BORMICON

Programmer's manual

Draft

Halldór Narfi Stefánsson
Hersir Sigurgeirsson
Höskuldur Björnsson

November 7, 1996

Contents

1 Concurrency. **7**
1.1 Left to do . . . 7

2 Streams. **11**
2.1 CommentStream . . . 11

3 Fundamental classes. **15**
3.1 TimeClass . . . 15
3.2 AreaClass . . . 19

4 Base classes. **21**
4.1 LivesOnAreas . . . 21
4.2 HasName . . . 23
4.3 BaseClass . . . 24
4.4 Descendants . . . 27

5 Error messages and optimization. **29**
5.1 StrStack . . . 29
5.2 Keeper . . . 30
5.3 Formula . . . 36
5.4 Formulavector . . . 38
5.5 Formulaindexvector . . . 40
5.6 ErrorHandler . . . 42
5.7 StochasticData . . . 42

6 Data repositories. **45**
6.1 vector<T> . . . 45
6.2 doublevector . . . 45
6.3 doubleindexvector . . . 47
6.4 bandmatrix . . . 50
6.5 bandmatrixvector . . . 54
6.6 Agebandmatrix . . . 56
6.7 Agebandmatrixvector . . . 60
6.8 BinarySearchTree . . . 63

7 Lengths and conversion. 67
7.1 LengthGroupDivision 67
7.2 ConversionIndex 71

8 Predator – prey. 79
8.1 Overview 79
8.2 Suitability 80
8.3 Predator 83
8.4 PopPredator 89
8.5 StockPredator 92
8.6 LengthPredator 95
8.7 TotalPredator 96
8.8 LinearPredator 98
8.9 Prey 100
8.10 LengthPrey 105
8.11 StockPrey 106
8.12 Notes on Predator. 108
8.13 Eating functions – details 109

9 Catch. 111
9.1 CatchData 111
9.2 Catch 113

10 Stock and supplamentary. 115
10.1 Transition 115
10.2 NaturalM 118
10.3 InitialCond 120
10.4 Migration 122
10.5 Grower 126
10.6 GrowthImplementparameters 129
10.7 GrowthCalc 131
10.8 Maturity 135
10.9 MaturityA 138
10.10MaturityB 139
10.11Spawner 141
10.12Renewaldata 143
10.13Stock 146

11 Supplamentary classes. 153
11.1 ActionAtTimes 153

12 Standard aggregation and printing. 155
12.1 Overview 155
12.2 Printer 155

12.3 StockStdPrinter . . . 158
12.4 PredPreyStdPrinter . . . 159
12.5 PredPreyStdLengthPrinter . . . 160
12.6 PredPreyStdAgePrinter . . . 161
12.7 StockFullPrinter . . . 162
12.8 AbstrPreyStdInfoByLength . . . 163
12.9 PreyStdInfoByLength . . . 164
12.10StockPreyStdInfoByLength . . . 166
12.11PredStdInfoByLength . . . 167
12.12AbstrPreyStdInfo . . . 169
12.13PreyStdInfo . . . 171
12.14StockPreyStdInfo . . . 172
12.15AbstrPredStdInfo . . . 173
12.16PredStdInfo . . . 176
12.17StockPredStdInfo . . . 177
12.18Details of computations. . . . 179

13 Nonstandard aggregation and printing. 181
13.1 Overview . . . 181
13.2 StockPrinter . . . 181
13.3 PredatorPrinter . . . 183
13.4 PredatorOverPrinter . . . 183
13.5 PreyOverPrinter . . . 184
13.6 StockPreyFullPrinter . . . 185
13.7 StockAggregator . . . 186
13.8 PredatorAggregator . . . 189
13.9 PredatorOverAggregator . . . 191
13.10PreyOverAggregator . . . 193

14 Statistics. 195
14.1 LinearRegression . . . 195
14.2 LogLinearRegression . . . 198
14.3 PopStatistics . . . 200

15 Likelihood. 203
15.1 Overview . . . 203
15.2 Likelihood . . . 203
15.3 SurveyIndices . . . 206
15.4 SIOnStep . . . 208
15.5 SIByLengthOnStep . . . 212
15.6 SIByAgeOnStep . . . 214

Chapter 1

Concurrency.

1.1 Left to do

Almost all of the classes are designed so as to allow for a simulation to make use of a multiprocessor machine, using an appropriate library. The classes that handle the simulation should be able to handle simultaneously the simulation of the areas on a given time step. And the classes that handle the writing to file should be able to write to file all at the same time.

There are several levels at which a class can support this parallelism, or multi-threading. Note that the first level is a standard level in the classification of multi-threading safety, but the others are not.

- A class can be fully MT-safe (MT - multi-threading). I.e. once it has been created, all its member functions may safely be called in different threads of control.
- A class can be minimally MT-safe, i.e. created and used in only one thread.
- A class can be not MT-safe at all, i.e. the thread creating and using the class must be the only running thread while the class exists.

All the classes should be considered minimally MT-safe unless explicitly documented otherwise.

In order to classify the classes and their member functions, the member functions have been marked:

$\mathcal{S}$ The member functions marked this way may be invoked in a running thread, but they must be the only (S for **S**ingle) member function of that instance of the class that is being called at the time. These member functions may require synchronization with other member functions, i.e. the other member functions may return different values after the one marked $\mathcal{S}$ has been invoked.

$\mathcal{C}$ The C stands for **C**are. I.e. care must be taken when using the member function. The member function is a bit safer than the ones marked with an S, but it may

- change the status of the object if used in a particular way, consider e.g. the following code fragment:

  ```
  doublevector d(10, 5.7); //d is of length 10,
      //initialized to 5.7
  ...
  double x = d[0]; //Fully safe usage of d
  d[1] = 3.84; //Care must be taken !
  ```

  Here, one must be sure that other threads do not try to access d[1] at the same time as it is set as conflicts would arise.

- require some synchronization with other member functions. There are e.g. access functions that return objects with reference to examine a class's status at the end of simulating one time step. The reference must be copied before the simulation of the next time step begins. E.g. `StockPredator::FPhi`.

$\mathcal{N}$ N for **N**ot. I.e. the member function must be in the only running thread.

$\mathcal{F}$ F for **F**ully. The member function is fully MT-safe, but care must be taken in the use of the member functions marked $\mathcal{S}$ and $\mathcal{C}$, though. The member functions marked $\mathcal{F}$ may be each be used in a thread of their own at wish, with no special need for synchronization.

$\mathcal{A}$ A stands for **A**rea, meaning that calls to the member function with different values of the parameter area may run in parallel.

In some classes, a sequence of member functions should be called in a particular order. Then it may be possible to thread a group of calls. This should be possible to infer from the pre- and postconditions of the member functions, but for simplicity a new token was added:

$\mathcal{G}$ G for **G**roup. When a member function is marked with G, it belongs to a group of functions, that may be threaded.

This is useful if a member function is marked with $\mathcal{A}$ and $\mathcal{G}$, because one can quickly see that the calls to the member functions may be as in Fig. 1.2 instead of as in Fig.

1.1.

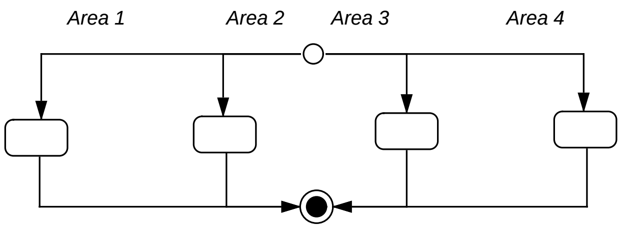

Figure 1.1: A trivial division of calls to a member function for different areas.

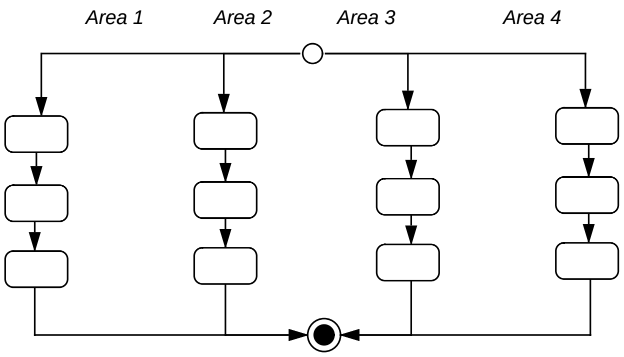

Figure 1.2: Threaded sequential calls to member functions, for different areas.

Chapter 2

Streams.

2.1 CommentStream

The class CommentStream is a simple link to istream. It is meant to function in exactly the same way as istream, except when it finds a semicolumn (';') – then the rest of that line is considered a comment which may be ignored.

When CommentStream finds a semicolumn, it searches for the next line not starting with a semicolumn (ignoring spaces and tabs). When that line has been found, or end of file reached, it stations the istream pointer at the first non-white space character in the line. [This is in fact a crude description of the function KillComments which handles this.]

Inheritance

class CommentStream

Public messages

`CommentStream(istream&)`

Use:	CommentStream cstr(istr)
Pre:	None.
Post:	cstr is of type CommentStream. Now, istr can be accessed through cstr.
NB:	The lifetime of istr has to be greater than or equal to the last operation on cstr. Note that the user has to close istr.

```
CommentStream& operator>>(int&)
CommentStream& operator>>(char* a)

CommentStream& operator>>(char& a)
```

`CommentStream& operator>>(short& a)`

`CommentStream& operator>>(long& a)`

`CommentStream& operator>>(float& a)`

`CommentStream& operator>>(double& a)`

`CommentStream& operator>>(unsigned char* a)`

`CommentStream& operator>>(unsigned char& a)`

`CommentStream& operator>>(unsigned short& a)`

`CommentStream& operator>>(unsigned long& a)`

`CommentStream& operator>>(__commentmanip func)`

Use: cstr » func
Pre: None.
Post: The operation func has been performed on cstr.

`CommentStream& get(char& c)`

`int peek()`

`int bad()`

`void clear(int state = 0)`

`int eof()`

`int fail()`

`int good()`

`CommentStream& seekg(streampos Pos)`

`streampos tellg()`

```
CommentStream& getline(char* ptr, int len, char delim =
'\n')
```

Furthermore, a ws operator is provided. It is a friend of CommentStream.

Friend functions

CommentStream& ws(CommentStream&)
The use of ws is identical to that of ws on ios.

Example

Assume that the beginning of testfile is:

```
A small text ;Comment in the end of line

1
; 2
3
```

Then the following program

```
int main()
{
  ifstream infile("testfile");
  CommentStream cstr(infile);
  char text[250];
  for (int i = 0; i < 3; i++){
    cstr >> text;
    cout << text;
    }
  int x;
  for (int i = 0; i < 2; i++){
    cstr >> x;
    cout << x;
    }
  infile.close();
  }
```

will write to the standard output:

```
Asmalltext13
```

Protected Characteristics

istream*	istrptr	//

Chapter 3

Fundamental classes.

3.1 TimeClass

A class that keeps track of time; the beginning and end of a run and the time therein between.

Inheritance

class TimeClass

Public messages

A step is a subdivision of a year, and is most likely only a synonym for a month. By "period in question" we mean the period between FirstStep on FirstYear and LastStep on LastYear, inclusive.

`TimeClass(CommentStream&)`
Use: TimeClass *TimeInfo*(*infile*)
Pre: *infile* has no error bits set and its format is correct.
Post: *TimeInfo* has read its information from *infile*.

$\mathcal{F}$ `int CurrentStep() const`
Use: *step* = *TimeInfo*.CurrentStep()
Pre: None.
Post: *step* has the value of current step. In particular, $1 \leq step \leq$ *TimeInfo*.NoStepsInYear(). If the current time is on the last year, $step \leq$ *TimeInfo*.LastStep() and if it is on the first year, *TimeInfo*.FirstStep() $\leq step$.

$\mathcal{F}$ `int CurrentYear() const`
Use: *year* = *TimeInfo*.CurrentYear()
Pre: None.

Post: *year* has the value of current year. *TimeInfo*.FirstYear() $\leq$ *year* $\leq$ *TimeInfo*.LastYear().

$\mathcal{F}$ `int CurrentTime() const`

Use: *time* = *TimeInfo*.CurrentTime()
Pre: None.
Post: *time* holds current time. $1 \leq$ *time* $\leq$ *TimeInfo*.TotalNoSteps(). This is the same as *time* = *TimeInfo*.CalcSteps(*TimeInfo*.CurrentYear(), *TimeInfo*.CurrentStep()).

$\mathcal{F}$ `int FirstStep() const`

Use: *step* = *TimeInfo*.FirstStep()
Pre: None.
Post: *step* holds the first step. $1 \leq$ *step* $\leq$ *TimeInfo*.StepsInYear().

$\mathcal{F}$ `int FirstYear() const`

Use: *year* = *TimeInfo*.FirstYear()
Pre: None.
Post: *year* holds the first year.

$\mathcal{F}$ `int LastStep() const`

Use: *step* = *TimeInfo*.LastStep()
Pre: None.
Post: *step* holds the last step. $1 \leq$ *step* $\leq$ *TimeInfo*.StepsInYear().

$\mathcal{F}$ `int LastYear() const`

Use: *year* = *TimeInfo*.LastYear()
Pre: None.
Post: *year* holds the last year.

$\mathcal{F}$ `double LengthOfCurrent() const`

Use: *len* = *TimeInfo*.LengthOfCurrent()
Pre: None.
Post: *len* holds the length of the current step. $0 <$ *len* $\leq$ *TimeInfo*.LengthOfYear().

$\mathcal{F}$ `double LengthOfYear() const`

Use: *len* = *TimeInfo*.LengthOfYear()
Pre: None.
Post: *len* equals the sum of the length of steps in one year, thus *len* > 0.

$\mathcal{F}$ `double LengthOfStep(int) const`

Use: *len* = *TimeInfo*.LengthOfStep(s)
Pre: $1 \leq$ s $\leq$ *TimeInfo*.StepsInYear().

Post: *len* equals the length of time step s. If s equals *TimeInfo*.CurrentStep(), *len* equals *TimeInfo*.LengthOfCurrent().

$\mathcal{F}$ `int CalcSteps(int,int) const`

Use: *time* = *TimeInfo*.CalcSteps(*year*, *step*)
Pre: The year and step *year*, *step* are within the period in question.
Post: *time* holds the number of steps from the first until year *year* and step *step*; beginning the count in 1 when *year* is first year and *step* is first step. Thus $1 \leq$ *time* $\leq$ *TimeInfo*.TotalNoSteps().

$\mathcal{F}$ `int TotalNoSteps() const`

Use: *no* = *TimeInfo*.TotalNoSteps()
Pre: None.
Post: *no* holds the number of steps in the period in question. This is the same as *no* = *TimeInfo*.CalcSteps(*TimeInfo*.LastYear(), *TimeInfo*.LastStep()).

$\mathcal{F}$ `int StepsInYear() const`

Use: *no* = *TimeInfo*.StepsInYear()
Pre: None.
Post: *no* holds the number of steps in one year.

$\mathcal{N}$ `void IncrementTime()`

Use: *TimeInfo*.IncrementTime()
Pre: current year and step are not last year and last step.
Post: The current time has been incremented, meaning that either current step has been incremented or current step set to 1 and current year incremented, the latter if current step was the last step on that year.

$\mathcal{F}$ `int IsWithinPeriod(int, int) const`

Use: *is* = *TimeInfo*.IsWithinPeriod(*year*, *step*)
Pre: The year and step *year*, *step* is within the period in question.
Post: *is* equals 1 if the step *step* on the year *year* is within the period.

$\mathcal{N}$ `void ResetToBeginning()`

Use: *TimeInfo*.ResetToBeginning()
Pre: None.
Post: *TimeInfo* has reset its time to the first step on the first year.

Protected Characteristics

`int`	`currentstep`	// current step
`int`	`currentyear`	// current year
`int`	`laststep`	// last step
`int`	`firstyear`	// first year
`int`	`lastyear`	// last year
`int`	`firststep`	// first step
`int`	`notimesteps`	// number of timesteps in a year.
`doubleindexvector`	`timesteps`	// [step] - length of timesteps.
`double`	`lengthofyear`	// The length of a year.

3.2 AreaClass

A class that keeps area information and converts from the area numbers read in input files to **inner areas** used in the program. These inner areas are numbered with 0,1,..,n -1, where n is the number of areas.

Inheritance

class AreaClass

Public messages

`AreaClass(CommentStream&, const TimeClass* const)`

Use: AreaClass Area(*infile*, *TimeInfo*)

Pre: *infile* has no error bits set and its format it correct. $TimeInfo \neq 0$.

Post: Area has read information from *infile* for the period in question (that information is got from *TimeInfo*).

NB: Area rearranges the number of areas, so they are numbered sequentially from 0 to number of areas - 1 and these are the so called inner areas. All references to areas in function calls to Area must be made through inner areas, unless otherwise noted. Also, when referencing to time, it has to be within the period obtained from *TimeInfo*, else it is illegal.

$\mathcal{F}$ `int NoAreas() const`

Use: *no* = Area.NoAreas()

Pre: None.

Post: *no* holds the number of areas.

$\mathcal{F}$ `double Size(int) const`

Use: *size* = Area.Size(*area*)

Pre: *area* is an inner area.

Post: *size* holds the size of area *area*.

$\mathcal{F}$ `double Temperature(int, int) const`

Use: *temp* = Area.Temperature(*area*, *time*)

Pre: *area* is an inner area and time a legal *time*, i.e. $0 \leq area <$ Area.NoAreas() and $1 \leq time \leq TimeInfo \rightarrow$TotalNoSteps()).

Post: *temp* has the temperature that Area read for area *area* on time *time*.

$\mathcal{F}$ `int InnerArea(int) const`

Use: *innerarea* = Area.InnerArea(*area*)

Pre: *area* is one of the areas Area read in the constructor.

Post: *innerarea* is the inner area that Area associates with *area*.

$\mathcal{F}$ `int OuterArea(int) const`

Use: *outerarea* = Area.OuterArea(*innerarea*)
Pre: *innerarea* is an inner area, i.e. $0 \leq$ *innerarea* < Area.NoAreas().
Post: *outerarea* has the value of outer area to which Area associated *innerarea*.

Protected Characteristics

`intvector`	`OuterAreas`	// The area numbers read in the constructor.
`doublevector`	`size`	// Area sizes.
`doublematrix`	`temperature`	// temperature[*time*][*area*] is the temp.

Chapter 4

Base classes.

4.1 LivesOnAreas

The class LivesOnAreas is meant to be a base class for every object that is defined on many areas. The class is a bit too open, though, since its data is only protected, instead of private.

The class knows the areas on which it lives, and keeps a list of inner areas. The inner areas are numbered sequentially, starting at 0.

Inheritance

class LivesOnAreas

Public messages

`LivesOnAreas()`

Use: LivesOnAreas L
Pre: None.
Post: L is of type LivesOnAreas and does not live on any areas.

`virtual ~LivesOnAreas()`

Use: ~L
Pre: None.
Post: All memory belonging to L has been freed.

`LivesOnAreas(const intvector&)`

Use: LivesOnAreas L(*Areas*)
Pre: The elements of *Areas* are ≥ 0.
Post: L is of type LivesOnAreas and lives on the areas *Areas*[i], $i = 0, \ldots,$ *Areas*.Size() - 1.

$\mathcal{F}$ `int IsInArea(int) const`

Use: *isin*= L.IsInArea(*area*)
Pre: None.
Post: If L lives on area *area*, *isin*equals 1, else 0.

Protected messages

$\mathcal{S}$ `void LetLiveOnAreas(const intvector&)`

Use: L.LetLiveOnAreas(*Areas*)
Pre: *Areas*.Size() $\geq$ 0, the elements of *Areas* are $\geq$ 0 and all unequal.
Post: L lives on the areas *Areas*[i], i = 0, ..., *Areas*.Size() - 1.

Since some derived classes will not know on creation time the areas on which they are defined, they have to use this message to inform LivesOnAreas of them.

Protected Characteristics

`intvector`	`areas`	// The areas on which the object lives.
`intvector`	`AreaNr`	// For conversion to inner areas.

Data invariant

`AreaNr`[`areas`[i]] == i, $i = 0, \ldots,$ `areas`.Size() - 1. If j is not an element of areas, and $0 \leq j <$ AreaNr.Size() then `AreaNr`[j] < 0.

This is the method for converting to inner areas; if the object is defined on area k and i is such that `areas`[i] == k, then the area k is converted to the inner area i == `AreaNr`[`areas`[i]] == `AreaNr`[k].

4.2 HasName

The class HasName is a base class for any object that has a name.

Inheritance

class HasName

Public messages

`HasName()`

Use: HasName H
Pre: None.
Post: H is of type HasName and the name of H is the empty string (`"\0"`).

`virtual ~HasName()`

Use: ~H
Pre: None.
Post: All memory belonging to H has been freed.

`HasName(const char*)`

Use: HasName H(*givenname*)
Pre: *givenname* is a nullterminated string.
Post: H is of type HasName; H has copied *givenname* which is now its name.

$\mathcal{F}$ `const char* Name() const`

Use: str = H.Name()
Pre: None.
Post: str points to the beginning of a nullterminated string containing the name of H.

Private Characteristics

`char*`	`name`	// Contains the name – nullterminated.

4.3 BaseClass

Warning: The description may be obsolete — better look at the header file for the functions First-, Second- and ThirdSpecialTransactions.

In the abstract class BaseClass we declare some important member functions. Since the member functions are pure virtual, they have no pre- or postconditions. On the other hand, there is a recommended usage which is described below.

Since the class is abstract, an object of this type can not be instantiated, nevertheless, the constructors are described as if this were possible.

Inheritance

class BaseClass : public HasName, public LivesOnAreas

Public messages

`BaseClass()`

Use: BaseClass()

`BaseClass(const char*)`

Use: BaseClass B(*givenname*)
Pre: *givenname* points to a nullterminated string.
Post: *givenname* is the name of B.

`BaseClass(const char*, const intvector&)`

Use: BaseClass B(*givenname*, *Areas*)
Pre: *givenname* points to a nullterminated string, *Areas* is a vector of integers ≥ 0. *Areas*.Size() ≥ 0.
Post: *givenname* is the name of B and B lives on the areas *Areas*.

`virtual ~BaseClass()`

Use: ˜B
Pre: None.
Post: All memory belonging to B has been freed.

`virtual void CalcEat(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.CalcEat(*area*, *Area*, *TimeInfo*)
NB: As the name suggest, this function should calculate the amount B wanted to eat and report it to the objects that B wants to eat.

`virtual void CheckEat(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.CheckEat(*area*, *Area*, *TimeInfo*)

NB: This function should check if the amount eaten by B was accepted. – call B.CalcEat(*area*, *Area*, *TimeInfo*) first.

`virtual void AdjustEat(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.AdjustEat(*area*, *Area*, *TimeInfo*)

NB: The function should adjust the eating according to the result from B.CheckEat which should be called first.

`virtual void ReducePop(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.ReducePop(*area*, *Area*, *TimeInfo*)

NB: Reduction of the population caused by some internal factors.

`virtual void Grow(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.Grow(*area*, *Area*, *TimeInfo*)

NB: Calculates the growth and updates accordingly.

`virtual void FirstSpecialTransactions(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.FirstSpecialTransactions(*area*, *Area*, *TimeInfo*)

NB: Special transactions may need to be split in three parts. The first transactions are between objects.

`virtual void SecondSpecialTransactions(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.SecondSpecialTransactions(*area*, *Area*, *TimeInfo*)

NB: This is an intermediate stage in the transactions to allow for update of internal information.

`virtual void ThirdSpecialTransactions(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.ThirdSpecialTransactions(*area*, *Area*, *TimeInfo*)

NB: The final stage in the transactions; here we do everything that depends on SecondTransactions to be finished.

`virtual void CalcNumbers(int, const AreaClass* const, const TimeClass* const) = 0`

Use: B.CalcNumbers(*area*, *Area*, *TimeInfo*)

NB: Internal update.

`virtual void Migrate(const TimeClass* const) = 0`

Use: B.Migrate(*TimeInfo*)

NB: Migration between areas.

`virtual void Reset() = 0`
Use: B.Reset()
Pre: None.
Post: B has recalculated its internal information and reset its internal variables and reestimated its error status.

$\mathcal{F}$ `int Error() const`
Use: *err*= B.Error() const
Pre: None.
Post: *err*is 0 if B does not have its error bit set. Else err equals 1, meaning that the effects of further actions on B are undefined until B.Clear() and B.Reset() have been successfully called.

$\mathcal{F}$ `virtual void Clear()`
Use: B.Clear()
Pre: None.
Post: The error status in B equals 0.

`virtual void Print(ofstream &) const = 0`
Use: B.Print(*outfile*)
NB: Print internal information to *outfile*.

Protected messages

$\mathcal{N}$ `void SetError()`
Use: B.SetError()
Pre: None.
Post: B has set its error bit.

Private Characteristics

`int`	`error`	// The error status.

4.4 Descendants

The classes LivesOnAreas and HasName are widely used. The following picture shows their descendants.

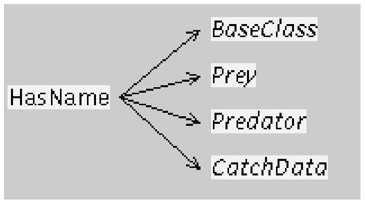

Figure 4.1: Descendants of HasName.

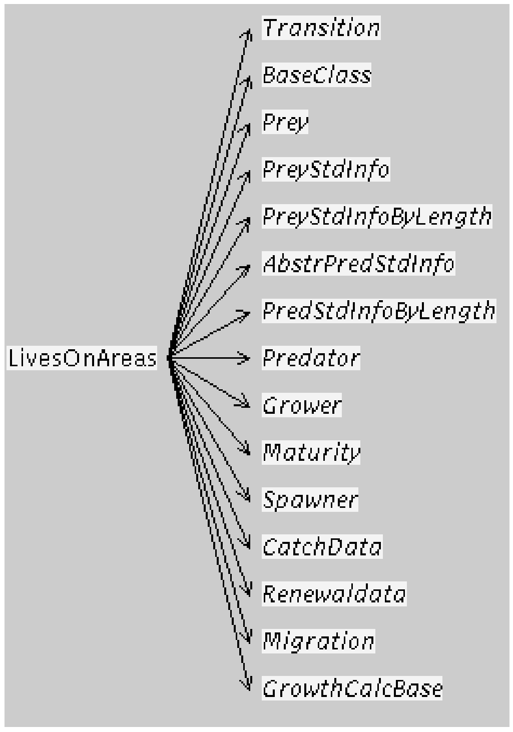

Figure 4.2: Descendants of LivesOnAreas.

Notice how many classes inherit from LivesOnAreas, as they are all area based. And those who need identification by name, all inherit from HasName.

The class BaseClass is inherited by Stock, OtherFood and Fleet as shown:

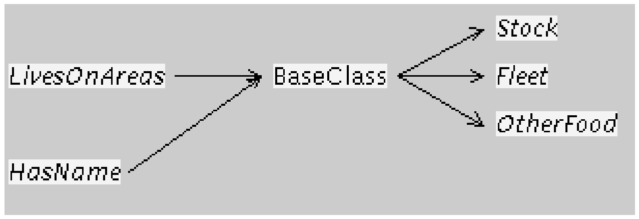

Figure 4.3: Descendants of BaseClass.

Chapter 5

Error messages and optimization.

The following classes are used for formulas with marked variables which are allowed to change between simulations in an optimization.

5.1 StrStack

This is a stack that keeps strings.

Inheritance

class StrStack

Public messages

`StrStack()`

Use: StrStack S
Pre: None.
Post: S is an empty StrStack.

`~StrStack()`

Use: ~S
Pre: None.
Post: All memory belonging to S has been freed.

`void OutOfStack()`

Use: S.OutOfStack()
Pre: None.
Post: If S is not empty, the top element has been taken off else nothing has been done.

`void PutInStack(const char*)`

Use: S.PutInStack(*str*)

Pre: *str* points to a nullterminated string.
Post: A copy of *str* is the top element of S.

`void ClearStack()`
Use: S.ClearStack
Pre: None.
Post: S is empty.

`char* SendAll() const`
Use: *str* = S.SendAll()
Pre: None.
Post: *str* points to a nullterminated string, created with new. The string contains the concatenation of all the strings in S, with the bottom string in S first and the top string last.
It is the users respolsibility to delete str.

Protected Characteristics

`int`	`sz`	// The number of strings in the stack.
`charptrvector`	`v`	// Contains the strings in the stack.

Data invariant

Every sequence of using messages must maintain the following data invariant.

- The size of `v` is greater than or equal to `sz`.

Details

When popping from the stack, `sz`is descreased, but the size of `v`is not decreased. This is done in order to increase performance both in the pop operation and when the next push occurs. The length of every characther array keeping a string on the stack is equal to the constant `MaxStrLength`, so this is the maximum string length for the stack.

5.2 Keeper

This class keeps the names and addresses of variables. It is useful for clearer error messages when reading from files but is mostly used when doing optimization.

A word of caution: When a Keeper has received information on a variable through KeepVariable(double&, int), it may be informed to change the value of that variable through the member function Update. Then it is of course vital that the original variable received in KeepVariable(...) still exists and is in the same place in memory – else undefined behaviour can be expected. This means e.g. that the member function resize in vector classes should be used with caution.

As a result of the previous paragraph, every class that allows Keeper to keep some of its variables must take care to avoid the situation described there. It must also be aware that every variable it allows Keeper to access may change in value and therefore, it may be necessary to provide a member function that recalculates inner state of the object according to the updated value of the variable(s). That member function should then be called after calling Update in Keeper. Also, the class has to be aware of the danger of Keeper assigning illegal values to the variables it has allowed the Keeper to access. It must then take some action, perhaps by setting an error status indicating a nonrecoverable error, or assign a new value to the variable and then proceed.

Inheritance

class Keeper

Public messages

`Keeper()`

Use: Keeper K
Pre: None.
Post: K is of type Keeper.

`~Keeper()`

Use: ~K
Pre: None.
Post: All memory belonging to K has been freed.

`void KeepVariable(double&,int)`

Use: K.KeepVariable(*var*,*attribute*)
Pre: *attribute* ≥ 0.
Post: If K.KeepVariable(double&, int) has been called before and K received a variable with the same *attribute* but not the same value, an error message is emitted (fatal). Else K keeps *var*, the address of *var* and *attribute*, and associates the kept strings to it.

`void DeleteParam(const double&)`

Use: K.DeleteParam(*var*)
Pre: None.
Post: If K keeps information on the variable *var*, it is deleted.

`void ChangeVariable(const double&, double&)`

Use: K.ChangeVariable(*pre*, *post*)

Pre: None.
Post: Nothing is done if K has no information on *pre*. If K has information on the variable *pre*, the address of *post* is kept instead. However, if *pre* and *post* do not have the same value, an error message is emitted (fatal).

`ClearLast()`
Use: K.ClearLast()
Pre: None.
Post: Last kept string, if any, is cleared.

`void ClearLastAddString(const char*)`
Use: K.ClearLastAddString(*str*)
Pre: *str* is a nullterminated string.
Post: The last kept string, if any, in K is cleared. A copy of the string *str* is kept in K.

`void ClearAll()`
Use: K.ClearAll()
Pre: None.
Post: All kept strings in K, if any, are cleared.

`void SetString(const char*)`
Use: K.SetString(*str*)
Pre: *str* is a nullterminated string.
Post: All previous strings kept in K are deleted. K keeps a copy of *str*.

`void AddString(const char*)`
Use: K.AddString(*str*)
Pre: *str* is a nullterminated string.
Post: A copy of the string *str* is kept in K.

`char* SendString() const`
Use: *str* = K.SendString()
Pre: None.
Post: *str* is a nullterminated string, containing the concatenation of all the strings kept in K. It is created with new and it is the users responsibility to delete it.

`void AddComponent(const char*)`
Use: K.AddComponent(*str*)
Pre: *str* is the name of a likelihood component.
Post: A copy of the string *str* is kept in K.

`char* SendComponent() const`

Use: *str* = K.SendComponent()
Pre: None.
Post: *str* is a nullterminated string, containing the concatenation of all names of likelihood components kept in K, separated with a tab character. It is created with new and it is the users responsibility to delete it.

`void ClearComponents()`

Use: K.ClearComponents()
Pre: None.
Post: All kept names of likelihood components in K, if any, are cleared.

`int ErrorInUpdate() const`

Use: *keeper* = K.ErrorInUpdate()
Pre: None.
Post: If K has the error bit set, *keeper* is 1, else 0.

`void Update(const doublevector&)`

Use: K.Update(*val*)
Pre: None.
Post: If *val*.Size() does not equal the number of different attributes K has received in KeepVariable, the error bit is set and no further action taken. Else, all the variables with the i-th attribute received in KeepVariable are given the value *val*[i], using the addresses of the variables received in KeepVariable, $0 \leq i \leq$ *val*.Size() - 1.

`void Clear()`

Use: K.Clear()
Pre: None.
Post: The error bit in K is set to zero.

`void ValuesOfVariables(doublevector&) const`

Use: K.ValuesOfVariables(*val*)
Pre: *val*.Size() == K.NoVariables()
Post: *val* has the values of the kept variables.

`void InitialValues(doublevector&) const`

Use: K.ScaledValues(*val*)
Pre: *val*.Size() == K.NoVariables()
Post: *val* has the initial values of the kept variables.

`void ScaledValues(doublevector&) const`

Use: K.ScaledValues(*val*)

Pre: *val*.Size() == K.NoVariables()

Post: *val* has the scaled values of the kept variables. Scaled values are the values of the variables divided by the initial values. The purpose of scaled values is to let all variables in optimization have approximately the same size. The scaled values are then sent to the optimization routine.

`int NoVariables() const`

Use: n = K.NoVariables()

Pre: None.

Post: n equals the number of attributes K has received through KeepVariable.

`void OpenFile(const char*)`

Use: K.OpenFile(*outfile*)

Pre: *outfile* is a nullterminated string. K has the permission to overwrite a file with the name *outfile*.

Post: K has opened the file *outfile* and is ready to write to it, starting at the beginning.

`void WriteInitialInformation(const Likelihoodptrvector&)`

Use: K.WriteInitialInformation(*Likely*)

Pre: K.OpenFile(const char*) has been called.

Post: K has written the names of the kept variables to file, along with their attributes, the names of the likelihood components, the types of the likelihood functions, and their weight, according to the information received in *Likely*.

`void WriteValues(double, const Likelihoodptrvector&)`

Use: K.WriteValues(*keepFuncValue*, *Likely*)

Pre: K.OpenFile(const char$*$) has been called.

Post: K has written the values of the variables received through KeepVariable to file and the values of the individual likelihood components **unweighted**, followed by *keepFuncValue*.

`void ScaleVariables()`

Use: K.ScaleVariables()

Pre: None. Values of all variables in the keeper must be set.

Post: Initial values in the keeper are equal to the values of the variables except the value of a variable is zero. If the value of a variable is zero the initial value is set to one and the scaled value to zero. Else the scaled value is set to one.

