A++/P++ – Quick Reference Manual

(version 0.7.5)

Daniel Quinlan
Lawrence Livermore National Laboratory
L-560
Livermore, CA 94550
925-423-2668 (office)
925-422-6287 (fax)
dquinlan@llnl.gov
Quinlan's Web Page: http://www.llnl.gov/casc/people/dquinlan
A++/P++ Web Page http://www.llnl.gov/casc/Overture/A++P++
A++/P++ Manual (postscript version)
A++/P++ Quick Reference Manual (postscript version)
LACC Number: LA-CC-96-1
LAUR Number: LA-UR-95-3273

August 16, 2000

August 16, 2000

Chapter 1

Reference

1.1 Legend

type	**double**, **float**, or **int**

Variables used in examples below

i,j,k,l	integers used as scalar index variables
Span_I,Span_J,Span_K,Span_L	objects of type **Range**
I,J,K,L	objects of type **Index**
List_I,List_J,List_K,List_L	objects of type **intArray**
A,B,C	*type***Array** variables
Mask	an **intArray** variable
n,m,o,p	any positive integer
Fortran_Array_Pointer	pointer to a Fortran array
x	variable of *type*
axis	dimension 0-3 of the 4D *type***Array**

1.2 Debugging A++P++ Code

1.2.1 Turning On Bounds Checking

Bounds Checking in A++P++ must be turned on and is OFF by default.

Turning On Bounds Checking For All But Scalar Indexing

Bounds checking in A++P++ must be turned on and is OFF by default.

```
Index::setBoundsCheck (On);     Turns ON array bounds checking!
Index::setBoundsCheck (Off);    Turns OFF array bounds checking!
```

Turning On Bounds Checking For Scalar Indexing

Scalar bounds checking in A++P++ must be set at compile time. Bounds checking is OFF by default. It may be set on the compile command line or at the top of each program file (**before** #include<A++.h>).

CC-DBOUNDS_CHECK *other options*	Turns on scalar index bounds checking.
#define BOUNDS_CHECK	Turns on scalar index bounds checking in file.

1.2.2 Using dbx with A++

dbx supports calling functions and with the correct version of dbx that understands C++ name mangling, member functions of the A++ array objects may be called with the following example syntax:

call A.display()	dbx calls the display member function for an A++P++ array A

1.2.3 Mixing C++ streams and C printf

Mixing of C++ "cout <<" like I/O syntax with C stype "printf" I/O syntax will generate strange behavior in the ordering of the user's I/O messages. To fix this insert the following call to the I/O Streams library of C++ at the start of your main program.

ios::sync_with_stdio();	Synchronize C++ and C I/O subsystems!

1.3 Range Objects

1.3.1 Constructors

Note: The base must be less than or equal to the bound to define a valid span of an array, if base > bound then the range is considered null.

Range Span_K (±base,±bound),±stride);	Range object Span_K from base, to bound, by stride
Range Span_I;	Range object which is null
Range Span_J = Span_I;	Span_J is a copy of Span_I (not an alias)

1.3.2 Operators

Span_J = Span_I;	assignment operator
Span_I+n;	builds new Range object with position of Span_I + n
n+Span_I;	builds new Range object with position of Span_I + n
Span_I-n;	builds new Range object with position of Span_I - n
n-Span_I;	builds new Range object with position of Span_I - n

1.3.3 Access Functions

Span_I.getBase();	returns base of Span_I
Span_I.getBound();	returns bound of Span_I
Span_I.getStride();	returns bound of Span_I
Span_I.length();	returns (bound-base)+1 for Span_I

1.4 Index Objects

1.4.1 Constructors

The stride in the examples below default to 1 (unit stride) if not specified. That we provide an Index constructor which takes a Range object allows Range objects to be used where ever Index objects are used (e.g. indexing operators).

Index K (±position,count);	Index object K references from position, for count elements, with default stride = 1
Index K (±position,count,stride);	Index object K references from position, for count elements, with stride
Index I;	Index which references all of any array object
Index I(±i);	Index with position=±i, count=1, stride=1
Index J = I;	J is a copy of I (not an alias)
Index K = Span_I;	Index K is built from a Range object, Span_K

1.4.2 Operators

I+n;	new Index with position of Index I + n
n+I;	new Index with position of Index I + n
I-n;	new Index with position of Index I - n
n-I;	new Index with position of Index I - n
J = I;	assignment operator

1.4.3 Access Functions

I.getBase();	returns base of I
I.getBound();	returns bound of I
I.getStride();	returns stride of I
I.length();	returns length of I (accounting for stride)

1.4.4 Display Functions

I.display("label");	Prints Index values and all other internal data for I along with character string "label" to sdtout

1.5 Array Objects

1.5.1 Constructors

A++ arrays are replicated on each processor in P++, while P++ arrays are distributable across processors using user defined distributions (not covered here). Note that the Range objects can be used to build an A++ array, if used, they define the size *and* the base of the array from the Range object provided for each dimension.

*type***Array** A;	array object A (zero length array)
*type***Array** B = A;	array B as a copy of A
*type***Array** C (n);	1D array C of length n
*type***Array** C (n,m);	2D array C of length n × m

*type***Array** C (n,m,o); 3D array C of length n × m × o
*type***Array** C (n,m,o,p); 4D array C of length n × m × o × p
*type***Array** C (Span_I); 1D array C of length of Span_I
*type***Array** C (Span_I,Span_J); 2D array C of length of Span_I × Span_J
*type***Array** C (Span_I,Span_J,Span_K); 3D array C of length of Span_I × Span_J × Span_K
*type***Array** C (Span_I,Span_J,Span_K,Span_L); 4D array C of length of Span_I × Span_J × Span_K × Span_L

A++ only

*type***Array** C (Fortran_Array_Pointer, n); 1D array C of length n using existing array
*type***Array** C (Fortran_Array_Pointer, n,m); 2D array C of length n × m using existing array
*type***Array** C (Fortran_Array_Pointer, n,m,o); 3D array C of length n × m × o using existing array
*type***Array** C (Fortran_Array_Pointer, n,m,o,p); 4D array C of length n × m × o × p using existing array

*type***Array** C (Fortran_Array_Pointer, Span_I); 1D array C using existing data
*type***Array** C (Fortran_Array_Pointer, Span_I,Span_J); 2D array C using existing data
*type***Array** C (Fortran_Array_Pointer, Span_I,Span_J,Span_K); 3D array C using existing data
*type***Array** C (Fortran_Array_Pointer, Span_I,Span_J,Span_K,Span_L); 4D array C using existing data

