text_printf: A Higher-Level Alternative to printf

Yixin Wang
Stanford University

Abstract

We present an output formatting function text_printf for C++ with similar usage as printf. Most general-purpose programming languages provide a function with similar capabilities as printf: the width, alignment, and precision of individual output elements can be specified, but no mechanism is provided to specify higher-level formatting, such as the width and alignment of whole groups of output elements. As a result, more complex outputs such as tables and text paragraphs are not easily achievable with printf and similar functions. We addressed these issues by developing a higher-level mini-language for use by the format string of text_printf that allows more complex formatting mechanisms to be described, including the aforementioned ability to specify width/alignment for groups of output elements. C++ code using text_printf was compared to standard C++ code that produced the same output. In each case, the code using text_printf was significantly simpler and more readable.

1. Introduction

Formatting output is a common task that comes up in all phases of a programming project. It can come in the form of debug prints, outputting unit test results, displaying the --help for a console program, etc. In many cases, the formatting needed for the output is minimal: inserting a few tabs into the string or specifying the width/precision of individual elements is typically all that's needed. For these purposes, the output formatting functions provided by most general-purpose programming languages will be adequate. This would be printf [6] or iomanip [1] for C++, System.out.format [3] for Java, and str.format() [5] for Python.

Unfortunately, there are instances where more complicated, higher-level formatting is needed. Examples include printing paragraphs of aligned text, printing content in multiple columns, and printing tables with ASCII borders. These cases are not easily handled by the aforementioned standard formatting functions since they require width constraints to be applied to groups of elements instead of just single elements. For example, suppose a programmer is working in C++ and wants to format output into two columns with each line containing multiple elements of varying length per column. That means for each output line, the programmer needs to be able to specify the width of all the elements in the first column as a group and the width of all the elements in the second column as a group. Neither printf nor iomanip can easily do that.

Situations that require such higher-level formatting, though less common, require a large programming effort to properly implement. The necessary code will likely be tedious with lots of low-level string manipulation and corner cases to consider. text_printf is our system aimed at addressing this problem: it's a C++ output formatting function with similar usage to printf whose format-specifier language provides the programmer with more powerful, higher-level formatting options.

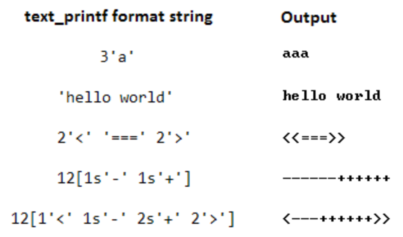
We'll first go over the details of the syntax of the text_printf mini-language that's used for describing the format (section 2). Next, there will be a quick overview of how the system handles errors (section 3). For our results (section 4), code using text_printf will be compared to standard C++ code that produces the equivalent output. The comparison will be made for several output scenarios.

2. The text_printf Mini-Language

The text_printf function is similar in usage to the printf function in C: it accepts a format string argument followed by a variable number of data arguments. Here, we'll go over the mini-language that describes the syntax of the text_printf format string.

2.1 Fillers

The most basic element in the mini-language are fillers, of which there are two types: repeated-char and string-literal. A **repeated-char** (first example in figure 1) is a length followed by a character literal (a single character surrounded by single quotes). Its corresponding output is the specified character repeated, with the number of times repeated equal to the length. In the first example of figure 1, the repeated-char has **fixed-length**, which is specified by a nonnegative integer value at the start of the content's description (3 in this example). Any element with fixed-length means the number of columns in its corresponding output will always be the same. A **string-literal** (second example in figure 1) is just a string surrounded in single quotes; its corresponding output is simply the string inside the quotes. A string-literal also has fixed-length, although it's implicit (the second example in figure 1 has fixed-length of 11). The third example of figure 1 shows how elements can be concatenated together.

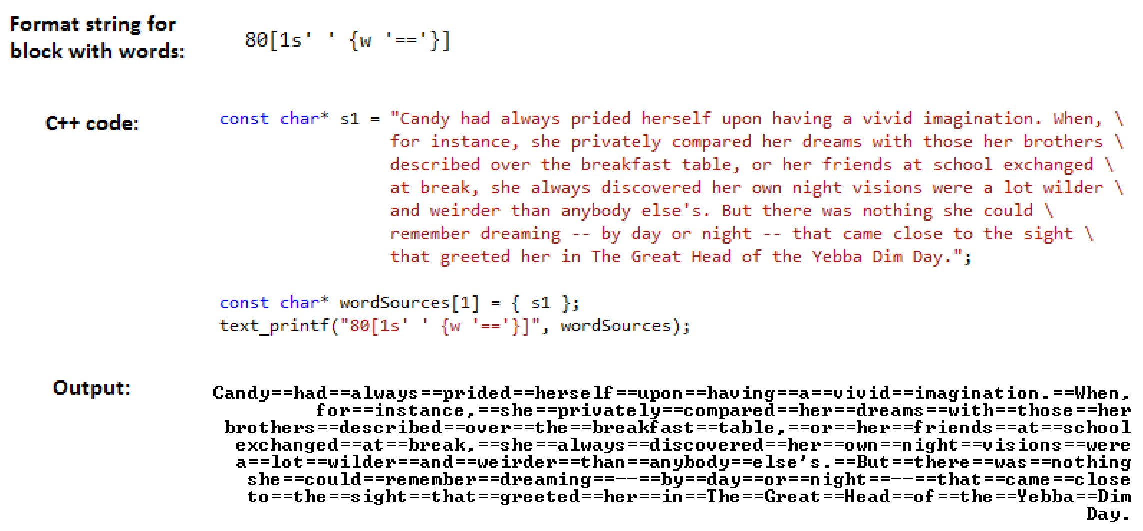


text_printf format string	Output
3'a'	aaa
'hello world'	hello world
2'<' '===' 2'>'	<<===>>
12[1s'-' 1s'+']	------++++++
12[1'<' 1s'-' 2s'+' 2'>']	<---++++++>>

Figure 1: Five examples of format strings for text_printf and the corresponding outputs they produce. The first two examples demonstrate the two different types of fillers with fixed length: a repeated-char and a string-literal; the third is a concatenation of three fillers with fixed length; the last two are blocks containing multiple fillers of fixed-length and share-length.

