Contents

14

Manual Pages and Documentation

This chapter is authored jointly with Evelyn Haak, Michael Seel, and Christian Uhrig.

Software requires documentation. In this chapter we explain:

- how to make LEDA-style manual pages,
- how to make a LEDA-style manual,
- and how to write documentations in the style of this book.

14.1 Lman and Fman

Lman and Fman are the LEDA tools for manual production and quick reference to manual pages. We will discuss Fman at the end of the section. The command

```
Lman T[.lw|.nw|.h] options
```

searches for a file with name T.lw, T.nw, T.h, or T (in this order) first in the current directory and then in the directory LEDAROOT/incl/LEDA and produces a LEDA-style manual page from it. Thus

```
Lman sortseq
Lman myproject.lw
```

produce the manual page of sorted sequences and of myproject, respectively.

The extraction of the manual page is guided by the so-called manual comments contained in the file-argument of Lman. A manual comment is any comment of the form

```
/*{\Mcommand ... arbitrary text ... }*/
```

```
/*{\Manpage {stack} {E} {Stacks} {S}}*/

template<class E> class _CLASSTYPE stack : private SLIST
{
/*{\Mdefinition
An instance |S| of the parameterized data type |\Mname| is a sequence of
elements of data type |E|, called the element type of |S|. Insertions or
deletions of elements take place only at one end of the sequence, called
the top of |S|. The size of |S| is the length of the sequence, a stack
of size zero is called the empty stack.}*/

  void copy_el(GenPtr& x)  const { x=Copy(ACCESS(E,x)); }
  void clear_el(GenPtr& x) const { Clear(ACCESS(E,x)); }
public:

/*{\Mcreation}*/

  stack() {}
/*{\Mcreate creates an instance |\Mvar| of type |\Mname| and initializes
it to the empty stack.}*/

  stack(const stack<E>& S) : SLIST(S) {}
 ~stack() { clear(); }
  stack<E>& operator=(const stack<E>& S)
    { return (stack<E>&)SLIST::operator=(S); }

/*{\Moperations 2.5 4}*/

E top()   const { return ACCESS(E,SLIST::head());}
/*{\Mop      returns the top element of |\Mvar|.\\
             \precond $S$ is not empty.}*/

void push(E x)  { SLIST::push(Copy(x)); }
/*{\Mop      adds $x$ as new top element to |\Mvar|.}*/

E pop()         { E x=top(); SLIST::pop(); return x; }
/*{\Mop      deletes and returns the top element of |\Mvar|.\\
             \precond $S$ is not empty.}*/

int  empty() { return SLIST::empty(); }
/*{\Mop      returns true if |\Mvar| is empty, false otherwise.}*/

}

\*{\Mimplementation
Stacks are implemented by singly linked linear lists.
All operations take time $O(1)$. }*/
```

Figure 14.1 A file decorated by manual comments. The file is part of the header file of the data type stack. Figure 14.2 shows the manual page produced by Lman.

where Mcommand is one of so-called manual commands. We discuss manual commands in Section 14.2.2. Every manual comment causes Lman to extract part of the manual. Figures 14.1 and 14.2 show a file augmented by manual comments and the manual page produced from it.

The layout of the manual page is fine-tuned by the options-argument of Lman. We will discuss the available options in Section 14.2.8. Options may also be put in a configuration file *Lman.cfg* in either the home directory or the working directory. Command line options

Stacks (stack)

1. Definition

An instance S of the parameterized data type *stack*<*E*> is a sequence of elements of data type E, called the element type of S. Insertions or deletions of elements take place only at one end of the sequence, called the top of S. The size of S is the length of the sequence, a stack of size zero is called the empty stack.

2. Creation

stack<*E*> *S*;	creates an instance *S* of type *stack*<*E*> and initializes it to the empty stack.

3. Operations

E	*S*.top()	returns the top element of *S*. *Precondition*: *S* is not empty.
void	*S*.push(*E x*)	adds *x* as new top element to *S*.
E	*S*.pop()	deletes and returns the top element of *S*. *Precondition*: *S* is not empty.
int	*S*.empty()	returns true if *S* is empty, false otherwise.

Figure 14.2 The manual page produced from the file in Figure 14.1.

take precedence over options in the working directory which in turn take precedence over options in the home directory.

Fman is our tool for quick reference to manual pages. The command

```
Fman T[.lw|.nw|.h] filter
```

searches for a file with name T.lw, T.nw, T.h, or T (in this order) first in the current directory and then in the directory LEDAROOT/incl/LEDA and extracts manual information from it. The information is displayed in ASCII-format. For example,

```
Fman sortseq insert
Fman sortseq creation
```

give information about operation insert of type sortseq and about the different ways of creating a sorted sequence, respectively.

```
Fman
```

gives information about Fman and the available filters.

Fman uses Perl [WS90] and Lman uses Perl, LaTeX [Lam86], and xdvi.

Please try out Lman and Fman before proceeding. If they do not work, the error is very likely to be one of the following (if not, you should refer to the LEDA installation guide):

- One of the required systems Perl, LaTeX, and xdvi is not installed.
- The environment variable LEDAROOT is not set to the root directory of the LEDA system.
- LEDAROOT/Manual/cmd is not part of your PATH.
- LEDAROOT/Manual/tex is not part of your TEXINPUTS.

14.2 Manual Pages

Figure 14.2 shows a typical LEDA manual page. It is produced from the file in Figure 14.1 by a call of the Lman utility. Observe that the file contains comments starting with `/*{\M...` and ending with `}*/`. They are called *manual comments*. They start with a so-called manual command, e.g., `Mdefinition` or `Mop` and control the extraction of the manual page from the header file. There are about twenty different manual commands. We will discuss them in turn in this section. Before doing so, we justify our decision to incorporate all manual information into the header files of the LEDA system.

In the early years of the LEDA project we kept the manual page of a data type separate from its implementation. The manual was contained in a tex-file and the implementation was contained in an h-file and a c-file. Updates of a data type usually required changes to all three files and this led to a consistency problem between the three files. The consistency between h-file and c-file is a minor issue since every compiler run checks it. However, we found it almost impossible to keep the manual pages consistent with the implementation. The inconsistencies between manual and implementation had two causes:

- Clerical errors: Frequently, things that were supposed to be identical were different, e.g., a type was spelled `sort_seq` in the manual and `sortseq` in the implementation, or the parameters of a function were permuted.
- Lack of discipline: We frequently forgot to make changes due to lack of time or other reasons. We were quite creative in this respect.