Example

```
void f(ifstream& infile, Keeper keeper)
{
  int fvar;
  keeper.AddString("fvar");
  infile >> fvar;
  if (infile.fail()) {

    cerr << "Failed reading variable " << keeper.SendString() << endl;
    exit(1);
  }
  keeper.ClearLast();
}


int main()
{
  Keeper keeper;
  ifstream infile("testfile");
  keeper.SetString("testfile");
  f(infile, keeper);
  infile.close();
  infile.open("secondtestfile", ios::in);
  keeper.ClearLastAddString("secondtestfile");
  f(infile, keeper);
  keeper.ClearLast();
  return 1;
}
```

This way, clearer error messages can be given. This were especially true if the function f would call another function to read from infile.

Protected Characteristics

`addr_keepmatrix`	`address`	// Information on all the variables received.
`doublevector`	`initialvalues`	// initial values.
`doublevector`	`scaledvalues`	// scaled values.
`doublevector`	`values`	// The values received.
`StrStack*`	`stack`	// The strings received.
`intvector`	`switches`	// The switches received.
`int`	`error`	// Internal error status. Is either 0 or 1.
`ofstream`	`outfile`	// The file for output.
`int`	`FileIsOpen`	// == 1 iff the file is open.

Data invariant

- `address`.Nrow() == `values`.Size() == `switches`.Size()
- In `values`[i] is the value of all the variables received with the attribute `switches`[i]. See also Keeper::KeepVariable(double &).
- In `address`[i] are all the variables received with attribute `switches`[i].

5.3 Formula

This class handles formulas. It is used for variables that are intended for optimization. It uses an object of type Keeper to change the values of the variables, but the user should not be concerned exactly how.

These steps should be followed when working with objects of type Formula. Note that an object of type Formula can only be given value by reading a formula from file.

1. The object is created with the default constructor:
   `Formula F;`
2. A formula is read from file:
   `infile >> F;`
3. keeper is informed about the marked variables in F:
   `F.Inform(keeper);`
4. The formula can be used as an rvalue in any expression as if it were a double:
   `double d = exp(F)/(0.2*F) + F;`

Inheritance

class Formula

Public messages

`Formula()`

Use: Formula F
Pre: None.
Post: F is of type Formula with value 0.

`friend CommentStream& operator >>(CommentStream&, Formula&)`

Use: *infile* » F
Pre: None.

Post: If *infile*'s format is correct, F has read the formula from it, and initialized itself. If *infile*'s format is incorrect, *infile*'s badbit has been set. If any switch read is < 0 it has been set to its absoulute value.

`operator double() const`

Use: double *val* = F

Pre: operator» and F.Inform(Keeper$*$) have been called.

Post: *val* contains the value of F.

`void Inform(Keeper*)`

Use: F.Inform(*keeper*)

Pre: operator» has been called.

Post: F has informed *keeper* of the marked variables in F.

`void Interchange(Formula&, Keeper*)`

Use: F.Interchange(NewF, *keeper*)

Pre: F.Inform(Keeper$*$) has been called.

Post: NewF is a copy of F, and *keeper*has been informed of the variables in NewF instead of F. This should be used when altering the size of an array of objects of type Formula.

Example

```
int main()
{
  Keeper keeper;
  Formula* F1 = new Formula;
  CommentStream infile("testfile");
  if( !(infile >> F1) ) exit(1); //FAILURE
  F.Inform(&keeper)
  Formula* F2 = new Formula;
  F1.Interchange(F2, &keeper);
  //keeper now has information on F2 instead of F1
  delete F1;
  double result = F2*F2/(F2 + 1);
  return 1;
}
```

Protected Characteristics

`double`	`init`	// The first variable.
`int`	`attr`	// The attribute of the first variable.
`doublevector`	`multipliers`	// The multipliers.
`intvector`	`attributes`	// The attributes of the multipliers.

Data invariant

- `attr` contains the switch of the first variable, if one was read, else it is -1.
- `multipliers`.Size() == `attributes`.Size().
- If `multipliers`.Size() > 0, `attributes`[i] contains the switch of the variable `multipliers`[i] for $0 \leq i <$ `multipliers`.Size().
- If failure occurs in operator» (due to wrong format of *infile*), F contains the formula read upto the error, and its state is ok (i.e. it is usable).

5.4 Formulavector

When initializing objects of type Formulavector, either either one of the following procedures can be used:

- The size is set, either with resize or at instantiation, the formulas are read from file and keeper informed:

```
Formulavector Fvec(4);
infile >> Fvec;
Fvec.Inform(keeper);
double d = Fvec[1] + Fvec[3];
```

- The vector is created of size 0 and one element added to it at a time:

```
Formulavector Fvec;
for (int i = 0; i < 4; i++){
  Fvec.resize(1, keeper);
  infile >> Fvec[i];
  Fvec[i].Inform(keeper);
}
double d = Fvec[1] + Fvec[4];
```

Inheritance

class Formulavector

Public messages

`Formulavector()`

Use: Formulavector Fvec
Pre: None.
Post: Fvec is a Formulavector of size 0.

`Formulavector(int)`

Use: Formulavector Fvec(*sz*)
Pre: None.
Post: Fvec is a Formulavector of size *sz*, if $sz > 0$, else of size 0.

`resize(int, Keeper*)`

Use: Fvec.resize(*sz*, *keeper*)
Pre: None.
Post: Fvec has enlarged its size by *sz*. The extra positions are added at the end of Fvec, and Fvec informs *keeper* of changed positions if necessary.

`friend CommentStream& operator >>(CommentStream&, Formulavector&)`

Use: *infile* » Fvec
Pre: None.
Post: If *infile*'s format is correct, Fvec has read the formulas from it, and initialized itself. If *infile*'s format is incorrect, *infile*'s badbit has been set. If any switch read is < 0 it has been set to its absoulute value.

`int Size() const`

Use: int $sz =$ Fvec.Size()
Pre: None.
Post: *sz* is equal to the size of Fvec.

`void Inform(Keeper*)`

Use: Fvec.Inform(*keeper*)
Pre: operator» has been called.
Post: Fvec has informed *keeper* of the marked variables in the formulas in Fvec.

`Formula& operator[](int)`

Use: Formula& F = Fvec[i]
Pre: $0 \leq i <$ Fvec.Size().
Post: F is a reference to the formula Fvec[i].

`const Formula& operator[](int) const`

Use: Formula& F = Fvec[i]
Pre: $0 \leq i <$ Fvec.Size().
Post: F is a reference to the formula Fvec[i].

Protected Characteristics

`int`	`size`	// The number of elements of the vector.
`Formula*`	`v`	// The Formulas.

Data invariant

- `size` ≥ 0.
- `size` $==$ 0 if and only if `v` $==$ 0.
- The formulas are contained in `v`[0], . . ., `v`[`size` - 1].
- If failure occurs in operator» (due to wrong format of *infile*), Fvec contains the formulas read upto the error, and the remaining ones are equal to the zero formula.

5.5 Formulaindexvector

The use of this vector is almost the same as Formulavector - see the discussion there.

Inheritance

class Formulaindexvector

Public messages

`Formulaindexvector()`
Use: Formulaindexvector Fvec
Pre: None.
Post: Fvec is a Formulaindexvector of size 0 with minpos 0.

`Formulaindexvector(int, int)`
Use: Formulaindexvector Fvec(sz, $minp$)
Pre: None.
Post: Fvec is a Formulaindexvector with minimum position $minp$ and of size sz, if $sz > 0$, else of size 0.

`resize(int, int, Keeper*)`

Use: Fvec.resize(*sz*, *minp*, *keeper*)
Pre: If Fvec is not empty, $minp \leq$ Fvec.Mincol().
Post: Fvec has enlarged its size by *sz* and put its minimum position to *minp*. The extra positions are added at the end of Fvec, and Fvec informs *keeper* of changed positions if necessary.

`friend CommentStream& operator >>(CommentStream&, Formulaindexvector&)`

Use: *infile* » Fvec
Pre: None.
Post: If *infile*'s format is correct, Fvec has read the formulas from it, and initialized itself. If *infile*'s format is incorrect, *infile*'s badbit has been set. If any switch read is < 0 it has been set to its absoulute value.

`int Size() const`

Use: int $sz =$ Fvec.Size()
Pre: None.
Post: *sz* is equal to the size of Fvec.

`int Mincol() const`

Use: int $minp =$ Fvec.Mincol()
Pre: None.
Post: *minp* is equal to the minimum position of Fvec.

`int Maxcol() const`

Use: int $maxp =$ Fvec.Maxcol()
Pre: None.
Post: *maxp* is equal to the maximum position of Fvec.

`void Inform(Keeper*)`

Use: Fvec.Inform(*keeper*)
Pre: operator» has been called.
Post: Fvec has informed *keeper* of the marked variables in the formulas in Fvec.

`Formula& operator[](int)`

Use: Formula& F = Fvec[i]
Pre: Fvec.Mincol() $\leq i \leq$ Fvec.Maxcol().
Post: F is a reference to the formula Fvec[i].

`const Formula& operator[](int) const`

Use: Formula& F = Fvec[i]

Pre: Fvec.Mincol() $0 \leq i \leq$ Fvec.Maxcol().
Post: F is a reference to the formula Fvec[i].

Protected Characteristics

```
int         minpos    // The lower bound of the indexvector.
int         size      // The number of elements of the vector.
Formula*    v         // The Formulas.
```

Data invariant

- `minpos` ≥ 0.
- `size` ≥ 0.
- `size` $== 0$ if and only if `v` $== 0$.
- The formulas are contained in `v`[0], ..., `v`[`size` - 1].
- If failure occurs in operator» (due to wrong format of *infile*), Fvec contains the formulas read upto the error, and the remaining ones are equal to the zero formula.

5.6 ErrorHandler

5.7 StochasticData

The class StochasticData is very simple. It handles the repeated reading of vectors from a file. This may be used e.g. to read values from a file that are later supplied to Keeper.

Inheritance

class StochasticData

Public messages

`StochasticData(const char* const)`
Use: StochasticData S(*filename*)
Pre: *filename* is a nullterminated string and there exists a file whose name is *filename* and format is correct.
Post: S is ready to read values from its input file, *filename*.
NB: S has to be able to access the *filename* during its lifetime.

`~StochasticData()`

Use: ~S()
Pre: None.
Post: All memory belonging to S has been freed.

`int DataIsLeft()`

Use: l = S.DataIsLeft()
Pre: S.Error() returns 0.
Post: l is 1 if there is any data left for S to read, else l is 0.

`int NoVariables() const`

Use: n = S.NoVariables() const
Pre: S.Error() returns 0.
Post: n keeps the number of variables S reads at a time from its input file.

`int Error() const`

Use: err = S.Error()
Pre: None.
Post: err is 1 if an error has occurred, else err is 0. If err is 1, no operation on S is defined.

`void ReadData(doublevector&)`

Use: S.ReadData(x)
Pre: S.Error() returns 0, S.DataIsLeft() returns 1 and x.Size() = S.NoVariables().
Post: S has put the values it read from its data file into x. S has reexamined its error status.
NB: Since S uses keeps one line in its buffer, the call to this function is guaranteed to succeed if the preconditions are met.

Protected messages

`void ReadDataFromNextLine(doublevector&)`

Use: S.ReadDataFromNextLine(x)
Pre: None.
Post: This function reads the next line from the data file and puts it in x. It may change the error status. However, if there is no data left to read in the input file (indicated by S.EndOfFile() equals 2), x has not been changed. And if an error occurred while reading the data, S.Error() may equal 1 after the function returnes, if it returns at all.

Protected Characteristics

`ifstream`	`infile`	// The input file.
`int`	`error`	// Error status.
`doublevector`	`values`	// For buffered input.
`int`	`EndOfFile`	// End of file status.

Details

The input is buffered, the last line read from infile is kept in values.

There are 3 possible values of EndOfFile:

- 0: end of file has not been reached.
- 1: end of file has been reached, but the values last read have not been returned.
- 2: end of file has been reached and there are no more values in the input buffer to return.

Chapter 6

Data repositories.

6.1 vector<T>

The class vector should be a template, vector<t>. However many of the compilers do have problems with instantiating the templates, especially the older ones, but that is rapidly changing.

Therefore,the approach was taken not to let the vectors be a template, but to use the class tvector instead of vector<t> and to automatically generate the declaration and definition files. And instead of vector<t*>, the corresponding vector is called tptrvector. Hence the classes doublevector, intvector and baseclassptrvector.

Note that the type of the template is written in lower case, i.e. baseclassptrvector instead of vector<BaseClass*>.

The same note applies for the class indexvector<T> $\rightarrow$ tindexvector.

6.2 doublevector

The class doublevector is a member of the vector-family.

Inheritance

class doublevector

Public messages

`doublevector()`

Use: doublevector d()
Pre: None.
Post: d is an empty doublevector.

`doublevector(int)`

Use: doublevector d(sz)

Pre: sz ≥ 0.
Post: d is a doublevector with space allocated for sz elements.

`doublevector(int, double)`
Use: doublevector d(sz, initial)
Pre: sz ≥ 0.
Post: d is a doublevector containing sz elements, initialized to initial.

`doublevector(const doublevector &)`
Use: doublevector d(initial)
Pre: None.
Post: d is a copy of initial, i.e. d.Size() = initial.Size() and d[i] = initial[i].

`~doublevector()`
Use: ~d
Pre: None.
Post: All memory belonging to d has been freed.

$\mathcal{S}$ `void resize (int, double)`
Use: d.resize(add, value)
Pre: add > 0
Post: d's length has been increased of add and the extra elements are initialized to value. The values of the 'old' elements are unchanged.
NB: Although the old values of d have not changed, their addresses may have.

$\mathcal{S}$ `void resize (int)`
Use: d.resize(add)
Pre: add > 0.
Post: Similar to resise(int, double), except there is no initialization guaranteed.
NB: Look at doublevector::resize(int, double).

$\mathcal{S}$ `void Delete(int)`
Use: d.Delete(pos)
Pre: $0 \leq$ pos $<$ d.Size()
Post: The element that was accessible through d[i] has now been deleted from d and its size adjusted accordingly.
NB: **Warning:** A side effect of this member function may be that the position of the elements in memory may be changed.

$\mathcal{F}$ `int Size() const`
Use: s = d.Size()
Pre: None.

Post: s equals the number of elements in d.

$\mathcal{C}$ `double& operator[] (int)`

Use: d[i] = e
Pre: $0 \leq$ i < d.Size()
Post: The number d[i] has been set to e.

$\mathcal{F}$ `const double& operator[] (int) const`

Use: e = d[i]
Pre: $0 \leq$ i < d.Size()
Post: e equals the element d[i].

Protected Characteristics

`double*`	`v`	//
`int`	`size`	//

6.3 doubleindexvector

The class doubleindexvector is just a member of the indexvector-family.

Inheritance

class doubleindexvector

Public messages

`doubleindexvector()`

Use: doubleindexvector dv
Pre: None.
Post: dv is a empty doubleindexvector.

`doubleindexvector(int,int)`

Use: doubleindexvector dv(sz, minpos)
Pre: sz ≥ 0, minpos ≥ 0.
Post: dv is of type doubleindexvector, with space allocated for size elements, indexed from minpos.

`doubleindexvector(int, int, double)`

Use: doubleindexvector dv(sz, minpos, initial)

Pre: sz ≥ 0, minpos ≥ 0.
Post: dv is of type doubleindexvector, with space allocated for size elements, indexed from minpos. The elements are initialized to initial.

`doubleindexvector(const doubleindexvector&)`
Use: doubleindexvector dv(initial)
Pre: None.
Post: dv is a copy of initial

`~doubleindexvector()`
Use: $\tilde{\text{d}}$v()
Pre: None.
Post: All memory belonging to dv has been freed.

$\mathcal{S}$ `void resize(int,int, double)`
Use: dv.resize(addsize,newminpos,value)
Pre: newminpos $\leq$ dv.Mincol(), addsize ≥ 0.
Post: dv has increased its size of addsize, dv.Mincol() equals newminpos and the extra elements are initialized with initial. The elements that dv contained before the call can still be accessed through dv[.], using the same index, but their position in memory may be different.
NB: The documentation here above does not explain what happens if addsize < dv.Mincol() - newminpos is valid before dv.resize is called. That call should not be illegal, but its effects are not quite certain.

$\mathcal{C}$ `double& operator[](int)`
Use: dv[i] = d
Pre: dv.Mincol() $\leq$ i < dv.Maxcol()
Post: dv[i] has been set to d.

$\mathcal{F}$ `const double& operator[](int) const`
Use: d = dv[i]
Pre: dv.Mincol() $\leq$ i < dv.Maxcol()
Post: d equals dv[i]

$\mathcal{F}$ `int Mincol() const`
Use: m = dv.Mincol()
Pre: None.
Post: m equals number of the first element in dv.

$\mathcal{F}$ `int Maxcol() const`

Use: m = dv.Maxcol()
Pre: None.
Post: m equals 1 + the number of the last element in dv.

$\mathcal{F}$ `int Size() const`
Use: sz = dv.Size()
Pre: None.
Post: sz equals the number of elements in dv.
NB: **Warning:** Use this member function with caution, as one might easily forget that dv is of type doubleindexvector and use it as a doublevector if one sees this member function called. The error would then be to let i in dv[i] be in the range of 0, . . . , dv.Size() - 1 instead of dv.Mincol(), . . . , dv.Maxcol() - 1.

$\mathcal{S}$ `void Delete(int)`
Use: dv.Delete(i)
Pre: dv.Mincol() $\leq$ i < dv.Maxcol()
Post: The element that was accessible through dv[i] has now been deleted from dv and its size adjusted accordingly.
NB: **Warning:** A side effect of this member function may be that the position of the elements in memory may be changed.

Protected Characteristics

`int`	`minpos`	//
`int`	`size`	// The number of elements.
`double*`	`v`	// Pointer to array of double.

6.4 bandmatrix

The class bandmatrix is just another class for keeping bandmatrices. It has a lot of constructors, though, and two kinds member functions to access its size, depending on whether one wants to look at it as a regular bandmatrix, or a bandmatrix where the first index is age and the second number of length group.

Inheritance

class bandmatrix

Public messages

`bandmatrix (const bandmatrix&)`

Use: bandmatrix B(*initial*)
Pre: None.
Post: B is a copy of *initial*, i.e. all the sizes of B and values are the same as in *initial*.

`bandmatrix()`

Use: bandmatrix B
Pre: None.
Post: B is an empty bandmatrix.

`bandmatrix(const intvector&, const intvector&, int = 0, double = 0)`

Use: bandmatrix B(*minl*, *size*, *Minage*, *initial*)
Pre: *minl*.Size() == *size*.Size(), *Minage* ≥ 0.
Post: B is a bandmatrix with minimum age equal to *Minage*, minimum length given with *minl* and the length of the rows according to *size*, initialized with *initial*.

`bandmatrix(const doublematrix&, int = 0, int = 0 )`

Use: bandmatrix B(*initial*, *Minage*, *minl*)
Pre: *Minage* ≥ 0, *minl* ≥ 0.

Post: B is a bandmatrix, initialized with *initial*, with the following changes:

- *initial* has been shifted *Minage* rows and *minl* columns, so that B.Minage() equals *Minage*; column 1 in *initial* corresponds to column *minl* in B and the first row in *initial* corresponds to the row *Minage* in B.

- Zeros in the beginning and end of lines in *initial* are cut off, so that the corresponding lines in B are shorter. If a line in *initial* contains only 0, the corresponding line in B may contain 1 element (which is then 0), or it may be an empty line.

`bandmatrix(const doubleindexvector&, int)`

Use: bandmatrix B(*initial*, *age*)
Pre: $age \geq 0$
Post: B is a bandmatrix whose only row is *age* and it is a copy of *initial*.

`bandmatrix(int, int, int, int, double = 0)`

Use: bandmatrix B(*minl*, *size*, *Minage*, *nrow*, *initial*)
Pre: $minl \geq 0$, $size \geq 0$, $Minage \geq 0$, $nrow \geq 0$.
Post: B is a shifted rectangular matrix. I.e. B is a bandmatrix with B.Mincol() equal to *minl*, the lines contain *size* elements, B.Minrow() equals *Minage*, B has *nrow* lines and is initialized with *initial*.

`~bandmatrix()`

Use: ~B
Pre: None.
Post: All memory belonging to B has been freed.

$\mathcal{C}$ `doubleindexvector& operator[](int)`

Use: *dv*= &B[*row*]
Pre: B.Minage() $\leq$ *row* $\leq$ B.Maxage()
Post: *dv*is a reference to row number *row* in B.
NB: This operator is most likely to be used when retrieving whole rows from B, and as an intermediary when accessing a single element of B, ie. B[*row*][*col*], where B.Mincol(*row*) $\leq$ *col* $<$ B.Maxcol(*row*). Then the usage could be B[*row*][*col*] = *num*, where *num*is of type double.

$\mathcal{F}$ `const doubleindexvector& operator[](int) const`

Use: *dv*= &B[*row*]
Pre: B.Minage() $\leq$ *row* $\leq$ B.Maxage()

Post: *dv*is a reference to a constant doubleindexvector, containing row number *row* in B.
NB: This operator is most likely to be used when retrieving whole rows from B, and as an intermediary when accessing a single element of B, ie. B[*row*][*col*], where B.Mincol(*row*) $\leq$ *col* $<$ B.Maxcol(*row*). Then the usage could be *num*= B[*row*][*col*], where *num*is of type double.

$\mathcal{F}$ `int Ncol(int) const`
Use: n = B.Ncol(*row*)
Pre: B.Minrow() $\leq$ *row* $\leq$ B.Maxrow().
Post: n is the number of columns in row *row*.

$\mathcal{F}$ `int Ncol() const`
Use: n = B.Ncol()
Pre: None.
Post: n is the number of columns in row B.Minrow().

$\mathcal{F}$ `int Minrow() const`
Use: m = B.Minrow()
Pre: None.
Post: m is the number of the first row in B.

$\mathcal{F}$ `int Maxrow() const`
Use: m = B.Maxrow()
Pre: None.
Post: m is the number of the last row in B.

$\mathcal{F}$ `int Mincol(int) const`
Use: m = B.Mincol(*row*)
Pre: B.Minrow() $\leq$ *row* $\leq$ B.Maxrow()
Post: m is the number of the first column in the line *row* in B, i.e. m equals B[*row*].Mincol().

$\mathcal{F}$ `int Maxcol(int) const`
Use: m = B.Maxcol(*row*)
Pre: B.Minrow() $\leq$ *row* $\leq$ B.Maxrow()
Post: m equals the length of the line *row* in B plus B.Mincol(*row*), i.e. m equals B[*row*].Maxcol().

$\mathcal{F}$ `int Minage() const`
Use: m = B.Minage()
Pre: None.
Post: m = B.Minrow().

$\mathcal{F}$ `int Maxage() const`

Use: m = B.Maxage()
Pre: None.
Post: m = B.Maxrow().

$\mathcal{F}$ `int Minlength(int) const`

Use: m = B.Minlength(age)
Pre: same as m = B.Mincol(age).
Post: ditto.

$\mathcal{F}$ `int Maxlength(int) const`

Use: m = B.Maxlength(age)
Pre: Same as for m = B.Maxlength(age)
Post: Ditto.

$\mathcal{F}$ `void Colsum(doublevector&) const`

Use: B.Colsum(Result)
Pre: Result.Size() $\geq$ the maximum of B.Maxcol(row).
Post: The sum of each column in B has been added to Result.

Protected Characteristics

```
doubleindexvector** v          // Array of pointers to doubleindexvectors.
int                 nrow       // The number of rows.
int                 minage     // The minimum age.
```

Data invariant

- vwas created with new doubleindexvector*[`nrow`].
- No one of the pointers `v`[i], $i = 0, \ldots,$ `nrow` - 1 equals 0.

6.5 bandmatrixvector

The class bandmatrixvector keeps a vector of matrices. It must be used with some caution because all the elements of the vector do not have to contain a matrix and these elements are illegal to access.

Inheritance

class bandmatrixvector

Public messages

`bandmatrixvector()`

Use: bandmatrixvector B
Pre: None.
Post: B is an empty bandmatrixvector.

`bandmatrixvector(int)`

Use: bandmatrixvector B(*size*)
Pre: $size \geq 0$.
Post: B is of type bandmatrixvector, with space allocated for *size* matrices. However, the matrices themselves are not created, so the user himself has to take care not to use operator[] before they have been created, using ChangeElement(int, const bandmatrix&).

`~bandmatrixvector()`

Use: ~B
Pre: None.
Post: All space belonging to B has been freed, i.e. the vector and all the matrices it contains.

$\mathcal{S}$ `void ChangeElement(int, const bandmatrix&)`

Use: B.ChangeElement(*pos*, *value*)
Pre: $0 \leq pos <$ B.Size()
Post: Matrix no. *pos* has been deleted from B, if there was any and now there is a copy of *value*.

$\mathcal{C}$ `bandmatrix& operator[](int)`

Use: $bm =$ B[*pos*]
Pre: $0 \leq pos <$ B.Size() and a matrix no. *pos* has been created.
Post: *bm* is a reference to matrix no. *pos* in B – most likely usage is though in conjunction with the operator[](int) on bandmatrix and indexvector – see bandmatrix.

$\mathcal{F}$ `const bandmatrix& operator[](int) const`

Use: bm = B[pos]
Pre: $0 \leq pos <$ B.Size() and there is a matrix no. pos.
Post: bm is a reference to matrix no. pos in B – most likely usage is though in conjunction with the operator[](int) on bandmatrix and indexvector – see bandmatrix.

$\mathcal{S}$ `void resize(int, const bandmatrix&)`
Use: B.resize(add, $initial$)
Pre: $add > 0$.
Post: The size of B has been increased off add and the new elements have been filled with copies of $initial$.

$\mathcal{S}$ `void resize(int)`
Use: B.resize(add)
Pre: $add > 0$.
Post: The size of B has been increased of add, and no new matrices are created.
NB: Since there have not been created any new matrices, the user has to take care to create them before trying to access them through the operator[](int).
This member function does not change the matrices in B in any way.

$\mathcal{F}$ `int Size() const`
Use: s = B.Size()
Pre: None
Post: s is equal to the size of B.

$\mathcal{S}$ `void Delete(int)`
Use: B.Delete(pos)
Pre: $0 \leq pos <$ B.Size()
Post: Matrix no. pos has been deleted from B, if one existed. The elements B[i], $0 \leq i \leq pos$ are unchanged, the elements that were accessed through B[$i + 1$] are now in B[i], $pos \leq$ i < B.Size().
NB: This member function does not change the previous matrices in any way.

Protected Characteristics

```
int            size          // The size of the vector.
bandmatrix**   v             // Array of pointers to the matrices.
```

Data invariant

`v` was created with new bandmatrix*[`size`].

6.6 Agebandmatrix

The class Agebandmatrix is designed to keep agelength keys, both numbers and mean weights.

Inheritance

class Agebandmatrix

Public messages

`Agebandmatrix(int, const intvector&, const intvector&)`

Use: Agebandmatrix Ab(*Minage*, *minl*, *size*)

Pre: *Minage* $>$ 0, *minl*.Size() == *size*.Size(), all the elements of *minl* and *size* are $\geq$ 0.

Post: Ab is of type Agebandmatrix, initialized to zero, with the minimum age *Minage*, *minl*.Size() rows and Minlength given with *minl*. The length of the rows is given with *size*.

`Agebandmatrix(int, const popinfomatrix&)`

Use: Agebandmatrix Ab(*Minage*, *initial*)

Pre: *Minage* $\geq$ 0.

Post: Ab is of type Agebandmatrix, with the minimum age *Minage* and initialized with a **reduced** copy of *initial* – see the constructor for bandmatrix for details.

`Agebandmatrix(int, int, const popinfomatrix&)`

Use: Agebandmatrix Ab(*Minage*, *minl*, *initial*)

Pre: *Minage* $\geq$ 0, *minl* $\geq$ 0.

Post: Ab is of type Agebandmatrix, with the minimum age *Minage*, mininum length *minl* and initialized with a **reduced** copy of *initial* – see constructor for bandmatrix for details.

`Agebandmatrix(int, const popinfoindexvector &)`

Use: Agebandmatrix Ab(age, *initial*)

Pre: age $\geq$ 0

Post: Ab is of type Agebandmatrix, with only the row age, initialized to be a copy of *initial*.

`Agebandmatrix(const Agebandmatrix&)`

Use: Agebandmatrix Ab(*initial*)

Pre: None.
Post: Ab is a copy of *initial*.

`Agebandmatrix()`
Use: Agebandmatrix Ab
Pre: None.
Post: Ab is an empty Agebandmatrix.

`~Agebandmatrix()`
Use: ~Ab
Pre: None.
Post: All memory belonging to Ab has been freed.

$\mathcal{F}$ `int Minage() const`
Use: ma = Ab.Minage
Pre: None.
Post: ma has the value of the minimum age in Ab.

$\mathcal{F}$ `int Maxage() const`
Use: ma = Ab.Maxage
Pre: None.
Post: ma has the value of the maximum age in Ab.

$\mathcal{F}$ `int Nrow() const`
Use: nr = Ab.Nrow()
Pre: None.
Post: nr equals the number of rows in Ab.

$\mathcal{C}$ `popinfoindexvector& operator[](int)`
Use: piv = &Ab[age]
Pre: Ab.Minage() $\leq$ age $\leq$ Ab.Maxage()
Post: piv is a reference to the row "*age*" in Ab.
NB: Of course the usage is more likely to be something like: Ab[age][len] = pop, where pop is of type popinfo and Ab.Minlength(age) $\leq$ len < Ab.Maxlength(age)

$\mathcal{F}$ `const popinfoindexvector& operator[](int) const`
Use: piv = &Ab[age]
Pre: Ab.Minage() $\leq$ age $\leq$ Ab.Maxage()
Post: piv is a reference to the (constant) row "*age*" in Ab.
NB: Of course the usage is more likely to be something like: pop = Ab[age][len], where pop is of type popinfo and Ab.Minlength(age) $\leq$ len < Ab.Maxlength(age)

$\mathcal{F}$ `int Minlength(int) const`

Use: l= Ab.Minlength(age)

Pre: Ab.Minage() $\leq$ age $\leq$ Ab.Maxage()

Post: $l$is the minimum length of age age in Ab.

$\mathcal{F}$ `int Maxlength(int) const`

Use: l = Ab.Maxlength(age)

Pre: Ab.Minage() $\leq$ age $\leq$ Ab.Maxage()

Post: $l$is the maximum length of age age in Ab.

$\mathcal{F}$ `void Colsum(popinfovector&) const`

Use: Ab.Colsum(*Result*)

Pre: *Result*.Size() $\geq$ the max_a(Ab.Maxlength(age)).

Post: The sum of the rows in Ab has been added to Result, i.e. $\sum_i$ Ab[i][j] has been added to Result[j].

$\mathcal{S}$ `void Multiply(const doublevector&, const ConversionIndex&)`

Use: Ab.Multiply(*Ratio*, *CI*)

Pre: *CI* contains the mapping from Ab to *Ratio* and *CI*.TargetIsFiner() returns 0. All the elements in *Ratio* are nonnegative (allowing a small numerical error, though).

Post: The number in each length group in Ab has been multiplied with the corresponding element in *Ratio*.

$\mathcal{S}$ `void Subtract(const doublevector&, const ConversionIndex&, const popinfovector&)`

Use: Ab.Subtract(*Consumption*, *CI*, *Nrof*)

Pre: The same as for Multiply(const doublevector&, const ConversionIndex&), and *Consumption*.Size() = *Nrof*.Size().

Post: Ab has been multiplied with the vector containing 1-*Consumption*[i]/*Nrof*[i].N (and 0 if *Nrof*[i].N equals 0).

NB: This is well suited for reducing stock size in **numbers** according to consumption.

$\mathcal{S}$ `void Multiply( const doublevector&)`

Use: Ab.Multiply(*Proportion*)

Pre: All the elements of *Proportion* are >= 0.

Post: For each i such that $0 \leq i$, $i <$ *Proportion*.Size() and $i <$ Ab.Maxage() - Ab.Minage() + 1, the number in the age group i + Ab.Minage() has been multiplied with *Proportion*[i].

$\mathcal{S}$ `void SettoZero()`

Use: Ab.SettoZero()

Pre: None.
Post: Ab has been set to zero.

$\mathcal{S}$ `void IncrementAge()`

Use: Ab.IncrementAge()
Pre: None.
Post: Ab has shifted every age group upwards of one, keeping the oldest one in its place and adding to it the one adjacent one. If some length groups in an age group falls out of length-range when it is shifted upwards, it is ignored.

$\mathcal{S}$ `void Grow(const doublematrix&, const doublematrix&)`

Use: Ab.Grow(*Lgrowth*, *Wgrowth*)
Pre: ??
Post: ??

$\mathcal{S}$ `void Grow(const doublematrix&, const doublematrix&, Maturity* const, const TimeClass* const, const AreaClass* const, const LengthGroupDivision* const, int)`

Use: Ab.Grow(*Lgrowth*, *Wgrowth*, *maturity*, *TimeInfo*, *Area*, *LgrpDiv*, *area*)
Pre: ??
Post: ??

Protected Characteristics

`int`	`minage`	// The minimum age.
`int`	`nrow`	// Number of rows.
`popinfoindexvector**`	`v`	// Array of pointers to popinfoindexvectors.

Data invariant

`v` was created with a call to new popinfoindexvector*[`nrow`]

6.7 Agebandmatrixvector

The classes Agebandmatrixvector and Agebandmatrix play a center role in the simulation. The format of their member functions lay down the lines for many other classes.

Inheritance

class Agebandmatrixvector

Public messages

`Agebandmatrixvector()`

Use: Agebandmatrixvector Av
Pre: None.
Post: Av is an empty Agebandmatrixvector.

`Agebandmatrixvector(int)`

Use: Agebandmatrixvector Av(*size*)
Pre: *size* > 0
Post: Av is of type Agebandmatrixvector, with **room for**, not containing, *size* Agebandmatrices.
NB: Unless further action is taken (using Av.ChangeElement), the elements of Av cannot be accessed.
Stat: Not implemented.

`Agebandmatrixvector(int, int, const intvector&, const intvector&)`

Use: Agebandmatrixvector Av(*size*, *Minage*, *minl*, *len*)
Pre: *size* > 0, *Minage* ≥ 0, the elements of *minl* and *len* are all ≥ 0 and *minl*.Size() == *len*.Size().
Post: Av is of type Agebandmatrixvector, containing *size* Agebandmatrices, each of which has the minimum age *Minage*, minimum length given with *minl* and length of the columns with *len*.

`~Agebandmatrixvector()`

Use: ~Av
Pre: None.
Post: Memory belonging to Av has been freed. Note the meaning of “belonging to” here – see resize(int, Agebandmatrix*).

$\mathcal{S}$ `void ChangeElement(int, const Agebandmatrix&)`

Use: Av.ChangeElement(*nr*, *value*)
Pre: $0 \leq nr <$ Av.Size().
Post: The Agebandmatrix Av[*nr*] has been deleted, if one existed and now Av[*nr*] is a copy of *value*.

