```
/*
  WString.cpp - String library for Wiring & Arduino
  ...mostly rewritten by Paul Stoffregen...
  Copyright (c) 2009-10 Hernando Barragan.  All rights reserved.
  Copyright 2011, Paul Stoffregen, paul@pjrc.com

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
*/

#include "WString.h"

/*********************************************/
/*  Constructors                             */
/*********************************************/

String::String(const char *cstr)
{
  init();
  if (cstr) copy(cstr, strlen(cstr));
}

String::String(const String &value)
{
  init();
  *this = value;
}

String::String(const __FlashStringHelper *pstr)
{
  init();
  *this = pstr;
}

#ifdef __GXX_EXPERIMENTAL_CXX0X__
String::String(String &&rval)
{
  init();
  move(rval);
}
String::String(StringSumHelper &&rval)
{
  init();
  move(rval);
}
#endif

String::String(char c)
{
  init();
  char buf[2];
  buf[0] = c;
  buf[1] = 0;
  *this = buf;
}
```

```
String::String(unsigned char value, unsigned char base)
{
  init();
  char buf[1 + 8 * sizeof(unsigned char)];
  utoa(value, buf, base);
  *this = buf;
}

String::String(int value, unsigned char base)
{
  init();
  char buf[2 + 8 * sizeof(int)];
  itoa(value, buf, base);
  *this = buf;
}

String::String(unsigned int value, unsigned char base)
{
  init();
  char buf[1 + 8 * sizeof(unsigned int)];
  utoa(value, buf, base);
  *this = buf;
}

String::String(long value, unsigned char base)
{
  init();
  char buf[2 + 8 * sizeof(long)];
  ltoa(value, buf, base);
  *this = buf;
}

String::String(unsigned long value, unsigned char base)
{
  init();
  char buf[1 + 8 * sizeof(unsigned long)];
  ultoa(value, buf, base);
  *this = buf;
}

String::String(float value, unsigned char decimalPlaces)
{
  init();
  char buf[33];
  *this = dtostrf(value, (decimalPlaces + 2), decimalPlaces, buf);
}

String::String(double value, unsigned char decimalPlaces)
{
  init();
  char buf[33];
  *this = dtostrf(value, (decimalPlaces + 2), decimalPlaces, buf);
}

String::~String()
{
  free(buffer);
}

/*********************************************/
/*  Memory Management                        */
/*********************************************/

inline void String::init(void)
{
  buffer = NULL;
  capacity = 0;
  len = 0;
}

void String::invalidate(void)
{
  if (buffer) free(buffer);
  buffer = NULL;
  capacity = len = 0;
}

unsigned char String::reserve(unsigned int size)
{
  if (buffer && capacity >= size) return 1;
  if (changeBuffer(size)) {
    if (len == 0) buffer[0] = 0;
    return 1;
  }
  return 0;
}

unsigned char String::changeBuffer(unsigned int maxStrLen)
{
  char *newbuffer = (char *)realloc(buffer, maxStrLen + 1);
  if (newbuffer) {
    buffer = newbuffer;
    capacity = maxStrLen;
    return 1;
  }
  return 0;
}

/*********************************************/
/*  Copy and Move                            */
/*********************************************/

String & String::copy(const char *cstr, unsigned int length)
{
  if (!reserve(length)) {
    invalidate();
    return *this;
  }
  len = length;
  strcpy(buffer, cstr);
  return *this;
}

String & String::copy(const __FlashStringHelper *pstr, unsigned int length)
{
  if (!reserve(length)) {
    invalidate();
    return *this;
  }
  len = length;
  strcpy_P(buffer, (PGM_P)pstr);
  return *this;
}

#ifdef __GXX_EXPERIMENTAL_CXX0X__
void String::move(String &rhs)
{
  if (buffer) {
    if (capacity >= rhs.len) {
      strcpy(buffer, rhs.buffer);
      len = rhs.len;
      rhs.len = 0;
      return;
    } else {
      free(buffer);
    }
  }
  buffer = rhs.buffer;
  capacity = rhs.capacity;
  len = rhs.len;
  rhs.buffer = NULL;
  rhs.capacity = 0;
  rhs.len = 0;
}
#endif

String & String::operator = (const String &rhs)
{
  if (this == &rhs) return *this;

  if (rhs.buffer) copy(rhs.buffer, rhs.len);
  else invalidate();

  return *this;
}

#ifdef __GXX_EXPERIMENTAL_CXX0X__
String & String::operator = (String &&rval)
{
  if (this != &rval) move(rval);
  return *this;
}