In 1994 we decided to end the separation between implementation and manual. We incorporated the manual into the h-files in the form of so-called manual comments and wrote a tool called *Lman* that extracts the tex-file for the manual page automatically from the h-file. Every manual comment produces part of the manual page, e.g., the manual comment starting with `\Mdefinition` produces the definition section of the manual page, and a comment starting with `\Mop` produces an entry for an operation of the data type. Such an entry consists of the return type, an invocation of the operation, and a definition of the semantics in the form of a text. Only the latter piece of information is explicitly contained in the Mop-comment, the other two pieces are generated automatically from the C++-text in the header file. Experience shows that our decision to incorporate manual pages into header files greatly alleviates the consistency problem:

- Clerical errors are reduced because things that should be identical are usually only typed once. For example, the fact that the C++-text in the manual is automatically generated from the C++-text in the header file guarantees the consistency between the two.
- Lack of discipline became a lesser issue since the fact that the header file of the implementation and the tex-file for the manual page are indeed the same file makes it a lot easier to be disciplined.

Lman produces manual pages in a two-step process. It first extracts a TEX-file from the header file and then applies LATEX. The first step is directed by the manual commands in the header file and the second step uses a specially developed set of TEX macros. We discuss the manual commands in Section 14.2.2 and the TEX macros in Section 14.2.5.

The first phase is realized by a Perl-program *lextract* that reads the file-argument and the options and produces a (temporary) TEX-file of the form:

```
\documentclass[a4paper,size pt]{article}
\usepackage{Lweb}
\begin{document}
  output of lextract
\end{document}
```

The program lextract is defined in the file *ext.nw* in LEDAROOT/Manual/noweb.

14.2.1 *The Structure of Manual Pages*

All manual pages of the LEDA system are organized in one of two ways depending whether the page defines a data type or a collection of functions. Since manual pages are extracted from header files, the corresponding header files are organized accordingly. Examples of header files for data types are stack.h, sortseq.h, and list.h, and examples of header files for collections of functions are plane_alg.h, plane_graph_alg.h, and mc_matching.h.

All *header files for classes* follow the format shown in Figure 14.3. The *header files for collections of functions* have no particular structure.

14.2.2 *The Manual Commands*

We discuss the manual commands in the order in which they are typically used in the header file of a class.

The Manpage Command: A manual comment of the form

```
/*{\Manpage {type} {parlist} {title} {varname}}*/
```

produces the header line of the manual page for `type`. The argument `parlist` is the list of

```
  /*{\Manpage Comment }*/
class DT {
  /*{\Mdefinition comment }*/
  /*{\Mtypes comment }*/
  // type definitions
private:
  // private data and functions
public:
  /*{\Mcreation comment }*/
  // constructors and destructors and their manual entries
  /*{\Moperations comment }*/
  // operations and their manual entries
};
  // friends and their manual entries
  /*{\Mimplementation comment }*/
  /*{\Mexample comment }*/
```

Figure 14.3 The generic structure of a header file for a class. Any of the parts may be omitted.

type parameters of the type, `title` is the title of the manual page, and the optional argument `varname` is used in the manual page as the name of a canonical object of the type. The argument `parlist` is empty if the type has no type parameters. The following comments produce the header lines for character strings, linear lists, and sorted sequences, respectively.

```
/*{\Manpage {string}  {}    {Character Strings} {s}}*/
/*{\Manpage {list}    {E}   {Linear Lists}  {L}    }*/
/*{\Manpage {sortseq} {K,I} {Sorted Sequences} {S} }*/
```

The Manpage command produces the header line for the manual page and defines placeholders `\Mtype`, `\Mname`, and `\Mvar`. The first placeholder stands for `type`, the second placeholder stands for either `type` or `type<parlist>` depending on whether `parlist` is empty or not, and the third placeholder stands for `varname`. In the last example the placeholders `\Mtype`, `\Mname`, and `\Mvar` have values `sortseq`, `sortseq<K,I>`, and `S`, respectively.

The placeholders can be used instead of their values in later manual comments. This helps to maintain consistency. The placeholders are also used in the generation of the manual entries for the constructors and member functions, e.g., in Figure 14.2 all operations are applied to the canonical stack variable S.

What does lextract do when it encounters a Manpage-command? It records the values of all placeholders and outputs

```
\section*{title (type')}
```

where `type'` is obtained from `type` by quoting all occurrences of the underscore character (i.e., replacing `_` by `\_`). When LaTeX executes this line it will produce the header line of the manual page. If a manual page is to be included into a larger document, it is convenient to number the manual pages. The option `numbered=yes` causes the preprocessor to output

```
\section{title (type')} \label{title}\label{type}
```

The labels can be used to refer to the data type in other parts of an enclosing document.

The manual page of a class consists of sections *Definition*, *Types Creation*, *Operations*, *Implementation*, and *Example*; any of the sections may be omitted. Accordingly, we have the manual commands `\Mdefinition`, `\Mypes`, `\Mcreation`, `\Moperations`, `\Mimplementation`, and `\Mexample`.

The Mdefinition Command: A manual command of the form

```
/*{\Mdefinition  body }*/
```

produces the definition part of a manual page. For example,

```
template <class E>
class list {
/*{\Mdefinition
An instance [[\Mvar]] of class |\Mname| is a ...
}*/
```

produces

---

1. Definition

An instance `L` of class *list*<*E*> is a ...

The body of a definition comment (and of any of the other comments to come) is an arbitrary LaTeX text. As suggested by the literate programming tools CWEB [KL93] and noweb [Ram94] we added the possibility of quoting code. *Quoted code* is given special typographic treatment. There are two ways of quoting code:

- By enclosing it between verticals bars (| . . . |), or
- By enclosing it between double square brackets (`[[...]]`).

Quoted code is typeset according to the following rules: first all occurrences of the placeholders `\Mtype`, `\Mname`, and `\Mvar` are replaced by their values. We call this step *placeholder substitution*. In the example above this step yields[1]:

```
template <class E>
class list {
/*{\Mdefinition
An instance [[L]] of class |list<E>| is a ...
}*/
```

In a second step we apply what we call *C++ to LaTeX conversion* to quoted code. For code quoted by double square brackets this means using typewriter font for the quoted code and for code quoted by vertical bars this produces a math-like appearance, e.g., all identifiers

[1] We assume that the Mdefinition command is executed in the context of the Manpage comment for lists given above, i.e., L is the name of the canonical list and $list\langle E\rangle$ is the type of the list. We make the analogous assumption for all examples to follow.

are put into math-italics and `<=` is typeset as $\leq$. All code in this book is typeset using one of the two quoting mechanisms.

We give some examples of the quoting mechanisms. Be aware that putting an identifier between vertical bars is different from putting it between dollar signs except for identifiers consisting of a single character.

