AUTODESK Instructables

Taming Arduino Strings -- How to Avoid Memory Issues

By drmpf in CircuitsArduino

Introduction: Taming Arduino Strings -- How to Avoid Memory Issues

Update 9th July 2021 - Added link to fixed versions of Arduino Strings files. Normally not needed.

Quick Start

For small sketches with a few Strings, just use them as convenient.

For small sketches with a few Strings, just use them. The good news is if you are using an UNO or Mega2560, then using Strings is extremely safe and won't crash your board*, even if you run out-of-memory. If you run out-of-memory, you will just not get all the text in the Strings that you expect. The program will continue to run. See What happens when UNO runs out of heap memory below (Step 8). Other boards will have more memory available and so you are unlikely to have memory problems using a few Strings.

In any case carefully read the Arduino documentation on the String Addition Operator to avoid common coding errors when using the String + operator.

*There are a few obscure bugs in the String library that can crash your sketch otherwise using Strings on AVR boards will not crash/reboot your board. This version of WString.cpp and this version of WString.h fixes those bugs. Just copy it over the WString.cpp and WString.h in your arduino **...\hardware\arduino\avr\cores** directory. A future version of Arduino will include fixes for these bugs.

For large sketches OR lots of Strings OR if you think you are having memory problems

If you have a large sketch, with lots of Strings OR you are concerned about possible memory problems, then following the guidelines below will make your sketch reliable and safe.

Guidelines for Using Arduino Strings

1) Declare long lived Strings as globals and reserve( ) space in setup(), starting with the smallest to the largest. Check the return from the last largest reserve( ) to see that you have enough memory for all the Strings

2) If you have created Strings in the loop() method, they are long lived, move them to Globals and repeat step 1.

3) Pass all Strings arguments to methods, as const String& . For results pass a String& result, that the method can update with the result. i.e. void strProcessing(const String &input1, const String &input2, ,, String &result) {..} See Using String& for arguments (Step 11)

4) Set the IDE File → Preferences Compiler Warning to ALL and watch for ***warning: passing 'const String' as 'this' argument discards qualifiers [-fpermissive]*** messages in the compile window. This says you are trying to modify a const String& arg.

5) Do string processing in small compact methods. In these methods use local Strings in preference to local char[]. These methods will completely recover all the heap and stack used by their local Strings and other variables on exit. See String versus char[] (Step 12)

6) Do not use the String + operator at all and avoid using the String(...) constructors as these create short lived temporary Strings. Use = and += operators or concat( ), as in result += “str”; result += 'c'; result += number; etc. See Minimizing Memory Usage (Step 11)

7) To monitor for Low Memory and fragmentation, add a StringReserveCheck to at least the last largest String reserve( ). Add StringReserveCheck to other Strings as necessary. See Using StringReserveCheck (Step 6) (download and install StringReserveCheck.zip)

8) If your Arduino program uses the Web e.g. ESP32 / EPS8266 webserver/httpClient, then the underlying web support libraries already use lots of Strings. If you project is suppose to run for a long time, add a periodic automatic reboot. See ESP32/ESP8266 Adding Periodic Automatic Reboots (Step 9)

9) Don't call String methods from within an interrupt routine. String uses malloc/realloc to get memory from the heap to store the chars. malloc/realloc on the UNO, Mega2560 etc (AVR processors and others) are not designed to called from the main loop and then interrupted and called again (a reentrant call). The result is a crash due to malloc/realloc loosing track of what is happening unless you add extra 'guard' code to prevent it.

Note: The links jump to the on-line version of this tutorial, the Step numbers in this instructable are shown in brackets

Supplies

Any Arduino Board

Step 1: Introduction

Arduino Strings have been getting bad press due to the extra memory they use to make copies and the memory fragmentation they can cause. These can eventually consume all the available memory and cause the micro to miss-behave and reboot. This tutorial will show you how to avoid these two memory issues when using Arduino Strings. Often the suggested solution is to replace Arduino Strings with low-level c-string methods and char[] manipulations. C-string methods and char[] manipulations are very prone to coding errors (see Why you should not use c-cstring methods below) and should not be used. The C++ String library, which Arduino Strings is based on, was created to avoid the systemic coding problems caused by c-strings and char[] manipulations. Arduino Strings, or alternatively the SafeString library, should always be used in preference to low level c-string methods or char[] manipulations.

The good news is if you are using an UNO or Mega2560, then using Strings is extremely safe, even if you run out-of-memory. If you run out-of-memory, you will just not get all the text in the Strings that you expect. The program will continue to run. See What happens when UNO runs out of heap memory below

If a Strings need more space for its text it will try and allocate more memory. If you want to keep complete control over your memory usage OR you are working with a library that returns a char* OR you want a richer set of readers, parsers and text functions OR you want detailed debugging/error messages, then you can install the SafeString library from the Arduino library manager. See the SafeString alternative below.

Step 2: Tutorial Outline

- Memory Fragmentation – Not the problem you were led to believe
- Using StringReserveCheck to find and remove the holes
- How Strings Handle Out-Of-Memory
  What happens when UNO or Mega2560 run out of heap memory
  Out-Of-Memory on ESP32 and ESP8266 – Add Periodic Automatic Reboots
- How to control String Memory Usage
  Using Strings in methods
  Temporary Strings
  Avoid creating Strings in the loop()
  Use String& for arguments and don't return String results when writing methods
  Don't use String(..) unless there is no alternative
  Do not use the + operator
- String versus char[], two sides of the same coin
- Why you should not use c-string methods or char[] manipulations
- Errors in Arduino's String
- The SafeString Alternative

Also see Arduino For Beginners – Next Steps

Taming Arduino Strings (this one)
How to write Timers and Delays in Arduino
SafeString Processing for Beginners
Simple Arduino Libraries for Beginners
Simple Multi-tasking in Arduino
Arduino Serial I/O for the Real World

Step 3: The Two String Memory Issues – Fragmentation and Extra Memory Used

Memory Fragmentation is not the problem you were led to believe. Using String reserve( ) and StringReserveCheck and following the guidelines above eliminates memory fragemenation. Extra Memory usage is avoided by passing String arguments as String& and by avoiding the creation of temporary Strings.