$\mathcal{S}$ `void resize(int, Agebandmatrix*)`

Use: Av.resize(*add*, *matr*)

Pre: $add > 0$

Post: Av's size has been increased of *add* and the extra elements are now *matr*.

NB: The extra elements **are** *matr*, so behaviour is undefined if $matr == 0$ and one tries to access the elements where *matr* is.
This member function may cause errors when Av's destructor is called if $add > 1$ and the elements have not been changed, because then *matr* is deleted more than once. It is the user's responsibility to see that this does not happen, either by using this function only with $add == 1$, or by changing some or all of the elements, using Av.ChangeElement(int, const Agebandmatrix&.

$\mathcal{S}$ `void resize(int, int, int, const popinfomatrix&)`

Use: Av.resize(*add*, *Minage*, *minl*, *matr*)

Pre: $add > 0$, $Minage > 0$, $minl \geq 0$.

Post: Av's size has been increased of *add* and the extra elements are now Agebandmatrices which are reduced copies of *matr*, only with the minimum age *Minage* and minimum length *minl*.

$\mathcal{S}$ `void resize(int, int, const intvector&, const intvector&)`

Use: Av.resize(*add*, *Minage*, *minl*, *len*)

Pre: $add > 0$, $Minage \geq 0$, the elements of *minl* and *len* are all ≥ 0 and *minl*.Size() $==$ *len*.Size().

Post: Av's size has been increased of *add* and the extra elements are Agebandmatrices with minimum age *Minage*, minimum length given with *minl* and length of the rows with *len*. The extra Agebandmatrices are all initialized to zero.

$\mathcal{F}$ `int Size() const`

Use: $size =$ Av.Size()

Pre: None.

Post: *size* hold the number of elements in Av.

$\mathcal{C}$ `Agebandmatrix& operator[](int)`

Use: Av[*pos*]

Pre:

Post:

$\mathcal{F}$ `const Agebandmatrix& operator[] (int) const`

Use: Av[*pos*]

Pre:

Post:

$\mathcal{S}$ `void Migrate(const doublematrix&)`

Use: Av.Migrate(*Migrationmatrix*)

Pre: *Migrationmatrix* is a square matrix, whose dimensions equal Av.Size().

Post: Av has migrated using *Migrationmatrix* as a migration matrix.

NB: Descriptions of migration matrices may be found in the file description for migration.

`int`	`size`	// The size of the vector.
`Agebandmatrix**`	`v`	// Array of pointers to Agebandmatrices.

Data invariant

`v` was created with new Agebandmatrix*[`size`].

6.8 BinarySearchTree

The class BinarySearchTree is a very simple implementation of a binary search tree. It does not store duplicates and it does not call destructors for the objects it contains.

It should be implemented as a template; at the moment nodeelem is typedef-ed to be int.

Inheritance

class TreeNode

Public messages

`TreeNode()`

Use: TreeNode T
Pre: None.
Post: T is an empty TreeNode with no left and right siblings.

`TreeNode(nodeelem)`

Use: TreeNode T(x)
Pre: None.
Post: T is of type TreeNode, containing the value x and has no left or right sibling.

`~TreeNode()`

Use: ~T
Pre: None.
Post: The memory belonging to T and its children has been freed. Note that T.value's destructor is not called (which is not needed when nodeelem is typedef-ed to int).

Public Characteristics

`nodeelem`	`value`	// The content of the node.
`TreeNode*`	`left`	// Left child.
`TreeNode*`	`right`	// Right child.

Inheritance

class BinarySearchTree

`BinarySearchTree()`
Use: BinarySearchTree T
Pre: None.
Post: T is an empty binary search tree.

`~BinarySearchTree()`
Use: ~T
Pre: None.
Post: All memory belonging to T has been freed. Destructor were not invoked for the elements of T.

`void Insert(nodeelem)`
Use: T.Insert(x)
Pre: None.
Post: x has been inserted into T, if it was there not already.

`void Delete(nodeelem)`
Use: T.Delete(x)
Pre: None.
Post: If x was in T, x has been deleted from T (without calling x's destructor).

`int IsIn(nodeelem) const`
Use: isin = T.IsIn(x)
Pre: None.
Post: isin equals 1 if x is in T, else isin is 0.

`int IsEmpty() const`
Use: isempty = T.IsEmpty()
Pre: None.
Post: isempty equals 1 if T is empty, else 0.

`nodeelem DeleteSmallest()`
Use: x = T.DeleteSmallest()
Pre: T is not empty.
Post: x is the previous smallest element of T, it has now been deleted from T.

Friend functions

- ostream& operator«(ostream&, const BinarySearchTree&)

Private messages

`TreeNode* Parent(nodeelem) const`

Use: N = T.Parent(n)
Pre: None.
Post: If x is not in the tree or x is in the root, T is 0, else T one of T's siblings contains x.

`nodeelem DeleteSmallestRight(TreeNode*)`

Use: n = T.DeleteSmallestRight(N)
Pre: T $\neq$ 0.
Post: n equals the smallest element in the right child of T.
NB: Pre- and post-conditions are not quite consistent – what if T has no right child?

`void Kill(TreeNode*)`

Use: T.Kill(N)
Pre: N $\neq$ 0.
Post: N and its subtree has been deleted.

`TreeNode* Place(nodeelem, TreeNode*) const`

Use: S = Place(n,N)
Pre: N $\neq$ 0
Post: S is the left or right sibling of N, depending on into which subtree of N x belongs, or S equals N if N->value = x.

Private Characteristics

`TreeNode*`	`root`	// The root of the tree.

Chapter 7

Lengths and conversion.

7.1 LengthGroupDivision

This class handles length group divisions. What is the mininum length corresponding to a length group i? What is the mean length in length group i?

Warning: The ε-error in the representation of real numbers as double may cause some problems in this class, especially in the member function NoLengthGroup(double) of LengthGroupDivision.

Inheritance

class LengthGroupDivision

Public messages

`LengthGroupDivision(double, double, double)`

Use: LengthGroupDivision L(*minlength*, *maxlength*, *dl*)

Pre: $0 \leq minlength < maxlength$, $dl > 0$ and dl divides the difference *maxlength* - *minlength*.

Post: L is a LengthGroupDivision where the mininum length of the first length group is *minlength* and the maximum length of the last length group is *maxlength*. The length groups are of equal length, *dl*.

NB: If *dl*does not divide (maxlength - minlength) the error bit in L is set and no operations on L are defined. Use the Error() member function to access the error status.

`LengthGroupDivision(const LengthGroupDivision&)`

Use: LengthGroupDivision L(*lgrpdiv*)

Pre: None.

Post: L is a copy of *lgrpdiv*.

`LengthGroupDivision(const doublevector&)`

Use: LengthGroupDivision L(vec)

Pre: vec.Size() > 1. The elements in vec are all nonnegative and ordered in a strictly increasing order.

Post: L is a LengthGroupDivision where the mininum length of length group i is $vec[i]$, and maximum length $vec[i+1]$, $0 \leq i <$ vec.Size() - 1. If the preconditions are not met, the error bit is set.

NB: When this constructor is used, it will **not** be noticed if the length groups are of even length.

`~LengthGroupDivision()`

Use: ~L()

Pre: None.

Post: All memory belonging to L has been freed.

$\mathcal{F}$ `double Meanlength(int) const`

Use: $len =$ L.Meanlength(i)

Pre: $0 \leq i <$ L.NoLengthGroups().

Post: len is the mean length in length group i in L.

$\mathcal{F}$ `double Minlength(int) const`

Use: $len =$ L.Minlength(i) $0 \leq i <$ L.NoLengthGroups().

Post: len is the minimum length in length group i in L.

$\mathcal{F}$ `double Maxlength(int) const`

Use: $len =$ L.Maxlength(i)

Pre: $0 \leq i <$ L.NoLengthGroups().

Post: len is the maximum length in length group i in L.

$\mathcal{F}$ `double dl() const`

Use: $len =$ L.dl()

Pre: None.

Post: If L was created with length groups of even length, len is the length of a length group. Else len is 0.

$\mathcal{F}$ `int Size() const`

Use: $n =$ L.Size()

Pre: None.

Post: n equals the number of length groups in L.

$\mathcal{F}$ `int NoLengthGroups() const`

Use: $n =$ L.NoLengthGroups()

Pre: None.

Post: n equals the number of length groups in L.

$\mathcal{F}$ `int NoLengthGroup(double) const`

Use: *no* = L.NoLengthGroup(*length*)

Pre: L.Minlength(0) - ε $\leq$ *length* $\leq$ ε + L.Maxlength(L.NoLengthGroups() - 1).

Post: *no* is the number of the length group such that L.Minlength(*no*) - ε $\leq$ *length* < LMaxlength(*no*) - ε, where ε is a small number, if such a number *no* exists.
If *length* is within an distance of ε from L.Maxlength(L.NoLengthGroups() - 1) *no* equals L.NoLengthGroups() - 1.
If *length* does not saitisfy the preconditions, *no* < 0.

$\mathcal{S}$ `int Combine(const LengthGroupDivision* const)`

Use: *i*=L.Combine(*addition*)

Pre: L and *addition* intersect and are the same in the intersection.

Post: L has been changed such that it contains every length group which is either in L or addition. *i* is 0 if the preconditions weren't fullfilled, else 1.

NB: This message is currently not fully implemented. It is only fully safe for lengthgroupdivisions with dl==0.

$\mathcal{F}$ `int Error() const`

Use: *err* = L.Error()

Pre: None.

Post: *err* is 0 if L is ok, else *err* equals 1 and no operation on L is defined.

Protected Characteristics

`doublevector`	`meanlength`	// keeps mean lengths.
`int`	`size`	// Number of length groups
`double`	`Dl`	// Length of length groups if equal spacing.
`doublevector`	`minlength`	// keeps minimum length.
`int`	`error`	// Error bit.

Details

If the spacing is not equal `minlength`keeps the minimum lengths, else it is empty.

Associated Functions

The error printing functions are useful when creation of LengthGroupDivision failed and there is the need to print out an fatal error message.

`void LengthGroupPrintError(double, double, double, const char*)`

Use: LengthGroupPrintError(*minlength*, *maxlength*, *dl*, *str*)

Pre: *str* is a nullterminated string.

Post: An error message has been written to cerr, that creation of a LengthGroupDivision with minimum length *minlength*, maximum length *maxlength* and interval length *dl* failed and *str* is printed as a string for explanations.

NB: This function is fatal.

`void LengthGroupPrintError(double, double, double, const Keeper* const)`

Use: LengthGroupPrintError(*minlength*, *maxlength*, *dl*, *keeper*)

Pre: $keeper \neq 0$.

Post: Same as for void LengthGroupPrintError(double, double, double, const char*), except that the string for explanations used here is *keeper*→SendString().

NB: This function is fatal.

`void LengthGroupPrintError(const doublevector&, const Keeper* const)`

Use: LengthGroupPrintError(*breaks*, *keeper*)

NB: To be used for printing error messages when length groups are not equally spaced.
This function is fatal.

`void LengthGroupPrintError(const doublevector&, const char*)`

Use: LengthGroupPrintError(*breaks*, *str*)

NB: To be used for printing error messages when length groups are not equally spaced.
This function is fatal.

7.2 ConversionIndex

This class is for assistance when converting from one length group division to another. This is done when adding one stock to another, subtracting catch or consumption from stocks, summing stocks up etc.

Not all of the functions are useful in all cases. Note the preconditions.

Inheritance

class ConversionIndex

`ConversionIndex(const LengthGroupDivision* const, const LengthGroupDivision* const, int = 0)`

Use: ConversionIndex CI(L_1, L_2, *interp*)

Pre: L_1 and L_2 are pointers to LengthGroupDivision. *interp* is 1 if the conversionindex is used for interpolation, else 0. If both $L_1 \rightarrow$dl() and $L_2 \rightarrow$dl() return 0, L_1 is finer than L_2. In all cases L_1 and L_2 have to be comparable, i.e. one has to be finer than or equal to the other.

Post: CI contains the mapping from L_1 to L_2. In the description that follows, L_1 is Lc and L_2 is Lf if CI.TargetIsFiner() equals 1, else L_1 is Lf and L_2 is Lc.

NB: Note that c stands for coarser and f for finer in Lc and Lf.

$\mathcal{F}$ `int TargetIsFiner() const`

Use: $t =$ CI.TargetIsFiner()

Pre: None.

Post: t is 1 if L_1 is stricktly coarser than L_2, else t is 0.

$\mathcal{F}$ `int Pos(int) const`

Use: $P =$ CI.Pos(j)

Pre: $0 \leq j <$ Lf.Size().

Post: P gives the number of the lengthgroup in Lc that lengthgroup j in Lf corresponds to. Precisely it is the number of the lengthgroup in Lc that the center of lengthgroup j in Lf is in.

NB: As a consequence of the preconditions in the constructor, all of the length group j in Lc is contained in a length group in Lc. CI.Pos(j) is the number of that length group.

$\mathcal{F}$ `int Minlength() const`

Use: $M =$ CI.Minlength()

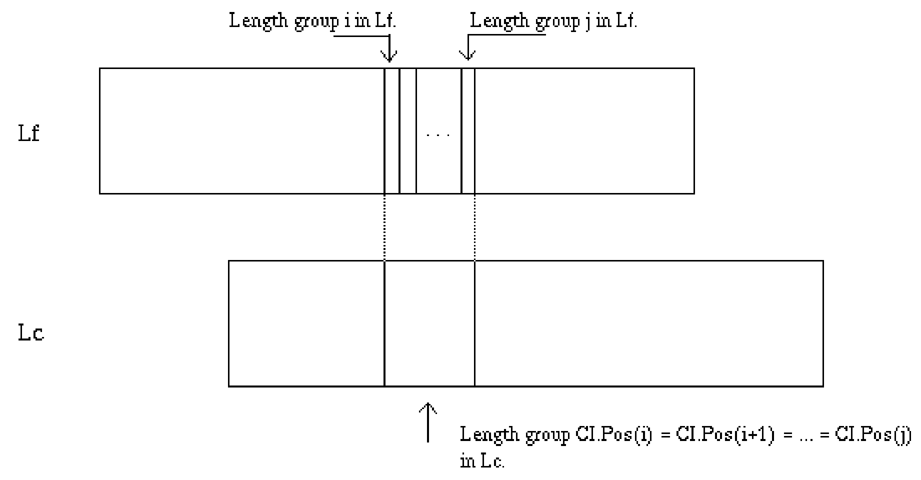
Length group i in Lf.
Length group j in Lf.
Lf
...
Lc
Length group CI.Pos(i) = CI.Pos(i+1) = ... = CI.Pos(j) in Lc.

Figure 7.1: Example of Conversion::Pos.

Pre: none
Post: M contains the smallest lengthgroup in Lf that maps to any lengthgroup in Lc. If any lengthgroups in Lf have lower number than M then these lengthgroups are below the range of Lc.

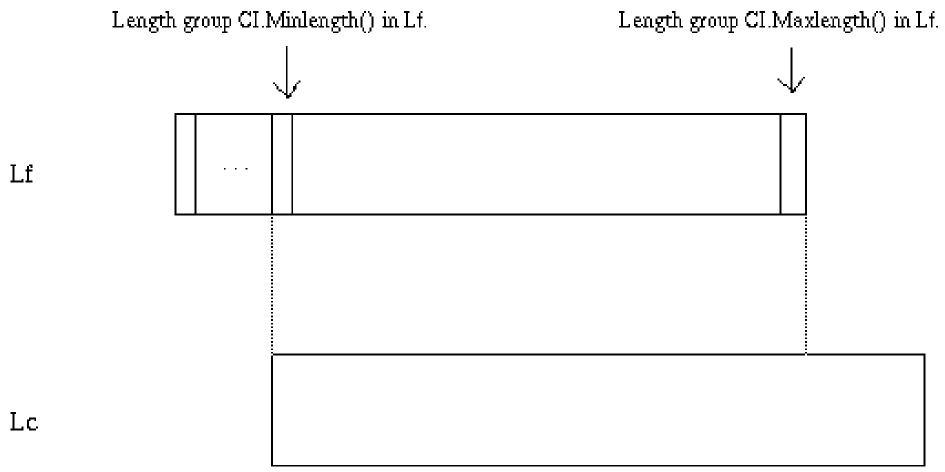
Length group CI.Minlength() in Lf.
Length group CI.Maxlength() in Lf.
Lf
...
Lc

Figure 7.2: Example of Conversion::Minlength and Maxlength.

$\mathcal{F}$ `int Maxlength() const`
Use: M = CI.Maxlength()
Pre: none

Post: M contains the highest lengthgroup in Lf that maps to any lengthgroup in Lc. If any lengthgroups in Lf have higher number than M then these lengthgroups are above the range of Lc.

$\mathcal{F}$ `int Minpos(int) const`

Use: M = CI.Minpos(i)

Pre: $0 \leq i <$ Lc.Size() and CI.SameDl() = 0.

Post: M gives the lowest lengthgroup in Lf that maps to lengthgroup i in Lc.

$\mathcal{F}$ `int Maxpos(int) const`

Use: M = CI.Maxpos(i)

Pre: $0 \leq i <$ Lc.Size() and CI.SameDl() = 0.

Post: M gives the highest lengthgroup in Lf that maps to lengthgroup i in Lf.

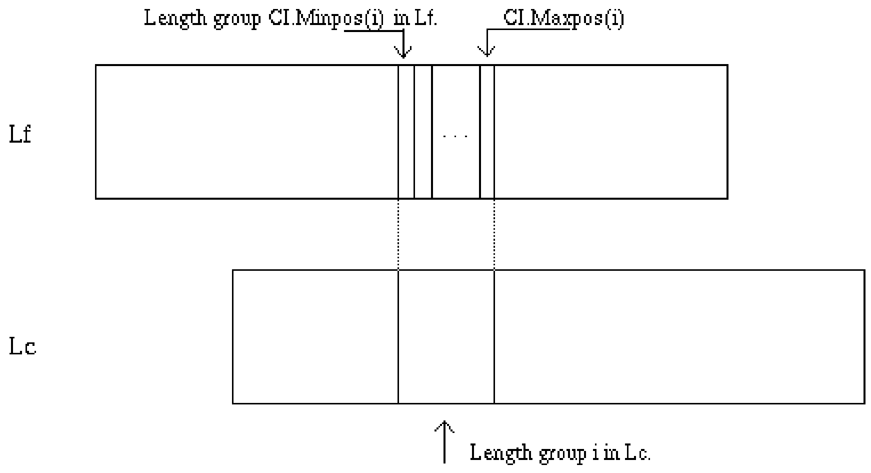

Figure 7.3: Example of Conversion::Minpos and Maxpos.

$\mathcal{F}$ `int Nrof(int) const`

Use: N = CI.Nrof(i)

Pre: $0 \leq i <$ Lf.Size(), CI.TargetIsFiner() = 1 and CI.SameDl() = 0.

Post: N gives the number of lengthgroups in Lf that map to the same lengthgroup in Lc as lengthgroup i in Lf.

NB: Therefore CI.Nrof(i) = CI.Maxpos(CI.Pos(i)) - CI.Minpos(CI.Pos(i)) + 1.

$\mathcal{F}$ `int Offset() const`

Use: O = CI.Offset()

Pre: CI is based on two lengthgroupdivisions with the same dl, not equal to 0 (i.e. CI.SameDl() = 1).
Post: O is an integer such that lengthgroup $i + O$ in L_1 corresponds to lengthgroup i in L_2.

$\mathcal{F}$ `int SameDl() const`
Use: S = CI.SameDl()
Pre: none
Post: S is one if and only if L_1 and L_2 have the same dl, not equal to 0.

$\mathcal{F}$ `int InterpPos(int) const`
Use: P = CI.InterpPos(i)
Pre: $0 \leq i <$ InterpRatio(i), CI was created with *interp* equal to 1.
Post: P equals the smallest length group in Lc such that Lc$\rightarrow$Meanlength(P) $>$ Lf$\rightarrow$Meanlength(i).

$\mathcal{F}$ `double InterpRatio(int) const`
Use: R = CI.InterpRatio(i)
Pre: $0 \leq i <$ Lf.Size(), CI was created with *interp* equal to 1.
Post: R is equal to ??
NB: This part is used by the program Interp to interpolate from Lc to Lf using linear interpolation.

$\mathcal{F}$ `int Nf() const`
Use: N = CI.Nf()
Pre: None.
Post: N contains the number of lengthgroups in Lf.

$\mathcal{F}$ `int Nc() const`
Use: N = CI.Nc()
Pre: none
Post: N contains the number of lengthgroups in Lc.

Protected Characteristics

The protected characteristics have nearly identical names to the public messages so they will not be described here.

```
int             targetisfiner       //
int             samedl              //
int             offset              //
int             nf                  //
int             nc                  //
int            minlength      //
int            maxlength      //
intvector      pos            //
intvector      nrof           //
intvector      minpos         //
intvector      maxpos         //
doublevector   interpratio    //
intvector      interppos      //
```

Associated Functions

The following functions are among those that use objects of type ConversionIndex. Note that some of them require the ConversionIndex to map one direction and not the other, and others do not.

$\mathcal{S}$ `void popinfovector::Sum(const popinfovector* const, const ConversionIndex&)`

Use: p.Sum(*Number*, *CI*)

Pre: *Number* $\neq 0$. *CI* contains the mapping between p and vector *Number* points to and the length group division of **Number* has to be finer or equal to that of p.

Post: *Number* has been added to p.

NB: This function is used in Prey::Sum and Grower::Sum. *Number* is the number in stock and has to be defined with finer or even resolution than the length group division corresponding to **this**

$\mathcal{S}$ `void PopinfoAdd(popinfoindexvector&, const popinfoindexvector&, const ConversionIndex&, double = 1)`

Use: PopinfoAdd(*target*, *addition*, *CI*, *mult*)

Use: PopinfoAdd(*target*, *addition*, *CI*)

Pre: *CI* contains the mapping from *addition* to *target* and *mult* > 0.

Post: The value of *mult***addition* has been added to *target*.

$\mathcal{S}$ `void AgebandmSubtract(Agebandmatrix &, const bandmatrix&, const ConversionIndex&)`

Use: AgebandmSubtract(*Alkeys*, *Catch*, *CI*)

Pre: *CI* contains the mapping from *Catch* to *Alkeys*.

Post: The value of *Catch* has been subtracted from *Alkeys*.

NB: This function is used to subtract catch directly from a stock. The length group division of the catch can be finer, coarser or with even resolution as the length group division corresponding to *Alkeys*. The catch is given as number of fish in the same units as in *Alkeys*.

$\mathcal{S}$ `void AgebandmAdd(Agebandmatrix&, const Agebandmatrix&, const ConversionIndex&, double, int, int)`

Use: void AgebandmAdd(*Alkeys*, *addition*, *CI*, *Ratio*, *minage*, *maxage*)

Pre: *CI* contains the mapping from *addition*to *Alkeys*. *Ratio*≥ 0.

Post: The part of the intersection of *Ratio***addition*with *Alkeys* whose age is between *minage* and *maxage* has been added to *Alkeys*. (As usually, that means *minage* $\leq$ age $<$ *maxage*).

NB: This function is used to add one stock to another. The classes in stock that use this function are Maturity, RenewalData, Initialcond and Transition. The lengthgroupdivision corresponding to *Alkeys* (L_1) can be finer, coarser or with the same resolution as the length group division of *addition*.

$\mathcal{S}$ `void Agebandmatrix::Multiply(const doublevector&, const ConversionIndex&)`

Use: *Alkeys*.Multiply(*Ratio*, *CI*)

Pre: See the documentation of Agebandmatrix.

Post: See the documentation of Agebandmatrix.

NB: *Alkeys* is multiplied by *Ratio*. The length group division corresponding to *Ratio*must have less or even resolution than the length group division corresponding to *Alkeys*. Used by Agebandmatrix::Subtract and AgebandmSubtract.

$\mathcal{S}$ `void Agebandmatrix::Subtract(const doublevector& ,const ConversionIndex&, const popinfovector&)`

Use: *Alkeys*.Subtract(*Consumption*, *CI*, *Nrof*)

Pre: See the documentation of Agebandmatrix.

Post: See the documentation of Agebandmatrix.

NB: *Consumption* has been subtracted from *Alkeys*. The length group division corresponding to *Consumption* (L_2) has to have even or less resolution than the length group division corresponding to `this`(L_1).

$\mathcal{S}$ `void Interp(doublevector&, const doublevector&, const ConversionIndex*)`

Use: Interp(V_f, V_c, *CI*)

Pre: *CI* has been created for interpolation and ???.

Post: ???

$\mathcal{F}$ `void CheckLengthGroupIsFiner(const LengthGroupDivision*, const LengthGroupDivision*, const char*, const char*)`

Use: CheckLengthGroupIsFiner(Lf, Lc, *finername*, *coarsername*)

Pre: Lf $\neq 0$, Lc $\neq 0$ and *finername* and *coarsername* are nullterminated strings, assumed to be descriptive names for the objects that the length group divisions Lf and Lc describe.

Post: If Lf is actually finer than or equal to Lc, the function returns, else it aborts and prints error messages on cerr.

Chapter 8

Predator – prey.

8.1 Overview

The predators introcuded in this section are related as shown in the following picture.

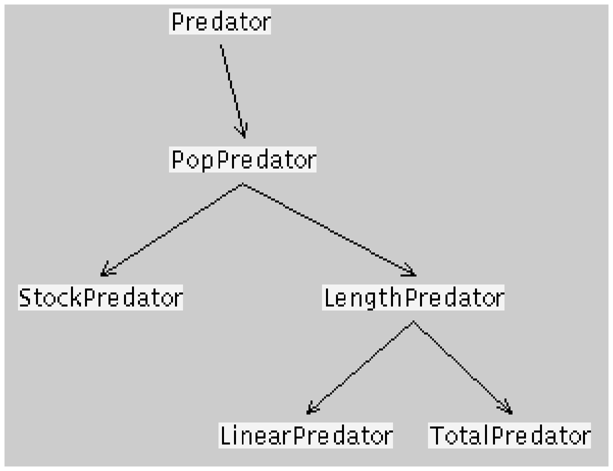

Figure 8.1: Descendants of Predator.

And the preys are related as follows:

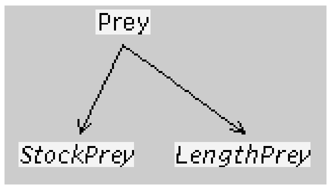

Figure 8.2: Descendants of Prey.

8.2 Suitability

The class Suits keeps suitability data. It allows two forms of suitability data, either a function and corresponding parameters, or a full suitability matrix. The preys for which the suitability is given as a function and parameters are called “function preys” or “FuncPreys” and the other are called “matrix preys” or “MatrixPreys”.

Inheritance

class Suits

Public messages

`Suits()`

Use: Suits S
Pre: None.
Post: S is of type Suitability.

`~Suits()`

Use: ~S
Pre: None.
Post: All memory belonging to S has been freed.

`Suits(const Suits&, Keeper* const)`

Use: S(*initial*, *keeper*)
Pre: *keeper* $\neq 0$.
Post: S is of type Suits. It has copied all the information from *initial* and replaced all the information *keeper* had on *initial* with the corresponding information for S — see Keeper::ChangeVariable(const double&, double&).

$\mathcal{S}$ `void AddPrey(const char*, SuitfuncPtr, const doublevector&, Keeper* const)`

Use: S.AddPrey(*preyname*, *funcptr*, *Parameters*, *keeper*)
Pre: *preyname* is a nullterminated string, *funcptr* is a pointer to a suitability function, *Parameters* is a doublevector, containing parameters for the function *funcptr* points to, *keeper* $\neq 0$.
Post: S has added *preyname* to its list of FuncPreys, and associated with it the pointer *funcptr* and parameters *Parameters*. S has replaced the information *keeper* had on *Parameters* with that of its internal objects — see Keeper::ChangeVariable(const double&, double&).

$\mathcal{S}$ `void AddPrey(const char*, double, const doublematrix&, Keeper* const)`

Use: S.AddPrey(*preyname*, *multiplication*, *suitabilities*, *keeper*)
Pre: *preyname* is a nullterminated string, $multiplication > 0$, *suitabilities* is a nonempty matrix, in which all lines are nonempty, $keeper \neq 0$.
Post: S has added *preyname* to its list of MatrixPreys and associated the suitability you get when multiplying *suitabilities* with *multiplication* to it. S has replaced the information *keeper* had on *multiplication* and *suitabilities* with that of its internal objects.

$\mathcal{S}$ `void DeletePrey(int, Keeper* const)`
Use: S.DeletePrey(*prey*, *keeper*)
Pre: $keeper \neq 0$, $0 \leq prey <$ S.NoPreys().
Post: S has deleted its information on the prey *prey* from itself and *keeper*.

$\mathcal{F}$ `int NoPreys() const`
Use: $no =$ S.NoPreys()
Pre: None.
Post: *no* is the number of preys S has suitabilities for.

$\mathcal{F}$ `const char* Preyname(int) const`
Use: $preyname =$ S.Preyname(p)
Pre: $0 \leq p <$ S.NoPreys().
Post: *preyname* points to a nullterminated string containing the name of S's prey number p.

$\mathcal{S}$ `void Reset(const Predator* const)`
Use: S.Reset(*pred*)
Pre: $pred \neq 0$, *pred*.SetPrey(Preyptrvector&, Keeper*) has been called.
Post: S has calculated the suitability *pred* has for its preys, according to the information S keeps and S accesses from *pred*.

$\mathcal{F}$ `const bandmatrix& Suitable(int) const`
Use: $bm =$ S.Suitable(*prey*)
Pre: $0 \leq prey <$ S.NoPreys() and S.Reset(const Predator*) has been called since the last call to AddPrey() and DeletePrey(int, Keeper* const).
Post: *bm* is a reference to a bandmatrix containing the suitability S keeps for the prey *prey* according to the last calculations in S.Reset(const Predator*). It is indexed with length group in predator and length group in prey.
bm.Minage() returns 0.
NB: The matrix *bm* is on the format of suitability matrices. I.e. no entry in it is less than 0. If any of the elements was read or calculated to be less than 0, it is changed to 0.

Protected messages

$\mathcal{S}$ `void DeleteFuncPrey(int, Keeper* const)`

Use: S.DeleteFuncPrey(*prey*, *keeper*)

Pre: $keeper \neq 0$, $0 \leq prey <$ S.NoFuncPreys().

Post: S has deleted its information on the FuncPrey *prey* from *keeper* and those objects of S that contain only information on the function-preys.

$\mathcal{S}$ `void DeleteMatrixPrey(int, Keeper* const)`

Use: S.DeleteMatrixPrey(*prey*, *keeper*)

Pre: $keeper \neq 0$, $0 \leq prey <$ S.NoMatrixPreys().

Post: S has deleted its information on the MatrixPrey *prey* from *keeper* and those objects of S that contain only information on the matrix-preys.

Protected Characteristics

	`charptrvector`	`FuncPreynames`	// [funcprey] - names of function preys.
	`charptrvector`	`MatrixPreynames`	// [matprey] - names of matrix preys.
κ	`doublematrix`	`FuncParameters`	// [funcprey] - for function preys.
	`SuitfuncPtrvector`	`FunctionPtrs`	// [funcprey] - for function preys.
κ	`doublevector`	`Multiplication`	// [matprey] - for matrix preys.
	`doublematrixptrvector`	`MatrixSuit`	// [matprey][predL][preyl]-for matpreys
	`bandmatrixvector`	`PrecalcSuitability`	// [prey]

Data invariant

- The sizes of `FuncPreynames`, `FuncParameters` and `FunctionPtrs` are equal.
- The elements `FuncParameters`[i] and `FunctionPtrs`[i] correspond to `FuncPreynames`[i].
- The sizes of `MatrixPreynames`, `Multiplication` and `MatrixSuit` are all equal.
- The elements `Multiplication`[i] and `MatrixSuit`[i] correspond to `MatrixPreynames`[i].

Details

`PrecalcSuitability`[i] contains a precalculated suitability matrix for the prey S.Preyname(i).

8.3 Predator

The class Predator is a abstract front end to an object that eats from other classes. Since this class is abstract, the usage shown is only how we recommend the usage to be in derived classes. The same applies for the pre- and post-conditions for the pure virtual functions.
The recommended order of the functions, after the object has been fully initialized is:

- P.Sum(. . .)
- P.Eat(. . .)
- P.AdjustConsumption(. . .)

The thing is that we don't want Sum(. . .) to have the same arguments for all derived classes . . . . You have to check with the derived classes.

Inheritance

class Predator : public HasName, public LivesOnAreas

Public messages

`Predator(const char*, const intvector&)`

Use: Predator P(*givenname*, *areas*)
Pre: *areas* is a nonempty vector whose elements are all unequal and $\geq$ 0, *givenname* points to a nullterminated string.
Post: P has the name *givenname* and is exists on the areas *areas*.
NB: P has not been fully initialized until P.SetPrey(. . .) and P.Reset(. . .) have been called.

`virtual ~Predator()`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.

$\mathcal{S}$ `void SetPrey(Preyptrvector&, Keeper* const)`

Use: P.SetPrey(*preyvec*, *keeper*)
Pre: *keeper* $\neq$ 0, *preyvec* is a nonempty vector of non-null pointers to Prey and contains pointers to all the preys of P.
For every prey of P, *preyvec* has to keep at the most one pointer to a prey with that name.
[Programmers of derived classes must ensure that they have called the protected function SetSuitability(. . .).]

Post: P keeps pointers to its preys. If P had a prey that was not in *preyvec*, a warning was given and information on that prey has been deleted [For programmers' information: using the virtual function DeleteParametersForPrey(...)].
preyvec has not been changed.
P has calculated the suitability for each of the preys found in *preyvec* and resized its objects [For programmers' information: using the virtual function ResizeObjects(...)].
NB: It is the users responsibility to guarantee enough lifetime of the objects pointed to in *preyvec*. It has to be at least equal to that of the last call to P.

$\mathcal{F}$ `int DoesEat(const char*) const`
Use: *eats* = P.DoesEat(*preyname*)
Pre: *preyname* points to a nullterminated string.
Post: *eats* is 1 if P has a prey whose name is *preyname*, else *eats* is 0.
NB: If this function is called before P.SetPrey(...) has been called, we only know that that P wants to eat *preyname* if *eats* is 1. If P.SetPrey(...) has been called and eats equals 1, we know that P wants to eat *preyname* and that it exists.

`virtual void Eat(int, double, double) = 0`
Use: P.Eat(*area*, *Temperature*, *AreaSize*)
Pre: P is defined on the area *area*, P.Reset(...) has been called and P.Sum(...) has been called for the area *area*. *Temperature* is the temperature on that area and *AreaSize* denotes its size.
Post: P has calculated what it wants to eat on the area and informed its preys of it through appropriate member functions.

`virtual void AdjustConsumption(int) = 0`
Use: P.AdjustConsumption(*area*)
Pre: P lives on the area *area*, P.Eat(...) has been called for the area *area*.
Post: P has checked if it was allowed to eat what it wanted of its preys on the area *area*, if not, it has made some adjustments.