P++ only

*type***Array** C (Fortran_Array_Pointer, n, Local_Size_n); 1D array C of length n using existing array
*type***Array** C (Fortran_Array_Pointer, m, Local_Size_m,
n, Local_Size_n); 2D array C of length n × m using existing array
*type***Array** C (Fortran_Array_Pointer, m, Local_Size_m,
n, Local_Size_n,
o, Local_Size_o); 3D array C of length n × m × o using existing array
*type***Array** C (Fortran_Array_Pointer, m, Local_Size_m,
n, Local_Size_n,
o, Local_Size_o,
p, Local_Size_p); 4D array C of length n × m × o × p using existing array

P++ only

*type***Array** C (n, Partition); Use existing Partitioning_Type
*type***Array** C (m, n, Partition); Use existing Partitioning_Type
*type***Array** C (m, n, o, Partition); Use existing Partitioning_Type
*type***Array** C (m, n, o, p, Partition); Use existing Partitioning_Type

1.5.2 Assignment Operators

A(I,J) = B(I-1,J+1); Set elements of A equal to elements of B
A = x; Set elements of A equal to x

1.5.3 Indexing Operators

Note that indexing support for Range objects is available because Index objects are constructed from the Range objects and the resulting Index object is used.

Indexing operators for scalar indexing: denotes a scalar

A(i) Scalar indexing of a 1D array object

A(i,j)	Scalar indexing of a 2D array object
A(i,j,k)	Scalar indexing of a 3D array object
A(i,j,k,l)	Scalar indexing of a 4D array object

Indexing operators for use with Index objects: denotes a *type***Array**

A(I)	Index object indexing of a 1D array object
A(I,J)	Index object indexing of a 2D array object
A(I,J,K)	Index object indexing of a 3D array object
A(I,J,K,L)	Index object indexing of a 4D array object

Indexing operators for use with Range objects: denotes a *type***Array**

A(Span_I)	Range object indexing of a 1D array object
A(Span_I,Span_J)	Range object indexing of a 2D array object
A(Span_I,Span_J,Span_K)	Range object indexing of a 3D array object
A(Span_I,Span_J,Span_K,Span_L)	Range object indexing of a 4D array object

Indexing operators for use with intArray objects: denotes a *type***Array**

A(List_I)	intArray object indexing of a 1D array object
A(List_I,List_J)	intArray object indexing of a 2D array object
A(List_I,List_J,List_K)	intArray object indexing of a 3D array object
A(List_I,List_J,List_K,List_L)	intArray object indexing of a 4D array object

1.5.4 Indirect Addressing

The subsection **Indexing Operators** (above) presents the use of intArrays to index A++ arrays (even other intArray objects). The value of the elements of the intArray are used to define the relevant elements of the indexed object (view). It is often required to convert between a mask returned by an relational operator and an intArray whose values represent the non-zero index positions in the mask, however this conversion of a mask to an intArray is currently supported only for 1D.

intArray Indirect_Address = Mask.indexMap()	builds intArray object with values of non-zero index position in Mask
intArray I = (A == 5).indexMap()	builds intArray I as a mapping (into A) of elements in A equal to 5

1.5.5 Arithmetic Operators

All arithmetic operators return a *type***Array** consistent with their input, no mixed type operations are allowed presently. Casting operators will be added soon to permit mixed operations. All operations are performed elementwise and the result returned in a separate *type***Array** (unless one of the operands is a result from a previous expression in which case the temporary operand is reused internally).

B + C;	Add B and C
B + x;	Add B and x
x + C;	Add x and C
B += C;	Add C to B store result in B
B += x;	Add x to B store result in B
B - C;	Subtract C from B
B - x;	Subtract x from B
x - C;	Subtract C from x

B -= C;	Subtract C from B store result in B
B -= x;	Subtract x from B store result in B
B * C;	Multiply B and C
B * x;	Multiply B and x
x * C;	Multiply x and C
B *= C;	Multiply C and B store result in B
B *= x;	Multiply x and B store result in B
B / C;	Divide B by C
B / x;	Divide B by x
x / C;	Divide x by C
B /= C;	Divide B by C store result in B
B /= x;	Divide B by x store result in B
B % C;	B Modulo C
B % x;	B Modulo x
x % C;	x Modulo C
B %= C;	B Modulo C to store result in B
B %= x;	B Modulo x store result in B

1.5.6 Relational Operators

All relational operators return an **intArray**, no mixed type operations are allowed presently. All operations are performed elementwise and return conformable mask (**intArray** object). Mask values are zero if the conditional test was false, and non-zero if operation was true. See **Indirect Addressing** for conversion of zero/non-zero masks into intArrays for use with indirect address indexing.

!B;	mask based on test for zero elements of B
B < C;	mask specifying elements of B < C
B < x;	mask specifying elements of B < x
x < C;	mask specifying elements of C where x < C
B <= C;	mask specifying elements of B <= C
B <= x;	mask specifying elements of B <= x
x <= C;	mask specifying elements of C where x <= C
B > C;	mask specifying elements of B > C
B > x;	mask specifying elements of B > x
x > C;	mask specifying elements of C where x > C
B >= C;	mask specifying elements of B >= C
B >= x;	mask specifying elements of B >= x
x >= C;	mask specifying elements of C where x >= C
B == C;	mask specifying elements of B == C
B == x;	mask specifying elements of B == x
x == C;	mask specifying elements of C where x == C
B != C;	mask specifying elements of B != C
B != x;	mask specifying elements of B != x
x != C;	mask specifying elements of C where x != C
B && C;	mask specifying elements of B && C
B && x;	mask specifying elements of B && x

x && C;	mask specifying elements of C where x && C
B || C;	mask specifying elements of B || C
B || x;	mask specifying elements of B || x
x || C;	mask specifying elements of C where x || C

1.5.7 Min Max functions

These functions (except in the case of the single input reduction operations) return array objects with an elementwise interpretation. Both "min" and "max" represent reduction operations in the case of a single array input. These functions thus return a scalar value from the array input. In A++ the operation is straightforward. In P++ the reduction operators return a scalar, but internally do the required message passing to force the same scalar return value on all processors (assuming a data parallel model of execution).

min (A);	return scalar minimum of all array elements
min (B,C);	min elements of B and C
min (B,x);	min elements of B and x
min (x,C);	min elements of x and C
min (A,B,C);	min elements of A,B and C
min (x,B,C);	min elements of x,B and C
min (A,x,C);	min elements of A,x and C
min (A,B,x);	min elements of A,B and x
max (A);	return scalar maximum of all array elements
max (B,C);	max elements of B and C
max (B,x);	max elements of B and x
max (x,C);	max elements of x and C
max (A,B,C);	max elements of A,B and C
max (x,B,C);	max elements of x,B and C
max (A,x,C);	max elements of A,x and C
max (A,B,x);	max elements of A,B and x

