Dynamic Partition of a Linear Store

WSTORE version 2021-11-17

M. Botje*
Nikhef, Science Park 105, 1098XG Amsterdam, the Netherlands

December 20, 2021

Abstract

The WSTORE package partitions a linear array into sets of one- or multi-dimensional tables. Routines are provided that dynamically partition the store, give access to the data in the store, and to navigate in the store. It is also possible to dump table-sets to disk and read these back-in.

*email m.botje@nikhef.nl

Contents

1 Introduction

The WSTORE package partitions a linear array into sets of tables. In this write-up we will call such a partitioned array a **workspace**. Dynamically storing tables in one or more workspaces gives great flexibility to large FORTRAN programs—like QCDNUM—while the in-house control over the table-indexing offers many opportunities for very fast data access. Although intended for code written in FORTRAN77, the WSTORE routines can also be called from a C++ program.

A minor drawback is that in FORTRAN the workspace must be allocated beforehand at compilation time. This is not the case in C++ which supports run-time allocation.

To show how the partitioning works in WSTORE, let us first declare a double precision array `w(n)` in FORTRAN, or `w[n]` in C++.

`double w[n]`	

The next step is to turn this array into a workspace.

`iws_wsinit`	`wsh`	`tsh`	`0`	

This call created two headers, one for the entire workspace (`wsh`), and one (`tsh`) for the first table-set in this workspace, which is empty of course. Note also the appearance of a trailer word (`0`). Now we can fill the first table-set with one or more tables.

`iws_wtable`	`wsh`	`tsh`	`table`	`table`	`0`	

When finished with the set we can create a new table-set,

`iws_newset`	`wsh`	`tsh`	`table`	`table`	`tsh`	`0`	

and fill it with one or more tables.

`iws_wtable`	`wsh`	`tsh`	`table`	`table`	`tsh`	`table`	`table`	`0`	

We can continue with this until the workspace is full (error message).

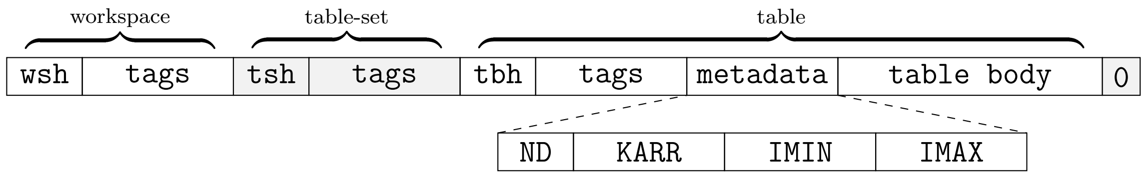
All the routines above return an integer array-index (pointer) indicating at which position in the workspace the newly created object is located. These pointers then serve to address the object later on (note that they are *not* C++ pointers).

In the next section we describe the workspace layout in a bit more detail.

2 Workspace layout

In the previous section we have mentioned that a workspace is organised into sets of tables. Therefore a table cannot exist on its own; it is always the member of a set. We have also indicated how to dynamically create a set and populate it with one or more tables—given that there is enough space of course. There is a routine in WSTORE to delete trailing objects from a workspace, but not embedded objects.

Below we show the layout of a workspace with one table-set and one table.



All three objects have a header field (`wsh`, `tsh`, `tbh`) and a tag field of a size set by the user at the initialisation of the workspace. In the first word of a header is stored a marker that identifies the object. The *workspace* marker is some version number that may change if a new version of WSTORE invalidates the current workspace layout. In the second word of a header is stored the distance (in words) of the object to the workspace root. This enables you to deal with index shifts that occur when an object is passed as an argument to a routine.

The workspace header `wsh` also contains a `null` word where you can store the return value of functions that fail to execute (default `1.D20`), and a drain word that can serve as a data sink.

Apart from a header and a tag field, a *table* contains a metadata field and a table body (contents). The metadata describe the table structure: the number of dimensions `ND`, an array of `ND`+1 pointer coefficients `K(i)`, and two arrays of size `ND` with the minimum and maximum index values. Note that all metadata are integers stored in double precision.

Let `i`, `j`, `k`, ... be table indices, all within their respective ranges. The *local* address (*i.e.* relative to the table address) of `T(i,j,k,...)` is then given by

```
ia = K(0) + K(1)*i + K(2)*j + K(3)*k + ...
```

To get the *workspace* address of `T(i,j,k,...)` the table address must be added to `ia`.