2.2 Blocks

The fourth and fifth examples in figure 1 both demonstrate a **block**. A block contains one or more elements and enforces a total length on those elements. Typically, at least one element inside a block will have **share-length** (specified by a nonnegative integer followed by the letter s) instead of fixed-length. All share-length elements inside a block will distribute among themselves the portion of the block's length that's not taken up by its fixed-length elements. The share-length of each element specifies how many "shares" of that remaining length should be given to that element. Since the actual length of share-length elements depend on their parent block, all share-length elements must be inside a block. The last two examples in figure 1 demonstrate share-length content. In the fourth example in figure 1, the block has length 12 and contains two filler elements, each with a share-length of 1. That means the block's length will be distributed amongst the two fillers equally, since they'll both claim 1 share of that length. As a result, the two fillers both end up having an actual length of 6. In the fifth example in figure 1, the block again has a length of 12, but this time it contains four filler



Format string for block with words:

```
80[1s' ' {w '=='}]
```

C++ code:

```
const char* s1 = "Candy had always prided herself upon having a vivid imagination. When, \
                 for instance, she privately compared her dreams with those her brothers \
                 described over the breakfast table, or her friends at school exchanged \
                 at break, she always discovered her own night visions were a lot wilder \
                 and weirder than anybody else's. But there was nothing she could \
                 remember dreaming -- by day or night -- that came close to the sight \
                 that greeted her in The Great Head of the Yebba Dim Day.";

const char* wordSources[1] = { s1 };
text_printf("80[1s' ' {w '=='}]", wordSources);
```

Output:

```
Candy==had==always==prided==herself==upon==having==a==vivid==imagination.==When,
         for==instance,==she==privately==compared==her==dreams==with==those==her
 brothers==described==over==the==breakfast==table,==or==her==friends==at==school
  exchanged==at==break,==she==always==discovered==her==own==night==visions==were
  a==lot==wilder==and==weirder==than==anybody==else's.==But==there==was==nothing
   she==could==remember==dreaming==--==by==day==or==night==--==that==came==close
   to==the==sight==that==greeted==her==in==The==Great==Head==of==the==Yebba==Dim
                                                                            Day.
```

Figure 2: An example of a block of length 80 with words. The format string and C++ code to generate the output are shown at the top and middle. The words in the output (bottom) will be right-aligned because of the 1-share-length filler that appears before the words element in the block.

elements, with the first and last of them having fixed lengths of 1 and 2 respectively. That means the two other elements of the block will have 12-1-2=9 length to distribute amongst them. Since these two share-length elements have share-lengths of 1 and 2, their actual lengths will be 3 and 6 respectively.

2.3 Words

The **words** element is used when the user wants to print paragraphs of text. Each words element that appears in the format string must be provided with a corresponding char* to tell it the words to print. The string pointed to by this char* will have its contents interpreted as whitespace-delimited words. The char* pointers for all the words elements in the format string should be passed to text_printf as an array of char*. The pointers in the array should be in the same order as their corresponding words elements in the format string.

The syntax for specifying a words element in the text_printf format string is as follows: it starts with an opening curly brace, followed by the letter w, followed by zero or more fillers, followed by a closing curly brace. These fillers are called the **inter-word fillers**: they specify what gets inserted between each word in the output. A words element will result in one or more lines of output depending on the source text. The length of each line of output depends on the parent (surrounding) block. The words element can be thought of as having a greedy length: in each line of output, it will try to take up as much length of its parent block as possible without breaking up a word across multiple lines. Because of this, a block cannot have more than one words element.

Figure 2 shows an example of a words element inside a block of length 80. This particular words element only has one inter-word filler: the string literal '=='. That means any two consecutive words on the same line of output will have '==' inserted between them. In each output line, the words element will try to take up as much of the 80 length of its parent block as possible without splitting a word across multiple output lines. Any remaining length in that line will be distributed to the share-length contents of the parent block. In this case, all of it will be distributed to the 1 share-length repeated-char at the start of the block, which means the remaining length in each line will appear as spaces at the front of the line. As a result, the words in the output are right-aligned (bottom of figure 2).

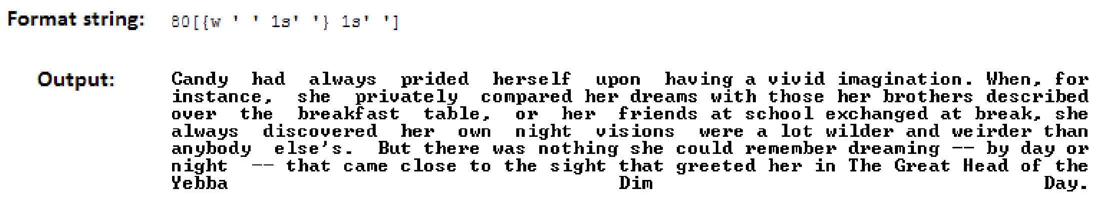
In real-world use, the most common inter-word filler would just be a single space since that's how text is usually displayed. Also, text is typically left-aligned, right-aligned, or center-aligned. The corresponding text_printf format strings for these three common cases are shown below with a block length of 80:

Left-align `80[{w ' '} 1s' ']`

Right-align `80[1s' ' {w ' '}]`

Center-align `80[1s' ' {w ' '} 1s' ']`

This is all fine, but what if the inter-word fillers have share-length instead of fixed length? Then, in each output line, any of the block's 80 length not taken up by words and fixed-length inter-word fillers will be distributed to the share-length inter-word fillers, instead of going to the other share-length content in the parent block. An example of this is shown in figure 3, which is the format string to print justified text. As we can see in the output, in each line the remaining length of the



Format string: `80[{w ' ' 1s' '} 1s' ']`

Output:

```
Candy  had  always  prided  herself  upon  having a vivid imagination. When, for
instance,  she  privately  compared her dreams with those her brothers described
over  the  breakfast  table,  or  her  friends at school exchanged at break, she
always  discovered  her  own  night  visions  were a lot wilder and weirder than
anybody  else's.  But there was nothing she could remember dreaming -- by day or
night  -- that came close to the sight that greeted her in The Great Head of the
Yebba                                  Dim                                  Day.
```

