Review of Pointers and Pointer Arithmetic

Pointers have traditionally been a stumbling block for many students learning C, but they do not need to be!

> *A C pointer is just a variable that stores the memory address of an object in your program.*

That is the most important thing to understand and remember about pointers - they essentially keep track of *where* a variable is stored in the computer's memory.

Introduction to pointers

Each variable in a program stores its contents in the computer's memory, and each chunk of the memory has an address number. The size of the memory chunk or the number of bytes associated with a variable is determined by the variable's type. For example, an `int` variable requires 4 bytes of memory while a `double` variable requires 8 bytes of memory. Conceptually, a pointer [variable] itself does not directly contain a value like an `int` or a `double`, but it contains the address in memory of another variable or function. When the `int` or `double` value is accessed through the pointer variable, then that value is accessed indirectly.

In order to produce a pointer to a variable, the unary *address-of* `&` operator is placed immediately before the variable. If a variable `x` of type `int` is defined as

```
1 int x = 100;
```

then expression `&x` determines the address of variable `x` in memory [that is the reason why the `&` operator is called *address-of* operator]. Expression `&x` evaluates to type `int *` (*pointer to* `int`) with value specified by the address number of the chunk of memory where `x` has been assigned storage. The following code fragment displays the hexadecimal memory addresses of variables `i` and `j`:

```
1 int i = 10, j = 20;
2 printf("Memory address of i is: %p\n", &i);
3 printf("Memory address of j is: %p\n", &j);
```

Let's now look at how pointer variables can be assigned. The declaration statement

```
1 int x = 100, *pi;
```

defines and initializes an `int` variable `x` to value 100 and defines a variable `pi` of type *pointer to* `int`. That is, `pi` will not be storing a regular integer value but instead will be storing the memory address where an `int` variable is given storage. Such a memory address is produced by applying the `&` operator on an `int` variable. The following assignment expression assigns the memory address where variable `x` is given storage to pointer variable `pi`:

```
1 pi = &x;
```

Supposing variable `x` is given memory storage at address 1000, then, what actually gets stored inside `pi` is value 1000 [that represents address 1000 and not a regular integer value].

Now consider assignment statement:

```
x = *pi + 2;
```

The `*` operator is called the *dereference* or *indirection* operator and it indirectly allows us to access the `int` value stored at the variable that pointer `pi` is pointing to. The compiler will evaluate expression `*pi` in the following manner:

1. In the first step, the compiler will read the value stored inside `pi` and treat it as a memory address where an `int` value is stored.
2. In the second step, the integer stored at this memory address is retrieved and that is the value of the expression. That is, expression `*pi` evaluates to a value 100 of type `int` and this is the value that is used as the left operand of the `+` operator.

The compiler will then evaluate the right hand part of the above statement to the value 102 which is then assigned to variable `x`.

In the following statement, we're indirectly assigning value 200 to variable `x`:

```
*pi = 200;
```

The expression `*pi = 200` takes integer value 200 and stores it at the memory address stored in (pointer) variable `pi`. Since `pi` contains address 1000 [where variable `x` is hypothetically given storage], value 200 will then be stored at memory address 1000, thus indirectly changing the value of variable `x` from 100 to 200.

Pointer types

Every pointer has an associated type. In general, if a variable `x` is of type `y`, then expression `&x` evaluates to a pointer to `x` and is of type `y*` [*pointer to* `y`]. If `ptry` is a variable of type `y*` (*pointer to* `y`) and is pointing to a variable of type `y`, then expression `*ptry` evaluates to the value of the pointee and is of type `y`. In other words, if `ptry` is of type pointer to `y`, then `*ptry` is of type `y`.

All data pointers of different types - data and function pointers are different - represent the values or addresses that they hold similarly. The difference, rather, is in the *type of the object* being addressed. The type of a pointer instructs the compiler how to interpret the memory found at a particular address as well as how much memory that interpretation would span.

An `int` pointer addressing memory location 1000 on a $64-$bit machine where `sizeof(int)` evaluates to 4 will span the 4 bytes ranging from memory location 1000 to location 1003. A `double` pointer addressing memory location 1000 will span the 8 bytes ranging from memory location 1000 to location 1007. Although variables of type `int` and type `double` will require different sized chunks of memory storage, all pointers - irrespective of the types of variables they point to - will be allocated the same sized chunks in memory [on 64-bit machines, the size would be 8 bytes while 32-bit machines will provide 4 bytes of storage]. This is illustrated in the following code fragment:

```
int i = 10, *pi = &i;
double d = 11.2, *pd = &d;

printf("The size of pointer to int pi is    %lu bytes\n", sizeof(pi));
printf("The size of pointer to double pd is %lu bytes\n", sizeof(pd));
```

Although pointer variables `pi` and `pd` have different types, they're allocated the same size in memory. Most modern platforms are 64-bit machines and the above code fragment will print values 8 to standard output.

Pointer declarations and definitions

A pointer is declared by prefixing the variable with the dereference operator `*`:

```
int    *pi_one, *pi_two;
char   *pc, c; // Note: pc is a char* while c is a char
double *pd;
```

Note that the dereference operator is a part of the declarator rather than the declaration type specifier. Therefore, in a comma-separated declarator list, the dereference operator must precede each variable intended to serve as a pointer. In the previous example, `pc` is interpreted as a pointer variable of type `char*` and `c` is interpreted as a variable of type `char` and not as `char*`.

