Declare the corresponding "partially implemented abstract classes".

- BTDoc.h
  Declares class BTDoc, #including the Document.h class (to get a definition of the base class Document), and BTree.h to get details of the storage structure used.

- Record.h and RecordWin.h
  Declare the `Record` and `RecordWin` classes. Class `RecordWin` is separate from the other Window classes. The main group of Window classes could be used in other programs; `RecordWin` is special for the type of programs considered in this chapter.

- My.h
  Declares the program specific "MyApp", "MyDocument", and "MyRec" classes.

- The ".cp" implementation files
  The code with definitions for the classes, additional #includes as needed.

The main program, in file main.cp, will simply declare an instance of the "MyApp" class and tell it to "run".

30.8 THE "STUDENTMARKS" EXAMPLE PROGRAM

We might as well start with the main program. It has the standard form:

```
int main()
{
    StudentMarkApp a;
    a.Run();
    return 0;
}
```

The three classes that have to be defined are `StudentMarkApp`, `StudentMarkDoc`, and `StudentRec`. The "application" and "document" classes are both simple:

Application and Document classes

```
class StudentMarkApp : public Application {
protected:
    virtual Document* DoMakeDocument();
};

class StudentMarkDoc : public BTDoc {
protected:
    virtual Record *DoMakeRecord(long recnum);
    virtual Record *MakeEmptyRecord();
};
```

The implementation for their member functions is as expected:

Implementation of Application and Document classes

```
Document *StudentMarkApp ::DoMakeDocument()
{
    return new StudentMarkDoc;
}

Record *StudentMarkDoc::DoMakeRecord(long recnum)
{
    return new StudentRec (recnum);
}

Record *StudentMarkDoc::MakeEmptyRecord()
{
    return new StudentRec (0);
}
```

Record class

The specialized subclass of class `Record` does have some substance, but it is all quite simple:

Redefine KeyedStorableItem functions

Redefine Record functions

Declare unique data members

```
class StudentRec : public Record {
public:
    StudentRec (long recnum);

    virtual void  WriteTo(fstream& out) const;
    virtual void  ReadFrom(fstream& in);
    virtual long  DiskSize(void) const ;

    virtual void  SetDisplayField(EditWindow *e);
    virtual void  ReadDisplayField(EditWindow *e);
protected:
    virtual void  ConsistencyUpdate(EditWindow *e);
    virtual void  AddFieldsToWindow();
    char   fStudentName[64];
    long   fMark1;
    long   fMark2;
    long   fMark3;
    long   fMark4;
    long   fMidSession;
    long   fFinalExam;

    NumberItem    *fN;
    long   fTotal;
};
```

The data required include a student name and marks for various assignments and examinations. There is also a link to a `NumberItem` that will be used to display the total mark for the student.

The constructor does whatever an ordinary `Record` does to initialize itself, then sets all its own data members:

```
StudentRec::StudentRec(long recnum) : Record(recnum)
{
    fMark1 = fMark2 = fMark3 = fMark4 =
     fMidSession = fFinalExam = 0;
    strcpy(fStudentName,"Nameless");
```

```
}
```

The function `AddFieldsToWindow()` populates the `RecordWindow` with the necessary `EditText` and `EditNum` editable subwindows:

Building the display structure

```
void StudentRec::AddFieldsToWindow()
{
    Record::AddFieldsToWindow();

    EditText *et = new EditText(1001, 5, 4, 60,
    "Student Name ");
    fRW->AddSubWindow(et);

    EditNum *en = new EditNum(1002, 5, 8, 30,
    "Assignment 1 (5) ", 0, 5,1);
    fRW->AddSubWindow(en);
    en = new EditNum(1003, 5, 10, 30,
    "Assignment 2 (10) ", 0, 10,1);
    fRW->AddSubWindow(en);
    …
    en = new EditNum(1007, 40, 10, 30, "Examination (50) ",
    0, 60,1);
    fRW->AddSubWindow(en);

    fTotal = fMark1 + fMark2 + fMark3 + fMark4
           + fMidSession + fFinalExam;
    fN = new NumberItem(2000, 40, 14,30, "Total ", fTotal);
    fRW->AddSubWindow(fN);
}
```

Adding an EditText

Adding some EditNum subwindows

Adding a NumberItem

The call to the base class function, `Record::AddFieldsToWindow()`, gets the `NumberItem` for the record number added to the `RecordWindow`. An extra `NumberItem` subwindow is added to hold the total.

