C++ strings vs C-Strings

C-strings are not first-class types like the C++ `string` type. They do not work like the built-in types. Look at this example, which tries to **assign**, **compare** and **concatenate** two strings:

```
string str1 = "Hello", str2 = "World";
char cstr1[] = "Hello", cstr2[] = "World";

str1 = "Goodbye";            // assignment OK
cstr1 = "Goodbye";           // ILLEGAL
if (str1 < str2) ...         // comparison OK
if (cstr1 < cstr2) ...       // INCORRECT
str1 += ", ";                 // OK
cstr1 += ", ";                // ILLEGAL
```

For the C++ **`string`** class, assignment, comparison and concatenation work in the same manner as the built-in types. Use the **assignment operator**, the **relational operators**, and the **`+=`**. **Not so** for C-strings, where you must use functions from the **`<cstring>`** header to perform the same functionality.

- **`strcpy(dest, src)`** is used instead of assignment
- **`strcat(dest, src)`** is used instead of **`+=`**
- **`strcmp(cstr1, cstr2)`** is used instead of the relational operators

In addition, in place of the member function **`size()`**, you use the **`strlen(cstr)`** function which counts the number of characters before the **`'\0'`**.

C-String Assignment

Assignment means "copy the thing on the right into the storage on the left". Instead of the assignment operator, used by the built-in types, C-strings use the `strcpy()` function, from the standard library header `<cstring>`, as shown below:

```
const size_t kMaxLen = 4096;
char dest [kMaxLen];
// Assume src is a C-style string
strcpy(dest, src);
```

Both `src` and `dest` are C-strings. (`src` is a common abreviation for *source*, where the characters are copied ***from***, while `dest` stands for *destination*, where the characters are copied ***to***). `strcpy(dest, src)` copies the characters, one by one, from `src` into `dest`, stopping the `'\0'` is copied. However:

- You don't know if the **actual size** of the C-string source is less than `4095` characters (+1 for the null character). **Thus this code contains a security flaw**.
- You normally won't need anywhere near `4096` characters allocated for destination, so **the code is inefficient**.

It is up to you to ensure that there is enough space in `dest` to hold a copy of `src`. The icon used in front of the code does not mean that the code is buggy; instead, it means that the **function itself is intrinsically dangerous**; it's like the symbol found on rat poison.

The library function itself **makes no attempt to check whether the destination has enough room** to hold a copy of the source string. Even if there is not enough memory the function keeps copying, possibly overwriting other data; this called a **buffer overflow**.

The strncpy Function

The possibly safer `strncpy()` function copies only a specified number of characters from `src` to `dest`. Here's how it works:

- Call `strncpy()` with a `dest`, a `src`, and a `count` of characters.
- If the `'\0'` in `src` is found **before** the specified number of characters have been copied, then `strncpy()` will fill the remainder with `'\0'`.
- If the `NUL` character is not found in `src` before the number of characters have been copied, then **you must manually append a terminating `NUL`**.

Here is a **semi-safe copy**, given the previous example that avoids overflow (although it doesn't ensure that all of `src` was actually copied; for that you need a loop).

```
dest[kMaxLen - 1] = '\0';   // pre-terminate
strncpy(dest, src, kMaxLen - 1);
```

If I've seemed somewhat equivocal about using `strncpy`, you should know that it's widely regarded as a still unsafe function. If you want to know more, here are some links:

- Stop Using strncpy already!
- strncpy? Just say no
- strncpy: Not the function you are looking for

The strcat Functions

Concatenation is the province of the `strcat()` (completely unsafe), and the `strncat()` (marginally safer) functions. Here is a (buggy) example using the functions:

```
const size_t kLen = 10;
char cstr[kLen] = "Goodbye";
strcat(cstr, " cruel world!");      // OOPS
cout << strlen(cstr) << " " << cstr << endl;
```

When you run, you'll likely see:

```
Goodbye cruel world!
```

The C-string **`cstr`** has room for 9 characters, but you **appear to** have stuffed **`21`** characters (including the **`NUL`**), into that smaller space. Not really, of course: **this is a buffer overflow** and the actual **results are undefined**.