String & String::operator = (StringSumHelper &&rval)
{
  if (this != &rval) move(rval);
  return *this;
}
#endif

String & String::operator = (const char *cstr)
{
  if (cstr) copy(cstr, strlen(cstr));
  else invalidate();

  return *this;
}

String & String::operator = (const __FlashStringHelper *pstr)
{
  if (pstr) copy(pstr, strlen_P((PGM_P)pstr));
  else invalidate();

  return *this;
}

/*********************************************/
/*  concat                                   */
/*********************************************/

unsigned char String::concat(const String &s)
{
  return concat(s.buffer, s.len);
}

unsigned char String::concat(const char *cstr, unsigned int length)
{
  unsigned int newlen = len + length;
  if (!cstr) return 0;
  if (length == 0) return 1;
  if (!reserve(newlen)) return 0;
  strcpy(buffer + len, cstr);
  len = newlen;
  return 1;
}

unsigned char String::concat(const char *cstr)
{
  if (!cstr) return 0;
  return concat(cstr, strlen(cstr));
}

unsigned char String::concat(char c)
{
  char buf[2];
  buf[0] = c;
  buf[1] = 0;
  return concat(buf, 1);
}

unsigned char String::concat(unsigned char num)
{
  char buf[1 + 3 * sizeof(unsigned char)];
  itoa(num, buf, 10);
  return concat(buf, strlen(buf));
}

unsigned char String::concat(int num)
{
  char buf[2 + 3 * sizeof(int)];
  itoa(num, buf, 10);
  return concat(buf, strlen(buf));
}

unsigned char String::concat(unsigned int num)
{
  char buf[1 + 3 * sizeof(unsigned int)];
  utoa(num, buf, 10);
  return concat(buf, strlen(buf));
}

unsigned char String::concat(long num)
{
  char buf[2 + 3 * sizeof(long)];
  ltoa(num, buf, 10);
  return concat(buf, strlen(buf));
}

unsigned char String::concat(unsigned long num)
{
  char buf[1 + 3 * sizeof(unsigned long)];
  ultoa(num, buf, 10);
  return concat(buf, strlen(buf));
}

unsigned char String::concat(float num)
{
  char buf[20];
  char* string = dtostrf(num, 4, 2, buf);
  return concat(string, strlen(string));
}

unsigned char String::concat(double num)
{
  char buf[20];
  char* string = dtostrf(num, 4, 2, buf);
  return concat(string, strlen(string));
}
```

```
unsigned char String::concat(const __FlashStringHelper * str)
{
  if (!str) return 0;
  int length = strlen_P((const char *) str);
  if (length == 0) return 1;
  unsigned int newlen = len + length;
  if (!reserve(newlen)) return 0;
  strcpy_P(buffer + len, (const char *) str);
  len = newlen;
  return 1;
}

/*********************************************/
/*  Concatenate                              */
/*********************************************/

StringSumHelper & operator + (const StringSumHelper &lhs, const String &rhs)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(rhs.buffer, rhs.len)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, const char *cstr)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!cstr || !a.concat(cstr, strlen(cstr))) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, char c)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(c)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, unsigned char num)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(num)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, int num)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(num)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, unsigned int num)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(num)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, long num)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(num)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, unsigned long num)
```

```
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(num)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, float num)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(num)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, double num)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(num)) a.invalidate();
  return a;
}

StringSumHelper & operator + (const StringSumHelper &lhs, const __FlashStringHelper
*rhs)
{
