NAME

string, text – SDL string and text attribute types

SYNOPSIS

```
// in SDL:
interface stg {
public:
    attribute string    astring;
    attribute string    stringarray[10];

};

// signature of C++ binding for string/text types
// (The public part of the class declarations for
// string and text attributes.)
class sdl_string
{
 public:
    char get(size_t n) const;    // get n’th character
    void get(char *s) const;    // get copy of entire string
    void get(char *s, size_t from, size_t len) const; // get a range of bytes

    void set(size_t n, char c);    // set n’th character
    void set(const char *s); // s is null-terminated
    void set(const char *s, size_t from, size_t len); // set a range of bytes

    int strlen(void) const;    //length of currently stored string
    int blen() const;          // available space as binary data.

    // C Library style names for modifying the string
    const char *memcpy(const char *s, size_t len);
    void    bcopy(const char *s, size_t len);

    // conversion to C string.
    operator const char *() const;

    // const void * conversion, for use with mem/bcopy routines.
    operator const void *() const;
    const char * operator=(const char *); // conversion from C string

    const char *strcpy(const char *s);
    const char *strcat(const char *s);

    int strcmp(const char *s) const;
    int strcmp(const sdl_string &string) const;

    int strncmp(const char *s, size_t len) const;
    int strncmp(const sdl_string &string, size_t len) const;
};
```

DESCRIPTION

The SDL *string* type is a variable length string for use in the definition of SDL interface object types.

Text Attributes

The SDL *text* type is similar, and allows the same operations as the *string* type, but there may be at most one text attribute per SDL object. The contents of the text attribute of a *registered* (named) SDL object is visible as a Unix file through the Shore Server.

Attributes of SDL objects declared as *string* or *text* can be read and written within SDL/C++ programs as if they were instances of a C++ class with the method signature shown above. The C++ binding of the SDL string type is designed for easy inter-operation with C-style null-terminated strings and `char ∗` pointers, and can be used with many of the functions described in **string(3)** (declared in the header file `<string.h>.`)

Memory-Areas

String attributes can be used to store a memory area (an array of characters bounded by a count, not terminated by a null character). String attributes can be used to store such an area by using one of the **memcpy** or **bcopy** member functions described below.

Use of string-style member functions and memory-area member functions cannot be intermixed.

Modifying String Values

The member functions that modify the value of a string attribute allocate temporary storage as necessary to store string or memory area values; the value of a string attribute may be treated as a `const char ∗` pointer value in many contexts, but the attribute may only be modified through member functions of the string class.

Examples

The following examples are base on use of an SDL C++ binding based on the SDL object type definition

```
    interface stg {
    public:
        attribute string astring;
        attribute string    stringarray[10];
        attribute text   atext;
    };
```

and C++/SDL variable declarations

```
    Ref<stg> sref_val;
    char ∗s;
    char ∗s1, ∗s2;
    size_t n;
    size_t len;
    int i;
    char c;
```

The **get** member functions retrieve character values from string attributes.

```
        sref->astring.get(n);
        sref->astring.get((size_t)3);
```

returns the value of the *n*th (and *3*rd) character of the string, or *null* if the length of the string is less than *n*(3).

The statement

```
        sref->astring.get(s);
```

copies the entire string (including the terminating null) into the space addressed by *s*, while

```
        sref->astring.get(s,n,len)
```

copies *len* bytes starting at the *i*th character in the space addressed by *s*. The two statements

```
        sref->astring.get(s, 0, sref->astring.strlen()+1)
        sref->astring.get(s)
```

are equivalent.

The **set** member functions change the string value or subranges of the string value:

```
        sref.update()->astring.set(n,c);
```

sets the value of the *n*th character of the string attribute to the character *c*, extending the length of the attribute as necessary to accommodate the character (without regard to the value of the argument *c*, so using **set** this way can leave you with a string that is not null-terminated). The statement

```
        sref.update()->astring.set(s);
```

copies the entire null-terminated string *s* into the string attribute, resetting the length of the attribute. The function

```
        sref.update()->astring.set(s,i,len)
```

copies *len* bytes, starting at *s*, into the portion of the string attribute that begins with the *i*th character of the attribute. This can leave the attribute terminated with a non-null character.

A string attribute can be used as a `const char *` pointer, either by implicit coercion or by explicit casting. For example, a string attribute could be printed using **printf** by explicitly casting the reference to a `const char *` value:

```
        printf("astring: %s\n",(const char *)(sref->astring))
```

Where a `char *` pointer is required by context, this conversion will be done implicitly, e.g.

```
    extern "C" long atoi(const char *);
        int nval = atoi(sref->astring);
```

If a string attribute is uninitialized, or if it has been set to NULL by an assignment operator, the value returned by the `const char *` conversion operator will be NULL.

The function **strlen,** as in the statement

```
        sref->astring.strlen()
```

returns the current length of the string attribute; this is similar to

```
        strlen(sref->str.attr);
```

except that the former cleanly handles null-valued strings by returning 0. The function **blen** is equivalent to **strlen,** but does not check for nulls, that is, if a string attribute is used to store binary data,

```
        sref->astring.blen()
```

will return the length of the memory area stored, ignoring embedded nulls.

The method **strcpy** works as follows:

```
        sref.update()->astring.strcpy(s2)
```

copies string *s2* to the string attribute *astring* until the null character has been copied. Space is allocated as necessary. This is equivalent to the operation

```
        sref.update()->astring = s2;
```

Strcatworks**similarly:**

```
        sref.update()->astring.strcat(s2)
```

appends a copy of string *s2* to the end of the string attribute astring.

Memcpy and **bcopy** ignore the null characters:

```
        sref.update()->astring.memcpy(s2,len)
        sref.update()->astring.bcopy(s2,len)
```

copy *len* bytes from memory at location s2 into the string attribute (starting at the beginning of the attribute), and sets the length to *len.* The difference between the two is that **memcpy** returns a pointer to the resulting string, whereas **bcopy** does not.

The functions **strcmp** and **strncmp** analogues to the **string(3)** functions:

```
        sref->astring.strcmp(s)
        sref->astring.strncmp(s, n)
```

returns In these examples,

```
        sref->astring.strcmp(sref->stringarray[i])
        sref->astring.strncmp(sref->stringarray[i], n)
```

the string attribute *astring* and *stringarray* are compared. Integers greater than, equal to, or less than zero are returned when *astring* > *stringarray*, *astring* == *stringarray,* and *astring* < *stringarray,* respectively. The function **strncmp** compares at most *n* characters.

VERSION

This manual page applies to Version 1.1 of the Shore software.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

COPYRIGHT

SEE ALSO

intro(cxxlb), method(cxxlb), ref(cxxlb), and the **Shore Data Language Reference Manual**