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`.*