  StringSumHelper &a = const_cast<StringSumHelper&>(lhs);
  if (!a.concat(rhs)) a.invalidate();
  return a;
}

/*********************************************/
/*  Comparison                               */
/*********************************************/

int String::compareTo(const String &s) const
{
  if (!buffer || !s.buffer) {
    if (s.buffer && s.len > 0) return 0 - *(unsigned char *)s.buffer;
    if (buffer && len > 0) return *(unsigned char *)buffer;
    return 0;
  }
  return strcmp(buffer, s.buffer);
}

unsigned char String::equals(const String &s2) const
{
  return (len == s2.len && compareTo(s2) == 0);
}

unsigned char String::equals(const char *cstr) const
{
  if (len == 0) return (cstr == NULL || *cstr == 0);
  if (cstr == NULL) return buffer[0] == 0;
  return strcmp(buffer, cstr) == 0;
}

unsigned char String::operator<(const String &rhs) const
{
  return compareTo(rhs) < 0;
}

unsigned char String::operator>(const String &rhs) const
{
  return compareTo(rhs) > 0;
}

unsigned char String::operator<=(const String &rhs) const
{
  return compareTo(rhs) <= 0;
}

unsigned char String::operator>=(const String &rhs) const
{
  return compareTo(rhs) >= 0;
}

unsigned char String::equalsIgnoreCase( const String &s2 ) const
{
  if (this == &s2) return 1;
  if (len != s2.len) return 0;
  if (len == 0) return 1;
  const char *p1 = buffer;
  const char *p2 = s2.buffer;
  while (*p1) {
    if (tolower(*p1++) != tolower(*p2++)) return 0;
  }
  return 1;
}

unsigned char String::startsWith( const String &s2 ) const
{
  if (len < s2.len) return 0;
  return startsWith(s2, 0);
}

unsigned char String::startsWith( const String &s2, unsigned int offset ) const
{
  if (offset > len - s2.len || !buffer || !s2.buffer) return 0;
  return strncmp( &buffer[offset], s2.buffer, s2.len ) == 0;
}

unsigned char String::endsWith( const String &s2 ) const
{
  if ( len < s2.len || !buffer || !s2.buffer) return 0;
  return strcmp(&buffer[len - s2.len], s2.buffer) == 0;
}

/*********************************************/
/*  Character Access                         */
/*********************************************/

char String::charAt(unsigned int loc) const
{
  return operator[](loc);
}

void String::setCharAt(unsigned int loc, char c)
{
  if (loc < len) buffer[loc] = c;
}

char & String::operator[](unsigned int index)
{
  static char dummy_writable_char;
  if (index >= len || !buffer) {
    dummy_writable_char = 0;
    return dummy_writable_char;
  }
  return buffer[index];
}

char String::operator[]( unsigned int index ) const
{
  if (index >= len || !buffer) return 0;
  return buffer[index];
}
```

```
void String::getBytes(unsigned char *buf, unsigned int bufsize, unsigned int index)
const
{
  if (!bufsize || !buf) return;
  if (index >= len) {
    buf[0] = 0;
    return;
  }
  unsigned int n = bufsize - 1;
  if (n > len - index) n = len - index;
  strncpy((char *)buf, buffer + index, n);
  buf[n] = 0;
}

/*********************************************/
/*  Search                                   */
/*********************************************/

int String::indexOf(char c) const
{
  return indexOf(c, 0);
}

int String::indexOf( char ch, unsigned int fromIndex ) const
{
  if (fromIndex >= len) return -1;
  const char* temp = strchr(buffer + fromIndex, ch);
  if (temp == NULL) return -1;
  return temp - buffer;
}

int String::indexOf(const String &s2) const
{
  return indexOf(s2, 0);
}

int String::indexOf(const String &s2, unsigned int fromIndex) const
{
  if (fromIndex >= len) return -1;
  const char *found = strstr(buffer + fromIndex, s2.buffer);
  if (found == NULL) return -1;
  return found - buffer;
}

int String::lastIndexOf( char theChar ) const
{
  return lastIndexOf(theChar, len - 1);
}

int String::lastIndexOf(char ch, unsigned int fromIndex) const
{
  if (fromIndex >= len) return -1;
  char tempchar = buffer[fromIndex + 1];
  buffer[fromIndex + 1] = '\0';
  char* temp = strrchr( buffer, ch );
  buffer[fromIndex + 1] = tempchar;
  if (temp == NULL) return -1;
  return temp - buffer;