A pointer should neither be initialized nor assigned with a non-address value or the address value of a variable of another type:

```
int i = 100, j = 200;
double d   = 3.14, *pd = &d;  // ok: pd points to d
int *pi1 = 0;     // ok: pi1 initialized to point nowhere
int *pi2 = &i;    // ok: pi2 initialized to point to i
pi1 = &j;         // ok: pi1 is now pointing to j
pi2 = pi1;   // ok: now both pi1 and pi2 point to j
pi2 = 0;     // ok: now pi2 points nowhere
pi2 = i;     // compile time error: pi1 assigned a non-lvalue
pi2 = pd;    // compile time error: invalid type assignment
pi_two = &d; // compile time error: invalid type assignment
```

`0` is the only non-address value that can be used as an initializer or as the rvalue in a pointer expression. Since Standard C guarantees that 0 is never a valid data address [that is a variable will never assigned storage at address 0], address 0 is used as a sentinel to indicate that a pointer is *pointing nowhere*.

Generic pointer

If all that is required is to hold an address value and possibly compare this address value to a second address value, then the actual type of the pointer does not matter. A *generic pointer* type is provided to support this behavior. A `void*` pointer can be assigned the address value of any data pointer type [a function pointer cannot be assigned to it]:

```
int i, *pi = &i;
void *pv = pi; // ok
```

`void*` indicates that the associated value is an address but that the type of the object at that address is unknown. Therefore, generic pointers cannot be dereferenced with the `*` operator, nor can they be operands of addition or subtraction operators. That is, type `void*` is considered to be neither a data pointer nor a function pointer.

Any pointer to an object [but not to a function type] can be converted to type `void*` and back without change:

```
int i = 100, *pi = &i;
char ch = 'a', *pc = &ch;
void *pv = pi; // ok
pi = pv; // ok
printf("*pi: %d\n", *pi); // will print 100 to standard output
pi = pc; // compile-time error: incompatible pointer assignment
pi = (int *) pc; // ok
printf("*pi: %d\n", *pi); // will print garbage because pi is pointing to
char
pv = pc; // ok
pi = pv; // ok - but incorrect runtime behavior
```

Practical use of generic pointers

Generic pointers provide additional flexibility in using function prototypes. When a function has a formal parameter that can accept a data pointer of any type, the formal parameter should be declared to be of type `void *`. If the formal parameter is declared with any other pointer type, the actual argument must be of the same type since different pointer types are not assignment compatible. For example, the `strcpy` function declared in standard library header file `<string.h>` copies character strings and therefore requires arguments of type `char*`:

```
char *strcpy(char *dest, char const *src);
```

However, the `memcpy` function declared in standard library header file `<string.h>` copies bytes from one memory location to another memory location without any regard to the contents of the data stored at the source memory location. Therefore, it takes a pointer to any type and so uses `void *`:

```
void *memcpy(void *destination, const void *source, size_t n);
```

The following code fragment illustrates the use of function `memcpy` to copy a set of 5 values of type `double` and a set of 10 values of type `int`. Notice the cast of pointer variables to type `void*`.

```c
#include <string.h> // for memcpy

#define SIZE_ONE 5
#define SIZE_TWO 10

double d[SIZE_ONE] = {1.1, 2.2, 3.3, 4.4, 5.5}, e[SIZE_TWO];
// copy 5 elements of array d to first five elements of array e
memcpy((void*)e, (void*)d, sizeof(double)*5);

int f[SIZE_TWO] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, g[SIZE_TWO];
// copy all elements of array f to array g
memcpy((void*)g, (void*)f, sizeof(int)*SIZE_TWO);
```

Null pointers

Every pointer type has a special value called a *null pointer*, which is different from every valid pointer of that type, compares equal to a null pointer constant `0`, and has the value *false* when used in a Boolean context. Because C guarantee that the integer constant 0 is never a valid data address, the null pointer constant is any integer constant expression with the value 0 or such an expression cast to type `void *`. The macro `NULL` is traditionally defined as the null pointer constant in the standard library header files `<stddef.h>` and `<stdio.h>` as:

```c
#define NULL ((void *) (0))
```

Here is an example code fragment that tests a pointer to determine whether it has a valid address before dereferencing it:

```c
int *pi = NULL, i = 10, j = 20;
pi = &j;
if (pi != NULL) {
  i = *pi; // ok to dereference pi since pi is pointing to valid address
}
```

Functions and pointers

Pointer parameters in functions

In C, all arguments to functions are passed using *call by value* semantic. When function `sqrt` is called to calculate the square root of a variable `x`, as in `sqrt(x)`, it is the value that argument `x` evaluates to that will get passed to the function. This means that function `sqrt` itself cannot change the value of `x`; all it can change is a copy of `x` that is made when the function is called.