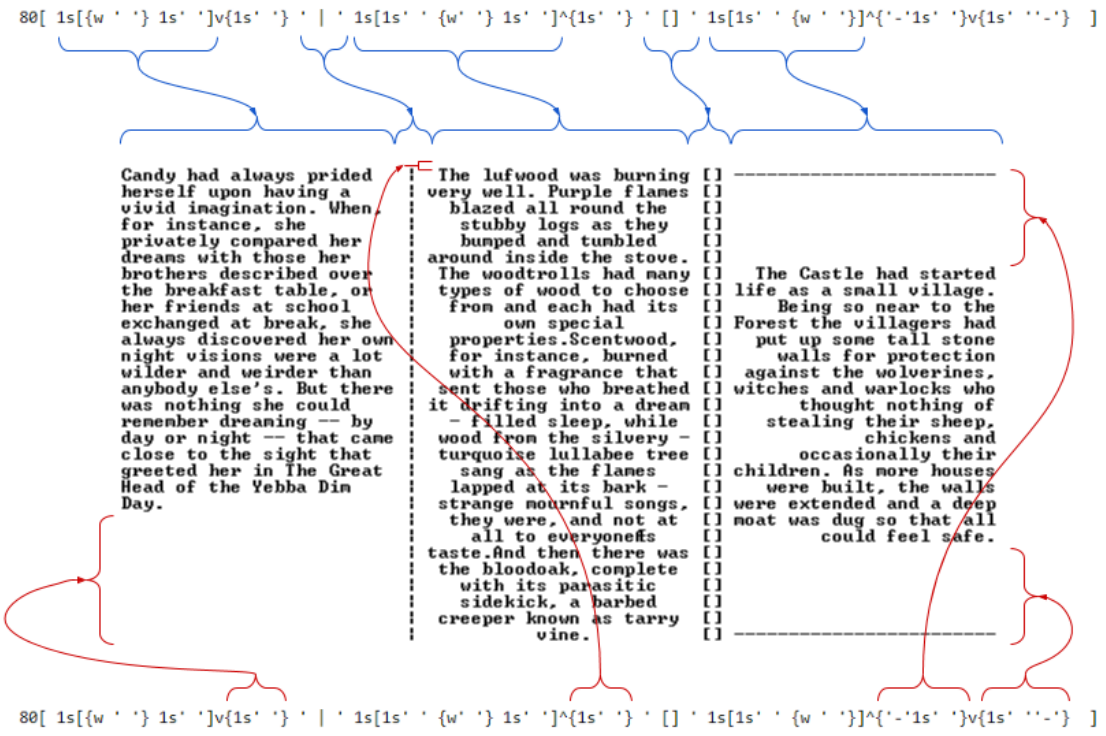
Figure 3: A block of length 80 that produces justified text. This is accomplished using share-length inter-word fillers, which allows the length in a block that's not taken up by words to be distributed to share-length elements between words instead of other share-length elements in the block.

parent block is evenly distributed to the shares of space between words as opposed to the 1 share of space at the end of the parent block. A natural question to ask is why we need the 1 share of space at the end of the block if the remaining length is always taken by the inter-word shares of space? That's because sometimes, a line of output may end up with only a single word, in which case no inter-word fillers will be inserted for that line (inter-word fillers are only inserted between two words on the same output line). In that case, there must be some share-length content in the block to distribute that remaining length to, which is why the format has the 1 share of space at the end of the block. As a result, any line with only a single word will appear left-aligned in the output.

2.4 Nested Blocks

We've shown how a block can contain multiple elements such as fillers and/or a words element. We'll now demonstrate how a block can contain other blocks. Figure 4 shows the format string for a block that contains three inner blocks, each containing a words element. The format string for each of the inner blocks should look familiar: the first produces left-aligned text, the second produces center-aligned text, and the last produces right-aligned text. The outer block also contains some string-literal fillers that serve as vertical borders between the outputs of adjacent inner blocks. The arrows originating from the top in figure 4 matches up the inner blocks and fillers with their corresponding output columns. Allowing a block to contain multiple inner blocks raises a problem: how many lines of output should the parent block produce? Each inner block could produce a different number of lines of output! Naturally, the number of lines produced by the parent block should equal the maximum number of lines produced by any inner block. But that means some inner blocks will produce fewer lines of output than the parent block, so how is this handled? This is where **vertical fillers** come in: t**op vertical fillers** and **bottom vertical fillers** allow the user to specify the output lines to be inserted above and below the output of an inner block respectively, so the total number of output lines matches that of the parent block. To specify the top vertical fillers for a block, append a carat character to the end of the block's format description, followed by a set of curly braces. Any fillers placed inside the curly braces will become the top vertical fillers for that block. To specify bottom vertical fillers for a block, the syntax is identical except a lowercase v is used instead of the carat character (figure 4). Even though these vertical fillers have the same syntax as fillers used anywhere else in the format string, their corresponding output will be different: instead of printing several characters in a line, a vertical filler prints several lines with each line repeating a character. The length of each line will be the same as the length of the block that the vertical filler is attached to. The difference in the number of output lines between a parent block and a child block are distributed to the vertical fillers of that child block in a familiar way: the fixed-length vertical fillers will take up some of those lines, and the remaining lines will be distributed to the share-length vertical fillers according to their share-lengths.

Figure 4 gives a concrete example of vertical fillers. The three inner blocks in the example each have top and/or bottom vertical fillers specified, and the arrows from the bottom of figure 4 show which vertical fillers correspond to which lines of output. The first inner block only has a bottom vertical filler: 1 share of the space character. That means any additional output lines needed to match the parent block will appear as lines of spaces below it (resulting in the text having top vertical alignment). The second inner block only has a top vertical filler: 1 share of space. That means any additional output lines it needs will appear above it as lines of spaces (resulting in the text having bottom vertical alignment). However, this child block has the same number of lines as the parent block since it has the most output lines out of the three child blocks. That means its top vertical filler, the 1 share of space, ends up producing zero lines of output. The last inner block



```
80[ 1s[{w ' '} 1s' ']v{1s' '} ' | ' 1s[1s' ' {w' '} 1s' ']^{1s' '} ' [] ' 1s[1s' ' {w ' '}]^{'-'1s' '}v{1s' ''-'}  ]
```

```
Candy had always prided   ¦  The lufwood was burning [] ------------------------
herself upon having a     ¦ very well. Purple flames []
vivid imagination. When,  ¦   blazed all round the   []
for instance, she         ¦    stubby logs as they   []
privately compared her    ¦    bumped and tumbled    []
dreams with those her     ¦ around inside the stove. []
brothers described over   ¦  The woodtrolls had many []   The Castle had started
the breakfast table, or   ¦  types of wood to choose [] life as a small village.
her friends at school     ¦   from and each had its  []     Being so near to the
exchanged at break, she   ¦        own special       [] Forest the villagers had
always discovered her own ¦   properties.Scentwood,  []   put up some tall stone
night visions were a lot  ¦   for instance, burned   []     walls for protection
wilder and weirder than   ¦   with a fragrance that  []  against the wolverines,
anybody else’s. But there ¦  sent those who breathed [] witches and warlocks who
was nothing she could     ¦ it drifting into a dream []       thought nothing of
remember dreaming -- by   ¦   - filled sleep, while  []    stealing their sheep,
day or night -- that came ¦  wood from the silvery - []             chickens and
close to the sight that   ¦  turquoise lullabee tree []       occasionally their
greeted her in The Great  ¦    sang as the flames    [] children. As more houses
Head of the Yebba Dim     ¦   lapped at its bark -   []    were built, the walls
Day.                      ¦  strange mournful songs, [] were extended and a deep
                          ¦   they were, and not at  [] moat was dug so that all
                          ¦     all to everyoneÆs    []         could feel safe.
                          ¦ taste.And then there was []
                          ¦  the bloodoak, complete  []
                          ¦    with its parasitic    []
                          ¦    sidekick, a barbed    []
                          ¦  creeper known as tarry  []
                          ¦          vine.           [] ------------------------
```

```
80[ 1s[{w ' '} 1s' ']v{1s' '} ' | ' 1s[1s' ' {w' '} 1s' ']^{1s' '} ' [] ' 1s[1s' ' {w ' '}]^{'-'1s' '}v{1s' ''-'}  ]
```