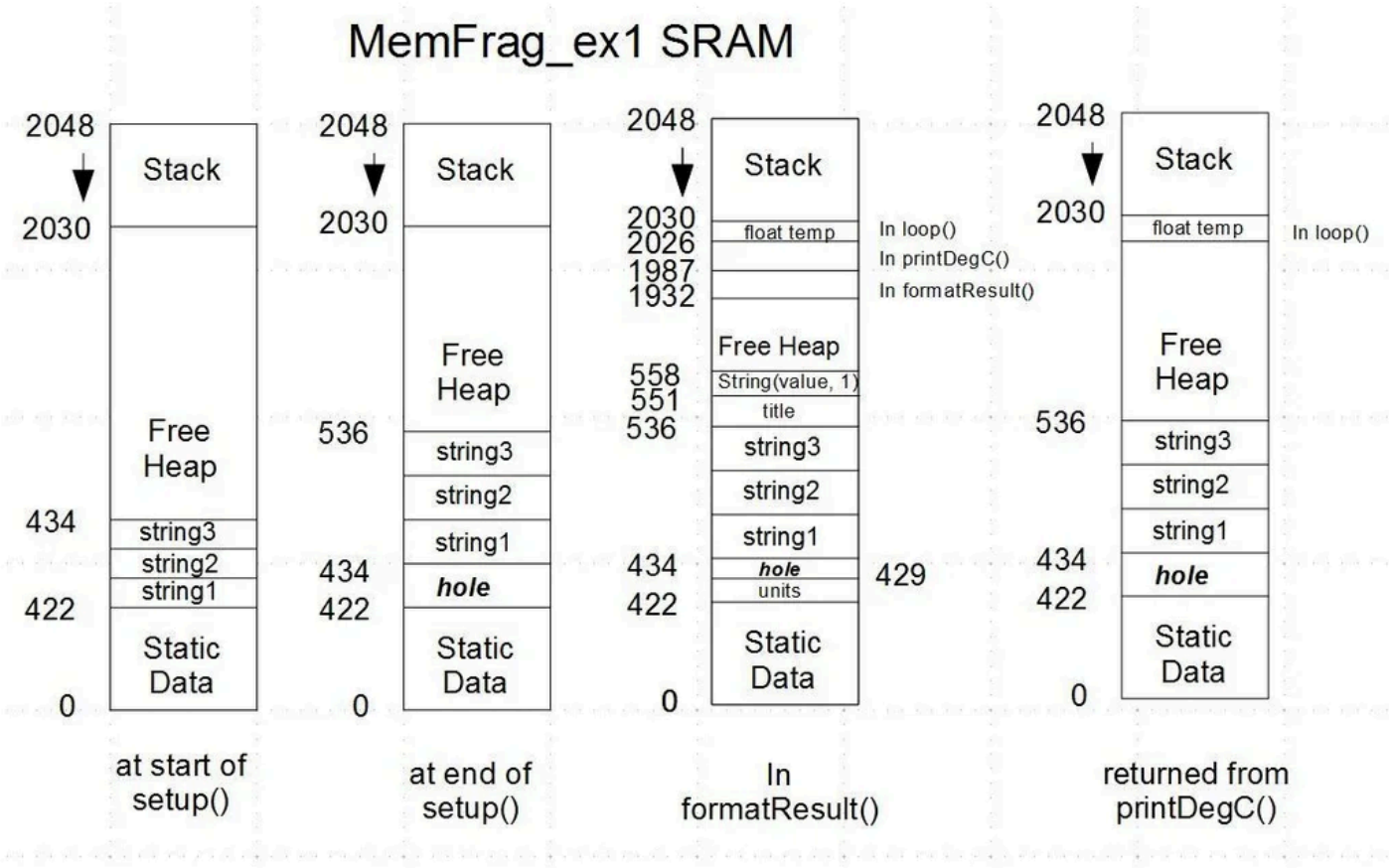
Memory Fragmentation – Not the problem you were led to believe

Arduino Strings use heap memory to store their text. Fragmentation occurs when a long lived String is allocated memory above a short lived String. When the short lived String is finished with and is discarded, it leaves a hole. The micro-processor memory manager keeps track of these holes in its heap FreeList and tries to reuse them, but if the program enlarges a lot large long lived Strings interleaved with creating small short lived Strings, the memory ends up with a lot of holes using up the heap and the next time heap memory is needed for a String or stack memory is needed for a function call, the program fails. However as you will see below, using Strings on UNO and Mega2560 this is not a problem even if you run out of memory. Other modern micro-processors, such as nRF52, ESP8266 and ESP32, have much more memory and if you follow steps 1 to 7 above you will not have any problems.

This section will cover the following topics:-
Examine the heap allocation for a small program using Strings.
Show how heap 'holes' can be created and why that is usually not a problem.
Using StringReserveCheck to find and remove the holes.
What happens when UNO and Mega2560 run out of heap memory. (Nothing much actually)
Out-Of-Memory on ESP32 and ESP8266. (Most likely not your fault, if you have a web project)

All the examples below are run on an UNO unless otherwise stated.