Consider the first attempt at writing a function to exchange or swap the values of two integer variables passed as function arguments.

```c
#include <stdio.h>

void swap(int left, int right) {
  int temp = left;
  left = right;
  right = temp;
}
```

```
8
9  int main(void) {
10   int x = 100, y = 200;
11   printf("x: %d and y: %d\n", x, y);
12   swap(x, y);
13   printf("x: %d and y: %d\n", x, y);
14   return 0;
15 }
```

Note that function `swap` was unable to switch the values stored in `x` and `y`. This is because when function `swap` is called, the values stored inside `x` and `y` are passed to the function. Automatic variables `left` and `right` are initialized with the values of 100 and 200, respectively. Therefore, any changes made to `left` and `right` affect only these local variables and have no effect whatsoever on variables `x` and `y`. Like all automatic variables, `left` and `right` "disappear" when function `swap` returns.

Functions in C can return only a single value [although that value can be a structure]. Sometimes, a programmer might require a function to set more than one value.

All of this means that we need to somehow alter variables within a function. One way to do that is to declare these variables at file (global) scope so that the function can explicitly change these variables. In general, this approach is not recommended because the variable can be either intentionally or inadvertently changed by other functions making it hard to control changes to the variable. Another approach is to pass pointers to the variables to be modified by the function. While a function will be unable to modify these pointers, it can indirectly modify what the pointers point to.

The correct way to swap the values of the two integer variables `x` and `y` is to provide as arguments to the function `swap` the pointers to [or, addresses of] variables `x` and `y`. The function will then use the dereference operator to indirectly change the values of these variables:

```
1  #include <stdio.h>
2
3  void swap(int *ptr_left, int *ptr_right) {
4    int temp = *ptr_left;
5    *ptr_left = *ptr_right;
6    *ptr_right = temp;
7  }
8
9  // in function main ...
10 int x = 100, y = 200;
11 printf("x: %d and y: %d\n", x, y);
12 swap(&x, &y);
13 printf("x: %d and y: %d\n", x, y);
```

Function `swap` is defined with two parameters called `ptr_left` and `ptr_right`, which are of type `int *`. On line 12, the call to function `swap` consists of arguments `&x` and `&y`. The results of evaluations of expressions `&x` and `&y` are used to initialize the corresponding parameters `ptr_left` and `ptr_right`, respectively. In the body of function `swap`, the values that are swapped are not the values of `ptr_left` and `ptr_right` themselves - these are pointers. Rather, it is the values that `ptr_left` and `ptr_right` point to that are swapped.

Returning a Pointer from a Function

You can also return a pointer from a function. However, extreme care must be taken to ensure that the object being pointed to doesn't go out of scope when the function finishes executing. If object being pointed to in the function goes out of scope after the function finishes executing, then the pointer will have undefined behavior at that point - the memory it is pointing to might be overwritten by other parts of the program.

In the example below, a pointer is passed into a function and that same pointer is returned. This is safe since the pointer being returned points to a variable that exists outside of the function and will not go out of scope in the function.

```
// increment the value of object pointed to by pi by count of 1
int* plus_one_ptr(int *pi) {
  ++*pi;
  return pi;
}

// in function main ...
int i = 10;
printf("object i has value %d\n", i);
int *pi = plus_one_ptr(&i);
printf("object that pointer pi points to has value %d\n", *pi);
```

The example below illustrates unsafe use of pointers and will have catastrophic repercussions during program execution:

```
int* plus_one_bad_ptr(int *pi) {
  int x = *pi;
  ++x;
  /*
  Bad!!! This function is returning address of local variable x whose storage
  wil be assigned to other variables when this function terminates. That is,
  the memory where x is given storage will be assigned to a variable defined
  by the next function that is called during the program's execution.
  */
  return &x;
}

// in function main ...
int i = 10;
printf("object i has value %d\n", i);

int *pi = plus_one_bad_ptr(&i); // undefined behavior!!!
/*
Variable x defined in function plus_one_bad_ptr no long exists!!!
Memory where x was given storage will now be assigned to a variable
defined by the next function - in this printf()!!!
*/
printf("object that pointer pi points to has value %d\n", *pi);
```

Most compilers will flag a warning when compiling this code. Therefore, it is best to turn on all possible warning options and look closely at the output messages generated by compilers.

Relationship between pointers and arrays

Given the following array definition

```
1  int ia[] = {5, 1, 0, 11, 23, 39, 44};
```

what does it mean to simply write the expression `ia`?

C evaluates an array name to the address of the first element of the array with type *pointer to type of array element*. Thus, the array name `ia` when used in an expression [except as operand to `sizeof` operator] has type `int *` and is equivalent to expression `&ia[0]`.

Consider the following definitions:

```
1  // str is a character string - a null-terminated array of 6 characters
2  char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
3  // ps is pointer to first element of array of characters
4  char *ps   = str;
```

The picture shows the layout of the array and the pointer in memory with the assumption that array `str` is given storage starting at address 1000, while pointer variable `ps` is given storage at address 2000. Note these memory addresses are hypothetical and the actual addresses are determined at runtime by the operating system. However, the hypothetical addresses illustrated in the picture will be used throughout this section to understand the relationship between pointers and arrays.