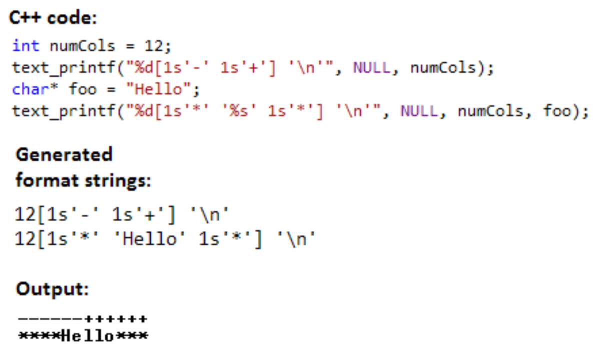
Figure 4: An example of a block with three nested blocks and its output. The format string is shown above and below the output: the arrows from the top format string show which part of the format corresponds to which columns of the output; the arrows from the bottom format string shows which part of the format corresponds to which rows of vertical filler.

has both top and bottom vertical fillers, this time with some fixed-length fillers thrown in: the top vertical fillers begin with a length 1 string literal containing the dash character, indicating the first output line above this block should be the dash character repeated. Likewise, the bottom vertical fillers end with the same string literal, indicating the last output line below this block should be the dash character repeated. The remaining output lines are distributed evenly between the share of space in the top vertical fillers and the share of space in the bottom vertical fillers (resulting in the text having center vertical alignment).

We should note that the text_printf mini-language has a constraint on blocks: a block cannot contain both a words-element and a child block. This ensures that the column positions where a block starts and ends are the same in all output lines of the block. To see why this is, recall that a words element has greedy length: it tries to fit as many words as possible in each output line of the parent block without splitting a word across different lines. However, depending on what words are in each line, the length taken up by the words element may vary from line to line. If the parent block also contains a child block, then that child block's length or its start/end column positions may vary from line to line due the words element taking up different amounts of length in each line. Allowing the output lines of a block to start/end at different column positions would be confusing, and for this reason, text_printf will prevent blocks from having both a words element and a child block.

2.5 printf Format Codes

The text_printf mini-language allows any of the printf format codes to be used inside it; it will replace them with the values of the corresponding data arguments before parsing the format string. Here is an example of how this can be used:



C++ code:

```cpp
int numCols = 12;
text_printf("%d[1s'-' 1s'+'] '\n'", NULL, numCols);
char* foo = "Hello";
text_printf("%d[1s'*' '%s' 1s'*'] '\n'", NULL, numCols, foo);
```

Generated format strings:

```
12[1s'-' 1s'+'] '\n'
12[1s'*' 'Hello' 1s'*'] '\n'
```

Output:

```
------++++++
****Hello***
```

This gives text_printf the same ability as printf to allow output content to change based on the values of variables.

The current way our system implements printf format codes is naïve: as a preprocessing step, the format string and the data arguments passed to text_printf are simply passed through to sprintf, and the resulting string is passed to the rest of the pipeline. That means printf format codes can be used anywhere in the text_printf format string and can expand to anything, including tokens of the text_printf mini-language (e.g. square brackets, curly braces, single quotes). This flexibility allows for confusing and nonsensical expansions, but also allows for some useful patterns. One such pattern would be to store the format strings of child blocks in their own strings first, and then use the %s format code to combine them into the format string for the parent block. This can lead to more readable code especially if a parent block has many child blocks with complex format strings or if there are many levels of nested blocks.

3. Error Handling

Errors in the format string syntax are caught during parsing and will immediately result in an error message indicating where in the format string the error occurred. Any other error will occur during one of the length evaluation phases, which refer to several steps in the pipeline where the actual lengths of elements with non-fixed length are computed. A common task in these phases is to distribute some length to a group of fixed-length and share-length elements. Two errors may occur: the sum of the fixed-length elements may exceed the total length, or the total length exceeds the sum of the fixed-length elements but there are no share-length elements to distribute the remaining length to. These errors will result in an error message that points to the block in the format string where the error occurred. Something of interest about the second of those two errors is the fact that there exist format strings containing words elements for which this error will only sometimes occur depending on what words end up in each output line (this is discussed further in section 5.3). This is an undesirable characteristic of our system in its current state.

4. Results

We apply text_printf to four scenarios found on the web where the programmer's goal was to print formatted text using C++. These scenarios were found as questions raised on stackoverflow.com and cplusplus.com/forum. In the first three scenarios, the code suggested by another user is compared to text_printf code that generates the same output. The fourth scenario did not have any user-suggested code, so only the text_printf solution will be presented.

Scenario 1: Aligning a group of elements

In the first scenario, the programmer is trying to print out the stats of a player in a game as name-value pairs [7]. Figure 5 shows how using text_printf can result in simpler code due to its support for grouping multiple output elements. On the other hand, C++'s std::setw [10] cannot be applied to groups of output elements. As a result, groups of elements need to be written into a separate string before using std::setw.