In the headers one word is reserved to store a structural fingerprint. A table fingerprint is a hash of the metadata. A table-set fingerprint is a hash of the header size, tag size (these are the same for all objects) and all table fingerprints. The hashes are made with the Pearson hash-function in MBUTIL. By comparing fingerprints you can quickly check if two objects have the same structure (they may have different contents). Note that fingerprints are integers stored as double precision numbers in the workspace headers.

The workspace itself also has a fingerprint which is a unique time-stamp: if `w1` and `w2` have equal fingerprints then this means that they are referring to the same workspace.

All objects—including the workspace itself—have a tag field of a size defined by the user at workspace initialisation. Tags can be used to store attributes (*e.g.* the particle code of a pdf table), or to build object hierarchies by storing pointers in one or more tags. For instance it is a good idea to store *local* table addresses in the table-set tag field so that only table-set addresses have to be remembered.

3 The WSTORE package

The WSTORE package is written in FORTRAN77 but interfaces are provided so that all FORTRAN routines can be called from a C++ program. The C++ wrappers reside in the namespace `WSTORE` and the routine names must be given in mixed case, as in this write-up. We refer to the QCDNUM manual for more on C++ interfaces.

The syntax of the WSTORE calls for a workspace `w` is as follows

`call xws_Name ( w, arguments )`	`WSTORE::xws_Name ( w, arguments );`

where `x` = `s` for subroutines and `x` = `l`, `i`, `r` or `d` for logical, integer, real and double-precision functions, respectively. Floating-point arguments are in double precision and input numbers must, in FORTRAN, be given in double precision format like `2.5D0` instead of `2.5`. In C++ the input format is free since the data-type is specified in the function prototype and the conversion is done automatically, if necessary.

In FORTRAN the (default) array indexing starts at `1` but in C++ at `0` so that the workspace root-address is set to `1` (`0`) in FORTRAN (C++). As a consequence, addresses will differ by one unit in the two languages. Routines that create an object in `w` return the array index `ia` of the first word of that object. The object is then later on referred to by passing its address `ia`.

Finally, the call `ivers = iws_Version()` gives you the current WSTORE version number.

4 Create a workspace

Here are the routines to initialise a workspace and fill it with objects. These routines are robust and cause a program abort (with an error message) if something is wrong. In particular, they will tell you how many words are needed in case `w` runs out of space.

In the following we will prefix output arguments by an ampersand (`&`) and denote the root-address (`1` in FORTRAN and `0` in C++) by `iaR`, a table-set address by `iaS` and a table address by `iaT`. Some routines accept addresses of different type which we will indicate by combinations like `iaRST`, *etc.* To not clutter the notation we use below the shorthand `w` for `w(iaR)` which is, in fact, allowed in standard FORTRAN77.

```
iaS = iws_WsInit ( w, nw, nt, 'comment' )   call sws_SetWsN ( w, nw )
```

Convert a double precision array into a workspace and create the first (empty) table-set.

`w`	Double precision array, dimensioned to `nw` in the calling routine.
`nw`	Size of `w` as declared in the calling routine.
`nt`	Size of the tag field (same for each object in the workspace).
`'comment'`	Optional comment line to be printed when `w` runs out of space.
`iaS`	Set, on exit, to the first table-set address.

If `w` is stored in a dynamic C++ array then `sws_SetWsN` should be called after each change in size to keep the size information in the workspace header up-to-date.

```
iaT = iws_WTable ( w, imin, imax, ndim )
```

Add a table to the current table-set in the workspace `w`.

`imin`, `imax`	Index limits dimensioned to at least `ndim` in the calling routine with, for each index, `imin(i)` < `imax(i)`.
`ndim`	Number of dimensions of the table [1–25].
`iaT`	Set, on exit, to the address of the new table object.

```
iaS = iws_NewSet ( w )
```

Create a new table-set. Acts as a do-nothing if a new (empty) set already exists. On exit, `iaS` is set to what is now the current table-set address.

```
iaST2 = iws_WClone ( w1, iaST1, w2 )
```

Clone a table-set or table with address `iaST1` in `w1` to the workspace `w2` (which can be the same as `w1`). On exit, `iaST2` is the address of the cloned object in `w2`. Note that the entire object is appended to `w2`, including fingerprints, tags and contents.

```
call sws_TbCopy ( w1, iaT1, w2, iaT2, itag )
```