All of the following statements will print string `"Hello"` to standard output:

```
1  printf("%s\n", str);
2  printf("%s\n", &str[0]);
3  printf("%s\n", ps);
```

In the first call to `printf` on line 1, argument `str` is the name of an array given storage from base address 1000 and therefore is evaluated as address 1000 of type `char*` [indicating that values stored at memory locations starting from 1000 are of type `char`].

In the second call to `printf` on line 2, argument `&str[0]` will evaluate as address 1000 of type `char*`. Why? Expression `str[0]` specifies the first element of array `str` which is given storage starting at memory location 1000, and the subsequent expression `&str[0]` will then evaluate to the address at which `str[0]` is given storage which is address 1000.

In the third call to function `printf`, argument `ps` evaluates to address 1000 of type `char *`. Why? The evaluation of expression `ps` means to read the contents of memory chunk labeled `ps` which in our example is assumed to be memory location 2000. For $64-$bit machines, the eight bytes in memory locations 2000 through 2007 will be interpreted as address 1000. Furthermore, the evaluation of expression `ps` will have type `char*` [indicating that the values stored at memory locations starting from 1000 are of type `char`] since `ps` is defined as variable of type `char*`.

If `ps` is a pointer to the first array element, other array elements can be referenced by performing pointer arithmetic on `ps`.

Pointer arithmetic

Pointer arithmetic in C is a powerful technique because the compiler does most of the hard work freeing the programmer from the messy details of addressing memory. C supports four forms of pointer arithmetic:

- adding integer to pointer,
- subtracting integer from pointer,
- subtracting one pointer from another, and
- comparing pointers

allows only two types of operations involving binary arithmetic operators $+$ and $-$: evaluating pointer offsets and computing the number of objects between two pointers using pointer subtraction.

In pointer offset expressions involving a pointer operand and an integer expression operand, the C compiler only permits the $+$ and $-$ binary arithmetic operators. In pointer subtraction expressions involving two pointer operands, the C compiler only permits the $-$ binary arithmetic operator.

Suppose `p` is a pointer to an object and `n` evaluates to an integer. The expression

```
p + n
```

evaluates to a pointer to an object offset by `n` objects after the object that `p` points to. More precisely, expressions

```
p + n
p - n
```

represent addresses up or down in memory from the object. To compute the new address, the compiler cannot use `n`'s value. If it did, pointer arithmetic would only access byte offsets. Instead, the compiler scales `n` by the size in bytes of the data type that `p` was declared a pointer to. For example, in the code fragment below, the addresses printed to standard output show that the first and sixth elements of an array of `char`s are 5 bytes apart:

```
char cBuff[80], *pc = cBuff;
// addresses are 5 bytes apart for character arrays */
printf("cBuff = %x and &cBuff[5] = %x\n", cBuff, pc+5);
```

In this code fragment involving an array of `double` elements, the first and sixth elements of the array are separated by 40 bytes:

```
double dBuff[80], *pd = dBuff;
// addresses are 40 bytes apart for double array since we assume
// that sizeof(double) is equivalent to 8 bytes
printf("dBuff = %x and &dBuff[5] = %x\n", dBuff, pd+5);
```

To compute the address of the sixth element of the two arrays `cBuff` and `fBuff`, the compiler is applying the following arithmetic:

```
pc + 5 == (char *)pc + 5*sizeof(char)
pd + 5 == (double *) ( (double *)pd + 5*sizeof(double) )
```

We can generalize the above concept for all C data types. Suppose `p` points to a C object of some type `y` and `n` is an integer expression, C uses the following formulas for the pointer offsets `p+n` and `p-n`:

```
p + n == (y *) ( (char *)p + n*sizeof(y) )
p - n == (y *) ( (char *)p - n*sizeof(y) )
```

If `p` and `q` point to objects of the same type, the expressions:

```
p - q
q - p
```

yield the number of objects between the two pointers. The result is a scalar that is an `int` or a `long`, depending on the implementation. Its sign may be negative, depending on which address in memory (`p`'s or `q`'s) is greater.

```
char cBuff[80], *pc1 = cBuff, *pc2 = &cBuff[5];
printf("Low address:  %x\n", pc1);
printf("High address: %x\n", pc2);

// pointers are 5 char objects apart
printf("Number of objects between pc1 and pc2: %d\n", pc2 - pc1);
```

Pointer subtraction is independent of a pointer's data type.

```
// pointers are 5 char objects apart
char   cBuff[80], *pc1 = cBuff, *pc2 = &cBuff[5];
printf("Number of objects between pc1 and pc2: %d\n", pc2 - pc1);

// pointers are 5 double objects apart
double dBuff[80], *pd1 = dBuff, *pd2 = &dBuff[5];
printf("Number of objects between pd1 and pd2: %d\n", pd2 - pd1);
```

The pointer subtraction `pc2 - pc1` is implemented by the compiler in the following manner:

```
( (unsigned int) pc2 - (unsigned int) pc1 ) / sizeof(char)
```

while the pointer subtraction `pd2 - pd1` is implemented by the compiler in the following manner:

```
( (unsigned int) pd2 - (unsigned int) pd1 ) / sizeof(double)
```

We can generalize the concept of pointer subtraction for all C data types, as we did with pointer offsets. Suppose pointers `p` and `q` point to the same object of type `y`. C use the following formula for expression `p - q`:

```
( (unsigned int) p - (unsigned int) q ) / sizeof(y)
```

The expression `sizeof(y)` is a scalar, and so is the expression's value. This means C allows pointer subtraction anywhere an `int` [or `long`] is legal.

