Revision Questions for MT262

Tutor : Rifat Hamoudi
Staff No. : 00567451
Pager No. : 07669-801 509

I have put this tutorial on the web. This tutorial can be viewed and downloaded from http://www.users.totalise.co.uk/~rifat then selecting MT262 Tutorials then Exam Revision.

Revision questions taken from various past exam papers and past MT262 tutorials

PART I example questions

Question 1 (5 marks)

The following design fragment is to change each occurrence of the character 'a' in a given string *Line* (implemented as an *AnsiString*), to the character 'A' and to count the number of changes made. The string *Line* is assumed to be initialised. The data table is as follows.

Type	Identifier	Description
Integer	*Count*	Count of the number of changes made to *Line*
Integer	*Index*	Index to individual characters of *Line*
String	*Line*	The string being manipulated

1 *Count* ← 0
2.1 **loop for** *Index* ← 1 to *Length*(*Line*)
2.2 **if** *Line*[*Index*] = 'a' **then**
2.3 *Line*[*Index*] = 'A'
2.4 *Count* ← *Count* + 1
2.5 **ifend**
2.6 **loopend**
3 write out "Revised string is ", *Line*
4 write out "Number of changes was ", *Count*

Write a C++ code fragment to implement this design fragment. You may assume that the `MT262io` library of functions is available for your use.

(SSR coding)

Question 2 (5 marks)

A TV tuner sold as an additional piece of hardware for a PC can be tuned to one of 60 channels numbered 1 to 60. Part of a program that will enable the user to tune the TV is to prompt the user with a channel number. This channel may be accepted by the user pressing 'A' (for Accept) or rejected by pressing 'N' (for Next channel number) in which case the user is prompted with the next channel number in the sequence. The first channel number presented to the user should be 1 and if a channel has not been accepted by the time channel 60 is displayed then channel 60 should be followed by channel 1. You may assume that all keys except A and N have already been disabled from the keyboard and that pressing either A or N will result in a capital letter being generated. (In other words there is no scope for user data entry errors and so such errors may be ignored in the question.) You have decided on the following top-level design and data table.

1 initialise data
2 **loop while** still going
3 process next outcome
4 **loopend**

Type	Identifier	Description
Character	*Choice*	Key pressed by the user
Integer	*Channel*	Current channel number
Boolean	*StillGoing*	true until user presses the key A.

Refine this design to a stage where it is ready for coding in C++. **(SSR Design)**

Question 3 (6 marks)

A golf club keeps data on its members in the form of records. Each record holds the following information.

- the name of the member
- the age of the member (as a whole number of years)
- the exact handicap of the member (as a real number)

There are 100 such records to store.

(a) Write down a C++ declaration for a record type, `MemberType`, to hold the four pieces of information. Declare a variable `Members` to hold a table of 100 such records.

(b) Write C++ code statements to initialise `Members`[20] to represent a member called Best, aged 52 whose handicap is 8.4.

(c) Assuming that all the records have been initialised write a fragment of code whose purpose is to search the table for the record relating to a member called Sargent and to change the handicap field of this record to 8.3. You may assume that there is a record in the table having Sargent in its name field. **(Datastructure)**

Question 4 (6 marks)

(a) Write definitions - heading and function body - for the function specified below.

QualifiedAverage

Integer *Scores*[20], Integer *Qualification*
Find and return the average of all the numbers in *Scores* which have value less than or equal to *Qualification*. If there are no such values then return -1
Real *QualifiedAverage*

(b) Assuming that *MyScore* is an initialised integer array of 20 elements write down a call to *QualifiedAverage* that will assign to a float variable, *MyAverage*, the average of all values less than or equal to 10 in *MyScore*. **(Functions)**

Question 5 (5 marks)

An incomplete code fragment to write three integer values to a file and then read them back again is as follows.

```
ofstream OutFile;
ifstream InFile;
int Count;

  //Associate OutFile with the external file Data.txt
  for (Count = 1; Count <= 3; Count = Count + 1)
   {
     //Store (Count + 10) in OutFile
     //Write out prompt that begins "Number stored to ... "
    }
  //Close the file
  Count = 0;
  //Associate InFile with the external file Data.txt
  while (// not at the end of the file)
  {
    //Read in the next number from the file
    Count = Count + 1;
    //Write out prompt that begins "Number retrieved from ... "
  }
  //Close the file
  //Write out the prompt that begins "Count of numbers retrieved was "
```

The following output should be obtained when this program is run.

```
Number stored to file was 10
Number stored to file was 11
Number stored to file was 12
Number retrieved from file was 10
Number retrieved from file was 11
Number retrieved from file was 12
Count of numbers retrieved was 3
```

Write the complete code corresponding to the fragment above (including that given above). You will need to replace each comment by a single C++ statement or expression. The standard output file `cout` should be used to generate prompts on the screen.

(Input/Output)

<u>Question 6</u> (5 marks)

In a program concerned with the emulation of a railway system there are 10 stations, numbered 0 to 9 between which a train continuously commutes. Each station has a name which, for simplicity, we will call A, B, ..., J so that station 0 has name A, station 1 name B and so on. The train starts at station 0 and works towards station 9. When it arrives at station 9 it reverses and works back towards station 0 and so on. The state of a train, when it is at a station, is to be modelled using the following class declaration.

```
class TrainType
{
private:
  AnsiString Names[10];
    // names of 10 stations
  int Location;
    //current location of a train
  bool Outward;
    //true if train is going in direction of station 0 to 9
public:
  void MoveOneStation(void);
  /*If the train is on an outward journey increment Location. If this
    results in Location 9 modify Outward. If the train is on an inward
    journey decrement Location. If this results in Location 0 modify
    Outward. */

  AnsiString CurrentStation(void);
  // Returns the name of the station indicated by Location.

  AnsiString NextStation(void);
  // Returns the name of the next station at which the train will stop.

  // There are other methods that do not concern us
};
```

Write the code for this class, giving definitions for the three listed methods.