`virtual void Print(ofstream&) const`
Use: P.Print(*outfile*)
Pre: *outfile* has no badbits set and P.Reset(...) has been called.
Post: P has written internal information to *outfile*.

`virtual const bandmatrix& Consumption(int, const char*) const = 0`
Use: *bm* = P.Consumption(*area*, *preyname*)

Pre: P is defined on the area *area* and *preyname* is the name of one of P's preys. P.AdjustConsumption(...) has been called since last call to P.Eat(...) and P.Sum(...) for the area *area*.
Post: *bm* is a reference to a bandmatrix indexed by [(undecided predator index)] [(undecided prey index)], containing the amount eaten in **biomass** units by (undecided predator index) of (undecided prey index) in the prey *preyname* on area *area*.
bm.Minage() returns 0.

`virtual const doublevector& Consumption(int) const = 0`
Use: *vec* = P.Consumption(*area*)
Pre: P lives on the area *area* and P.AdjustConsumption(...) has been called since last call to P.Eat(...) and P.Sum(...) for the area *area*.
Post: *vec* is a reference to a vector indexed by (undecided), containing the total amount they ate in the area *area* in **biomass** units.

`virtual const doublevector& OverConsumption(int) const = 0`
Use: *vec* = P.OverConsumption(*area*)
Pre: P lives on the area *area* and P.AdjustConsumption(...) has been called since last call to P.Eat(...) and P.Sum(...) for the area *area*.
Post: *vec* is a reference to a vector indexed by (undecided), containing the overconsumption of P in area *area* in **biomass** units.

`virtual const LengthGroupDivision* ReturnLengthGroupDiv() const = 0`
Use: *lgrpdiv* = P.ReturnLengthGroupDiv()
Pre: None.
Post: *lgrpdiv* points to the LengthGroupDivision for P.

`virtual int NoLengthGroups() = 0`
Use: *no* = P.NoLengthGroups()
Pre: None.
Post: *no* keeps the number of length groups in P.

`virtual double Length(int) const = 0`
Use: *l* = P.Length(*length*)
Pre: *length* is a length group in P, i.e. $0 \leq length <$ P.NoLengthGroups().
Post: *l* contains the mean length in length group *length*.

$\mathcal{S}$ `virtual void Reset()`
Use: P.Reset()
Pre: P.SetPrey(...) has been called.

Post: P has (re)calculated its suitability for the preys. [For programmers' information: This has effects on the return value of Suitability(...)].

Data invariant

In the protected access messages, all matrices and vectors are ordered the same way as P.Preyname(...), i.e. in every vector (and matrix) returned from the access functions, the i-th element contains information for the prey whose name is P.Preyname(i).
The description above applies after P.SetPrey(...) has been called.

Protected messages

$\mathcal{S}$ `virtual void DeleteParametersForPrey(int, Keeper* const)`

Use: P.DeleteParametersForPrey(*prey*, *keeper*)
Pre: $0 \leq prey <$ P.NoPreys(), $keeper \neq 0$.
Post: P has deleted the information it had on the prey whose name is P.Preyname(*prey*) and informed the *keeper* of it. The elements of P.Preyname(...) which come after *prey* will be shifted down one place.
Of course, this function keeps the data invariant concerning Preyname(...).
NB: This function is called in SetPrey(...). The programmer of a derived class of this one may want to supply it with a new instance of this function; one that calls this one and adds its own code in order to make its keep its data invariants.
Note that this function is called **before** ResizeObjects(...) is called. I.e. this function resizes objects whose size is set in the constructor.

$\mathcal{S}$ `virtual void ResizeObjects()`

Use: P.ResizeObjects()
Pre: P.SetPrey has been called.
Post: P has resized its internal objects.
NB: This function is called at the end of SetPrey(...). The programmer of a derived class may want to do the same with this function as DeleteParametersForPrey(...), that is supply the class with a replacement of this method, one that calls this function and adds its own code.

$\mathcal{S}$ `void SetSuitability(const Suitability* const, Keeper* const)`

Use: P.SetSuitability(S, *keeper*)
Pre: $S \neq 0$, *keeper* $\neq 0$, P.SetSuitability(...) has not been called before.
Post: P has set its preys and the corresponding suitability to be that of S. The information *keeper* had on S has now been replaced with that of an private object of P.

$\mathcal{F}$ `int NoPreys() const`
Use: sz = P.NoPreys()
Pre: P.SetPrey(...) has been called.
Post: sz contains the number of P's preys.

$\mathcal{F}$ `const char* Preyname(int) const`
Use: *preyname* = P.Preyname(i)
Pre: $0 \leq i <$ P.NoPreys()
Post: *preyname*points to a nullterminated string, containing the name of P's i-th prey.

$\mathcal{F}$ `Prey* Preys(int) const`
Use: *preyptr* = P.Preys(i)
Pre: P.SetPrey(...) has been called and $0 \leq i <$ P.NoPreys().
Post: *preyptr* points to the prey whose name is P.Preynames(i).
NB: In this function, there is a const cast away. Any one who can come up with a better solution is encouraged to do so.
Even though *preyptr* is not of type const Prey*, it does not mean that a programmer can delete it. In fact he is strictly discouraged from doing so; this const cast away is only made to allow the programmer to call non-const member functions of Prey within const functions of Predator, not to allow him to delete the pointers at wish.

$\mathcal{F}$ `const bandmatrix& Suitability(int) const`
Use: bm = P.Suitability(i)
Pre: P.Reset(...) has been called.
Post: bm is a reference to a bandmatrix. That matrix is indexed with length group of predator and length group of prey. The name of the prey, whose matrix of suitabilities is bm is given with P.Preyname(i).
The matrix bm is not necessarily a square one. If the beginning or end of lines in it are missing, it is because the calculated suitability was ≤ 0.

Friend classes

class Suits.
The class Suits has to access the protected member functions of Predator, NoPreys and Preys to be able to calculate the suitabilities.

Private Characteristics

`Preyptrvector`	`preys`	// Pointers to the preys.
`Suits*`	`Suitable`	// Keeps suitability information.

8.4 PopPredator

The class PopPredator is an abstract base class – it is a bit closer to an actual class than Predator, but not quite enough to be instantiated.
The difference between PopPredator and Predator is mainly that in PopPredator it has been decided how to store the abundance numbers, the consumption and information concerning length.
This documentation serves two purposes: to explain those member functions that are not redefined in a derived class, so that a user of the derived class can access it somewhere, and this document has also to be of some assistance to the programmer of derived classes. This can readily be seen from notes in the class descriptions.

Inheritance

class PopPredator : `public` Predator

Public messages

`PopPredator(const char*, const intvector&, const LengthGroupDivision* const, const LengthGroupDivision* const)`

Use: PopPredator P(*givenname*, *areas*, *OtherLgrpDiv*, *GivenLgrpDiv*)
Pre: *areas* is a nonempty vector whose elements are all unequal and ≥ 0, *givenname* points to a nullterminated string. *OtherLgrpDiv* $\neq 0$, *GivenLgrpDiv* $\neq 0$. The lenght group division *OtherLgrpDiv* points to has to have equal spacing (i.e. *OtherLgrpDiv*$\rightarrow$dl() $\neq 0$) and *GivenLgrpDiv* has to be coarser than or equal to *OtherLgrpDiv*.
Post: P has the name *givenname* and is exists on the areas *areas*. The length group division of P is given with *GivenLgrpDiv* and P expects to receive abundance numbers according to *OtherLgrpDiv*.
For programmers' information: The conversion index has been set to convert from *OtherLgrpDiv* to *GivenLgrpDiv*.
NB: P has not been fully initialized until P.Reset() has been called.

`virtual ~PopPredator()`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.

`virtual void Print(ofstream&) const`

Use: P.Print(*outfile*)
Pre: *outfile* has no badbits set and P.Reset() has been called.
Post: P has written internal information to *outfile*.

$\mathcal{C}$ `virtual const bandmatrix& Consumption(int, const char*) const`

Use: *bm* = P.Consumption(*area*, *preyname*)

Pre: P is defined on the area *area* and *preyname* is the name of one of P's preys. P.AdjustConsumption(...) has been called since last call to P.Eat(...) and P.Sum(...) for the area *area*.

Post: *bm* is a reference to a bandmatrix indexed by [predLenghtGroup][PreyLengthGroup], containing the amount eaten by length groups in P of lengths groups in the prey *preyname* on area *area* in **biomass** units.

$\mathcal{C}$ `virtual const doublevector& Consumption(int) const`

Use: *vec* = P.Consumption(*area*)

Pre: P lives on the area *area* and P.AdjustConsumption(...) has been called since last call to P.Eat(...) and P.Sum(...) for area *area*.

Post: *vec* is a reference to a vector indexed by length groups in P, containing the total amount they ate in the area *area* in **biomass** units.

$\mathcal{C}$ `virtual const doublevector& OverConsumption(int) const`

Use: *vec* = P.OverConsumption(*area*)

Pre: P lives on the area *area* and P.AdjustConsumption(...) has been called for area *area* since last call to P.Eat(...) and P.Sum(...) for area *area*.

Post: *vec* is a reference to a vector indexed by length groups in P, containing the overconsumption of P in area *area* in **biomass** units.

$\mathcal{F}$ `virtual const LengthGroupDivision* ReturnLengthGroupDiv() const`

Use: *lgrpdiv* = P.ReturnLengthGroupDiv()

Pre: None.

Post: *lgrpdiv* points to a LengthGroupDivision describing P.

$\mathcal{F}$ `virtual int NoLengthGroups()`

Use: *no* = P.NoLengthGroups()

Pre: None.

Post: *no* is equal to the number of length groups in P.

$\mathcal{F}$ `virtual double Length(int) const`

Use: *l* = P.Length(*length*)

Pre: *length* is a length group in P, i.e. $0 \leq length <$ P.NoLengthGroups().

Post: *l* is equal to the mean length in length group *length*.

$\mathcal{S}$ `virtual void Reset()`

Use: P.Reset()

Pre: P.SetPrey(...) has been called.

Post: P has called Predator::Reset().
For programmers' information: P has changed consumption, so that its size is the same as of P.Suitabilities(...).

Protected messages

The protected functions DeleteParametersForPrey(...) and ResizeObjects(...) are mere additions to these same functions in Predator.

$\mathcal{S}$ `virtual void DeleteParametersForPrey(int, Keeper* const)`
Use: P.DeleteParametersForPrey(*prey*, *keeper*)
Pre: $0 \leq prey <$ P.PreySize(), $keeper \neq 0$.
Post: See Predator::DeleteParametersForPrey(...). This function calls it, and also updates the protected variables declared in PopPredator.

$\mathcal{S}$ `virtual void ResizeObjects()`
Use: P.ResizeObjects()
Pre: P.SetPrey(...) has been called.
Post: See Predator::ResizeObjects(). This function calls it and also updates the protected variables declared in PopPredator.

Protected Characteristics

```
LengthGroupDivision*LgrpDiv             //
ConversionIndex*    CI                  // Conv. betw. int. and ext. lengths.
popinfomatrix       Prednumber          // Abundance numbers – [area][predLengthgroup].
doublematrix        overconsumption     // – [area][predLengthgroup].
bandmatrixmatrix    consumption         // – [area][prey][predLengthgroup][preylengthgr.]
doublematrix        totalconsumption    // – [area][predLengthgroup].
```

Details

In the indices above, the maximum range is (inclusive):

area	0	areas.Size() - 1
predLengthGroup	0	`LgrpDiv`→NoLengthGroups() - 1
prey	0	Preynames().Size() - 1
preyLengthGroup	0	Preys(prey)→NoLengthGroups() - 1

However, the actual range can be less (in bandmatrix).

8.5 StockPredator

The class StockPredator can be instantiated. It implements eating and has an access function for the feeding level.

Inheritance

class StockPredator : `public` PopPredator

Public messages

`StockPredator(CommentStream&, const intvector&, const char*, const LengthGroupDivision* const, const LengthGroupDivision* const, int, int, Keeper* const)`

Use: StockPredator P(*infile*, *areas*, *givenname*, *OtherLgrpDiv*, *GivenLgrpDiv*, *minage*, *maxage*, *keeper*)
Pre: *infile* has no badbits set and its format is correct and according to *area*s and *GivenLgrpDiv*. *keeper* $\neq 0$. *minage* $\leq$ *maxage*. Refer also to the preconditions for the constructor of PopPredator.
Post: P is of type StockPredator and lives on the areas *areas*[*i*] and has the name *givenname*. The length group division of P is given with *GivenLgrpDiv* and P expects to receive abundance numbers according to *OtherLgrpDiv*, using *minage* as its minimum age group and *maxage* as the maximum age group.
NB: P has not been fully initialized until P.SetPrey(...) has been called.

`virtual ~StockPredator()`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.

$\mathcal{AG}$ `virtual void Sum(const Agebandmatrix&, int)`

Use: P.Sum(*Alkeys*, *area*)
Pre: *Alkeys* is according to the *OtherLgrpDiv* received in the constructor and P is defined on the area *area*.
Post: P has set its abundance numbers on the area *area* according to *Alkeys*, using the part of *Alkeys* whose age is between *minage* and *maxage*, inclusive.

$\mathcal{AG}$ `virtual void Eat(int, double, double)`

Use: P.Eat(*area*, *Temperature*, *AreaSize*)
Pre: P.SetPrey(...) has been called, P is defined on the area *area*, *Temperature* is the temperature on that area and *AreaSize* its size.
Post: P has calculated what it wants to eat on the area and called AddConsumption(...) in its preys.

$\mathcal{AG}$ `void AdjustConsumption(int)`

Use: P.AdjustConsumption(*area*)
Pre: P lives on the area *area*, P.Eat(...) has been called for the area *area*.
Post: P has checked if it was allowed to eat what it wanted of its preys on area *area*, if not, it has made some adjustments.

`virtual void Print(ofstream&) const`

Use: P.Print(*outfile*)
Pre: *outfile* has no badbits set and P.SetPrey(...) has been called.
Post: P has written internal information to *outfile*.

$\mathcal{C}$ `const bandmatrix& Alproportion(int) const`

Use: bm = P.Alproportion(*area*)
Pre: P is in area *area* and P.Eat(...) has been called on the area *area*.
Post: $bm[a][l]$ contains the age-length group (a,l)'s proportion in the predation of the length group l on the area *area*, i.e.

$$\frac{bm[a][l] = biomasspredatedby(a,l)}{biomasspredatedby(l).}$$

$\mathcal{C}$ `const doublevector& FPhi(int) const`

Use: vec = P.FPhi(*area*)
Pre: P is in area *area* and P.AdjustConsumption(...) has been called for the area *area*.
Post: vec is a reference to a doublevector of length P.NoLenghtGroups() containing the feeding level on area *area*(see eq. 10 of Bogstad et.al (1992)).

$\mathcal{C}$ `const doublevector& MaxConByLength(int) const`

Use: vec = P.MaxConByLength(*area*)
Pre: P is in area *area* and P.AdjustConsumption(...) has been called for the area *area*.
Post: vec is a reference to a doublevector of length P.NoLengthGroups() containing P's maximum consumption on the area *area*.
NB: **Warning:** ???

Protected messages

$\mathcal{S}$ `virtual void ResizeObjects()`

Use: P.ResizeObjects()
Pre:

Post: P has called PopPredator::ResizeObjects(...) and adjusted the size of its own protected variables.

$\mathcal{S}$ `virtual void CalcMaximumConsumption(double, int)`

Use: P.CalcMaximumConsumption(T, *area*)

Pre: P is in the area *area* and T is the temperature on it.

Post: P has calculated the maximum consumption on the area *area* and set Alproportion.

NB: Currently, this function is called in the beginning of the member function Eat(...).

Protected Characteristics

κ	`doublevector`	`Maxconsumption`	// Parameters for function.
κ	`double`	`HalfFeedingValue`	// Feeding level half value (mass units/km^2).
	`doublematrix`	`Phi`	// – [area][predLengthgroup].
	`doublematrix`	`fphi`	// Feeding level – [area][predLengthgroup].
	`bandmatrixvector`	`Alprop`	// [area][age][length group]
	`doublematrix`	`MaxconByLength`	// [area][length group]
	`Agebandmatrixvector`	`Alkeys`	// [area][age][length group]

8.6 LengthPredator

The class LengthPredator is for those predators that are only length based.

Inheritance

class LengthPredator : `public` PopPredator

Public messages

`LengthPredator(const char*, const intvector&, const LengthGroupDivision* const, const LengthGroupDivision* const)`

Use: LengthPredator P(*givenname*, *areas*, *OtherLgrpDiv*, *GivenLgrpDiv*)
Pre: *givenname* is a nullterminated string, *areas* is a nonempty vector containing unique nonnegative integers, $OtherLgrpDiv \neq 0$ and $GivenLgrpDiv \neq 0$.
Post: P is a LengthPredator with the name *givenname* that lives on the areas *areas*. Its length group division is given with *GivenLgrpDiv* and is expects to communicate through the length group division *OtherLgrpDiv*.

`virtual ~LengthPredator()`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.

$\mathcal{AG}$ `virtual void Sum(const popinfovector&, int)`

Use: P.Sum(*NumberInArea*, *area*)
Pre: *NumberInArea* is according to the *OtherLgrpDiv* received in the constructor and P is defined on the area *area*.
Post: P has set its abundance numbers on the area *area* according to *NumberInArea*.

$\mathcal{F}$ `double Scaler(int) const`

Use: s = P.Scaler(*area*)
Pre: None.
Post: s is equal to the scaler used to scale **??????**

Protected Characteristics

```
doublevector       scaler                   //
```

8.7 TotalPredator

The class TotalPredator tries to let the total amount eaten on each area be equal to the biomass received in Sum, distributed on preys and length groups according to the suitability functions.

Inheritance

class TotalPredator : `public` LengthPredator

Public messages

`TotalPredator(CommentStream&, const char*, const intvector&, const LengthGroupDivision* const, const LengthGroupDivision* const, Keeper* const, const char*)`

Use: TotalPredator P(*infile*, *givenname*, *areas*, *OtherLgrpDiv*, *GivenLgrpDiv*, *keeper*, *finalstring*)

Pre: *infile* has no badbits set and its file format is correct. *givenname* is a nullterminated string, *areas* is not a nullvector and its elements are distinct and ≥ 0, *OtherLgrpDiv* $\neq 0$, *GivenLgrpDiv* $\neq 0$, *keeper* $\neq 0$, *finalstring* is a nullterminated String. See also the documentation of PopPredator's constructor.

Post: P has read its information from *infile*. P read until the end of *infile*, or until the string *finalstring* was read from *infile*, in which case *infile* is positioned after the occurence of *finalstring*.
P has set its length group division to be that of *GivenLgrpDiv* and expects that *OtherLgrpDiv* describes the length group division of the objects it receives in the subsequent calls to P.Sum.
P lives on the areas *areas*[i], $i = 0, \ldots,$ *areas*.Size() - 1.

NB: Beware: In calls to most of the member functions of P where area is a parameter, P expects that it is one of the areas on which P lives.

`virtual ~TotalPredator()`

Use: ~P

Pre: None.

Post: All memory belonging to P has been freed.

$\mathcal{AG}$ `virtual void Eat(int, double, double, double)`

Use: P.Eat(*area*, *LengthOfStep*, *Temperature*, *AreaSize*)

Pre: See the description in the base classes. In addition, *LengthOfStep* is equal to the length of the current time step.

Post: See the description in the base classes.

$\mathcal{AG}$ `virtual void AdjustConsumption(int)`

Use: P.AdjustConsumption(*area*)
Pre: See the description in the base classes.
Post: See the description in the base classes.

`virtual void Print(ofstream&) const`

Use: P.Print(*outfile*)
Pre: *outfile* has no badbits set and P.SetPrey(...) has been called.
Post: P has written internal information to *outfile*.

8.8 LinearPredator

The class LinearPredator is almost identical to TotalPredator, except for a slightly different call to a constructor and that the amount each length group of the LinearPredator eats of its preys is proportional to the biomass of that length group received through the member function Sum.

Inheritance

class TotalPredator : `public` LengthPredator

Public messages

`LinearPredator(CommentStream&, const char*, const intvector&, const LengthGroupDivision* const, const LengthGroupDivision* const, Keeper* const, const char*, int = 0)`

Use: LinearPredator P(*infile*, *givenname*, *areas*, *OtherLgrpDiv*, *GivenLgrpDiv*, *keeper*, *finalstring*, *ShallReadMultiplicative*)
Pre: See TotalPredator; *ShallReadMultiplicative* is either 0 or 1.
Post: See TotalPredator. If *ShallReadMultiplicative* is ommitted or has the value 0, the postconditions are identical to those of TotalPredator.
If *ShallReadMultiplicative* was 1, TotalPredator finishes its reading from *infile* when a multiplicative constant was read and the value of *finalstring* was ommitted.

`virtual ~LinearPredator()`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.

`virtual void Eat(int, double, double, double)`

Use: P.Eat(*area*, *LengthOfStep*, *Temperature*, *AreaSize*)
Pre: See the description in the base classes. *LengthOfStep* is equal to the length of the current time step.
Post: See the description in the base classes.

`virtual void AdjustConsumption(int)`

Use: P.AdjustConsumption(*area*)
Pre: See the description in the base classes.
Post: See the description in the base classes.

`virtual void Print(ofstream&) const`

Use: P.Print(*outfile*)
Pre: *outfile* has no badbits set and P.SetPrey has been called.
Post: P has written internal information to *outfile*.

Protected Characteristics

κ	`double`	`Multiplicative`	//

8.9 Prey

The class Prey is a abstract front end to some other object. Derived classes should inform it of abundance through a member function and then it can be eaten through the member function AddConsumption(...).
Derived classes need to provide a way of informing Prey of its abundance, in the documentation below that is referred to as the member function P.Sum(...). The functionality of that member function is described below.

Inheritance

class Prey : public HasName, public LivesOnAreas

Public messages

`Prey(CommentStream&, intvector, const char*, Keeper* const)`

Use: Prey P(*infile*, *areas*, *givenname*, *keeper*)
Pre: *infile* has no badbits set, its format is correct and according to *areas*, which is a nonempty intvector whose elements are ≥ 0 and unique. *givenname* is a nullterminated string and *keeper* $\neq 0$.
Post: P is of type Prey and has read its information from *infile*. It is defined on the areas *areas*[i], $i = 0, \ldots,$ *areas*.Size() - 1 and its name is *givenname*.
NB: The initialization of P is not complete until P.SetCI(...) has been called.

`Prey(const doublevector&, const intvector&, const char*)`

Use: Prey P(*lengths*, *areas*, *givenname*)
Pre: *lengths*.Size() > 0, the elements of *lengths* are unique and in ascending order. *givenname* is a nullterminated string and the elements of areas are unique and ≥ 0. *areas*.Size() > 0.
Post: P is of type Prey whose length group division has its endpoints in *lengths*[i], $i = 0, \ldots,$ *lengths*.Size() - 1.
NB: The initialization of P is not complete until P.SetCI(...) has been called.

`virtual ~Prey() = 0`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.
NB: The destructor is declared pure virtual in order to make Prey an abstract class. The destructor is still provided.

$\mathcal{AG}$ `void Subtract(Agebandmatrix&, int)`

Use: P.Subtract(*Alkeys*, *area*)

Pre: P is in area *area*, P.SetCI(...) has been called and *Alkeys* is according to the LengthGroupDivision received there.

Post: P has subtracted the amount eaten of P on area *area* from *Alkeys*. The mean weights in *Alkeys* are unchanged.

$\mathcal{AG}$ `void AddConsumption(int, const doubleindexvector &)`

Use: P.AddConsumption(*area*, *predconsumption*)

Pre: P.SetCI(...) has been called, *predconsumption* is consistent with the length group division of P.

Post: P has added *predconsumption* to the consumption of P on area *area*, where *predconsumption* is taken to be in **biomass** units.

$\mathcal{S}$ `void SetCI(const LengthGroupDivision* const)`

Use: P.SetCI(*OtherLgrpDiv*)

Pre: P.SetCI(...) has not been called before, *OtherLgrpDiv* $\neq 0$, *OtherLgrpDiv*$\rightarrow$dl() $\neq 0$ and *OtherLgrpDiv* is finer than or equal to P's length group division.

Post: P is fully initialized. When receiving information on numbers or when subtracting amount eaten, P will assume that the objects received are divided into length groups according to *OtherLgrpDiv*.

`virtual void Print(ofstream&) const`

Use: P.Print(*outfile*)

Pre: P.SetCI(...) has been called.

Post: P has written internal information to *outfile*.

$\mathcal{F}$ `double Biomass(int, int) const`

Use: b = P.Biomass(*area*, *length*)

Pre: P is defined on the area *area*, length is a length group of P and P.Sum(...) has been called for the area *area*.

Post: b contains the biomass of length group *length* in P on area *area*.

$\mathcal{F}$ `double Biomass(int) const`

Use: b = P.Biomass(*area*)

Pre: P is defined on area *area* and P.Sum(...) has been called for area *area*.

Post: b contains the biomass of P on the area *area*.

$\mathcal{F}$ `int TooMuchConsumption(int) const`

Use: t = P.TooMuchConsumption(*area*)

Pre: P is defined on the area *area* and P.CheckConsumption(...) has been called for area *area*.

Post: t is 1 if the consumption of P in area *area* exceeded the number of P in that area in any length group.

$\mathcal{AG}$ `void CheckConsumption(int)`

Use: P.CheckConsumption(*area*)

Pre: P is defined on the area *area* and P.Sum(…) has been called for area *area*.

Post: P has checked whether the consumption of P, received through P.AddConsumption(…) since last call of P.Sum(…) for area *area*, exceeds the population of P on area *area*.

$\mathcal{F}$ `double Ratio(int, int) const`

Use: r = P.Ratio(*area*, *length*)

Pre: P is defined on area *area*, P.CheckConsumption(…) has been called for area *area*since last calls to P.AddConsumption(…) and P.Sum(…) for area *area*, and *length* is a length group in P.

Post: r equals the ratio between the consumption that P received through AddConsumption(…) since last call to P.Sum(…) and the population of P, received through last call of P.Sum(…), on area *area* and in length group *length*.

$\mathcal{F}$ `double Length(int) const`

Use: l = P.Length(j)

Pre: j is a length group in P.

Post: l is the mean length of length group j in P.

$\mathcal{F}$ `int NoLengthGroups() const`

Use: *no* = P.NoLengthGroups()

Pre: None.

Post: *no* is the number of length groups in P.

NB: This function determines whether a number *length* is a valid length group in P or not. It is if $0 \leq$ *length* $<$ P.NoLengthGroups().

$\mathcal{C}$ `const doublevector& Bconsumption(int) const`

Use: *dv* = P.Bconsumption(*area*)

Pre: P.CheckConsumption(…) has been called since the last call to either P.Sum(…) or P.AddConsumption(…) for area *area* and P lives in the area *area*.

Post: *dv* is a reference to a vector of length P.NoLengthGroups() – in *dv*[i] is the predation in biomass units on length group i in P on area *area*.

$\mathcal{C}$ `const doublevector& OverConsumption(int) const`

Use: *dv* = P.OverConsumption(*area*)

Pre: P.CheckConsumption(...) has been called since last call to either P.Sum(...) or P.AddConsumption(...) for area *area*.
Post: *dv* is a reference to a vector of length P.NoLengthGroups() and in *dv*[*i*] is the overconsumption of length group *i* in P on area *area*.

$\mathcal{F}$ `const LengthGroupDivision* ReturnLengthGroupDiv() const`
Use: *lgrpdiv* = P.ReturnLengthGroupDiv()
Pre: None.
Post: *lgrpdiv* points to a LengthGroupDivision that describes P's length group division.

Details

The preconditions here above might seem a bit confusing at first. However, they are just to ensure that complete nonsense is not produced when calling the member functions. Let P be a fully initialzed object of type Prey, then the sequence of the informative and consumptions functions should be:
P.Sum(*NumberInArea*,*area*)
P.AddConsumption(*area*, *predconsumption*)
P.CheckConsumption(*area*)
As the name suggests, there may of course be several calls to P.AddConsumption(...).
Now the following member function calls make sense:
P.TooMuchConsumption(*area*)
P.Ratio(*area*, *length*)
P.OverConsumtion(*area*)

Protected messages

$\mathcal{S}$ `void InitializeObjects()`
Use: P.InitializeObjects()
Pre: None.
Post: P has initialized the size of its objects.

Protected Characteristics

`ConversionIndex*`	`CI`	// Conversion (see SetCI(...)).
`LengthGroupDivision*`	`LgrpDiv`	// The length group division.
`popinfomatrix`	`Number`	// Number and weight of prey.
`doublematrix`	`biomass`	// Biomass of prey in Area, [area][length].
`doublematrix`	`ratio`	// Ratio eaten hopefully < 1, [area][length].

`doublematrix`	`consumption`	// Consumption of prey, [area][length].
`intvector`	`tooMuchConsumption`	// Set if any lengthgr is overconsumed in area.
`doublevector`	`total`	// Total biomass of prey in the area.
`doublematrix`	`overconsumption`	// Indexed with [area][lengthgroup].

Details

The matrices `biomass`, `Number`, `ratio` and `consumption` are indexed with [area][length group].
`CI` converts from *OtherLgrpDiv* to `LgrpDiv`.
The member function Sum(...) should:

- Set `tooMuchConsumption`, `overconsumption` and `consumption` to 0.
- Set `Number`, `biomass` and `total`.

8.10 LengthPrey

The class LengthPrey is meant for those preys that are only length based.

Inheritance

class LengthPrey : `public` Prey

Public messages

`LengthPrey(CommentStream&, intvector, const char*, Keeper* const)`

Use: LengthPrey P(*infile*, *areas*, *givenname*, *keeper*)
Pre: *infile* has no badbits set, its format is correct and according to *areas*, which is a nonempty intvector whose elements are ≥ 0 and unique. *givenname* is a nullterminated string and $keeper \neq 0$.
Post: P is of type LengthPrey and has read its information from *infile*. It is defined on the areas *areas*[*i*], $i = 0, \ldots,$ *areas*.Size() - 1 and its name is *givenname*.
NB: The initialization of P is not complete until P.SetCI(...) has been called.

`LengthPrey(const doublevector&, const intvector&, const char*)`

Use: LengthPrey P(*lengths*, *areas*, *givenname*)
Pre: See the documentation of the base class.
Post: P is of type LengthPrey whose length group division has its endpoints in *lengths*[*i*], $i = 0, \ldots,$ *lengths*.Size() - 1.
NB: The initialization of P is not complete until P.SetCI(...) has been called.

`~LengthPrey()`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.

$\mathcal{AG}$ `void Sum(const popinfovector&, int)`

Use: P.Sum(*NumberInArea*, *area*)
Pre: P is in area *area*, P.SetCI(...) has been called and *NumberInArea* is according to the LengthGroupDivision received there.
Post: P has set its population in the area *area* to *NumberInArea*.

8.11 StockPrey

The class StockPrey is a interface for preys that are divided by age and length. It also provides access functions to compute the effects of the predation on the age groups.

Inheritance

class StockPrey : `public` Prey

Public messages

`StockPrey(CommentStream&, const intvector&, const char*, int, int, Keeper* const)`

Use: StockPrey P(*infile*, *areas*, *givenname*, *minage*, *maxage*, *keeper*)

NB: Refer to the documentation of Prey, except for *minage* and *maxage*. The preconditions are that $0 \leq minage \leq maxage$. The postconditions include that P is of type StockPrey, with minimum age equal to *minage* and maximum age equal to *maxage*; both included.

`StockPrey(const doublevector&, const intvector&, int, int, const char*)`

Use: StockPrey P(*lengths*, *areas*, *minage*, *maxage*, *givenname*)

Pre: *lengths* is in strictly ascending order, *lengths*.Size() > 1. Refer to the other constructor for the other parameters.

Post: P's length group division is given with *lengths*. Refer otherwise to the other constructor.

`virtual ~StockPrey()`

Use: ~P()

Pre: None.

Post: All memory belonging to P has been freed.

$\mathcal{AG}$ `virtual void Sum(const Agebandmatrix&, int)`

Use: P.Sum(*Alkeys*, *area*)

Pre: P is in the area *area* and *Alkeys* is according to the length group division received in SetCI(. . .).

Post: P has set its population to be that of *Alkeys* (using only that age groups of *Alkeys* whose age is between minage and maxage, both included.

$\mathcal{C}$ `const Agebandmatrix& AlkeysPriorToEating(int) const`

Use: *alk* = P.AlkeysPriorToEating(*area*)

Pre: P.Sum(. . .) has been called for the area *area*.

Post: *alk* is a reference to an Agebandmatrix, containing the age-length keys of P as received in the last call to P.Sum(...) for area *area*, divided to length groups according to P's length group division.

`virtual void Print(ofstream&) const`

Use: P.Print(*outfile*)

NB: Refer to the documentation of Prey.

Protected Characteristics

`Agebandmatrixvector Alkeys` // The population

Details

`Alkeys` contains the population as received in P.Sum(...), divided into length groups according to P's length group division.

8.12 Notes on Predator.

How to avoid the restriction of length based eating.
Change Predator in the following way:

- Delete the member function ReturnLengthGroupDiv().
- Insert the member functions double Minlength(int) and double Maxlength(int), which return the minimum and maximum lengths of group i. [Note, the frase "length group i" was not used here, but "group i"].
- Allow Predator to be **somehow** divided into groups, each of which has its minimum and maximum length, not restrict the group division to being a length group division.
- Change the name of the function NoLengthGroups() to NoGroups().

This requires hardly any changes in PopPredator or Prey. Some changes are, however, needed in the aggregator classes PredatorAggregator and PredOverAggregator.
PopPredator could continue to implement eating based on length groups since it does that fast, using LengthGroupDivision and ConversionIndex. However, a new derived class from Predator could now be made, one that implements this new strategy of division of predators. It could be used to implement age dependent eating habits.

8.13 Eating functions – details

This sections describes how the amount a predator eats of its preys is calculated in the member function Eat of Predator.
In what follows,
$prey$ will be a prey,
$pred$ a fixed predator.
l a length group in $prey$,
L a length group of the predator in question,
$S(l, L, prey)$ the suitability of length group l in $prey$ as food for length group L of the predator.
$N_{prey}(l)$ the number of length group l of $prey$.
$W_{prey}(l)$ the mean weight of length group l of $prey$.
$N_{pred}(L)$ the number of length group L of $pred$.
$W_{pred}(L)$ the mean weight of length group L of $pred$.
$\phi(l, L, prey) := S(l, L, prey) * N_{prey}(l)W_{prey}(l)$.
Note In the multispecies litterature, there is often used the concept *other food* to designate, loosely speeking, "everything that is too small to be modelled". Here that is incorporated into the class OtherFood which is also a prey, so that *other food* is actually contained in the description that follows.

