C strings and C++ strings

C strings (a.k.a. null-terminated strings)

Declaration

A C string is usually declared as an array of `char`. However, an array of `char` is **NOT** by itself a C string. A valid C string requires the presence of a terminating "null character" (a character with ASCII value 0, usually represented by the character literal `'\0'`).

Since `char` is a built-in data type, no header file is required to create a C string. The C library header file `<cstring>` contains a number of utility functions that operate on C strings.

Here are some examples of declaring C strings as arrays of `char`:

```
char s1[20];             // Character array - can hold a C string, but is not yet a valid C string

char s2[20] = { 'h', 'e', 'l', 'l', 'o', '\0' };     // Array initialization

char s3[20] = "hello";                               // Shortcut array initialization

char s4[20] = "";        // Empty or null C string of length 0, equal to ""
```

It is also possible to declare a C string as a pointer to a `char`:

```
char* s3 = "hello";
```

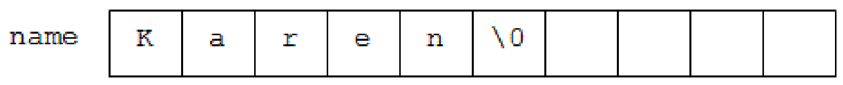
This creates an unnamed character array just large enough to hold the string (including the null character) and places the address of the first element of the array in the `char` pointer `s3`. This is a somewhat advanced method of manipulating C strings that should probably be avoided by inexperienced programmers. If used improperly, it can easily result in corrupted program memory or runtime errors.

Representation in Memory

Here is another example of declaring a C string:

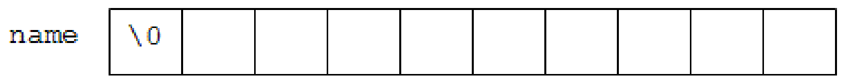
```
char name[10] = "Karen";
```

The following diagram shows how the string `name` is represented in memory:



The individual characters that make up the string are stored in the elements of the array. The string is terminated by a null character. Array elements after the null character are not part of the string, and their contents are irrelevant.

A "null string" is a string with a null character as its first character:

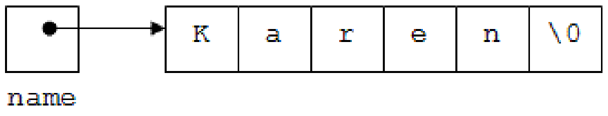


The length of a null string is 0.

What about a C string declared as a `char` pointer?

```
char* name = "Karen";
```

This declaration creates an unnamed character array just large enough to hold the string `"Karen"` (including room for the null character) and places the address of the first element of the array in the `char` pointer `name`:



Subscripting

The subscript operator may be used to access the individual characters of a C++ string:

```
cout << s3[1] << endl;         // Prints the character 'e', the second character in the string "Hello"
```

Since the name of a C string is converted to a pointer to a `char` when used in a value context, you can also use pointer notation to access the characters of the string:

```
cout << *s3 << endl;           // Prints the character 'h', the character pointed to by s3

cout << *(s3 + 4) << endl;     // Prints the character 'o', the fifth character in the string "Hello"
```

String Length

You can obtain the length of a C string using the C library function `strlen()`. This function takes a character pointer that points to a C string as an argument. It returns an `unsigned int`, the number of valid characters in the string (not including the null character).

Examples

```
char s[20] = "Some text";

cout << "String length is " << strlen(s) << endl;     // Length is 9

// Loop through characters of string
for (int i = 0; i < (int) strlen(s); i++)
   cout << s[i];
cout << endl;
```

String Comparison

Comparing C strings using the relational operators `==`, `!=`, `>`, `<`, `>=`, and `<=` does not work correctly, since the array names will be converted to pointers. For example, the expression

```
if (s1 == s2)
   {
   ...
   }
```

actually compares the addresses of the first elements of the arrays `s1` and `s2`, not their contents. Since those addresses are different, the relational expression is always false.