Step 4: Examine the Heap Allocation for a Small Program Using Strings.



The is the example sketch, MemFrag_ex1.ino

```
String string1;
String string2;
String string3;

void setup() {
  Serial.begin(9600);
  for (int i = 10; i > 0; i--) {
    Serial.print(i); Serial.print(' ');
    delay(500);
  }
  Serial.println();
  Serial.println(F("Memory Non-Fragementation Example 1"));
  string1.reserve(32);
  string2.reserve(32);
  if (!string3.reserve(32)) { // check the last largest reserve
    while (1) { // stop here and print repeating msg
      Serial.println(F("Strings out-of-memory"));
      delay(3000); // repeat msg every 3 sec
    }
  }
}

void printDegC(float value, String& result) {
  String title = F("Temperature ");
  String units = F("degC");
  formatResult(title, units, value, result);
}
void formatResult(const String& title, const String& units, float value, String& result) {
  result = title;
  result += String(value, 1); //temp to 1 decimal only
  result += units;
}

void loop() {
  float temp = 27.35;
  printDegC(temp, string2);
  Serial.println(string2);
  Serial.println(F(" -- loop() returns --")); Serial.println();
}
```

The output on UNO is

```
Memory Non-Fragementation Example 1
Temperature 27.4degC
 -- loop() returns --
```

```
Temperature 27.4degC
 -- loop() returns -
```

The UNO memory SRAM for MemFrag_ex1 looks like above (pdf version)

Points to NOTE:
a) When string1, string2 and string3 have extra memory reserved in setup(), they are re-allocated higher up on the heap and leave behind a small 12byte hole.

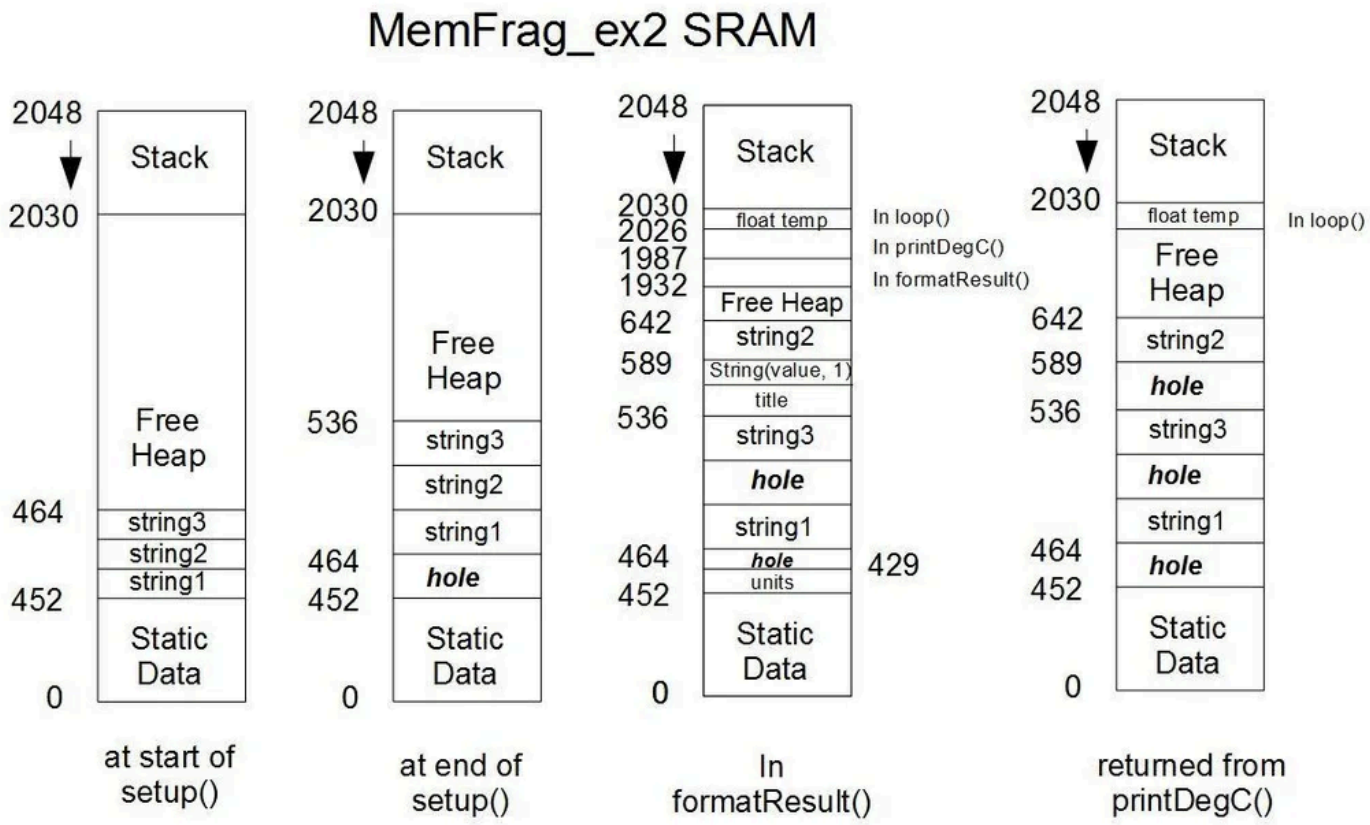
b) When the execution is in formatResult(), some of the 12byte hole has been reused by the units String created in printDegC(). The title string was too large and was allocated higher in the heap, as was the String created by String(value,1).

c) The method calls and their local variables cause the more stack to be used. (i.e. the end of stack moves down)

d) Very Important: When the printDegC() method call returns to the loop(), ALL the stack and String heap allocations made in the method calls are completely recovered. The only 'hole' left is the initial 12byte hole from the reserve()s.