TotalPredator

The TotalPredator tries to let the total amount eaten of its preys to be $N_{pred}(L)W_{pred}(L)$. This means that the amount eaten of length group l of a prey $prey$ is

$$\frac{N_{pred}(L)W_{pred}(L)\phi(l, L, prey)}{\sum_{prey,l} \phi(l, L, prey)}.$$

LinearPredator

The relationship between the amount each length group of LinearPredator eats and its biomass is linear; i.e. the amount each length group L of $pred$ wants to eat of a length group l of a prey $prey$ is

$$c * N_{pred}(L)W_{pred}(L)\phi(l, L, prey),$$

where c is a constant.

StockPredator

Define

$$f(L) := \frac{\sum_{l,prey} \phi(l, L, prey)}{\sum_{l,prey} \phi(l, L, prey) + A * E_1},$$

where E_1 is a constant and A is the size of the area in question. The amount length group L of *pred* eats of length group l of *prey* is then

$$N_{pred}(L)H_{pred}(L,T)f(L)\frac{\phi(l,L,prey)}{\sum_{l,prey}\phi(l,L,prey)}.$$

where T is the temperature of the area and H is a function, called *maximum consumption*. The function f is often called *feeding level*.

Chapter 9

Catch.

9.1 CatchData

The class CatchData reads matrices from file, describing the catch of various species. It can then be used to access these matrices.

Inheritance

class CatchData : protected LivesOnAreas, protected HasName

Public messages

`CatchData(CommentStream&, const char*, const AreaClass* const, const TimeClass* const)`

Use: CatchData CD(*infile*, *givenname*, *Area*, *TimeInfo*)
Pre: The format of *infile* is correct, *givenname* is a nullterminated string, $Area \neq 0$, $TimeInfo \neq 0$.
Post: CD is of type CatchData. It has copied the value of *givenname* and considers that to be its name. It has read information from *infile*, using information from *Area* and *TimeInfo*.

`~CatchData()`

Use: ~CD
Pre: None.
Post: All memory belonging to CD has been freed.

$\mathcal{F}$ `int FindStock(const Stock*)`

Use: $t =$ CD.FindStock(*stock*)
Pre: *stock* is not a nullpointer.
Post: If CD associates itself with the object pointed to by *stock*, t is 1, else found is 0. To determine the association, CD compares its name with that of *stock*'s, using partial match.

$\mathcal{F}$ `const LengthGroupDivision* ReturnLengthGroupDiv() const`

Use: *lgrpdiv* = CD.ReturnLengthGroupDiv()
Pre: None.
Post: *lgrpdiv* points to a LengthGroupDivision describing the catch data matrices CD keeps.

$\mathcal{F}$ `const bandmatrix* GetCatch(int, const TimeClass* const) const`

Use: *bmptr* = CD.GetCatch(*area*, *TimeInfo*)
Pre: *TimeInfo* $\neq 0$. The current time in *TimeInfo* is within the period got from *TimeInfo* in the constructor.
Post: *bmptr* points to a bandmatrix CD has for the catch in the area *area* on the current time if CD has one, else *bmptr* is 0.
Note that if *bmptr* $\neq 0$, *bmptr*$\rightarrow$Minage() does not have to equal 0.

Protected Characteristics

`LengthGroupDivision*`	`LgrpDiv`	// Describes the matrices.
`bandmatrixptrmatrix`	`Catchmatrix`	// –[area][time]
`int`	`numberofstocks`	// How many share the matrices?

Details

The vector `areas` contains the areas for which catch is kept. In `Catchmatrix`[a][t] is the catch for the area `areas`[a] on the timestep t. Note that $0 < t \leq$ *TimeInfo*$\rightarrow$NoTimeSteps(). If `numberofstocks` $\neq 1$, the values in `Catchmatrix` were divided with `numberofstocks` in the constructor.

9.2 Catch

The class Catch is a front to CatchData. Each instance of Catch links to a CatchData and retrieves the catch from it.

Inheritance

class Catch

Public messages

`Catch(CatchData*)`

Use: Catch C(*CD*)
Pre: $CD \neq 0$.
Post: C is of type Catch and it will get all its information from *CD*. Note that the lifetime of *CD* has at least to equal that of C. It is the users responsibility to delete *CD*.
NB: C has not been fully initialized until C.SetCI(const LengthGroup-Division* const) has been called.

`~Catch()`

Use: ~C
Pre: None.
Post: All memory belonging to C has been freed.

$\mathcal{F}$ `void Subtract(Agebandmatrix&, int, const TimeClass* const)`

Use: C.Subtract(*Alkeys*, *area*, *TimeInfo*)
Pre: C.SetCI(const LengthGroupDivision* const) has been called and *Alkeys* is according to the LengthGroupDivision received in SetCI(const LengthGroupDivision* const). $TimeInfo \neq 0$ and the current time in *TimeInfo* does is within the period *CD* got when it was created.
Post: If C found any catch data for the area *area* and current time, it was subtracted from *Alkeys*.

$\mathcal{F}$ `const bandmatrix* GetCatch(int, const TimeClass* const) const`

Use: *bmptr* = C.GetCatch(*area*, *TimeInfo*)
Pre: The current time in *TimeInfo* is within the period *CD* got at creation time.
Post:

$\mathcal{S}$ `void SetCI(const LengthGroupDivision* const)`

Use: C.SetCI(*GivenLDiv*)

Pre:	*GivenLDiv* $\neq 0$ and C.SetCI(const LengthGroupDivision* const) has not been called before.
Post:	C assumes that the Agebandmatrices it receives will be according to *GivenLDiv*.

Protected Characteristics

```
doublevector        Ratio                        // ???
ConversionIndex*    CI                           // Conversion.
CatchData*          Catchdata                    // To get matrices from.
```

Details

`CI` is for converting from the length group division got from *CD* to the LengthGroupDivision received in SetCI(const LengthGroupDivision* const).

Chapter 10

Stock and supplamentary.

10.1 Transition

This class moves an agegroup of one stock to another stock on a given step. If the latter one has birthday on that step, the age of the agegroup is incremented of 1.

Inheritance

class Transition : `protected` LivesOnAreas

Public messages

`Transition(CommentStream&, const intvector&, int, int,`
`int, Keeper* const)`

Use: Transition T(*infile*, *areas*, *age*, *minl*, *size*,*keeper*)
Pre: *areas*.Size() > 0, *age* ≥ 0, *minl* ≥ 0, *size* > 0, *keeper* $\neq 0$, *infile* has no badbits set and its format is correct.
Post: T has read the name of transition stock from *infile* and the transition step. T can only be called with the parameter *area* as one of the elements of *areas*. Every parameter of type Agebandmatrix must have a agegroup whose age is *age* in calls to T.

`~Transition()`

Use: ~T
Pre: None.
Post: All memory belonging to T has been freed.

$\mathcal{S}$ `void SetCI (const LengthGroupDivision* const)`

Use: T.SetCI(*GivenLDiv*)
Pre: *GivenLDiv* $\neq 0$ and T.SetStock has been called.
Post: T is ready to move to transition stock. All Agebandmatrix-es received are assumed to be according to *GivenLDiv*.

$\mathcal{S}$ `void SetStock(Stockptrvector& )`

Use: T.SetStock(*stockvec*)

Pre: In *stockvec* there is one and only one pointer to a stock whose name is the name of the transition stock associated with T.

Post: T has a pointer to its transition stock. A warning message is printed if transition stock is not defined on all the areas that T received in the constructor.

$\mathcal{AG}$ `void KeepAgegroup(int, Agebandmatrix&, const TimeClass* const)`

Use: T.KeepAgegroup(*area*, *Alkeys*, *TimeInfo*)

Pre: See the constructor regarding *area* and *Alkeys*, *TimeInfo* $\neq 0$ and T.SetCI(const LengthGroupDivison* const) has been called.

Post: If transition stock is not defined on area *area*, nothing is done. If *TimeInfo*→CurrentStep() equals the transition step in T, *Alkeys*'s agegroup *age* is set to zero and its previous values are kept in T. Else nothing is done. (Here, *age* is the parameter in the constructor).

$\mathcal{AG}$ `void MoveAgegroupToTransitionStock(int, const TimeClass*, int)`

Use: T.MoveAgegroupToTransitionStock(*area*, *TimeInfo*, *HasLgr*)

Pre: See constructor regarding *area*, *TimeInfo* $\neq 0$ and T.SetCI(const LengthGroupDivison* const) has been called. T.KeepAgegroup(…) has been called.

Post: If transition stock is not defined on area *area*, nothing is done. If *TimeInfo*→CurrentStep() equals the transition step in T the values last received in KeepAgegroup(…) are added to transition stock according to the rules described below. Else nothing is done.
If the transition stock has birthday on the current step, the addition to it is made a year older, else the age of the addition is not changed. If *HasLgr* is 0, the addition is just a regular one, made through Stock::Add(…). If *HasLgr* is 1, the addition is made through Stock::Renewal(…), using the number in the first length group in the last call to KeepAgegroup(…) as the stock abundance number.

`void Print(ofstream&) const`

Use: Print(*outfile*) const

Pre: T.SetCI(const LengthGroupDivison* const) has been called.

Post: T has written internal status information to *outfile.*

Protected Characteristics

```
int                  TransitionStep       // The step on which movements are possible.
char*                TransitionStockName  // Name of target stock.
Stock *              TransitionStock      // Pointer to target stock.
Agebandmatrixvector Agegroup             // Data repository.
ConversionIndex*     CI                   // For conversion when moving to TransitionStock.
int                  age                  // The age of the age group.
```

Data invariant

- If `TransitionStock` $\neq 0$, `TransitionStock`$\rightarrow$Name() $==$ `TransitionStockName`.

Details

Transition only works with one agegroup, its age is `age`.

10.2 NaturalM

This simple class is to keep the natural mortality rates.

Inheritance

class NaturalM

Public messages

`NaturalM(CommentStream&, int, int, const TimeClass* const, Keeper* const )`

Use: NaturalM NM(*infile*, *minage*, *maxage*, *TimeInfo*, *keeper*)

Pre: *infile* has no error bits set and its format is correct, $0 \leq minage \leq maxage$, $TimeInfo \neq 0$, $keeper \neq 0$.

Post: NM has read natural mortality rates from *infile* for the period got from *TimeInfo*.

$\mathcal{F}$ `const doublevector& ProportionSurviving(const TimeClass* const) const`

Use: *mort* = NM.ProportionSurviving(*TimeInfo*) const

Pre: The current year and step are within the period read in the constructor.

Post: In *mort*[*i*] there is the proportion of the population of age *i*+ *minage* that, on the time given with *TimeInfo*, dies accoring to the natural mortality rates read in the constructor.

$\mathcal{S}$ `void ReCalc()`

Use: NM.ReCalc()

Pre: None.

Post: NM has recalculated all the proportions that are returned in ProportionSurviving.

`void Print(ofstream& )`

Use: Print(*outfile*)

Pre: None.

Post: NM's internal information has been written to *outfile*.

Protected Characteristics

κ	`doubleindexvector`	`mortality`	// Natural mortality rates.
	`doublematrix`	`proportion`	// The proportion that dies.
	`doublevector`	`lengthofsteps`	// The length of the steps

Details

`proportion` is indexed with [*step*-1][*age*-*minage*] and `lengthofsteps` with [*step*-1].

10.3 InitialCond

Inheritance

class InitialCond

Public messages

`InitialCond(CommentStream&, const intvector&, Keeper* const)`

Use: InitialCond IC(*infile*, *area*, *keeper*)
Pre: *infile* has no badbits set and its format is correct, *keeper* $\neq 0$.
Post: IC has read data from *infile* and associates it with the elements of *area*.
NB: When receiving Agebandmatrixvectors, IC will assume that it is defined on all the areas in *area*.

`~InitialCond()`

Use: ~IC
Pre: None.
Post: Memory belonging to IC has been freed.

$\mathcal{S}$ `void Initialize(Agebandmatrixvector&) const`

Use: IC.Initialize(Alkeys)
Pre: IC.SetCI(const LengthGroupDivison* const) has been called and the size of Alkeys is consistent with that call, and Alkeys.Size() equals areas.Size().
Post: Alkeys has been initialized according to the information read in the constructor.

$\mathcal{S}$ `void SetCI(const LengthGroupDivision* const)`

Use: IC.SetCI(*GivenLDiv*)
Pre: *GivenLDiv* $\neq 0$.
Post: IC is ready to initialize.
NB: When initializing, IC, will assume that Agebandmatrixvectors received will be according to *GivenLDiv*.

`void Print(ofstream&) const`

Use: IC.Print(*outfile*)
Pre: SetCI(const LengthGroupDivison* const) has been called.
Post: Internal information in IC has been written to *outfile*.

Protected Characteristics

	`LengthGroupDivision*`	`LgrpDiv`	// Corresponds to read data.
	`ConversionIndex*`	`CI`	// Conversion between read data and target.
	`Agebandmatrixvector`	`AreaAgeLength`	// Data for initialization.
	`doublematrix`	`Distribution`	// Data for initialization - not used.
κ	`doubleindexvector`	`agemultiple`	// Multiplication by age.
	`intvector`	`areas`	// The areas to which read data is associated.

Details

For length group l of an age group a on area i, the initial stock size is set to be agemultiple[a]*AreaAgeLength[i][a][l].

10.4 Migration

Objects that migrate can use this class to read and access the migration matrices.

Inheritance

class Migration : `protected` LivesOnAreas

Public messages

`Migration(CommentStream&, int, const intvector&, const AreaClass* const, const TimeClass* const, Keeper* const)`

Use: Migration M(*infile*, *AgeDepMig*, *areas*, *Area*, *TimeInfo*, *keeper*)
Pre: *infile* has no badbits set and its format is correct, $Area \neq 0$, $TimeInfo \neq 0$, $keeper \neq 0$. *AgeDepMig* is 0 if M is not age dependent, 1 if it is.
Post: M has read from *infile* the information found for the period in question (based on *TimeInfo*).
NB: When calling member functions taking TimeClass as a parameter, the time must be within the period got from *TimeInfo* in the constructor.

`~Migration()`

Use: ~M()
Pre: None.
Post: All memory belonging to M has been freed.

$\mathcal{F}$ `const doublematrix& Migrationmatrix(const TimeClass* const, int = 0)`

Use: *migmatrixptr* = &M.Migrationmatrix(*TimeInfo*, *age*)
Use: *migmatrixptr* = &M.Migrationmatrix(*TimeInfo*)
Pre: *TimeInfo*.CurrentTime() must be within the period received from TimeClass in the constructor. If M was created with age dependent migration, migration matrices for *age* must have been read from *infile*.
Post: *migmatrixptr* points to the migration matrix M keeps for the current time (according to *TimeInfo*) - and agegroup *age* if M is age dependent.

$\mathcal{S}$ `void MigrationRecalc()`

Use: M.MigrationRecalc()
Pre: None.
Post: M has recalculated its migration matrices. If an error occurs, the error bit is set.

NB: See the description of the protected member function AdjustMigListAndCheckIfError(MigrationList&) to see when the error bit might be set.

$\mathcal{F}$ `void Error() const`

Use: *err* = M.Error()
Pre: None.
Post: *err* equals 1 if an error occurred in the last call to M.MigrationRecalc and then the migration matrices received from M may be unusable. If no error occurred in the last call to MigrationRecalc, *err* equals 0.

$\mathcal{S}$ `void Clear()`

Use: M.Clear()
Pre: None.
Post: M.Error() now returns 0.

`void Print (ofstream&) const`

Use: M.Print(*outfile*)
Pre: *outfile* has no error bits set.
Post: M has written internal information to *outfile*.

Protected Characteristics

	`int`	`AgeDepMigration`	// Depends migration on age?
	`intmatrix`	`MatrixNumbers`	// Numbers of matrices used.
	`MigrationList`	`ReadMigList`	// Matrices read from file.
	`MigrationList`	`CalcMigList`	// Precalculated matrices.
κ	`VariableInfoptrvector`	`OptInfo`	// Info about variables.
κ	`doublevector`	`OptVariables`	// Values of variables.
	`int`	`error`	// internal error status.
	`intmatrix`	`ages`	// Used if migration is agedependent.
	`intvector`	`AgeNr`	// Used if migration is agedependent.

Protected messages

$\mathcal{S}$ `void CopyFromReadToCalc()`

Use: M.CopyFromReadToCalc()
Pre: None.
Post: M has copied the values in `ReadMigList` to `CalcMigList`.

$\mathcal{S}$ `void AdjustMigListAndCheckIfError(MigrationList&)`

Use: M.AdjustMigListAndCheckIfError(*MigList*)
Pre: None.
Post: If all elements of every matrix in *MiglList* were ≥ 0 and the sum of every column in every matrix in *MigList* was 1, the error bit in M is not set. If these conditions are not fulfilled, it may, however, be set.
A possible strategy might be to

- take some action if a element of a migration matrix is < 0.
- scale the columns of the migration matrices so their sum equals 1, set the error bit if this is not possible (i.e. the sum equals 0).

d and the action taken if an element $x_{i,j}$ is < 0 could be one of:

- replace $x_{i,j}$ with -$x_{i,j}$
- replace $x_{i,j}$ with 0
- set the error bit and return.

Private messages

$\mathcal{S}$ `void ReadNoMigrationMatrices(CommentStream&, const TimeClass* const, Keeper* const)`
Use: `this`→ReadNoMigrationMatrices(*infile*, *TimeInfo*, *keeper*)
NB: This function is called in the constructor to read from *infile*. It reads the numbers of the migration matrices that are to be used in the period got from *TimeInfo*.

$\mathcal{S}$ `void ReadOptVariables(CommentStream&, intvector&, Keeper* const)`
Use: `this`→ReadOptVariables(*infile*, *novariables*, *keeper*)
NB: This function is called in the constructor to read from *infile*. It reads information on the migration variables.
Pre: *novariables* is an empty vector.
Post: `this`has read information on the migration variables from *infile* and put it in `OptVariables`. *novariables* contains the numbers of the variables read from *infile*. I.e. *novariables*.Size() = `OptVariables`.Size() and the variable *novariables*[*i*] had the value `OptVariables`[*i*].

$\mathcal{S}$ `void ReadCoefficients(CommentStream&, const AreaClass* const, Keeper* const)`

Use: `this`→ReadCoefficients(*infile*, *Area*, *keeper*)
NB: This function is called in the constructor to read from *infile*. It reads additions to the migration matrices.
Pre:
Post: The vector `OptInfo` has been set. The areas read from *infile* were converted to the inner areas of `this`.

$\mathcal{S}$ `void CheckInfoAndDelete(intvector&, Keeper* const)`

Use: `this`→CheckInfoAndDelete(*novariables*, *keeper*)
NB: This function should only be called once. It is currently called from within the constructor.
Pre: *novariables* should be the same vector as in the call to `this`→ReadOptVariables and so should *keeper*.
Post: `this`has checked if all the migration variables that were read in ReadOptVariables(...) cover the coefficients read in ReadCoefficients(...) and emitted error messages if not. Also, all the migration variables that were read but not needed have been deleted from `this`.
The references to the numbers of migration variables in `OptInfo` have been changed from the numbers read from file to inner numbers that are sequential, starting in 0.
I.e., now `OptInfo`[*i*]->coefficients[*j*] is the coefficient for the variable whose value is kept in `OptVariables`[`OptInfo`[*i*]->indices[*j*]].

Details

- If migration is age dependent, all the ages that use the same migration matrices are in `ages`[*i*], else `ages` is empty.
- `MatrixNumbers` keeps numbers of matrices used. If migration is age dependent, the number of the migration matrix for the ages in `ages`[*i*] is in `MatrixNumbers`[*i*][*t*], else in `MatrixNumbers`[1][*t*], where *t* is between 1 and *TimeInfo*→TotalNoSteps() + 1, inclusive.
- `ReadMigList`[*i*] is a pointer to matrix no. *i*, if it was read and is needed (meaning that it will be used in the simulation). Else `ReadMigList`[*i*] equals 0.

10.5 Grower

The class Grower calculates the growth of a population and updates it accordingly.

Inheritance

class Grower : `public` LivesOnAreas

Public messages

`Grower(CommentStream&, const LengthGroupDivision* const, const LengthGroupDivision* const, const intvector&, const TimeClass* const, Keeper* const)`

Use: Grower G(*infile*, *OtherLgrpDiv*, *GivenLgrpDiv*, *areas*, *TimeInfo*, *keeper*)

Pre: *infile* has no badbits set and its format is correct. *OtherLgrpDiv* $\neq 0$, *GivenLgrpDiv* $\neq 0$, *areas* is not empty and all its elements are unique and ≥ 0, *TimeInfo* $\neq 0$ and *keeper* $\neq 0$. The length group division *OtherLgrpDiv* points to has to have equal spacing (i.e. *OtherLgrpDiv*$\rightarrow$dl() $\neq 0$) and *GivenLgrpDiv* has to be coarser or equal to *OtherLgrpDiv*.

Post: G is of type Grower and has read its information from *infile*. G will work on the length group division determined by *GivenLgrpDiv* and expects to communicate through the length group division given by *OtherLgrpDiv*. G lives on the areas in *areas*. In the member functions where area is a parameter, it has to be one of the elements of *areas*.

`~Grower()`

Use: ~G

Pre: None.

Post: All memory belonging to G has been freed.

$\mathcal{AG}$ **`void GrowthCalc(int, const AreaClass* const, const TimeClass* const, const doublevector&, const doublevector&)`**

Use: G.GrowthCalc(*area*, *Area*, *TimeInfo*, *feeding*, *consumption*)

Pre: G was defined for the area *area*, *Area* $\neq 0$, *TimeInfo* $\neq 0$ and *feeding*.Size() and *consumption*.Size() equal to *GivenLgrpDiv*$\rightarrow$NoLengthGroups(). G.Sum has been called for the area *area*.

Post: G has calculated the growth on the area *area*, using *feeding* as the feeding level and *consumption* as the consumption in **biomass** units.

NB: Call G.GrowthImplement(...) to finish growth calculations.

$\mathcal{AG}$ `void GrowthCalc(int, const AreaClass* const, const TimeClass* const)`

Use: G.GrowthCalc(*area*, *Area*, *TimeInfo*)

Pre: G was defined for the area *area*, $Area \neq 0$ and $TimeInfo \neq 0$. G.Sum has been called for the area *area*.

Post: Has the same effects as calling G.GrowthCalc(*area*, *Area*, *TimeInfo*, *feeding*, *consumption*), with *feeding* and *consumption* containing only 0.

$\mathcal{AG}$ `void GrowthImplement(int, const popinfovector&, const LengthGroupDivision* const)`

Use: G.GrowthImplement(*area*, *NumberInArea*, *LgrpDiv*)

Pre: G is defined for the area *area*, $LgrpDiv \neq 0$. G.GrowthCalc has been called for the area *area*. *NumberInArea* **most likely** should be the same as in the call to G.Sum(. . .). and *LgrpDiv* the same as *OtherLgrpDiv*.

Post: G has finished its growth calculations on the area *area*.

$\mathcal{AG}$ `const doublematrix& LengthIncrease(int) const`

Use: dm = G.LengthIncrease(*area*)

Pre: G is defined on the area *area* and G.GrowthImplement has been called for the area *area*.

Post: dm is a reference to a doublematrix containing the length increase G calculated on the area *area* in the last call to G.GrowthImplement(. . .). $dm[g][l]$ is the proportion of length group l that is to move to length group $l + g$.

$\mathcal{AG}$ `const doublematrix& WeightIncrease(int) const`

Use: dm = G.WeightIncrease(*area*)

Pre: G is defined on the area *area* and G.GrowthImplement(. . .) has been called for the area *area*.

Post: dm is a reference to a matrix containing the weight increase G calculated on the area *area* in the last call to G.GrowthImplement(. . .). $dm[g][l]$ is the weight increase (positive or negative) of the part of the population in length group l that is to move to length group $l + g$.

$\mathcal{AG}$ `void Sum(const popinfovector&, int)`

Use: G.Sum(*NumberInArea*, *area*)

Pre: G is defined on the area *area* and *NumberInArea* contains the population of the area *area*.

Post: G keeps the information on the population for future use.

`void Print(ofstream&) const`

Use: G.Print(*outfile*)
Pre: *outfile* has no badbits set.
Post: G has written internal information to *outfile*.

$\mathcal{S}$ `void Reset()`
Use: G.Reset()
Pre: None.
Post: G has reset its internal status – i.e. cleared all previous values (useful in conjunction with Grower::Print).

Protected Characteristics

`LengthGroupDivision*`	`LgrpDiv`	//
`popinfomatrix`	`GrEatNumber`	// The population
`ConversionIndex*`	`CI`	// Conversion.
`doublematrix`	`InterpLgrowth`	// Length increase – [area][LengthGroup]
`doublematrix`	`InterpWgrowth`	// Weight increase – [area][LengthGroup]
`doublematrix`	`CalcLgrowth`	// Length increase – [area][LengthGroup]
`doublematrix`	`CalcWgrowth`	// Weight increase – [area][LengthGroup]
`doublematrixptrvector`	`lgrowth`	// – [area][LengthGroup][growth]
`doublematrixptrvector`	`wgrowth`	// – [area][LengthGroup][growth]
`doublevector`	`Fphi`	// Not used???
`GrowthImplementparameters*`	`GrIPar`	//
`GrowthCalcBase*`	`growthcalc`	// Pointer to growth function.

10.6 GrowthImplementparameters

The class GrowthImplementparameters is a supplamentary class when calculating the growth.
The class is only a simple data repository as can be seen from the member functions – they are only access functions to protected data.

Inheritance

class GrowthImplementparameters

Public messages

`GrowthImplementparameters(CommentStream&)`
Use: GrowthImplementparameters GI(*infile*)
Pre: The format of *infile* is correct and *infile* has no badbits set.
Post: GI is of type GrowhImplementparameters.

`void Print(ofstream&) const`
Use: GI.Print(*outfile*)
Pre: *outfile* has no badbits set.
Post: GI has written internal information to *outfile*.

$\mathcal{F}$ `int Maxlengthgroupgrowth() const`
Use: m = GI.Maxlengthgroupgrowth()
Pre: None.
NB: m is taken to be the maximum growth, measured in number of length groups.

$\mathcal{F}$ `double Resolution() const`
Use: r = GI.Resolution()
Pre: None.

$\mathcal{F}$ `double Power() const`
Use: b = GI.Power()
Pre: None.

$\mathcal{F}$ `const doublematrix& Distribution() const`
Use: dm = GI.Distribution()
Pre: None.
Post: dm is a reference to a matrix with GI.Maxlengthgroupgrowth() columns.
NB: $dm[i]$ is expected to be the distribution of the length increase, given that the mean length increase is $i*$GI.Resolution().

Protected Characteristics

`double`	`power`	//
`double`	`resolution`	//
`int`	`maxlengthgroupgrowth`	//
`double`	`maxmeangrowth`	//
`doublematrix`	`distribution`	//

10.7 GrowthCalc

The classes derived from the abstract base class GrowthCalcBase are only wrappers around growth functions. They handle the reading from file and the possible precalculations and define the interface to the growthfunction, i.e. the member function GrowthCalc.

Inheritance

class GrowthCalcBase : `protected` LivesOnAreas

Public messages

`GrowthCalcBase(const intvector&)`

Use: GrowthCalcBase G(*areas*)
Pre: *areas* is a nonempty vector, containing nonnegative unique integers.
Post: G is of type GrowthCalcBase and is defined on the areas *areas*.

`virtual ~GrowthCalcBase()`

Use: ~G
Pre: None.
Post: All memory belonging to G has been freed.

```
virtual void GrowthCalc(int, doublevector&,
doublevector&, const popinfovector&, const AreaClass*
const, const TimeClass* const, const doublevector&,
const LengthGroupDivision* const) const = 0
```

Use: G.GrowthCalc(*area*, *Lgrowth*, *Wgrowth*, *GrEatNumber*, *Area*, *TimeInfo*, *Fphi*, *Consumption*, *LgrpDiv*)
Pre: G is defined on the area *area*, *Lgrowth*.Size() = *Wgrowth*.Size() = *GrEatNumber*.Size() = *Fphi*.Size() = *Consumption*.Size() = *LgrpDiv*$\rightarrow$NoLengthGroups(). *Area* $\neq 0$ *TimeInfo* $\neq 0$ and *LgrpDiv* $\neq 0$
Post: G has calculated the length increase and weight change for the area *area*, using *GrEatNumber* as information on the population, *Fphi* as the feeding level and *LgrpDiv* as the length group division of the population.

Inheritance

class GrowthCalcA : `public` GrowthCalcBase

Public messages

`GrowthCalcA(CommentStream&, const intvector&, Keeper* const)`
  Use: GrowthCalcA G(*infile*, *areas*, *keeper*)
  NB: G contains the Multspec growthfunction.

`~GrowthCalcA()`
  Use: ~G

$\mathcal{A}$ `virtual void GrowthCalc(...)`
  Use: G.GrowthCalc(. . .)

Protected Characteristics

	`int`	`NumberOfGrowthConstants`	//
κ	`doublevector`	`Growthpar`	//

Inheritance

class GrowthCalcB : `public` GrowthCalcBase

Public messages

`GrowthCalcB(CommentStream&, const intvector&, const TimeClass* const, const LengthGroupDivision* const, Keeper* const)`
  Use: GrowthCalcB(*infile*, *areas*, *TimeInfo*, *LgrpDiv*, *keeper*)
  NB: B has read the growth from *infile*.

`~GrowthCalcB()`
  Use: ~G

$\mathcal{A}$ `virtual void GrowthCalc(...)`
  Use: G.GrowthCalc(. . .)

Protected Characteristics

`doublematrixptrvector`	`lgrowth`	// [area][timestep][length][group]
`doublematrixptrvector`	`wgrowth`	// [area][timestep][length][group]

Inheritance

class GrowthCalcC : `public` GrowthCalcBase

Public messages

`GrowthCalcC(CommentStream&, const intvector&, Keeper* const)`
Use: GrowthCalcC G(*infile*, *areas*, *keeper*)
NB: G contains the van Bertanlanfy growth function.

`~GrowthCalcC()`
Use: ~G

$\mathcal{A}$ `virtual void GrowthCalc(...)`
Use: G.GrowthCalc(. . .)

Protected Characteristics

	`int`	`NumberOfWGrowthConstants`	//
	`int`	`NumberOfLGrowthConstants`	//
κ	`doublevector`	`WGrowthpar`	//
κ	`doublevector`	`LGrowthpar`	//
	`doublevector`	`Wref`	//

Inheritance

class GrowthCalcD : `public` GrowthCalcBase

Public messages

`GrowthCalcD(CommentStream&, const intvector&, const LengthGroupDivision* const, Keeper* const)`
Use: GrowthCalcD G(*infile*, *areas*, *LgrpDiv*, *keeper*)
NB: G contains a growth function based on R. Jones.

`~GrowthCalcD()`
Use: ~G

$\mathcal{A}$ `virtual void GrowthCalc(...)`

Use: G.GrowthCalc(. . .)

Protected Characteristics

```
   int              NumberOfWGrowthConstants//
   int              NumberOfLGrowthConstants//
κ  doublevector     Wgrowthpar              //
κ  doublevector     Lgrowthpar              //
   doublevector     Wref                    //
```

Inheritance

class GrowthCalcE : `public` GrowthCalcBase

Public messages

`GrowthCalcE(CommentStream&, const intvector&, Keeper* const)`

Use: GrowthCalcE G(*infile*, *areas*, *keeper*)

NB: G contains the van Bertanlanfy growth function with additional year, step and area effects.

`~GrowthCalcE()`

Use: ~G

$\mathcal{A}$ `virtual void GrowthCalc(...)`

Use: G.GrowthCalc(. . .)

Protected Characteristics

```
   int              NumberOfWGrowthConstants//
   int              NumberOfLGrowthConstants//
κ  doublevector     WGrowthpar              //
κ  doublevector     LGrowthpar              //
   doublevector     Wref                    //
   doublevector     YearEffect              //
   doublevector     StepEffect              //
   doublevector     AreaEffect              //
```

10.8 Maturity

The class Maturity is an abstract base class. It should take care of transitions from an immature stock to mature stocks. Derived classes specify further the nature of these transactions, the time of the year, the proportion of the immature that becomes mature etc.

Inheritance

class Maturity : `protected` LivesOnAreas

Public messages

`Maturity()`

Use: Maturity M
Pre: None.
Post: M is of type Maturity.
NB: Not implemented.

`Maturity(const intvector&, int, const intvector&, const intvector&, const LengthGroupDivision*)`

Use: Maturity M(*areas*, *minage*, *minabslength*, *size*, *LgrpDiv*)
Pre: *areas* is a non-empty vector of unique positive integers, $minage \geq 0$, *minabslength*.Size() == *size*.Size(). The vector *size* is not empty and contains only positive integers. *minabslength* contains non-negative integers. $LgrpDiv \neq 0$.
Post: M is of type Maturity. It lives on the areas *areas*, its minimum age is *minage*, the mininum number of length group for age group a is $minabslength[a-minage]$ and the number of length groups for that age group is $size[a-minage]$. M assumes the length group division of the population it receives to be according to *LgrpDiv*.
NB: M is not fully functional until SetStock(Stockptrvector&) has been called.

`virtual ~Maturity()`

Use: ~M
Pre: None.
Post: All memory belonging to M has been deleted.

$\mathcal{S}$ `void SetStock(Stockptrvector&)`

Use: M.SetStock(*stockvec*)
Pre: *stockvec* is a non-empty vector of unique non-null pointers. It contains pointers to the mature stocks that M is associated with.
Post: M keeps pointers to the mature stocks, it is associated with.

`virtual void Print(ofstream&) const`

Use: M.Print(*outfile*)

Pre: M.SetStock(Stockptrvector&) has been called and *outfile* has no badbits set.

Post: M has printed internal information to *outfile*.

$\mathcal{S}$ `virtual void Precalc()`

Use: M.Precalc()

Pre: M.SetStock(Stockptrvector&) has been called.

Post: M has done the precalculations of maturity ratio it can.

NB: It is quite plausible that derived classes require this function to be called after Keeper::Update(const doublevector&) has been called.

`virtual int IsMaturationStep(int, const TimeClass* const) = 0`

Use: is = M.IsMaturationStep(area, TimeInfo)

Pre: TimeInfo $\neq 0$.

Post: is equals 1 if these is any probability of the immature stock becoming mature on the current time step and area.

`virtual double MaturationProbability(int, int, int, const TimeClass* const, const AreaClass* const, int) = 0`

Use: r = M.MaturationProbability(*age*, *length*, *Growth*, *TimeInfo*, *Area*, *Area*)

Pre: M.Precalc has been called. *TimeInfo* $\neq 0$, *Area* $\neq 0$. $minage \leq age < minage +$ *minabslength*.Size() where *minage* and *minabslength* are from the call to the constructor. M lives on the area *area* (see the constructor documentation).

Post: r contains the proportion that becomes mature of immature fish of age *age*, in the length group *length* that is to grow *Growth* at the current step on the area *area*.

$\mathcal{AG}$ `virtual int PutInStorage(int, int, int, double, double, const TimeClass* const)`

Use: M.PutInStorage(*area*, *age*, *length*, *number*, *weight*, *TimeInfo*)

Pre: M.IsMaturationStep(*area*, *TimeInfo*) equals 1. See MaturationProbability for the parameters *area*, *age*, *length* and *TimeInfo*. $number \geq 0$, $weight \geq 0$.