1.5.8 Miscellaneous Functions

All functions return a *type***Array** consistent with their input, no mixed type operations are allowed presently. Functions fmod and mod apply to double or float arrays and integer arrays, respectively. Functions log, log10, exp, sqrt, fabs, ceil, floor, cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh; only apply to **doubleArray** and **floatArray** objects. Function abs applies to only **intArray** objects.
For P++ operation of reduction functions ("sum," for example) see note on reduction operators in P++ in previous subsection (Min Max functions).

fmod (B,C);	B modulo C equivalent to operator B % C
fmod (B,x);	B modulo x equivalent to operator B % x
fmod (x,C);	x modulo C equivalent to operator x % C
mod (B,C);	B modulo C equivalent to operator B % C
mod (B,x);	B modulo C equivalent to operator B % x
mod (x,C);	B modulo C equivalent to operator x % C
pow (B,C);	$B(i)^{C(i)}$ for elements of B and C
pow (B,x);	$B(i)^{x}$ for elements of B and x

pow (x,C);	$x^{C(i)}$ for elements of x and C
sign (B,C);	C with sign of B
sign (B,x);	array with values of x but with sign of B
sign (x,C);	C with sign of x
sum (B);	sum of elements of B
log (B);	log of elements of B
log10 (B);	log10 of elements of B
exp (B);	exp of elements of B
sqrt (B);	sqrt of elements of B
fabs (B);	fabs of elements of B
ceil (B);	ceil of elements of B
floor (B);	floor of elements of B
abs (B);	abs of elements of B
cos (B);	cosine of elements of B
sin (B);	sine of elements of B
tan (B);	tangent of elements of B
acos (B);	arccosine of elements of B
asin (B);	arcsine of elements of B
atan (B);	arctangent of elements of B
atan2 (B,C);	arctangent of elements of B/C
cosh (B);	hyperbolic cosine of elements of B
sinh (B);	hyperbolic sine of elements of B
tanh (B);	hyperbolic tangent of elements of B
acosh (B);	arc hyperbolic cosine of elements of B
asinh (B);	arc hyperbolic sine of elements of B
atanh (B);	arc hyperbolic tangent of elements of B

1.5.9 Replace functions

Replacement of elements is done for non-zero mask elements. Mask and input arrays must be conformable. Since this feature of A++/P++ is redundant with the **where** statement functionality, the replace member function may be devalued at a later date and then removed from A++/P++ sometime after that.

A.replace (Mask , B);	replace elements in A with elements in B depending on value of Mask
A.replace (Mask , x);	replace elements in A with scalar x depending on value of Mask
A.replace (x , B);	replace elements in A with elements in B depending on value of x (equivalent to `if (x) A = B;`)

1.5.10 Array Type Conversion Functions

The conversion between array types is commonly represented by casting operators. However, such casting operators could be called as part of automate conversion which can be especially problematic to debug. To facilitate the conversion between types of arrays we provide member functions that cast an array of one type to an array of another type explicitly. These member functions can, for example, convert an array of type intArray to an array of type floatArray. Or we can convert a floatArray to an intArray. As and example, this mechanism simplifies the visualization of intArray objects using graphics functionality only written for floatArray or doubleArray types. Future work implement casting operators that make the conversion implicit.

A.convertTo_intArray();	return an intArray (convert *type***Array** A to an intArray

A.convertTo_floatArray();	return a floatArray (convert *type***Array** A to a floatArray
A.convertTo_doubleArray();	return a doubleArray (convert *type***Array** A to a doubleArray

1.5.11 User defined Bases

A++/P++ array object may have user defined bases in each array dimension. This allows for array objects to have a base of 1 (as in FORTRAN), or any other positive or negative value.

A.setBase(±n);	Set base to ±n along all axes of A
A.setBase(±n,axis);	Set base to ±n along axis of A
setGlobalBase(±n);	Set base to ±n along all axes for all future array objects
setGlobalBase(±n,axis);	Set base to ±n along axis for all future array objects

1.5.12 Indexing of Views

The base and bound of a view of an array object are dependent on the base and bound of the **Index** or **Range** object used to build the view. Thus a view, A(I), of an array, A, is another array object which carries with it the index space information about it's view of the subset of data in the original array, A.

1.5.13 Array Size functions

Array axis numbering starts at zero and ends with the max number of dimensions (a constant MAX_ARRAY_DIMENSION stores this value) for the A++/P++ array objects minus one. These provide access into the A++ objects and assume an A++ object is being used. An alternative method is defined to permit access to the same data if a raw pointer is being used, this later method is required if a pointer to the array data is being passed to FORTRAN. The access functions for this data have the names *getRawBase()*, *getRawBound()*, *getRawStride()*, *getRawDataSize()*.

A.getBase();	Get base along all axes of A (bases must be equal)
A.getBase(axis);	Get base along axis of A
A.getRawBase(axis);	Get base along axis of A
getGlobalBase();	Get base along all axes for all future array objects
getGlobalBase(axis);	Get base along axis for all future array objects
A.getStride(axis);	Get stride along axis of A
A.getRawStride(axis);	Get stride along axis of A
A.getBound(axis);	Get bound along axis of A
A.getRawBound(axis);	Get bound along axis of A
A.getLength(axis);	Get dimension (array size) of A along axis
A.getFullRange(axis);	return a Range object (base,bound,stride of the array)
A.dimension(axis);	Get dimension (array size) of A along axis (returns a Range object)
A.elementCount();	Get total array size of A
A.numberOfDimensions();	Get total number of dimensions of A
A.isAView();	returns TRUE if A is a subArray (view) of another array object
A.isNullArray();	returns TRUE if A is an array of size zero
A.isTemporary();	returns TRUE if A is a result of an expression
A.rows();	Get number of rows of A (for 2D array objects)
A.cols();	Get number of cols of A (for 2D array objects)

1.5.14 Array Object Similarity test functions

Array axis numbering starts at zero and ends with the max number of dimensions (a constant MAX_ARRAY_DIMENSION stores this value) for the A++/P++ array objects minus one. These member functions allow for the testing of Bases, Bounds, Strides, etc along each axis for two array objects. For example, the return value is TRUE if the Bases match along all axes, and FALSE if they differ along any axis.
A conformability test is included to allow the user to optionally test the conformability of two array objects before the array operation.