Copy the contents of a table in `w1`, with address `iaT1`, to an existing table in `w2`, with address `iaT2`. Set `itag` = `1` (`0`) to (not) copy also the tag field. The routine checks beforehand that the source and target tables have the same dimension and index ranges. The workspace `w2` can be the same as `w1`.

```
call sws_WsWipe ( w, iaRST )
```

Wipe `w`, starting at object `iaRST`. If `iaRST` is the root-address the workspace will be in a state as after the call to `iws_WsInit`.

```
call sws_TsDump ( 'filename', key, w, iaS, &ierr )
```

Dump a table-set (including tags) with address `iaS` to disk. The input integer `key` is also dumped. You can set the key to zero, or to some kind of stamp (*e.g.* a hash-code).

`ierr` =	`0`	Table-set successfully dumped.
	`-1`	Problem to open or write the output file.

Acts as a do-nothing upon error. Note that a dump is unformatted and cannot be exchanged across platforms, and also that it cannot contain more than one table-set.

iaS = iws_TsRead ('filename', key, w, &ierr)

Read a table-set (including tags)[1] from disk and append it to `w`, at address `iaS`. If you enter a non-zero `key` value then it must match the key on the file; such a key-check protects against reading a wrong or outdated file.

`ierr` =	`0`	Table-set successfully read.
	`-1`	Problem to open or read the input file.
	`-2`	Incompatible input file (wrong key, tag field size or WSTORE version).

The routine acts as a do-nothing upon error, in which case `iaS` is undefined.

call sws_WsMark (&mws, &mset, &mtab)

In the first word of an object is stored a specific marker. This routine returns the marker-word values of a workspace (`mws`), table-set (`mset`), or table (`mtab`).

ntab = iws_TbSize (imin, imax, ndim)   nh = iws_HdSize()

Compute the size of a table object (without header and tags). The arguments `imin`, `imax` and `ndim` are as for `iws_WTable`. The workspace size can be pre-computed from

$$\mathtt{nw} = 1 + \mathtt{hskip} + \sum_{\text{sets}} \left[\mathtt{hskip} + \sum_{\text{tables}} (\mathtt{hskip} + \mathtt{ntab}) \right] \quad \text{with} \quad \mathtt{hskip} = \mathtt{nh} + \mathtt{nt}.$$

Here `nh` is the header size (from `iws_HdSize`), `nt` the tag field size (defined by you in the call to `iws_WsInit`) and `ntab` a table size (from `iws_TbSize`).

In this way you can compute the size of a C++ workspace before creating it dynamically.

5 Query a workspace

Below we list the functions to query a workspace. We mention here that the query functions are free-running without verifying that the input is correct. This is to avoid slow-down by unnecessary checks; in fact, it is trivial to pack a few routines together into a wrapper which is robust, if so desired.

Function	Description
	Workspace
`iws_IaRoot()`	Returns `1` for FORTRAN and `0` for C++
`iws_IsaWorkspace(w)`	Returns `1` (`0`) if `w` is (not) a workspace
`iws_SizeOfW(w)`	Total size of the array `w`
`iws_WordsUsed(w)`	Number of words used (without trailer)
`iws_Nheader(w)`	Number of header words (same for all objects)
`iws_Ntags(w)`	Number of tag words (same for all objects)

continued on next page

[1]Remember to re-set tags that contain links to other objects in the workspace.

continued from previous page

`iws_HeadSkip(w)`	Size of header + tag field	
`iws_IaDrain(w)`	Address of the drain word	(1)
`iws_IaNull(w)`	Address of the null word	
Object (workspace, table-set or table)		
`iws_ObjectType(w,ia)`	Object type	(2)
`iws_ObjectSize(w,ia)`	Size of object	
`iws_Nobjects(w,ia)`	Number (`n`) of objects in object `ia` (`0` = empty)	
`iws_ObjectNumber(w,ia)`	Serial number of object [`1`,`n`]	
`iws_FingerPrint(w,ia)`	Fingerprint of object	
`iws_IaFirstTag(w,ia)`	Address of the first tag word of an object	
Table		
`iws_TableDim(w,ia)`	Number of table dimensions	
`iws_IaKARRAY(w,ia)`	Address of the first word of `KARRAY`	
`iws_IaIMIN(w,ia)`	Address of the first word of `IMIN`	
`iws_IaIMAX(w,ia)`	Address of the first word of `IMAX`	
`iws_BeginTbody(w,ia)`	Address of the first word of the table-body	
`iws_EndTbody(w,ia)`	Address of the last word of the table-body	

(1) All addresses returned by the query routines are absolute addresses in `w`. They differ by one unit in in FORTRAN and C++.