To compare the contents of two C strings, you should use the C library function `strcmp()`. This function takes two pointers to C strings as arguments, either or both of which can be string literals. It returns an integer less than, equal to, or greater than zero if the first argument is found, respectively, to be less than, to match, or be greater than the second argument.

The `strcmp()` function can be used to implement various relational expressions:

```
if (strcmp(s1, s2) < 0)      // If the C string s1 is less than the C string s2
   {
   ...
```

```
      }

   if (strcmp(s1, s2) == 0)     // If the C string s1 is equal to the C string s2
      {
      ...
      }

   if (strcmp(s1, s2) > 0)      // If the C string s1 is greater than the C string s2
      {
      ...
      }

   if (strcmp(s1, s2) <= 0)     // If the C string s1 is less than or equal to the C string s2
      {
      ...
      }

   if (strcmp(s1, s2) != 0)     // If the C string s1 is not equal to the C string s2
      {
      ...
      }

   if (strcmp(s1, s2) >= 0)     // If the C string s1 is greater than or equal to the C string s2
      {
      ...
      }
```

Assignment

A character array (including a C string) can **not** have a new value assigned to it after it is declared.

```
   char s1[20] = "This is a string";
   char s2[20];

   s1 = "Another string";     // error: invalid array assignment

   s2 = s1;                   // error: invalid array assignment
```

The C++ compiler interprets these assignment statements as attempts to change the *address* stored in the array name, not as attempts to change the *contents* of the array. The address stored in an array's name may not be changed, since this could result in loss of access to the array storage.

To change the contents of a character array, use the C library function `strcpy()`. This function takes two arguments: 1) a pointer to a destination array of characters that is large enough to hold the entire copied string (including the null character), and 2) a pointer to a valid C string or a string literal. The function returns a pointer to the destination array, although this return value is frequently ignored.

Examples

```
   char s1[20];
   char s2[20] = "Another new string";

   strcpy(s1, "");              // Contents of s1 changed to null string

   strcpy(s1, "new string");     // Contents of s1 changed to "new string"

   strcpy(s1, s2);              // Contents of s1 changed to "Another new string"
```

If the string specified by the second argument is larger than the character array specified by the first argument, the string will overflow the array, corrupting memory or causing a runtime error.

Input and Output

The stream extraction operator `>>` may be used to read data into a character array as a C string. If the data read contains more characters than the array can hold, the string will overflow the array.

The stream insertion operator `<<` may be used to print a C string or string literal.

Concatenation

The C library function `strcat()` can be used to concatenate C strings. This function takes two arguments: 1) a pointer to a destination character array that contains a valid C string, and 2) a pointer to a valid C string or string literal. The function returns a pointer to the destination array, although this return value is frequently ignored.

```
char s1[20] = "Hello";
char s2[20] = "friend";

strcat(s1, ", my ");     // s1 now contains "Hello, my "

strcat(s1, s2);          // s1 now contains "Hello, my friend"
```

The destination array must be large enough to hold the combined strings (including the null character). If it is not, the array will overflow.

Passing and returning

Regardless of how a C string is declared, when you pass the string to a function or return it from a function, the data type of the string can be specified as either `char[]` (array of `char`) or `char*` (pointer to `char`). In both cases, the string is passed or returned *by address*.

A string literal like `"hello"` is considered a constant C string, and typically has its data type specified as `const char*` (pointer to a `char` constant).

C++ string objects

Declaration

A C++ string is an object of the class `string`, which is defined in the header file `<string>` and which is in the standard namespace. The `string` class has several constructors that may be called (explicitly or implicitly) to create a string object.

