Looping Forward Through the Characters of a C String

A lot of C string algorithms require looping forward through all of the characters of the string. We can use a `for` loop to do that. The first character of the string will be at array index 0. The most efficient way to detect that we’ve reached the end of the string is to check for the null character:

```c
char str[80] = "Some text";
for (int i = 0; str[i] != '\0'; i++)
{
    // Do something with the character str[i].
}
```

Looping Backward Through the Characters of a C String

Some C string algorithms require looping backward through all of the characters of the string instead. Once again, we can use a `for` loop to do that. The key here is to recognize that the last non-null character of the string is always located at array index (`length of the string – 1`). For example, the C string "cat" has a length of 3, so its last non-null character is located at array index 2:

```
str   0 1 2 3
      c a t \0
```

We can use the library function `strlen()` to find the length of a C string, and then loop backward from the last non-null character of the string to the first character:

```c
char str[80] = "Some text";
for (int i = strlen(str) - 1; i >= 0; i--)
{
    // Do something with the character str[i].
}
```

Converting a Lowercase Letter Character to Uppercase (Or Vice Versa)

The library function `toupper()` from the header file `<cctype>` can be used to convert a lowercase letter character to uppercase, like so:

```c
ch = toupper(ch);
```

If the character variable `ch` contains a lowercase letter, the uppercase equivalent of that letter will be returned by the function. Otherwise, the value of the character will be returned unchanged. In both cases, we can then assign the return value from the function back to the character variable.

Of course, this works identically with the individual elements of an array of `char`:

```c
str[i] = toupper(str[i]);
```
A similar function called `tolower()` can be used to convert uppercase letters to lowercase.

**Determining the Type of a Character**

The header file `<cctype>` also contains several functions that allow you to test whether or not a character belongs to a particular type: a digit, an alphabetic character, a lowercase letter, etc.

These functions are part of the older C library and predate the `bool` data type added in C++, so they take a character as their argument and return an integer – either 0 (meaning `false`), or “not 0” (meaning `true`).

Note that “not 0” is not the same thing as 1. Any non-zero value is evaluated as true by C and C++, and the exact value the function returns is compiler-dependent. It’s important to take that fact into account when you code a test:

```c
// Wrong way to code the test - there is no guarantee that
// the isdigit() function will return 1 if str[i] is a digit.
if (isdigit(str[i]) == 1)
{
    // The character str[i] may or may not be a numeric digit.
}

// Correct way to code the test - C++ will treat any non-zero
// return value as true and zero as false.
if (isdigit(str[i]))
{
    // The character str[i] is a numeric digit.
}
```

The “is” functions available in the library include:

- `isalpha()` – is the character an alphabetic character (i.e., a letter)?
- `isalnum()` – is the character an alphanumeric character (i.e., a letter or a numeric digit)?
- `isdigit()` – is the character a numeric digit?
- `islower()` – is the character a lowercase letter?
- `isupper()` – is the character an uppercase letter?
- `ispunct()` – is the character a punctuation character? (Read the documentation to see what qualifies.)
- `isspace()` – is the character a white-space character (space, tab, newline, carriage return, etc.)?

**Comparing One Character of a String**

Another common thing you might want to do is compare a character of a C string to another character. For example, you might want to answer the question, “Does this string contain this particular character?” or you might want to count how many times a particular character occurs in a string.
Since a C string is an array of `char` and the `char` data type can be compared using the standard relational operators, this is straightforward to code:

```c
if (str[i] == ch)
{
    // The character str[i] is equal to the character ch.
}
```

**Passing C Strings to and Returning C Strings from Functions**

If you look at online documentation for most of the C string functions, you might see some data types in the prototypes that you may not recognize yet. For example, here is the prototype for the `strcpy()` function:

```c
char* strcpy(char* destination, const char* source);
```

If you were able to look at the actual code for the `strcpy()` function, it might look something like this:

```c
char* strcpy(char* destination, const char* source)
{
    int i;

    for (i = 0; source[i] != '\0'; i++)
        destination[i] = source[i];

    destination[i] = '\0';

    return destination;
}
```

The return data type and the data type of the parameter `destination` are `char*` (pointer to a `char`). The data type of the parameter `source` is `const char*` (pointer to a `char` constant).

Every variable in C and C++ has an address, a number that identifies its location in the computer’s memory. A **pointer** is a special type of variable that can hold the address of another variable. The pointer can then be used to access the value of the variable to which it points.

Normally, when you pass a variable to a function what gets passed is a copy of the variable’s **value**. The value is copied into a local variable that you declared as a function parameter.

