Section 13

C/C++ Notes

Pointers and Arrays

NOTE 13.1 – (Review of NOTE 6.14)

Arithmetic Operations on Pointers

Why do Pointers Have Types?
A pointer specifies the address of an object or function. When declaring a pointer or using a pointer typecast it is necessary to specify the type to which it points. This ensures that the correct type will be accessed when the pointer is dereferenced and in the case of object pointers, that all pointer math will be done correctly.

Three arithmetic operations are permitted on object pointers (none are meaningful on void/function pointers):

- Comparison of two pointers of the same type;
- Addition of a pointer and an integer expression;
- Subtraction of two pointers of the same type.

Comparison of two pointers

Two pointers may be compared for less than, greater than, equal to, etc. only if they are of the same type. Such a comparison usually only makes sense if both point to locations within the same object (same array, same structure, same **int**, etc.).

Addition of a pointer and an integer expression

Only integer types may be added to pointers. The result is not necessarily the sum of the value of the pointer plus the value of the integer operand, but is rather the sum of the value of the pointer plus the product of the value of the operand times the size, in bytes, of the type of the object pointed to by the pointer. That is, assuming *pointer* is a pointer to any arbitrary object data type:

(**long**)(pointer + integerOperand) == (**long**)((**long**)pointer + (integerOperand * **sizeof**(*dataType*)))

Another way to state this is that what really gets added to a pointer is data type offsets. That is, if 2 is added to a **double** pointer, what really gets added is the number of bytes in 2 doubles (typically 16 bytes). That is why it is always necessary to declare the type of the object a pointer points to. The following examples assume a meaningful conversion of pointer types to arithmetic types:

```
    long *p = (long *)0;        /*  initialize p to 0  ---  Assume sizeof(long) is 4 bytes */
then
    p + 1              /*  expression type == (long *)    expression value == 4       p final value == 0  */
    (int)p + 1         /*  expression type == (int)       expression value == 1       p final value == 0  */
    (float)p + 1       /*  expression type == (float)     expression value == 1.0F    p final value == 0  */

    p++                /*  expression type == (long *)    expression value == 0       p final value == 4  */
    p  =  0            /*  reinitialize p to 0 */
    (int)p++           /*  expression type == (int)       expression value == 0       p final value == 4  */
    (int)(p++)         /*  …same as above, but with unnecessary parentheses…  */
    p  =  0            /*  reinitialize p to 0 */
    ((int)p)++         /*  expression type == (int)       expression value == 0       p final value == 1  */
            /* this last case is not permitted in standard C/C++ because a typecast does not yield an lvalue */
```

Subtraction of two pointers

This works in a manner similar to the addition previously described. Both pointers must be of the same type and be pointing to locations within the same object. The result of the subtraction will not necessarily be the numerical difference between the values of the two pointers but rather the number of objects between the two pointers. Assume type **double** occupies 8 bytes:

```
    double *pointerA = (double *)32;          /* declare and initialize a pointer to double */
    double *pointerB = (double *)0;           /* declare and initialize a pointer to double */

    ptrdiff_t items = pointerA − pointerB;    /* items = 4 (number of doubles between the pointers) */
```

NOTE 13.2 – (Review of NOTE 6.1)

The ***Right-Left*** Rule

(From the *C Programming Guide*, Jack Purdum, Que Corp., 1983)

With the many legal combinations of pointers, arrays, and functions that are possible in the C and C++ languages, deciphering a program author's variable declarations or typecasts, or writing your own, can sometimes be bewildering if much more than a simple pointer or array is involved. As an example, assume that the following declaration for the variable *table* and its English translation:

char **(*table(**void**))[24];

"*table* is a function returning a pointer to an array of 24 pointers to pointers to **chars**"

Note that the function takes **void** parameters but this is obvious and need not be stated. Whether or not the translation actually means anything to you depends upon your understanding of the language itself. While all translations may be determined by knowledge of operator precedence, there is an easier, totally mechanical method called the *Right-Left* rule.

To use the Right-Left rule, we must first define the English translation for the four legal attributes that can occur in a declaration. Whenever one of these attributes is found, merely replace it with the English translation according to the following table:

	Attribute	Meaning	English
	()	function	function returning
	[]	array	array of *n*
	*	pointer	pointer to
(C++ only)	&	reference	reference to

Be careful to discern the "function" attribute from the parentheses used as punctuators. This is always evident from context. The procedure for the Right-Left rule is as follows:

1. Start with the identifier (or in the case of a typecast, where the identifier would be);
2. Look to the right for an attribute and if found, substitute its English equivalent and repeat this step;
3. Look to the left for an attribute and if found, substitute its English equivalent and repeat this step;
4. Continue steps 2 and 3, working your way out until the data type is reached on the left.