Functions `SetDisplayField()` and `ReadDisplayField()` transfer data to/from the `EditText` (for the name) and `EditNum` windows:

Data exchange with editable windows

```
void StudentRec ::SetDisplayField(EditWindow *e)
{
    long id = e->Id();
    switch(id) {
case 1001:
     ((EditText*)e)->SetVal(fStudentName, 1);
    break;
case 1002:
     ((EditNum*)e)->SetVal(fMark1,1);
    break;
case 1003:
     ((EditNum*)e)->SetVal(fMark2,1);
    break;
case 1004:
     ((EditNum*)e)->SetVal(fMark3,1);
    break;
case 1005:
     ((EditNum*)e)->SetVal(fMark4,1);
    break;
```

Setting an EditText

Setting EditNum

```
case 1006:
     ((EditNum*)e)->SetVal(fMidSession,1);
     break;
case 1007:
     ((EditNum*)e)->SetVal(fFinalExam,1);
     break;
     }
}

void StudentRec::ReadDisplayField(EditWindow *e)
{
    long id = e->Id();
    switch(id) {
case 1001:
    char* ptr = ((EditText*)e)->GetVal();
    strcpy(fStudentName, ptr);
    break;
case 1002:
    fMark1 = ((EditNum*)e)->GetVal();
    break;
…
…
case 1007:
    fFinalExam = ((EditNum*)e)->GetVal();
    break;
    }
}
```

Reading an EditText

Reading an EditNum

The `ReadFrom()` and `WriteTo()` functions involve a series of low level read (write) operations that transfer the data for the individual data members of the record:

Reading from file

```
void StudentRec::ReadFrom(fstream& in)
{
    in.read((char*)&fRecNum, sizeof(fRecNum));
    in.read((char*)&fStudentName, sizeof(fStudentName));
    in.read((char*)&fMark1, sizeof(fMark1));
    in.read((char*)&fMark2, sizeof(fMark2));
    in.read((char*)&fMark3, sizeof(fMark3));
    in.read((char*)&fMark4, sizeof(fMark4));
    in.read((char*)&fMidSession, sizeof(fMidSession));
    in.read((char*)&fFinalExam, sizeof(fMidSession));
}
```

Writing to file

```
void StudentRec::WriteTo(fstream& out) const
{
    out.write((char*)&fRecNum, sizeof(fRecNum));
    out.write((char*)&fStudentName, sizeof(fStudentName));
    out.write((char*)&fMark1, sizeof(fMark1));
    …
    out.write((char*)&fFinalExam, sizeof(fMidSession));
}
```

(The total does not need to be saved, it can be recalculated.)

The DiskSize() function computes the size of a record as held on disk:

```
long StudentRec::DiskSize(void) const
{
    return sizeof(fRecNum) + sizeof(fStudentName)
    + 6 * sizeof(long);
}
```

Function `ConsistencyUpdate()` is called after an editable subwindow gets changed. In this example, changes will usually necessitate the recalculation and redisplay of the total mark:

```
void StudentRec::ConsistencyUpdate(EditWindow *e)
{
    fTotal = fMark1 + fMark2 + fMark3 + fMark4
    + fMidSession + fFinalExam;
    fN->SetVal(fTotal, 1);
}
```

EXERCISES

1. Implement all the code of the framework and the StudentMark application.

2. Create new subclasses of class Collection and class Document so that an AVL tree can be used for storage of data records.

3. Implement the Loans program using your AVLDoc and AVLCollection classes.

   The records in the files used by the Loans program are to have the following data fields:

   customer name (64 characters)
   phone number (long integer)
   total payments this year (long integer)
   an array with five entries giving film names (64 characters) and weekly charge
   for movies currently on loan.

   The program is to use a record display similar to that illustrated in Figure 30.3.

4. There is a memory leak. Find it. Fix it correctly.
   (Hint, the leak will occur with a disk based collection. It is in one of the Document functions. The "obvious" fix is wrong because it would disrupt memory based collection structures.)