Strings in C, C++ and Java

C strings Strou. (5.2-5.3), HS. (13)

C-string: null terminated array of char

char greet[5] = { 72, 73, 76, 65, 0 };
char greet[] = { 'H', 'I', 'L', 'A', '\0' };//"AUTO COUNT"
char greet[] = "HILA";  // Initializer shorthand
.data
greet: .byte 72
    .byte 73
    .byte 76
    .byte 65
    .byte 0
greet: .asciiz "HILA"
**String Literal:** "This is a string" (The null char is added.)

Except when it is used to initialize a char array, a string literal is "array of the appropriate number of const characters"

Since an array name signifies a **pointer**, a string literal can be assigned to a **char**

```cpp
char greet[] = "HILA";
    // IMPORTANT
    // DIFFERENCE!!
char * pchar = "LUKY";

greet[1] = 'E'; // OK
pchar[1] = 'E'; // BAD!!
cout << "LUKY";
    // might now print LEKY
```

---

```cpp
char greet[] = "HILA";
char * pchar = "LUKY";

pchar = greet;

greet[0] = 'A';
pchar[1] = 'B';
*(greet+2) = 'C';
*(pchar+3) = 'D';
cout << greet << pchar;
    // prints ABCDABCD
pchar = pchar + 1;
cout<< pchar <<' '<<greet;
    // prints BCD ABCD
```
Correspondence summary:

$\&a[0]$: Synonymous with \texttt{a}

\texttt{a[0]}: Synonymous with \texttt{*a}

$\&a[i]$: Synonymous with \texttt{a+i}

\texttt{a[i]}: Synonymous with \texttt{*(a+i)}

Equivalent statements:

\begin{align*}
\texttt{a[8]} &= \text{"!"}; \\
\texttt{*}(&\texttt{a+8}) &= \text{"!"};
\end{align*}

\begin{align*}
\texttt{px} &= \texttt{a + 8}; \\
\texttt{px} &= \texttt{a}; \\
\texttt{px} &= +\texttt{8};
\end{align*}
Summary: After the code segment

```c
char a[10]; char *px = a;
```
the characters in `a` can be accessed as

```c
a[0], a[1], ..., a[9]
```
or as

```c
*a, *(a+1), ..., *(a+9)
```
or as

```c
*px, *(px+1), ..., *(px+9)
```

Program examples (using subscript arithmetic)

**Example 1:** Computing the length of a string. (The length does not include the ‘\0’ char.)

(a) Array version:
```c
size_t strlen (const char *s) {
    int i = 0;
    while (s[i] != '\0')
        { // s[0...i] CONTAINS NO NULL CHARs
            i++;
        }
    // s[0...i-1] CONTAINS NO NULLs, s[i]==\0
    return (size_t) i;
}
```
Slide 7

Notes:

- The type `size_t` is usually defined to be `unsigned int` or `unsigned long`.
- The `const` qualifier indicates that string `s` is not modified by the function.
- As a function parameter, "char *s" and "char s[]" are equivalent.
- A typecast is used in the `return` statement.

Slide 8

(b) Using pointer arithmetic:

```c
size_t strlen (const char *s) {
    char *p = s;
    while (*p != '\0')
    {
        // bytes located from s up to p are NON-NULL
        p++;
    }
    // p == address of first NULL byte
    //  located at or after s
    return (size_t) (p-s);
}
```
Notes:

- The statement “char *p = s;” initializes the pointer p.
- The loop could also have been written as follows:

```c
while (*p)
    p++;
```

**Reason:** The character ‘\0’ has the (ASCII) value zero. So, the loop would end as soon as that character is seen.

- The expression (p-s) gives the length of the string.

---

**Example 2:** Copying a string. (The function considered here is slightly different from the `strcpy` function in `<string.h>`.)

**Function header:**

```c
void copy_str (char dest[], char src[])
```

(a) **Array version:** Exercise.
(b) Pointer version:
void copy_str (char dest[], char src[]) {
  while ( *(dest++) = *(src++) )
    // all copied chars are NON-NUL
  ;  // Empty loop body.
  // Last copied character is NULL
}

Notes:

- The function uses the fact that an assignment statement is an expression.
- The assignment statement ensures that the '\0' character is also copied.
- Loop terminates just after copying '\0', since the (ASCII) value of that character is zero.

C style string handling library:

- Documentation: Strou. (20.4.1), HS. (13)
  man string in Lab.
  (various C/C++ books)
- Header file: <string.h> or <cstring>
- Tests and manipulations on C strings.
- Functions operate on null terminated array of char data.
  They expect strings ending with '\0'

Some examples:
char *strcpy (char *s1,
             const char *s2)

- Copies the string s2 to s1. Returns a pointer to s1. (The '\0' character is also copied.)
`strcpy(s1, s2)` is **UNSAFE** if you cannot trust `s2`. Space at `s1` might be smaller than `length(s2)+1`.

```c
#include <string.h>
#include <iostream.h>
main(int argc, char *argv[])
{
  char BUFFER[20]; // 20 ought to be enough REALLY???
  if( argc >= 2 )
    strcpy( BUFFER, argv[ 1 ] );
  else
    ...;
  cout << BUFFER << endl; ...; }
```

$ a.out csi333

```
```

Segmentation error (or bus error)

---

Bounded size string copy function:

```c
char *strncpy (char *s1,
    const char *s2,
    size_t  n)
```

- - Copies **at most** `n` chars from `s2` to `s1`. Returns a pointer to `s1`.

- - If `s2` has `n` or more chars, then the **first** `n` chars are copied to `s1`.

- - If `s2` has less than `n` chars, then all the chars of `s2` are copied to `s1`. The remaining spots are filled with `'\0'`.

---
**Example:**

**Initial:**

\[
\begin{array}{cccccc}
\text{s1} & x & y & z & w & q & \text{\textbackslash 0} \\
\text{s2} & a & b & c & \text{\textbackslash 0} \\
\end{array}
\]

**After strncpy(s1, s2, 3):**

\[
\begin{array}{cccccc}
\text{s1} & a & b & c & w & q & \text{\textbackslash 0} \\
\text{s2} & a & b & c & \text{\textbackslash 0} \\
\end{array}
\]

**After strncpy(s1, s2, 5):**

\[
\begin{array}{cccccc}
\text{s1} & a & b & c & \text{\textbackslash 0} & \text{\textbackslash 0} & \text{\textbackslash 0} \\
\text{s2} & a & b & c & \text{\textbackslash 0} \\
\end{array}
\]

C++’s standard library’s string class provides “first class” string objects.

typedef basic_string<char> string;

```cpp
template<class Ch, class Tr = char_traits<Ch>,
        class A = allocator<Ch> >
class std::basic_string {
  public: // ...
};
```

C++ strings keep track of their own length.

**String** is an instantiated **Container Class**. C++’s **Standard Template Library** provides lots of different container classes with different efficiency and access capabilities.

They all have **value** semantics. The assignment operator = makes a **deep copy**.
#include <string>  //NO .h here!!!
main() {
    string s1 = "Knold";
    string s2 = "Tot";
    s1 = s2;
    s2[1] = 'u'; //Change Tot to Tut
    cout << s1 << " is not " << s2 << endl;
    ...
    Tot is not Tut
}

string fname = "Frodo";
string lname = "Baggins";
string fullname;
    fullname = fname + " " + lname;
    cout << fname << " plus " << lname;
    cout << " is " << fullname << endl;
    ...
Frodo plus Baggins is Frodo Baggins
Java arrays are fixed length first class objects that are manipulated through reference variables.