Note that some declarations are illegal in C/C++, although they are syntactically correct. Some illegal ones are: a function returning an array, an array of functions, and a function returning a function. You may, however, use pointers to these. Similarly, you cannot have an array of references. Here are some examples of declarations deciphered using the Right-Left rule:

double (*cat)();	"*cat* is a pointer to a function returning a **double**"
int *dog[];	"*dog* is an array of pointers to **int**s"
char (*house)[10];	"*house* is a pointer to an array of 10 **char**s"
short car[][20];	"*car* is an array of arrays of 20 **short**s"
int *&golf;	"*golf* is a reference to a pointer to an **int**"
float *&(*(**(*pen)())[6][9])(**int** y);	"*pen* is a pointer to a function returning a pointer to a pointer to an array of 6 arrays of 9 pointers to functions returning references to pointers to **float**s"

NOTE 13.3A

Reversing The ***Right-Left*** Rule

Once programmers have mastered the Right-Left rule and can easily decipher any declarations/typecasts they may encounter, they are still sometimes unable to actually write the proper syntax themselves even though they can state in words what they want. The Right-Left rule can be used in reverse to mechanically solve this problem. First, using the same sentence structure produced by the Right-Left rule itself, write in words what you would like to declare. Then use the steps in the following flowchart to produce the proper C/C++ declaration syntax. If you instead wish to create a typecast, simply write a declaration first, then remove the identifier name and the semicolon and place parentheses around what remains!

Flowchart Steps

1. Write the name of the identifier being declared by the sentence and go to step 2, 3, or 4 as appropriate.

2. **If** "*array of*" or "*function returning*" are next in the sentence, place the appropriate attribute to the right of everything and go to step 2, 3, or 4 as appropriate.

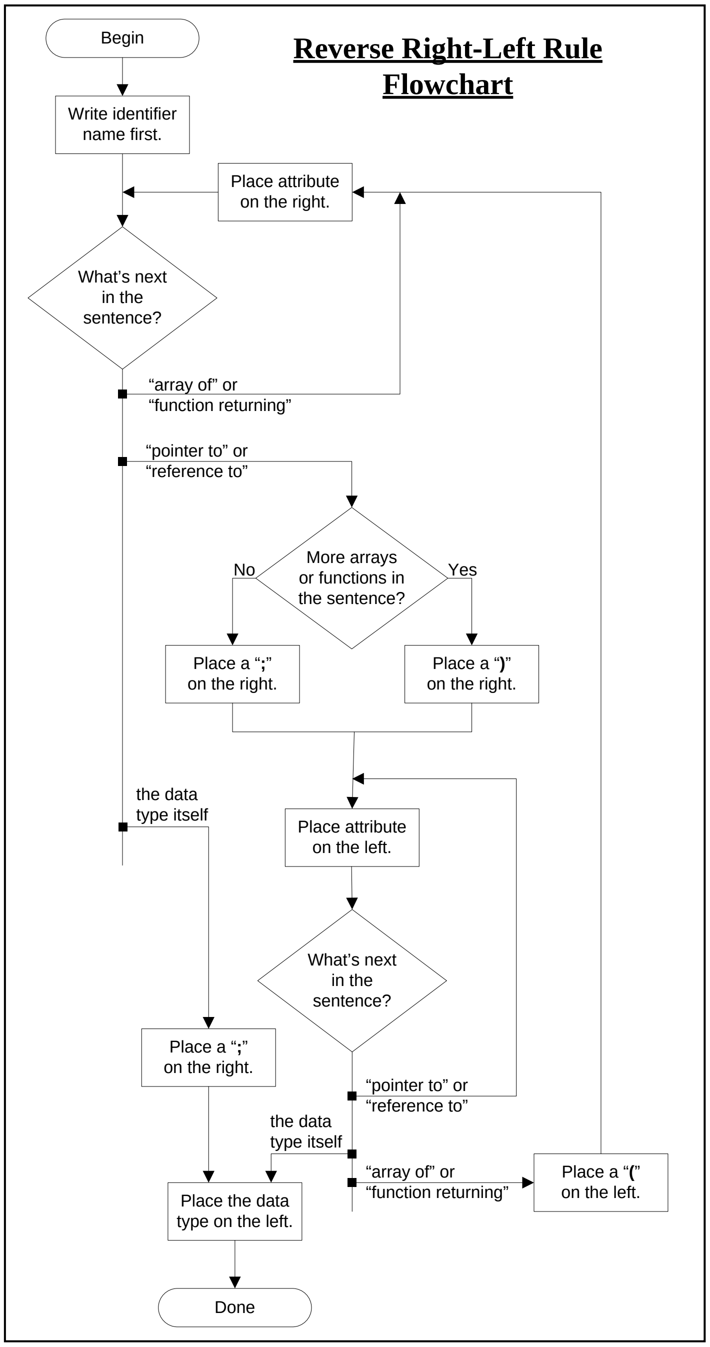
3. **Else if** "*pointer to*" or "*reference to*" are next in the sentence, notice whether "*array of*" or "*function returning*" appear anywhere else in what remains of the sentence. If so, place a closing parenthesis to the right of everything. Otherwise, use a semicolon. Then place the appropriate attribute to the left of everything and go to step 5, 6, or 7 as appropriate.

4. **Else**, the data type <u>must</u> be next in the sentence (or something is wrong). So place a semicolon to the right of everything, place the data type to the left of everything, and <u>you're done</u>!

5. **If** "*pointer to*" or "*reference to*" are next in the sentence, place the appropriate attribute to the left of everything and go to step 5, 6, or 7 as appropriate.

6. **Else if** "*array of*" or "*function returning*" are next in the sentence, place an opening parenthesis to the left of everything, place the appropriate attribute to the right of everything, and go to step 2, 3, or 4 as appropriate.

7. **Else**, the data type <u>must</u> be next in the sentence (or something is wrong). So place the data type to the left of everything and <u>you're done</u>!