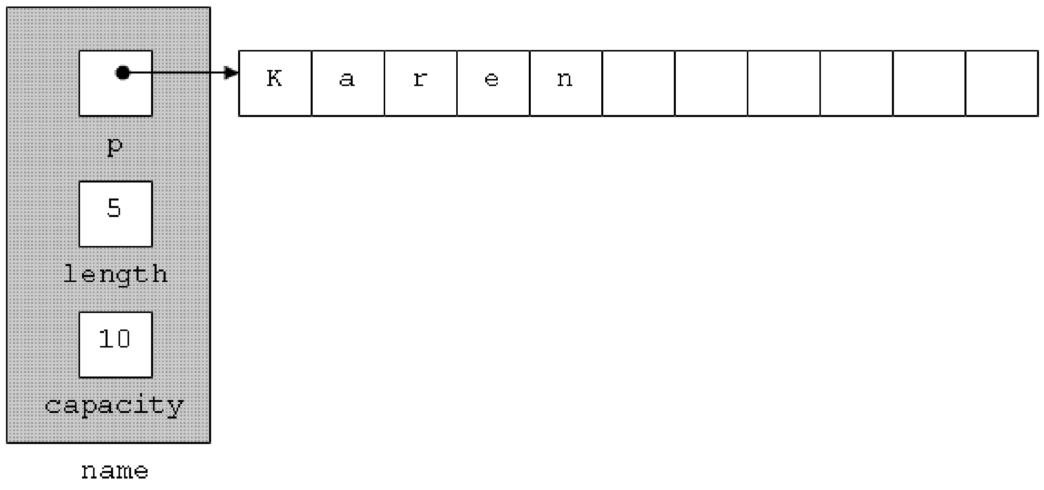
Examples

```
string s1;               // Default constructor - creates an empty or null C++ string of length 0, equal to ""

string s2("hello");      // Explicit constructor call to initialize new object with C string

string s3 = "hello";     // Implicit constructor call to initialize new object with C string

string s4(s2);           // Explicit constructor call to initialize new object with C++ string

string s5 = s2;          // Implicit constructor call to initialize new object with C++ string
```

Representation in Memory

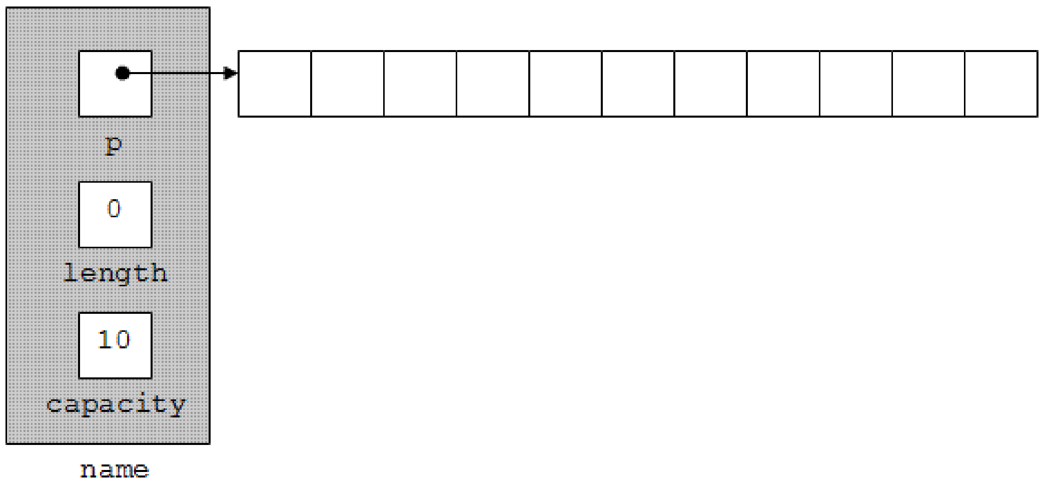
Here is another example of declaring a C++ string:

```
string name = "Karen";
```



`name` is a `string` object with several data members. The data member `p` is a pointer to (contains the address of) the first character in a dynamically-allocated array of characters. The data member `length` contains the length of the string. The data member `capacity` contains the number of valid characters that may currently be stored in the array.

A "null string" is a string with a `length` of 0:



The length of a null string is 0.