A.isSameBase(B);	Check bases of both arrays along all axes (all bases equal return TRUE)
A.isSameBound(B);	Check bounds of both arrays along all axes (all bounds equal return TRUE)
A.isSameStride(B);	Check strides of both arrays along all axes (all strides equal return TRUE)
A.isSimilar(B);	Check bases, bounds, and strides of both arrays along all axes
A.isConformable(B);	Checks conformability of both arrays

1.5.15 Array Object Internal Consistancy Test

This function tests the internal values for consistancy it is mostly included for completeness. It is most usefull within P++ where there is significant testing that can be done between local and global data to verify consistant internal behavior. It is used within A++ and P++ when internal debugging is turned on (not the default in distribution versions of A++ and P++.

A.isConsistant();	Checks internal consistancy of array object

1.5.16 Shape functions

These shape functions redimension an existing array object. The reshape function allows the conversion of an nxm array to an mxn array (2D example), the total number of elements in the array must remain the same and the data values are preserved. The redim function redimensions an array to a different total size (larger of smaller), but does not preserve the data (data is left uninitialized). The resize function is similar to the redim function except that it preserves the data (truncating the data if the new dimensions are smaller and leaving new values uninitialized if the new dimensions are larger. Each function can be used with either scalar or Range object input parameters, additionally each function may be provided an example array object from which the equivalent Range objects are extracted (internally). All these member functions preserve (save and reset) the original base of the array object.

A.reshape(i,j,k,l);	Change dimensions of array using the same array data (same size)
A.reshape(Span_I,Span_J,Span_K,Span_L);	Change dimensions of array using the same array data (same size)
A.reshape(*type***Array**);	Change size of array object using another array object
A.resize(i,j,k,l);	Change size of array object (old data is copied and truncated)
A.resize(Span_I,Span_J,Span_K,Span_L);	Change size of array object using Range objects
A.resize(*type***Array**);	Change size of array object using another array object
A.redim(i,j,k,l);	Change size of array object (old data is lost)
A.redim(Span_I,Span_J,Span_K,Span_L);	Change size of array object using Range objects
A.redim(*type***Array**);	Change size of array object using another array object
transpose (A);	transpose of elements of A

1.5.17 Display Functions

A(I,J).display("label"); Prints array data for the view A(I,J) along with character string "label" to sdtout
A.view("label"); Prints array data and all other internal data for A along with character string "label" to sdtout

Details of the display of the values within an array by the display function are controled by the values assigned to the *type***Array**::DISPLAY_FORMAT variable. This variable has a default value of *type***Array**::SMART_DISPLAY_FORMAT which allows for the auto selection of either DECIMAL or EXPONENTIAL format depending upon the values within the array. Display Format Control Values:

*type***Array**::DISPLAY_FORMAT = *type***Array**::DECIMAL_DISPLAY_FORMAT; Uses xxx.yyyy format
*type***Array**::DISPLAY_FORMAT = *type***Array**::EXPONENTIAL_DISPLAY_FORMAT; Uses x.yyyye±zz format
*type***Array**::DISPLAY_FORMAT = *type***Array**::SMART_DISPLAY_FORMAT; Auto-selects either of above formats

1.5.18 Array Expressions Used For Function Input

Functions passing array objects by reference can't be passed an expression since expressions return temporaries that are managed differently internally. Functions passing expressions by value require no special handling.

foo (evaluate (A+B)); Force (A+B) temporary to be persistent for function *foo* , which passes an array object by reference

1.5.19 Array Aliasing

A++ and P++ arrays can be aliased however all caveats apply as in the use of FORTRAN equivalence. This permits array object to be views of other array objects or indexed parts of other array objects. Note that P++'s adopt function must build the distributed array from the collection of pointers to local memory in each processor and so requires both global and local domain size information (P++ organizes any communication that is required to build the distributed array (currently there is no communication required)).

B.reference (A(I,J)); Force B to reference A(I,J)
B.breakReference (); Break reference to A(I,J) (builds a copy of previous reference)

A++ only

C.adopt (Fortran_Array_Pointer, n); 1D array C of length n using existing array
C.adopt (Fortran_Array_Pointer, n,m); 2D array C of length n × m using existing array
C.adopt (Fortran_Array_Pointer, n,m,o); 3D array C of length n × m × o using existing array
C.adopt (Fortran_Array_Pointer, n,m,o,p); 4D array C of length n × m × o × p using existing array

P++ only

C.adopt (Fortran_Array_Pointer, n, Local_Size_n); 1D array C of length n using existing array
C.adopt (Fortran_Array_Pointer, m, Local_Size_m,
n, Local_Size_n); 2D array C of length n × m using existing array
C.adopt (Fortran_Array_Pointer, m, Local_Size_m,
n, Local_Size_n,
o, Local_Size_o); 3D array C of length n × m × o using existing array
C.adopt C (Fortran_Array_Pointer, m, Local_Size_m,
n, Local_Size_n,
o, Local_Size_o,
p, Local_Size_p); 4D array C of length n × m × o × p using existing array

1.5.20 Fill Function

More fill functions will be added to later releases of A++/P++. Its purpose is to initialize an array object to value or set of values.

B(I,J).fill(x);	Set elements of B(I,J) equal to x
B(I,J).seqAdd(Base,Stride);	Set elements of B(I,J) equal to Base, Base+Stride, ... , Base+n*Stride default value for Base and Stride are 0 and 1

1.5.21 Access To FORTRAN Ordered Array

A++/P++ provides access to the internal data of the array object using the following access functions. Arrays are stored internally in FORTRAN order and a pointer to the start of the array can be obtained using the getDataPointer member function. In the case of a view the pointer is to the start of the view. It is up to the user to correctly manipulate the data (good luck). Similar access is provide to the array descriptor (though info for it's use is not contained in this Quick_Reference_Manual).

Fortran_Array_Pointer = A.getDataPointer(); Array_Descriptor_Type = A.getDescriptorPointer();

1.6 "where" Statement

Example of **where** statement support in A++/P++. Note that elsewhere statements may be cascaded and that an optional parameter (Mask) can be specified. Note that **elsewhere** must have a set of parenthesis even if no parameter is specified. The mask must be conformable with the array operations in the code block. *On the Cray, and with the GNU g++ compiler, the statement* **elsewhere(mask)** *taking a mask as a parameter is called* **elsewhere_mask(mask)**. *This is due to a problem with parameter checking of macros. The syntax for* **elsewhere()**, *not taking a mask, does not change.* This aspect of A++ syntax may be changed slightly to accommodate these non-portable aspects of the C preprocessor.

where (A == 0)	
{	
B = 0;	elements of B set to zero at positions where A = 0
A = B + C;	B added to C and assigned to A at positions where A = 0
}	
elsewhere (B > 0)	Use **elsewhere_mask** on the Cray and with GNU g++
{	
B = A;	elements of B set to A at positions where A $\neq$ 0 and B > 0
}	
elsewhere ()	
{	
B = A;	elements of B set to A at positions where A $\neq$ 0 and B $\leq$ 0
}	

1.7 P++ Specific Information

There are access functions to the lower level objects in P++ which can be manipulated by the user's program. Specifically we provide access to the **Partitioning_Type** that each array uses internally (if it is not using the default distribution). The purpose of providing manual ghost boundary updates is to permit override of the message passing interpretation provide in P++. The resulting reduced overhead provides a simple means to optimize performance of operations the user recognizes as not requiring more than an update of the

internal ghost boundaries. The "displayPartitioning" member function prints out ASCII text which describes the distribution of the P++ array on the multiprocessor system. The same functions exist in A++ but don't do anything, this supports backward compatibility between P++ and A++.