Arrays in C and C++ work differently. Making a copy of an entire array every time you pass it to function could potentially be expensive in terms of both time and memory. Instead, the designers of C chose to pass a copy of the **address** of the array (technically, the address of the first element of the array) to the function. That address is copied into a local pointer variable declared as a function parameter. The pointer can then be used to access the values stored in the array in the calling function.

In the prototype above, `source` and `destination` are pointers to the first element of arrays of `char`. For the function to work correctly, the pointer `destination` must point to an array of `char`
that is a C string (i.e., that contains a null character), while destination just needs to point to an array of char. The strcpy() function can access the contents of both arrays, but all that actually gets copied to the function are the addresses of the arrays, which are only four or eight bytes each.

A potential downside of passing the address of an array to a function is that the function will be able to change the contents of the array in the calling routine. While we want strcpy() to be able to change the contents of the array destination in the calling function, we don’t want it to be able to accidentally change the contents of source. The pointer source might be pointing to a C string literal, for example. Changing that could have unforeseen effects on our program.

That’s where the const keyword comes in. The const keyword that is part of the declaration of source means that pointer can’t be used to change the values stored in the array to which it points. They can be read or copied, but not altered. If you try to assign a new value to source[i], you will get a syntax error. You’ll see the data type const char* used for any C string parameter that could potentially be a C string literal.

Note that the code in the function body treats source and destination as if they were arrays (which in fact they are). In C/C++, array elements are always accessed by their address. Any pointer variable may be subscripted like an array name, and an array name is automatically converted into a pointer to the first element of the array when you subscript it.

You could rewrite the strcpy() function to declare the data types using array syntax rather than pointer syntax:

```c
char[] strcpy(char destination[], const char source[])
{
    int i;

    for (i = 0; source[i] != '\0'; i++)
        destination[i] = source[i];

    destination[i] = '\0';

    return destination;
}
```

That changes nothing. The variables source and destination are still pointers to the first element of arrays of char, so most C/C++ programmers will use the pointer syntax.

The strcpy() function also returns the pointer destination. That pointer points to the first element of the copied string. A lot of the C string functions do this.

When you call the strcpy() function, you might ignore the return value. For example:

```c
char name[21] = "John Smith";    // Initialize name to "John Smith".
strcpy(name, "Amy Jones");       // name now contains "Amy Jones".
```

However, we can use the value returned by a C string function like strcpy() or strcat() to perform nested function calls:
char firstName[] = "Amy";
char lastName[] = "Amy";
char fullName[21];

strcat(strcat(strcpy(fullName, lastName), ", "), firstName);

The final statement in the code above is equivalent to doing these three separate function calls:

strcpy(fullName, lastName);
strcat(fullName, ", ");
strcat(fullName, firstName);

Nesting function calls like the example above doesn't actually make your code more efficient – you're still performing three function calls, just all in one line of code. It is a bit less typing, but it's also a lot less readable and harder to understand.

**The size\_t Data Type**

Another new data type that you may encounter in reading the documentation for the C and C++ string functions is size\_t. For example, here is the prototype for the strlen() function:

```c
size_t strlen(const char* str);
```

And here is the prototype for the length() method of the C++ string class:

```c
size_t length() const noexcept;
```

The name size\_t essentially means “size type”, and you will typically see this data type used to specify the size or length of things – like the length of a C string returned by the strlen() function, for example.

This is not one of the “built-in” data types of C/C++. Instead, it was defined in several header files using the typedef command:

```c
typedef unsigned int size_t;
```

The statement above defines a new data type called size\_t as an unsigned int. It's basically just creating a shorter alias for the unsigned int data type.

An unsigned int is an integer that can be 0 or greater than 0 but cannot be negative. If you think about it a bit, it makes sense to use this data type for something like the length of a C string. You can have a C string that is length 0 (nothing but a null character), but how could you possibly have a string with a negative length?

For the most part, you can treat a size\_t variable as if it were an integer without any issues. One problem that does come up however is that the C++ compiler will give you a warning if you compare an integer to an unsigned integer:
string str = "something something something";

for (int i = 0; i < str.length(); i++)
    str[i] = toupper(str[i]);

Here, i is data type int while the return value from the length() method is data type size_t (an unsigned int). The compiler will flag this comparison with a warning.

There are two ways to fix this kind of warning.

1. We can type cast the return value of the length() method to an integer.

```cpp
string str = "something something something";

for (int i = 0; i < (int) str.length(); i++)
    str[i] = toupper(str[i]);
```

2. We can declare i as data type size_t or unsigned int instead of data type int.

```cpp
string str = "something something something";

for (size_t i = 0; i < str.length(); i++)
    str[i] = toupper(str[i]);
```