Reusable Software
A Collection of C-Modules

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Reusable Software
A Collection of C-Modules

Josef Grosch

Aug. 4 1992

Report No. 30

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Abstract

A brief description of my personal collection of reusable modules written in C is given. The modules are oriented towards compiler construction. Originally, the modules have been written in MODULA-2.

1. Overview

The following modules exist currently (Aug. 1992):

Module	Task
Memory	dynamic storage (heap) with free lists
DynArray	dynamic and flexible arrays
StringMem	string memory
Idents	identifier table - unambiguous encoding of strings
Sets	sets of scalar values (without run time checks)
Positions	handling of source positions
Errors	error handler for parsers and compilers
Source	provides input for scanners
General	miscellaneous functions
System	machine dependent code
Time	access to cpu-time

2. Memory: dynamic storage (heap) with free lists

```
extern unsigned long MemoryUsed;
                        /* Holds the total amount of memory managed by  */
                        /* this module.                                 */

extern void     InitMemory      ();
                        /* The memory module is initialized.            */

extern char *   Alloc           (register unsigned long ByteCount);
                        /* Returns a pointer to dynamically allocated   */
                        /* space of size 'ByteCount' bytes.             */

extern void     Free            (unsigned long ByteCount, char * a);
                        /* The dynamically allocated space starting at  */
                        /* address 'a' of size 'ByteCount' bytes is     */
                        /* released.                                    */
```

3. DynArray: dynamic and flexible arrays

This module provides dynamic and flexible arrays. The size of a dynamic array is determined at run time. It must be passed to a procedure to create the array. The size of such an array is also flexible, that means it can grow to arbitrary size by repeatedly calling a procedure to extend it.

```
extern void MakeArray    (char * * ArrayPtr, unsigned long * ElmtCount,
                           unsigned long ElmtSize);
                         /* 'ArrayPtr' is set to the start address of a  */
                         /* memory space to hold an array of 'ElmtCount' */
                         /* elements each of size 'ElmtSize' bytes.      */

extern void ExtendArray  (char * * ArrayPtr, unsigned long * ElmtCount,
                           unsigned long ElmtSize);
                         /* The memory space for the array is increased  */
                         /* by doubling the number of elements.          */

extern void ReleaseArray (char * * ArrayPtr, unsigned long * ElmtCount,
                           unsigned long ElmtSize);
                         /* The memory space for the array is released.  */
```

Example:

```
# define         InitialSize    100
typedef ...      ElmtType;
typdef  ElmtType ArrayType [100000];
unsigned long    ActualSize = InitialSize;
ElmtType *       ArrayPtr;

MakeArray (& ArrayPtr, & ActualSize, sizeof (ElmtType);

(* Case 1: continuously growing array *)

Index := 0;
for (;;) {
   Index ++;
   if (Index == ActualSize)
      ExtendArray (& ArrayPtr, & ActualSize, sizeof (ElmtType);

   ArrayPtr [Index] = ... ;   /* access an array element */
    ... = ArrayPtr [Index];   /* "      "  "     "       */
}

(* Case 2: non-continuously growing array *)

for (;;) {
   Index = ... ;
   while (Index >= ActualSize)
      ExtendArray (& ArrayPtr, & ActualSize, size of(ElmtType);

   ArrayPtr [Index] = ... ;   /* access an array element */
    ... = ArrayPtr [Index];   /* "      "  "     "       */
}

ReleaseArray (& ArrayPtr, & ActualSize, sizeof (ElmtType);
```

4. StringMem: string memory

```
typedef unsigned short * tStringRef;

extern  tStringRef PutString    (register char * s, register cardinal length);
                        /* Stores string 's' in the string memory and   */
                        /* returns a reference to the stored string.    */

extern  void    StGetString     (register tStringRef r, register char * s);
                        /* Returns the string 's' from the string       */
                        /* memory which is referenced by 'r'.           */

/* extern cardinal LengthSt     (register tStringRef r); */
# define LengthSt(stringref) (* stringref)
                        /* Returns the length of the string 's'         */
                        /* which is referenced by 'r'.                  */

extern  bool    IsEqualSt       (tStringRef r, register char * s);
                        /* Compares the string referenced by 'r' and    */
                        /* the string 's'.                              */
                        /* Returns true if both are equal.              */

extern  void    WriteString     (FILE * f, tStringRef r);
                        /* The string referenced by 'r' is printed on   */
                        /* the file 'f'.                                */

extern  void    WriteStringMemory ();
                        /* The contents of the string memory is printed */
                        /* on standard output.                          */

extern  void    InitStringMemory ();
                        /* The string memory is initialized.            */
```