Post: M keeps the numbers *number* and *weight* for the age-length group (*age*, *length*) on the area *area*. It overwrites any previous information for that age-length group on that area. These numbers will then be used as stock numbers and mean weights of the population that is to be moved to the mature stocks when Move is called, assuming they have not been overwritten before then.
If iskept $\neq 1$, iskept equals 0, and nothing has been done.

$\mathcal{AG}$ `virtual void Move(int, const TimeClass* const)`

Use: M.Move(*area*, *TimeInfo*)
Pre: M lives on the area *area* and *TimeInfo* $\neq 0$.
Post: M has added to the mature stocks the immature population that is to be added to it on the current timestep on the area *area*.

Protected Characteristics

`charptrvector`	`NameOfMatureStocks`	// Set by derived classes.
`doublevector`	`Ratio`	//
`Stockptrvector`	`MatureStocks`	// Pointers to MatureStocks.
`ConversionIndexptrvector`	`CI`	// Convert betw. immature and mature.
`LengthGroupDivision*`	`LgrpDiv`	// The length group division.

Data invariant

- Before SetStock(Stockptrvector&) has been called, `Ratio`[i] is the proportion of the population of the immature stock that moves to the mature stock `NameOfMatureStocks`[i].
- After SetStock(Stockptrvector&) has been called, `Ratio`[i] is the proportion of the population of the immature stocks that moves to the mature stock pointed to in `MatureStocks`[i].
- After SetStock(Stockptrvector&) has been called `CI`[i] is a pointer to a ConversionIndex that converts from the immature stock to the mature stock `MatureStocks`[i].
- `NameOfMatureStocks`[i] is either 0 or a nullterminated string, created with new char[.].

Details

Derived classes should set `Ratio` and `NameOfMatureStocks` before any member function of M is called. They should not need to change the vectors `CI` and `MatureStocks`; M should do that itself in SetStock(Stockptrvector&).

Private Characteristics

`Agebandmatrixvector Storage` // Keeps immature waiting to be mature.

10.9 MaturityA

The class MaturityA implements growth dependent maturity. The probability that immature fish of length l and age a to become mature is:

$$\frac{1}{1-M}\frac{dM}{dt}$$

where

$$M(l(t),a(t)) = \frac{1}{1+e^{-\alpha-\beta l(t)-\gamma a(t)}}$$

is the maturity ogive.
Pre- and postconditions are often omitted in the description below, in which case the reader should refer to the documentation of the base class.

Inheritance

class MaturityA : `public` Maturity

Public messages

```
MaturityA(CommentStream&, const TimeClass* const,
Keeper* const, int, const intvector&, const intvector&,
const intvector&)
```

Use: MaturityA M(*infile*, *TimeInfo*, *keeper*, *minage*, *minabslength*, *size*, *areas*)
Pre: *infile* has no badbits set and its format is correct. *TimeInfo* $\neq 0$, *keeper* $\neq 0$, see the documentation of Maturity for explanations and the precoditions of *minage*, *minabslength*, *size* and *areas.*
Post: M is of type MaturityA.

```
virtual ~MaturityA()
```

Use: ~M()
Pre: None.
Post: All memory belonging to M has been freed.

$\mathcal{S}$ `virtual void Precalc()`
Use: M.Precalc()

$\mathcal{F}$ `virtual int IsMaturationStep(int, const TimeClass* const)`

Use: t = M.IsMaturationStep(*area*, *TimeInfo*)

$\mathcal{F}$ `virtual double MaturationProbability(int, int, int, const TimeClass* const, const AreaClass* const, int)`

Use: r = M.CalcMaturationRatio(*age*, *length*, *Growth*, *TimeInfo*, *Area*, *area*)

`virtual void Print(ofstream&)`

Use: M.Print(*outfile*) const

Protected Characteristics

	`bandmatrix`	`PrecalcMaturation`	//
κ	`doublevector`	`Coefficients`	// α, β, γ

Details

Since

$$1 - M = \frac{e^{-\alpha-\beta l-\gamma a}}{1+e^{-\alpha-\beta l-\gamma a}}$$

we obtain that

$$\begin{aligned} \frac{dM}{dt} &= \frac{\partial M}{\partial l}\frac{dl}{dt} + \frac{\partial M}{\partial a}\frac{da}{dt} \\ &= (\beta\frac{dl}{dt} + \gamma\frac{da}{dt})\frac{e^{-\alpha-\beta l-\gamma a}}{(1+e^{-\alpha-\beta l-\gamma a})^2} \\ &= (\beta\frac{dl}{dt} + \gamma\frac{da}{dt})(1-M)M \end{aligned}$$

Here we regarded age as continous in time, not a step function. Therefore

$$\frac{1}{1-M}\frac{dM}{dt} = (\beta\frac{dl}{dt} + \gamma\frac{da}{dt})M,$$

so we precalculate M and keep in `PrecalcMaturation`.

10.10 MaturityB

The class MaturityB implements annual maturity. It can use several functions to calculate the proportion that becomes mature.
Pre- and postconditions are often omitted in the description below, in which case the reader should refer to the documentation of the base class.

Inheritance

class MaturityB : `public` Maturity

Public messages

`MaturityB(CommentStream&, const TimeClass* const,`
`Keeper* const, int, const intvector&, const intvector&,`
`const intvector&)`

Use: M.MaturityB(*infile*, *TimeInfo*, *keeper*, *minage*, *minabslength*, *size*, *areas*)

`virtual ~MaturityB()`

Use: ~M

`virtual void Print(ofstream&) const`

Use: M.Print(*outfile*)

$\mathcal{S}$ `virtual void Precalc()`

Use: M.Precalc()

$\mathcal{F}$ `virtual int IsMaturationStep(int, const TimeClass*`
`const)`

Use: t = M.IsMaturationStep(*area*, *TimeInfo*)

$\mathcal{F}$ `virtual double MaturationProbability(int,int,int, const`
`TimeClass* const, const AreaClass* const, int)`

Use: M.MaturationProbability(*age*, *length*, *Growth*, *TimeInfo*, *Area*, *area*)

Protected Characteristics

`const int`	`maturitytype`	//
`intvector`	`maturitystep`	// When to become mature.
`doublevector`	`maturitylength`	// Maturity by length.

10.11 Spawner

The class Spawner handles the spawning process which may include mortality and weightloss.

Inheritance

class Spawner : `protected` LivesOnAreas

Public messages

`Spawner(CommentStream&, int, int, const AreaClass* const, const TimeClass* const, Keeper* const)`

Use: Spawner S(*infile*, *minage*, *maxage*, *Area*, *TimeInfo*, *keeper*)

Pre: *infile* has no error bits set and its format is correct, $0 \leq$ *minage* $\leq$ *maxage*, *Area* $\neq 0$, *TimeInfo* $\neq 0$, *keeper* $\neq 0$. The information in *infile* is given for the age groups from *minage* to *maxage*, both included.

Post: S has read its information from *infile* and can handle the spawning process.

$\mathcal{A}$ `void Spawn (Agebandmatrix&, int, const AreaClass* const, const TimeClass* const)`

Use: S.Spawn(*Alkeys*, *area*, *Area*, *TimeInfo*)

Pre: *area* ≥ 0, *Area* $\neq 0$, *TimeInfo* $\neq 0$.

Post: If S handles spawning actions on the area *area* and spawning takes place on the current time step, S has changed the age groups in *Alkeys* whose age is between *minage* and *maxage*, inclusive, according to the effects of the spawning [spawning mortality and weightloss].

Protected Characteristics

	`intindexvector`	`Spawningstep`	// [age]
	`doubleindexvector`	`Spawningratio`	// [age]
κ	`double`	`Spawningmortality`	//
	`doubleindexvector`	`SpawningmortalityPattern`	// [age]
κ	`double`	`Spawningweightloss`	//
	`doubleindexvector`	`SpawningweightlossPattern`	// [age]
	`double`	`Eggproduction`	// Not used.
	`doublevector`	`EggproductionPattern`	// Not used.

Details

The spawning mortality used for age group a is `Spawningmortality*SpawningmortalityPattern`$[a]$, similar for weight loss in the spawning.

10.12 Renewaldata

This class is meant to handle some of the additions to Stock.

Inheritance

class RenewalData : `protected` LivesOnAreas

Public messages

`Renewaldata(CommentStream&, const intvector&, const AreaClass* const, const TimeClass* const, Keeper* const)`

Use: Renewaldata R(*infile*, *areas*, *Area*, *TimeInfo*, *keeper*)
Pre: *infile* has no error bits set and its format is correct, $Area \neq 0$, $TimeInfo \neq 0$, $keeper \neq 0$. *areas*.Size() contains unique nonnegative integers.
Post: R has read information from *infile* for the period obtained from *TimeInfo* and expects *area* in the consequent calls to be one of the elements of *areas*.

`~Renewaldata()`

Use: ~R
Pre: None.
Post: All memory belonging to R has been freed.

$\mathcal{S}$ `void SetCI(const LengthGroupDivision* const)`

Use: R.SetCI(*GivenLDiv*)
Pre: $GivenLDiv \neq 0$.
Post: R is ready to add to Agebandmatrix-es.
NB: R assumes that *GivenLDiv* describes the length group division of all Agebandmatrix-es received.

$\mathcal{S}$ `void Addrenewal(Agebandmatrix& ,int, const TimeClass* const ,double = 0)`

Use: R.Addrenewal(*Alkeys*, *area*, *TimeInfo*, *Ratio*)
Use: R.Addrenewal(*Alkeys*, *area*, *TimeInfo*)
Pre: R.SetCI(…) has been called and *Alkeys* is consistent with that call. $Ratio \geq 0$. Unless R.Reset() has been called since last call to R.Addrenewal(…), the current time in *TimeInfo* has to be greater than or equal to the current time in the TimeClass in the last call to R.Addrenewal(…).

Post: The renewal data R for area *area* and the current time has been added to *Alkeys*, if one was read when R was created. Else nothing is done. If *Ratio* $\neq 0$ and the read number for the renewal data is 0 (see file format description), *Ratio* was regarded as a multiplicative factor to the length distribution R added to *Alkeys*. If read number and *Ratio* both equal to 0, nothing is done, and also if read number and *Ratio* are both different from 0.

NB: Using *Ratio* $\neq 0$ is a way of transforming from a Stock with no length group division to one that has one, using a predetermined length distribution. Then the number R read when created must be 0.

Warning: When using this function with *Ratio* $\neq 0$, **all** the length distributions with number equal to 0 are multiplied with *Ratio* and added to *Alkeys*.

`void Print(ofstream&) const`

Use: R.Print(*outfile*)
Pre: R.SetCI(...) has been called.
Post: R has written internal information to *outfile*.

$\mathcal{S}$ `void Reset()`

Use: R.Reset()
Pre: None.
Post: R has reset its internal information, so that current time in the next call to R.Addrenewal(...) does not have to be greater than or equal to the current time in the last call to R.Addrenewal(...).

Protected Characteristics

	`intvector`	`RenewalTime`	// times when renewal takes place.
	`intvector`	`RenewalArea`	// areas on which renewal takes place.
	`Agebandmatrixvector`	`Distribution`	// Lengths and mean weights for renewal.
κ	`doublevector`	`Number`	// Total stock numbers.
	`int`	`Minr`	// Used when searching for renewal data.
	`int`	`Maxr`	// Used when searching for renewal data.
	`ConversionIndex*`	`CI`	// Converting between read data and target.
	`LengthGroupDivision*`	`LgrpDiv`	// Describes renewal data.

Details

- The elements of the vector `RenewalTime` are in the are in the same order as when they were read at creation time.

- For the time `RenewalTime`[i], further information is in `RenewalArea`[i], `Number`[i] and `Distribution`[i].
- When `Number`[i] is not 0, it is considered a multiplicative factor to the length distribution in `Distribution`[i] (See Addrenewal(...) for further information).
- `Minr` and `Maxr` are set in the constructor and used in Addrenewal(...) only. They are reset to their initial values in Reset().

Data invariant

In the end of each call to Addrenewal(...), `RenewalTime`[`Maxr`] > *TimeInfo*→CurrentTime() or (`RenewalTime`[`Maxr`] == *TimeInfo*→CurrentTime() and `Maxr` == `RenewalTime`.Size() - 1).

10.13 Stock

The class Stock is capable of doing many things as can easily be seen from the access functions doeseat, doesmigrate, iseaten, . . .

Inheritance

class Stock : `public` BaseClass

Public messages

`Stock(CommentStream&, const char*, const AreaClass* const, const TimeClass* const, Keeper* const)`

Use: Stock S(*infile*, *givenname*, *Area*,*TimeInfo*, *keeper*)

Pre: *infile* has no error bits set and its format is correct, so is the format of all of its subfiles, *givenname* is a nullterminated string, $Area \neq 0$, $TimeInfo \neq 0$, $keeper \neq 0$.

Post: S has been created and has read information for the period of time obtained from *TimeInfo*.

NB: In all calls to member functions of S, the current time of TimeClass must be within the period obtained from *TimeInfo* at the point of creation. Also, current time must be increasing in calls to S; it may never decrease between subsequent calls.
In every member function where area is a parameter, it has only effect to S (and perhaps other objects) on that area. This is usually not mentioned in the postconditions.
S read from *infile* on which areas it is defined. That can be accessed through the function BaseClass::IsInArea(int).
To initialize S, the functions SetCatch(CatchDataptrvector&), SetStock(Stockptrvector&, SetCI() and Reset() must be called. Furthermore, if S is eaten its Prey has to be initialized, and if S eats the Predator has to be initialized.

`virtual ~Stock()`

Use: ~S()

Pre: None.

Post: Memory belonging to S has been freed.

$\mathcal{AG}$ `virtual void CalcNumbers(int, const AreaClass* const, const TimeClass* const)`

Use: S.CalcNumbers(*area*, *Area*, *TimeInfo*)

Pre: S is defined on area *area*, S has been initialized.

Post: S is ready to grow and eat.

$\mathcal{AG}$ `void ReducePop(int, const AreaClass* const,const TimeClass* const)`

Use: S.ReducePop(*area*, *Area*, *TimeInfo*)

Pre: S is defined on area *area*, $Area \neq 0$, $TimeInfo \neq 0$ and S has been initialized.

Post: If S is eaten, its population has been reduced accordingly, if it is caught, the catch has been subtracted. Deaths caused by natural mortality have been subtracted.

NB: Action if predation or catch exceeds size of S.

$\mathcal{AG}$ `void Grow(int, const AreaClass* const, const TimeClass* const) = 0`

Use: S.Grow(*area*, *Area*, *TimeInfo*)

Pre: S is defined on area *area* and S has been initialized.

Post: If S grows, it has calculated its growth and updated its lengths and mean weights on the area *area*.

`virtual void FirstSpecialTransactions(int, const AreaClass* const, const TimeClass* const)`

Use: S.FirstSpecialTransactions(*area*, *Area*, *TimeInfo*)

Pre: S is defined on area *area*, $Area \neq 0$, $TimeInfo \neq 0$ and S has been initialized.

Post: If S spawns, spawning has been calculated.

`virtual void SecondSpecialTransactions(int, const AreaClass* const, const TimeClass* const)`

Use: S.SecondSpecialTransactions(*area*, *Area*, *TimeInfo*)

Pre: S is defined on area *area*, $Area \neq 0$, $TimeInfo \neq 0$ and S has been initialized.

Post: Maturity movements on the area are finished. If there is any renewal in S, it has been added on the area.

$\mathcal{AG}$ `virtual void FirstUpdate(int, const AreaClass* const,const TimeClass* const)`

Use: S.FirstUpdate(*area*, *Area*, *TimeInfo*)

Pre:

Post: First part of age related update on area *area* is finished, i.e. if S moves to another stock, subtraction for the area is finished.

$\mathcal{AG}$ `virtual void SecondUpdate(int, const AreaClass* const,const TimeClass* const)`

Use: S.SecondUpdate(*area*, *Area*, *TimeInfo*)

Pre: S.FirstUpdate has been called for the area *area* on the current time step.

Post: Age has been updated on area *area*.

$\mathcal{AG}$ `virtual void ThirdUpdate(int, const AreaClass* const,const TimeClass* const)`

Use: S.ThirdUpdate(*area*, *Area*, *TimeInfo*)
Pre: S.SecondUpdate has been called for the area *area* on the current time step.
Post: Last part of age related update on area *area* is finished, i.e. if S moves to another stock, the transition on the area is finished.

$\mathcal{S}$ `void Renewal(int, const TimeClass* const, double = 0)`

Use: S.Renewal(*area*, *TimeInfo*, *ratio*)
Pre: S is defined on area *area*, $TimeInfo \neq 0$ and S has been initialized.
Post: Renewal for the current time and area has been added to S.
NB: This function should be in a different scope - protected.

$\mathcal{A}$ `void Add(const Agebandmatrix&, const ConversionIndex* const, int, double = 1, int = 0, int = 100)`

Use: S.Add(*Addition*, *CI*, *area*, *ratio*, *MinAge*, *MaxAge*)
Pre: *CI* points to a ConversionIndex converting from *Addition* to S.AgeLengthkeys(), S is defined on area *area*.
Post: *Addition* has been added to the population of S on area *area*.
NB: Needs expl. of param. *ratio*, *MinAge* and *MaxAge*.

$\mathcal{S}$ `virtual void Migrate(const TimeClass* const)`

Use: S.Migrate(*TimeInfo*)
Pre: $TimeInfo \neq 0$.
Post: If S migrates, it has done so.

$\mathcal{AG}$ `virtual void CalcEat(int, const AreaClass* const, const TimeClass* const)`

Use: S.CalcEat(*area*, *Area*, *TimeInfo*)
Pre: S is defined on area *area*, $Area \neq 0$, $TimeInfo \neq 0$ and S has been initialized. If S eats, CalcNumbers(...) must have been called on the current timestep.
Post: If S eats, its predation has been calculated, else nothing was done.

$\mathcal{AG}$ `virtual void CheckEat(int, const AreaClass* const, const TimeClass* const)`

Use: S.CheckEat(*area*, *Area*, *TimeInfo*)
Pre: S.CalcEat(*area*, *Area*, *TimeInfo*) has been called.
Post: If S eates, it has checked whether it could eat the amount it wanted to of its preys, else nothing was done.

$\mathcal{AG}$ `virtual void AdjustEat(int, const AreaClass* const, const TimeClass* const)`

Use: S.AdjustEat(*area*, *Area*, *TimeInfo*)

Pre: S.CheckEat(*area*, *Area*, *TimeInfo*) has been called.
Post: If S eats it has made some adjustments according to the results of the last call of CheckEat(...), else nothing was done.

$\mathcal{S}$ `void Reset()`
Use: S.Reset()
Pre: None.
Post: S has reset its population to the initial value and recalculated its inner status, including the error status.

$\mathcal{S}$ `virtual void Clear()`
Use: S.Clear()
Pre: None.
Post: S has cleared its error status.

$\mathcal{F}$ `StockPrey* ReturnPrey() const`
Use: *prey* = S.ReturnPrey()
Pre: None.
Post: *prey* points to the Prey S uses.

$\mathcal{F}$ `StockPredator* ReturnPredator() const`
Use: *pred* = S.ReturnPredator()
Pre: None.
Post: *pred* points to the predator S uses.

$\mathcal{C}$ `const Agebandmatrix& Agelengthkeys(int) const`
Use: *alkeys* = S.Agelengthkeys(*area*)
Pre: S.IsInArea(*area*) == 1.
Post: *alkeys* is a reference to an Agebandmatrix describing S's population on the area *area*.

$\mathcal{S}$ `void SetStock(Stockptrvector&)`
Use: S.SetStock(*stockvec*)
Pre: *stockvec* contains no 0 pointers.
Post: S has set its pointers to other stocks. *stockvec* has not been changed.

$\mathcal{S}$ `void SetCatch(CatchDataptrvector&)`
Use: S.SetCatch(*CDvector*)
Pre: *CDvector* contains no null pointers and one of its elements keeps catch data for S if S is caught .
Post: S has a pointer to its CatchData and can access its catch.

$\mathcal{S}$ `void SetCI()`
Use: S.SetCI()

Pre: SetStock(Stockptrvector&) has been called and, if S is caught, SetCatch(CatchDataptrvector&) has been called.
Post: S has set its conversion indices ready for transactions between stocks.

`void Print(ofstream&) const`
Use: S.Print(*outfile*)
Pre: SetCatch(CatchDataptrvector&), SetCI() and Set-Stock(Stockptrvector&) have been called, if necessary.
Post: S has written internal information to *outfile*.

$\mathcal{F}$ `int Birthday(const TimeClass* const) const`
Use: t = S.Birthday(*TimeInfo*)
Pre: *TimeInfo* $\neq 0$.
Post: If S's age will be increased of 1 on the current timestep, t equals 1, else it is 0.

$\mathcal{F}$ `const LengthGroupDivision* ReturnLengthGroupDiv() const`
Use: *lgrpdiv* = S.ReturnLengthGroupDiv()
Pre: None.
Post: *lgrpdiv* points to a LengthGroupDivision describing S's length group division.

$\mathcal{F}$ `int Minage() const`
Use: m = S.Minage()
Pre: None.
Post: m equals the age of the first age group in S.

$\mathcal{F}$ `int Maxage() const`
Use: m = S.Maxage()
Pre: None.
Post: m eaquals the age of the oldest age group in S.

$\mathcal{F}$ `int IsEaten() const`

$\mathcal{F}$ `int IsCaught() const`

$\mathcal{F}$ `int DoesSpawn() const`

$\mathcal{F}$ `int DoesMove() const`

$\mathcal{F}$ `int DoesEat() const`

$\mathcal{F}$ `int DoesMature() const`

$\mathcal{F}$ `int HasLgr() const`

$\mathcal{F}$ `int DoesRenew() const`

$\mathcal{F}$ `int DoesGrow() const`

$\mathcal{F}$ `int DoesMigrate() const`

Protected Characteristics

`Agebandmatrixvector`	`Alkeys`	// Keeps the population.
`Spawner*`	`spawner`	// Spawning process.
`Catch*`	`catchptr`	// For catch subtraction
`Renewaldata*`	`renewal`	// Addition to stock through import.
`Maturity*`	`maturity`	// Calculations and movements because of maturity.
`Transition*`	`transition`	// Transition to other stock.
`int`	`AgeDepMigration`	//
`Migration*`	`migration`	// Keeps migration matrices.
`StockPrey*`	`prey`	//
`StockPredator*`	`predator`	//
`InitialCond*`	`initial`	// keeps the initial population.
`LengthGroupDivision*`	`LgrpDiv`	// Division into length groups.
`Grower*`	`grower`	// Calculates and updates.
`NaturalM*`	`NatM`	// Keeps the mortality rates.
`popinfomatrix`	`NumberInArea`	// Intermediate data repository.
`doublematrix`	`ConsumptionInArea`	// Intermediate data repository.
`int`	`iscaught`	//
`int`	`doeseat`	//
`int`	`doesmove`	//
`int`	`iseaten`	//
`int`	`doesspawn`	//
`int`	`haslgr`	//
`int`	`doesmature`	//
`int`	`doesrenew`	//
`int`	`doesgrow`	//
`int`	`doesmigrate`	//

Chapter 11

Supplamentary classes.

11.1 ActionAtTimes

The class ActionAtTimes is made for assisting with events that happen only at predefined times.
The class can read from file, and information can be added to it through member functions. Then it can be asked at any time, whether the event is to take place at the current time or not.

Inheritance

class ActionAtTimes

Public messages

`ActionAtTimes()`
Use: ActionAtTimes AAT
Pre: None.
Post: AAT is of type ActionAtTimes.

$\mathcal{S}$ `int ReadFromFile(CommentStream&, const TimeClass* const)`
Use: *ok* = AAT.ReadFromFile(*infile*, *TimeInfo*)
Pre: *infile* has no badbits set and its format is correct, *TimeInfo* $\neq 0$.
Post: if *ok* is 0 an error occurred and no action on AAT is defined. Else AAT has read information from *infile* and kept it for the period *TimeInfo* marks.

$\mathcal{S}$ `void AddActions(const intvector&, const intvector&, const TimeClass* const)`
Use: AAT.AddActions(*years*, *steps*, *TimeInfo*)

Pre: *years*.Size() == *steps*.Size(), *TimeInfo* $\neq 0$. For all $i = 0, \ldots,$ *steps*.Size() - 1, $1 \leq$ *steps*[i] $\leq$ *TimeInfo*→StepsInYear().
Post: AAT has added to its list of dates those of the dates *years*[i], *steps*[i] that are within the period from *TimeInfo*, $i = 0, \ldots,$ *years*.Size() - 1.

$\mathcal{S}$ `void AddActionsAtAllYears(const intvector&, const TimeClass* const)`
Use: AAT.AddActionsAtAllYears(*steps*, *TimeInfo*)
Pre: *TimeInfo* $\neq 0$ and $1 \leq$ *steps*[i] $\leq$ *TimeInfo*→StepsInYear() for all $i = 0, \ldots,$ *steps*.Size() - 1.
Post: AAT has added to its list of events, all the dates with step equal to *steps*[i], for some i, that are within the period *TimeInfo* marks.

$\mathcal{S}$ `void AddActionsAtAllSteps(const intvector&, const TimeClass* const)`
Use: AAT.AddActionsAtAllSteps(*years*, *TimeInfo*)
Pre: *TimeInfo* $\neq 0$.
Post: AAT has added to its list of events all the dates that are within the period *TimeInfo* marks, and whose year is in *years*.

$\mathcal{F}$ `int AtCurrentTime(const TimeClass* const) const`
Use: t = AAT.AtCurrentTime(*TimeInfo*)
Pre: *TimeInfo* $\neq 0$.
Post: t is 1 if the current time, got from *TimeInfo*, is in the list of event times AAT keeps, else t is 0.

Protected Characteristics

Here, 'it is true', means that the function AtCurrentTime returns 1.

```
int          EveryStep      // Is it true on every step?
intvector    TimeSteps      // The timesteps on which it is true.
intvector    Years          // The years on which it is always true.
intvector    Steps          // The steps on which it is always true.
```

Chapter 12

Standard aggregation and printing.

12.1 Overview

In this section, the following classes that write data to files will be documented:

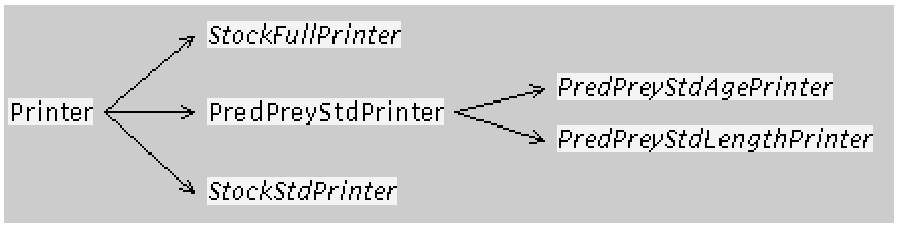

Figure 12.1: Standard Printer classes derived from Printer.

In this section the following classes that collect data on the impact of predation on the preys are documented:

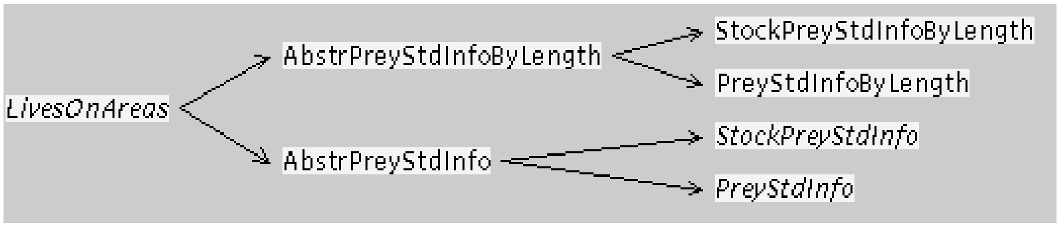

Figure 12.2: Classes that collect standard data from Prey and descendants.

And the classes that collect data on the predation by a predator on a prey are:

12.2 Printer

The class Printer is an abstract base class for handling printing. It is described here as if it could be instantiated.

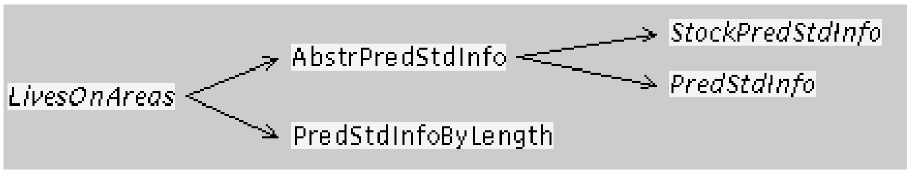

Figure 12.3: Classes that collect standard data on predation.

Inheritance

class Printer

`Printer(PrinterType)`

Use: Printer P(TYPE)
Pre: None.
Post: P is a printer and its type function will return TYPE – see also Type().

`virtual ~Printer()`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.

$\mathcal{C}$ `virtual void Print(const TimeClass* const) = 0`

Use: P.Print(TimeInfo)
Pre: TimeInfo $\neq 0$.
Post: P has printed its information, using TimeInfo to get the current time.
NB: All derived classes should support to some extent parallelism, i.e. a call to this member function ought to be able to run in the background, given that the state of the objects the information is being printed about does not change during the call.

`PrinterType Type() const`

Use: t = P.Type()
Pre: None.
Post: t is the type of P.
NB: This function is to be used for run-time type identification of derived classes.

Protected Characteristics

```
ActionAtTimes       aat                                // Keeps printing times.
```

Private Characteristics

```
const PrinterType   type                     // The type of Printer.
```

12.3 StockStdPrinter

The class StockStdPrinter implements the standard printout of stock abundance and predation data.

Inheritance

class StockStdPrinter : `public` Printer

Public messages

`StockStdPrinter(CommentStream&, const AreaClass* const,`
`const TimeClass* const)`
Use: StockStdPrinter P(infile, Area, TimeInfo)

`virtual ~StockStdPrinter()`
Use: ~P()

`void SetStock(Stockptrvector&)`
Use: P.SetStock(stockvec)
Pre: stockvec contains unique nonnull pointers. P.SetStock has not been called before. stockvec contains a pointer to a Stock having the name P read from file in the constructor.
Post: P has kept a pointer to its stock and is ready to print. stockvec has not changed.

C `virtual void Print(const TimeClass* const)`
Use: P.Print(TimeInfo)
NB: Refer to the documentation of the base class, Printer.

Protected Characteristics

`char*`	`stockname`	//
`LengthGroupDivision*`	`LgrpDiv`	//
`intvector`	`outerareas`	//
`intvector`	`areas`	//
`int`	`minage`	//
`StockAggregator*`	`aggregator`	// Collects abundance data.
`StockPreyStdInfo*`	`preyinfo`	// Collects data on predation.
`ofstream`	`outfile`	//
`double`	`Scale`	//

Details

- LgrpDiv is a copy of the length group division of the Stock.
- preyinfo is 0 if the stock does not eat.
- minage is the minimum age of the stock in question.

12.4 PredPreyStdPrinter

The classes derived from PredPreyStdPrinter handle the standard printout for predation. The predation can be printed by length or age.

Inheritance

class PredPreyStdPrinter : `public` Printer

Public messages

`PredPreyStdPrinter(CommentStream&, const AreaClass* const, const TimeClass* const)`

Use: PredPreyStdPrinter P(infile, Area, TimeInfo)
Pre: infile has no error bits set and its format is correct. Area $\neq 0$, TimeInfo $\neq 0$.
Post: P is of type PredPreyStdPrinter and has read its information from infile.
NB: P is not fully initialized until the member function SetStocksAndPredAndPrey has been called.

`virtual ~PredPreyStdPrinter()`

Use: ~P
Pre: None.
Post: All memory belonging to P has been freed.

`void SetStocksAndPredAndPrey(const Stockptrvector&, const PopPredatorptrvector&, const Preyptrvector&)`

Use: P.SetStocksAndPredAndPrey(stockvec, predvec, preyvec)
Pre: None of the vectors contains null pointers. The objects in preyvec and predvec may not be one of the preys and predators in stockvec. The vectors contain pointers to the predator and prey about which P is to print information.
Post: P has kept pointers to the predator and prey it is associated with.
NB: The lifetime of P's predator and prey has at least to equal that of the last call to P.Print.

Protected messages

`virtual void P.SetPredAndPrey(const PopPredator*, const Prey*, int, int) = 0`

Use: P.SetPredAndPrey(predator, prey, IsStockPredator, IsStockPrey)
Pre: predator $\neq 0$, prey $\neq 0$. If IsStockPredator is 1, predator is of type StockPredator. If IsStockprey is 1, prey is of type StockPrey.
Post: P is fully initialized.
NB: This virtual function is called from within P.SetStocksAndPredAndPrey.

Protected Characteristics

`char*`	`predname`	// The name of the predator
`char*`	`preyname`	// The name of the prey
`ofstream`	`outfile`	// To which output will be written
`intvector`	`outerareas`	// Outer areas
`intvector`	`areas`	// Inner areas

12.5 PredPreyStdLengthPrinter

Inheritance

class PredPreyStdLengthPrinter : `public` PredPreyStdPrinter

Public messages

`PredPreyStdLengthPrinter(CommentStream&, const AreaClass* const, const TimeClass* const)`

Use: PredPreyStdLengthPrinter P(infile, Area, TimeInfo)
Pre:
Post:

`~PredPreyStdLengthPrinter()`

Use: ~P

$\mathcal{C}$ `virtual void Print(const TimeClass* const)`

Use: Print(TimeInfo)

Protected messages

`virtual void P.SetPredAndPrey(const PopPredator*, const Prey*, int, int)`
Use: P.SetPredAndPrey(predator, prey, IsStockPredator, IsStockPrey)
NB: Works as defined in the documentation of PredPreyStdPrinter.

Protected Characteristics

`PredStdInfoByLength*`	`predinfo`	// Computes the predation
`const PopPredator*`	`predator`	//
`const Prey*`	`prey`	//

12.6 PredPreyStdAgePrinter

Inheritance

class PredPreyStdAgePrinter : `public` PredPreyStdPrinter

Public messages

`PredPreyStdAgePrinter(CommentStream&, const AreaClass* const, const TimeClass* const)`
Use: PredPreyStdAgePrinter P(infile, Area, TimeInfo)
Pre:
Post:

`~PredPreyStdAgePrinter()`
Use: ~P

$\mathcal{C}$ `virtual void Print(const TimeClass* const)`
Use: Print(TimeInfo)

Protected messages

`virtual void P.SetPredAndPrey(const PopPredator*, const Prey*, int, int)`
Use: P.SetPredAndPrey(predator, prey, IsStockPredator, IsStockPrey)
NB: Works as defined in the documentation of PredPreyStdPrinter.

Protected Characteristics

```
AbstrPredStdInfo*   predinfo                  // Computes the predation information.
const PopPredator*  predator                  //
const Prey*         prey                      //
```

12.7 StockFullPrinter

The class StockFullPrinter implements the "not-completely-standardized" full printout of stock abundance data.