Pointer offsets

Expressions in which an integer expression is added or subtracted from an array name or a pointer variable are called *pointer offsets*. Let's use the picture illustrating the memory layout of array `str` and pointer variable `ps` to understand pointer offsets in the following code fragment:

```
1  // using array name to reference subscripted variable str[2]
2  printf("%c\n", *(str + 2)); // displays character 'l'
3  // pointer ps points to the first element of the array
4  // ps+2 is the address at an offset of 2 char elements from the
5  // first element of the array ...
6  printf("%c\n", *(ps + 2)); // displays character 'l'
```

In the first call to `printf` on line 2, argument `*(str+2)` is evaluated as follows. First, address value 1002 of type `char*` is computed from pointer offset `str+2` by adding 2 to 1000 - the address of the first array element. Next, the value (of type `char`) stored at memory location 1002 is retrieved.

In the second call to `printf` on line 6, argument `*(ps+2)` is evaluated as follows. First, address value 1002 of type `char*` is computed from pointer offset `ps+2`. Next, the value (of type `char`) stored at memory location 1002 is retrieved.

The only difference between the two arguments `*(str+2)` and `*(ps+2)` is that `*(ps+2)` requires an
additional memory read compared to `*(str+2)`. First, the memory location with mnemonic `ps` is read to obtain an address value 1000 which is added to 2 to obtain pointer offset 1002. Next, memory location 1002 is read to obtain an integral value 108 which is interpreted in the ASCII character set to be equivalent to character `'l'`. This is the value used to initialize the corresponding parameter in both calls to function `printf`.

Now, consider the following fragment of code:

```
1  int ia[] = { 0, 1, 5, 11, 23, 39, 44, 56, 67, 13, 22 };
2  int *pi = ia;
3  printf("%d\n", *(ia + 2) ); // displays 5
4  printf("%d\n", *(pi + 3) ); // displays 11
```

Based on earlier discussions, it should be clear that argument `ia+2` is evaluated as `&ia[2]` while argument `pa+3` is equivalent to `&ia[3]`.

Arrays and pointers: Basic rule

The close relationship between arrays and pointers is presented here as a *basic rule*: If `name` is an array of elements and `i` is an integer expression, the basic rule can be written as:

```
1  name[i] == *(name + i)
```

The basic rule implies that *anywhere an array reference is legal, an equivalent pointer expression may be substituted*. Note that the basic rule is independent of the data type of the element of array name.

The basic rule can be extended by using the commutative property of arithmetic addition:

```
name[i] == *(name + i) == *(i + name) == i[name]
```

This proves that array references don't really exist because C converts array references to pointer offsets. The following code fragment references the same element twice from an array of characters:

```
char str[] = "SeaToShiningC";
int i = 5;

printf("%c\n", str[i]); // displays sixth element of str: 'S'
printf("%c\n", i[str]); // displays sixth element of str: 'S'
```

The basic rule can be extended to derive equivalent pointer expressions from array references. To the following expression:

```
name[i] == *(name + i) == *(i + name) == i[name]
```

the address-of operator `&` can be applied to both sides:

```
&name[i] == &*(name+i)
```

We know that the indirection operator `*` applied to a pointer offset evaluates to the object that the pointer offset points to. If we subsequently apply the address operator `&`, the compiler produces the pointer offset we started with. That is, the indirection and address operators cancel each other out. This makes sense since the pointer offset points to an object; the indirection operator evaluates to the object itself, and the address operator applied to this object evaluates to the address at which the object is stored in memory:

```
&name[i] == (name+i)
```

All of this again means that either an array name or pointer can be used to refer to an array element:

```
char str[] = "SeaToShiningC", *ps = str;

printf("%x %x\n", str+5, &*(str+5) );
printf("%x %x\n", ps+5,  &*(ps+5) );
```

Both `printf` statements display the address of the sixth element of `str` [in terms of subscript operator, the address is `&str[5]`) and the character stored at that element (which is `'S'`]. Again, all of this implies that it does not matter if we use a pointer or an array name with a pointer offset.

Consider the basic rule between arrays and pointers

```
name[i] == *(name + i)
```

This can be extended to show that the name of an array evaluates to the address of the first element of an array by replacing index `i` with value `0`:

```
name[0] == *name
```

Applying the address-of operator to both sides of the expression:

```
&name[0] == name
```

This rule implies that in the following two function calls to function `foo`, both argument expressions evaluate to the value of the first element of array `ia`:

```
// definition of function foo:
void foo(int x) {
  printf("%d\n", x);
}

int ia[] = { 10, 20, 30, 40, 50};
foo(ia[0]); // pass foo the value of the first element of array
foo(*ia); // pass foo the value of the first element of array
```