5. Idents: identifier table - unambiguous encoding of strings

```
typedef cardinal        tIdent;

extern  tIdent  NoIdent; /* A default identifer (empty string)          */

extern  tIdent  MakeIdent       (char * string, cardinal length);
                        /* The string (of length) is mapped to a unique */
                        /* identifier (an integer) which is returned.   */

extern  void    GetString       (tIdent ident, char * string);
                        /* Returns the string whose identifier is 'ident'.*/

extern  tStringRef GetStringRef (tIdent ident);
                        /* Returns a reference to the string identified */
                        /* by 'ident'.                                  */

extern  tIdent  MaxIdent        ();
                        /* Returns the currently maximal identifier.    */

extern  void    WriteIdent      (FILE * file, tIdent ident);
                        /* The string encoded by the identifier 'ident' */
                        /* is printed on the file.                      */

extern  void    WriteIdents     ();
                        /* The contents of the identifier table is      */
                        /* printed on the standard output.              */

extern  void    InitIdents      ();
                        /* The identifier table is initialized.         */

extern  void    WriteHashTable  ();
```

6. Sets: sets for scalar values

The following module provides operations on sets of scalar values. The elements of the sets can be of the types int, unsigned, char, or long. The size of the sets, that is the range the elements must lie in, is not restricted. The elements can range from 0 to 'MaxSize', where space for arbitrary large sets.

The sets are implemented as bit vectors (long []) plus some additional information to improve performance. So don't worry about speed too much because procedures like Select, Extract, or Card are quite efficient. They don't execute a loop over all potentially existing elements always. This happens only in the worst case.

```
# define BitsPerBitset          32
# define LdBitsPerBitset        5
# define MaskBitsPerBitset      0x0000001f

# define IsElement(Elmt, Set) ((int) ((Set)->BitsetPtr [(Elmt) >> LdBitsPerBitset] \
                                    << ((Elmt) & MaskBitsPerBitset)) < 0)
# define Size(Set)                  ((Set)->MaxElmt)
# define Select(Set)                Minimum (Set)
# define IsNotEqual(Set1, Set2)     (! IsEqual (Set1, Set2))
# define IsStrictSubset(Set1, Set2) (IsSubset (Set1, Set2) && \
                                     IsNotEqual (Set1, Set2))

typedef long    BITSET          ;

typedef struct  {
      cardinal  MaxElmt         ;
      cardinal  LastBitset      ;
      BITSET *  BitsetPtr       ;
      short     Card            ;
      cardinal  FirstElmt       ;
      cardinal  LastElmt        ;
   } tSet;

extern void     MakeSet         (tSet * Set, cardinal MaxSize);
extern void     ReleaseSet      (tSet * Set);
extern void     Union           (tSet * Set1, tSet * Set2);
extern void     Difference      (tSet * Set1, tSet * Set2);
extern void     Intersection    (tSet * Set1, tSet * Set2);
extern void     SymDiff         (tSet * Set1, tSet * Set2);
extern void     Complement      (tSet * Set);
extern void     Include         (tSet * Set, cardinal Elmt);
extern void     Exclude         (tSet * Set, cardinal Elmt);
extern cardinal Card            (tSet * Set);
/* extern cardinal      Size            (tSet * Set); */
extern cardinal Minimum         (tSet * Set);
extern cardinal Maximum         (tSet * Set);
/* extern cardinal      Select          (tSet * Set); */
extern cardinal Extract         (tSet * Set);
extern bool     IsSubset        (tSet * Set1, tSet * Set2);
/* extern bool  IsStrictSubset  (tSet * Set1, tSet * Set2); */
extern bool     IsEqual         (tSet * Set1, tSet * Set2);
/* extern bool  IsNotEqual      (tSet * Set1, tSet * Set2); */
/* extern bool  IsElement       (cardinal Elmt, tSet * Set); */
extern bool     IsEmpty         (tSet * Set);
extern bool     Forall          (tSet * Set, bool (* Proc) ());
extern bool     Exists          (tSet * Set, bool (* Proc) ());
extern bool     Exists1         (tSet * Set, bool (* Proc) ());
extern void     Assign          (tSet * Set1, tSet * Set2);
extern void     AssignElmt      (tSet * Set, cardinal Elmt);
extern void     AssignEmpty     (tSet * Set);
extern void     ForallDo        (tSet * Set, void (* Proc) ());
extern void     ReadSet         (FILE * File, tSet * Set);
extern void     WriteSet        (FILE * File, tSet * Set);
extern void     InitSets        ();
```

Two parameters of type ’tSet’ passed to one of the above procedures must have the same size, that is they must have been created by passing the same value ’MaxSize’ to the procedure ’MakeSet’. A parameter representing an element (of type CARDINAL or equivalent) passed to one of the above procedures must have a value between 0 and ’MaxSize’ of the involved set which is the other parameter passed. If the two conditions above, which can be verified at programming time, don't hold then strange things will happen, because there are no checks at run time, of course.