<table>
<tr><td>|diff|</td><td>produces</td><td>diff</td></tr>
<tr><td>$diff$</td><td>produces</td><td>$diff$</td></tr>
<tr><td>|x1|</td><td>produces</td><td>x1</td></tr>
<tr><td>$x1$</td><td>produces</td><td>$x1$</td></tr>
<tr><td>x</td><td>produces</td><td>x</td></tr>
<tr><td>|x|</td><td>produces</td><td>x</td></tr>
<tr><td>[[diff]]</td><td>produces</td><td>diff</td></tr>
</table>

Sometimes, one wants to produce vertical bars and/or double square brackets in the output. We provide TeX-macros to this effect. The macros `\Lvert`, `\DLK` and `\DRK` expand to |, `[[`, and `]]`, respectively. The TeX-macro `\Labs{...}` puts its argument between vertical bars, Lvert and Labs can only be used in math-mode.

We close this paragraph with a *warning*. The quoting mechanism by vertical bars is not perfect. In principle one can put any piece of text between vertical bars. The preprocessor attempts to understand the C++ structure of the text and generates output accordingly. Since the preprocessor has only limited knowledge of the syntax of C++, it succeeds only in simple cases:

<table>
<tr><td>|diff|</td><td>produces</td><td>diff</td></tr>
<tr><td>|diff + x1|</td><td>produces</td><td>$\mathit{diff} + \mathit{x1}$</td></tr>
<tr><td>|diff+x1|</td><td>produces</td><td>$\mathit{diff} + \mathit{x1}$</td></tr>
<tr><td>|list_item|</td><td>produces</td><td>list_item</td></tr>
<tr><td>|GRAPH<POINT,int>|</td><td>produces</td><td>$\mathit{GRAPH}<\mathit{POINT}, \mathit{int}>$</td></tr>
<tr><td>|mark[v] <= cur_mark|</td><td>produces</td><td>$\mathit{mark}[v] \leq \mathit{cur\_mark}$</td></tr>
<tr><td>$|source|(e_0)$</td><td>produces</td><td>$\mathit{source}(e_0)$</td></tr>
</table>

The Mtypes and Mtypemember Commands: A manual command of the form

```
/*{\Mtypes w}*/
```

produces the header line of the type part of the manual. The argument `w` is optional. The argument w governs the layout of the entries for the local types of the data type. We will discuss it below. The manual entries for the local types are produced by Mtypemember commands. We give an example which is taken from the header file for the LEDA extension package for higher-dimensional geometry.

```
/*{\Mtypes 4}*/
typedef ch_Simplex<CHTRAITS,POINT,PLANE>* ch_simplex;
/*{\Mtypemember the item type for simplices of the complex.}*/

typedef ch_Simplex<CHTRAITS,POINT,PLANE>* ch_facet;
/*{\Mtypemember the item type for facets of the complex.}*/
```

```
typedef rc_Vertex<CHTRAITS,POINT>*        ch_vertex;
/*{\Mtypemember the item type for vertices of the complex.}*/
```

produces

---

2. Types

ch_simplex	the item type for simplices of the complex.
ch_facet	the item type for facets of the complex.
ch_vertex	the item type for vertices of the complex.

Each Mtypemember command produces a manual entry for a local type. Each manual entry is typeset on a line of its own and a two-column layout is followed. There is a column of width w containing the name of the local type and a column containing the text explaining the local type. The name of the type is extracted automatically from the type definition preceding the manual comment.

The Mcreation and Mcreate Commands: A manual command of the form

```
/*{\Mcreation name w}*/
```

produces the header line of the creation part of the manual. The arguments `name` and `w` are optional. If name is present, it is used as the value of the placeholder `\Mvar`. We recommend that you define `\Mvar` already in the Manpage command and keep the possibility to define it in the Mcreation command for reasons of backward compatibility. The argument w governs the layout of the entries for the constructors of the data type. We will discuss it below. The manual entries for the constructors are produced by Mcreate commands. We give an example.

```
/*{\Mcreation}*/

vector();
/*{\Mcreate creates an instance |\Mvar| of type |\Mname|;
|\Mvar| is initialized to the zero-dimensional vector.}*/

vector(int d);
 /*{\Mcreate creates an instance |\Mvar| of type |\Mname|;
|\Mvar| is initialized to the zero vector of dimension $d$.}*/
```

produces (assuming that Mvar stands for v and Mname stands for vector)

---

3. Creation

vector v;	creates an instance *v* of type *vector*; *v* is initialized to the zero-dimensional vector.
vector v(*int d*);	creates an instance *v* of type *vector*; *v* is initialized to the zero vector of dimension d.

Each Mcreate command produces a manual entry for a constructor. The manual entries are typeset in the form of a variable declaration for a variable Mvar of type Mname, i.e., for the default constructor the entry has the form

```
Mname Mvar;
```

and for a constructor taking arguments the entry has the form

```
Mname  Mvar(parameter list);
```

In the second case the parameter list is extracted automatically from the code unit preceding the manual comment. What is a code unit?

A *code unit* is a maximal sequence of consecutive non-blank lines not containing a comment. In other words, the line preceding a code unit is either empty or the end of a comment, the line following a code unit is either empty or the beginning of a comment, and all lines in a code unit are non-empty and do not belong to a comment. A code unit from which the preprocessor is supposed to extract a function declaration should contain exactly one such declaration. The general form for generating an entry for a constructor is therefore:

```
<empty line or end of a comment>
<code unit>
<zero or more empty lines>
/*{\Mcreate   body  }*/
```

The body of the Mcreate command contains the text that explains the constructor. Placeholder substitution and C++ to LaTeX conversion are applied to it. We give some more examples.

```
vector(double d, double e)
{ ... inline implementation of constructor ...}
/*{\Mcreate This is okay.}*/

vector(double d, double e, double f)
/*{\Mcreate This is also okay.}*/
{ ... inline implementation of constructor ...}

vector();
vector(int d);
/*{\Mcreate illegal, since code unit contains more
than one constructor.}*/

vector(double d)

{ ... inline implementation of constructor ...}
/*{\Mcreate illegal, since code unit preceding
the manual comment contains no constructor.}*/

vector(long d); /*{\Mcreate illegal, since manual comment
must start on a new line}*/
```

We still need to discuss the role of the optional argument w. The layout for the manual entry of a constructor follows either the two-column format shown in Figure 14.4 or the

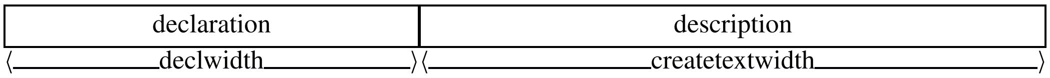

Figure 14.4 The two-column layout for constructors.

