Review of Compact Pointer Expressions

What are 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 may potentially produce 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. The precedence and associativity of `++`, `--`, `*`, and `&` operators are:

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:

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

The following table illustrates effects of compact pointer expressions. Rows are not cumulative - for each expression, pointer `p` is initialized as: `char *p = str+5;`.

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

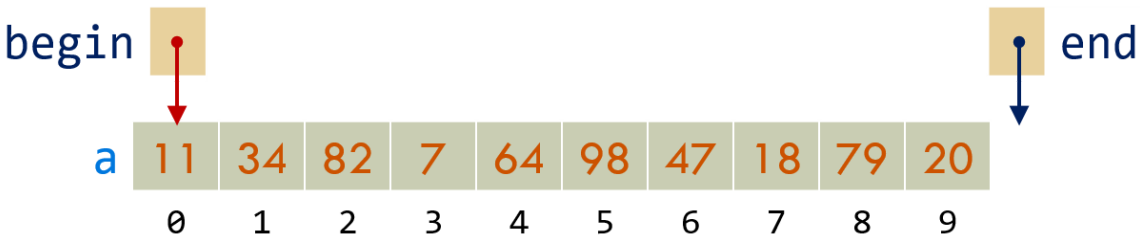
Function `accumulate` is typically used to compute the sum of an entire array:

```
#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 a front half slice, a back half slice, 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.