Assignment Three (20 points extra-credit)

Overview

In this assignment, you will implement a simple object-oriented program in assembler. In it, we will implement a class similar to a simplified version of the StringBuilder class in Java, named Sb. The higher-level code, however, will be in C++, and there is a compiled version in C++ (that uses C versions of our util.s functions) that you can run as an example.

Description

The idea of our Sb class is to allow the build-up of a string piecemeal using a character buffer. At any point in its construction, the buffer contents can be copied to create a string. The buffer can then continue to be appended, or be cleared.

The purpose of Sb is to minimize the allocation of individual copies of short strings, and the continual measuring of lengths. Normally, appending two strings together consists of several steps:

1. measure the length of the first string
2. measure the length of the second string
3. allocate a new area to hold a copy of the two strings concatenated
4. copy each string to the new buffer, appending the second string to the first.

Although smart compilers should be able to optimize this, if the build of a string occurs in a loop, it is incredibly wasteful. Instead, using our Sb class, we use a preallocated buffer to copy the pieces to, keeping track of the current length of the string in the buffer. In this case, the entire string only needs copying when we want a copy or when the preallocated buffer runs out of room and it must be copied to a new larger buffer before continuing.

Example

If you have an instance of an Sb named mysb and you invoke the following sequence of function calls on mysb:

```c
mysb.append("hello ");
mysb.append("there");
cout << mysb.toString();
```

The output is `hello there` (with no newline.) If you then continue with the following calls:

```c
mysb.append(", ");
mysb.append("my ");
mysb.append("friend.");
mysb.append(\n");
cout << mysb.toString();
```

The output is `hello there, my friend.` (with a newline).

You can then either continue to append pieces to the current buffer, or call `mysb.clear()` to reset the number of characters in the buffer to zero. This allows you to reuse the current buffer when you begin creating a new string. (Note here that `mysb` has been allocated as a local variable rather than using a simulated call to `new`, which allocates a pointer to an Sb object.)

Procedure

The class definition follows on the next page. I have written and translated the code to MIPS for the constructor and for the private member function `resize`. I have also written a main program, which does the following:

1. allocates and constructs an Sb object
2. asks the user to enter a string.
3. If the string is empty, the current buffer is copied to a new string and output in a message. Then
the user is asked if they want to clear the buffer, quit, or continue. Unless quitting, repeat at 2.
4. Otherwise, if the input is multi-character, the string is appended to the buffer. If the input is a
single character, the single character is appended to the buffer. Then repeat at 2.

The main program is already written for you in C++. You can write your own if you like. Here is the
definition of the Sb class:

class Sb {
    // the buffer is allocated in units of 2^chunk_nbits
    int chunk_nbits;
    // the size of the current buffer
    int buffer_size;
    // the number of characters in the current buffer
    int len;
    char *buffer;
    // call resize to resize the buffer to add more space.
    void resize(int additional_bytes_wanted);

public:
    Sb(void);
    void append(const char * str);
    void append(char c);
    char * toString(void);
    void clear(void);
    int length(void);
};

Note: buffer is not terminated with a null character (null-terminated). It is a char array, whose end is
kept track of by the len member. (For you C++ folks, it is not a C-string). The function toString
returns a copy of the contents of buffer as a C-string (a null-terminated string).

Requirements

• You must follow the ideas discussed in the class notes about a simple implementation of objects
at the assembly level. This consists of rules for
  ◦ naming of member functions, e.g., the name of the clear member function is Sb$clear$v. For
    argument types, we will adopt the rule that lower-case letters indicate value parameters
    (scalars, such as int or char) and upper-case letters indicate pointers. Thus the suffix letter
    for a char would be c, while the suffix letter for a char * would be C.
  ◦ the size and layout of objects. The first data member starts at the beginning of the object and
each successive data member is addressed sequentially. Thus in our class above, the
location of len is 8 + the start of the object.
  ◦ the calling interface, which adds a pointer to the current object as the first argument. (Thus
the clear member function actually has a single hidden argument, this.)
• You must treat buffer as a character buffer without a null-terminator and only rely on
null-termination of the result of copying buffer to a string using toString. (Hint: use the
strncpy function in util.s)
• You must honor the implied access restrictions in the class, even though there is no compiler to
enforce them. (Thus, the data members and the member function resize are private, and are
only accessible to member functions.)

Procedure

You have been given a MIPS assembly file sb.asm with the following pieces written

asmt03
Your job is to
• convert the main program to MIPS assembler
• write the remaining member functions using the hints in the code. You will have use for the `strcpy`, `strlen` and `malloc` functions in `util.s`
• get the whole thing working!

There is a C++ `a.out` file for the compiled C++ version of the program in the `asmt03` directory on hills. You can run it to test how it works. It accepts a single option `-d` that turns on debugging information. This will output the function calls it makes.

Details of the object members follows:
• `chunk_nbits, buffer, buffer_size` are all initialized by the constructor and `len` is set to 0. `len` is the current number of characters in the buffer. Remember, the null byte is not in the buffer.
• `resize` ensures there is at least `additional_bytes_wanted` bytes available in the buffer. If not, a new buffer is allocated whose minimum size is `len + additional_bytes_wanted` bytes long, the existing buffer is copied to the new buffer, and the existing buffer then replaced with the new buffer. `buffer_size` is, of course, set to the new buffer size. If there is `additional_bytes_wanted` bytes remaining in the current buffer, `resize` simply returns.
• the `append` functions append either a string or a single character to the current buffer, updating `len` appropriately. They call `resize` to ensure there is enough room in the current buffer to add the data.
• `toString` copies the contents of the current buffer to a newly allocated area, null-terminates the new data and returns the new data. The current contents of the buffer are unchanged.
• `clear` simply sets the number of characters in the buffer to zero.
• `length` simply returns the number of characters in the current buffer. (you may not need this function)