Also, this rule implies that in the following two function calls to function `boo`, both argument expressions evaluate to the address of the first element of the array `ia`:

```
// definition of function boo
void boo(int *px) {
  printf("%d\n", *px);
}

int ia[] = { 10, 20, 30, 40, 50};
boo(&ia[0]); // pass boo the address of first element of array
boo(ia); // pass boo the address of first element of array
```

Finally, we combine the previous rules to show array addressing is nothing but pointer offsets computed using the address of the first element of the array. First, we begin with the basic rule:

```
name[i] == *(name + i)
```

Applying the address-of operator `&` to both sides, we get:

```
&name[i] == name + i
```

In the right expression, we replace array name `name` with expression `&name[0]`:

```
&name[i] == &name[0] + i
```

Pointer arithmetic

Pointer arithmetic in C is a powerful technique because the compiler does most of the hard work freeing the programmer from the messy details of addressing memory. C allows only two types of operations involving binary arithmetic operators $+$ and $-$: evaluating pointer offsets and computing the number of objects between two pointers using pointer subtraction.

In pointer offset expressions involving a pointer operand and an integer expression operand, the C compiler only permits the $+$ and $-$ binary arithmetic operators. In pointer subtraction expressions involving two pointer operands, the C compiler only permits the $-$ binary arithmetic operator.

Suppose `p` is a pointer to an object in memory and `n` evaluates to an integer. The expressions

```
p + n
p - n
```

represent addresses up or down in memory from the object. To compute the new address, the compiler cannot use `n`'s value. If it did, pointer arithmetic would only access byte offsets. Instead, the compiler uses the size in bytes of the object to scale `n`. For example, in the code fragment below, the addresses printed to standard output show that the first and sixth elements of an array of `char`s are 5 bytes apart:

```
char cBuff[80], *pc = cBuff;
// addresses are 5 bytes apart for character arrays */
printf("cBuff = %x and &cBuff[5] = %x\n", cBuff, pc+5);
```

In this code fragment involving an array of `double` elements, the first and sixth elements of the array are separated by 40 bytes:

```
double dBuff[80], *pd = dBuff;
// addresses are 40 bytes apart for double array since we assume
// that sizeof(double) is equivalent to 8 bytes
printf("dBuff = %x and &dBuff[5] = %x\n", dBuff, pd+5);
```

To compute the address of the sixth element of the two arrays `cBuff` and `fBuff`, the compiler is applying the following arithmetic:

```
pc + 5 == (char *)pc + 5*sizeof(char)
pd + 5 == (double *) ( (double *)pd + 5*sizeof(double) )
```

We can generalize the above concept for all C data types. Suppose `p` points to a C object of some type `y` and `n` is an integer expression, C uses the following formulas for the pointer offsets `p+n` and `p-n`:

```
p + n == (y *) ( (char *)p + n*sizeof(y) )
p - n == (y *) ( (char *)p - n*sizeof(y) )
```

If `p` and `q` point to objects of the same type, the expressions:

```
p - q
q - p
```

yield the number of objects between the two pointers. The result is a scalar that is an `int` or a `long`, depending on the implementation. Its sign may be negative, depending on which address in memory [`p`'s or `q`'s] is greater.

```
char cBuff[80], *pc1 = cBuff, *pc2 = &cBuff[5];
printf("Low address:  %x\n", pc1);
printf("High address: %x\n", pc2);

// pointers are 5 char objects apart
printf("Number of objects between pc1 and pc2: %d\n", pc2 - pc1);
```

Pointer subtraction is independent of a pointer's data type.

```
// pointers are 5 char objects apart
char   cBuff[80], *pc1 = cBuff, *pc2 = &cBuff[5];
printf("Number of objects between pc1 and pc2: %d\n", pc2 - pc1);

// pointers are 5 double objects apart
double dBuff[80], *pd1 = dBuff, *pd2 = &dBuff[5];
printf("Number of objects between pd1 and pd2: %d\n", pd2 - pd1);
```

The pointer subtraction `pc2 - pc1` is implemented by the compiler in the following manner:

```
( (unsigned int) pc2 - (unsigned int) pc1 ) / sizeof(char)
```

while the pointer subtraction `pd2 - pd1` is implemented by the compiler in the following manner:

```
( (unsigned int) pd2 - (unsigned int) pd1 ) / sizeof(double)
```

We can generalize the concept of pointer subtraction for all C data types, as we did with pointer offsets. Suppose pointers `p` and `q` point to the same object of type `y`. C use the following formula for expression `p - q`:

```
( (unsigned int) p - (unsigned int) q ) / sizeof(y)
```

The expression `sizeof(y)` is a scalar, and so is the expression's value. This means C allows pointer subtraction anywhere an `int` [or `long`] is legal.