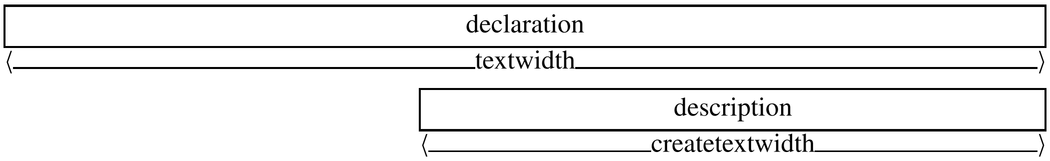

Figure 14.5 The two-row layout for constructors.

two-row format shown in Figure 14.5. The argument w defines the value of *declwidth*. The default value of declwidth is 40% of the textwidth. The value of *createtextwidth* is defined by

$$\text{createtextwidth} = \text{textwidth} - \text{declwidth}.$$

We use two-column layout if the declaration is short enough to fit into a box of width declwidth and use two-row layout otherwise. The argument w is either a pure number or a number followed by one of the TEX units of length (mm, cm, in, pt, or em). A missing unit is taken to be cm, i.e., 3.2 is equivalent to 3.2cm.

The Mdestruct Command: Mdestruct applies to the destructor of a class.

```
~vector();
/*{\Mdestruct The destructor ...}*/
```

produces

$\sim$*vector*()	The destructor ...

It is customary in LEDA to produce *no* manual entry for the assignment operator, the copy constructor, and the destructor of a class because the semantics of these operations is defined in a uniform way for all LEDA types (see Section 2.3) and hence there is no need to define them again for each data type. In fact, it would be confusing. Think twice before you break this rule.

We now come to the section for the operations of a data type. It is started by a Moperations comment.

The Moperations Command: A comment of the form

```
/*{\Moperations a b }*/
```

generates the header line of the operations part. The length arguments a and b are optional. An entry in the operations part is displayed in either a three-column layout as shown in Figure 14.6 or a two-row layout as shown in Figure 14.7. The values of *typewidth* and

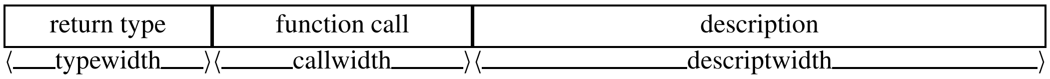

Figure 14.6 The three-column layout for the operations of a data type.

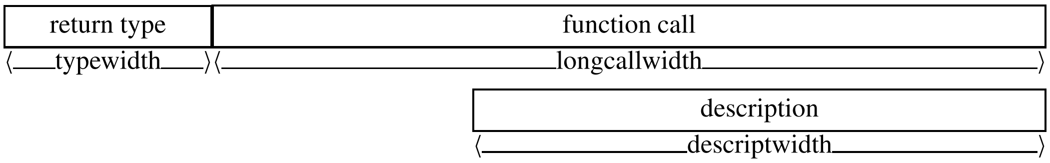

Figure 14.7 The two-row layout for the operations of a data type.

callwidth are set to a and b, respectively, and the value of *descriptwidth* is defined by the equation

$$\text{descriptwidth} = \text{textwidth} - \text{typewidth} - \text{callwidth}.$$

We choose the three-column layout if the function call fits into a box of width callwidth and the two-row layout otherwise[2]. If the return type does not fit into a box of width typewidth, we combine the return type and the function call into a single unit and attempt to put it into a box of width typewidth + callwidth. If the combined unit fits, we use a modified three-column layout, if it does not fit, we use a modified two-row layout.

An operation of a data type is either a member or a friend. In either case it can be a function or an operator. Operators may be binary or unary. We have a manual command for each case. The existence of distinct manual commands for the distinct cases is a historical relict. The current version of the extractor knows the syntax of C++ sufficiently well to be able to distinguish the cases without guidance by the manual command; this was not the case for an earlier version of the extractor. We find that the use of distinct manual commands increases readability.

The Mop Command: The Mop command applies to member functions of a data type. For example,

```
list_item  append(E x);
/*{\Mop appends a new item \Litem{x} to list |\Mvar| and
returns it
(equivalent to |\Mvar.insert(x,\Mvar.last(),after)|).}*/
```

generates (assuming that Mvar has value L)

list_item	L.append($E\ x$)	appends a new item $\langle x \rangle$ to list L and returns it (equivalent to $L.insert(x, L.last(), after)$).

[2] In earlier versions of the preprocessor the choice between the two layout styles had to be done manually. We therefore had two versions of each manual command. The standard version selected three-column layout and the version with an appended character "l" selected two-row layout. You can still find manual commands Mopl and Mfuncl in many LEDA header files.

Note how the content of the first two columns is extracted from the code unit preceding the manual comment. Also note that we use member-function-call-syntax for the second column and that the function is applied to the canonical object of the type (which is the value of placeholder Mvar). We give some more examples.

```
list_item append(const E& x);
/*{\Mop      appends a new item \Litem{x} to
             list |\Mvar| and returns it\\
     (equivalent to |\Mvar.insert(x\,Mvar.last(),after)|).}*/
```

also produces the manual entry above. This reflects our view that a const-reference-parameter is equivalent to a value-parameter. The option `constref=yes` does not suppress const-ref pairs. The next function is long and hence is typeset in two-row layout.

```
list_item  insert(E x, list_item it, int direction = after);
/*{\Mop inserts a new item \Litem{x}  after or
         before item |it|.                  }*/
```

produces (assuming that Mvar has value L)

---

list_item *L*.insert(*E x*, *list_item it*, *int direction* = *after*)

inserts a new item $\langle x \rangle$ after or before item *it*.

In either layout style it may happen that the return type does not fit into a box of width typewidth. In this case we combine return type and function call into a single unit for which we allot a box of width typewidth + callwidth. For example,

```
two_tuple<int,int> strange();
/*{\Mop a strange function. }*/
```

produces (assuming that Mvar has value L)

---

two_tuple<*int*, *int*> *L*.strange() a strange function.

The Mbinop Command: Mbinop applies to *binary operators* defined as member functions.

```
integer  operator+(const integer& y);
/*{\Mbinop   returns |\Mvar + y|. }*/
```

produces (assuming that Mvar has value x)

---

integer $x + y$ returns $x + y$.

There are two facts worth noting about this output. First, we use operator-call-syntax for the second column. Second, we suppress the type of the argument y. The rule is as follows. For an operator of class T the type of any value argument of type T is not shown. The option `partypes=yes` turns off this behavior.

The Munop Command: Munop applies to *unary operators* defined as member functions.

```
integer  operator++(){....}
 /*{\Munop returns the value of |\Mvar| and increments it.}*/
```

produces (assuming that Mvar has value x)

integer	$++x$	returns the value of x and increments it.

We put the operator applied to the canonical variable into the second column. Of course, unary operators are typeset as either prefix or postfix operators as prescribed by the syntax of C++.