1.7.1 Control Over Array Partitioning (Distributions)

The distribution of P++ array objects is controled though partitioning objects that are associated with the array objects. The association of a partitioning object with and array is done either at construction of the array objects or later in the probram. An unlimited number of array objects may be associated with a given partitioning object. The manipulation of the partitioning object translates directly to manipulation of each of the array objects associated with the partitioning. This feature makes it easier to manipulate large number of arrays with a simple interface. Partitioning objects are valid object in A++, but have no meaningful effect, so they are only functional in P++. This is to permit bidirectional portability between A++ and P++ (the serial and parallel environments). An unlimited number of **Partitioning_Type** objects may be used within an application. One of the main purposes of the partitioning objects is to define the distribution of P++ arrays and permit the dynamic redistribution. The expected usage is to have many P++ arrays associated with a relatively small number of **Partitioning_Type** objects.

Constructors

At present the constructor taking a intArray as a parameter is not implemented, it's purpose is to provide a simple means to control load balancing; it is the interface for a load balancer. But load balancing is not a part of A++/P++, load balancers used with parallel P++ applications are presently separate from P++. The most common usage of the partitioning object is to either call the constructor which specifies a subrange of the virtual processor space (this will be truncated to the exitisting virtual processor space if too large a range is specified), or call the default constructor (the whole virtual processors space) and then call member functions to modify the partitioning object.

Partitioning_Type P ();	Default constructor
Partitioning_Type P (Load_Map);	Load_Map is a intArray specifying the work distribution
Partitioning_Type P (Number_Of_Processors);	integer input specifies number of processors to use (start=0)
Partitioning_Type P (Span_P);	**Range** input specifies range of processors to use
Partitioning_Type P1 = P;	Deep copy constructor

Member functions

The operations on a **Partitioning_Type** object are done to all P++ arrays that are associated through that **Partitioning_Type** object. This provides a powerful mechanism for the dynamic control of array distributions; load balancers are expected to take advantage of this feature. The "applyPartition" member function is provided so that multiple modifications to the partitioning object may be done and a single restructuring of the P++ arrays associated with the partitioning object completed subsequently. P++ operation is undefined if the partitioning is never applied to it's associated objects. At present, only the partitionAlongAxis member function does not call the applyPartition function automatically. This detail of the interface may change in the near future to allow a more simple usage. The partitionAlongAxis member function takes three parameters: int Axis, bool Partitioned, int GhostBoundaryWidth. This simplifies the setting and modification of the partitioning. Afterward this only takes effect once the applyPartition member function is called. Then all distributed arrays associated with the partitioning object are redistributed with the ghost boundaries that were specified.

SpecifyDecompositionAxes (Input_Number_Of_Dimensions_To_Partition);	Integer input
SpecifyInternalGhostBoundaryWidths (int,int,int,int);	Default input is zero
display (Label);	printout partition data
displayDefaultValues (Label);	printout default partition data
displayPartitioning (Label);	graphics display of partition data
displayDefaultPartitioning (Label);	graphics display of default partition data
updateGhostBoundaries (X);	X is a P++ array
partitionAlongAxis (int Axis, bool PartitionAxis, int GhostBoundaryWidth);	input specifies axis
applyPartition ();	force partitioning of previously associated P+

1.7.2 Array Object Member Functions

Array objects have some specific member functions that are meaningful only within P++, as A++ array objects the member functions are defined, but have do nothing. This is done for backward compatability.

Partitioning_Type *X = A.getPartition();	get the internal partition
A.partition(**Partition**);	repartition dynamically
A.partition(*type***Array**);	repartition same as existing array object
A.getLocalBase(axis);	return base of local processor data
A.getLocalBound(axis);	return bound of local processor data
A.getLocalStride(axis);	return stride of local processor data
A.getLocalLength(axis);	return length of local processor data
A.getLocalFullRange(axis);	return a Range object (base,bound,stride of the local array)
A.getSerialArrayPointer();	return a pointer to the local array (and A++ array)
A.getLocalArray();	return a shallow copy of the local array (and A++ array)
A.getLocalArrayWithGhostBoundaries();	return a shallow copy (with ghost boundaries)
A.updateGhostBoundaries();	updates all ghost bondaries
A.displayPartitioning();	prints info on distribution of array data
A.getGhostCellWidth(Axis);	access to ghost boundary width
A.getInternalGhostCellWidth(Axis);	access to ghost boundary width (devalued, will be removed in future r
A.setInternalGhostCellWidth(int,int,int,int);	dynamicly adjusts ghost boundary width
A.setInternalGhostCellWidthSaveData(int,int,int,int);	as above but preserves the data and updates ghost boundaries

1.7.3 Distributed vs Replicated Array Data

Within P++ arrays are distributed, distributions have the following properties:

- 1 An array is distributed in some or all of the dimensions of the array (the user selects such details).
- 2 An array is distributed over a subset of processors.
- 3 An array is distributed over only a single processor (a trivial case of #2 above).
- 4 An array is built onto only one processor and only that processor knows about it (i.e. an A++ array object is built locally on a processor).
- 5 An array is replicated onto all processors (this is really a trival case of #4 above where each array is built locally on each processor). In this case the user is responciple for maintaining a consistant representation of the data which is replicated. This later case is useful for when a small array is required and is analogous to the case of replication of scalars onto every processor since no overhead of parallel support.

P++ also contains **SerialArrays**, (e.g. **doubleSerialArray**). These arrays are simply A++ array objects on each processor. In a data parallel way, if all processors build a serial array object, then each processor builds an array and the array is replicated across all processors. It is up to the user to maintain the consistancy of the array data across all processors in this case. Many arrays that are small are simply replicated, this costs little in additional space and avoid any communication when data is accessed.