Inheritance

class StockFullPrinter : `public` Printer

Public messages

`StockFullPrinter(CommentStream&, const AreaClass* const, const TimeClass* const)`

Use: StockFullPrinter P(*infile*, *Area*, *TimeInfo*)
Pre: *infile* has no error bits set and its file format is correct $Area \neq 0$, $TimeInfo \neq 0$.
Post: P has read its information from *infile*.

`virtual ~StockFullPrinter()`

Use: ~P()
Pre: None.
Post: All memory belonging to P has been freed. P has closed its output files.

`void SetStock(Stockptrvector&)`

Use: P.SetStock(*stockvec*)
Pre: *stockvec*.Size() > 0, *stockvec* contains unique nonnull pointers and *stockvec* contains pointers to the stock P read from *infile*.
Post: P has associated itself with the stock it read from *infile*.

$\mathcal{C}$ `virtual void Print(const TimeClass* const)`

Use: P.Print(*TimeInfo*)
Pre: $TimeInfo \neq 0$.

Post: P has collected information from its stock and written it to its output file.

Protected Characteristics

`intvector`	`areas`	//
`intvector`	`outerareas`	//
`intvector`	`ages`	//
`int`	`minage`	// Minimum age of stock
`char*`	`stockname`	// Name of stock.
`StockAggregator*`	`aggregator`	// Collects information.
`LengthGroupDivision*`	`LgrpDiv`	// The lgrpdiv of the stock.
`ofstream`	`outfile`	// File for output data.

Details

- P collects information for the areas in P.areas and the ages in P.ages.
- outerareas[i] is the outerarea number for area[i].

12.8 AbstrPreyStdInfoByLength

The class AbstrPreyStdInfoByLength is an abstract front end to classes collecting information on the predation on a prey. It defines the data structures they should use and provides public access functions to them.

Inheritance

class AbstrPreyStdInfoByLength : `protected` LivesOnAreas
The class contains only access functions for the effects of predation on length groups. They return doublevectors, indexed with no. of length group, l, where $0 \leq l < prey \rightarrow$NoLengthGroups(). If numbers eaten can not be calculated, 0 should be returned in the corresponding vector and 0 or MAX_MORTALITY if the mortality induced by the predation cannot be calculated.

Public messages

`AbstrPreyStdInfoByLength(const Prey* p, const intvector&)`
Use: AbstrPreyStdInfoByLength A(*prey*, *areas*)
Pre: $prey \neq 0$ and *prey* occupies the areas in *areas*

Post: A is ready to collect information from *prey* on the effects of the predation on its length groups on areas *areas*.

`virtual ~AbstrPreyStdInfoByLength()`
Use: ~A

$\mathcal{AC}$ `const doublevector& NconsumptionByLength(int) const`
Use: A.NconsumptionByLength(*area*)

$\mathcal{AC}$ `const doublevector& BconsumptionByLength(int) const`
Use: A.BconsumptionByLength(*area*)

$\mathcal{AC}$ `const doublevector& MortalityByLength(int) const`
Use: A.MortalityByLength(*area*)

`virtual void Sum(const TimeClass* const, int) = 0`
Use: A.Sum(*TimeInfo*, *area*)
Pre: *TimeInfo* $\neq 0$, A calculates information for the area *area*. *prey* has fully calculated its predation on the current time step.
Post: A has accessed information from *prey* on the predation on it on the area.

Protected Characteristics

`doublematrix`	`MortbyLength`	// [area][length group]
`doublematrix`	`NconbyLength`	// [area][length group]
`doublematrix`	`BconbyLength`	// [area][length group]

12.9 PreyStdInfoByLength

The class PreyStdInfoByLength accesses the class Prey and retrieves information from it on predation and calculates consumption in biomass and the mortality induced. Since it cannot calculate the predation in numbers, 0 is returned.
The length group division PreyStdInfoByLength uses is that of Prey's.

Inheritance

class PreyStdInfoByLength : `protected` LivesOnAreas

Public messages

`PreyStdInfoByLength(const Prey*, const intvector&)`

Use: PreyStdInfoByLength P(prey, areas)
Pre:
Post:
NB: In all the subsequent calls to P where area is a parameter, it has to equal one of the elements of areas.

`virtual ~PreyStdInfoByLength()`

Use: ~P
Pre:
Post:

Note: In the access functions, the length of the vectors equals the number of length groups in prey and they return the values computed from the data received from prey in the last call to P.Sum for that area.

$\mathcal{AC}$ `const doublevector& NconsumptionByLength(int) const`

Use: bm = P.NconsumptionByLength(area)
Pre:
Post: bm is a reference to a vector containing only 0.

$\mathcal{AC}$ `const doublevector& BconsumptionByLength(int) const`

Use: bm = P.BconsumptionByLength(area)
Pre:
Post: bm is a reference to a vector that keeps the consumption in biomass units of prey, by length.

$\mathcal{AC}$ `const doublevector& MortalityByLength(int) const`

Use: bm = P.MortalityByLength(area)
Pre:
Post: bm contains the mortality the predation on prey induced on it calculated on a year's basis, by length.

$\mathcal{AC}$ `virtual void Sum(const TimeClass* const, int)`

Use: P.Sum(TimeInfo, area)
Pre: All predation on prey on the current time step has taken place.
Post: P has collected information from prey on the predation on the area aera on the current time step.

Protected Characteristics

`doublematrix`	`MortbyLength`	// [area][length group]
`doublematrix`	`NconbyLength`	// [area][length group]
`doublematrix`	`BconbyLength`	// [area][length group]

Private Characteristics

`const Prey*`	`prey`	//

12.10 StockPreyStdInfoByLength

+The class StockPreyStdInfoByLength accesses the class StockPrey, retrieves information from it on predation and abundance and calculates consumption in numbers, biomass and the mortality induced.
The length group division StockPreyStdInfo uses is that of StockPrey's.

Inheritance

class StockPreyStdInfoByLength : `protected` LivesOnAreas

Public messages

`StockPreyStdInfoByLength(const StockPrey*, const intvector&)`

Use: *SPBL*(*prey*, *areas*)
Pre: *prey* $\neq 0$ and *areas* is a nonempty vector containing unique nonnegative integers. The prey *prey* exists on all the areas *areas*.
Post: *SPBL* is of type StockPreyStdInfoByLength and is set to collect information from the StockPrey *prey* on the areas in *areas*.
NB: When calling the member functions of *SPBL*, the parameter area has to equal one of the elements of *areas*.

`virtual ~StockPreyStdInfoByLength()`

Use: *~SPBL*()
Pre: None.
Post: All memory belonging to *SPBL* has been freed.

$\mathcal{AC}$ `const doublevector& NconsumptionByLength(int) const`

Use: dv = *SPBL*.NconsumptionByLength(*area*)
Pre: See the Sum member function and the constructor.
Post: dv[l] is the predation in numbers of length group l of *prey* on the area *area*.

$\mathcal{AC}$ `const doublevector& BconsumptionByLength(int) const`

Use: dv = *SPBL*.BconsumptionByLength(*area*)
Pre: Refer to the member function Sum and the constructor.
Post: dv[l] is the predation in biomass of length group l of *prey* on the area *area*.

$\mathcal{AC}$ `const doublevector& MortalityByLength(int) const`

Use: dv = *SPBL*.MortalityByLength(*area*)
Pre: Refer to the member function Sum and the constructor.
Post: dv[l] is the mortality the predation on *prey* induced on length group l of *prey* on the area *area*.

$\mathcal{AC}$ `void Sum(const TimeClass* const, int)`

Use: *SPBL*.Sum(TimeInfo, *area*)
Pre: *SPBL* is set to collect information on the area *area*. All the predation on the prey *prey* on the current time step has taken place.
Post: *SPBL* has collected information from *prey* on the predation on the area *area* on the current time step.

Protected Characteristics

`const StockPrey*`	`prey`	//

Private Characteristics

`doublematrix`	`MortbyLength`	// [area][length group]
`doublematrix`	`NconbyLength`	// [area][length group]
`doublematrix`	`BconbyLength`	// [area][length group]

12.11 PredStdInfoByLength

Inheritance

class PredStdInfoByLength : `protected` LivesOnAreas

Public messages

`PredStdInfoByLength(const PopPredator*, const Prey*, const intvector&)`

Use: PredStdInfoByLength P(predator, prey, areas)

Pre: predator $\neq 0$, prey $\neq 0$, areas is a nonempty vector containing unique nonnegative elements and predator and prey are defined on the areas areas.
Post: P is of type PredStdInfoBylength and can collect information on the areas areas on predator's predation on prey, by length group of predator and prey.

`PredStdInfoByLength(const PopPredator*, const StockPrey*, const intvector&)`
Use: PredStdInfoByLength P(predator, prey, areas)
Pre: The same as above.
Post: Ditto.
NB: The difference between the two constructors.

`virtual ~PredStdInfoByLength()`
Use: ~P

$\mathcal{AC}$ `const bandmatrix& NconsumptionByLength(int) const`
Use: bm = P.NconsumptionByLength(area)
Pre: P collects information for the area area and P.Sum has been called for the area area.
Post:
- bm is a reference to a bandmatrix holding the predation in numbers of P's predator on P's prey in the area area.
- bm is indexed with [length group of predator][length group of prey].
- bm.Minage() equals 0, do does bm.Minlength.

NB: The effect of the constructors.

$\mathcal{AC}$ `const bandmatrix& BconsumptionByLength(int) const`
Use: bm = P.BconsumptionByLength(area)
Pre:
Post: bm holds the predation in biomass.

$\mathcal{AC}$ `const bandmatrix& MortalityByLength(int) const`
Use: bm = P.MortalityByLength(area)
Pre:
Post: bm holds the mortality the predation of P's predator on P's prey induced on it, converted to the time scale of one year.

$\mathcal{AC}$ `virtual void Sum(const TimeClass* const, int)`
Use: P.Sum(TimeInfo, area)
Pre: TimeInfo $\neq 0$ and P is defined on the area area.

Post: P has collected information from its predator and prey for the area area.

Protected messages

$\mathcal{S}$ `void AdjustObjects()`

Use: P.AdjustObjects()
Pre: P.AdjustObjects has not been called before.
Post: P has adjusted the size of its objects.

Private Characteristics

```
PreyStdInfoByLength*preyinfo             // computed info for the prey.
const PopPredator*  predator             //
const Prey*         prey                 //
bandmatrixvector    MortbyLength         // [area][pred l][prey l]
bandmatrixvector    NconbyLength         // [area][pred l][prey l]
bandmatrixvector    BconbyLength         // [area][pred l][prey l]
```

12.12 AbstrPreyStdInfo

The class AbstrPreyStdInfo is an abstract base class that defines how to obtain information on the amount and effects of the predation on a prey.
It should be possible e.g. to obtain the biomass eaten of each age group of a prey.
The class defines the data structures that derived classes should use and public access functions to them.
The naming convention of the member functions is very simple, all those member functions whose name begins with 'N' return the number consumed, those that begin with 'B' return the biomass consumed and those with 'Mort' return the mortality induced by the predation, converted to the time scale of a year .
Those member functions whose name ends with 'AgeAndLength' return the data by age- and length group, indexed in that order, if they end with 'Age' the data is by age and by length group if it ends with 'Length', $0 \leq l <$*prey*→NoLengthGroups() and minage $\leq$ age $<$ maxage.

Inheritance

class AbstrPreyStdInfo : `protected` LivesOnAreas

Public messages

`AbstrPreyStdInfo(const Prey*, const intvector&, int = 0, int = 0)`

Use: AbstrPreyStdInfo A(prey, *areas*, minage, maxage)
Pre: prey $\neq 0$, $0 \leq$ minage $\leq$ maxage.
Post: A is of thype AbstrPreyStdInfo, ready to work onthe areas *areas*.
NB: Programmers of derived classes should look at the effects of the parameters on the protected variables.

`virtual ~AbstrPreyStdInfo()`

Use: ~A

$\mathcal{AC}$ `virtual const doublevector& NconsumptionByLength(int) const = 0`

Use: dv = A.NconsumptionByLength(*area*)

$\mathcal{AC}$ `virtual const doublevector& BconsumptionByLength(int) const = 0`

Use: dv = A.BconsumptionByLength(*area*)

$\mathcal{AC}$ `virtual const doublevector& MortalityByLength(int) const = 0`

Use: dv = A.MortalityByLength(*area*)

$\mathcal{AC}$ `const bandmatrix& NconsumptionByAgeAndLength(int) const`

Use: bm = A.NconsumptionByAgeAndLength(*area*)

$\mathcal{AC}$ `const doubleindexvector& NconsumptionByAge(int) const`

Use: div = A.NconsumptionByAge(*area*)

$\mathcal{AC}$ `const bandmatrix& BconsumptionByAgeAndLength(int) const`

Use: bm = A.BconsumptionByAgeAndLength(*area*)

$\mathcal{AC}$ `const doubleindexvector& BconsumptionByAge(int) const`

Use: div = A.BconsumptionByAge(*area*)

$\mathcal{AC}$ `const bandmatrix& MortalityByAgeAndLength(int) const`

Use: bm = A.MortalityByAgeAndLength(*area*)

$\mathcal{AC}$ `const doubleindexvector& MortalityByAge(int) const`

Use: div = A.MortalityByAge(*area*)

`virtual void Sum(const TimeClass* const, int) = 0`

Use: A.Sum(*TimeInfo*, *area*)
Pre: *TimeInfo* $\neq 0$ and A collects information for area *area*.
Post: A has collected and calculated information for the area *area* and will return that data in the calls to the access functions [until the next call to A.Sum on that area].

Protected Characteristics

`bandmatrix`	`NconbyAge`	// [area][age]
`bandmatrix`	`BconbyAge`	// [area][age]
`bandmatrix`	`MortbyAge`	// [area][age]
`bandmatrixvector`	`NconbyAgeAndLength`	// [area][age][l.group]
`bandmatrixvector`	`BconbyAgeAndLength`	// [area][age][l.group]
`bandmatrixvector`	`MortbyAgeAndLength`	// [area][age][l.group]

12.13 PreyStdInfo

Inheritance

class PreyStdInfo : `public` AbstrPreyStdInfo

`PreyStdInfo(const Prey*, const intvector&)`
Use: PreyStdInfo P(prey, areas)
Pre:
Post:

`virtual ~PreyStdInfo()`
Use: ~P()
Pre:
Post:

$\mathcal{AC}$ `virtual const doublevector& NconsumptionByLength(int) const`
Use: dv = P.NconsumptionByLength(area)
Post: dv is a vector containing only zeroes.

$\mathcal{AC}$ `virtual const doublevector& BconsumptionByLength(int) const`
Use: dv = P.BconsumptionByLength(area)

$\mathcal{AC}$ `virtual const doublevector& MortalityByLength(int) const`

Use: dv = P.MortalityByLength(area)

$\mathcal{AC}$ `virtual void Sum(const TimeClass* const, int)`
Use: P.Sum(*TimeInfo*, *area*)

Private Characteristics

```
PreyStdInfoByLength PSIByLength            //
const Prey*         prey                   //
```

12.14 StockPreyStdInfo

The class StockPreyStdInfo communicates with an instance of the class StockPrey, receives data on the predation on it and abundance numbers and uses these to distribute the predation (initially only given by length) on age groups and calculates the mortalities induced.
The length group division StockPreyStdInfo uses is that of StockPrey's.

Inheritance

class StockPreyStdInfo : `public` AbstrPreyStdInfo

Public messages

`StockPreyStdInfo(const StockPrey*, int, int, const intvector&)`
Use: StockPreyStdInfo *SPSI*(*prey*, *minage*, *maxage*, *areas*)
Pre: *prey* $\neq 0$, *minage* $\leq$ *maxage* and *areas* is a nonempty vector containing unique nonnegative integers. The StockPrey *prey* exists on the areas in *areas*.
Post: *SPSI* is of type StockPreyStdInfo. It collects information from the prey *prey*, for the age groups *minage* to *maxage*, both included, and for the areas in *areas*.

`virtual ~StockPreyStdInfo()`
Use: ~*SPSI*
Pre: None.
Post: All memory belonging to *SPSI* has been freed.

$\mathcal{AC}$ `virtual const doublevector& NconsumptionByLength(int) const`
Use: dv = P.NconsumptionByLength(area)

$\mathcal{AC}$ `virtual const doublevector& BconsumptionByLength(int) const`

Use: dv = P.BconsumptionByLength(area)

$\mathcal{AC}$ `virtual const doublevector& MortalityByLength(int) const`

Use: dv = P.MortalityByLength(area)

$\mathcal{AC}$ `void Sum(const TimeClass* const, int)`

Use: *SPSI*.Sum(*TimeInfo*, *area*)
Pre: *TimeInfo* $\neq 0$, *SPSI* is set to collect information for the area *area*. All the predation on the prey *prey* has taken place for the current time step.
Post: *SPSI* has collected information on the predation on *prey* on the area *area*.

Private Characteristics

```
PreyStdInfoByLength PSIByLength              //
const Prey*         prey                     //
```

12.15 AbstrPredStdInfo

The class AbstrPredStdInfo is an abstract base class that defines how to obtain information on the amount and effects of the predation of a predator on a prey.
It should be possible e.g. to obtain the biomass eaten by each age group of a predator of each age group of a prey.

Inheritance

class AbstrPredStdInfo : `protected` LivesOnAreas

Public messages

`AbstrPredStdInfo(const intvector&, int = 0, int = 0, int = 0, int = 0)`

Use: AbstrPredStdInfo P(*areas*, predminage, predmaxage, preyminage, preymaxage)
Pre: $0 \leq$ predminage $<$ predmaxage and $0 \leq$ preyminage $<$ preymaxage.
Post: P is of type AbstrPredStdInfo, ready to work on the areas *areas*.

NB: Programmers of derived classes should look at the effects of the parameters on the protected variables.

`virtual ~AbstrPredStdInfo()`

Use: ~A()

Note:

- If it is not possible to calculate the data to put in the matrices, 0 should be returned for the numbers and biomass and either 0 or MAX_MORTALITY for the mortality.
- The following restrictions are put on the matrices for data by length:
  - They are indexed with [length group of predator][length group of prey].
  - They have Minrow() equal to 0 and Mincol(predlengthgroup) equal to 0.
- The following restrictions are put on the matrices for data by age:
  - They are indexed with [age group of predator][age group of prey], meaning they are shifted, i.e. Minrow() does not necessarily have to equal 0, nor does Mincol(predage).
  - If it is not possible to divide by age of predator the data for the predator is presented as if it only had one agegroup, age 0. The same applies when it is not possible to divide by age of prey.

`virtual const bandmatrix& NconsumptionByLength(int) const = 0`

Use: bm = P.NconsumptionByLength(area)

Pre:

Post: bm contains the consumption in numbers on the area area, by length.

`virtual const bandmatrix& BconsumptionByLength(int) const = 0`

Use: bm = P.BconsumptionByLength(area)

Pre:

Post: bm contains the consumption in biomass units on the area area, by length.

`virtual const bandmatrix& MortalityByLength(int) const = 0`

Use: bm = P.MortalityByLength(area)

Pre:

Post: bm contains the mortality the predation of P's predator induced on P's prey, by length.

$\mathcal{AC}$ `const bandmatrix& NconsumptionByAge(int) const`

Use: bm = P.NconsumptionByAge(area)
Pre:
Post: bm contains the consumption in numbers by age.

$\mathcal{AC}$ `const bandmatrix& BconsumptionByAge(int) const`

Use: bm = P.BconsumptionByAge(area)
Pre:
Post: Consumption in biomass units, by age.

$\mathcal{AC}$ `const bandmatrix& MortalityByAge(int) const`

Use: MortalityByAge(area)
Pre:
Post: Mortality the predation induced, by age. The mortality is given on a year's scale.

`virtual void Sum(const TimeClass* const, int) = 0`

Use: P.Sum(TimeInfo, area)
Pre:
Post:

Protected Characteristics

```
bandmatrixvector   NconbyAge           // [area][pred. age][prey age]
bandmatrixvector   BconbyAge           // [area][pred. age][prey age]
bandmatrixvector   MortbyAge           // [area][pred. age][prey age]
```

Details

The constructor initializes the bandmatrixvector to be of length *areas*.Size(), its matrices to be initialized to zero and

Column	equals
Minrow	predminage
Maxrow	predmaxage
Mincol	preyminage
Maxcol	preymaxage + 1

12.16 PredStdInfo

The classes PredStdInfoByLength and PredStdInfo compute how the predation of a predator on a prey is divided by length and age groups.
Note that this class is only conserned with the predator being of type PopPredator which is not divided into age groups and therefore the output is not divided by age of predator. It is divided by age of prey if possible.

Inheritance

class PredStdInfo : `public` AbstrPredStdInfo

Public messages

`PredStdInfo(const PopPredator*, const Prey*, const intvector&)`

Use: PredStdInfo P(predator, prey, *areas*)

Pre: *areas* is a nonempty vector containing unique nonnegative integers and predator and prey live on all the areas in *areas*.

Post: P is ready to collect information on the the predation of predator on the prey prey on the areas in *areas*. The ages of predator and prey are set to 0 in the member functions returning data on age dependent eating.

`PredStdInfo(const PopPredator*, const StockPrey*, const intvector&)`

Use: PredStdInfo P(predator, prey, areas)

Pre: The same as for the other constructor.

Post: Similar to the other constructor, except that now the prey can be divided into agegroups, but the predator cannot.

`virtual ~PredStdInfo()`

Use: ~P

$\mathcal{AC}$ `virtual const bandmatrix& NconsumptionByLength(int) const`

Use: bm = P.NconsumptionByLength(area)

$\mathcal{AC}$ `virtual const bandmatrix& BconsumptionByLength(int) const`

Use: bm = P.BconsumptionByLength(area)

$\mathcal{AC}$ `virtual const bandmatrix& MortalityByLength(int) const`

Use: bm = P.MortalityByLength(area)

$\mathcal{AC}$ `virtual void Sum(const TimeClass* const, int)`
Use: P.Sum(TimeInfo, area)

Private Characteristics

```
PreyStdInfo*           preyinfo          //
PredStdInfoByLength*predinfo             //
const PopPredator*     predator          //
const Prey*            prey              //
```

12.17 StockPredStdInfo

The class StockPredStdInfo calculates the effects of the predation of a predator on its preys.
See also the documentation of AbstrPredStdInfo.
The class restricts itself to the predator being of type StockPredator, meaning that it is divided into age groups, and therefore so is it output. If the prey is of type StockPrey, the output is also divided by age of prey, else we pretend as if the only agegroup of the prey is 0.

Inheritance

class StockPredStdInfo : `public` AbstrPredStdInfo

Public messages

`StockPredStdInfo(const StockPredator*, const Prey*, const intvector&)`
Use: StockPredStdInfo(predator, prey, areas)
Pre:
Post:

`StockPredStdInfo(const StockPredator*, const StockPrey*, const intvector&)`
Use: StockPredStdInfo(predator, prey, areas)
Pre:
Post:

`virtual ~StockPredStdInfo()`
Use: ~P()

$\mathcal{AC}$ `virtual const bandmatrix& NconsumptionByLength(int) const`

Use: bm = P.NconsumptionByLength(area)

$\mathcal{AC}$ `virtual const bandmatrix& BconsumptionByLength(int) const`

Use: bm = P.BconsumptionByLength(area)

$\mathcal{AC}$ `virtual const bandmatrix& MortalityByLength(int) const`

Use: bm = P.MortalityByLength(area)

$\mathcal{AC}$ `virtual void Sum(const TimeClass* const, int)`

Use: P.Sum(TimeInfo, area)

Private Characteristics

`PreyStdInfo*`	`preyinfo`	// Computes the cons. of prey
`PredStdInfoByLength*`	`predinfo`	// For the predator's consumption.
`const StockPredator*`	`predator`	//
`const Prey*`	`prey`	//

12.18 Details of computations.

In this section we describe the details of the simple computations done in StockPreyStdInfo and StockPredStdInfo.
In what follows, we let the predator and prey be fixed, and restrict out attention entirely to one length group of predator and one length group of prey. These variables will be used:

A	age group of predator
a	age group of prey
n_a	number in age group a of prey prior to predation
w_a	mean weight in age group a of prey
${}_tn_a$	total number eaten of age group a of prey
${}_tb_a$	total biomass eaten of age group a of prey
${}_tb_.$	total biomass eaten of prey
${}_tn_.$	total number eaten of prey
${}_pn_a$	number eaten of age group a of prey by the (fixed) predator
${}_pb_a$	biomass eaten of age group a of prey by the (fixed) predator
${}_pn_.$	total number eaten of the prey by the predator
${}_pb_.$	total biomass eaten of the prey by the predator
p_A	the proportion of the predation accounted for by age group A.

For all predators, the mean weight of the prey eaten from a length group is the same, equal to the mean weight of the prey in that length group. Therefore

$$\frac{{}_tb_.}{{}_tn_.} = \frac{{}_pb_.}{{}_pn_.} = \frac{\sum_{a'} {}_pb_{a'}}{\sum_{a'} {}_pn_{a'}},$$

so

$$\frac{{}_tb_.}{{}_pb_.} = \frac{{}_tn_.}{{}_pn_.}.$$

And with similar arguments p_A equals both (total number eaten by age group A)/(total number eaten by all age groups) and (total biomass eaten by age group A)/(total biomass eaten by all age groups).
According to the current strategy **(which should be better documented)**, the predation on the fixed length group of the prey is distributed on the age groups in proportion to their abundance numbers.
Since

$${}_tn_. = \frac{{}_tb_.}{\frac{\sum_{a'} n_{a'} w_{a'}}{\sum_{a'} n_{a'}}},$$

we have that

$${}_tn_a = {}_tn_. \frac{n_a}{\sum_{a'} n_{a'}} = n_a \frac{{}_tb_.}{\sum_{a'} n_{a'} w_{a'}}.$$

And clearly the biomass subtracted from each age group is w_a times the above.

This suits our purposes because now

$$\sum_a {}_tn_a w_a = {}_tb_..$$

The predation by age group A on age group a is then

$$p_A \, {}_pb_a = p_{A t}b_a \frac{{}_pb_.}{{}_tb_.},$$

in biomass units, and in numbers it is

$$p_A \, {}_tn_a \frac{{}_pn_.}{{}_tn_.} = p_A \, {}_tn_a \frac{{}_pb_.}{{}_tb_.},$$

because of the note here above.
In the code, the biomass eaten of age group a, b_a is calculated first as

$${}_tb_a := n_a w_a \frac{{}_tb_.}{\sum_{a'} n_{a'} w_{a'}},$$

and then

$${}_tn_a := n_a \frac{{}_tb_.}{\sum_{a'} n_{a'} w_{a'}}.$$

When calculating the predation of a single age group of the predator we let

$$prop := p_A \frac{\sum_{a'} {}_pb_{a'}}{{}_tb_.},$$

and then the biomass eaten of age group a of the prey by age group A of the predator is

$$prop \, {}_tb_a = p_A \frac{\sum_{a'} {}_pb_{a'}}{{}_tb_.} {}_tb_a,$$

which is immediately seen to be correct, and the number eaten is

$$prop \, {}_tn_a = p_A \frac{\sum_{a'} {}_pb_{a'}}{{}_tb_.} {}_tn_a = p_A \frac{\sum_{a'} {}_pn_{a'}}{{}_tn_.} {}_tn_a.$$

Chapter 13

Nonstandard aggregation and printing.

13.1 Overview

In this section, the following classes that write data to files will be documented:

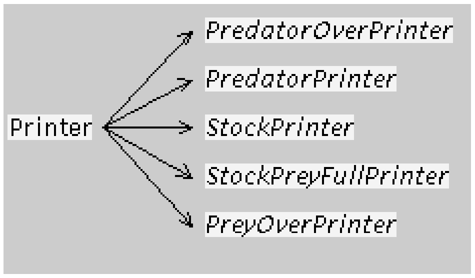

Figure 13.1: Non-standard Printer classes derived from Printer.

13.2 StockPrinter

The class StockPrinter writes abundance numbers to file.

Inheritance

class StockPrinter : `public` Printer

```
StockPrinter(CommentStream&, const AreaClass* const,
const TimeClass* const)
```

Use: StockPrinter SP(*infile*, *Area*, *TimeInfo*)

Pre: *infile* has no badbits set and its format is correct. *Area* and *TimeInfo* are not null pointers.

Post: SP is of type StockPrinter.
NB: SP is not ready to print until SP.SetStock has been called.

`virtual ~StockPrinter()`
Use: ~SP
Pre: None.
Post:

`void SetStock(Stockptrvector&)`
Use: SP.SetStock(*stockvec*)
Pre: *stockvec*.Size() > 0, none of the pointers in *stockvec* is null and it contains pointers to all the objects of type Stock whose name SP keeps.
Post: SP keeps pointers to the Stocks it prints information on. *stockvec* has not been modified.

$\mathcal{C}$ `virtual void Print(const TimeClass* const)`
Use: SP.Print(*TimeInfo*)
Pre: SP.SetStock has been called.
Post: SP has printed information for the current timestep.

Protected Characteristics

`intmatrix`	`areas`	// Areas to aggregate.
`intmatrix`	`ages`	// Ages to aggregate.
`LengthGroupDivision*`	`LgrpDiv`	// Describes the output.
`charptrvector`	`stocknames`	// Names of stocks.
`StockAggregator*`	`aggregator`	// Handles the aggregation.
`ofstream`	`noutfile`	// Output file for abundance numbers.
`ofstream`	`woutfile`	// Output file for mean weights.

outfile is opened in the constructor and closed in the destructor.

13.3 PredatorPrinter

The PredatorPrinter is very much alike StockPrinter. Therefore it will not be described here. It may also be helpful to read about PredatorAggregator.

Inheritance

class PredatorPrinter : `public` Printer

`PredatorPrinter(CommentStream&, const AreaClass* const,`
`const TimeClass* const)`
Use: PredatorPrinter PP(*infile*, *Area*, *TimeInfo*)
NB: PP is not fully initialized until SetPredAndPrey has been called.

`virtual ~PredatorPrinter()`
Use: ~PP

`void SetPredAndPrey(Predatorptrvector&, Preyptrvector&)`
Use: PP.SetPredAndPrey(*predators*, *preys*)

C `virtual void Print(const TimeClass* const)`
Use: PP.Print(*TimeInfo*)
Pre: PP.SetPredAndPrey has been called.

Protected Characteristics

```
intmatrix             areas               // Areas to aggregate.
LengthGroupDivision*predLgrpDiv           // Describes output.
LengthGroupDivision*preyLgrpDiv           // Describes output.
charptrvector         predatornames       // Names of Predators.
charptrvector         preynames;          // Names of Preys.
PredatorAggregator* aggregator            // This one aggregates.
ofstream              outfile             // Where output is written.
```

13.4 PredatorOverPrinter

The class PredatorOverPrinter prints the overconsumption of predators. Look at StockPrinter for pre- and postconditions.

Inheritance

class PredatorOverPrinter : `public` Printer

`PredatorOverPrinter(CommentStream&, const AreaClass* const, const TimeClass* const)`
Use: PredatorOverPrinter POP(*infile*, *Area*, *TimeInfo*)

`virtual ~PredatorOverPrinter()`
Use: ~POP

`void SetPredator(Predatorptrvector&)`
Use: POP.SetPredator(*predators*)

𝒞 `virtual void Print(const TimeClass* const)`
Use: POP.Print(*TimeInfo*)

Protected Characteristics

```
intmatrix                 areas                      // Areas to aggregate.
LengthGroupDivision*predLgrpDiv                      // Describes output.
charptrvector             predatornames              // Names of Predators.
PredatorOverAggregator*aggregator                    // This one aggregates.
ofstream                  outfile                    // Output written to this one.
```

13.5 PreyOverPrinter

The class PreyOverPrinter prints the overconsumption of preyators. Look at StockPrinter for pre- and postconditions.

Inheritance

class PreyOverPrinter : `public` Printer

`PreyOverPrinter(CommentStream&, const AreaClass* const, const TimeClass* const)`
Use: PreyOverPrinter POP(*infile*, *Area*, *TimeInfo*)
NB: POP is not fully initialized until POP.SetPrey has been called.

`virtual ~PreyOverPrinter()`
Use: ~POP

`void SetPrey(Preyptrvector&)`
Use: POP.SetPrey(*preys*)

𝒞 `virtual void Print(const TimeClass* const)`

Use: POP.Print(*TimeInfo*)
Pre: POP.SetPrey has been called.

Protected Characteristics

```
intmatrix             areas              // Areas to aggregate.
LengthGroupDivision*preyLgrpDiv          // Describes output.
charptrvector         preynames          // Names of Preys.
PreyOverAggregator* aggregator           // This one aggregates.
ofstream              outfile            // Output written to this one.
```

13.6 StockPreyFullPrinter

The class StockPreyFullPrinter is useful for debugging. It prints out all the information on the consumption of the prey at the level of most disaggregation.

Inheritance

class StockPreyFullPrinter : `public` Printer

Public messages

`StockPreyFullPrinter(CommentStream&, const AreaClass* const, const TimeClass* const)`

Use: StockPreyFullPrinter P(infile, Area, TimeInfo)
Pre: infile has no error bits set and its format it correct, Area $\neq 0$, TimeInfo $\neq 0$.
Post: P has read information from infile and is ready to print when P.SetStock has been called.

`virtual ~StockPreyFullPrinter()`

Use: ~StockPreyFullPrinter()

`void SetStock(Stockptrvector&)`

Use: P.SetStock(stockvec)
Pre: stockvec is a nonempty vector containing unique nonnull pointers. P.SetStock has not been called before. stockvec contains pointers to Stocks having the names P read from file in the constructor and they are preys.
Post: P keeps pointers to its preys and is ready to print. stockvec has not changed.

$\mathcal{C}$ `virtual void Print(const TimeClass* const)`

Use: P.Print(TimeInfo)

NB: Refer to the documentation of the base class, Printer.

Protected Characteristics

`char*`	`stockname`	//
`LengthGroupDivision *`	`LgrpDiv`	//
`intvector`	`outerareas`	//
`intvector`	`areas`	//
`StockPreyStdInfo*`	`preyinfo`	// Computes the info.
`ofstream`	`outfile`	//

Data invariant

- areas.Size() = outerareas.Size().
- outerareas[i] is the outer area number of the area areas[i].

Details

LgrpDiv is a copy of the length group division of the StockPrey.

13.7 StockAggregator

The StockAggregator receives information in the constructor regarding how to sum up information on the population of Stocks. It will do so when asked to, using a member function, and return the sum using another member function.

Inheritance

class StockAggregator

`StockAggregator(const Stockptrvector&, const LengthGroupDivision* const, const intmatrix&, const intmatrix&)`

Use: StockAggregator SA(*Stocks*, *LgrpDiv*, *Areas*, *Ages*)
Pre: All of the objects are nonzero. *Stocks* contains no 0 pointers. The length groups in *LgrpDiv* have to be unions of the length groups of the stocks in *Stocks*.
Post: SA is ready to aggregate. Its length group division is given with *LgrpDiv*. For each row i in *Areas* and *Ages*, the areas in *Areas*[i] will be combined into one and the age groups *Ages*[i] will be combined into one when summing up the abundance numbers.
NB: The lifetime of the objects pointed to by *Stocks* has of course to exceed that of SA.