(Object & Classes)

<u>Question 7</u> (10 marks)

This question is to do with preparation for writing a proper Database Application using Object Oriented C++ :

(a) Write a function to convert a character to uppercase

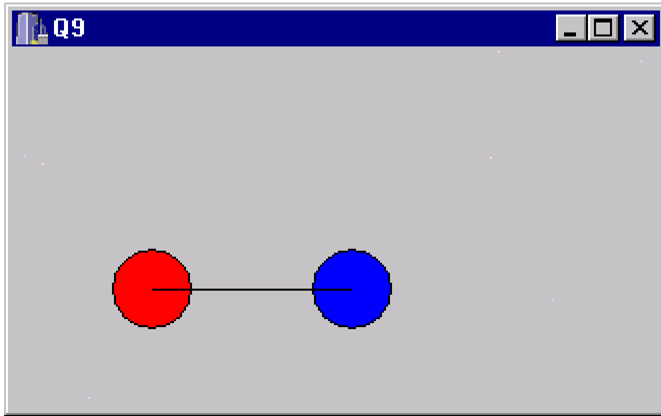
(b) The company want to create a database to keep track of its employees name, address and salary. Write a class with appropriate method definitions any other necessary variables to capture what the company wants. Your methods should be one for initialisation, one for adding an employee and one for displaying the employee details

(c) Sketch design for files needed to write the database application and what should go in each file

(Graphics Form)

<u>Question 8</u> (5 marks)

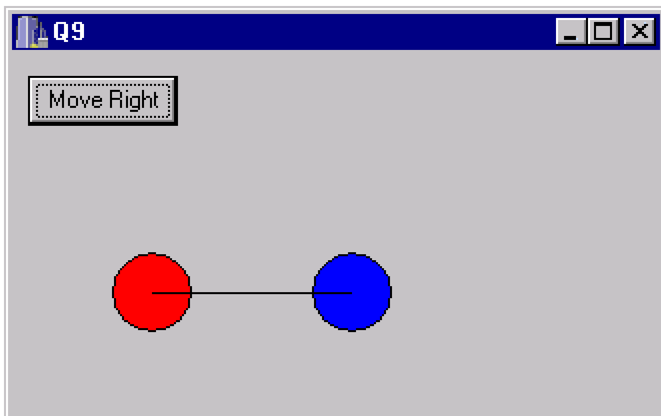
This question is concerned with generating the display shown below, consisting of two circles of the same size with their centres joined by a horizontal line. The circle to the left should be coloured red, that on the right should be coloured blue and the line should be black.



The code to draw the picture should only appear in the `OnPaint` event handler.

(a) Write the code for the `OnPaint` event handler that will produce the diagram above in such a way that the red circle has diameter 40 and the top left of its bounding rectangle is at pixel position (50,100). The centres of the two circles should be 100 pixels apart.

The application is now to be modified so that the figure can be made to track across the screen to the right one pixel at a time. Users will make it track by pressing the button called Move Right in the figure below.

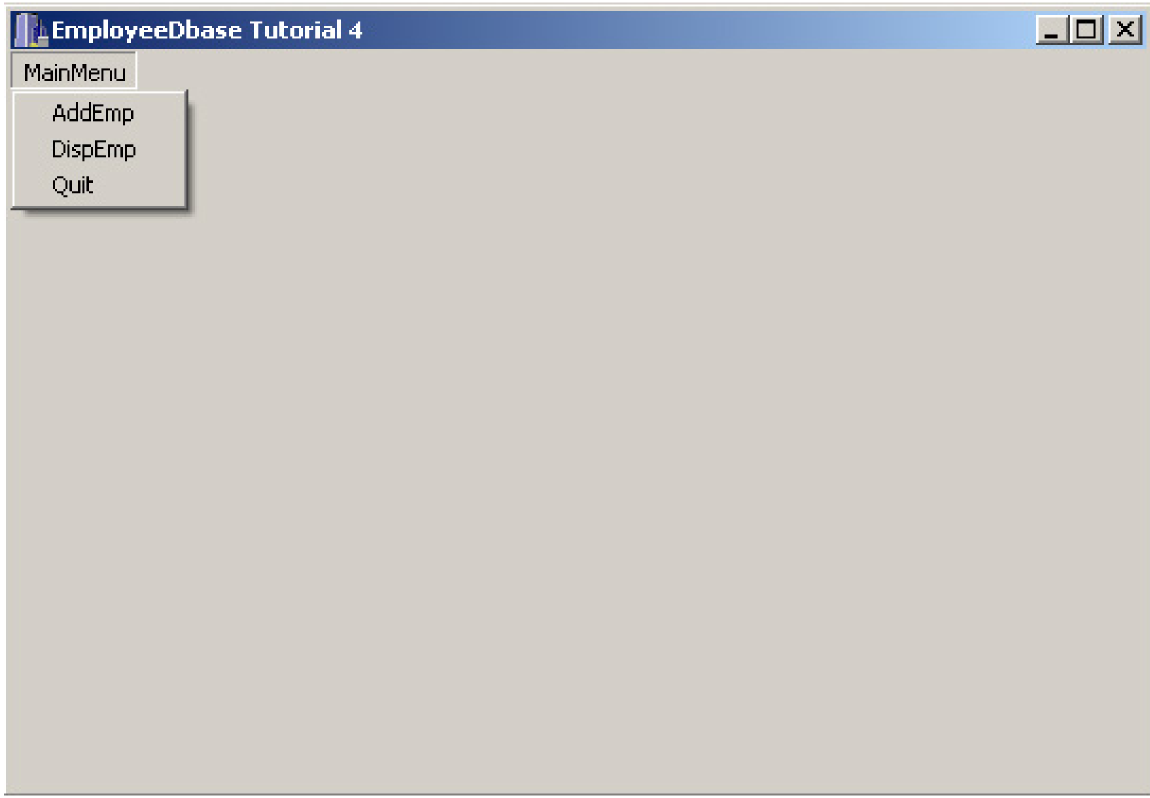


(b) Write down the revised code for the `OnPaint` handler and the code for the button `OnClick` event handler. You may assume that a Form variable called `X` has been declared and has been initialised to 0. You may reference or reset the value of this variable in your event handlers.