Scenario 2: Center-aligned text

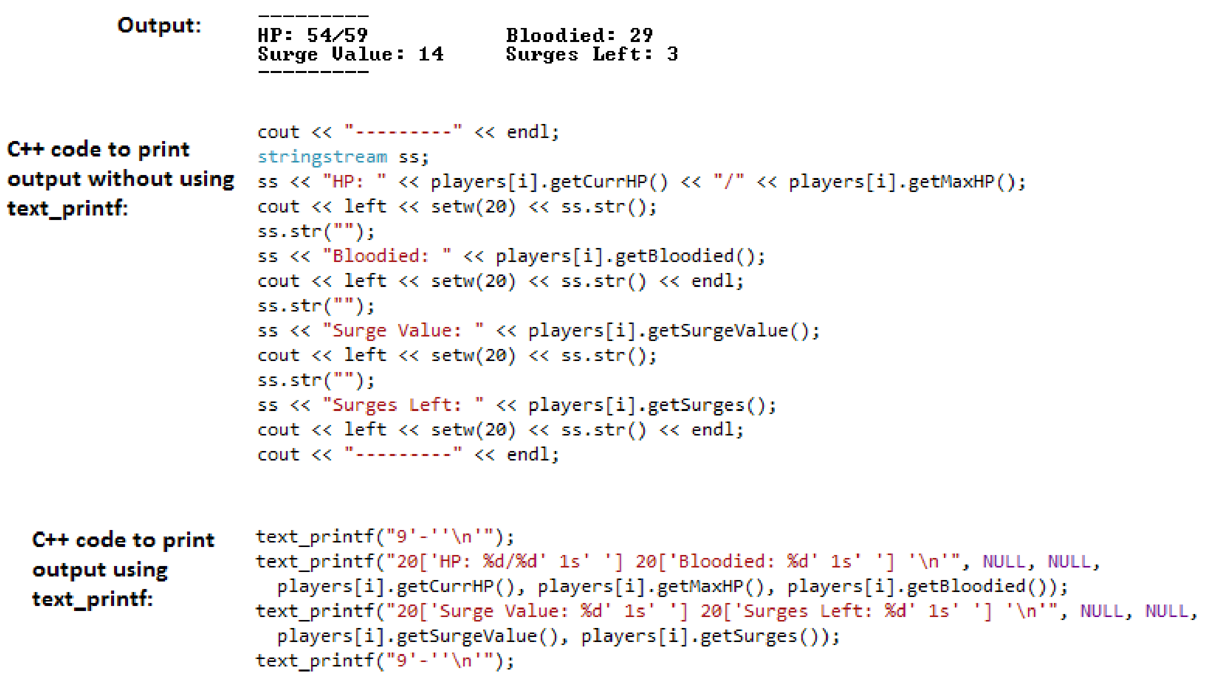
Next up, we have a programmer who wants to print some column headers that are centered in their respective columns [2]. Figure 6 shows how the task is trivial when using text_printf while the other solution requires the programmer to implement his own center-align format flag.

Scenario 3: Justified text.

For this scenario, we have a programmer who wishes to print a paragraph of justified text [4] (figure 7). Again, this is done trivially using text_printf and requires quite a bit of C++ code otherwise.

Scenario 4: Table.

Finally, we have a programmer who wishes to print an ASCII table that lists the printable characters [11] (figure 8). This is a more complicated example: each line of the output requires its own text_printf call. Since each row of the table is split into 12 columns, the text_printf format string will need to have a root block with 12 child blocks, each specifying the format for center-aligned text. This results in a very long format string. We could define it directly using a very long string literal, but instead we'll build it up in a loop since the format description is repeated for every pair of columns. After the format strings are built, we simply pass them to text_printf and print the lines of the table in a loop.


Output:

```
----------
HP: 54/59           Bloodied: 29
Surge Value: 14     Surges Left: 3
----------
```

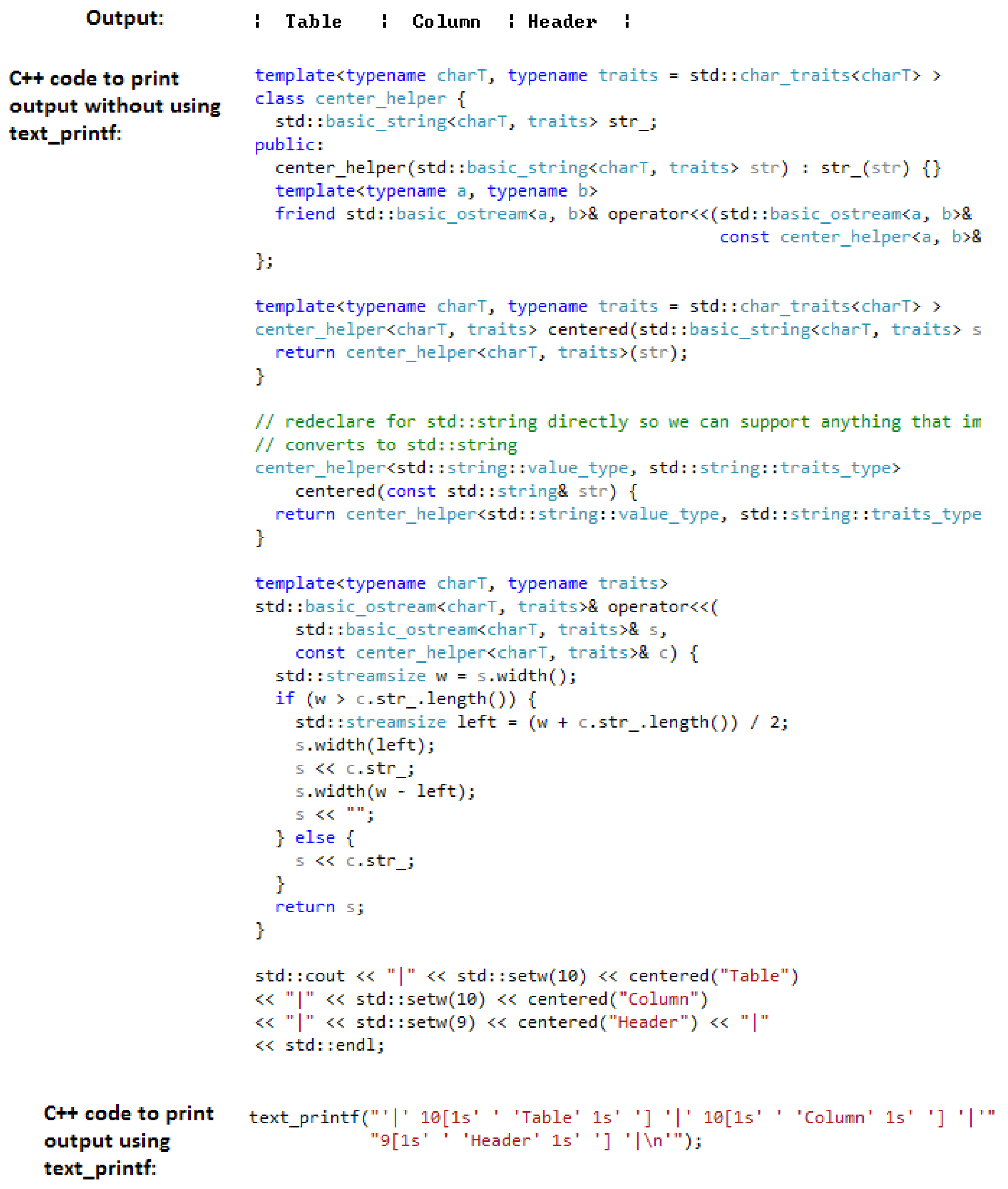
C++ code to print output without using text_printf:

```
cout << "---------" << endl;
stringstream ss;
ss << "HP: " << players[i].getCurrHP() << "/" << players[i].getMaxHP();
cout << left << setw(20) << ss.str();
ss.str("");
ss << "Bloodied: " << players[i].getBloodied();
cout << left << setw(20) << ss.str() << endl;
ss.str("");
ss << "Surge Value: " << players[i].getSurgeValue();
cout << left << setw(20) << ss.str();
ss.str("");
ss << "Surges Left: " << players[i].getSurges();
cout << left << setw(20) << ss.str() << endl;
cout << "---------" << endl;
```