Reverse Right-Left Rule Flowchart
Begin
Write identifier name first.
Place attribute on the right.
What's next in the sentence?
"array of" or "function returning"
"pointer to" or "reference to"
More arrays or functions in the sentence?
No
Yes
Place a ";" on the right.
Place a ")" on the right.
the data type itself
Place attribute on the left.
What's next in the sentence?
Place a ";" on the right.
"pointer to" or "reference to"
the data type itself
"array of" or "function returning"
Place a "(" on the left.
Place the data type on the left.
Done

..............CONTINUED

NOTE 13.3B　　**...............CONTINUATION**

Reversing The ***Right-Left*** Rule, cont'd.

The following examples illustrate using the Right-Left rule in reverse to convert declarations stated in words into proper C/C++ syntactical declarations:

Declare *w* such that: "*w* is a reference to an **int**"

```
1.   w                                  //  step 1
2.   &w;                                //  step 3
3.   int &w;                            //  step 7 – done!
```

Declare *x* such that: "*x* is an array of 3 pointers to **int**s"

```
1.   x                                  //  step 1
2.   x[3]                               //  step 2
3.   *x[3];                             //  step 3
4.   int *x[3];                         //  step 7 – done!
```

Declare *y* such that: "*y* is a pointer to a pointer to a function returning a **double**"

```
1.   y                                  //  step 1
2.   *y)                                //  step 3
3.   **y)                               //  step 5
4.   (**y)()                            //  step 6
5.   double (**y)();                    //  step 4 – done! (don't forget function
                                        //                  parameters, if any)
```

Declare *z* such that: "*z* is an array of 3 arrays of 4 pointers to functions returning pointers to **int**s"

```
1.   z                                  //  step 1
2.   z[3]                               //  step 2
3.   z[3][4]                            //  step 2
4.   *z[3][4])                          //  step 3
5.   (*z[3][4])()                       //  step 6
6.   *(*z[3][4])();                     //  step 3
7.   int *(*z[3][4])();                 //  step 7 – done! (don't forget function
                                        //                  parameters, if any)
```

Create a typecast for:

"a reference to a pointer to an array of 9 pointers to functions returning pointers to floats"

First a variable named *t* of that same type is declared:

```
1.   t                                  //  step 1
2.   &t)                                //  step 3
3.   *&t)                               //  step 5
4.   (*&t)[9]                           //  step 6
5.   *(*&t)[9])                         //  step 3
6.   (*(*&t)[9])()                      //  step 6
7.   *(*(*&t)[9])();                    //  step 3
8.   float *(*(*&t)[9])();              //  step 7
```

Finally, the declaration is converted to a typecast:

```
9.   (float *(*(*&)[9])())              //  done! (don't forget function
                                        //             parameters, if any)
```

NOTE 13.4 – (Review of NOTE 7.13)

Partially Indexed Arrays and the ***Decayed Right-Left*** Rule

Array Designators and Automatic Type Conversions
An *array designator* is any expression, such as an array's name, whose actual type is "array of …". With only four exceptions an array designator is automatically converted to (treated as) a pointer to the first element in that array, where the type of that pointer is often referred to as the array's *decayed* type. For example, the name of an array of **int**s would decay to a pointer to the first **int** in the array, resulting in a decayed type of "pointer to **int**". The four exceptions to this decay occur when the designator is the sole operand of the address, **sizeof**, or **_Alignof** operators, or is a string literal used to initialize a character array.

Partially Indexed Arrays
Expressions involving array designators with one or more rightmost indices missing (partially indexed) are often used. The actual types as well as the decayed types of any of these expressions can be determined easily using the Right-Left rule or a modification of it known as the "decayed" Right-Left rule. To determine the data type of such an expression, look at the original declaration and circle the portion of it corresponding to the partially indexed expression. Then use the Right-Left rule or the decayed Right-Left rule on that declaration, treating the circled portion as the entity being declared.

The Decayed Right-Left Rule
To obtain a statement of the decayed type of any partially indexed expression, simply substitute the words "decays to" in place of the word "is" and substitute the words "pointer to" in place of the first occurrence of the words "array of *n*" in the Right-Left rule definition.

Declaration: **int** z[2][3][4];		
Expression	**Actual Type**	**Decayed Type**
z[i][j][k]	"*z[i][j][k]* is an **int**"	"*z[i][j][k]* is an **int**"
z[i][j]	"*z[i][j]* is an array of 4 **int**s"	"*z[i][j]* decays to a pointer to an **int**"
z[i]	"*z[i]* is an array of 3 arrays of 4 **int**s"	"*z[i]* decays to a pointer to an array of 4 **int**s"
z	"z is an array of 2 arrays of 3 arrays of 4 **int**s"	"z decays to a pointer to an array of 3 arrays of 4 **int**s"

The following table demonstrates how pointers produced by various fully and partially indexed array expressions can be numerically equal although their types differ. Unless otherwise indicated, decayed types are shown. The actual data type may be found by replacing "pointer to" with "array of *n*".