(2) Object types are: not-an-object (`0`), workspace (`1`), table-set (`2`) and table (`3`).

A listing of the workspace tree can be obtained from a call to `sws_WsTree(w)`. A call to `sws_WsHead(w,ia)` prints a dump of the object header.

6 Example of a table-set

In this section we present a table-set with a 3-dimensional table of 50 bins in x, 25 bins in μ^2, and a third index for $n_\mathrm{f} = (3, 4, 5, 6)$. Two 1-dimensional tables hold the bin-limits in x (51 limits) and μ^2 (26 limits). The *local* addresses of the tables are stored in the tag field of the table-set. Here is the FORTRAN code to create such a set.

```
      integer function mytab(w)
      implicit double precision (a-h,o-z)
      dimension w(*), imi(3), ima(3)
      data imi/1,1,3/, ima/50,25,6/
      ias      = iws_NewSet(w)
      iax      = iws_WTable(w,imi(1),ima(1)+1,1)
      iaq      = iws_WTable(w,imi(2),ima(2)+1,1)
      ixq      = iws_WTable(w,imi,ima,3)
      iat      = iws_IaFirstTag(w,ias)
      w(iat)   = dble(iax-ias)
      w(iat+1) = dble(iaq-ias)
      w(iat+2) = dble(ixq-ias)
      mytab    = ias
      return
      end
```

It is important to store not the global but the local addresses because these are preserved when the table-set is cloned or read back from disk.

Here is a routine to extract addresses and pointer coefficients from the table-set `ias`.

```
      subroutine tabinfo(w,ias,iax,iaq,ixq,k3)
      implicit double precision (a-h,o-z)
      dimension w(*), k3(0:3)
      iat      = iws_IaFirstTag(w,ias)
      itx      = int(w(iat))+ias
      itq      = int(w(iat+1))+ias
      ixq      = int(w(iat+2))+ias
      iax      = iws_BeginTbody(w,itx)-1
      iaq      = iws_BeginTbody(w,itq)-1
      ikk      = iws_IaKARRAY(w,ixq)
      k3(0)    = int(w(ikk))
      k3(1)    = int(w(ikk+1))
      k3(2)    = int(w(ikk+2))
      k3(3)    = int(w(ikk+3))
      return
      end
```

In the code below we print the limits and contents of a bin in the 3-dimensional table. Also shown in the snippet is an inline pointer function for the table.

```
      IAijk(i,j,k) = k3(0)+k3(1)*i+k3(2)*j+k3(3)*k+ixq
      ..
      call tabinfo(w,ias,iax,iaq,ixq,k3)
      write( .. ) 'Limits of x-bin 10 :', w(iax+10),w(iax+11)
      write( .. ) 'Limits of q-bin  5 :', w(iaq+5),w(iaq+6)
      write( .. ) 'Value of T(10,5,4) :', w(IAijk(10,5,4))
```

7 Pointer functions

A pointer function gives the address of a table element as a function of the indices. In the previous section we have shown an inline 3-dim pointer function but WSTORE also provides a general one, with a boundary check on the indices.

```
iaddr = iws_Tpoint ( w, iaT, index, n )
```

`w`	Workspace.
`iaT`	Address of a table object in the workspace.
`index`	Array, dimensioned to at least the number of dimensions `ndim` of the table. The first `ndim` elements must be set to in-range index values.
`n`	Dimension of `index` as declared in the calling routine.

This function is rather slow because it checks everything and is also a bit clumsy to use—with indices stored in an array. Thus it is better to write your own fast pointer functions for n-dimensional tables and just use `iws_Tpoint` to verify that they are correct.