}

int String::lastIndexOf(const String &s2) const
{
  return lastIndexOf(s2, len - s2.len);
}

int String::lastIndexOf(const String &s2, unsigned int fromIndex) const
```

```
{
    if (s2.len == 0 || len == 0 || s2.len > len) return -1;
  if (fromIndex >= len) fromIndex = len - 1;
  int found = -1;
  for (char *p = buffer; p <= buffer + fromIndex; p++) {
    p = strstr(p, s2.buffer);
    if (!p) break;
    if ((unsigned int)(p - buffer) <= fromIndex) found = p - buffer;
  }
  return found;
}

String String::substring(unsigned int left, unsigned int right) const
{
  if (left > right) {
    unsigned int temp = right;
    right = left;
    left = temp;
  }
  String out;
  if (left >= len) return out;
  if (right > len) right = len;
  char temp = buffer[right];  // save the replaced character
  buffer[right] = '\0';
  out = buffer + left;  // pointer arithmetic
  buffer[right] = temp;  //restore character
  return out;
}

/*********************************************/
/*  Modification                             */
/*********************************************/

void String::replace(char find, char replace)
{
  if (!buffer) return;
  for (char *p = buffer; *p; p++) {
    if (*p == find) *p = replace;
  }
}

void String::replace(const String& find, const String& replace)
{
  if (len == 0 || find.len == 0) return;
  int diff = replace.len - find.len;
  char *readFrom = buffer;
  char *foundAt;
  if (diff == 0) {
    while ((foundAt = strstr(readFrom, find.buffer)) != NULL) {
      memcpy(foundAt, replace.buffer, replace.len);
      readFrom = foundAt + replace.len;
    }
  } else if (diff < 0) {
    char *writeTo = buffer;
    while ((foundAt = strstr(readFrom, find.buffer)) != NULL) {
      unsigned int n = foundAt - readFrom;
      memcpy(writeTo, readFrom, n);
      writeTo += n;
      memcpy(writeTo, replace.buffer, replace.len);
      writeTo += replace.len;
      readFrom = foundAt + find.len;
      len += diff;
    }
    strcpy(writeTo, readFrom);
  } else {
    unsigned int size = len; // compute size needed for result
    while ((foundAt = strstr(readFrom, find.buffer)) != NULL) {
      readFrom = foundAt + find.len;
      size += diff;
    }
    if (size == len) return;
    if (size > capacity && !changeBuffer(size)) return; // XXX: tell user!
    int index = len - 1;
    while (index >= 0 && (index = lastIndexOf(find, index)) >= 0) {
      readFrom = buffer + index + find.len;
      memmove(readFrom + diff, readFrom, len - (readFrom - buffer));
      len += diff;
      buffer[len] = 0;
      memcpy(buffer + index, replace.buffer, replace.len);
      index--;
    }
  }
}

void String::remove(unsigned int index){
  // Pass the biggest integer as the count. The remove method
  // below will take care of truncating it at the end of the
  // string.
  remove(index, (unsigned int)-1);
}

void String::remove(unsigned int index, unsigned int count){
  if (index >= len) { return; }
  if (count <= 0) { return; }
  if (count > len - index) { count = len - index; }
  char *writeTo = buffer + index;
  len = len - count;
  strncpy(writeTo, buffer + index + count,len - index);
  buffer[len] = 0;
}

void String::toLowerCase(void)
{
  if (!buffer) return;
  for (char *p = buffer; *p; p++) {
    *p = tolower(*p);
  }
}

void String::toUpperCase(void)
{
  if (!buffer) return;
  for (char *p = buffer; *p; p++) {
    *p = toupper(*p);
  }
}

void String::trim(void)
{
  if (!buffer || len == 0) return;
  char *begin = buffer;
  while (isspace(*begin)) begin++;
  char *end = buffer + len - 1;
  while (isspace(*end) && end >= begin) end--;
  len = end + 1 - begin;
  if (begin > buffer) memcpy(buffer, begin, len);
  buffer[len] = 0;
}

/*********************************************/
/*  Parsing / Conversion                     */
/*********************************************/

long String::toInt(void) const
```

```
{
  if (buffer) return atol(buffer);
  return 0;
}

float String::toFloat(void) const
{
  if (buffer) return float(atof(buffer));
  return 0;
}
```