`~StockAggregator()`

Use: ~SA
Pre: None.
Post: All memory belonging to SA has been freed.

$\mathcal{C}$ `void Sum()`

Use: SA.Sum()
Pre: None.
Post: SA has summed up the abundance numbers from its stocks.

$\mathcal{C}$ `const Agebandmatrixvector& ReturnSum() const`

Use: aptr = &SA.ReturnSum()
Pre: SA.Sum has been called.
Post: aptr points to an Agebandmatrixvector containing the result of the last call to SA.Sum(). In (*aptr)[i] we have an rectangular Agebandmatrix containing the sum over the areas *Areas*[i], in (*aptr)[i][j] we the sum over the age groups *Ages*[j] and the division of that vector into length groups is given with *LgrpDiv*.
The Agebandmatrix (*aptr)[i] has *Ages*.Nrow() lines, *LgrpDiv*→NoLengthGroups() columns, its Minage() is equal to 0 and so is its Minlength().
NB: The objects *Areas*, *Ages* and *LgrpDiv* mentioned here are those in the call to the constructor when SA was created.

Protected Characteristics

`Stockptrvector`	`stocks`	// Pointers to the stocks.
`ConversionIndexptrvector`	`CI`	// Conversion from stocks to total.
`intmatrix`	`areas`	// Areas to aggregate.
`intmatrix`	`ages`	// Ages to aggregate
`intvector`	`AreaNr`	// For quicker access.
`intvector`	`AgeNr`	// For quicker access.
`Agebandmatrixvector`	`total`	// Keeps the sum.

Details

The ConversionIndex CI[i] converts from the length group division of stocks[i] to total, i = 0, . . . , stocks.Size() - 1

13.8 PredatorAggregator

The class PredatorAggregator is very similar to StockAggregator. Its member functions serve the same purpose in a new situation. In the case of the StockAggregator we are concerned about abundance figures, but in this case out interest is the amount predators eat of their preys, divided into length groups of predators and prey.

Inheritance

class PredatorAggregator

Public messages

```
PredatorAggregator(const Predatorptrvector&,
const Preyptrvector&, const intmatrix&, const
LengthGroupDivision* const, const LengthGroupDivision*
const)
```

Use: PredatorAggregator PA(*predators*, *preys*, *Areas*, *predLgrpDiv*, *preyLgrpDiv*)

Pre: None of the objects is null and the vectors *predators* and *preys* contain no null pointers. The length of each row of *Areas* is greater than zero.The each length group in *predLgrpDiv* should be a union of length groups in *predators*[.]; likewise for *preyLgrpDiv* and *preys*[.].

Post: PA is ready to aggregate consumption numbers from the predators in *predators*, regarding their consumption of the preys in *preys*. See also description of ReturnSum and the following discussion for further explanations of the parameters..

$\mathcal{C}$ `void Sum()`

Use: PA.Sum()

Pre: None.

Post: PA has summed up the consumption numbers from its predators.

$\mathcal{C}$ `const bandmatrixvector& ReturnSum() const`

Use: bmvptr = &PA.ReturnSum() const

Pre: PA.Sum has been called.

Post: bmvptr points to a bandmatrixvector. In (*bmvptr)[i] we have the sum over the areas *Areas*[i]. The matrix (*bmvptr)[i] is indexed with [predLength][preyLength], where predLength is a length group of *predLgrpDiv* and preyLength a length group of *preyLgrpDiv* (see the constructor).
Each of the bandmatrices in bmvptr has Minage equal to 0.

The postconditions of ReturnSum may be put somewhat clearer. In (*bmvptr)[i][j][k] we have:

$$\sum_{a \in Areas[i]} \sum_{pred} \sum_{prey} \sum_{L \in predLgrpDiv(j)} \sum_{l \in preyLgrpDiv(k)} Consumption_{pred,a}(l, L, prey)$$

where $Consumption_{pred,a}(l, L, prey)$ is the consumption of length group L of the predator *pred* of length group l of *prey* on the area a, and

$$\sum_{L \in predLgrpDiv(j)}$$

is the sum over all length groups L of the predator that are within the length group j of *predLgrpDiv* (see the constructor). Likewise for

$$\sum_{l \in preyLgrpDiv(k)} .$$

Protected Characteristics

```
Predatorptrvector   predators          //
Preyptrvector       preys              //
intmatrix           predConv;          // [predator][predatorLengthGroup]
intmatrix           preyConv;          // [prey][preyLengthGroup]
intmatrix           areas              //
intvector           AreaNr             //
intmatrix           doeseat;           // [predator][prey] – does predator eat prey?
bandmatrixvector    total              // [area][pred.LengthGr.][preyLengthGr.]
```

Details

predConv[p] is meant for converting from the length group division of predators[p] to the length group division according to which total is. If predConv[p][l] is ≥ 0, predConv[p][l] is the number of length group in total to which length group l of the predator predators[p] belongs. If predConv[p][l] < 0, length group l of predators[p] falls out of the range of total.
The same applies for preyConv.

13.9 PredatorOverAggregator

The class PredatorOverAggregator is just another aggregator class. Its use and meaning of parameters should be very clear after reading about the classes PredatorAggregator and StockAggregator. It aggregates the overconsumption of predators.

Inheritance

class PredatorOverAggregator

`PredatorOverAggregator(const Predatorptrvector&, const intmatrix&, const LengthGroupDivision* const)`

Use: PredatorOverAggregator POA(*predators*, *Areas*, *predLgrpDiv*)
Pre: See PredatorAggregator.
Post: POA is ready to aggregate overconsumption data of the predators *predators*. See also PredatorAggregator.

$\mathcal{C}$ `void Sum()`

Use: POA.Sum()
Pre: See PredatorAggregator.
Post: See PredatorAggregator.

$\mathcal{C}$ `const doublematrix& ReturnSum() const`

Use: dmptr = &POA.ReturnSum()
Pre: See PredatorAggregator.
Post: See PredatorAggregator. Note however that the matrix pointed to by dmptr is indexed with [area][lengthgroup], where area is the aggregation of the areas in *Areas*[areas] and lengthgroup is a length group in *predLgrpDiv* – see the constructor and PredatorAggregator.

Protected Characteristics

```
Predatorptrvector   predators        // Pointers to the predators.
intmatrix           predConv         // [pred][predLengthGroup] – faster access.
intmatrix           areas            // How areas are aggregated.
doublematrix        total            // [area][predLgrp] – keeps the last sum.
```

Details

predConv[p] is meant for converting from the length group division of predators[p] to the length group division according to which total is. If predConv[p][l] is ≥ 0, predConv[p][l] is the number of length group in total to which length group l of the predator predators[p] belongs. If predConv[p][l] $<$ 0, length group l of predators[p] falls out of the range of total.

13.10 PreyOverAggregator

The class PreyOverAggregator is almost just like PredAggregator. It aggregates the overconsumption of preys. Read the description of PredatorAggregator for a better description of the parameters and meaning of the functions.

Inheritance

class PreyOverAggregator

`PreyOverAggregator(const Preyptrvector&, const intmatrix&, const LengthGroupDivision* const)`

Use: PreyOverAggregator(*preys*, *Areas*, *preyLgrpDiv*)
Pre: See PredatorAggregator.
Post: See PredatorAggregator.

$\mathcal{C}$ `void Sum()`

Use: POA.Sum()
Pre: See PredatorAggregator.
Post: See PredatorAggregator.

$\mathcal{C}$ `const doublematrix& ReturnSum() const`

Use: dmptr = &POA.ReturnSum()
Pre: See PredatorAggregator.
Post: See PredatorAggregator — dmptr is indexed with [area][preylength].

Protected Characteristics

`Preyptrvector`	`preys`	// Pointers to the preys.
`intmatrix`	`preyConv`	// [prey][preyLengthGroup] – fast access.
`intmatrix`	`areas`	// areas to aggregate.
`doublematrix`	`total`	// [area][length group] – the last sum.

Details

preyConv[p] is meant for converting from the length group division of preys[p] to the length group division according to which total is. If preyConv[p][l] is ≥ 0, preyConv[p][l] is the number of length group in total to which length group l of the prey preys[p] belongs. If preyConv[p][l] < 0, length group l of preys[p] falls out of the range of total.

Chapter 14

Statistics.

14.1 LinearRegression

The class LinearRegression fits a straigt line through a set of given points (x, y), i.e. finds a and b to minimize

$$\sum_i (y_i - (a + bx_i))^2.$$

If the x_i-s are all the same, the fitted line is $a = \bar{y}$, $b = 0$.

Inheritance

class LinearRegression

Public messages

`LinearRegression()`
Use: LinearRegression LR
Pre: None.
Post: LR is of type LinearRegression.

$\mathcal{S}$ `virtual void Fit(const doublevector&, const doublevector&)`
Use: LR.Fit(x, y)
Pre: x.Size() $= y$.Size() > 0.
Post: LR has fitted a line through the points $(x[i], y[i])$ – a least squares fit. See the discussion above.
If the preconditions are not met, LR.Error() returns 1, else 0.

$\mathcal{S}$ `virtual void Fit(const doublevector&, const doublevector&, double)`
Use: LR.Fit(x, y, b)

Pre: x.Size() = y.Size() > 0.
Post: LR has fitted a line whose slope is b through the points $(x[i], y[i])$. See the discussion above. If the preconditions are not met, LR.Error() returns 1, else 0.

$\mathcal{S}$ `virtual void Fit(const doublevector&, const doublevector&, double, double)`
Use: LR.Fit(x, y, b, a)
Pre: x.Size() = y.Size() > 0.
Post: LR has set its line to have the slope b and the intercept a. If the preconditions are not met, LR.Error() returns 1, else 0.

$\mathcal{S}$ `virtual double Funcval(double)`
Use: y = LR.Funcval(x)
Pre: LR.Fit(...) has been called and LR.Error() returns 0.
Post: y is the value of the line at the point x.

$\mathcal{F}$ `int Error() const`
Use: *err* = LR.Error()
Pre: None.
Post: *err* is 1 if an error occurred in the last call to LR.Fit(...), else *err* is 0.

$\mathcal{F}$ `double SSE() const`
Use: *sse* = LR.SSE()
Pre: LR.Fit(...) has been called and LR.Error() returns 0.
Post: *sse* keeps the sum of the squares of errors in the last fit.

$\mathcal{F}$ `double intersection() const`
Use: a = LR.intersection()
Pre: LR.Fit(...) has been called and LR.Error() returns 0.
Post: a is the intersection of the straight line with the y-axis, i.e. $a ==$ LR.Funcval(0).

$\mathcal{F}$ `double slope() const`
Use: b = LR.slope()
Pre: LR.Fit(...) has been called and LR.Error() returns 0.
Post: b is the slope of the fitted line.

Protected Characteristics

`int`	`error`	// Error status.

`double`	`sse`	// Sum of squares of erros in last fit.
`double`	`a`	// Coefficients for the line y = a + bx.
`double`	`b`	// Coefficients for the line y = a + bx.

14.2 LogLinearRegression

The class LogLinearRegression calculates a log-log fit through given set of datapoints (x, y). I.e. the model

$$y = ax^b.$$

The model fitted is

$$\log(y) = \log(a) + b\log(x).$$

That is why we often mention the 'fitted line' in the class descriptions.

Inheritance

class LogLinearRegression

Public messages

`LogLinearRegression()`

Use: LogLinearRegression LLR
Pre: None.
Post: LLR is of type LogLinearRegression.

$\mathcal{S}$ `void Fit(const doublevector&, const doublevector&)`

Use: LLR.Fit(x, y)
Pre: Omitting the point (0,0): x.Size() $=$ y.Size() $>$ 0 and no element of x or y is ≤ 0. The point (0,0) may be in $(x[i], y[i])$ for some i's but they are ignored.
Post: LLR has fitted a line through the points $(\log(x), \log(y))$, where log is base e (natural logarithm). If the preconditions are met, LLR.Error() returns 0, else 1.

$\mathcal{S}$ `virtual void Fit(const doublevector&, const doublevector&, double)`

Use: LLR.Fit(x, y, b)
Pre: See above.
Post: LLR has fitted a line whose slope is b through the points $(\log(x[i]), \log(y[i]))$. See the discussion above. If the preconditions are not met, LLR.Error() returns 1, else 0.

$\mathcal{S}$ `virtual void Fit(const doublevector&, const doublevector&, double, double)`

Use: LLR.Fit(x, y, b, a)
Pre: See above.
Post: LLR has set its line to have the slope b and the intercept a. If the preconditions are not met, LLR.Error() returns 1, else 0.

$\mathcal{F}$ `int Error()`

Use: *err* = LLR.Error()

Pre: None.

Post: *err* is 1 if an error has occured, else *err* is 0. See the other functions to find out when an error occurs.

$\mathcal{F}$ `double SSE() const`

Use: *sse* = LLR.SSE()

Pre: LLR.Fit(…) has been called and LLR.Error() returns 0.

Post: *sse* is the sum of squares of error in the fit of the line in the last call to LLR.Fit(…). If the preconditions are met, LLR.Error() returns 0, else 1.

$\mathcal{F}$ `double SSE(const doublevector&, const doublevector&)`

Use: *sse* = LLR.SSE(x, y)

Pre: LLR.Fit(…) has been called and LLR.Error() returns 0. x.Size() == y.Size() and all elements of x, y are ≥ 0.

Post: *sse* is the sum of the squares of (LLR.LogFuncval($x[i]$) - $y[i]$). If the preconditions are met, LLR.Error() returns 0, else 1.

$\mathcal{F}$ `double WeightedSSE(const doublevector&,const doublevector&, const doublevector&)`

Use: *sse* = LLR.WeightedSSE(x, y, *weights*)

Pre: LLR.Fit(…) has been called and LLR.Error() returns 0. x.Size() == y.Size() and all elements of x, y are ≥ 0.

Post: *sse* is the sum of $weights[i](LLR.LogFuncval(x[i]) - y[i])^2$. If the preconditions are met, LLR.Error() returns 0, else 1.

$\mathcal{F}$ `double Funcval(double)`

Use: y = LLR.Funcval(x)

Pre: $x > 0$, LLR.Fit(…) has been called and LLR.Error() returns 0.

Post: y has the value of the fitted model at the point x; $y = ax^b$. If the preconditions are met, LLR.Error() returns 0, else 1.

$\mathcal{F}$ `double LogFuncval(double)`

Use: y = LLR.LogFuncval(x)

Pre: $x > 0$, LLR.Fit(…) has been called and LLR.Error() returns 0.

Post: y has the value of the fitted line at the point x; $y = \log(a) + b\log(x)$, compare with the equations above.
If the preconditions are met, LLR.Error() returns 0, else 1.

$\mathcal{F}$ `double intersection() const`

Use: a = LLR.intersection()

Pre: LLR.Fit(…) has been called and LLR.Error() returns 0.

Post: a is the intersection of the fitted line with the y-axis. Compare with the equations above (i.e. a equals what is denoted with $\log(a)$ there.

$\mathcal{F}$ `double slope() const`

Use: b = LLR.slope()
Pre: LLR.Fit has been called and LLR.Error() returns 0.
Post: b is the slope of the fitted line. Compare the equations above.

Protected Characteristics

```
LinearRegression    LR                          // Calculates the fits.
int                 error                       // Error status.
```

14.3 PopStatistics

The class PopStatistics handles some of the statistics one might want to compute, given abundance numbers and mean weights by length.
Warning messages: If the class receives a popinfo with mean weight equal to 0 but nonzero abundance numbers, a warning message is printed.

Inheritance

class PopStatistics

Public messages

`PopStatistics(const popinfoindexvector&, const LengthGroupDivision* const)`

Use: PopStatistics P(*pop*, *LgrpDiv*)

`PopStatistics(const popinfovector&, const LengthGroupDivision* const)`

Use: PopStatistics P(*pop*, *LgrpDiv*)
Pre: *pop* is a nonempty vector, *LgrpDiv* is $\neq 0$ and describes the length group division of *pop*.
Post: P is of type PopStatistics, and is ready to give statistics about the population received in *pop*.

`~PopStatistics()`

Use: ~P

Pre: None.
Post: All memory belonging to P has been freed.

$\mathcal{F}$ `double MeanLength() const`
Use: m = P.MeanLength()
Pre: None.
Post: m holds the mean length of the population given to P at the time of creation.

$\mathcal{F}$ `double MeanWeight() const`
Use: m = P.MeanWeight()
Pre: None.
Post: m is the mean weight of the population of P.

$\mathcal{F}$ `double TotalNumber() const`
Use: t = P.TotalNumber()
Pre: None.
Post: t is the total abundance number of the population of P.

$\mathcal{F}$ `double StdDevOfLength() const`
Use: s = P.StdDevOfLength()
Pre: none.
Post: s is the standard deviation of the length distribution of P.

Protected messages

$\mathcal{S}$ `void CalcStatistics(const popinfovector&, const LengthGroupDivision* const)`
Use: P.CalcStatistics(*pop*, *LgrpDiv*)
Pre: *pop*.Size() = *LgrpDiv*→NoLengthGroups()
Post: P has calculated the statistics for the vector *pop*, assuming *LgrpDiv* is its length group division, and keeps the results so they can be accessed through the member functions.

Protected Characteristics

```
double          meanlength          //
double          meanweight          //
double          totalnumber         //
double          stddevOflength      //
```

Chapter 15

Likelihood.

15.1 Overview

The classes introduced in this section are related as follows:

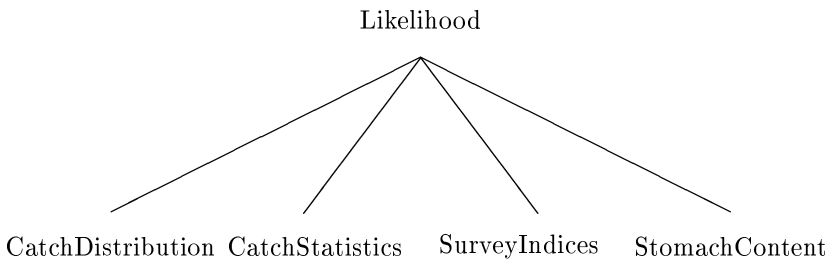

Figure 15.1: Descendants of Likelihood.

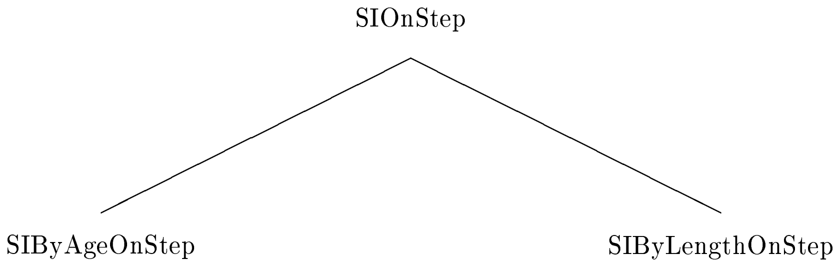

Figure 15.2: Descendants of SIOnStep.

15.2 Likelihood

The class Likelihood is an abstract base class for all those classes that compute likelihood.

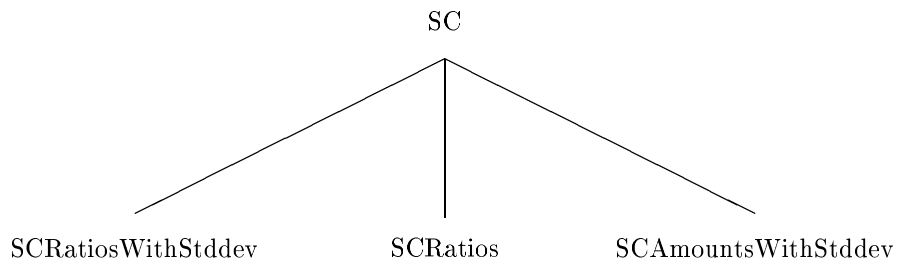

Figure 15.3: Descendants of SC.

As with all abstract base classes, Likelihood will be described as if it could be instantiated to show the recommended meaning of the member functions.
When using this class in a simulation, the member function AddToLikelihood should be called on every timestep.

Inheritance

class Likelihood

Public messages

`Likelihood(LikelihoodType)`
Use: Likelihood L(t)
Pre: None.
Post: L is of type Likelihood and L.Type() will return t. See below.

`virtual  Likelihood() = 0`
Use: ˜L
Pre: None.
Post: All memory belonging to L has been freed.

`virtual void AddToLikelihood(const TimeClass* const) = 0`
Use: L.AddToLikelihood(*TimeInfo*)
Pre: *TimeInfo* $\neq 0$.
Post: L has added to its likelihood, based on the current time of *TimeInfo*.

$\mathcal{S}$ `virtual void Reset()`
Use: L.Reset()
Pre: None.
Post: L has reset itself, i.e. deleted from itself the information it had kept in the call to AddToLikelihood, so that it can now collect likelihood information again.

`virtual void Print(ofstream&) const = 0`
Use: L.Print(*outfile*)
Pre: *outfile* has no bad bits set.
Post: L has printed its internal information to outfile.

$\mathcal{F}$ `double ReturnLikelihood() const`
Use: ℓ = L.ReturnLikelihood()
Pre: None.
Post: ℓ is the current value of L's accumulated **weighted** likelihood.

$\mathcal{F}$ `double UnweightedLikelihood() const`
Use: ℓ = L.UnweightedLikelihood()
Pre: None.
Post: ℓ is the current value of L's accumulated **unweighted** likelihood.

$\mathcal{F}$ `double Weight() const`
Use: w = L.ReturnLikelihood()
Pre: None.
Post: w is equal to L's weight, i.e. L.ReturnLikelihood() == L.Weight() $*$ L.UnweightedLikelihood().

$\mathcal{F}$ `LikelihoodType Type() const`
Use: t = L.Type()
Pre: None.
Post: t has the type of L. This member function is to be used for run-time type identification of derived classes.

Protected Characteristics

`double`	`likelihood`	// Accumulated likelihood.
`double`	`weight`	//

Private Characteristics

`Likelihoodtype`	`type`	//

15.3 SurveyIndices

The class SurveyIndices calculates likelihood based on using survey indices as time series. It only works with a single set of areas that are joined together in all calculations – better read the description of the file format.

Inheritance

class SurveyIndices : `public` Likelihood

Public messages

`SurveyIndices(CommentStream&, const AreaClass* const,`
`const TimeClass* const )`

Use: SurveyIndices SI(*infile*, *Area*, *TimeInfo*)
Pre: *infile* has no badbits set and the pointers are not null. *infile*'s format is correct with respect to the information in *Area* and *TimeInfo*.
Post:

`virtual ~SurveyIndices()`

Use: ~SI
Pre: None.
Post: All memory belonging to SI has been freed.

$\mathcal{C}$ `virtual void AddToLikelihood(const TimeClass* const)`

Use: SI.AddToLikelihood(*TimeInfo*)
Pre: *TimeInfo* $\neq 0$.
Post: SI has added to its likelihood function value.

$\mathcal{S}$ `void SetStocks(Stockptrvector&)`

Use: SI.SetStocks(*Stocks*)
Pre: *Stocks*.Size() > 0 and no pointer in *Stocks* is null. In *Stocks* all the names of the stocks read from *infile* are found.
Post: SI keeps pointers to its stocks.
NB: The lifetime of the objects pointed to in *Stocks* has to be at least equal to that of SI.

$\mathcal{S}$ `virtual void Reset()`

Use: SI.Reset()
Pre: None.
Post: SI has reset itself, i.e. deleted from itself the information it had kept in the call to AddToLikelihood(...), so that it can now collect likelihood information again.

`virtual void Print(ofstream&) const`

Use: SI.Print(*outfile*)
Pre: *outfile* has no bad bits set.
Post: SI has printed its internal information to *outfile*.

Protected messages

$\mathcal{S}$ `void ReadLengths(CommentStream&, const AreaClass* const, const TimeClass* const )`

Use: SI.ReadLengths(*infile*, *Area*, *TimeInfo*)
Pre: The usual stuff about error bits in *infile* and nonzero pointers. And *infile* is positioned at the beginning of the file format for survey indices by length.
Post: SI has created a new SIOnStepByLength and keeps a pointer to the class there.

$\mathcal{S}$ `void ReadAges(CommentStream&, const AreaClass* const, const TimeClass* const )`

Use: SI.ReadAges(*infile*, *Area*, *TimeInfo*)
NB: This is similar to ReadLengths(. . .); this function works with survey indices by age instead of by length. Therefore, `SI` keeps a pointer to SIByAgeOnStep. [Note: here `SI` is the protected variable of type SIOnStep∗.]

Protected Characteristics

`SIOnStep*`	`SI`	// Collects the info.
`intvector`	`areas`	// Our areas.
`charptrvector`	`stocknames`	// Names of Stock.

The class lets objects of type SIOnStep do really most of the work.

15.4 SIOnStep

The class SIOnStep is an abstract base class, a guideline as to how to write likelihood classes that compare some data obtained from the stocks to a matrix of values.
The class SIOnStep uses survey indices on a fixed step as time series to use in a regression. The sum of squares in that regression can be obtained.
The fit of survey indices, (I_i), v.s. the observed variables, (N_i), is $\log(N_i) \sim \alpha + \beta \log(I_i)$, where one of the following applies:

- Both α and β are free.
- β is fixed and α is free.
- Both α and β are fixed.

Inheritance

class SIOnStep

Public messages

`SIOnStep(CommentStream&, int, const TimeClass* const)`

Use: SIOnStep S(*infile*, *ncol*, *TimeInfo*)
Pre: *infile* has no badbits set and its file format is correct. *TimeInfo* $\neq 0$. *ncol* is the number of columns in the input file (refer to the documentation of the file format in the Users Manual).
Post: S is of type SIOnStep.
NB: The initialization of S is not complete until S.SetStocks(...) has been called.

`virtual ~SIOnStep()`

Use: ~S
Pre: None.
Post: All memory belonging to S has been freed.

`virtual void Sum(const TimeClass* const) = 0`

Use: S.Sum(*TimeInfo*)
Pre: *TimeInfo* $\neq 0$, S.SetStocks(...) has been called.
Post: S has summed up information from its stocks.

`virtual void SetStocks(const Stockptrvector&) = 0`

Use: S.SetStocks(*Stocks*)
Pre: *Stocks*.Size() > 0, no pointer in *Stocks* is null.
Post: S is ready to obtain data from the objects pointed to in *Stocks*.
NB: S will use **all** the objects pointed to in *Stocks*.

$\mathcal{F}$ `int Error() const`

Use: *err* = S.Error()
Pre: None.
Post: *err* equals 1 if an error has occurred in last call to Regression(), else 0.

$\mathcal{S}$ `void Clear()`

Use: S.Clear()
Pre: None.
Post: Internal error status is set to 0.

$\mathcal{F}$ `virtual double Regression()`

Use: *SSE* = S.Regression()
Pre: S.SetStocks(. . .) has been called.
Post: *SSE* is the sum of squares of error in the regression S made, if S had obtained enough data to make the fit. If S did not have enough data to do the fit, *SSE* is 0 and the error status in S is set to 1.

$\mathcal{S}$ `virtual void Reset()`

Use: S.Reset()
Pre: None.
Post: S has reset itself, i.e. deleted from itself the information it had kept from the calls to S.Sum(. . .), so that it can now collect abundance information again.

`virtual void Print(ofstream &) const`

Use: S.Print(*outfile*)
Pre: S.SetStocks(. . .) has been called.
Post: S has written internal information to *outfile*.

Protected messages

$\mathcal{S}$ `void SetError()`

Use: S.SetError()
Pre: None.
Post: S has set its error bit.

$\mathcal{F}$ `int IsToSum(const TimeClass* const) const`

Use: t = S.IsToSum(*TimeInfo*)
Pre: *TimeInfo* $\neq 0$.
Post: If S is to sum on the current timestep, t equals 1, else 0.

$\mathcal{S}$ `void KeepNumbers(const doublevector&)`

Use:	S.KeepNumbers(*numbers*)
Pre:	S.SetStocks(...) has been called, *numbers*.Size() equals *ncol* (the argument in the constructor) and S.IsToSum(...) is true on the current timestep.
Post:	S has added the contents of numbers to its private variable `abundance`.
NB:	This function is intended for derived classes to use from within the virtual member function Sum(...).

Private enum

FitType	LogLinearFit=0
	FixedSlopeLogLinearFit
	FixedLogLinearFit

Private messages

$\mathcal{F}$ `double Fit(const doublevector&, const doublevector&)`

Use:	ℓ = S.Fit(*stocksize*, *indices*)
Pre:	*stocksize*.Size() = *indices*.Size()
Post:	ℓ contains the sum of squares of errors in the fit of *stocksize* to *indices*.

Private Characteristics

`doublematrix`	`Indices`	// [year][???] – survey indices.
`doublematrix`	`abundance`	// [year][???] – abundance numbers.
`intvector`	`Years`	// Vector of years.
`intvector`	`Steps`	// Vector of steps.
`ActionAtTimes`	`AAT`	// Should we sum?
`int`	`error`	// Error status.
`int`	`NumberOfSums`	//
`FitType`	`fittype`	// The type of fit.
`double`	`slope`	// The slope in the fit.
`double`	`intercept`	// The intercept in the fit.

Data invariant

- The matrices `Indices` and `abundance` are of the same size and their number of rows equals the length of `Years` and `Steps`.

- Each line `Indices`[i] and `abundance`[i] corresponds to the elements `Years`[i] and `Steps`[i].
- The list obtained from (`Years`, `Steps`) is in chronological order.

Details

The private variable `fittype` is of the type SIOnStep::FitType and describes what kind of fit to make in SIOnStep::Fit(...). Depending on the value of `fittype`, the private variables `slope` and `intercept` may be used.

15.5 SIByLengthOnStep

The class SIByLengthOnStep collects abundance numbers from Stock and compares them with abundance indices. It works by joining together a set of areas into one by summing over them the up abundance numbers of stocks.
The regression SIByLengthOnStep calculates is described below.

Inheritance

class SIByLengthOnStep : `public` SIOByOnStep

Public messages

`SIByLengthOnStep(CommentStream&, const intvector&, const doublevector&, const TimeClass* const)`

Use: SIByLengthOnStep S(*infile*, *areas*, *lengths*, *TimeInfo*)
Pre: The format of *infile* is correct, *areas* is a nonempty vector of nonnegative, unique elements, *lengths* is a nonempty vector whose elements are distinct and in ascending order, *TimeInfo* $\neq 0$.
Post: S is of type SIByLengthOnStep. It has read its information from *infile* and kept the information for the period *TimeInfo* marks. *lengths* is understood to be a length group division for the information read from *infile*.
NB: S is not fully initialized until SetStocks(...) has been called.

`virtual ~SIByLengthOnStep()`

Use: ~S
Pre: None.
Post: All memory belonging to S has been freed.

$\mathcal{C}$ `virtual void Sum(const TimeClass* const)`

Use: S.Sum(*TimeInfo*)
Pre: *TimeInfo* $\neq 0$.
Post: If the current time *TimeInfo* shows is in the list of times S keeps, it sums information from its Stocks, else nothing is done.

$\mathcal{S}$ `virtual void SetStocks(const Stockptrvector&)`

Use: S.SetStocks(*Stocks*)
Pre: See SIOnStep and: the length group division of S (argument in the constructor) has to be coarser than or equal to that of all the Stocks pointed to in *Stocks*.
Post: See SIOnStep.

Details

The regression that SIByLengthOnStep calculates is as follows:
Let N_y be stock number on a year y and I_y a measured abundance index for the same year. Then we calculate the regression of $\log(I)$ as a linear function of $\log(N)$ to get α and β, i.e. the relationship $\log(I) = \alpha + \beta \log(N)$.
Then we let SSE be the sum of squares of errors from the fit, i.e.

$$SSE = \sum_y [\log(I_y) - (\alpha + \beta \log(N_y))]^2.$$

SIByLengthOnStep sums up abundance numbers from several stocks (the argument in SetStocks(...)) on several areas (the argument in the constructor). It divides the abundance numbers into length groups according to the division given in the constructor and calculates this sum of squares of errors for every length group.
The value returned from S.Regression() is the sum of SSE over all the length groups in S.

Protected Characteristics

```
StockAggregator*    aggregator             // This one aggregates.
LengthGroupDivision*LgrpDiv                // Describes lengths.
intvector           Areas                  // Over which to sum.
```

Details

The abundance matrix is indexed with year – just like SIOnStep::`Indices`. These two matrices will be compared when calculating the regression.

15.6 SIByAgeOnStep

The class SIByAgeOnStep is quite similar to SIByLengthOnStep, except that the survey indices it uses are to be for age groups, not for length groups.

Inheritance

class SIByAgeOnStep : `public` SIOnStep

Public messages

`SIByAgeOnStep(CommentStream&, const intvector&, const doublevector&, const TimeClass* const)`

Use: SIByAgeOnStep S(*infile*, *areas*, *ages*, *TimeInfo*)
Pre: The format of *infile* is correct, *areas* is a nonempty vector of nonnegative, unique elements, *ages* is a nonempty matrix with nonempty lines, $TimeInfo \neq 0$.
Post: S is of type SIByAgeOnStep. It has read its information from *infile* and kept the information for the period *TimeInfo* marks. *ages* is describe the information read from *infile*, i.e. each line in *ages* contains the agegroups aggregated to obtain the survey indices.
NB: S is not fully initialized until SetStocks has been called.

`~SIByAgeOnStep()`

Use: ~S
Pre: None.
Post: All memory belonging to S has been freed.

$\mathcal{C}$ `virtual void Sum(const TimeClass* const)`

Use: S.Sum(*TimeInfo*)
Pre: $TimeInfo \neq 0$.
Post: If the current time *TimeInfo* shows is in the list of times S keeps, it sums information from its Stocks, else nothing is done.

$\mathcal{S}$ `virtual void SetStocks(const Stockptrvector&)`

Use: S.SetStocks(*Stocks*)
Pre: See SIOnStep.
Post: See SIOnStep.

$\mathcal{S}$ `virtual void Reset()`

Use: S.Reset()
Pre: None.
Post: S has reset itself, i.e. deleted from itself the information it had kept from the calls to Sum(...), so that it can now collect abundance information again.

`virtual void Print(ofstream &) const`

Use: S.Print(*outfile*)
Pre: S.SetStocks(. . .) has been called.
Post: S has written internal information to *outfile*.

Protected Characteristics

`StockAggregator*`	`aggregator`	// It aggregates.
`intmatrix`	`Ages`	// How to aggregate agegroups
`doublematrix`	`abundance`	// -[year][age]
`intvector`	`Areas`	// Areas to aggregate.

Details

The line `Ages`[i], $0 \leq i <$ `Ages`.Nrow(), is a collection of age groups that are to be merged into one. See SIByLengthOnStep for further details on, e.g. how the regression is done.