Here is an example that addresses a 3-dimensional table. First we write a small routine that copies the pointer coefficients to an integer array `kk` to avoid, as much as possible, double-to-integer conversions. Also stored is the table fingerprint to ensure that we load the correct coefficients in our pointer function. Below we list both FORTRAN and C++.

```fortran
subroutine K3(w, ia, kk)
dimension kk(0:4)
double precision w(*)
iak   = iws_IaKARRAY(w,ia)
kk(0) = int(w(iak))
..
kk(3) = int(w(iak+3))
kk(4) = iws_FingerPrint(w,ia)
return
end
```

```cpp
void K3(double *w, int ia, int (&kk)[5]) {
int iak   = iws_IaKARRAY(w,ia);
kk[0]     = int(w[iak]);
..
kk[3]     = int(w[iak+3]);
kk[4]     = iws_FingerPrint(w,ia);
}
```

Now we can write our fast pointer function $P(i, j, k)$ of 3 indices.

```fortran
integer function iP3(w, ia, i, j, k)
double precision w(*)
dimension kk(0:4)
save kk
ifp = iws_FingerPrint(w,ia)
if(kk(4).ne.ifp) call K3(w, ia, kk)
ip  = kk(0)+kk(1)*i+kk(2)*j+kk(3)*k
iP3 = ip + ia
return
end
```

```cpp
int iP3(double *w, int ia, ..., int k) {
static int kk[5];
int ifp = iws_FingerPrint(w,ia);
if( kk[4] != ifp ) { K3( w, ia, kk ); }
int ip  = kk[0]+kk[1]*i+kk[2]*j+kk[3]*k;
return ip + ia;
}
```

This is about as fast as we can get with pointer functions but much gain can be obtained by reducing the calls to these functions.

Here we show a fast loop construct, in C++, over the elements of a 3-dimensional table. We assume that the `K3` routine has been called before.

```cpp
int di = kk[1]; int dj = kk[2]; int dk = kk[3];   // address increments
int ia = iP3(w, iaT, i1, j1, k1);                      // start address
for( int i=i1; i<=i2; i++ ) { int ja = ia;
  for( int j=j1; j<=j2; j++ ) { int ka = ja;
    for( int k=k1; k<=k2; k++ ) { double Tijk = w[ka];  // table(i,j,k)
      ka += dk;
    }
    ja += dj;
  }
  ia += di;
}
```

The addresses are obtained from cheap running sums with only *one* call to `iP3`. Note that this scheme works for any nesting of the loops.

Note also that we do not need a nested loop if we traverse an entire n-dimensional table and are not interested in the index values. Here is a fast routine that initialises a table.

```
subroutine IniTab(w, ia, val)
double precision w(*), val
i1 = iws_BeginTbody(w,ia)
i2 = iws_EndTbody(w,ia)
do i = i1,i2
  w(i) = val
enddo
return
end
```

```
void IniTab(double *w, int ia, double val) {
int i1 = iws_BeginTbody(w,ia);
int i2 = iws_EndTbody(w,ia);
for(int i=i1; i<=i2; i++) {
  w[i] = val;
  }
}
```

For other fast loop constructs we mention that tables are stored column-wise with the first index running fastest, like a FORTRAN array (native C++ arrays are stored row-wise). Thus we always have `k(1)` $= 1$ so that one does not have to multiply, in a pointer function, the first index by its coefficient.

If you loop often over one index it is advantageous to make it the first index of a table. To illustrate this we compute—as a weighted sum—the convolution $f \otimes C$ of a pdf $f(x, \mu^2)$ and a coefficient function $C(x, n_{\rm f})$. The pdf is stored in a table `F(ix,iq)` at address `iaF` and the weights in `C(i,ix,nf)` at address `iaC`. The weighted sum runs over the first index from `1` to `ix` so that we can use the MBUTIL routine `dmb_VdotV(a,b,n)` to compute the convolution as the dot-product of two vectors.

```
jaF = iP2(w,iaF,1,iq)              !Address of F(1,iq)
jaC = iP3(w,iaC,1,ix,nf)           !Address of C(1,ix,nf)
FxC = dmb_VdotV(w(jaF),w(jaC),ix)  !Convolution at (ix,iq) for nf flavours
```

The pointer function `iP2` used here is a 2-dim version of `iP3` shown above.

8 Navigation

The table-sets form a linked list in the workspace and tables a linked list in a table-set. Thus if you want to continue beyond the end of a table list, you have to skip to the next table-set and from there to the next table. Four routines are provided to navigate a workspace by skipping forward or backward through the table-sets in the workspace, or through the tables in a table-set. The routines do not return an address but the distance (in words) to the target object. This distance is positive (negative) if the target address is after (before) the current address.

```
nw = iws_TFskip ( w, iaRST )        nw = iws_TBskip ( w, iaRST )
```

Get the (signed) distance to the next (`TFskip`) or previous (`TBskip`) table address. Can be called from a table, table-set or workspace-root address. The routine returns zero if there is no next or previous table, or if `iaRST` is not a valid input address.

```
nw = iws_SFskip ( w, iaRST )        nw = iws_SBskip ( w, iaRST )
```

As above, but now give the distance to the next or previous table-set address.