The following table explains the semantics of the set operations:

Procedure	Semantics
MakeSet	allocates space for a set to hold elements ranging from 0 to 'MaxSize'.
ReleaseSet	releases the space taken by a set.
Union	Set1 := Set1 ∪ Set2
Difference	Set1 := Set1 - Set2
Intersection	Set1 := Set1 ∩ Set2
SymDiff	Set1 := Set1 Set2 (* corresponds to exclusive or *)
Complement	Set := { 0 .. MaxSize } - Set
Include	Set := Set ∪ { Elmt }
Exclude	Set := Set - { Elmt }
Card	returns number of elements in Set
Size	returns 'MaxSize' given at creation time
Minimum	returns smallest element x from Set
Maximum	returns largest element x from Set
Select	returns arbitrary element x from Set
Extract	returns arbitrary element x from Set and removes it from Set
IsSubset	Set1 ⊆ Set2
IsStrictSubset	Set1 ⊂ Set2
IsEqual	Set1 = Set2
IsNotEqual	Set1 ≠ Set2
IsElement	Elmt ∈ Set
IsEmpty	Set = Ø
Forall	∀ e ∈ Set : Proc (e) /* predicate Proc must hold for all elements */
Exists	∃ e ∈ Set : Proc (e) /* predicate Proc must hold for at least 1 element */
Exists1	| { e ∈ Set : Proc (e) } | = 1
Assign	Set1 := Set2
AssignElmt	Set1 := { Elmt }
AssignEmpty	Set1 := Ø

ForallDo	FOR e := 0 TO MaxSize DO
	    IF e ∈ Set THEN Proc (e); END;
	END;

ReadSet	read external representation of a set from file 'tFile'.
WriteSet	write external representation of a set to file 'tFile'. Example output: { 0 5 6 123}

7. Positions: handling of source positions

A simple representation of the position of tokens in a source file consisting of a line and a column field. This module should be copied and tailored to the user's needs, if necessary. Modifications may be necessary if the type SHORTCARD is to small to count the lines or an extra field is needed to describe the source file.

```
typedef struct { unsigned short Line, Column; } tPosition;

extern tPosition NoPosition;
                        /* A default position (0, 0).                  */

extern int  Compare       (tPosition Position1, tPosition Position2);
                        /* Returns -1 if Position1 < Position2.         */
                        /* Returns  0 if Position1 = Position2.         */
                        /* Returns  1 if Position1 > Position2.         */

extern void WritePosition (FILE * File, tPosition Position);
                        /* The 'Position' is printed on the 'File'.     */
```

8. Errors: error handler for parsers and compilers

This module is needed by parsers generated with the parser generators *lalr* or *ell*. It can also b used to report error messages found during scanning or semantic analysis. Note: This module has to be copied, too, if the module *Positions* is copied and modified because it depends upon this module.

```
# define xxNoText                0
# define xxSyntaxError           1       /* error codes          */
# define xxExpectedTokens        2
# define xxRestartPoint          3
# define xxTokenInserted         4
# define xxTooManyErrors         5

# define xxFatal                 1       /* error classes        */
# define xxRestriction           2
# define xxError                 3
# define xxWarning               4
# define xxRepair                5
# define xxNote                  6
# define xxInformation           7

# define xxNone                  0
# define xxInteger               1       /* info classes         */
# define xxShort                 2
# define xxLong                  3
# define xxReal                  4
# define xxBoolean               5
# define xxCharacter             6
# define xxString                7
# define xxSet                   8
# define xxIdent                 9
```

```
extern void (* Errors_Exit) ();
                        /* Refers to a procedure that specifies         */
                        /* what to do if 'ErrorClass' = Fatal.          */
                        /* Default: terminate program execution.        */

extern void StoreMessages (bool Store);
                        /* Messages are stored if 'Store' = TRUE        */
                        /* for printing with the routine 'WriteMessages'*/
                        /* otherwise they are printed immediately.      */
                        /* If 'Store'=TRUE the message store is cleared.*/

extern void ErrorMessage  (int ErrorCode, int ErrorClass, tPosition Position);
                        /* Report a message represented by an integer   */
                        /* 'ErrorCode' and classified by 'ErrorClass'.  */

extern void ErrorMessageI (int ErrorCode, int ErrorClass, tPosition Position,
                           int InfoClass, char * Info);
                        /* Like the previous routine with additional    */
                        /* information of type 'InfoClass' at the       */
                        /* address 'Info'.                              */

extern void Message       (char * ErrorText, int ErrorClass, tPosition Position);
                        /* Report a message represented by a string     */
                        /* 'ErrorText' and classified by 'ErrorClass'.  */

extern void MessageI      (char * ErrorText, int ErrorClass, tPosition Position,
                           int InfoClass, char * Info);
                        /* Like the previous routine with additional    */
                        /* information of type 'InfoClass' at the       */
                        /* address 'Info'.                              */

extern void WriteMessages (FILE * File);
                        /* The stored messages are sorted by their      */
                        /* source position and printed on 'File'.       */
```