C++ code to print output using text_printf:

```
text_printf("9'-''\n'");
text_printf("20['HP: %d/%d' 1s' '] 20['Bloodied: %d' 1s' '] '\n'", NULL, NULL,
  players[i].getCurrHP(), players[i].getMaxHP(), players[i].getBloodied());
text_printf("20['Surge Value: %d' 1s' '] 20['Surges Left: %d' 1s' '] '\n'", NULL, NULL,
  players[i].getSurgeValue(), players[i].getSurges());
text_printf("9'-''\n'");
```

Figure 5: Two versions of C++ code to print the four lines of text shown (top): the first version (middle) uses standard C++ functions; the second version (bottom) uses text_printf. Since std::setw is only applied to the next thing that's inserted into the cout stream, the programmer must do the following for each name-value pair: write the output of that name-value pair into a separate string, and then apply std::setw to that string when inserting into the cout stream. This way, std::setw will be applied to the entire output of that name-value pair. In contrast, the text_printf block structure easily allows specifying widths for multiple pieces of output. As a result, the text_printf version of the code is shorter and more readable.



Output:

```
¦  Table    ¦  Column   ¦ Header  ¦
```

C++ code to print output without using text_printf:

```cpp
template<typename charT, typename traits = std::char_traits<charT> >
class center_helper {
  std::basic_string<charT, traits> str_;
public:
  center_helper(std::basic_string<charT, traits> str) : str_(str) {}
  template<typename a, typename b>
  friend std::basic_ostream<a, b>& operator<<(std::basic_ostream<a, b>&
                                              const center_helper<a, b>&
};

template<typename charT, typename traits = std::char_traits<charT> >
center_helper<charT, traits> centered(std::basic_string<charT, traits> s
  return center_helper<charT, traits>(str);
}

// redeclare for std::string directly so we can support anything that im
// converts to std::string
center_helper<std::string::value_type, std::string::traits_type>
    centered(const std::string& str) {
  return center_helper<std::string::value_type, std::string::traits_type
}

template<typename charT, typename traits>
std::basic_ostream<charT, traits>& operator<<(
    std::basic_ostream<charT, traits>& s,
    const center_helper<charT, traits>& c) {
  std::streamsize w = s.width();
  if (w > c.str_.length()) {
    std::streamsize left = (w + c.str_.length()) / 2;
    s.width(left);
    s << c.str_;
    s.width(w - left);
    s << "";
  } else {
    s << c.str_;
  }
  return s;
}

std::cout << "|" << std::setw(10) << centered("Table")
<< "|" << std::setw(10) << centered("Column")
<< "|" << std::setw(9) << centered("Header") << "|"
<< std::endl;
```

C++ code to print output using text_printf:

```cpp
text_printf("'|' 10[1s' ' 'Table' 1s' '] '|' 10[1s' ' 'Column' 1s' '] '|'"
            "9[1s' ' 'Header' 1s' '] '|\n'");
```

Figure 6: Two versions of C++ code to print the line of text shown (top): the first version (middle) does not use text_printf and instead defines some helper functions and classes to add center-align support to C++'s ostream; the second version (bottom) uses a single text_printf call to achieve the same output.

,

Output:

```
Candy  had  always  prided  herself upon having a vivid imagination. When, for
instance,  she privately compared her dreams with those her brothers described
over  the  breakfast  table,  or her friends at school exchanged at break, she
always  discovered  her  own  night visions were a lot wilder and weirder than
anybody else's. But there was nothing she could remember dreaming -- by day or
night  --  that  came close to the sight that greeted her in The Great Head of
the Yebba Dim Day. It was a city, a city built from the litter of the sea. The
street  beneath  her  feet  was made from timbers that had clearly been in the
water  for  a  long  time,  and the walls were lined with barnacle - encrusted
stone.There  were  three  columns supporting the roof, made of coral fragments
cemented  together.They  were  buzzing  hives  of  life unto themselves; their
elaborately constructed walls pierced with dozens of windows, from which light
poured.
```

C++ code to print output without using text_printf:

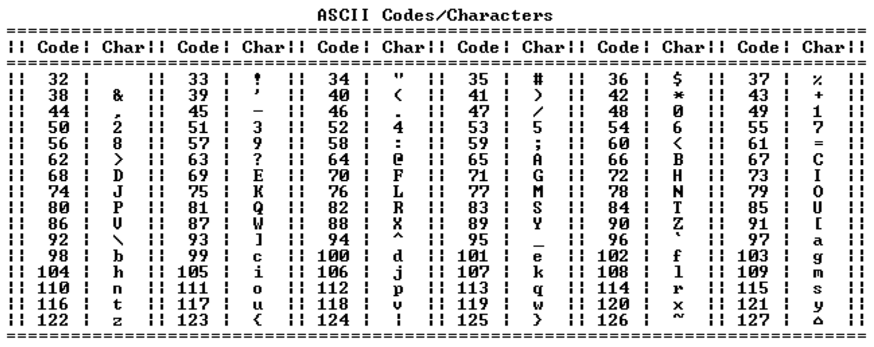
```cpp
const int pageWidth = 78;
typedef std::list<std::string> WordList;

WordList splitTextIntoWords(const std::string &text)
{
  WordList words;
  std::istringstream in(text);
  std::copy(std::istream_iterator<std::string>(in),
    std::istream_iterator<std::string>(),
    std::back_inserter(words));
  return words;
}

void justifyLine(std::string line)
{
  size_t pos = line.find_first_of(' ');
  if (pos != std::string::npos) {
    while (line.size() < pageWidth) {
      pos = line.find_first_not_of(' ', pos);
      line.insert(pos, " ");
      pos = line.find_first_of(' ', pos + 1);
      if (pos == std::string::npos) {
        pos = line.find_first_of(' ');
      }
    }
  }
  std::cout << line << std::endl;
}

void justifyText(const std::string &text)
{
  WordList words = splitTextIntoWords(text);

  std::string line;
  for (const std::string& word : words) {
    if (line.size() + word.size() + 1 > pageWidth) { // next word doesn't fit into the line.
      justifyLine(line);
      line.clear();
      line = word;
    } else {
      if (!line.empty()) {
        line.append(" ");
      }
      line.append(word);
    }
  }
  std::cout << line << std::endl;
}

justifyText(string(s1));
```

C++ code to print output using text_printf:

```cpp
text_printf("78[{w ' ' 1s' '} 1s' ']", &s1, NULL);
cout << endl;
```

Figure 7: Two versions of C++ code to print the paragraph of text shown (top): the first version (middle) does not use text_printf and instead defines some helper functions to print a block of justified text; the second version (bottom) uses a single text_printf call plus a manual newline insertion to achieve the same output.

Output:



```
                           ASCII Codes/Characters
================================================================================
¦¦ Code ¦ Char ¦¦ Code ¦ Char ¦¦ Code ¦ Char ¦¦ Code ¦ Char ¦¦ Code ¦ Char ¦¦ Code ¦ Char ¦¦
================================================================================
¦¦  32  ¦      ¦¦  33  ¦  !   ¦¦  34  ¦  "   ¦¦  35  ¦  #   ¦¦  36  ¦  $   ¦¦  37  ¦  %   ¦¦
¦¦  38  ¦  &   ¦¦  39  ¦  '   ¦¦  40  ¦  (   ¦¦  41  ¦  )   ¦¦  42  ¦  *   ¦¦  43  ¦  +   ¦¦
¦¦  44  ¦  ,   ¦¦  45  ¦  -   ¦¦  46  ¦  .   ¦¦  47  ¦  /   ¦¦  48  ¦  0   ¦¦  49  ¦  1   ¦¦
¦¦  50  ¦  2   ¦¦  51  ¦  3   ¦¦  52  ¦  4   ¦¦  53  ¦  5   ¦¦  54  ¦  6   ¦¦  55  ¦  7   ¦¦
¦¦  56  ¦  8   ¦¦  57  ¦  9   ¦¦  58  ¦  :   ¦¦  59  ¦  ;   ¦¦  60  ¦  <   ¦¦  61  ¦  =   ¦¦
¦¦  62  ¦  >   ¦¦  63  ¦  ?   ¦¦  64  ¦  @   ¦¦  65  ¦  A   ¦¦  66  ¦  B   ¦¦  67  ¦  C   ¦¦
¦¦  68  ¦  D   ¦¦  69  ¦  E   ¦¦  70  ¦  F   ¦¦  71  ¦  G   ¦¦  72  ¦  H   ¦¦  73  ¦  I   ¦¦
¦¦  74  ¦  J   ¦¦  75  ¦  K   ¦¦  76  ¦  L   ¦¦  77  ¦  M   ¦¦  78  ¦  N   ¦¦  79  ¦  O   ¦¦
¦¦  80  ¦  P   ¦¦  81  ¦  Q   ¦¦  82  ¦  R   ¦¦  83  ¦  S   ¦¦  84  ¦  T   ¦¦  85  ¦  U   ¦¦
¦¦  86  ¦  V   ¦¦  87  ¦  W   ¦¦  88  ¦  X   ¦¦  89  ¦  Y   ¦¦  90  ¦  Z   ¦¦  91  ¦  [   ¦¦
¦¦  92  ¦  \   ¦¦  93  ¦  ]   ¦¦  94  ¦  ^   ¦¦  95  ¦  _   ¦¦  96  ¦  `   ¦¦  97  ¦  a   ¦¦
¦¦  98  ¦  b   ¦¦  99  ¦  c   ¦¦ 100  ¦  d   ¦¦ 101  ¦  e   ¦¦ 102  ¦  f   ¦¦ 103  ¦  g   ¦¦
¦¦ 104  ¦  h   ¦¦ 105  ¦  i   ¦¦ 106  ¦  j   ¦¦ 107  ¦  k   ¦¦ 108  ¦  l   ¦¦ 109  ¦  m   ¦¦
¦¦ 110  ¦  n   ¦¦ 111  ¦  o   ¦¦ 112  ¦  p   ¦¦ 113  ¦  q   ¦¦ 114  ¦  r   ¦¦ 115  ¦  s   ¦¦
¦¦ 116  ¦  t   ¦¦ 117  ¦  u   ¦¦ 118  ¦  v   ¦¦ 119  ¦  w   ¦¦ 120  ¦  x   ¦¦ 121  ¦  y   ¦¦
¦¦ 122  ¦  z   ¦¦ 123  ¦  {   ¦¦ 124  ¦  ¦   ¦¦ 125  ¦  }   ¦¦ 126  ¦  ~   ¦¦ 127  ¦  ⌂   ¦¦
================================================================================
```