The links are in fact stored in the workspace headers as follows.

`w(ia+1)` Distance to the workspace root (unsigned).[2]
`w(ia+2)` Distance to the next table address (signed, `0` = no next table).
`w(ia+3)` Distance to the previous table address (signed, `0` = no previous table).
`w(ia+4)` Distance to the next table-set address (signed, `0` = no next table-set).
`w(ia+5)` Distance to the previous table-set address (signed, `0` = no previous table-set).

This gives fast navigation without the overhead of calling a routine, as is shown in the C++ example below where we navigate through an object-pointer.

```
double *obj = w + ia;                              //pointer to w[ia]
    ..
void sub(double *obj) {
  int ia = int(*(obj+1));  double *w    = obj-ia; //pointer to w
  int nt = int(*(obj+2));  double *ntab = obj+nt; //pointer to table after obj
  int pt = int(*(obj+3));  double *ptab = obj+pt; //pointer to table before obj
  int ns = int(*(obj+4));  double *nset = obj+ns; //pointer to tbset after obj
  int ps = int(*(obj+5));  double *pset = obj+ps; //pointer to tbset before obj
  ..
  }
```

9 Dynamic allocation

In C++ we can dynamically allocate a workspace, provided that we know beforehand how many words are needed. In Section 4 it is shown how to pre-compute a workspace size but it may be easier to create the workspace in a large (oversized) temporary buffer and then copy it to an array that fits the workspace, as is done below.

```
int nbuf    = 1000000, ntags = 10;
double *buf = new double[nbuf];               //large temporary buffer
int ias     = iws_WsInit(buf,nbuf,ntags," "); //convert buffer into a workspace
int iat     = iws_IaFirstTag(buf,ias);        //address of the first tag-word
buf[iat]    = double(iws_WTable(buf,..)-ias); //store local address table 1
buf[iat+1]  = double(iws_WTable(buf,..)-ias); //store local address table 2
  ..                                          //create more tables
int nw      = iws_WordsUsed(buf)+1;           //workspace size including trailer
double *w   = new double[nw];                 //new array of the right size
MBUTIL::smb_vcopy(buf,w,nw);                  //copy workspace to w
sws_SetWsN(w,nw);                             //store the size of w in the header
delete[] buf;                                 //get rid of the buffer
```

The code above can serve as the constructor of a C++ table-set class.

Sometimes you may want to make multiple workspace copies using `smb_vcopy` (do not forget to call `sws_SetWsN` if necessary). To avoid that these will all have the same time-stamp (fingerprint) you can call `sws_Stampit(w)` to stamp them individually.

[2] Note that an object address is `int(w(ia+1))+iaRoot` with `iaRoot` = `1` (`0`) in FORTRAN (C++).

A Integer store

An integer store (`istore`) should be declared `iw(n)` in FORTRAN or `int iw[n]` in C++. The layout of the store is much simpler than that of a workspace `w` since it holds only 1-dimensional arrays. An array in `iw` has a header and a body, but no tag field. Nor are the arrays organised into sets. An `istore` thus looks like this:

header	array	array	$\cdots$	0	free

In QCDNUM an `istore`—instead of a workspace—is used to store integer pointer-tables thus avoiding numerous double-to-integer conversions in loops.

```
call sws_IwInit ( iw, niw, 'comment' )     call sws_SetIwN ( iw, niw )
```

Convert an integer array into an `istore`.

`iw`	Integer array, dimensioned to `niw` in the calling routine.
`niw`	Size of `iw` as declared in the calling routine.
`'comment'`	Optional comment line to be printed when `iw` runs out of space.

If `iw` is stored in a dynamic C++ array then `sws_SetIwN` should be called after each increase in size to keep the size information in the `istore` header up-to-date.

```
ia = iws_Iarray ( iw, imin, imax )
```

Add an array to the store `iw`.