Decayed Types of Full and Partial Array Expressions						

declarations:
 	
int x[a];
int y[b][a];
int z[c][b][a];
	pointer to array of *c* arrays of *b* arrays of *a* **int**s	pointer to array of *b* arrays of *a* **int**s	pointer to array of *a* **int**s	pointer to **int**	**int** (actual)	relative pointer value (2-byte **int**s)

int x[24];			&x (actual)	x	x[0]	00
int y[3][8];		&y (actual)	y	y[0]	y[0][0]	00
			y + 1	y[1]	y[1][0]	16
			y + 2	y[2]	y[2][0]	32
int z[2][3][4];	&z (actual)	z	z[0]	z[0][0]	z[0][0][0]	00
			z[0] + 1	z[0][1]	z[0][1][0]	08
			z[0] + 2	z[0][2]	z[0][2][0]	16
		z + 1	z[1]	z[1][0]	z[1][0][0]	24
			z[1] + 1	z[1][1]	z[1][1][0]	32
			z[1] + 2	z[1][2]	z[1][2][0]	40

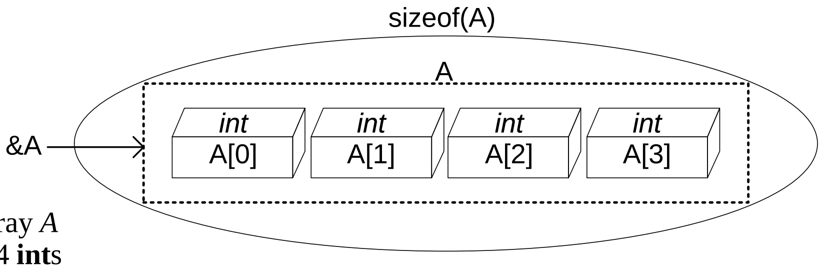
NOTE 13.5 – (Review topics from NOTE 6.16)

Array Designator Decay Examples

In the declaration ***int** A[4];* the Right-Left rule says, "A is an array of 4 **int**s". That is, A is the entire array that contains 4 **int**s, as circled below, and its data type is "array of 4 **int**s". The expression ***sizeof**(A)* is one of the four cases in which an array type does not decay so a count of the total number of bytes in the array *A* is produced. Likewise, *&A* is the another non-decaying case so the address of *A*, that is, a pointer to an array of 4 **int**s is produced. The following illustrates these concepts:

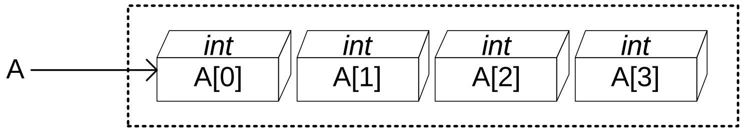
int A[4]; // sample array declaration

Case 1: No Conversion Occurs – Expression *A* remains type "array of 4 **int**s"



sizeof(A) Number of bytes in array *A*
&A Pointer to an array of 4 **int**s

Case 2: Conversion Occurs – Expression *A* decays to type "pointer to **int**" (same value and type as *&A[0]*):



A	Same as	*&A[0]*	Pointer to 1st element
A[0]	Same as	*(&A[0])[0]*	Value of 1st element
*A	Same as	**&A[0]*	Value of 1st element
A + 2	Same as	*&A[0] + 2*	Pointer to 3rd element
sizeof(A + 3)	Same as	***sizeof**(&A[0] + 3)*	Number of bytes in "*a pointer to an **int***"
p = A	Same as	*p = &A[0]*	Assign *&A[0]* to another pointer
function(A)	Same as	*function(&A[0])*	*&A[0]* passed to *function*
etc.			

Array Name Modification

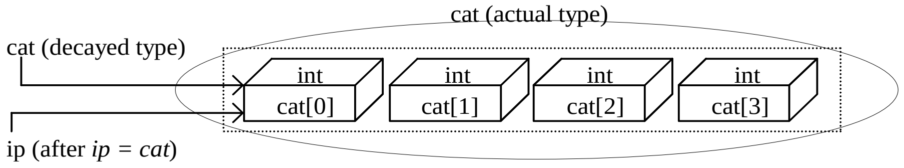
Array Names are Non-Modifiable
All objects and executable code reside at memory addresses determined by the execution environment and no program may change these addresses. For example the declaration ***int** x;* might result in memory address 0x2000 being reserved for *x*. An expression such as *&x = (int *)0x3000*, which attempts to change this address, will instead cause a compile error. Similarly, the location of an array may not be changed. Since the names of arrays in expressions like *arrayName = (int *)0x4000* and *++arrayName* decay to pointers to the starting addresses of these arrays, these expressions will also not compile.

NOTE 13.6 – (Review topics from NOTE 6.16)

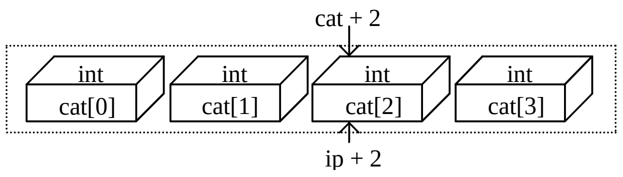
Array Accesses Are Merely Pointer Dereferences

Since the syntax used when accessing an array element, *Array[element]*, is not one of the four non-decaying cases, the result is the decay of the array type into a pointer to its first element. Such an access is, therefore, nothing more than a pointer dereference and any pointer of the same type and pointing to the same place can be used in place of the array type in all such cases:

```
int cat[4];                  /* declares an array of 4 ints */
int *ip = cat;               /* declares a pointer to an int and points it where cat points */
```



```
cat        or   ip          /* a pointer to the first int in the array (same as &cat[0]) */
*cat       or   *ip         /* dereferences the pointer cat/ip and accesses cat[0] */
cat[0]     or   ip[0]       /* dereferences the pointer cat/ip and accesses cat[0] */

cat + 1    or   ip + 1      /* a pointer to the second int in the array (same as &cat[1]) */
*(cat + 1) or   *(ip + 1)   /* dereferences the pointer (cat/ip + 1) and accesses cat[1] */
cat[1]     or   ip[1]       /* dereferences the pointer (cat/ip + 1) and accesses cat[1] */

cat + 2    or   ip + 2      /* a pointer to the third int in the array (same as &cat[2]) */
*(cat + 2) or   *(ip + 2)   /* dereferences the pointer (cat/ip + 2) and accesses cat[2] */
cat[2]     or   ip[2]       /* dereferences the pointer (cat/ip + 2) and accesses cat[2] */
```



```
cat + i    or   ip + i      /* a pointer to the (ith + 1) int in the array (same as &cat[i]) */
*(cat + i) or   *(ip + i)   /* dereferences the pointer (cat/ip + i) and accesses cat[i] */
cat[i]     or   ip[i]       /* dereferences the pointer (cat/ip + i) and accesses cat[i] */
```