1.7.4 Virtual Processors

P++ uses a number of processors independent upon the number of actual processors in hardware. On machines that support it the excess processors are evenly distributed among the hardware processors. This allows for greater control of granularity in the distribution of work. Where it is important to take advantage of this is application dependent. For most of the development this has allowed us to test problems on a number of processors indepentend of the actual number of machines that we have in our workstation cluster.

1.7.5 Synchronization Primative

Note that the Communication_Manager::Sync() is helpful in verifying the all processors reach a specific point in the parallel execution. This is helpful most often for debugging parallel codes.

Communication_Manager::Sync();	Call barrier function to sync all processors

1.7.6 Access to specific Parallel Environment Information

Although access to the underlying parallel information such as processor number, etc. can be used to break the data parallel model of execution such information is made available within P++ because it can be useful if used correctly. As an example of correct useage moving an application using graphics from A++ to P++ often is simplified if a specific processor is used for all the graphics work while others are idle. Access to the process number allows the code on each processor to branch dependent upon the processor number and thus simplifies (at initially) the movement of large scale A++ applications onto parallel machines using P++. Some of the data is only valid for either PVM or MPI, and some data is interpreted different by the two communication libraries.

Communication_Manager::numberOfProcessors();	get number of virtual processors
Communication_Manager::localProcessNumber();	get processor id number
Communication_Manager::Sync();	barrier primative
Communication_Manager::My_Task_ID;	get process id
Communication_Manager::MainProcessorGroupName;	Name of MPI Group

1.7.7 Escaping from the Data Parallel Execution Model

Since the data parallel style is only assumed for the execution of P++ array operations, but not enforced, it is possible to break out of the Data Parallel model and execute any parallel code desired. Users however are expected to handle their own communication. Since some degree of syncronization is helpful in moving into and out of the data parallel modes, the Communication_Manager::Sync() function is expected to be used (though not required).

1.7.8 Access to the local array

Each P++ distributed array on each processor contains a local array (a SerialArray object (same as an A++ array object)). The local array is availabel with and without ghost boundaries.

Access to the local array without ghost boundaries

The local array stores the local part of the distributed array data. Access to the localArray is obtained from:

A.getLocalArray(); return a shallow copy of the local array (an A++ array)

Access to the local array with ghost boundaries

Ghost boundaries are not visible within the local array since the local array is a view of the partition of the distributed space on the current processor. The ghost boundaries (if the ghost boundary width is nonzero) **are** present, but access to them from the view would result in an out of range error. Another mechanism for accessing the local array is required to get the local array containing the ghost boundaries.

A.getLocalArrayWithGhostBoundaries(); return a shallow copy (with include ghost boundaries)

The access to the ghost boundaries is possible from this view, but the user must know how to interpret the ghost boundaries within the returned local array object. (Hint: they are at the boundaries and the widths along each access are given by the ghost boundary widths obtained from the partitions.)

1.7.9 Examples of P++ specific operations

We provide some simple examples within the A++/P++ manual, please consult that chapter on Examples to see illustrations of the useage of the P++ specific functions.

1.8 Optimization Manager

Optimization manager is an object whose member functions control properties of the execution of the A++ and P++ array class (see reference manual). More member functions later will allow for improved optimization potential. The setup of the "Virtual Machine" may be separated outside of the P++ interface since not all machine environments require it (both MPI and PVM do, so it is present in P++ currently).
The "Program_Name" should be initialized with the complete name of the executable (including path), however in environments where it is supported P++ will automatically search for the string if only "" is specified. This is a feature that can not be supported on all architectures (or PVM would handle it internally).

```
Initialize_Virtual_Machine ( char* Program_Name = "" , int Num_Processors = 1, int argc, char** argv );   First P++ statement
Exit_Virtual_Machine ();                                                                                   Last P++ statement
setOptimizedScalarIndexing ( On_Off_Type On_Off = On );                                                    Optimize performance
```

1.9 Diagnostic Manager

There are times when you want to know details about what is happening internally within A++/P++. We provide a limited number of ways of seeing what is going on internally and getting some data to understand the behavior of the users application. More will be added in future versions of A++/P++.

1.9.1 Report Generation

There are a number of Diagnostic manager function which generate reports of the internal useage. Some reports are quite long, other are brief and summarize the execution history for the whole application.

getSizeOfClasses();	Reports the sizes of all internal classes in A++/P++
getMemoryOverhead();	returns memory overhead for all arrays
getTotalArrayMemoryInUse();	returns memory use for array elements
getTotalMemoryInUse();	reports total memory use for A++/P++
getnumberOfArraysConstantDimensionInUse(dimension, arrayType);	reports number of arrays of type and dimension
getMessagePassingInterpretationReport();	Communication Report
getReferenceCountingReport();	Reference Counting Report
displayCommunication (const char* Label = "");	communication report by processor
displayPurify (const char* Label = "");	Displays memory leaks by processor (uses purify)
report();	Generates general report of A++/P++ behavior
setTrackArrayData(Boolean trueFalse = TRUE);	Track and report on A++/P++ diagnostics
getTrackArrayData();	get Boolean value for diagnostic mechanism
buildCommunicationMap ();	Builds map of communications by processor
buildPurifyMap ();	Builds map of purify errors by processor
getPurifyUnsupressedMemoryLeaks();	Total Memory leaked

Features and counted quantities include:

- The use of **int Diagnostic_Manager::getSizeOfClasses()** displays a text report of the sizes of different internal structures in A++P++.
- The use of **int Diagnostic_Manager::getMemoryOverhead()** returns an integer that represents the number fo byte of overhead used to store intenal array descriptors, partitioning information (P++ only), etc.; for the whole application at the time that the function is called.
- The use of **int Diagnostic_Manager::getTotalArrayMemoryInUse()** returns an integer representing the total number of array elements in use in all array objects at the time that function is called.
- The use of **int Diagnostic_Manager::getTotalMemoryInUse()**

returns the total number of bytes in use within A++/P++ for all overhead and array elements at the time the function is called.

- The use of **int Diagnostic_Manager::getnumberOfArraysConstantDimensionInUse()** returns the number of arrays of a particular dimension and of a particular type. this function is an example of the sort of diagnostic questions that can be written which interogate the runtime system to find out both global and local properties of its operation.

- The use of **int Diagnostic_Manager::getMessagePassingInterpretationReport()** generates a report (organized from each processor, but reported on processor 0). The report details the number of MPI sends, MPI receives, the number of ghost boundary updates (one update implies the update of all ghost boundaries on an array, even if this generates fewer MPI messages than ghost boundaries), and the number VSG updates *regular section transfers* (the more general communication model which permits operations between array objects independent of the distribution across multiple processors).