`imin`, `imax`	Index limits.
`ia`	Set, on exit, to the address of the new array object.

```
ia = iws_I|DAread ( iw, i|darr, n )
```

Put `n` elements of an integer (`IAread`) or double precision array (`DAread`) into the store `iw`. For this, a new `istore` array is created at address `ia`, with index range `1:n`.

```
call sws_IwWipe ( iw, ia )
```

Wipe `iw`, starting at array `ia`. Fatal error if `ia` is not the root or an array address.

The marker-word values of an `istore` and its arrays are the same as those for a workspace and table (see `sws_WsMark` in Section 4).

It is trivial to pre-compute the size of an `istore`: each array occupies $i_{\rm max} - i_{\rm min} + 1$ words plus $n_{\rm h}$ words for the header. The header size is returned by `iws_IhSize()`. The store itself also has a header (and trailer) so that we get for the total size

$$n_{\rm iw} = 1 + n_{\rm h} + \sum_{\rm arrays} (i_{\rm max} - i_{\rm min} + 1 + n_{\rm h}).$$

Navigation is also trivial (no routines provided) because the links to the workspace root and to the previous and next arrays are stored in the headers, as is done for a workspace:

`iw(ia)`	Marker word
`iw(ia+1)`	Distance to the workspace root (unsigned).[3]
`iw(ia+2)`	Distance to the next array address (signed, `0` = no next array).
`iw(ia+3)`	Distance to the previous array address (signed, `0` = no previous array).

Here are a few query routines to access information stored in `iw`.

Function	Description	
	Store	
`iws_IsaIstore(iw)`	Returns `1` (`0`) if `iw` is (not) an `istore`	
`iws_IwSize(iw)`	Total size of the store `iw`	
`iws_IwNused(iw)`	Number of words used (without trailer)	
`iws_IwNarrays(iw)`	Number (`n`) of arrays in `iw`	
`iws_IaLastObj(iw)`	Address of last object in `iw`	
`iws_IwNheader(iw)`	Header size	
	Array	
`iws_IwObjectType(iw,ia)`	Type of object at `ia`	(1)
`iws_IwFprint(iw,ia)`	Fingerprint	(2)
`iws_IwAsize(iw,ia)`	Array size (header + body)	
`iws_IwAnumber(iw,ia)`	Array serial number [`1`,`n`]	
`iws_IwAdim(iw,ia)`	Array dimension (always `1`)	
`iws_IwAimin(iw,ia)`	Array lower index limit	
`iws_IwAimax(iw,ia)`	Array upper index limit	
`iws_IaAbegin(iw,ia)`	Address of first word of the array-body	
`iws_IaAend(iw,ia)`	Address of last word of the array-body	
`iws_IwKnul(iw,ia)`	Pointer coefficient `k0`	
`iws_ArrayI(iw,ia,i)`	Address of element `i` of array `ia`	(3)

(1) Object types are: not-an-object (`0`), istore (`1`) and array (`2`).
(2) Setting `ia` = `iroot` yields the `istore` fingerprint, which is a unique time stamp.
(3) Pointer function with array boundary check `imin` $\leq$ `i` $\leq$ `imax`.

If you do not want the array boundary check of `iws_ArrayI` you should code inline

```
        iaddr(i) = ia + k0 + i
```

with `ia` the array address and `k0` taken from `iws_IwKnul`.

[3] Note that an array address is `iw(ia+1)+iaRoot` with `iaRoot` = `1` (`0`) in FORTRAN (C++).

B List of FORTRAN routines

For the query routines we refer to the lists on page 7 and 14.

Routine	Description
`iws_Version()`	Returns WSTORE version number
`iws_WsInit( w, nw, nt, 'comment' )`	Initialise workspace
`sws_SetWsN( w, nw )`	Enter workspace size limit
`iws_WTable( w, imin, imax, ndim )`	Add new table
`iws_NewSet( w )`	Add new table-set
`iws_WClone( w1, ia1, w2 )`	Clone table set or table
`sws_TbCopy( w1, ia1, w2, ia2, itag )`	Copy table content
`sws_WsWipe( w, ia )`	Wipe workspace
`sws_TsDump( 'fname', key, w, ia, &ierr )`	Dump table-set to disk
`iws_TsRead( 'fname', key, w, &ierr )`	Read table-set from disk
`sws_WsMark( &mws, &mset, &mtab )`	Get object markers
`iws_TbSize( imin, imax, ndim )`	Compute table size
`iws_HdSize()`	Return header size
`sws_WsTree( w )`	Print workspace tree
`sws_WsHead( w, ia )`	Print object header
`iws_Tpoint( w, ia, index, n )`	Address of a table element
`iws_TF|Bskip( w, ia )`	Forward/backward table skip
`iws_SF|Bskip( w, ia )`	Forward/backward table-set skip
`sws_Stampit( w )`	Set new time-stamp (fingerprint)
`sws_IwInit( iw, nw, nt, 'comment' )`	Initialise integer store
`sws_SetIwN( iw, nw )`	Enter `istore` size limit
`iws_Iarray( iw, imin, imax )`	Add new integer array
`iws_I|DAread( iw, i|darr, n )`	Read array into `istore`
`sws_IwWipe( iw, ia )`	Wipe `istore`
`iws_IhSize()`	Return header size
`sws_IwTree( iw )`	Print `istore` tree
`sws_IwHead( iw, ia )`	Print object header