The Marrop Command: Marrop applies to the *array access operator*.

```
E&  operator[](list_item it)  { ... return ... }
/*{\Marrop returns a reference to the
           entry |it| of |\Mvar|.}*/
```

produces (assuming that Mvar has value L)

E&	L[*list_item it*]	returns a reference to the entry *it* of L.

The Mfunop Command: Mfunop applies to the *function call operator*.

```
string operator()(int i, int j)  const { return sub(i,j); }
/*{\Mfunop    returns the substring of |\Mvar| ... }*/
```

produces (assuming that Mvar has value s)

string	s(*int i*, *int j*)	returns the substring of s ...

The Mstatic Command: Mstatic applies to *static member functions*. For example, the type bigfloat has a static member *round_mode* that determines the current rounding mode. A static member function *set_round_mode* is used to set the rounding mode.

```
static void set_round_mode(rounding_modes m =TO_NEAREST);
{round_mode = m;}
/*{\Mstatic sets |round_mode| to |m|.}*/
```

produces (assuming that Mname has value bigfloat)

void	*bigfloat*::set_round_mode(*rounding_modes m* = *TO_NEAREST*)	
		sets *round_mode* to m.

The Mfunc Command: Mfunc applies to *non-member functions* of a data type.

```
friend integer abs(const integer& x);
/*{\Mfunc returns the absolute value of |x|.}*/
```

produces

integer	abs(*integer x*)	returns the absolute value of x.

Note that the `friend` qualifier does not appear in the manual. After all, it has nothing to do with the semantics of the operation but is only an information for the compiler.

The Mbinopfunc Command: Mbinopfunc applies to *binary operators* that are non-member functions. You have probably got the rule by now. Commands ending with op apply to members and commands ending with func apply to non-members.

```
friend string  operator+(const string& x, const string& y);
/*{\Mbinopfunc  returns the concatenation of |x| and |y|.}*/

friend ostream& operator<<(ostream& O, const string& s);
/*{\Mbinopfunc  writes string |s| to output stream |O|. }*/
```

produces

string	$x + y$	returns the concatenation of x and y.
ostream&	*ostream&* $O \ll s$	writes string s to the output stream O.

The Munopfunc Command: Munopfunc applies to *unary operators* that are nonmember functions.

```
friend integer  operator-(const integer& x)
/*{\Munopfunc     unary minus ... }*/
```

produces

integer	$-x$	unary minus ...

The Mconversion Command: Mconversion applies to *user-defined conversion operators*. The following definition within class integer

```
operator rational()
/*{\Mconversion converts an |\Mtype| to a rational.}*/
```

produces (assuming that Mvar has value x)

rational	x	converts an *integer* to a rational.

Invisible Functions: Sometimes there is the need to generate a manual entry for a function or operator that does not exist. A typical situation is as follows. A type A is derived from a type B and inherits a function from B. We want the function to appear in the manual page of type A but we do not want the function to appear in the header file (because type A inherits it and including it in the header file would obscure the situation). The solution is to put the function inside a comment, e.g.,

```
/* inherited
void sort_edges() { graph::sort_edges(); }
*/
/*{\Mop the edges of $G$ are sorted increasingly according
         to their contents. }*/
```

The begin and the end of the comment must be on separate lines. The starting line may contain a text that explains the situation.

Code Units with More than One Function Definition: The restriction that a code unit contains only one function definition is sometimes unnatural. An example is two closely related functions for which one wants to produce only one manual entry.

```
friend bool operator==(const string& x, const char* y);
friend bool operator==(const string& x, const string& y);
/*{\Mbinopfunc     true iff $x$ and $y$ are equal.}*/
```

produces

bool	*string x* == *string y*	true iff x and y are equal.

Another example is conditional definitions, e.g., the access function in the array data type which depends on the compiler flag LEDA_CHECKING_OFF.

```
#if defined(LEDA_CHECKING_OFF)
E&  operator[](int x) { return LEDA_ACCESS(E,v[x-Low]); }
#else
E&  operator[](int x) { return LEDA_ACCESS(E,entry(x)); }
#endif
/*{\Marrop     returns $A(x)$.\\
               \precond $a\le x\le b$.  }*/
```

produces (assuming that Mvar has value A)

E&	*A*[*int x*]	returns $A(x)$. *Precondition*: $a \le x \le b$.

If a code unit contains more than one function definition our preprocessor attempts to extract the *last* definition. It outputs the extracted definition on standard output (except with option `warnings=no`) and asks for an acknowledgment (except with option `ack=no`).

The Mimplementation Command: A command of the form

```
/*{\Mimplementation body}*/
```

produces the header line of the implementation part and typesets body. For example,

```
/*{\Mimplementation The data type |\Mtype| is realized
by doubly linked linear lists. All operations take
constant time except
for the following operations: |search| and |rank|
take linear time $O(n)$, ...
}*/
```

produces

---

5. Implementation

The data type *list* is realized by doubly linked linear lists. All operations take constant time except for the following operations: *search* and *rank* take linear time $O(n)$, ...

The Mexample Command: The Mexample command is used to produce the header line of the example part and to include program code into the manual. The simplest way to include program code is to use the verbatim environment of LaTeX.

```
/*{\Mexample The following little example illustrates
the list data type.
\begin{verbatim}
#include <LEDA/list.h>
main()
{
  list<string> L;
  L.append("hello world");
}

\end{verbatim} }*/
```

produces

---

6. Example

The following little example illustrates the list data type.

```
#include <LEDA/list.h>
main()
{
  list<string> L;
  L.append("hello world");
}
```

The Mtext Command: The Mtext command can be used to add arbitrary text to the manual. For example,

```
/*{\Mtext
\headerline{Additional Operations for two-dimensional Points}
The following operations are only available for points
in two-dimensional space.
We will not mention this precondition in the sequel.
}*/
```

produces

Additional Operations for two-dimensional Points

The following operations are only available for points in two-dimensional space. We will not mention this precondition in the sequel.

Generally,

```
/*{\Mtext  body  }*/
```

adds body to the document. The body is subject to placeholder substitution and C++ to LaTeX conversion. The Mtext command can be used to include arbitrary LaTeX commands into the output of the preprocessor. We did this already for the header line command in the example above. Another frequent use of the Mtext command is to change the values of the parameters governing the layout. For example

```
/*{\Mtext
\settowidth{\typewidth}{|void|}
\addtolength{\typewidth}{\colsep}
\computewidths
}*/
```

sets the width of the first column to the width of *void* plus the value of colsep, where colsep is predefined as 1.5em. The command `\computewidths` causes the recomputation of the dependent variable descriptwidth.

The Moptions Command: The Moptions command allows us to include preprocessor options directly into the header file. For example, the header file for LEDA's window type contains

```
/*{\Moptions
usesubscripts=yes
}*/
```

and hence this section of the LEDA-manual is typeset with subscripts, see also Section 14.2.4.