- The use of **int Diagnostic_Manager::getReferenceCountingReport()** generates a report of the internal reference counts used in the execution of array expressions. This function is mostly for internal debugging of reference counting problems.

- The use of **int Diagnostic_Manager::report()** generates a summary report of the execution of the A++/P++ application at the point when it is called.

- The use of **int Diagnostic_Manager::setTrackArrayData()** turns on the internal tracking of array objects as part of the internal diagnostics and permits the summary report to report more detail. It is off by default so that there is no performance penalty associated with internal diagnostics. This must be set at the top of an application before the first array object is built.

1.9.2 Counting Functions

Optional mechanisms in A++/P++ permit many details to be counted internally as part of the report generation mechanisms. All functions return an integer.

resetCommunicationCounters ();	reset the internal message passing counting mechanism
getNumberOfArraysInUse();	returns the number of arrays inuse
getMaxNumberOfArrays();	returns the max arrays in used at any point in time

getNumberOfMessagesSent();	returns the number of messages sent
getNumberOfMessagesReceived();	returns the number of messages received
getNumberOfGhostBoundaryUpdates();	returns number of updates to ghostboundaries
getNumberOfRegularSectionTransfers();	# of uses of general communication mechanism
getNumberOfScalarIndexingOperations();	scalar indexing
getNumberOfScalarIndexingOperationsRequiringGlobalBroadcast();	Global broadcast with communication

Features and counted quantities include:

- The use of **int Diagnostic_Manager::resetCommunicationCounters()** permits the internal counters to be reset to ZERO.
- Number of arrays in use **int Diagnostic_Manager::getNumberOfArraysInUse()** The number of arrays in use at any point in the execution is useful for gauging the relative use od A++/P++ and spotting potential memory leaks.
- Max arrays in use **int Diagnostic_Manager::getMaxNumberOfArrays()** This function tallies the most number of arrays in use at any one time durring the execution history (note: records use in increments of 300).
- Reset message counting **int Diagnostic_Manager::resetCommunicationCounters()** Resest the message counters to ZERO to permit localized counting of messages generated from code fragements.
- Number of messages (sent) **int Diagnostic_Manager::getNumberOfMessagesSent()** returns the total messages since the beginning of execution or from the last call to **Diagnostic_Manager::resetCommunicationCounters()**.
- Number of messages (received) **int Diagnostic_Manager::getNumberOfMessagesReceived()** returns the total messages since the beginning of execution or from the last call to **Diagnostic_Manager::resetCommunicationCounters()**.
- Number of messages (received) **int Diagnostic_Manager::getNumberOfGhostBoundaryUpdates()** Returns the total number of calls to update the ghost boundaries of arrays. Note that some calls will not translate into message passing (e.g. if only run on one processor or if the ghost boundary width is ZERO). Reports on number of messages since the beginning of execution or from the last call to **Diagnostic_Manager::resetCommunicationCounters()**.

1.9.3 Debugging Mechanisms

These functions provide mechanisms to simplify the error checking and debugging of A++/P++ applications.

getPurifyUnsupressedMemoryLeaks();	Total Memory leaked
setSmartReleaseOfInternalMemory(On/Off);	Smart Memory cleanup
getSmartReleaseOfInternalMemory();	get Boolean value for smart memo
setExitFromGlobalMemoryRelease(Boolean);	setup exit mechanism
getExitFromGlobalMemoryRelease();	get Boolean value for exit mechani
test (*type***Array**);	Destructive test of array object
displayPurify (const char* Label = "");	Displays memory leaks by processo
buildPurifyMap ();	Builds map of purify errors by pro

- The use of **void Diagnostic_Manager::setSmartReleaseOfInternalMemory()** (called from anywhere in an A++/P++ application) will trigger the mechanism to cleanup all internally used memory within A++/P++ after the last array object has been deleted. Specifically it counts the number of arrays in use (and the number of arrays used internally (e.g. where statement history, etc.) and when the two values are equal it calls the **void globalMemoryRelease()** function which then deletes existing arrays in use and other data used internally (reference count arrays, etc.). The user is warned in the output of the **void globalMemoryRelease()** function to not call any functions that would use A++/P++ since the results would be *undefined*.

- The use of the **void Diagnostic_Manager::setExitFromGlobalMemoryRelease()** will force the application to exit after the global memory release (and from within the **void globalMemoryRelease()** function itself. The user may then specify that the normal exit from the base of the **main** function is an error and thus detect the proper cleanup of memory in test programs using the exit status (stored in the **$status** *enviroment variable* on all POSIX operating systems (most flavors of UNIX). If purify is in use (both A++/P++ configured to use **purify** and running with purify) then **purify_exit(int)** is called. This function or's the memory leaks, memory in use, and purify errors into the exist status so that the **$status** *enviroment variable* can be used to detect purify details within test codes. A++/P++ test codes are tested this way when A++/P++ is configured to use PURIFY. P++ applications can not always communicate detected purify problems on other processes AND output the correct exit status, this is only a limitation of how **mpirun** returns it's exit status.

- The use of **void Diagnostic_Manager::test(*type*Array A)** allows for exhaustive (destructive) tests of an arrya object. This is useful in testing an array object for internal correctness (more robust testing than the nondestructive testing done in the Test_Consistancy() array member function).
- The use of **void Diagnostic_Manager::displayPurify()** generates a report of purify problems found (currently this mechanism does not work well, since many purify errors can only be found at exit).

1.9.4 Misc Functions

All other functions not yet documented in detail.

getMessagePassingInterpretationReport();	Communication Report
getReferenceCountingReport();	Reference Counting Report
getSizeOfClasses();	Reports the sizes of all internal classes in A++/P++
getMemoryOverhead();	returns memory overhead for all arrays
getTotalArrayMemoryInUse();	returns memory use for array elements
getTotalMemoryInUse();	reports total memory use for A++/P++
getnumberOfArraysConstantDimensionInUse(dimension,inputTypeCode);	returns Count of array dimension
displayPurify (const char* Label = "");	Displays memory leaks by processor (uses purify)
getPurifyUnsupressedMemoryLeaks();	Total Memory leaked
report();	Generates general report of A++/P++ behavior
setSmartReleaseOfInternalMemory(On/Off);	Smart Memory cleanup
getSmartReleaseOfInternalMemory();	get Boolean value for smart memory cleanup
setExitFromGlobalMemoryRelease(Boolean);	setup exit mechanism
getExitFromGlobalMemoryRelease();	get Boolean value for exit mechanism
setTrackArrayData(Boolean trueFalse = TRUE);	Track and report on A++/P++ diagnostics
getTrackArrayData();	get Boolean value for diagnostic mechanism
test (*type*Array);	Destructive test of array object
buildCommunicationMap ();	Builds map of communications by processor
buildPurifyMap ();	Builds map of purify errors by processor
displayCommunication (const char* Label = "");	communication report by processor
resetCommunicationCounters ();	reset the internal message passing counting mechanism

1.10 Deferred Evaluation

Example of user control of Deferred Evaluation in A++/P++. Deferred Evaluation is a part of A++ and P++, though it is not well tested in P++ at present.