Output arguments are pre-fixed with an ampersand (`&`).

C List of C++ prototypes

Addresses returned by the routines are integer array indices and not C++ pointers.

Workspace	
int	iws_Version()
int	iws_WsInit(double *w, int nw, int nt, string comment)
void	sws_SetWsN(double *w, int nw)
int	iws_WTable(double *w, int *imin, int *imax, int ndim)
int	iws_NewSet(double *w)
int	iws_WClone(double *w1, int ia, double *w2)
void	sws_TbCopy(double *w1, int ia1, double *w2, int ia2, int itag)
void	sws_WsWipe(double *w, int ia)
void	sws_TsDump(string fnam, int key, double *w, int ia, int &ierr)
int	iws_TsRead(string fnam, int key, double *w, int &ierr)
void	sws_WsMark(int &mws, int &mset, int &mtab)
int	iws_TbSize(int *imin, int *imax, int ndim)
int	iws_HdSize()
void	sws_WsTree(double *w)
void	sws_WsHead(double *w, int ia)
int	iws_Tpoint(double *w, int ia, int *index, int n)
int	iws_TF|Bskip(double *w, int ia)
int	iws_SF|Bskip(double *w, int ia)
void	sws_Stampit(double *w)
int	iws_IaRoot()
int	iws_IsaWorkspace(double *w)
int	iws_SizeOfW(double *w)
int	iws_WordsUsed(double *w)
int	iws_Nheader(double *w)
int	iws_Ntags(double *w)
int	iws_HeadSkip(double *w)
int	iws_IaDrain(double *w)
int	iws_IaNull(double *w)
int	iws_ObjectType(double *w, int ia)
int	iws_ObjectSize(double *w, int ia)
int	iws_Nobjects(double *w, int ia)
int	iws_ObjectNumber(double *w, int ia)
int	iws_FingerPrint(double *w, int ia)
int	iws_IaFirstTag(double *w, int ia)
int	iws_TableDim(double *w, int ia)
int	iws_IaKARRAY(double *w, int ia)
int	iws_IaIMIN(double *w, int ia)
int	iws_IaIMAX(double *w, int ia)
int	iws_BeginTbody(double *w, int ia)
int	iws_EndTbody(double *w, int ia)

Integer Store	
void	sws_IwInit(int *iw, int niw, string comment)
void	sws_SetIwN(int *iw, int niw)
int	iws_Iarray(int *iw, int imin, int imax)
int	iws_IAread(int *iw, int *inputarray, int n)
int	iws_DAread(int *iw, double *inputarray, int n)
void	sws_IwWipe(int *iw, int ia)
int	iws_IhSize()
void	sws_IwTree(int *iw)
void	sws_IwHead(int *iw, int ia)
int	iws_IsaIstore(int *iw)
int	iws_IwSize(int *iw)
int	iws_IwNused(int *iw)
int	iws_IwNarrays(int *iw)
int	iws_IaLastObj(int *iw)
int	iws_IwNheader(int *iw)
int	iws_IwObjectType(int *iw, int ia)
int	iws_IwFprint(int *iw, int ia)
int	iws_IwAsize(int *iw, int ia)
int	iws_IwAnumber(int *iw, int ia)
int	iws_IwAdim(int *iw, int ia)
int	iws_IwAimin(int *iw, int ia)
int	iws_IwAimax(int *iw, int ia)
int	iws_IaAbegin(int *iw, int ia)
int	iws_IaAend(int *iw, int ia)
int	iws_IwKnul(int *iw, int ia)
int	iws_ArrayI(int *iw, int ia, int i)