(Graphic Code)

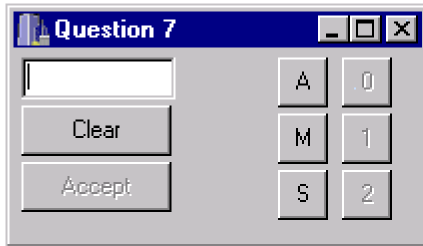
Question 9 (5 marks)

Write a menu driven software for the employee database and implement an event handler for quitting the software. Save the software as EmpDB and don't worry about implementing event handlers for adding and displaying employees for this tutorial. The graphical output should look similar to the figure below :



(Event Handlers)

Question 10 (7 marks)



As part of a project a form is to be created for input of data. The user is to enter a code which must begin with one of the letters A, M or S and thereafter must contain at least one of the digits 0, 1 or 2. So M102, S1 are examples of legal entries whereas MS284, M and M23S are examples of illegal inputs.

The proposed interface is shown above in its initial state. To ensure that the user does not enter illegally formatted codes the interface must be implemented so that only the buttons may be used to enter data and so that buttons are only active at appropriate points in the input cycle.

For example, for a user to enter the code M102, they would first press the M button at which point the edit box should display M. The letter buttons should now become inactive and the digit buttons become active. Pressing the digit button 1 results in the edit box showing M1. Further digits may be entered so that pressing the digit 0 would result in the edit box showing M10 and finally pressing 2 would give M102.

The *Clear* button is to enable the user to abort the input. This should have the effect of returning the form to its initial state. This is the only way that the input may be edited.

The *Accept* button is used when the entered code is as the user would wish.

The methods *DigitsOffLettersOn*(), *DigitsOnLettersOff*() may be assumed to be available. The former disables all the digits buttons and enables the letter buttons. The latter does the converse.

(a) Give a design-time properties for the edit box (which is implemented as a *TEdit* component called *Edit1*) which is different from its default behaviour.

(b) Assuming that the names of the buttons with letter captions are the same as their captions, and that the digit buttons have names *one*, *two* and *three* respectively, write the code for the body of the method *DigitsOnLettersOff*()

(c) Write the event handler code for the button labelled A.

(d) Write the code for the event handler for the *Clear* button.

(Graphics)

PART II   example questions

You attempt **TWO** *the questions in this part and are advised to spend about* **85 minutes** *on it.*
The marks for each question are given beside the question.

Question 11

A route planner, part of which is shown below, shows the distances in miles between 25 towns. In the figure the first 4 towns of the planner are shown. By looking along the row containing Cardiff to the column containing Birmingham it can be seen that it is 106 miles between Cardiff and Birmingham. Note that the planner has no entries in its 'upper triangle'. So you cannot find the distance between Birmingham and Dover by looking along the row containing Birmingham. Instead you must look along the row containing Dover to the column containing Birmingham.

	Aberdeen	Birmingham	Cardiff	Dover
Aberdeen	0			
Birmingham	473	0		
Cardiff	524	106	0	
Dover	619	202	233	0

A C++ class will be used to model the planner and its behaviour using the declarations below. The planner will store exactly 25 names.

```
class PlannerType
{//We assume parameters are always valid names
    AnsiString Names[25]
    int Table[25][25];
  public:
    void Initialize();
    int Distance(AnsiString From, AnsiString To);
    int Journey(AnsiString From, AnsiString To, AnsiString ViaName);
  protected:
    int FindIndex(AnsiString AName);
};
```

The array *Names* stores the 25 place names in alphabetical order and the array *Table* stores the distance between them as shown in the table. So, for example, the entry *Table*[2][0] will store the value 524 and will represent the distance between Cardiff, which is stored at index 2 in *Names* and Aberdeen, which is stored at index 0 in *Names*. Elements of *Table* like *Table*[0][2], whose second index is larger than the first, correspond to the 'upper triangular' entries in the planner and are therefore unused. The protected method *FindIndex* returns the index at which a given place name is stored.

(a) The method *Initialize* has to assign appropriate contents to the array *Table*. Elements of *Table* that are unused, as described above, should be set to -1. Write down a sequence of C++ statements which will make the necessary assignments for the data corresponding to the Aberdeen and Birmingham rows (indexed 0 and 1 respectively) of the table. [*6 marks*]

(b) Write the implementation of the method *FindIndex* as it would appear in the **.cpp** file. You should assume that *AName* is a name that exists in *Names*.

(c) Write the implementation of the method *Distance* whose purpose is to return the distance between the towns whose names are given in the two parameters. You may assume that the two parameters are valid names. [*3 marks*]

(d) Write the implementation of the method *Journey* whose purpose is to return the distance between two towns when the journey is carried out via an intermediate town. So, for example, for the journey from Aberdeen to Birmingham via Cardiff the function would return the value 630, this being the sum of the distances from Aberdeen to Cardiff and from Cardiff to Birmingham. You may assume that the three names are valid names. [*4 marks*]

(e) Give a reason why you think *FindIndex* is declared **protected** and what the consequences of this are to users of the class.