Summary: String usage is completely stable if long lived Strings are pre-reserved, the result String is passed by reference &, and any other Strings are created in methods so their memory is completely recovered when the method returns.

Step 5: Show How Heap 'holes' Can Be Created and Why That Is Usually Not a Problem.



The next example MemFrag_ex2.ino
It is the same as MemFrag_ex1.ino except that **printDegC()** now has a longer title, greater then 32 chars

```
void printDegC(float value, String& result) {
  String title = F("This is the temperature in the boiler room ");
  String units = F("degC");
  formatResult(title, units, value, result);
}
```

The output on UNO is

```
Memory Non-Fragementation Example 2
This is the temperature in the boiler room 27.4degC
 -- loop() returns --

This is the temperature in the boiler room 27.4degC
 -- loop() returns -
```

The UNO memory SRAM for MegFrag_ex2 looks like above (pdf version)

Points to NOTE:
a) string2's reserve is no longer large enough to hold result with the longer title.

b) When the execution is in formatResult(), string2 is re-allocate with more space, to the top of the heap some, to accommodate the longer title.

c) The method calls and their local variables cause the more stack to be used. (i.e. the end of stack moves down), but 2 additional holes are left due to string2's re-allocation.

d) **Very Important:** The sketch automatically handled the larger title and continued to run and is completely stable. Although there are additional holes, that memory is available for other Strings to use, if they fit within the hole. If you were using low level c-string methods like strcat, strcpy and char[], the program would have crashed because title exceed the alloted char[] space.

Summary: If there is sufficient memory, the String sketch continues to run stably even if the initial reserve is not large enough. On the other hand, a sketch using char[] and c-string methods would have crashed. As we will see below, on UNO and Mega2560, the sketch will continue to run even if it is out-of-memory. While on other micro-processors, the larger available SRAM makes it much less likely out-of-memory will occur from one re-allocation.

Step 6: Use StringReserveCheck to Find and Remove the Holes

To detect when you have not reserved enough space, you can use the **StringReserveCheck** class. Download the StringReserveCheck.zip file and install it using the ***IDE Sketch → Include Library → Add .ZIP library...***

Then you can add StringReserveCheck objects to track each of your long lived Strings. StringReserveCheck is used as follows:-

Allocate one StringReserveCheck object immediately after each String you want to check. e.g. for global Strings

```
#include "StringReserveCheck.h"
String string1;
StringReserveCheck string1Check;<br></stringreservecheck.h>
```

This ensures both the String and StringReserveCheck object had the same scope and life cycle.

In setup(), immediately after you reserve() space for the String, call **init( )** on its StringReserveCheck object to capture the initial state. **init( )** also lets you specify where to write the checkReserve messages. e.g.

```
void setup() {
  . . .
  string1.reserve(32);
  string1Check.init(string1); // checkReserve( ) will not print any msgs, you need to check its return value
  string2.reserve(32);
  string2Check.init(string2, Serial); // init( ) will print a msg if memory is low and checkReserve( ) will print a msg
  if (!string3.reserve(32)) { // check the last largest reserve
    while (1) { // stop here and print repeating msg
      Serial.println(F("Strings out-of-memory"));
      delay(3000); // repeat msg every 3 sec
    }
  }
  if (!string3Check.init(string3, Serial)) {    // check return for low memory
    Serial.println(F("Memory Low after reserves()"));
  }
  . . .
}
```

If you are running low on memory the **stringCheck.init(string, Serial);** will print a warning message and return false. **stringCheck.init(string);** (*without the Serial arg*) won't print any messages so you need to check the return value.

Then when you want to check the reserve you use **stringCheck.checkReserve()**. e.g.

```
void loop() {
  float temp = 27.35;
  printDegC(temp, string2);
  Serial.println(string2);
  string1Check.checkReserve(); // init(string1,Serial); was specified so msg printed
  if (!string2Check.checkReserve()) { // check return,  init(string2,Serial); was specified so msg also printed
    Serial.print(F("string2 reserve too small. Current length():")); Serial.println(string2.length());
  }
  if (!string3Check.checkReserve()) { // init(string3); // no output specified, check return
    Serial.print(F("string3 reserve too small. Current length():")); Serial.println(string3.length());
  }
  Serial.println();
}
```

MemFrag_ex3.ino has these StringReserveCheck's added. The output of MemFrag_ex3.ino is

```
Memory Fragementation Example 3
This is the temperature in the boiler room 27.4degC
!! reserve() NOT large enough for String 'This is th...' Current len:51
string2 reserve too small. Current length():51
reserve() large enough for String 'string3 in...' Current len:21
```

This output clearly shows that string2's reserve needs to be increased and the current length gives you a hit as to what the new reserve should be. string3's reserve is OK. The message is printed because Serial was set as the output in the **string3Check.init(string3,Serial);** statement. No message is printed for string1 because **string1Check.init(string1);** did not specify where to send msgs.

Summary: Use the **StringReserveCheck** class to check the initial reserve is large enough for the application. If **checkReserve()** returns true there is no fragmentation due to re-allocation of this String.

Step 7: How Strings Handle Out-Of-Memory

For String str = String(...), and for the + operator, if there is not enough heap memory, then the resulting string will be empty. **Note carefully** in this case c_str() method will return a NULL pointer.