Set_Of_Tasks Task_Set;	build an empty set of tasks
Deferred_Evaluation (Task_Set)	start deferred evaluation

```
    {
    B = 0;                          array operation to set B to zero – DEFERRED
    A = B + C;                      array operation to set A equal to B plus C – DEFERRED
    }
Task_Set.Execute();                 now execute the deferred operations
```

1.11 Known Problems in A++/P++

- Copy constructors are aggressively optimized away by some compilers and this results in the equivalent of shallow copies being built in the case where an A++/P++ array is constructed from a view. Note that as a result shallow copies of A++ arrays can be made unexpectedly. A fix for this is being considered, but it is not implemented.
- Performance of A++ is at present half that of optimized FORTRAN 77 code. This is because of the binary processing of operands and the associated redundent loads and stored that this execution model introduces. A version of A++/P++ using expression templates will resolve this problem, this implementation is available and is present as an option within the A++/P++ array class library. However, compile times for expression templates are quite long.
- Internal debugging if turned on at compile time for A++/P++ will slow the execution speed. The effect on A++ is not very dramatic, but for P++ it is much more dramatic. This is because P++ has much more internal debugging code. The purpose of the internal debugging code is to check for errors as agressively as possible before they effect the execution as a segment fault of other mysterious error.
- Performance of P++ is slower if the array operations are upon array data that is distributed differently across the multiple processors. This case requires more communication and for arrays to be built internally to save the copies originally located upon different processors. P++ performance is most efficient if the array objects are aligned similarly across the multiple processors. This case allows the most efficient communication model to be used internally. This more efficient communication model introduces no more communication than an explicitly hand coded parallel implementation on a statement by statement basis.

The `ChangeLog` in the top level of the A++P++ distribution records all modifications to the A++/P++ library.

Chapter 2

Appendix

2.1 A++/P++ Booch Diagrams

Booch diagrams detail the object oriented design of a class library. The separate clouds represent different classes. Those which are shaded represent classes that are a part of the user interface, all others are those which are a part of the implementation. The connections between the "clouds" represent that the class uses the lower level class (the one with out the associated "dot") within its implementation.

2.2 A++/P++ Error Messages

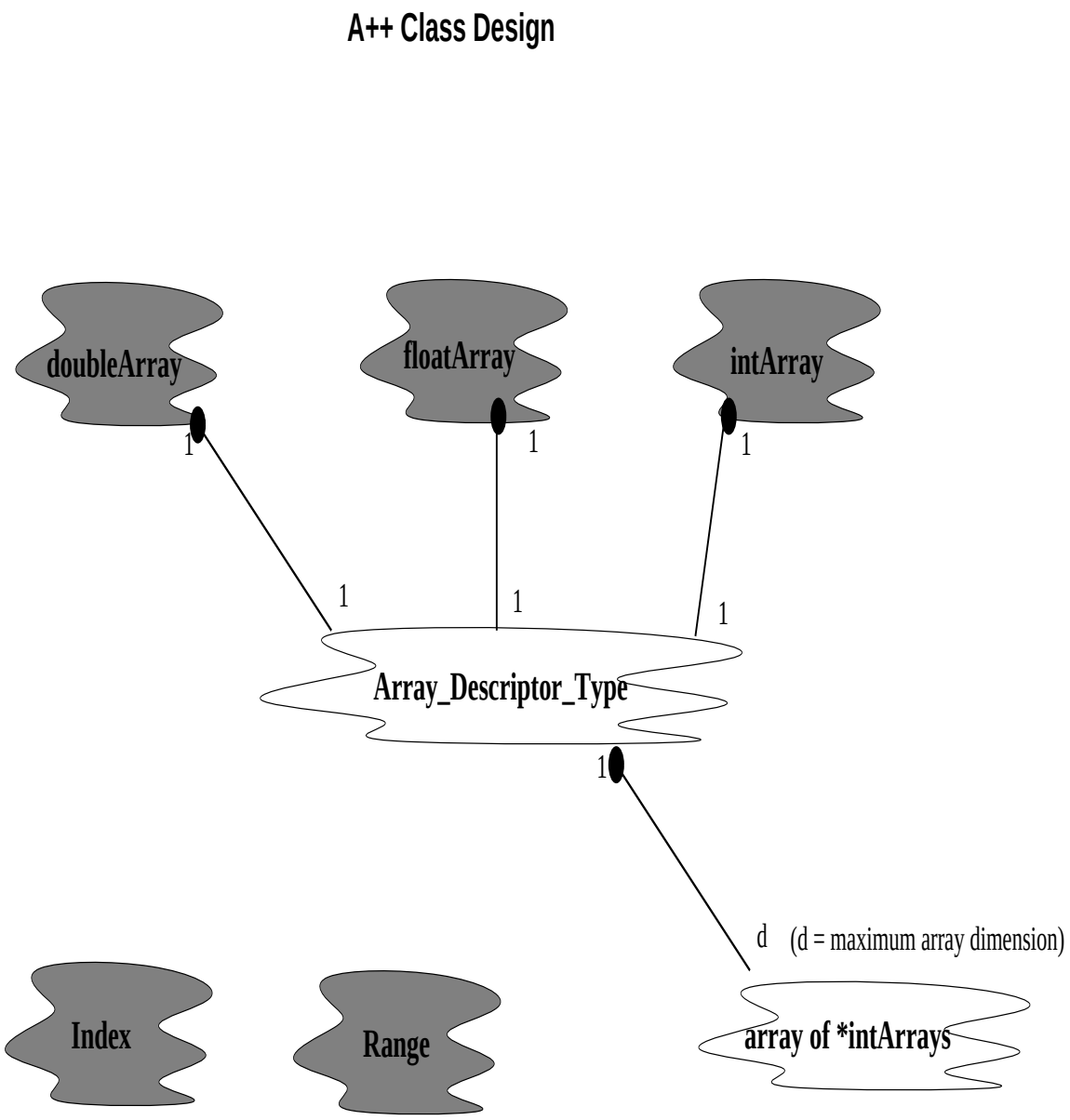

Figure 2.1: A++ Class Design.

P++ Class Design