Question 12

This question concerns a prototype for a system that will enable aircraft seat bookings to be made over the internet. Part of the system requires the user to choose the departure and destination airports, an interface for which is shown below. In its initial state, the departure and destination airports are both set to London. When the departure and destination airports differ then the route is shown on the map by means of a red line.

The state illustrated below shows that the departure airport is London, that the destination is Inverness.

This journey is shown by a line between these two places on the map. The example also shows that the user is about to select the new destination of Glasgow.

The user selects an airport by means of the **ComboBoxes**. whose names on the Builder's Form are `Departure` and `Destination` respectively.

For experienced users an alternative method of departure and destination airport selection is to be provided. A left mouse click on the map in the vicinity of a place having an airport will select the departure airport. A right click will select the destination.

The map is provided by a bitmap file measuring 200 pixels across and 365 pixels down which is positioned in the top left of the screen. (All co-ordinates mentioned in the question will be relative to (0, 0), this being at the top left as described in the course units.) In addition a class `AirPortType` is provided whose task is to keep information about airports. The part of its declaration which is public is:

```
class AirportType
{
  public:
    int GetX(AnsiString APlace);
    int GetY(AnsiString APlace);
    void Init();
    AnsiString IsNear(int AnX, int AY);
 };
```

These methods perform the following tasks:

`GetX` Return the X co-ordinate of the airport whose name is `APlace`. It should be assumed that `APlace` is a valid name of an airport represented by the class.

`GetY` Return the Y co-ordinate of the airport whose name is `APlace`. It should be assumed that `APlace` is a valid name of an airport represented by the class.

`Init` Initialize an instance. You need not know any details of this except that airports at London, Exeter, Glasgow and Inverness are all represented.

`IsNear` Answer with an `AnsiString` that is the name of an airport whose co-ordinate are 'near' to those of X, Y. If there is no 'near' airport then return the empty string. You do not need to know how 'near' is measured.

(a) The `Form` used to build the application contains a `TImage` component called `Image1` which contains the map shown above. Explain why the `Visible` property of this component is set to false at design time.

(b) The Form's `OnCreate` handler includes the following code.

```
Departure->ItemIndex = Departure->Items->IndexOf("London");
```

Explain what the left hand side refers to and explain how the value which is assigned to it arises from the expression on the right hand side.

(c) The `OnChange` handler for each of the ComboBoxes, `Departure` and `Destination` has to cause a line to be drawn on the map. So each of these handlers will cause the form to be repainted. Write the code for the `OnPaint` handler, remembering that both the image and the line have to be painted. (If the destination and departure points are the same then the line is simply drawn from one point to itself!)

(d) The following incomplete code corresponds to the handler which deals with those users who prefer to use a mouse to select the departure and destination airports. The first **if** statement covers the possibility that the user clicks close enough to an airport. The second one distinguishes between a left mouse click and a right mouse click. You should submit the complete code for the handler (including that given in the question).

```
void fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
      TShiftState Shift, int X, int Y)
{
  if ()
  {
    if (Button == mbLeft)
      //statement(s) to handle a left mouse button click
    else
      //statement(s) to handle a right mouse button click
  };
//statements
}
```

Question 13

This question is concerned with an electronically controlled set of bathroom scales. The figure below shows the state of the scales when not in use - this mode will be referred to as 'sleep' mode. The scales can record data on up to four people but it is up to those people to member which of the buttons corresponds to them.

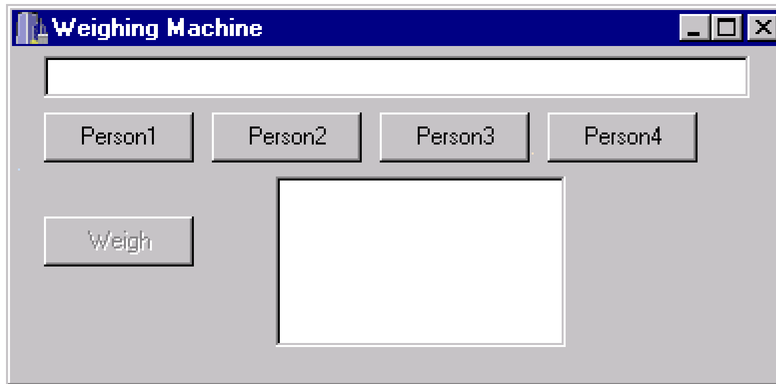


Figure 1

To activate the scales a user presses the button that they have agreed with the other users is theirs. Figure 2 below shows the state of the machine after the Person1 button has been pressed.

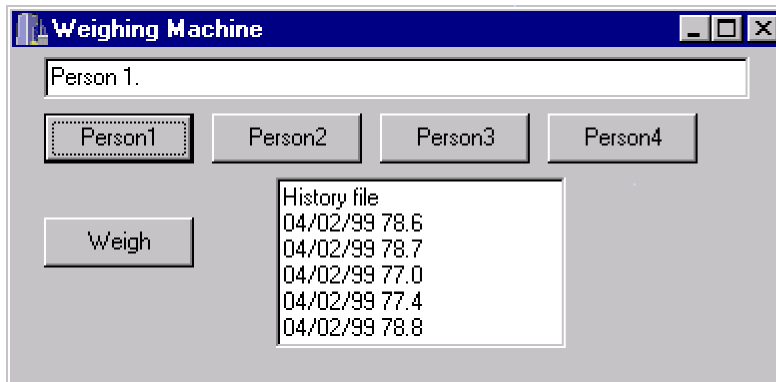


Figure 2

The last 5 weights of that person are shown as a history file together with the date on which each weight was taken. The button labelled Weigh has also become activated.

On pressing the Weigh button, the weight of the person on the scales is taken and displayed. This weight is also recorded in the machine's memory and the historical data is updated. The figure below shows the outcome when Person1 pressed the Weigh button with the machine in the state shown in Figure 2.