For all other methods, e.g concat(), replace() += etc, if there is not enough heap memory, then the existing String is left unchanged, so an indication of Out-Of-Memory is missing text from a String.

Checking the return of **reserve()** will also indicate Out-Of-Memory if it is false.

Summary: Running out-of-memory is not a problem for Strings. They will just be missing some of the text, but that will not cause the program crash.

Step 8: What Happens When UNO or Mega2560 Run Out of Heap Memory

Out-Of-Memory is fatal when the stack collides with the heap. As the SRAM diagrams above show, with each method call the stack uses more space for the method return, the method arguments and any method local variables. The stack does not check where the heap is and just takes the space it needs. The result is, when memory runs out, the stack writes over existing heap objects leading to memory corruption and program crashes.

The Arduino UNO has only 2048 bytes of SRAM and so it is easy to run out-of-memory on that board. Just using “text... “ instead of F(“text... “) can do it. However when you use Strings on the UNO or Mega2560 they are very robust in the face of heap out-of-memory due to the build in **__malloc_margin** of 128 bytes. There is always at least that amount of space for the stack to use to keep the program running.

In MemFrag_ex4.ino, the reserve sizes of string1 and string3 have been carefully adjusted for UNO so that almost all the memory is used. Then when the sketch tries to re-allocate string2 from its reserve of 32 to a larger value to handle the longer title it runs out-of-memory.

```
void setup() {
  . . .
  Serial.println(F("Memory Fragementation Example 4 - Out Of Memory"));
  string2.reserve(32); // do reserve in order smallest first string2
  string2Check.init(string2, Serial); // init( ) will print a msg if memory is low and checkReserve( ) will print a msg
  string1.reserve(600); // next largest string1
  string1Check.init(string1); // checkReserve( ) will not print any msgs, you need to check its return value
  if (!string3.reserve(930)) { // do largest last and check its return
    while (1) { // stop here and print repeating msg
      Serial.println(F("Strings out-of-memory"));
      delay(3000); // repeat msg every 3 sec
    }
  }
  if (!string3Check.init(string3, Serial)) {    // check return for low memory
    Serial.println(F("Memory Low after reserves()"));
  }
  . . .
}
```

Then when the MemFrag_ex4.ino sketch is run the output is

```
!! Low memory available, stability problems may occur.
Memory Low after reserves()
Memory Fragementation Example 4 - Out Of Memory
27.4degC
!! reserve() NOT large enough for String '27.4degC' Current len:8
string2 reserve too small. Current length():8
reserve() large enough for String 'string3 in...' Current len:21

27.4degC
!! reserve() NOT large enough for String '27.4degC' Current len:8
string2 reserve too small. Current length():8
reserve() large enough for String 'string3 in...' Current len:21
```

Even though the UNO ran out-of-memory, and could not make string2 large enough to add the title, the sketch just kept running stably and did not crash.

Where as the output should have been
This is the temperature in the boiler room 27.4degC
it is now missing the title.
27.4degC

Points to NOTE:
1) The sketch keeps running because on the UNO and Mega2560, the heap memory allocation always keeps a **__malloc_margin** of 128 bytes so there is always at least that amount of space for the stack to use to keep the program running. If you have a lot of nested method calls with a lot of local variables you can get the sketch to crash, but that is extremely very rare.

2) It is safer to use local Strings in methods then to use char[], because if the memory is low, the local Strings memory allocation just fails and always leaves at least 128 bytes free. Where as local char[] stack allocations never fail, they just overwrite the heap and cause memory corruption.

Summary: UNO and Mega2560 boards are extremely robust in the face of out-of-memory when you use Strings. The sketch just keep running but the Strings will be missing some of the text.

Step 9: Handling Out-Of-Memory on ESP32 and ESP8266 – Adding Periodic Automatic Reboots

The ESP32 and ESP8266 have much more SRAM available so provided
you follow the guidelines for using Arduino Strings you are unlikely to have any memory problem due to your code. However the web libraries of these boards make extensive use of Strings so if you are coding a web project you may eventually run out-of-memory memory due to problems in the underlying web libraries. In that case the board will either lock up or reboot. Removing Strings from your code won't change that. The best you can do it to arrange for an automatic reboot of the board on a periodic basis to clear out the memory. Both the ESP32 and EPS8266 have a watch dog timer, so enabling the watch dog timer and putting the code into tight loop will cause a reboot. e.g.

```
// millisDelay included in SafeString library, from Arduino library manager or
// download zip from https://www.forward.com.au/pfod/ArduinoProgramming/SafeString/index.html
#include <millisDelay.h>

millisDelay rebootDelay;
unsigned long REBOOT_DELAY_MS = 24ul * 60 * 60 * 1000; // 1day in mS

void setup() {
  // . . .
  rebootDelay.start(REBOOT_DELAY_MS); // start reboot timer
#if defined(ARDUINO_ARCH_ESP32)
  enableLoopWDT(); // default appears to be 5sec
#elif defined(ARDUINO_ARCH_ESP8266)
  ESP.wdtEnable(5000); // arg ignored :-( default appears to be 3sec
#else
#error Only ESP2866 and ESP32 reboot supported
#endif
}

void loop() {
  if (rebootDelay.justFinished()) {
    while (1) {} // force watch dog timer reboot
  }
  // . . . rest of loop code
}
```

Summary: ESP32 and ESP8266 use Strings extensively in their web support libraries. So there is no advantage to not using Strings in your code, if you follow the guidelines. If you what to run the project for a long time, build in an periodic automatic reboot.