The last group of expressions illustrates the general case of the simple pointer arithmetic used to access an array element, where *i* is any integer expression and the other operand is any type of object pointer. Therefore,

cat + i	points to the same element as	*&cat[i]*
**(cat + i)*	accesses the same element as	*cat[i]*

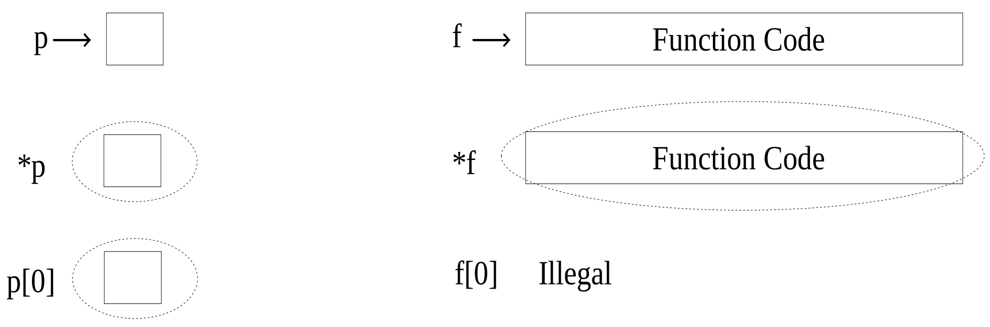
The notation used on the left is known as "pointer offset" notation while that on the right is called "index" notation. Note the following important facts:

- The compiler always converts index notation into pointer offset notation, that is:
  cat[i] becomes **(cat + i)* and *i[cat]* becomes **(i + cat)*
- Addition is commutative, that is, *(cat + i) == (i + cat)*
- Therefore, *cat[i]* and *i[cat]* are functionally identical! Both represent addition between a pointer and an integer type, which is one of the three legal pointer arithmetic operations;
- The additional implication of all of this is that a properly initialized pointer of the appropriate type may be used in place of an array name to access array elements.

NOTE 13.7

All Indexed Expressions Are Merely Pointer Dereferences

***p* is Identical to *p[0]* for Object Pointers**
The use of index notation is not limited to accessing array elements but in fact may be used to access any object. The illustrations below show that for any object pointer *p* the forms **p* and *p[0]* are just two ways of doing exactly the same thing: accessing the object to which *p* points by dereferencing *p*. The form used is totally a matter of personal preference (When a function pointer is used, however, the *f[0]* form is not permitted). The form *p[i]* is more versatile than **p* because in addition to being able to access the object to which *p* directly points, objects at addresses offset from address *p* may be referenced by using non-zero index values. That is, if *p* is a pointer to a **double**, *p[1]* accesses the **double** located one **double** beyond *p* in memory while *p[−1]* accesses the **double** located one **double** before *p* in memory. Assuming *p* is an object pointer and *f* is a function pointer:



For example:

```
        long double x, *pointer;

        pointer = &x;                   // initialize pointer  to point to x

        *pointer   = 36.8L;             // store into x
   or   pointer[0] = 36.8L;             // store into x
   or   0[pointer] = 36.8L;             // store into x
```

What Do Indices Really Mean?
The index (array) operator, *[]*, is just one of many C/C++ operators and its operation is no more magical than any other. Consider the expression *p[i][j]* where, moving left-to-right, the compiler parses the expression as

the individual tokens: p [i] [j]   grouped as: (p[i]) [j]

The parentheses indicate operator precedence. There are two binary operators and, thus, 4 operands. The first operator is the leftmost *[]* and its operands are *p* and *i*. The *[]* operator requires one operand to be an object pointer and the other an integer expression (it doesn't matter which is which). *p[i]*, thus, dereferences the pointer (*p* or *i*) at the specified offset, *i* or *p*. The second operator is the rightmost *[]* and its operands are *p[i]* and *j*. *p[i][j]*, thus, dereferences the pointer (*p[i]* or *j*) at the specified offset, *j* or *p[i]*.

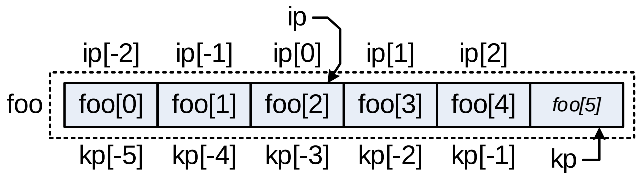
NOTE 13.8

Negative Array Indices

In C/C++ all arrays start with an index of 0 and end with an index one less than the total element count. It may, however, occasionally be desirable to access elements using negative indices.