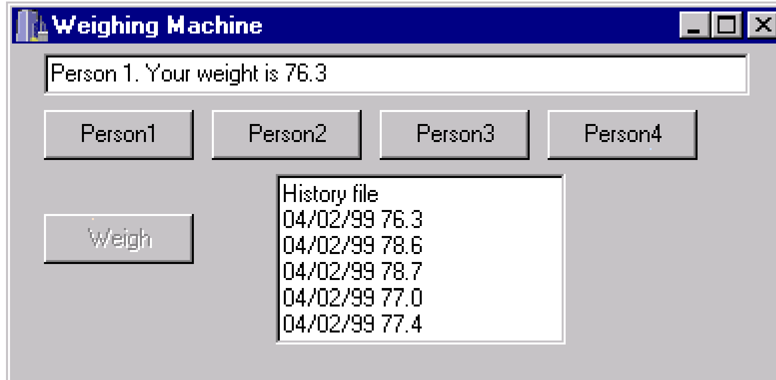


Figure 3

Note that this figure shows that the Weigh button has become disabled and it can only be reactivated by a user pressing one of the buttons `Person1`, `Person2`, `Person3` or `Person4`.

In the event that a user does not press any button for a period of 10 seconds the machine will go into sleep mode as shown in Figure 1.

The engine for this application will make use of a class *ScalesType* whose declaration is as follows.

```
class ScalesType
{
    struct WeightRecord
      {
       float Weight;
       AnsiString Date;
      };
    WeightRecord History[5];

  public:
     void Init(void);
     AnsiString Weigh(void);
     AnsiString HistoryOf(void);
};
```

The methods are specified as follows:

Init Initialise all four elements of *History* to records whose *Weight* field is 0 and *Date* field is the null string.

HistoryOf Return a string consisting of pairs of date, weight data each separated by a comma. So for example for the data in Figure 3 this method would produce the string "04/02/99 76.3, 04/02/99 78.6, 04/02/99 78.7, 04/02/99 77.0, 04/02/99 77.4,".

Weigh This method first simulates weighing the person and this is achieved by a random number generator. The randomly generated weight and the time it was generated are then assigned to History[0] with all other values being pushed down the array (so that History[5] gets overwritten). The randomly generated weight is returned as a string.

(a) Write down an implementation of the method *Init*.

(b) Write an implementation of the method *HistoryOf*.

(c) Part of the body of the implementation of *Weigh* is shown below. Complete the missing details

```
float Weight;
AnsiString WeightStr, DateStr;
```

```
    int i;

    Weight = 76.0 + (float)random(300)/100;
    WeightStr = FormatFloat("0.0", Weight); //see comment below
    DateStr = DateToStr(Date());
    //Missing details go here
  }
```

Comment: The function `FormatFloat` converts a real number to a string using the pattern given by the first parameter. So here WeightStr will be a string representation of a real number correct to 1 decimal place.

The application *includes* the following Form declarations

```
__published:      // IDE-managed Components
    TEdit *Edit1;      //used for the current weight display
    TButton *Person1;
    TButton *Person2;
    TButton *Person3;
    TButton *Person4;
    TMemo *Memo1;      //used for the history data
    TButton *Weigh;
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall Person1Click(TObject *Sender);
    void __fastcall Person2Click(TObject *Sender);
    void __fastcall Person3Click(TObject *Sender);
    void __fastcall Person4Click(TObject *Sender);
    void __fastcall WeighClick(TObject *Sender);
public:      // User declarations
  ScalesType MyScales[4];
  int CurrentUser;
  void WriteHistory();
  void ReInitTimer();
```

The button `Person1` will deal with a `CurrentUser` numbered 0 whose associated `ScalesType` object is `MyScales[0]`. The button `Person2` will deal with a `CurrentUser` numbered 1 whose associated `ScalesType` object is `MyScales[1]` and so on. The method `WriteHistory` produces the text in the `Memo1` field and the method `ReInitTimer` causes the state of the machine to revert to 'sleep' mode after 10 seconds.

(d) Write the body of the code for the Form's `OnCreate` event.

(e) Write the event handler `Person1Click`.

(f) Write the event handler `WeighClick`.

Answers

Answer to question 1

Note that it is assumed that *Line* is initialised but despite this we include the code fragment that contains its declaration.

```
int Count, Index;
AnsiString Line;

Count = 0;
for (Index = 1; Index <= Length(Line); Index = Index + 1)
  if (Line[Index] == 'a')
  {
    Line[Index] = 'A';
    Count = Count + 1;
  }
WriteStringPrCr("Revised string is ", Line);
WriteIntPr("Number of changes was ", Count);
```

Comment

The question tests the use of == for equality and the fact that string indexing starts at 1. There are many different ways to code the output using a variety of the MT262io routines.

Answer to question 2

1.1 *Channel* <– 1
1.2 *StillGoing* <– true
2.1 **loop while** *StillGoing*
3.1 write out "Channel = ", *Channel*
3.2 read in *Choice*
3.3 **if** *Choice* = 'N' **then**
3.4 *Channel* <– (*Channel* mod 60) + 1
3.5 **else**
3.6 *StillGoing* <– false
3.7 **ifend**
4 **loopend**

Comment

Any alternative to our use of *mod* is acceptable. The test at 3.3 can be expressed in a number of equivalent ways. Step numbering must be consistent with that given in the question.

Answer to question 3

(a) The declarations are as follows.

```
struct MemberType
{
  AnsiString Name;
  int Age;
  float Handicap;
};
MemberType Members[100];
```

Comment

Note the CourseTeam style is separate declarations, so a semicolon after the **struct** declaration is vital.