The Msubst Command: The Msubst command allows us to define additional placeholders. For example,

```
/*{\Msubst
int_type integer
quot_type rational
}*/
```

introduces the placeholders `int_type` and `quot_type` with values `integer` and `rational`, respectively.

14.2.3 *Warnings and Acknowledgments*

The preprocessor issues warnings and error messages and asks the user to acknowledge them. With the option `ack=no` no acknowledgments are necessary and the option `warnings=no` suppresses the warnings. One can also suppress warnings for a single manual comment, e.g.,

```
/*{\Moptions nextwarning=no }*/
point head();
point start();
/*{\Mop returns the start point of |\Mvar|}*/
```

suppresses the warning that there is more than one function definition in the current code section. We recommend running Lman with `warnings=yes` and `ack=yes` and using the mechanism above to turn off warnings individually.

14.2.4 *Subscripts*

Sometimes program variables are numbered and it would be nice to typeset the numbers as subscripts. The option `usesubscripts=yes` does exactly this. Within the context of this option `|x0|` is typeset as x_0 and `|x11|` is typeset as x_{11}. Note that the subscript rule is applied only to identifiers consisting of a single character. Thus `|diff1|` is still typeset as *diff1*.

14.2.5 *TEX macros*

We defined a collection of TEX-commands that facilitate the production of manual pages; they are contained in `MANUAL.mac` in LEDAROOT/Manual/tex.

Many data types in LEDA are defined in terms of items. We have adopted the convention that items are enclosed in angular braces. The command `\Litem` produces items. It takes a single argument and encloses it in angular braces. The argument is typeset in math-mode, i.e., `\Litem{x}` produces $\langle x\rangle$, `\Litem{x,y}` produces $\langle x, y\rangle$, and `\Litem{diff}` produces $\langle diff\rangle$. The last example shows that identifiers of length more than one should be enclosed in vertical bars, e.g., `\Litem{|diff|}` produces ⟨*diff*⟩.

The word *Precondition* appears frequently in manual pages; `\precond` produces it. The macro `\CC` produces C++. The command `\headerline{arg}` produces a header line, i.e., it prints its argument in boldface and disallows pagebreaks after the header line. The commands `\DLK` and `\DRK` produce `[[` and `]]`, respectively.

Vertical bars require some care. Recall that vertical bars have a special meaning (they bracket C++ text) and therefore we need to make special provisions to produce vertical bars in LaTeX-text produced by our preprocessor. The command `\Lvert` expands to a vertical bar, i.e., the preprocessor leaves it alone and its TeX-definition is `\def\Lvert{|}`. A frequent use of vertical bars in mathematical text is to denote absolute values. The command `\Labs` produces absolute values, e.g., `$x + \Labs{|diff|} + z$` produces $x+|\mathit{diff}|+z$. The commands `\Lvert` and `\Labs` can only be used in math-mode, i.e., in order to produce a | within text you need to write `$\Lvert$`.

`MANUAL.mac` also defines the LaTeX environment *manual*. This environment sets parindent to zero, parskip to 14pt and increases baselineskip slightly above its standard value. The manual is typeset in this environment.

The file `MANUAL.pagesize` in LEDAROOT/Manual/tex defines textwidth, textheight, topmargin, evensidemargin, and oddsidemargin. Values which work well with European a4-size paper and US legal-size paper are predefined in this file.

14.2.6 *Applying Lman to Web-Files*

Followers of literate programming do not split their implementations into h-files and c-files but combine them into a single file. This causes no problem for Lman as it ignores all but the manual commands and the code units preceding them.

A problem may arise if the web-system in use allows the user to put formatting instructions into the code chunks as, for example, CWEB does. In this case the manual extractor must purge the code of formatting instructions. The standard version of ext knows how to remove CWEB's formatting instructions. In order to adapt the manual extractor to another web-system which allows formatting instructions in code chunks you need to edit the code chunk <*purge code unit*...> in ext.nw. We have used Lman successfully on CWEB, noweb, and Lweb-files.

14.2.7 *Redirecting Output*

Lman and Ldoc write the extracted manual page to the file `outfile`. In the case of Ldoc the default value of `outfile` is equal to `basename.man` where `basename.lw` is the input file to Ldoc. In the case of Lman the outfile is some internal file. You may redirect the output to a different file by assigning to `outfile` in an Moptions command, e.g., after

```
/*{\Moptions outfile=type.man }*/
```

the output will be written to file `type.man`. This feature is useful for at least two purposes.

The first use is to generate several manual pages from the same source. This can be achieved by always directing the output to the appropriate man-file. There is a small inconvenience: LaTeX expects manual pages to be enclosed in the manual environment. However, the required `\begin{manual}` and `\end{manual}` commands are generated automatically only for the default outfile. So one needs to write:

```
/*{\Moptions outfile=type.man }*/
/*{\Mtext \begin{manual} }*/
   now come the commands than generate the manual
/*{\Mtext \end{manual} }*/
```

The second use of redirecting output is to rearrange the material within a single manual page. It is conceivable that one wants to use a different order of presentation in the manual page and in the implementation. Assume that the manual consists of two parts and that we want to arrange the two parts in reverse order in the manual and in the documentation. Write:

```
\section{The Manual Page}

\begin{manual}
\input{part1.man}
\input{part2.man}
\end{manual}

\section{Code}

/*{Moptions outfile=part2.man }*/
         the stuff that goes into part 2

/*{Moptions outfile=part1.man }*/
         the stuff that goes into part 1
```

14.2.8 *The Lman Options*

The behavior of Lman can be fine-tuned by options. A call `Lman` without arguments gives a short survey of all available options. Options are specified in assignment syntax `variable=value`. There must be no blank on either side of the equality sign. In the list of options to follow we list the default value of each option first.

size={12, 11, 10}: Determines the font size.

constref={no, yes}: Determines how const-ref parameters are displayed. With the no-option a const-ref parameter `const T& x` is displayed as a value parameter `T x` and with the yes-option it is displayed in full.

partypes={no, yes}: Determines how parameters of unary and binary operators are displayed. Consider, for example, an operator `+` of a class number. With the no-option the operator `operator+(number x, number y)` is displayed as `x + y` and with the yes-option it is displayed as `number x + number y`.

numbered={no, yes}: Determines whether the header line of the manual page is numbered. You probably want it numbered when the manual page becomes part of a larger document.

title={yes, no}: If title is set to no, the manpage comment produces no output.

warnings={no, yes}: Determines whether Lman gives warnings. You probably want to use the no-option when you inspect LEDA manual pages and the yes-option when you design manual pages yourself.

ack={no, yes}: Determines whether Lman asks for acknowledgments of warnings.

usesubscripts={no, yes}: Determines whether variables consisting of a single character followed by a number are displayed as subscripted variables.

filter={all, signatures, definition, creation, operations, implementation, example, opname}: Determines which part of the manual page is shown. The all-option shows the complete manual page, the signature-option shows the signatures of all operations of the data type, the next five options show only the corresponding section of the manual page, and the opname-option shows only the operation with the same name.
outfile={string}: Determines whether the TEX-file generated is only written on a temporary file (the default option) or on the file with name string.

latexruns={1, 0, 2}: Determines the number of LATEX runs used to produce the manual page. LATEX needs to be run twice if the manual page contains cross references.

xdvi={yes, no}: Determines whether the manual page is displayed by xdvi. If latexruns is at least one and xdvi is no then the resulting dvi-file is copied into file T.dvi in the working directory.