The Technique

The most common way to access array elements using negative indices is to first initialize a pointer to some element other than the first one, then use negative (and possibly positive) indices on that pointer. This is possible because of the identity *p[i] == *(p + i)* where *p* is any object pointer (including a decayed array name) and *i* is any integer expression. For example:

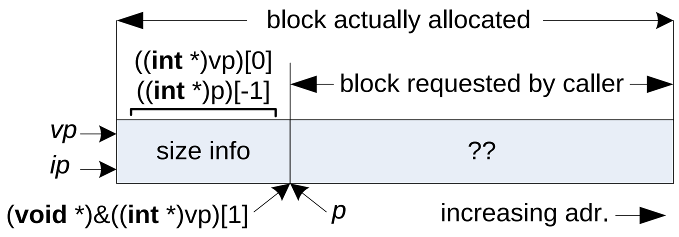


```
int foo[5];
int *ip = &foo[2];          /* ip[-2] through ip[2] are equivalent to foo[0] through foo[4] */
int *kp = &foo[5];          /* kp[-5] through kp[-1] are equivalent to foo[0] through foo[4] */

ip[-2]  and  kp[-5]         /* access foo[0] */
ip[0]  and  kp[-3]          /* access foo[2] */
ip[2]  and  kp[-1]          /* access foo[4] */
```

Note that the *address* of non-existent element *foo[5]* is used. The language standards state that the only address outside the boundaries of an array that can still be considered related to it is the first address past the end of that array. This address may be evaluated but *may not* be dereferenced. In some implementations &*foo[-1]* is an equally valid address at the beginning of the array, but on many others it is not

malloc*, *calloc*, *realloc*, and *free

A practical application of negative indices is used by the dynamic memory allocation functions *malloc*, *calloc*, and *realloc*. They must have some way of determining exactly how much memory belongs to any given allocation so that only that much will be affected if a *free* or a *realloc* is called later. For the sake of this example, assume that this size information can be represented by 1 **int**, although any suitable object type could be used. All any of the allocation functions need to do is allocate a block of memory 1 **int** larger than what the programmer actually requests, store the size information as an **int** at the beginning of that block, and return a pointer to the beginning of the memory following that **int**. In the following example, how the functions actually manipulate the heap is left to your imagination. Only the process of dealing with the negative index and typecasts is shown:



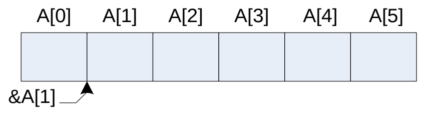
```
void *MyMalloc(size_t byteCount)
{
    void *vp;

    vp = GetHeapBlock(byteCount + sizeof(int));        /* get memory requested + 1 int */
    ((int *)vp)[0] = (int)(byteCount + sizeof(int));   /* save size of block as an int */
    return( (void *)&((int *)vp)[1] );                 /* return pointer to caller */
}
void MyFree(void *p)
{
    int *ip = &((int *)p)[-1];                         /* pointer to beginning of block */

    ReleaseHeapBlock((void *)ip, *ip);                 /* release entire block */
}
```

NOTE 13.9

Arrays Starting at One

In C/C++ all arrays start with an index of 0 and end with an index one less than the total element count. Sometimes programmers just learning C/C++ and familiar with another language are uncomfortable with this and wish they started with 1 instead. There are two ways to provide such notation, although neither technique is recommended since they introduce more problems than they solve. So, C/C++ programmers should simply learn to program like C/C++ programmers.

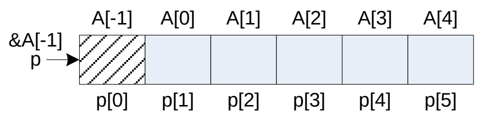
Technique 1:
This technique is legal and portable in every way and requires that an array be declared with one more element than actually needed. That extra element in every such array (element 0) is wasted and although that might not seem like much, consider cases where an element contains hundreds, thousands, or millions of bytes. As an example of this technique, assume we would like to have an array of 5 **int**s for which we could use index values of 1-5 instead of the standard 0-4.



```
#define ELEMENTS 5              /* the number of elements to use */

int A[ELEMENTS + 1];            /* has one extra element: range is [0] through [ELEMENTS] */

for (idx = 1; idx <= ELEMENTS; ++idx)       /* uses indices 1 through ELEMENTS */
        printf("%d", A[idx]);               /* prints A[1] through A[ELEMENTS] */
```

Aside from wasting an array element, this technique is a potential trouble spot simply due to the confusion it causes. The programmer must be continually aware of the fact that all standard library functions accepting or returning arrays/pointers assume that the first index is 0 and, therefore, he/she is continually forced to make adjustments to compensate for this. For example, *&A[1]* must be passed as an argument rather than simply passing the name of the array as is normally done. When a pointer is returned by one of these functions the programmer must jump through even more hoops to maintain this non-standard style of indexing.