9. Source: provides input for scanners

This module is needed by scanners generated with the scanner generator *rex*.

```
extern int  BeginSource  (char * FileName);

   /*
      BeginSource is called from the scanner to open files.
      If not called input is read form standard input.
   */

extern int  GetLine      (int File, char * Buffer, int Size);

   /*
      GetLine is called to fill a buffer starting at address 'Buffer'
      with a block of maximal 'Size' characters. Lines are terminated
      by newline characters (ASCII = 0xa). GetLine returns the number
      of characters transferred. Reasonable block sizes are between 128
      and 2048 or the length of a line. Smaller block sizes -
      especially block size 1 - will drastically slow down the scanner.
   */

extern void CloseSource  (int File);

   /*
      CloseSource is called from the scanner at end of file respectively
      at end of input. It can be used to close files.
   */
```

10. General: miscellaneous functions

```
# define Min(a,b) ((a <= b) ? a : b)
                        /* Returns the minimum of 'a' and 'b'.          */
# define Max(a,b) ((a >= b) ? a : b)
                        /* Returns the maximum of 'a' and 'b'.          */

extern cardinal         Log2 (register unsigned long x);
                        /* Returns the logarithm to the base 2 of 'x'.  */
extern unsigned long    Exp2 (register cardinal x);
                        /* Returns 2 to the power of 'x'.               */
```

11. System: machine dependent code

This module provides a few machine dependent operations.

```
/* interface for machine dependencies */

# define tFile int

/* binary IO */

extern tFile    OpenInput       (char * FileName);
                        /* Opens the file whose name is given by the    */
                        /* string parameter 'FileName' for input.       */
                        /* Returns an integer file descriptor.          */

extern tFile    OpenOutput      (char * FileName);
                        /* Opens the file whose name is given by the    */
                        /* string parameter 'FileName' for output.      */
                        /* Returns an integer file descriptor.          */

extern int      Read            (tFile File, char * Buffer, int Size);
                        /* Reads 'Size' bytes from file 'tFile' and     */
                        /* stores them in a buffer starting at address  */
                        /* 'Buffer'.                                    */
                        /* Returns the number of bytes actually read.   */

extern int      Write           (tFile File, char * Buffer, int Size);
                        /* Writes 'Size' bytes from a buffer starting   */
                        /* at address 'Buffer' to file 'tFile'.         */
                        /* Returns the number of bytes actually written.*/

extern void     Close           (tFile File);
                        /* Closes file 'tFile'.                         */

extern bool IsCharacterSpecial  (tFile File);
                        /* Returns TRUE when file 'tFile' is connected  */
                        /* to a character device like a terminal.       */


/* calls other than IO */

extern char *   SysAlloc        (long ByteCount);
                        /* Returns a pointer to dynamically allocated   */
                        /* memory space of size 'ByteCount' bytes.      */
                        /* Returns NIL if space is exhausted.           */

extern long     Time            ();
                        /* Returns consumed cpu-time in milliseconds.   */

extern int      GetArgCount     ();
                        /* Returns number of arguments.                 */

extern void     GetArgument     (int ArgNum, char * Argument);
                        /* Stores a string-valued argument whose index  */
                        /* is 'ArgNum' in the memory area 'Argument'.   */

extern void     PutArgs         (int Argc, char * * Argv);
                        /* Dummy procedure that passes the values       */
                        /* 'argc' and 'argv' from Modula-2 to C.        */

extern int      ErrNum          ();
```

```
                        /* Returns the current system error code.       */

extern int      System          (char * String);
                        /* Executes an operating system command given   */
                        /* as the string 'String'. Returns an exit or   */
                        /* return code.                                 */

extern void     Exit            (int Status);
                        /* Terminates program execution and passes the  */
                        /* value 'Status' to the operating system.      */

extern void     BEGIN_System    ();
                        /* Dummy procedure with empty body.             */
```

12. Time: access to cpu-time

```
extern int      StepTime        ();
                        /* Returns the sum of user time and system time */
                        /* since the last call to 'StepTime' in milli-  */
                        /* seconds.                                     */

extern void     WriteStepTime   (char * string);
                        /* Writes a line consisting of the string       */
                        /* 'string' and the value obtained from a call  */
                        /* to 'StepTime' on standard output.            */
```

Contents