(b) The code is

```
Members[20].Name = "Best";
Members[20].Age = 52;
Members[20].Handicap = 8.4;
```

(c) Since we are told the search will succeed a simple while loop will do. The declaration of a looop control variable is expected.

```
int Index;
while (Members[Index].Name != "Sargent")
   Index = Index + 1;
Members[Index].Handicap = 8.3;
```

Answer to question 4

(a) The function code is as follows.

```
float QualifiedAverage(int Scores[20], int Qualification)
{
  int Count, Index, Total;

  Total = 0;
  Count = 0;
  for (Index = 0; Index <= 19; Index = Index + 1)
  {
    if (Scores[Index] <= Qualification)
    {
      Total = Total + Scores[Index];
      Count = Count + 1;
    }
  }
  if (Count == 0)
    return -1;
  else
    return (float) Total/Count;
  };
```

Comment

The division that calculates the average must be done on floats hence Total has to be cast into a float. Indexes must run from 0 to 19.

(b) The required statement is

```
MyAverage = QalifiedAverage(MyScore, 10);
```

Answer to question 5

```
OutFile.open("Data.txt");
for (Count = 1; Count <= 3; Count = Count + 1)
{
  OutFile << (Count + 10) << "\n";
  cout << "Number stored to file was " << (Count + 10) << endl;
 }
OutFile.close();
Count = 0;
InFile.open("Data.txt");
while (! InFile.eof())
{
  InFile >> ANumber >> ws;
  Count = Count + 1;
  cout << "Number retrieved from file was " << ANumber << endl;
}
InFile.close();
cout << "Count of numbers retrieved was " << Count;
```

Comment

The loop illustrates the two different ways in which a new line may be inserted into the streams. Either can be used.

Answer to question 6

```
void TrainType::MoveOneStation(void)
{
  if (Outward)
  {
    Location = Location + 1;
    if (Location == 9)
      Outward = false;
  }
  else
  {
    Location = Location - 1;
    if (Location == 0)
      Outward = true;
  }
};

AnsiString TrainType::CurrentStation(void)
{
  return Names[Location];
};

AnsiString TrainType::NextStation(void)
{
  if (Outward)
    return Names[Location + 1];
  else
    return Names[Location - 1];
};
```

Answer to question 7

(a)

```
char Tut3_uppercase(char selection)
{
    if ((selection >= 'a') && (selection <= 'z'))
        selection = selection - 32;

    return selection;
}
```

(b)

```
#define MaxEmp 2
class CompanyType
{
  private:

  struct EmployeeType
  {
    int Salary;
    AnsiString  Address;
    AnsiString  FullName;
  };
  EmployeeType  Employee[MaxEmp];

  public:

  void Init(void);
  AnsiString AddEmp(AnsiString EmpName, AnsiString EmpAddress, int
EmpSalary);
  void DispEmp(void);
};
```

(c)

The files can be divided into 5 files as follows :

Tut3_3.cpp : Main file which contains the main driving code.

Tut3_3_methodimp.cpp : File containing the implementation of the methods in the Company class

Tut3_3_methodimp.h : contains the class definition used by the database

Tut3_3_funcimp.cpp : contains the implementation of functions used by the methods in the Company class and the main driver

Tut3_3_funcimp.h : contains function prototypes (definition)

Answer to Question 8

(a) The code is

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas ->Brush->Color= clRed;
  Canvas -> Ellipse(50,100,90,140);
  Canvas ->Brush->Color= clBlue;
  Canvas -> Ellipse(150,100,190,140);
  Canvas ->Brush->Color= clBlack;
  Canvas -> MoveTo(70,120);
  Canvas->LineTo(170, 120);
}
```

(b) The code above has to be modified so that the x coordinates can be adjusted (using the Form variable *X*) thus enabling the shapes to track right.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas ->Brush->Color= clRed;
  Canvas -> Ellipse(50 + X,100,90 + X,140);
  Canvas ->Brush->Color= clBlue;
  Canvas -> Ellipse(150 + X,100,190 + X,140);
  Canvas ->Brush->Color= clBlack;
  Canvas -> MoveTo(70 + X,120);
  Canvas->LineTo(170 + X, 120);
}
```

The *OnClick* event handler then has to increment *X* each time the button is pressed and request a repaint. We have used the default name for the button (and hence the default name for the handler).

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  X = X + 1;
  Repaint();
}
```

Answer to question 9

To solve this do the following with C++ Builder :

1) Click on **File** then **New Application**
2) Save the project by clicking on **File** then **Save Project As**, call the .cpp file as EmpDBU and the project as EmpDB. **DO NOT** use the same name for both as this will crash the program and causes problems.
3) On the Form drag the menu button
4) Click on Form and change the **Caption** to EmployeeDatabase
5) Double click on the menu button on the Form and change the **Caption** to MainMenu
6) Press Enter and then change the Caption to **AddEmp** do the same for the rest
7) Click on Quit then on Events and double click on the right of **OnClick** this will take you to the event handler for what the program should do when the user presses the Quit button. Type Application->Terminate();
8) Save the program and run. This should give you your graphical database menu application.

Code for EmpDB software

EmpDB.cpp (generated by Builder)

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
USERES("EmpDB.res");
USEFORM("EmpDBU.cpp", MainForm);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TMainForm), &MainForm);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//-----------------------------------------------------------------------------
```

EmpDBU.cpp

```
//-----------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "EmpDBU.h"
//--------------------------------------------------------------------------------
-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
```

```
//---------------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::QuitMClick(TObject *Sender)
{
    Application->Terminate();
}
//---------------------------------------------------------------------------
```

EmpDBU.h

```
//---------------------------------------------------------------------------
#ifndef EmpDBUH
#define EmpDBUH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Menus.hpp>
//---------------------------------------------------------------------------
class TMainForm : public TForm
{
__published:      // IDE-managed Components
    TMainMenu *MainMenu1;
    TMenuItem *MainMenu2;
    TMenuItem *AddEmpM;
    TMenuItem *DispEmpM;
    TMenuItem *QuitM;
    void __fastcall QuitMClick(TObject *Sender);
private:    // User declarations
public:           // User declarations
    __fastcall TMainForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------------------------------------
#endif
```

Answer to question 10

(a) It must have its *ReadOnly* property set to true because users are not allowed to modify the text in the box. It must also have its initial *Text* property set empty and it should have the focus (which can be set by defining its *TabOrder* to be 0).