Step 10: How to Control String Memory Usage

Using Strings in methods

As noted above in the guidelines long lived Strings should be used with **reserve( )** to prevent fragmentation. Usually these long lived Strings are globals. If you have any Strings in the **loop()** method, then they exist while all your other code is running. So they are also 'long lived', so move them out to be globals and reserve space for them in the **setup()**.

Sometimes you will have relatively long lived Strings in a method. An example is a field String that is used repeatedly for parsing CSV (comma separated values) or GPS data. In that case you should reserve sufficient space at the top of the parse method to prevent local fragmentation. All the fragmentation will be cleaned up when the parsing method returns, but in the mean time it is using extra memory which may be in short supply. Also see Minimizing String Memory usage below. E.g.

```
bool parse(String& lineOfData) {
  String field;
  StringReserveCheck fieldCheck;
  field.reserve(32); // longest expected field
  fieldCheck.init(field,Serial);  // when parse exits the StringReserveCheck destructor will perform a checkReserve() and print a msg to
  . . .
}
```

Temporary Strings

Minimizing String Memory, below, shows you how to avoid creating short lived temporary Strings most of the time. However some times they are unavoidable. Examples are adding floating point numbers or adding integers in hex format and using the substring function which creates and returns a String. e.g. in statements like

```
  int i = 32;
  str = input.substring(15);
  str += F(" 0x");
  str += String(i,2); // i in hex format
```

Modern compilers seem to be good at optimizing away those temporary Strings once they are not longer needed and in any case they are completely disposed of at the end of the method. If you have a long complicated method and want to 'force' these temporary Strings to be cleaned up you can just use **{ }** around that bit of code e.g.

```
  int i = 32;
  {
    str = input.substring(15);
    str += F(" 0x");
    str += String(i,16); // i in hex format
  } // any temporary Strings are cleaned up here by the compiler.
```

Step 11: Minimizing String Memory Usage

As we have seen above you can use **reserve( )** and **StringReserveCheck** to prevent fragmentation. However creating Strings in your text processing methods still uses heap memory. All though the memory is completely recovered when the method returns, on small memory micro-processors you want to avoid creating extra Strings in your processing methods when you don't have to.

Rules for Avoiding String creation

Avoid creating Strings in the loop()

Strings created in the **loop()** method have a relatively long life as they exist while the loop() does its work and calls the various methods. These Strings should be created as globals or their creation moved to the method where they are used so that they are discarded when the method returns. See the MemFrag example sketches above.

Use String references (String&) for arguments and don't return String results when writing methods

To avoid extra memory usage when calling methods with String arguments and returns write them like this

```
void addMeasurementToString(const String& units, float value, String& rtn) {
  rtn += String(value,3); // float to 3 digits. String(value,3)
  rtn += units;
}
```

The important points to note are
1) Pass the Strings as references i.e. **String&**. This avoids the extra memory and time to copy the String arguments
2) Arguments that are not expected to be modified should be passed as **const String&** Then set the IDE Preferences ***File*** → ***Preferences*** → ***Compiler Warning == ALL***, so that the compiler will warn you with

```
warning: passing 'const String' as 'this' argument discards qualifiers [-fpermissive]
```

if you code attempts to modify that **const String&** argument.

3) The rtn String, **String& rtn**, should have had **reserve()** called on it to so that the extra text can be added without re-allocating its size.

Don't use String(..) unless there is no alternative

Use += or concat() instead e.g.

```
  rtn += 'c';  // or
  rtn += "s1"; // or
  rtn += num
```

The one time you need to use String( ) is when formatting a float or integer. E.g.

```
 rtn += String(f,3); // float to 3 decimals (default is 2) OR
 rtn += String(i,16); // int in hex format
```

For == != or .equals( ) you can compare to either a String or a “str” without creating any extra Strings

String <=, <, >=, > equalsIgnoreCase( ), startsWith( ), endsWith( ), indexOf( ), lastIndexOf( ) and replace( ) when using “...”

When comparing a String to a string (i.e. “text”), the operators <=, <, >=, > and the functions equalsIgnoreCase( ), startsWith( ), endsWith( ), indexOf( ), lastIndexOf( ) and replace( ), first create a String from the “text”. If you are doing the same compare/replace a lot, you might consider creating a global String for the “text”, but usually that is not necessary.

Substring( ) always creates a temporary String

substring( ) is the only String method (apart from the constructors) that returns a new String. So the substring( ) ALWAYS creates a temporary String. However if you assign the returned String to a result String&, the temporary returned String is immediately discarded and its memory recovered.

Do not use the + operator

Don't use the + operator to concatenate Strings as it creates a full String of the all the parts before copying it to the result String. So instead of

```
  rtn = "123"+sb+"345"; // DO NOT do this
```

use these statements that don't create any temporary Strings and, when rtn has been reserved with enough space, no extra memory is used.

```
  rtn = "123";
  rtn += sb;
  rtn +="345";
```

Avoid using begin() and end()

The String begin() and end() functions give uncontrolled access to the underlying String buffer which makes it easy for your code to 'break' the String. So for safety don't use begin() or end() in your sketch.

Step 12: Memory Usage of String Versus Char[], Two Sides of the Same Coin

The stack and heap use the same free memory space. The stack grows down from high memory and the heap grow up from low memory, see the images above. You run out of memory when they overlap and the code stops working as expected.

Using a method local String instead of a fixed char[] is basically just swapping one type of memory usage for another. That is swapping String heap for char[] stack. For example, on UNO and Mega2560, the difference between

```
void localMethod() {
  char test[128];
  test[0] = '\0';
  . . .
}
```

which uses 128 bytes of stack and

```
void localMethod() {
  String test;
  if (!test.reserve(127)) {
    String.println(F("Out-Of-Memory"));
  }
  . . .
}
```

which has a 128 byte heap buffer (127 + 1 for the terminating null) and 8 bytes of stack for the String class, is only 8 bytes.