C++ code to print output using text_printf:

```
int totalWidth = 80;

// construct format strings for the headers row and the data rows.
string headersFormat = "%d[ '||' ";
string rowFormat = "%d[ '||' ";
for (int i = 0; i < 6; ++i) {
  headersFormat += "1s[1s' ' 'Code' 1s' '] '|' 1s[1s' ' 'Char' 1s' '] '||'";
  rowFormat += "1s[1s' ' '%d' 1s' '] '|' 1s[1s' ' '\\%c' 1s' '] '||'";
}
headersFormat += " ] '\n'";
rowFormat += " ] '\n'";

// print the table
text_printf("%d[1s' ' 'ASCII Codes/Characters' 1s' '] '\n'", NULL, NULL, totalWidth);
text_printf("%d'=' '\n'", NULL, totalWidth);
text_printf(headersFormat.c_str(), NULL, totalWidth);
text_printf("%d'=' '\n'", NULL, totalWidth);
for (int i = 32; i < 127; i+=6) {
  text_printf(rowFormat.c_str(), NULL, totalWidth,
    i, (char)i, i+1, (char)(i+1), i+2, (char)(i+2), i+3, (char)(i+3), i+4, (char)(i+4), i+5, (char)(i+5));
}
text_printf("%d'=' '\n'", NULL, totalWidth);
```

Figure 8: C++ code using text_printf (bottom) to print the ASCII table shown (top). Format strings for the row of column headers and for the data rows are built in a loop since they're impractically long and contain many repeating parts. Each output line is then printed one by one, with the data rows being printed in a loop.

5. Future Work

5.1 Constraints on printf format codes

As mentioned in section 2.5, the current system allows printf format codes to be inserted anywhere in the format string and expand to anything. While this provides maximum flexibility, it also allows the %s format code to expand to arbitrary characters, including tokens of the text_printf mini-language. There should be some restrictions placed on where format codes can be inserted and what they can expand to. For example, we may allow any format codes inside string literals, but only allow format codes outside string literals to appear in certain places. The %d format code could be restricted to only the places where a length is expected. The %s format code could be restricted to only expand to a string that corresponds to a valid AST in the text_printf mini-language (which is how metaprogramming Terra with Lua [12] is done).

5.2 More inter- filler options

Currently, text_printf allows inter-word fillers to be specified for words elements. A natural extension to this is to allow inter-line fillers to be specified, which would allow double-spaced paragraphs or "underlined" (using dashes) paragraphs. The ability to specify inter-letter fillers could also be added, though this may not have enough use cases to justify it.

5.3 Add warnings for dangerous format strings.

As mentioned in section 3, there are several steps in the pipeline where a total length needs to be distributed to both fixed-length and share-length elements. This process can result in error if:

- The sum of the lengths of the fixed-length elements exceeds the total length.
- There are no share-length elements to distribute the remaining length to after distributing to the fixed-length elements.

The first of these is straightforward to detect and will always result in an error. The second case is more interesting: it's actually fine to have no share-length elements if there's no remaining length to distribute. In other words, if the sum of the length of the fixed-length elements happens to equal the total length being distributed, then it's perfectly valid to have no share-length elements. However, format strings that lead to these situations should be considered "dangerous" since it's likely they're only avoiding the error out of luck. The system should be able to detect such format strings and warn the programmer. The following are two examples of what the system should consider as dangerous format strings:

```
6[3'-' 3'+']

80[{w ' ' 1s' '}]
```

The first format string is a block that contains no share-length elements, which usually results in an error since a block needs share-length elements to distribute any length not taken by fixed-length elements. In this case, it just happens that the sum of the fixed-length elements is exactly equal to the length of the block, so there isn't any remaining length to distribute. Currently, this format string would not result in an error, but this type of format string should be avoided. The second format string is one that sometimes causes an error, and the reason why was explained in the justified text example at the end of section 2.3: if one of the output lines happens to end up with only one word, then no inter-word fillers will be inserted in that line, which means that line will have no share-length content to distribute any of the remaining block length to. Such a format string will probably work most of the time but will result in an error if given an "unlucky" word source as input. This is undesirable and again should be avoided. Ideally, the system would detect format strings like these two examples and warn the user.

5.4 Stream/object-based version

As scenario 4 in the results section demonstrated, it's common to come across situations where the necessary format string is impractically long. In these cases, it would be convenient to have a stream version of text_printf that uses flags, like how C++'s cout stream can build up a formatted output using the flags std::left, std::right [8], std::setw [10], and std::setfill [9]. Alternatively, an object-based version of text_printf would also help, where the basic elements such as fillers, blocks, and words are each represented by classes, and instances of those classes are used to build up a text_printf format object.

6. Conclusion

We've described a system for formatting output text in C++ that's similar in usage as printf. The mini-language of its format string is higher level than that of printf, and as a result, text_printf is able to specify more complex formats. We developed the mini-language concepts of blocks, fillers, and share-lengths, which allow output elements to be grouped together and have their collective widths and alignments easily specified. These groups can themselves be grouped, which in turn allows as many levels of grouping in the format specification as the programmer desires. We developed the concept of words elements, which combined with blocks provides a straightforward way to describe the format for paragraphs of text. We showed how different horizontal and vertical alignments of the text are specified in the format string. Next, we demonstrated the use of printf's format codes in the text_printf format string, giving it the ability to change based on data variables and allowing them to be built up from multiple smaller format strings. Finally, we compared versions of C++ code with and without text_printf that produce a specific formatted output. This comparison was performed for three actual scenarios from questions posed by programmers online. In all scenarios, the text_printf version of the code was shorter and simpler.

References

[1] A. Allain. Formatting Cout Output in C++ using iomanip. http://www.cprogramming.com/tutorial/iomanip.html

[2] Center text in fixed-width field with stream manipulators in C++. 2013. http://stackoverflow.com/questions/14861018/center-text-in-fixed-width-field-with-stream-manipulators-in-c

[3] Formatting Numeric Print Output. https://docs.oracle.com/javase/tutorial/java/data/numberformat.html

[4] How to print "justified" text in the console using modern C++. 2014. http://stackoverflow.com/questions/22983008/how-to-print-justified-text-in-the-console-using-modern-c

[5] Input and Output. https://docs.python.org/2/tutorial/inputoutput.html

[6] printf. http://www.cplusplus.com/reference/cstdio/printf/

[7] Problem with <iomanip> alignment. 2011. http://www.cplusplus.com/forum/beginner/38488/

[8] std::left, std::right, std::internal. 2015. http://en.cppreference.com/w/cpp/io/manip/left

[9] std::setfill. 2014. http://en.cppreference.com/w/cpp/io/manip/setfill

[10] std::setw. 2015. http://en.cppreference.com/w/cpp/io/manip/setw

[11] Table format? 2012. http://www.cplusplus.com/forum/beginner/84082/

[12] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A Multi-Stage Language for High-Performance Computing. 2013. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13).