Subscripting

The subscript operator may be used to access the individual characters of a C++ string:

```
   cout << s3[1] << endl;     // Prints the character 'e', the second character in the string "Hello"
```

The reason this works is a C++ feature called *operator overloading*. Using the subscript operator with a C++ `string` object actually calls a special member function named `operator[]` that has been defined as part of the `string` class. The subscript specified inside the brackets is passed as an argument to the member function, which then returns the character at that position in the string.

The name of a C++ `string` object is **not** a pointer and you can not use pointer notation with it or perform pointer arithmetic on it.

String Length

You can obtain the length of a C++ string using the `string` class methods `length()` or `size()`. Both of methods return an `unsigned int`, the number of valid characters in the string (not including the null character).

Examples

```
string s = "Some text";

cout << "String length is " << s.length() << endl;     // Length is 9

// Loop through characters of string
for (int i = 0; i < (int) s.size(); i++)
   cout << s[i];
cout << endl;
```

String Comparison

C++ strings may be compared using the relational operators `==`, `!=`, `>`, `<`, `>=`, and `<=`. A C++ string may be compared to either another C++ string or a valid C string, including a string literal. All such relational expressions resolve to the Boolean values `true` or `false`.

Examples

```
if (s1 > s2)         // Compare two C++ strings
   {
   ...
   }

if ("cat" == s2)     // Compare C string literal and C++ string
   {
   ...
   }

if (s3 != cstr)      // Compare C++ string and array containing C string
   {
   ...
   }
```

Like subscripting, this works because of operator overloading.

Assignment

You can assign a C++ string, a C string, or a C string literal to a C++ string.

Examples

```
string s1 = "original string";
string s2 = "new string";
char s3[20] = "another string";

s1 = s2;                       // s1 changed to "new string"

s1 = s3;                       // s1 changed to "another string"

s1 = "yet another string";     // s1 changed to "yet another string"
```

Once again, this works because of operator overloading.

Input and Output

The stream extraction operator `>>` may be used to read data into a C++ string object.

The stream insertion operator `<<` may be used to print a C++ string object.

Concatenation

The operator `+` may be used to concatenate C++ strings. C++ strings, C strings, and string literals may all be concatenated together in any order. The result is a C++ string object that may be assigned to another C++ string object, passed to a function that takes a C++ string object as an argument, printed, etc.

```
string s1 = "Hello";
string s2 = " good ";
char s3[10] = "friend";

s1 = s1 + ", my " + s2 + s3;     // s1 now contains "Hello, my good friend"
```

Passing and returning

C++ string objects are passed and returned *by value* by default. This results in a copy of the string object being created.

To save memory (and a possible call to the copy constructor), a string object is frequently passed *by reference* instead.

Converting one string type to the other

Sometimes you have one type of string, but you want to use a function or method that requires the other type. In that case, it's useful to be able to convert one string type to the other.

You can easily create a C++ string object from a C string or string literal. Declare the string object and pass the C string or string literal as a constructor argument.

What if you have a C++ string object and need to convert it to a C string? The `string` class provides a method called `c_str()` that returns a pointer to the underlying array of characters that holds the contents of the string. If the array does not already contain a null character (it usually does), one is appended. The C string returned by this method can not be modified, but it can be used, printed, copied, etc.

```
char s1[20];
string s2 = "My C++ string";

strcpy(s1, s2.c_str());     // Copies the C string "My C++ string" into the array s1
```

So which of these string types should I use?

Use C++ strings whenever possible, since arrays are evil. Unfortunately, it's not always possible to avoid using C strings.

- Command line arguments are passed into `main()` as C strings
- File names have to be specified as C strings when opening a file
- There are a number of useful C string library functions that have no equivalent in the C++ `string` class
- C++ strings can't be serialized in binary format without writing a bunch of extra code
- etc.

In short, a good C++ programmer needs to understand and be able to manipulate both types of strings.