So using a local String instead of a char[] costs about 8 bytes of SRAM, but gives you bounds checking and protection against buffer overflow and a lot of code convenience. The String allocation is actually safer since there are no checks when the char[] is allocated on the stack. Where as creating a String will not use heap memory, if there is none left. You can check the return from **reserve( )** to see if there is enough memory. That is not possible when allocating local char[] on the local stack. Also on UNO and Mega2560, using local Strings is very safe, since when the program tries to allocate the buffer, it ALWAYS keeps 128 bytes free for stack use. If there is not an extra 128 bytes available for the stack, the String allocation fails

Step 13: Why You Should Not Use C-string Methods, E.g. Strcat, Strcpy Etc, or Char[] Manipulations

C-string methods have been the bane of programmers for over 30 years and the cause of so many programming errors and security breaches that Microsoft has banned their programmers from using them (local copy here) and text books have been written on why they should not be used. For example Secure Coding in C and C++, 2nd Edition by Robert C. Seacord, Ch5 (local copy here) which discusses the dangers of particular c-string methods and Ch2 (local copy here) which talks about how easy it is to get coding errors. Low level operations on char[]s, a char at a time, are also a major source of errors. Hackers love the use of c-string methods and char[] code as the associated coding errors allow them to force buffer overflows and gain access to computer systems. See the Wikipedia Buffer_overflow entry and for just the latest example of a commercial coding error using c-strings and char[] operations, see the 'bug' in Unix's sudo code. The lastest IPhone security failing is another recent example.

Since C++ 1.0 was released there have been many implementations of a String class to overcome the systemic coding errors associated with using c-string methods. std::string was standarized in C++98. Arduino has its own version of the String class which this tutorial covers.

A Buffer Overflow sketch – c-string versus String versus SafeString

It is trivial to write a small sketch that exhibits a c-string coding error. Dedicated C programmers complain that that is just a user coding error and could be fixed with careful coding. However over 30 years of experience has shown that these types of coding errors are very, very common and can be difficult to find. The coding errors often remain hidden until a particular program sequence is run or the input data fills the char[]. For a life example see this posting on the Arduino Forum Use of string char causes Arduino code to restart

The c-string / char[] alternative

Consider the following sketch bufferOverflow_ex1.ino

```
void setup() {
 . . .
}
void appendCharsTo(char* strIn) {
  // should really check the bounds here but..
  // i) cannot tell from char* how big the char[] is
  // ii) no one ever seems to add the bounds checking code anyway.
  strcat(strIn, " some more text");
  Serial.print(" appendCharsTo returns:");  Serial.println(strIn);
}
void loop() {
  Serial.println("--------- start of loop()");
  char str1[24] = "some str1"; // allow extra space for strcat
  char str2[] = "some str2 other text";
  appendCharsTo(str1);
  Serial.print("str1:");   Serial.println(str1);
  Serial.print("str2:");   Serial.println(str2);
  Serial.println("--------- end of loop()");
```

When you run this sketch on an UNO it appears to work as expected. However when you run it on an ESP8266, str2 is cleared, and on an ESP32 the sketch repeatedly reboots. The error is not too hard to find from the ESP32 stack dump, but on the UNO you would not even be looking for the error and on the ESP8266 it is not obvious what is causing the empty str2.

The Arduino String Alternative

Changing to Arduino Strings is simple and avoids the error completely, bufferOverflow_ex2.ino

```
void setup() {
. . .
}
void appendCharsTo(String & strIn) {
  strIn += " some more text";
  Serial.print(" appendCharsTo returns:");  Serial.println(strIn);
}
void loop() {
  Serial.println("--------- start of loop()");
  String str1 = "some str1";  // Note: declaring Strings in the loop() is not recommended as it can lead to memory fragmentation
  String str2 = "some str2 other text";
  appendCharsTo(str1);
  Serial.print("str1:");   Serial.println(str1);
  Serial.print("str2:");   Serial.println(str2);
  Serial.println("--------- end of loop()");
}
```

Using Arduino Strings, the str1 is automatically expanded to store the additional text. In bufferOverflow_ex2.ino the expansion of str1 leaves a hole in the heap, but that does not stop the program from running correctly and using the guidelines at the top of this tutorial completely avoids that memory fragmentation.

The SafeString Alternative

You can also use the SafeString library as an alternative to Arduino Strings. In this case the SafeString wraps the char[] and protects it from buffer overflows and out of bounds indexing. Basically adding all the checking code a proficient, dedicated programmer would add. bufferOverflow_ex3.ino

```
#include "safestring.h"
// install the SafeString library V4+ from Arduino library manager or
// download the zip file from <a href="https://www.forward.com.au/pfod/ArduinoProgramming/SafeString/index.html"> https://www.forward.com
void setup() {
 . . .
}
void appendCharsTo(SafeString& strIn) {
```

```
  // pass strIn as a reference &
  strIn += " some more text"; // this does all the bounds checks
  Serial.print(" appendCharsTo returns:");  Serial.println(strIn);
}
void loop() {
  Serial.println("--------- start of loop()");
  char str1[24] = "some str1"; // allow extra space for appendCharsTo
  char str2[] = "some str2 other text";
  createSafeStringFromCharArray(sfStr1, str1); // or cSFA(sfStr1,str1); for short.  Wrap str1 in a SafeString
  appendCharsTo(sfStr1);
  if (SafeString::errorDetected()) { // set true if any SafeString has an error.  Use hasError() on each SafeString to narrow it down or
    Serial.println(F("Out of bounds error detected in appendCharsTo"));
  }
  Serial.print("str1:");   Serial.println(str1);
  Serial.print("str2:");   Serial.println(str2);
  Serial.println("--------- end of loop()");
```

When you run bufferOverflow_ex3.ino it prints out.

```
--------- start of loop()<br> appendCharsTo returns:some str1
Out of bounds error detected in appendCharsTo
str1:some str1
str2:some str2 other text
--------- end of loop()
```

You can also put **SafeString::setOutput(Serial)** in the setup() to get more detailed error messages.

