of one of the areas that makes C++ unique compared to other programming languages. Pointer management is complicated but offers opportunities to efficient coding at professional level.

We have learnt to declare and assign values to pointers of different data types. One of the great advantages with pointers is pointer arithmetics, where you can step through the items of an array in an efficient way.

We have also learnt to send pointers as parameters to functions and we have learnt some about dynamic memory allocation. C++ programmers must often pay attention to memory allocation at a detailed level not required by other language programmers. This might seem to be unnecessarily complicated from the beginner's point of view, but provides rich opportunities to effective programming aiming at memory minimizing programs with high performance.

We will experience further advantages of pointers in the next chapter about structures.

8.13 Exercises

1. Write a little program which declares an integer variable and initiates it to the value 25. Then declare a pointer to that value. Print the value by means of the pointer.
2. Write a program similar to the previous applying it to a string with your own name instead of an integer.
3. Start from the program in the section 'Price Program with Pointers'. Complete it with a facility to enter a discount percentage to be deducted from the total price. Use a pointer for the discount.
4. Write a program which prompts the user for a driven number of miles and the fuel consumption for the trip. The program should then calculate and print the fuel consumption per mile. Use pointers like in the previous programs.

5. Complete the previous program to also allow entry of the car brand, which should be included in the printout in a suitable way.
6. Write a program that reads 5 integers to an array. The integers should then be printed. Use pointer arithmetics.
7. Complete the program to also calculate the sum of the integers. Use a pointer variable for the sum.
8. Start from the program in the section 'Tax Program' and modify it so that the user enters the salaries and tax percentages.
9. Modify the previous program to instead read the information from a file instead of from the keyboard.
10. Write a program which reads product id:s and prices from a file and stores them in arrays, one array for the product id:s and one for the prices. Use pointers. The program should then print a nice table of products and prices.
11. Modify the program in the previous exercise so that the user can enter a product id and get the corresponding price printed. Use a pointer also for the product id entered by the user.
12. Start from the last version of the program with the find() function in the section 'Functions and Pointers', where the @ character is replaced by a blank. Modify the function so that it prints the updated string from inside of the function. Use the main() program to test the function.
13. Change the previous program so that both the original and the updated email adress is printed.
14. Write a function which replaces lower case characters in a string to upper case. The string should be sent to the function as a pointer. Test the function in a program which prompts the user for his name and then sends the name to the function. The program should print the updated string.
15. Write a function which takes a pointer to an integer array and the number of items of the array as parameters, finds the greatest item and returns it. In the main() program the user should enter 8 numbers to be stored in the array. The function is called and the returned greatest item is printed.
16. Modify the previous exercise so that the user first enters the number of integers being entered. The array should be stored in the dynamic memory.
17. Start from the last program of the section 'Dynamic Memory' where you entered the names of 5 course mates. Since telephon numbers can contain blanks and hyphen (e.g. 0522-23 23 23) they are of char type. Use a second two-dimensional array with pointers to the dynamic memory exactly as for the names.
18. Expand the previous program with checks about enough memory available.