Lman can be customized by putting options in a file Lman.cfg in either the home directory or the working directory. Command line options take precedence over options in the working directory which in turn take precedence over options in the home directory.

14.3 Making a Manual: The Mkman Command

Many manual pages combined into a single document make a manual. We explain a simple mechanism to produce LEDA-style manuals. Assume that we want to produce a document consisting of a title page, an introduction, and the manual pages of types A and B. Assume also that the manual information about types A and B is contained in files with extension `ext`[3] in a common directory `dir` and that the working directory contains a master TeX-file as shown in Figure 14.8 and also a file `Introduction.tex`. The command

```
Mkman dir ext
```

cycles through all files `f.ext` in `dir` and calls

```
lextract f.ext /extract/f.tex
```

for each one of them. This creates files `extract/A.tex` and `extract/B.tex` after which the master file may be processed with LATEX.

All header files of LEDA are contained in the directory LEDAROOT/incl/LEDA and the master file for manual production is called MANUAL.tex and is contained in the directory LEDAROOT/Manual/MANUAL. Thus an execution of

```
Mkman $LEDAROOT/incl/LEDA h
latex MANUAL.tex
```

in the latter directory produces the dvi-file of the LEDA manual. Since LEDAROOT/incl/LEDA and h are the default values of the first and second argument of Mkman, respectively, the

[3] Typical extensions are h, nw, and lw.

```
\documentclass[12pt,a4paper]{book}
\usepackage{Lweb}
\begin{document}

\title {A Simple Manual}
\maketitle
\input{Introduction.tex}
\input{extract/A.tex}
\input{extract/B.tex}
\end{document}
```

Figure 14.8 A master tex-file for a simple manual.

```
#!/bin/csh -f
if ($1 == "") then
  set source = $LEDAROOT/incl/LEDA
  set ext = h
else
  set source = $1
  if ($2 == "") then
    set ext = h
  else
    set ext = $2
  endif
endif

\rm -r -f extract

mkdir extract

echo Extracting manual pages ...
echo " "

foreach f ($source/*.$ext)
   echo "extracting manual from $f"
   lextract $f  extract/`basename $f .$ext`.tex
end
```

Figure 14.9 The shell script Mkman for manual production.

first line may actually be abbreviated to Mkman. Figure 14.9 shows the shell script that realizes Mkman.

14.4 The Manual Directory in the LEDA System

The subdirectory Manual of the LEDA directory contains all files that are relevant for manual production, see Figure 14.10.

```
                 LEDAROOT
                    |
                  Manual
  ----------------|-----------------
  |    |      |      |       |     |
 DVI  MANUAL  cmd  contrib  noweb  tex
```

Figure 14.10 The subdirectory Manual of the LEDA directory.

- MANUAL contains the tex-sources for the LEDA-Manual.
- DVI contains the dvi-files obtained by applying Lman to all header files of the LEDA system. The dvi-files in DVI are accessed by the online manual viewer xlman.
- cmd contains the commands Lman, Mkman, ...
- contrib contains sources of contributions made by persons outside the LEDA group.
- noweb contains the noweb-sources for all programs used for manual production. In particular, the noweb-file `ext.nw` contains the Perl programs and shell scripts for lextract, Lman, Mkman, ...
- tex contains the TeX files required for manual production.

14.5 Literate Programming and Documentation

Many data types and algorithms of the LEDA system are documented in the literate programming system noweb [Ram94] and its LEDA-dialect Lweb. Ldoc and lweave are our tools to turn noweb- and Lweb-files into nice looking documents.

Literate programming advises to integrate specification, implementation, and documentation into a single file and to use tools (usually called *tangle* and *weave*) to extract program and to typeset documentation. Among the many literate programming systems we have used CWEB[KL93] and noweb[4]: our current favorite is noweb and its LEDA-dialect Lweb. We used Lweb to produce this book.

14.5.1 *Noweb and Lweb*

We start with a brief review of noweb, see also Section 2.7. Noweb provides commands *notangle* and *noweave* that can be applied to so-called noweb-files. A noweb-file foo.nw contains program source code interleaved with documentation. When notangle is given a noweb-file, it extracts the program and writes it to standard output, and when noweave is given a noweb-file it produces a LaTeX source on standard output.

[4] noweb can be obtained by anonymous ftp from CTAN, the Comprehensive TeX Archive Network, in directory web/noweb.

```
     ______ noweave________  LaTeX source
    |
    |
foo.nw
    |
    |______ notangle_______  program
```

A noweb-file is a sequence of *chunks*. A chunk is either a *documentation chunk* or a *code chunk*. Documentation chunks begin with a line that starts with an at-sign (@) followed by a space or newline. Code chunks begin with

<<code chunk name >>=

on a line by itself. Chunks are terminated by the beginning of another chunk or by the end of the file. Several code chunks may have the same name. Notangle concatenates their definitions to produce a single chunk. Code chunks contain source code and references to other code chunks.

Notangle extracts code by expanding one code chunk. In the expansion process code chunk definitions behave like macro definitions, i.e., if the definition of chunk XXX contains references to other code chunks then these chunks are also expanded, and so on.

Noweave produces a LaTeX source from a noweb-file. To this end it copies the documentation chunks verbatim to standard output (except for quoted code, see below) and it typesets code chunks in typewriter font. Note that this implies that documentation chunks starting with an @-sign followed by a newline-character start a new paragraph in the sense of LaTeX and that documentation chunks containing non-white stuff on the same line as the @-sign do not. Code may be *quoted* within documentation chunks by placing double square brackets (`[[...]]`) around it. Noweave typesets quoted code in typewriter font.

