Padded String: Treating String as Sequence of Machine Words*

Pei-Chi Wu Feng-Jian Wang
Institute of Computer Science and Information Engineering
National Chiao Tung University
1001 Ta-Hsueh Road, Hsinchu
Taiwan, Republic of China
{pcwu, fjwang}@csie.nctu.edu.tw

ABSTRACT

A *string* is a sequence of characters. The operations such as copy and comparison on strings are usually performed character by character. This note presents a data type called *padded string*, a string type with faster operations. A padded string is a sequence of machine words. For 32-bit machines, four characters can be operated in one machine instruction. Operations on padded strings can then run faster than traditional strings such as char* in C language. An experiment sorting an array of strings shows speedup 24% using padded string.

Keywords: string, data structure, string sorting.

1. Motivation

A *string* is a sequence of characters. The operations such as copy and comparison on strings are usually performed character by character. They utilize only 1/4 computing power of 32-bit machines, since four characters (a machine word) can be operated in one instruction in these machines. Surprisingly, this issue is rarely addressed in the literature (cf. [3, 4] for classical data structures of string).

This note presents a data type of string, called *padded string*. A padded string is a sequence of machine words. There are two design rationale:

1) provide faster comparison operations.

There are two kinds of comparison operations:

- relation operations: <, >, <=, >=
- equality operations: ==, !=

Sorting an array of strings may be a daily work in some commercial applications. It needs O(*n* log *n*) comparison operations on strings. Replacing the traditional string type with a more efficient implementation can easily speedup these applications. For example, the expected speedup is about 300% in best case, if the new string type utilizes the full computing power of 32-bit machines.

2) provide nearly free conversion to traditional string type such as char* in C language.

Since string is a basic data type, many of its utilities are already provided by programming languages or operating systems. The C functions such as `printf()` take an argument of zero-

* This research was partly supported by National Science Council, Taiwan, R.O.C., under Contract No. NSC 82-0408-E009-280.

terminated array of characters. If a free conversion to traditional string type is provided, the new string type can then be processed by these utilities. A C++ conversion operator is defined for the class of new string type:

```
        operator char* ();
```

The conversion should involve only simple computation of the address of string.

2. Technical Issues

There are three issues in handling a string as a sequence of words:

1) Word Alignment: A string may start in an address that is not properly aligned for machine words. Some architectures need (integer) data to be aligned in word boundary. For string data, the allocation in memory hardly meets this requirement. A string may start in any address. Consider a string "word alignment", even if the first token "word" is properly aligned (e.g., address 0x200), the second token "alignment" is not (e.g., address. 0x205). A *preprocessing* of traditional string type (copy the contents to an aligned address) is needed to provide proper alignment.

2) Varied Length: A string may not completely fill a number of machine words. Consider a series of operations on a word representation of string. The length of a string may not be a multiple of 4 for a 32-bit machine, thus the operation on the last word needs special treatments or the operation may access some data outside the string. The former causes some overhead, while the latter causes incorrect result. A solution is to pad zeros at the end of a string. For example, the token "alignment":

"alignment" => ("alig", "nmen", "t\0\0\0").

3) Byte Order: The relation operations on strings should guarantee alphabetical order, i.e., maintaining the order like "A100" < "B000". Executing string comparison by comparing machine words has two kinds of results according to the byte order in a machine word: *big-endian* and *little-endian.*

	"A100"		"B000"
big-endian:	0x41313030	<	0x42303030
little-endian:	0x30303141	>	0x30303042

(in ASCII code, 'A'=0x41, 'B'=0x42, '0'=0x30, '1'=0x31)

Only the result by big-endian meets the alphabetical order. For equality operations, there is no problem in byte order.

3. Padded String Type

This section presents padded string in C++ [1] class. The presented string type meets the first two issues in Section 2, but does not attack the byte-order issue, which may be left to computer architects (e.g., [2]).

A *padded string* consists of an integer denoting its length and a sequence of machine words for the contents. A padded string is aligned in word boundary and is zero-terminated. For example,

"word" => (4, "word", "\0\0\0\0").