Technique 2:
This technique, although shown in some books containing C programming "tricks", violates the language standards and is non-portable in that it simply won't work in some cases. Unlike the previous technique it doesn't waste an array element, but it does have the same inherent incompatibility with standard library functions. The technique consists of declaring an array with exactly the number of elements desired, then setting a pointer to the address of what would be the element at *array[−1]*, and finally using that pointer instead of the array name with indices starting at 1. As an example, assume we would like to have an array of 5 **int**s for which we could use index values of 1-5 instead of the standard 0-4.



```
#define ELEMENTS 5     /* the number of elements to use */

int A[ELEMENTS];       /* has number of elements needed: range [0] through [ELEMENTS - 1] */
int *p = &A[-1];       /* point to (non-existent) element A[-1] */

for (idx = 1; idx <= ELEMENTS; ++idx)       /* use indices 1 through ELEMENTS */
        printf("%d", p[idx]);               /* prints A[0] through A[ELEMENTS - 1] */
```

The language standards state that the only address outside the boundaries of an array that can be considered related to that array is that of what would be its last element if it had one additional element, that is, *&A[5]* in the above example. This address may be evaluated but not dereferenced. In some cases *&A[-1]* is an equally valid address on the other end of the array, but that is not guaranteed.

NOTE 13.10

Array Boundary Violations and Memory Segmentation

Array Boundaries Are Not Enforced
C/C++ does not attempt to detect any type of array boundary violations, although such violations can cause all kinds of problems, from program or system crashes in the best case to no apparent problem in the worst case. Although it would theoretically be possible to detect an out of bounds reference when the actual array name is used, using a pointer in place of that name could easily circumvent this detection. Therefore, no detection is attempted anywhere.

Segmented Memory Models
In the interest of efficiency, backward compatibility, and increasing programmer confusion, some systems implement what is known as a segmented memory model. In one version of this, memory addresses are not specified by one single value (known as linear addressing), but are instead produced by adding together two values, known as the "base address" and the "address offset". In a typical implementation the base value is shifted left by 4 (one hex digit) before adding. For example, if absolute address 0xABCDE is required, the base address might be 0xABC0 while the address offset might be 0x00DE. When the two are added together (after shifting the base address) the desired absolute address is produced. The same address would be obtained if the base address were 0xABCD and the offset were 0x000E, or by many other combinations. Luckily all of this happens automatically as far as the C/C++ programmer is concerned. The system sets up the base address while any addresses and pointers used within the program refer to the address offset. On such systems, however, programs using large arrays (>= 64Kb) or out of bounds array addresses must consider some of the ramifications of this scheme.

```
Base:       0xABC0
Offset:     0x00DE

             0xABC00
            +0x000DE
Absolute:    0xABCDE
```

A Legal Out of Bounds Array Address
The language standards state that the only address outside the boundaries of an array that can be considered related to that array is the address of what would be its last element, if it had one additional element. This address may be evaluated but *may not* be dereferenced. That is, for an array declared by ***int*** *foo[5]*, *&foo[5]* is a meaningful address but since there is not actually an element there it may not be dereferenced. One of the "tricks" sometimes shown in some C programming books involves the use of the address of the non-existent element just before the beginning of the array, that is *&foo[0] - 1*. On some implementations this works but it violates the language standards and is definitely non-portable. In both of the following examples assume that an **int** is 2 bytes and that the array declared by ***int*** *foo[5]* is arbitrarily located at absolute memory address 0xABC10. This means that *&foo[0] - 1* would be at absolute address 0xABC0E:

```
int foo[5];       /* Assume system chooses base address 0xABC0 and offset address 0x0010 */

&foo[0];          /* has a value of 0x0010 (which gets added to 0xABC00) */
&foo[1];          /* has a value of 0x0012 (which gets added to 0xABC00) */
&foo[0] - 1;      /* has a value of 0x000E (which gets added to 0xABC00) - it works! */


int foo[5];       /* Assume system chooses base address 0xABC1 and offset address 0x0000 */

&foo[0];          /* has a value of 0x0000 (which gets added to 0xABC10) */
&foo[1];          /* has a value of 0x0002 (which gets added to 0xABC10) */
&foo[0] - 1;      /* has a value of ??? (which gets added to 0xABC10) - it fails! */
```

What is the value of *&foo[0] - 1* (that is, the value of 0x0000 - 2) that gets added to *0xABC1* in the second example? Is it 0xFFFE, 0x0000, or something else? Whatever it is it is not consistent between implementations and certainly doesn't produce the intended absolute address.

NOTE 13.11 – (Review topics from NOTE 6.17)

Sequential Array Access – Index, Pointer Offset, and Compact Pointer Notation

Compact Pointers
Index and pointer offset notation are functionally identical as indicated by the identity *p[i] == *(p + i)*. As a result, an addition operation occurs every time either is used. When an array is to be accessed sequentially, forward or backward, using compact pointers can eliminate the index/offset variable and accompanying addition. A compact pointer expression is one in which a pointer dereference and an increment/decrement occur in the same compact expression. Many processors can efficiently mimic some of these operations in hardware. The following table shows the various forms:

Compact Pointer Expressions			
operations that affect the pointer		operations that affect the object pointed to	
Expression	Operation	Expression	Operation
*p++	post-increment	(*p)++	post-increment
*p--	post-decrement	(*p)--	post-decrement
*++p	pre-increment	++*p	pre-increment
*--p	pre-decrement	--*p	pre-decrement

The Index Method
The following methods are often used to step through an entire array of arbitrary type. They use array index notation and because of the identity *p[i] == *(p + i)*, an addition operation is required for every array reference in order to calculate the address of the desired element as an ever-changing offset from the beginning/end of the array.

```
int test[ELEMENTS], index;

for (index = 0; index < ELEMENTS; ++index)                  // forward
    cout << test[index];

for (index = ELEMENTS - 1; index >= 0; --index)             // backward
    cout << test[index];
```