(b) The code is

```
void TForm1::DigitsOnLettersOff()
{
  One -> Enabled = true;
  Two -> Enabled = true;
  Zero -> Enabled = true;
  A -> Enabled = false;
  M -> Enabled = false;
  S -> Enabled = false;
}
```

(c) The code is

```
void __fastcall TForm1::AClick(TObject *Sender)
{
  Edit1 -> Text = Edit1->Text + "A";
  DigitsOnLettersOff();
}
```

(d) The code is

```
void __fastcall TForm1::ClearClick(TObject *Sender)
{
  Edit1 -> Clear();
  DigitsOffLettersOn();
  Accept -> Enabled = false;
}
```

Comment

The key property for the *Edit* box is that it should be *ReadOnly*.

Answer to question 11

(a) Unused elements have to be initialized to -1 and so every element in each row must be assigned a value.

```
for (I = 0; I < 25; I = I + 1)
  {
    Table[0][I] = -1;
    Table[0][0] = 0;
  };
for (I = 0; I < 25; I = I + 1)
  {
    Table[1][I] = -1;
    Table[1][0] = 473;
    Table[1][1] = 0;
  };
```

(b) The code is

```
int PlannerType::FindIndex(AnsiString AName)
{
  int Index;

  Index = 0;
  while (Names[Index] < AName)
    Index = Index + 1;
  return Index;
};
```

(c) The specification does not insist that the parameters are supplied in alpha order and so the code has to allow for them not being so ordered.

```
float PlannerType::Distance(AnsiString From, AnsiString To)
{
  if (From > To )
    return Table[FindIndex(From)][FindIndex(To)];
  else
    return Table[FindIndex(To)][FindIndex(From)];
};
```

(d) The implementation must use the existing method. Failure to do so would be penalized heavily.

```
float PlannerType::Journey(AnsiString From, AnsiString To, AnsiString
Via)
{
  return Distance(From, Via) + Distance(Via, To);
};
```

(e) The method *FindIndex* is required in order to simplify the impelmentation of other methods. It is not intended for end users, who should not be aware of how the table is represented (and who therefore do not need to know about indexes at all). To stop end users sending this message, *FindIndex* needs to be declared **private** or **protected**. Here it is declared **protected** which means developers who subclass *PlannerType* may call *FindIndex*. Had it been declared **private** then not even subclass developers would have had access to *FindIndex* and this would have severeley limited their ability to extend the functionality of the subclass; they would not be able to find the index at which a name is stored. (Neither would they be able directly to access *Names* because it is declared **private**.)

Comment

This is somewhat longer than an expected solution but some discussion of the issues realting to subclassing and 'end using' would be expected.

Answer to question 12

(a) The image has to be drawn on the canvas because a purpose of the programme is to draw lines on the map. This can only be achieved if the image is part of the canvas. So the image itself must be not visible but must be transferred, pixel by pixel, to the canvas. If we left the image itself as visible then we would get two copies of the bit-map, one of the image and one of the copy on the Canvas.

(b) The left-hand side *Departure->ItemIndex* specifies the index of the item that will be shown in the ComboBox window. By default this should read "London". So the right-hand side calculates the index of London in the list of stored strings in *Departure*. *Departure->Items* returns the strings (in fact *TStrings*) and *IndexOf("London")* is a message which is sent to the latter and which asks for the index of the occurrence of London to be returned.

(c) The code is

```
void __fastcall TForm1::DepartureChange(TObject *Sender)
{
  Repaint();
}
//---------------------------------------------------------------------

void __fastcall TForm1::DestinationChange(TObject *Sender)
{
  Repaint();
}
//---------------------------------------------------------------------

void __fastcall TForm1::FormPaint(TObject *Sender)
{
//Pixel by pixel copying is OK as an exam answer. However it runs
//very slowly.
//  int i, j ;
//  for (i=0; i < 202; i = i+ 1)
//    for (j =0; j <365; j = j + 1)
//      Canvas->Pixels[i][j] = Image1->Canvas->Pixels[i][j];
//Use of CopyRect solves this and is a better solution.

  Canvas->CopyRect(ClientRect,Image1->Canvas, ClientRect);
  Canvas->Pen->Color = clRed;
  Canvas->MoveTo(Airports.GetX(Departure ->Text),
      Airports.GetY(Departure ->Text));
  Canvas->LineTo(Airports.GetX(Destination ->Text),
      Airports.GetY(Destination ->Text));
}
//--------------------------------------------------------------------
```

Comment

The major point here is that all the *ComboBox OnChange* handlers have to do is to call *Repaint* which does all the work (via the handler *OnPaint*). The *OnPaint* handler must first copy the image to the *Form*'s *Canvas*. The solution shows two ways in which this can be done. The pixel by pixel method is slow but would get full marks if accurately done. Next the pen has to be set to the correct colour. (We should really have reset this colour to the default at the end of this method but nothing else uses a `Pen` and so this answer would get full marks.) The required line may be drawn either from the departure airport to the destination or the other way around. It does not matter. Here we have chosen to move to the departure airport and then draw the line to the destination. To move to the co-ordinates of the

departure airport we need to find the co-ordinates from its name. Its name is given by *Departure->Text*, namely what is in the text part of the *Departure ComboBox. Airports* keeps track of co-ordinates and so we send it *GetX* and *GetY* messages, with parameter which is the name of the airport. The expression *Canvas->MoveTo* uses these points as parameters.