The **`strncat()`** function is marginally safer, if **fairly tricky to use correctly**. If used incorrectly, it overflows just like **`strcat()`**. Here is the prototype:

```
char * strncat(char *dest, const char *src, size_t count);
```

The tricky part is that **`count`** is not the maximum size of the result, but the maximum number of characters to be copied; you must first calculate the **correct combined maximum**, before calling the function.

```
const size_t kLen = 39;         // max total characters
const cstr[kLen + 1] = "This is the intial string";
const char *str2 = "Extra text to add to the string";
strncat(cstr, str2, kLen - strlen(cstr));
```

This **isn't efficient** (since you need to count the characters in **`cstr`** first), but it **does stop copying when the destination string is full**.

> ***Security Note**: **`strncat()`** does not check for sufficient space in **`dest`**; it is therefore a potential cause of buffer overruns. Keep in mind that count limits the number of characters appended; it is not a limit on the size of **`dest`**.*

Comparing C-Strings

Do not use the relational operators (`<`, `==`, etc.) to compare C-strings. Instead, use the library function **`strcmp()`**, which compares **`s1`** and **`s2`** **lexicographically** and returns an integer indicating their relationship:

- **Zero** if the two strings are equal.
- **Negative** if the first string lexicographically precedes the second string. (Lexicographically simply means "in dictionary order").
- **Positive** if the first string lexicographically follows the second string.

To use **`strcmp()`** correctly:

- Call the function and save the int it returns.
- Use the returned value with a relational operator.
- **Don't** treat the return value from **`strcmp()`** as a Boolean expression.
- Don't repeatedly call **`strcmp()`** on the same strings (inefficient).

Here's a quick example. The C-strings **`s1`** and **`s2`** are initialized elsewhere. Since we don't need to modify either argument, we can use "pointer-style" C-strings.

```
const char *s1 = ..., *s2 = ...;

int result = strcmp(s1, s2);

if (result == 0) ...            // equal
else if (result < 0) ...        // s1 < s2
else ...                        // s1 > s2
```

The strlen Function

In this lesson we're going to look at several implementations of the standard library functions beginning with `strlen()`. We'll finish by learning to write your own C-String functions.

To find the length of a string, you **count characters** until you reach the `'\0'`. Here is an implementation that uses array notation.

```
size_t strlen(const char str[])
{
    size_t len = 0;
    while (str[len] != '\0') len++;
    return len;
}
```

*Note that the return type **must be `size_t`** (not `int`), because we can't have a negative length on a string. The array must be `const`, otherwise it **would be illegal** to call the function using a C-string literal.*

Another alternative is to advance the pointer until it reaches the end of the string, and then to **use pointer subtraction** (or **pointer difference**) to determine the number of characters. Here's a version that does that:

```
size_t strlen(const char *str)
{
    const char *cp = str;
    while (*cp != '\0') cp++;
    return cp - str;
}
```

We can actually write this in an even more cryptic style. I **don't encourage you to write code like this**, since it is quite a bit more error prone, but it **is a common C idiom** so you should recognize it when you see it.

```
size_t strlen(const char *str)
{
    const char *cp = str;
    while (*cp++) /* do nothing */;
    return cp - str - 1; // cp points to 1 past the NUL
}
```

The strcpy Function

The `strcpy()` function is often even more cryptic than `strlen()`.

```
char * strcpy(char *dest, const char *src)
{
    char *result = dest;
    while (*dest++ = *src++) /* do nothing */;
    return result;
}
```

This **very, very common idiom** has so many potential pitfalls, that it is likely that your IDE will mark it with a warning. Although technically not incorrect, it is intrinsically dangerous code, since a small mistake can break the loop entirely.

- The **body of the `while` loop is empty**; all of the work occurs in the extremely streamlined test expression: **`*dest++ = *src++`**
- This expression is **not a comparison**, but an **embedded assignment**. If you accidently use a comparison, the loop will not work.
- The expression copies the character addressed by **`src`** into the address indicated by **`dest`**, incrementing each pointer after the character is copied. **If you use prefix increment instead of postfix, this does not work** .
- The result is zero—and therefore **`false`**—only when the code copies the **`NUL`** character at the end of the string.

Note that this leaves both pointers pointing one-past the **`NUL`** characters in their respective strings.