Compact pointer expressions

Compact pointer expressions involve the use of indirection operator `*`, unary increment `++` and decrement `--` operators with pointer operands. When compact pointer expressions are used, the compiler produces assembly code that is smaller in size and runs faster because most modern microprocessors have addressing modes or instructions that combine indirection with increments and decrements. Before we continue the discussion of compact pointer expressions, the list of precedence and associativity of the `++`, `--`, `*`, and `&` operators:

Precedence	Operator	Description	Associativity
1	`++` `--`	postfix increment and decrement	left-to-right

2	`++` `--`
	`*`
	`&`
		prefix increment and decrement
		dereference
		address-of
			right-to-left

Suppose a pointer variable `p` points to an array element. The following table lists four forms of compact pointer expressions:

Expression	Operation	Affects	Reads as
`*p++`	post increment	pointer	`*(p++)`
`*p--`	post decrement	pointer	`*(p--)`
`*++p`	pre increment	pointer	`*(++p)`
`*--p`	pre decrement	pointer	`*(--p)`

Expressions from the table above modify the pointer and not the pointer's object. Pointers may point to any C data type and typically access elements of an array. These expressions improve a program's execution speed if the machine implements fetch-and-increment instructions.

Again, suppose a pointer variable `p` points to an array element. The following table lists a second set of four compact pointer expressions:

Expression	Operation	Affects	Reads as
`++*p`	pre increment	object	`++(*p)`
`--*p`	pre decrement	object	`--(*p)`
`(*p)++`	post increment	object	`(*p)++`
`(*p)--`	post decrement	object	`(*p)--`

Expressions from the second table affect the pointer's object rather than the pointer. These objects may be simple data types like integers, characters, or array elements of simple types. Note that a pointer to a structure, union, or function *cannot* be used with these expressions.

Consider the following code fragment:

```
1 char str[] = "SeaToShiningC";
2 char *p    = str+5;
```

The following table illustrates the effects of compact pointer expressions. For each expression, the pointer `p` is assumed to be initialization as: `char *p = str+5;`.

Expression	Result	Pointer pointing to	Resultant string
`*p++`	`'S'`	`'h'`	`"SeaToShiningC"`

Expression	Result	Pointer pointing to	Resultant string
`*p--`	`'S'`	`'S'`	`"SeaToShiningC"`
`*++p`	`'h'`	`'h'`	`"SeaToShiningC"`
`*--p`	`'o'`	`'o'`	`"SeaToShiningC"`
`++*p`	`'T'`	`'T'`	`"SeaToThiningC"`
`--*p`	`'R'`	`'R'`	`"SeaToRhiningC"`
`(*p)++`	`'S'`	`'T'`	`"SeaToThiningC"`
`(*p)--`	`'S'`	`'R'`	`"SeaToRhiningC"`

Applications of compact pointer expressions

Standard C library function `strcpy` is declared in `<string.h>` as:

```
char *strcpy(char *dst, char const *src);
```

Function `strcpy` copies the string whose first element is pointed to by `src` to the destination character array whose first element is pointed to by `dst`, assuming the destination array is large enough to hold the characters in string `src`. Further, the function returns a pointer to the first element of the destination array. The following code fragment illustrates sample use cases of `strcpy`:

```
#include <string.h> // for strcpy
#include <stdio.h>  // for printf

int main(void) {
  char str1[81];
  char str2[] = "today is a good day";
  strcpy(str1, str2);
  printf("str1: %s\n", str1); // prints to stdout: today is a good day
  strcpy(str1, "tomorrow will be a better day");
  printf("str1: %s\n", str1); // prints to stdout: tomorrow is a better day
  return 0;
}
```

Here, the use of compact pointer expressions is examined by writing different versions of function `strcpy` called `my_strcpy`. The first version uses array subscripting:

```
// version 1: uses array subscripting
char *my_strcpy(char *dst, char const *src) {
  int i = 0;
  // iterate through each element of array whose first element is pointed
  // to by pointer src until the null character is encountered.
  // copy each encountered element from array whose first element is pointed
  // to by src to the array whose first element is pointed to by dst
```

```
  while (src[i] != '\0') {
    dst[i] = src[i];
    ++i;
  }
  dst[i] = '\0';
  return dst;
}
```

The second version continues to use array subscripting but writes a simpler `while` condition using the knowledge that null character `'\0'` has decimal value 0:

```
// version 2: uses array subscripting
char *my_strcpy(char *dst, char const *src) {
  int i = 0;
  while (src[i]) {
    dst[i] = src[i];
    ++i;
  }
  dst[i] = '\0';
  return dst;
}
```

The third version again uses array subscripting but is simpler than the previous version. This version uses the insight that an assignment expression evaluates to the value written to the lvalue operand to left of the assignment operator:

```
// version 3: uses array subscripting
char *my_strcpy(char *dst, const char *src) {
  int i = 0;
  // the while expression will copy characters from source string to
  // destination string including the null character.
  // when the null character is copied, the while condition evaluates to
  // zero (or false), and the loop terminates.
  while ((dst[i] = src[i])) { // extra parantheses to avoid naggy compiler
    ++i;
  }
  return dst;
}
```

For the fourth version of function `strcpy`, we replace the subscript operator `[]` with pointer offsets and dereference operator `*`:

```
// version 4: uses pointer offsets and dereference operator
char *my_strcpy(char *dst, const char *src) {
  int i = 0;
  while ((*(dst+i) = *(src+i))) { // extra parantheses to avoid naggy
compiler
    ++i;
  }
  return dst;
}
```