The code listed here does not consider the dynamic allocation of data. A padded string is given a predefined bounded space. The implementation stores length-1 (data member `len_1`) instead of length of string.

```
static int const MAX_N_BYTES = 256;
static int const MAX_N_WORDS=MAX_N_BYTES/4;

class Pad_String
{
public:
  Pad_String(const char *str, int len);
  ~Pad_String();
  operator char *() const { return a; }
  int length() const { return len_1 + 1; }

  friend int operator==(const Pad_String& ps1, const Pad_String& ps2);
  friend int operator!=(const Pad_String& ps1, const Pad_String& ps2);

  friend int strcmp(const Pad_String& ps1, const Pad_String& ps2);
  friend int operator<=(const Pad_String& ps1, const Pad_String& ps2)
       { return strcmp(ps1,ps2)<=0; }
  friend int operator>=(const Pad_String& ps1, const Pad_String& ps2)
       { return strcmp(ps1,ps2)>=0; }
  friend int operator<(const Pad_String& ps1, const Pad_String& ps2)
       { return strcmp(ps1,ps2)<0; }
  friend int operator>(const Pad_String& ps1, const Pad_String& ps2)
       { return strcmp(ps1,ps2)>0; }
private:
  int len_1;
  union {
    unsigned int  w[MAX_N_WORDS]; /* word */
    char a[MAX_N_BYTES]; /* alphabet: a[0:n-1], zero: a[n:]*/
  };
  // number of words - 1
  int nwords_1() const { return len_1>>2;}
};

Pad_String::Pad_String(const char* str, int len)
{
  int i, sz;

  len_1 = len-1;
  for(i=0; i<len; i++) // byte by byte
    a[i] = str[i];
  sz = (len+1 + 0x3) & ~0x3;
  for(i=len; i<sz; i++) // padding '\0'
    a[i] = '\0';
}

// return <0 for ps1<ps2, ==0 for ps1==ps2, >0 for ps1>ps2
int strcmp(const Pad_String& ps1, const Pad_String& ps2)
{
  int i, nwords_1, flag;

  nwords_1 = ps1.nwords_1(); // one's last word
  for(i=0; i<=nwords_1; i++) { // compare w[0:nwords_1]
    if (flag=ps1.w[i] - ps2.w[i])  // ps1.w[i] != ps2.w[i]
      return (flag);
  }
  return 0;
}
```

4. An Experimental Result

We have tested the padded string type in sorting (quick sort) a dictionary in UNIX system (/usr/dict/words). The dictionary contains about 25,000 words. It is a sequence of words partly sorted by the alphabetical order in ASCII code. (Words are listed in the order that ignores case and skip non-letter symbols). The following is an experimental result sorting the dictionary according to the alphabetical order. String class is an encapsulated version of char*. The compiler used is GNU C++ version 2.5.8 [2] with optimization flag "-O". All operations are inlined. The host machine is SPARC station ELC (a big-endian machine) with 28M bytes memory.

String	Padded String	Speedup
1.18 sec	0.95 sec	24%

5. Discussions

Common prefix in strings. The performance gains on padded strings depend on the set of input strings. For strings with common prefix, the speedup may be greater than the result in Section 4. Symbols automatically generated by programs usually have long common prefix to distinguish the use of symbols. For example, compilers usually add some prefix to program's symbols. C compilers add a prefix "_" to "i" and "j". Some preprocessor symbols use prefix "__", e.g., "__LINE__" and "__FILE__". Label numbers may be represented as "L0010" and "L0020". C++ compilers expand the templates with the name of template as prefix, e.g., "_qsort__FPP6Stringii" and "_qsort__FPP10Pad_Stringii".

Preprocessing. A padded string can be created by copying a traditional string to a word aligned memory and padding zeros. This overhead sometimes defeats the applications of padded strings. An example is symbol processing in compilers. A token from a scanner is usually a pointer to a character buffer. The address of token cannot be properly aligned. The time in preprocessing may dominate the time in equality operations on a dictionary of existing strings. Sorting a set of strings is a case that preprocessing is not significant. All the words read from a file should be allocated in a memory. There is no additional cost for allocating it in word boundary, since most memory allocators allocate objects in word boundary.

References

1. Ellis, M.A., and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
2. Free Software Foundation, *GNU C++ Compiler*, Version 2.5.8, 1994.
3. Hennessy, J., Patterson, D., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990, p.95.
4. Horowitz, E., Sahni, S., *Fundamentals of Data Structures in Pascal*, Computer Science Press, 1984, p.175.
5. Knuth, D. E., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, 1973, p.460.