The strcmp Function

Like `strcpy()`, most implementations of `strcmp()` are cryptic. Here's the version from GNU C:

```
int strcmp(const char *s1, const char *s2)
{
    const unsigned char *a1, *a2;
    for (a1 = reinterpret_cast<const unsigned char *>(s1),
         a2 = reinterpret_cast<const unsigned char *>(s2);
         *a1 == *a2; a1++, a2++)
        if (*a1 == '\0') return 0;
    return *a1 - *a2;
}
```

The GNU version of **`strcmp()`** returns **the difference** between the first two mismatched characters. **`a1`** and **`a2`** are temporary pointers to **`unsigned char`**, so the characters can be interpreted as raw values between **`0-255`**. The pointers are initializaed by using a **`reinterpret_cast`**.

Here is an alternate (Apple/Next/PPC) version of the same function, which returns **`0`**, **`+1`** and **`-1`** instead of the difference between the characters. This version, written in 1992, uses traditional C-style casts to handle the **`signed`**/**`unsigned`** instead of a C++ **`reinterpret_cast`**.

```
int strcmp(const char *s1, const char *s2)
{
    for ( ; *s1 == *s2; s1++, s2++)
        if (*s1 = '\0') return 0;   // reached both NULs. Equal
    return ((*(unsigned char *)s1
            < *(unsigned char *)s2) ? -1 : +1);
}
```

Writing Your Own Functions

To write your own C-String functions you can use either array notation or pointer notation, whichever you find more convenient; **neither** is more efficient than the other. The things you need to remember are:

- **Find** the `NUL` character in the string. All C-String functions rely on this.
- **Preserve** the `NUL` character in the string. It is up to you to make sure that any destination strings are correctly terminated.

To make this more concrete, let's look at a couple of examples.

Find First

To find the first occurrence of a particular character in a string, you'd employ the **linear search** algorithm:

```
Loop through a string until the NUL character
    If current character is the target
        Return its index
Return the error code
```

Assuming that we use `-1` for the error code then an **array-notation** implementation of the function could look like this:

```
int find(const char a[], char target)
{
    for (int i = 0; a[i] != '\0'; ++i)
        if (a[i] == target)
            return i;
    return -1;
}
```

A (more cryptic) **pointer-notation** implementation might look like this:

```
int find(const char* s, char target)
{
    auto *p = s;
    while (*p && *p != target) p++;
    if (*p) return p - s;
    return -1;
}
```

The **temporary pointer** `p` is moved through the C-string `s`. The expression `*p` is false when the `NUL` is encountered. Since the loop **must end** when you encounter the `NUL`, or, when you find the `target`, you know that the loop **terminates in every case**.

After the loop is over, there are **two** possibilities. If `p` is pointing at **any** character, it **must** be the `target` character. That means you can use **pointer difference** to return the index. Otherwise, `p` **must** be pointing at the `NUL` character and you can return `-1`.

Find Last

You might think that the easiest thing would be to start at the back of the string and then loop towards the front. That's what you'd do with a C++ `string`. However, with C-strings, you can't find the length **without first looking at every character**, so looping backwards is actually more inefficient than simply going forward, saving the position each time the target is found.

Here's an efficient **array-notation** implementation of the function:

```
int find_last(const char a[], char target)
{
    int result = -1;
    for (int i = 0; a[i] != '\0'; ++i)
        if (a[i] == target)
            result = i;
    return result;
}
```

Find First of Any

Suppose you want to find the position of the first digit inside a C-string. You can't just use `find()` since you want to look for **any** digit. You'd want a function you could call like this:

```
int pos = first_of_any(cstr, "0123456789");
```

Here's the algorithm you use:

```
Look through every character in str
   Compare the character to every character in target
     If found return the index (in str)
return error code
```

Here's an implementation of this algorithm:

```
int first_of_any(const char *str, const char* target)
{
    const char * p1 = str;
    while (*p1 != '\0')
    {
        const char * p2 = target;
        while (*p2 != '\0')
        {
            if (*p2 == *p1) return p1 - str;  // found it
            p2++;
        }
        p1++;
    }
    return -1;
}
```

Finding Substrings

Searching for a C-style substring inside another C-string is a little bit of work. Similar to `first_of_any()` this is most easily done by using three temporary pointers. Here's the algorithm:

```
String str, string target
Pointer p set to str
While *p != 0
   Pointer p1<-p
   Pointer p2<-target
   While *p1 && *p2 && *p1 == *p2
      p1++, p2++
   If *p2 == '\0' return p - str
   p++
Return the error code
```

You can implement it yourself on the next page.

find_target.cpp

CodeCheck Reset