The previous version provides a path to replacing pointer offsets with pointers that point to appropriate locations in the source and destination strings.

```
// version 5: uses deference and increment operators
char *my_strcpy(char *dst, const char *src) {
  // since the function must return a pointer to the first element in the
  // destination string, a copy of the address is required
  char *tmp = dst;
  while ((*dst = *src)) { // extra parantheses to avoid naggy compiler
    ++src; // pointer to next character in source string
    ++dst; // pointer to next element in destination string
  }
  return tmp;
}
```

The sixth and final version of function `my_strcpy` uses compact pointer expressions:

```
// version 6: uses compact pointer expressions
char *my_strcpy(char *dst, const char *src) {
  char *tmp = dst;
  while ((*dst++ = *src++)) { // extra parantheses to avoid naggy compiler
    // empty by design
  }
  return tmp;
}
```

Ranges

When an array is passed to a function, the array's base address is passed by value. This makes the transfer of arrays between functions extremely efficient since copies of individual array elements don't have to be passed to the function. The function can use the array's base address in conjunction with integer offsets to iterate through every array element. To prevent too few elements from being accessed or accessing out-of-bounds elements, the array's size must also be passed. Since strings are terminated by the null character `'\0'`, functions that process strings don't require the array's size to be passed.

The function parameter initialized with the base address can be declared using either array or pointer notation. Consider the definition of function `accumulate` with array notation:

```
int accumulate(int const arr[], int size) { // array notation
  int sum = 0;
  for (int i = 0; i < size; ++i) {
    sum += arr[i]; // using subscript operator
  }
  return sum;
}
```

The same function can be defined with pointer notation:

```
int accumulate(int const *arr, int size) { // pointer notation
  int sum = 0;
  for (int i = 0; i < size; ++i) {
    sum += *(arr+i); // using pointer offset
  }
  return sum;
}
```

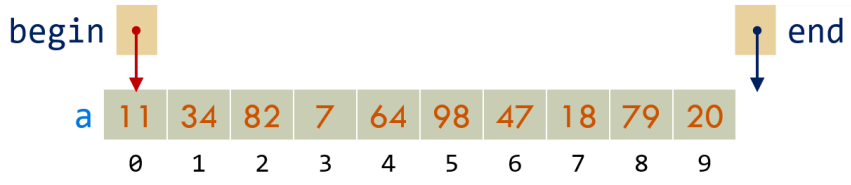
Function `accumulate` can be used to compute the sum of an entire array:

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
printf("sum: %d\n", accumulate(a, SIZE));
```

The function can also be used to compute the sum of a range of elements that comprise a slice of the array. For example, the array can be implicitly sliced into two halves and a third slice consisting of the middle six elements:

```
printf("sum: %d\n", accumulate(a, SIZE/2)); // sum of first half
printf("sum: %d\n", accumulate(a+SIZE/2, SIZE/2)); // sum of second half
printf("sum: %d\n", accumulate(a+2, 6)); // sum of middle six elements
```

Ranges of elements that might, but are not required to, specify all elements of an array can be more flexibly specified using *half-open ranges*. A half-open range is defined so that it includes the element used as the beginning of the range but excludes the element used as the end. This concept is described by the mathematical notation for half-open ranges as $[begin, end)$ where pointers `begin` and `end` point to the elements at the beginning and end of the range, respectively. The range consisting of all 10 elements of array `a` is defined with pointer `begin` pointing to the first element of `a` while pointer `end` is a *past-the-end* pointer pointing to the element after the last array element.



Function `accumulate` is redefined to incorporate the concept of half-open ranges:

```
int accumulate(int const *begin, int const *end) {
  int sum = 0;
  while (begin < end) {
    sum += *begin++;
  }
  return sum;
}
```

Parameters `begin` and `end` in the above functions represent a range of array elements. Parameter `begin` points to the first element in the range while parameter `end` is a *past-the-end* pointer pointing to the element after the last element in the range. Thus, `begin` and `end` define a *half-open range* `[begin, end)` that includes the first element [pointed to by `begin`] but excludes the last element [pointed to by `end`].

The range-based version of `accumulate` can be used to compute different ranges of elements of array `a`:

```
printf("sum: %d\n", accumulate(a, a+SIZE)); // sum of all array elements
printf("sum: %d\n", accumulate(a, a+SIZE/2)); // sum of first half
printf("sum: %d\n", accumulate(a+SIZE/2, a+SIZE)); // sum of second half
printf("sum: %d\n", accumulate(a+2, a+8)); // sum of middle six elements
```

A half-open range has two advantages. First, you can define a simple end criterion for loops that iterate over the elements: they start at `begin` and simply continue as long as `end` is not reached. Second, special handling for empty ranges is avoided. For empty ranges, `begin` is equal to `end`. The following code fragment illustrates the function for an empty range:

```
printf("sum: %d\n", accumulate(a, a)); // sum should be zero
```

Although half-open ranges provide a flexible interface, they're also dangerous. The caller must ensure that the first two arguments define a valid range. A range is valid if the end of the range is reachable from the beginning by incrementing the pointer to successively point to each array element. This means that it is up to the caller to ensure that both pointers point to elements in the same array and that the beginning is at a lower memory address than the end. Otherwise, the behavior is undefined, and endless loops or out-of-bounds memory accesses may result.