This completes our review of noweb. *Lweb* is our local dialect of noweb which we developed for the production of this book and for the documentation of the LEDA system. Lweb-files have extension `.lw`. Figure 14.11 shows an Lweb file and Figure 14.12 shows the result of applying lweave to it. The differences between Lweb and noweb are the following:

- Code can be quoted by either double square brackets (`[[...]]`) or vertical bars (| . . . |). Code quoted in double square brackets is set in typewriter font and code quoted in vertical bars is typeset in mathitalics font. This was already discussed in Section 14.2.2.
- Program examples can be included in documentation chunks by lines that start with @c. The text after the program example must start with an @-sign followed by a space-character or a newline-character.
- Empty lines in program chunks generate somewhat less vertical space than an empty line in a verbatim-like environment. This makes code chunks look better.
- Page breaks are forbidden between the first few and the last few lines of a code chunk.

```
        _______lweave_____________  LaTeX source
       |
       |
   foo.lw
       |
       |______ notangle___________  program
```

Lweb-files have extension .lw. Notangle applies also to Lweb-files and noweave is replaced by lweave; lweave is realized as a pair of pre- and postprocessor to noweave. The preprocessor handles the code quoted by vertical bars and the program examples and the postprocessor takes care of empty lines in code chunks. The implementation of lweave is part of ext.nw in LEDAROOT/Manual/noweb.

14.5.2 *Documentation*

Many classes and programs of the LEDA-system are documented using Lweb and this book is also an Lweb document. We recommend having at least the following major sections in a documentation:

- A preamble consisting of the title page, the table of contents, and maybe an abstract and an introduction.
- A manual page as discussed in the previous section.
- A section containing the header file augmented by manual comments so as to allow manual extraction.
- A section containing the c-file.
- A section containing test, example, or demo programs.

Figure 14.13 shows a simple Lweb-file stack.lw having the recommended structure. More substantial examples can be found in the subdirectory Lweb of the LEDAROOT directory.

14.5.3 *Ldoc*

Ldoc combines the functionality of Lman and lweave. A call

```
Ldoc XXX[.lw] options
```

produces a file XXX.man in the working directory and a temporary file temp.lw. The former file contains the manual and is essentially the file produced by Lman (except for the preamble and postamble required by LaTeX). The Lman-options constref, partypes, warnings, ack, and usesubscripts apply. The file temp.lw is obtained by the deletion of all manual comments (except for Mpreamble comments) from the input file. The option `delman=no` suppresses the deletion. The temporary file temp.lw is then sent through lweave and the result is moved to XXX.tex in the working directory.

We introduced an additional manual comment for the use with Ldoc, the Mpreamble comment. As far as Lman is concerned it is equivalent to the Mtext command, i.e., its

```
\documentclass[a4paper]{article}
\usepackage{Lweb}
\begin{document}
\subsubsection{Jordan Sorting}

We proceed to describe an implementation. Its global
structure is given by:

@c
<<include statements>>;
<<typedefs and global variables>>;
<<class point>>;
<<class bracket>>;
<<procedure Jordan sort>>;

@ As outlined above, we construct three data structures
simultaneously: the sorted list of the numbers processed so
far, call it |L|, and the
upper and lower tree of brackets. Each item of the
list |L| contains its abscissa (a |float|) and pointers
to the brackets in the two trees containing it.

<<class point>>=

class point{
private:
float abscissa;
bracket* bracket_in_upper_tree;
bracket* bracket_in_lower_tree;

public:
<<member functions of class point>>
}

@ A node of either tree corresponds to a bracket.
A bracket needs to know its two endpoints
(as items in the list |L|), its sorted sequence
of sub-brackets (a |sortseq<bracket*,>| which we
abbreviate as |children_list|),
and its position among its siblings (a |seq_item|).
\end{document}
```

Figure 14.11 An Lweb file: It is part of the section on sorted sequences of this book.

body is included into the produced tex-file after placeholder substitution and C++ to LaTeX conversion. Ldoc produces two output files, namely XXX.man and temp.lw. The output of Mpreamble commands is put into the latter file instead of the former. A typical use of the Mpreamble command is the definition of a LaTeX-command whose body should be subjected to C++ to LaTeX-conversion. The following example is taken from LEDA's geo_rep class.

We proceed to describe an implementation. Its global structure is given by:

⟨*include statements*⟩ ;
⟨*typedefs and global variables*⟩ ;
⟨*class point*⟩ ;
⟨*class bracket*⟩ ;
⟨*procedure Jordan sort*⟩ ;

As outlined above, we construct three data structures simultaneously: the sorted list of the numbers processed so far, call it L, and the upper and lower tree of brackets. Each item of the list L contains its abscissa (a *float*) and pointers to the brackets in the two trees containing it.

⟨*class point*⟩≡

```
class point{
private:
float abscissa;
bracket* bracket_in_upper_tree;
bracket* bracket_in_lower_tree;
public:
⟨member functions of class point⟩
}
```

A node of either tree corresponds to a bracket. A bracket needs to know its two endpoints (as items in the list L), its sorted sequence of sub-brackets (a *sortseq*<*bracket*∗, > which we abbreviate as *children_list*), and its position among its siblings (a *seq_item*).

Figure 14.12 The result of applying lweave + latex to the file of Figure 14.11.

```
/*{\Mpreamble
\newcommand{\grsummary}
{The class |geo_rep| is used to represent points, hyperplanes,
directions, and vectors. The latter ...}
}*/
```

14.5.4 *The Implementation of Ldoc*

Ldoc is based on the commands *lextract*, *ldel*, and *weave*, where weave is noweave for noweb and lweave for Lweb.

```
foo.[lw|nw|w] - ldel - foo-del.[lw|nw|w] - weave - foo.tex
  |                                                     |
lextract                                         \input{foo.man}
  |                                                     |
foo.man ----------------------------------------------
```

Ldoc first uses lextract to extract the manual and ldel to remove the manual comments, it then applies the appropriate weave command to the output of ldel, and it finally applies LaTeXand xdvi to the resulting file. All Lman options apply. In order to try out Ldoc copy

```
\documentclass[a4paper]{article}
\usepackage{Lweb}
\begin{document}
\title{Stack\\   |stack| }
\author{Kurt Mehlhorn}
\maketitle
\tableofcontents
\section{The Manual Page of Type Stack}
\input{stack.man}
@ \section{The Header File}
<<stack.h>>=  the file of Figure 1.2
@ \section{The Implementation}
<<stack.c>>= ...
@ \section{A Test Program}
<<stack-test.c>>= ...
@
\end{document}
```

Figure 14.13 The generic structure of a documentation.

sortseq.lw from LEDAROOT/Lweb to a directory where you have write-permission and then call Ldoc sortseq.

Bibliography

[KL93] D. Knuth and S. Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, 1993.

[Lam86] L. Lamport. *LaTeX*. Addison-Wesley, 1986.

[Ram94] N. Ramsey. Literate programming simplified. *IEEE Software*, pages 97–105, 1994.

[WS90] L. Wall and R.L. Schwartz. *Programming perl*. O'Reilly & Associates, 1990.

Index