Summary: Over 30 years of experience has shown it is very easy to make coding errors using c-strings and char[] manipulations. C++ Strings were introduced to overcome these systemic coding errors. Sometimes the c-string errors don't show up immediately. Arduino Strings avoid the errors, while SafeString flags them. Both of Arduino Strings and SafeStrings allow the sketch to continue to run. Either Arduino Strings or SafeString should be used in preference to c-string methods or char[] manipulations

Step 14: Errors in Arduino's String

The current the Wstring.cpp code supplied with Arduino contains a number of bugs which cause programs using it to either fail or give un-expected results. For example

```
  String a;
  a = "12345";
  Serial.println(a);
  a += a;
  Serial.println(a);
```

causes an Arduino UNO to continually reboot.

Another example is

```
  String aStr;
  aStr = "";
  aStr.concat('\0');
  Serial.print("aStr .length():");Serial.println(aStr .length());
  Serial.print("strlen(aStr.c_str()):");Serial.println(strlen(aStr .c_str()));
```

which outputs

```
aStr.length():1
strlen(aStr.c_str()):0
```

That is the String object length() is no longer the same as the strlen() of the underlying char buffer.

The number conversion method, Arduino String **toInt()** can give odd errors, but that is due to the underlying c-string method and cannot be easily fixed. For example

```
String a_str("123456789012345");
Serial.println(a_str.toInt());
```

Outputs

```
-2045911175
```

and

```
String a_str("5.95"); OR String a_str("5a");
Serial.println(a_str.toInt());
```

Outputs

```
5
```

Although some programmers don't consider that an error.

Step 15: The SafeString Alternative

As has been shown above Arduino Strings are a very practical and safe method of programming text processing. However the SafeString library offers a number of advantages over using Strings.

Complete control over memory use

SafeStrings are a wrapper around a char[] that provides safety from buffer overruns and out of range indexing. Because SafeString uses a fixed char[], the memory usage is fixed when you create the SafeString and does not change. When you create global SafeStrings their memory usage is reported by the compiler so it is simpler to see how much SRAM your code is using. SafeStrings also use less memory because they don't need to convert “text” to a SafeString before using it in compare and indexOf functions.

Catches, flags and ignores Errors and prints a detailed message to help you fix the error

SafeString checks every one of its function calls for null pointers, buffer-overrun and out of range indexing. Function calls the would cause an error are ignored, i.e. the SafeString remains unchanged, and the error is flagged and a detailed error messge displayed that allows you to pin point the problem and fix it. SafeString also does not have the errors that are in Arduino String's code

Simple non-blocking Serial Readers

The Arduino String reader, Serial.readStringUntil() can block the loop waiting while waiting for data. While you can code non-blocking readers using Strings, the methods in the SafeString are simpler and have a number of advanced functions like tokenizing the input, non-blocking input timeouts, selectable echo of the input and a skipToDelimiter function to skip over un-wanted or partial input. The SafeString readers also count the number of chars read so you can check when an entire web response has been received.

Better number parsing methods

As noted above, the String toInt() method can give odd results. The SafeString toInt(int i), toLong(long l) etc, return true if the string is a valid number and updates the argument, otherwise they return false. The String toInt() method just returns 0 if the string is not a number and returns 5 for “5.9” and for “5a”

Easier to use with libraries

Third party libraries often return their text results as a char*. If you use Strings, you will be making a complete copy of the text when you assign the result to a String, using extra memory. On the other hand with SafeString, you can just wrap the char* in a SafeString without taking a copy and then safely parse and process the data.

More text manipulation functions

While Strings have startsWith, endsWith, indexOf, lastIndexOf, substring, replace and remove methods, SafeString has all of those methods plus a full set of print( ) methods to print to a SafeString and prefix( ), startsWithIgnoreCase( ), indexOfCharFrom( ), removeBefore( ), removeFrom( ), removeLast( ), stoken( ), nextToken( ) and non-blocking readFrom( ), writeTo( ), read( ), readUntil( ), readUntilToken( )

The SafeString library also has a SafeStringStream class that lets you automate the testing of sketches that expect to process Stream data input.

There are also BufferedInput and BufferedOutput classes that provide larger input buffers and non-blocking Serial output. The non-blocking millisDelay timerclass and a loopTimer class are also included in the SafeString library.

Step 16: Conclusion

Arduino Strings are not as bad as they have been made to be and by following a few guidelines, can be used with confidence in your sketches. Arduino Strings use minimal extra SRAM memory over a plain char[], about 8bytes/String, but that 8 bytes provides you with complete safety from buffer overruns and missing string terminators as well as convenient coding methods.

If you want to keep complete control over your memory usage or you are working with a library that returns a char* or you want a richer set of readers, parsers and text functions or want detailed debugging/error messages, then you can install the SafeString library from the Arduino library manager.

Either of these alternatives, String or SafeString, are preferable to using error prone low level char[] manipulations and c-string methods, like strcat and strcpy, which are the systemic cause of so many programming errors.