The Compact Pointer Method
The following methods of stepping through an array of arbitrary type present a vastly more efficient alternative to the index methods. They avoid the address calculation by continually incrementing/decrementing a pointer, which starts out pointing to the first (or first illegal) element in the array, and stops when it has traversed the entire array. Since *test + ELEMENTS* is constant, its value is not calculated each time through the loop.

```
int test[ELEMENTS], *pointer;

for (pointer = test;  pointer < test + ELEMENTS; )          // forward
    cout << *pointer++;

for (pointer = test + ELEMENTS;  pointer > test; )          // backward
    cout << *--pointer;

for (pointer = test + ELEMENTS - 1;  pointer >= test; )     // backward? - NO! NO! (why not?)
    cout << *pointer--;
```

NOTE 13.12

Copying Arrays

There are four basic ways to copy one array to another, shown in order of increasing efficiency and speed. Simply attempting the assignment *arrayA = arrayB* is illegal and fails because array designators are constant. But even if they weren't, it would still fail to copy each element since the designators would decay to pointers and all that would be copied would be the pointer value itself.

Method 1: Element-by-Element using indexing
This method is probably the most common but is also the slowest since the program must explicitly process each individual element. The first version is the least efficient of all since it uses array index notation, which results in a math operation for every element accessed.

```
int arrayA[ELEMENTS], arrayB[ELEMENTS], index;

for (index = 0; index < ELEMENTS; ++index)
    arrayA[index] = arrayB[index];
```

Method 2: Element-by-Element using compact pointers
A much better version uses compact pointers to avoid the math operations and speeds things up somewhat. It still, however, must explicitly process every element:

```
int arrayA[ELEMENTS], arrayB[ELEMENTS], *pointerA, *pointerB;

for (pointerA = arrayA, pointerB = arrayB; pointerB < arrayB + ELEMENTS; )
    *pointerA++ = *pointerB++;
```

Method 3: Block copy using the *memcpy* Function
This method is much more efficient than the previous method and is the cleanest looking of all the methods. It uses the standard library function *memcpy*, which copies arbitrary memory blocks very efficiently since it can be tailored to the underlying hardware and is not bound to using programmer created variables and loops. Many processors have a "block copy" machine instruction that will quickly copy a block of memory without requiring any software intervention once the transfers have started.

```
int arrayA[ELEMENTS], arrayB[ELEMENTS];

memcpy(arrayA, arrayB, sizeof(arrayB));
```

Method 4: Block copy using structure/class/union assignment
The efficiency of this method is similar to that of *memcpy* since it normally uses the same underlying mechanism. Since structures, classes, and unions of identical type can be copied to one another by simple assignment and all members get copied in their entirety regardless of type, an entire array gets copied if it is one of those members.

```
struct Kluge { int arrayA[ELEMENTS]; };   /* structure type definition with an array member */

struct Kluge structA, structB;            /* declare two identical structures */

structA = structB;            /* copies structure structB into structure structA, including the entire array */
```

NOTE 13.13

The *strtok* Function

By making a sequence of calls to the *strtok* library function a delimited string may easily be broken down into multiple tokens (substrings). For example, assume a string consists of any number of comma, semicolon, or forward-slash delimited substrings, such as:

"John Doe/ Sally; 6, Plain Jane, Elvis Q. Presely III"

The substrings may be extracted using various techniques ranging from *sscanf* to stepping through the string character at a time. The most straightforward method, however, is to use *strtok*:

```
char *strtok(char *s1, const char *s2);
```

When called with a first argument that is not a null pointer, this function searches the string in *s1* for the first character that is not in separator string *s2*. If not found, there are no tokens in *s1* and the function returns a null pointer. If such a character is found it is the start of the first token and *strtok* continues to search *s1* for the first occurrence of any character that is in *s2*. If no such character is found then the token extends to the end of *s1*. If such a character is found it is overwritten with the null terminator to signify the end of the token and pointer to the first character of the token is returned. *strtok* saves a pointer to the character following the overwritten character so that searching can resume from that point when it is called the next time. Each subsequent call to *strtok* with a null pointer as its first argument resumes searching from that point, looking for any character from *s2* and repeating the sequence previously described. The separator characters in *s2* may be changed if desired between successive calls to *strtok*. The following examples extract and output the tokens from the string in *myArray*:

```
    char *cp,  myArray[] = "John Doe/ Sally; 6, Plain Jane, Elvis Q. Presely III";
```

Version with *strtok*:

```
    for (cp = myArray; cp = strtok(cp, ",;/"); cp = NULL)      /* get each token */
        puts(cp);                                              /* output the token */
    puts(myArray);                                             /* "try" to output original string */
```

Version with *sscanf*:

```
    int n = 0;
    char *cp;

    if (sscanf(myArray, "%*[,;/]%n", &n) != EOF)               /* if string contains other than separators */
        for (cp = myArray + n, n = 0; sscanf(cp, "%*[^,;/]%n", &n) != EOF; cp += n + 1, n = 0) {
            if (cp[n] == '\0') {                               /* last token in string */
                puts(cp);                                      /* output it */
                break;                                         /* end search */
            }
            cp[n] = '\0';                                      /* terminate current token */
            if (n != 0)                                        /* if token is not empty… */
                puts(cp);                                      /* …then output it */
        }
    puts(myArray);                                             /* "try" to output original string */
```

Output from both versions:

```
    John Doe
     Sally
     6
     Plain Jane
     Elvis Q. Presely III
    John Doe                  /* What went wrong here?  Why wasn't the original string output? */
```