It is expected that you remember that the canvas of a form is called *Canvas* and so this name was not mentioned in the question. We would also expect you to have some idea of what a handler header looks like. In particular we would expect you to know that the name given to it by Builder is derived from the components name and that all handlers have at least one parameter, namely the Sender of the message. We would not expect you to remember the precise syntax.

(d) The required code is:

```
void __fastcall TForm1::FormMouseDown(TObject *Sender,
    TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if (Airports.IsNear(X,Y) != "")
    {if (Button == mbLeft)
      Departure->ItemIndex =
         Departure->Items->IndexOf(Airports.IsNear(X,Y));
    else
      Destination->ItemIndex =
         Destination->Items->IndexOf(Airports.IsNear(X,Y));}
  Repaint();
}
```

Comment

IsNear is sent to *Airports* and it returns either the empty string or the name of a place with an airport. In the case of the former we do nothing. In the case of the latter we first determine whether a right click or a left click was responsible for the event. For a left button click we use the code exploited in part (b) but the parameter here is *Airports.IsNear(X, Y)*. So the text displayed by *Departure* has to be updated to the name returned by *IsNear*. The only way to do this is to find the index at which it is stored in the *Items* list of *Departure*. The analogous situation holds for a right click. Whatever the outcome, a *Repaint* message is sent to update the display.

Answer to question 13

(a) The solution is:

```
#include "Vegetable.h"

class MachineType
{
    Vegetable Items[10];
  public:
     AnsiString Weigh(AnsiString AVegName);
     AnsiString VegStr();
     float PriceOf(AnsiString AVegName);
     void ChangeItem(AnsiString OldName, AnsiString NewName, float
UnitCost);
     void Init();
     void ChangePrice(AnsiString AVegName, float NewPrice);
  private:
     int IndexOf(AnsiString AVegName);
};
```

Comment

The file needs an *include* statement because it references the class vegetable whose definition is in **Vegetable.h**. The final semi-colon is vital.

(b) The solution is:

```
int MachineType::IndexOf(AnsiString AVegName)
{int i;

  i = 0;
  for (i=0; i < 10; i = i + 1)
    if (AVegName == Items[i].GetName())
      return i;
  return i;
};
```

Comment

The specification tells us that it should be assumed that *AVegName* is a name in one of the objects in *Items* and so a simple search is bound to succeed. (Parts (e) and (f) of the question investigate this further.) Answers that included redundant testing would be penalised. In this code *Items*[*i*] represents a *Vegetable* object and to retrieve its name we send it a *GetName* message.

There are many alternative solutions to this code. On finding a match, the first **return** statement causes an immediate exit from method (thereby exiting the **for** loop). The second **return** statement is not strictly required because *we* know (but the compiler does not) that a match is guaranteed within the for loop. Failure to include the second **return** would cause a compiler warning but it would compile and run. Indeed it returns 10 in the event of failing to find a match.

(c) The solution is:

```
float MachineType::PriceOf(AnsiString AVegName)
{
  return (Items[IndexOf(AVegName)].GetPrice());
}
```

Comment

This method needs to find the index of Items which holds the object with name *AVegName*. The expression *Items*[*IndexOf*(*AVegName*)] does this. Then we send this object the *GetPrice* message which returns the unit cost. Again the specification tells us not to worry about *AVegName* not being a name in the *Items* objects.

(d) The solution is:

```
void MachineType::ChangeItem(AnsiString OldName, AnsiString NewName,
float UnitCost)
{ Vegetable NewVeg;

    NewVeg.Init(NewName, UnitCost);
    Items[IndexOf(OldName)]= NewVeg;
}
```

Comment

A local variable is needed which will be a new instance of *Vegetable*. This instance is initialised with the parameters *NewName* and *UnitCost*. This object then replaces the element of *Items* which contains an object having name *OldName*.

(e) The interface can obtain all the names by sending the engine the message *VegStr*. This returns a string of all the names (separated by commas). This string can then be used to assign Captions to the buttons and items to the ComboBox.

Comment

A solution along these lines would get full marks. The essential information that your answer needs to convey is that the interface gets its information from the engine via the method VegStr. This method passes back the names of the 10 vegetables it holds. Had we been developing a real application (rather than an exam question) then we would have used a const definition rather than the number 10. The designer of the interface would then have to ensure that there are the same number of buttons as there are objects in *Items*.

(f) The person updating the weighing machine selects an item from the drop down-list for updating. The resulting choice (a string) is bound to exist in *Items* because that is how the drop down-list was created (see part (e)).

Comment

The key point is that users do not type in the name of an existing vegetable - they select from a list. So the selected name is bound to exist - users are not given the opportunity to make input typing errors.

(g) The way in which the *Vegetable* objects are stored is of no concern to users of *MachineType* and they should not be given access to it.

Comment

This explanation would get full marks but we expand upon it here. As implementors of *MachineType* we have frequent need to find the index at which an object is stored and so it makes sense to provide ourselves with a function, *IndexOf*, which does this. That function needs to be potentially available to all the methods we implement and so we declare it as part of the class. Declaring the method to be private means we, as implementors of MachineType can use it when implementing other methods of the class, but users do not have access to it.

To extend this further, users should be blissfully unaware how the individual *Vegetable* objects are being stored and should certainly not be given access to the chosen way. We, as implementors of *MachineType*, used an array to store the 10 *Vegetable* objects. But we could have used a different method such as a linked list (as used in the implementation of a stack in Block IV Unit 2) in which case *IndexOf* would be inappropriate and would not be needed. (We may well find need of an analogous function but that would depend upon the precise structure we chose instead of an array.)