Overview of C++ `strings`

References:

The material in this handout is collected from the following references:

- Sections 3.2 of the text book C++ Primer.
- Chapter 13 of The C++ Standard Library.

Introduction

Next to numbers, strings are the most important data type used by programs. The term *string* generally means an ordered sequence of characters, with a first character, a second character, and so on. When we talk about strings in C++, we must be careful because C++ can deal with strings just like C and in addition the C++ standard library has the `std::string` type to provide more convenience and flexibility when dealing with strings.

C programmers represent a string as a `char` array with the array element after the last character represented as a *null character*. A null character is a character with numeric value of zero and is represented by character constant `'\0'`. C++ can also represent strings using null-terminated `char` arrays. However, since the C++ standard library provides a more convenient type to represent strings, we'll use the term *C-string* to denote this inherited (from C) representation of strings as a null-terminated array of `char`s. C++ programmers can continue to use the C standard library C-string handling functions such as `strlen`, `strcpy`, `strcat`, `strcmp` by including header file `<cstring>` and using standard namespace `std`. In addition, class `std::istream` overloads operator `>>` to extract C-strings from the standard input stream. Class `std::ostream` overloads operator `<<` operator to insert C-strings into the standard input stream.

```cpp
#include <cstring>
char pest[25] {"Bart"};
std::strcat(pest, " Simpson");
std::cout << pest << " is a secret genius!!!";
```

Note that the null character may very well not be the very last character in a C-string array, but it will be the first character beyond the last character of the actual string data in that array. For example in `char` array `pest` having size 25, the characters of word `"Bart"` will be in elements with subscripts 0 to 3 with the null character at subscript 4, and elements with subscripts 5 to 24 zeroed-out. In any case, it's the null character at subscript 4 that makes the otherwise ordinary character array `pest` a C-string.

C denotes a sequence of zero or more characters enclosed in double quotes, as in `"digipen"`, or `"Bart"`, or `"32467"` as a *string literal*. Each character in the string takes up one byte with the compiler automatically appending a null character to designate the end of the string. So the physical storage required for a string literal is one more than the number of characters written between the double quotes. String literals have type *array of* `char` in C and type *array of* `const char` in C++.

> ***String literals have type array of `char const` in C++ and type array of `char` in C. That is, string literal `"Hello"` has type `char const [6]` in C++ and type `char [6]` in C.***

String literals are often used in output statements when we wish to display text on the screen for the benefit of our users:

```
printf("Enter file name: ");      // in C
std::cout << "Enter file name: "; // in C++
```

Since C++ inherited string literals from C but doesn't provide other representations for string literals, we'll continue to use the term *string literal* to represent zero or more characters enclosed in double quotes.

The C++ standard library provides a flexible and convenient way to represent a character string consisting of `char`s using an object of class `std::string` that is defined in header `<string>`. We'll often restrict the meaning of the term *string* to refer to objects of type `std::string`.

```
#include <string>
std::string pest {"Bart"};
pest += " Simpson"; // or, pest += pest + " Simpson";
std::cout << pest << " is a secret genius!!!";
```

As a C++ programmer, then, you must distinguish between the following four things:

1. An *ordinary* array of `char`s, which is just like any other array and has no special properties that other arrays do not have.
2. A *string literal* which is a sequence of zero or more characters enclosed in double quotes with type *array of* `const char`. Since C++ converts type *array of* `type` to an expression with type *pointer to* `type` that points to the initial element of the array, string literal `"Hello"` will evaluate to expression with type `char const*` that points to the memory location where the first character `'H'` of the string literal is given storage.
3. A *C-string* consisting of an array of `char`s terminated by null character `'\0'`, and which therefore is different from an ordinary array of characters. The presence of the null character is made use of by the string handling functions presented in `<cstring>`.
4. A C++ *string* object, which is an instance of a class data type `std::string` whose actual internal representation you need not know or care about, as long as you know what you can and can't do with objects (that is, variables) of this type. There are many things programmers can do with objects of type `std::string` and this rich interface of functions is presented in `<string>`.

Both the C-string library functions presented in `<cstring>` and the C++ string library functions presented in class `std::string` and defined in `<string>` are available to C++ programs. But, don't forget that these are two *different* function libraries, and the functions of the first library have a different notion of what a string is from the corresponding notion held by the functions of the second library. There are two further complicating aspects to this situation:

- Though a function from one of the libraries may have a counterpart in the other library (that is, a function in the other library designed to perform the same operation), the functions may not be used in the same way, and may not even have the same name;
- Because of backward compatibility many functions from the C++ string library can be expected to work fine and do the expected thing with C-style strings, but not the other way around.

The last statement above might seem to suggest we should use C++ strings and forget about C-strings altogether, and it is certainly true that there is a wider variety of more intuitive operations available for C++ strings. Dealing with text in C is an error-prone endeavor since C-strings and string literals are implemented using arrays and are susceptible to the same problems as arrays:

out-of-bounds accesses, accessing arrays through incorrect or uninitialized pointers, forgetting to return allocated memory back to the heap, and leaving dangling pointers after the array has been deallocated. Using C-strings puts a tremendous burden on the programmer to manually locate storage space for `char` sequences. A common error is moving a C-string into a smaller array. For efficiency's sake, there is no check against this possibility, and it is all too easy for the inexperienced programmer to corrupt adjacent variables. C++ strings handle all these chores completely automatically. They're designed to behave as if they were a kind of fundamental data type that is easy to use correctly and hard to use incorrectly. Since the physical representation of data is hidden from programmers, you can copy, assign, and compare strings as easily as fundamental types without dealing with arrays, pointers, pointer arithmetic, nor worrying about whether there is enough memory or how long the internal memory is valid. You simply use operators, such as assignment using operator `=` rather than function `strcpy`, comparison using operators `==`, or `>`, or `<` rather than function `strcmp`, and concatenation using operators `+` or `+=` rather than function `strcat`. In short, C++ strings enable you to deal with strings just as you would deal with values of arithmetic types such as `int` or `double`.

Modern data processing is mostly string processing and since C-strings are a source of bugs and trouble, programmers coming from C must quickly learn about and gain experience in C++ strings. However, since C-strings are more primitive, you may therefore find them more efficient in certain situations (such as embedded systems) as long as you remember a few simple rules relating to arrays, pointers, null character, and memory storage space. In addition, if you read other, older programs you will see lots of C-strings. Be extra careful when you occasionally need to mix C- and C++ strings. Finally, there are certain situations in which C-strings *must* be used and we'll come across such use cases throughout the semester. Otherwise, in all other cases, just stick to C++ strings.

Defining and initializing C++ strings

A C++ string is an *object* of class `string`, which is defined in header `<string>` and is part of `std` namespace. A *class* is a user-defined data type. Recall that a *data type* specifies a set of values and a set of operations on those values. A built-in type like the `int` type has addition, subtraction, and other arithmetic functions defined by operators built into the compiler. Likewise, a class defines a valid set of data values and a set of operations that can be used on them. In general, C++ programmers use built-in and user-defined types in much the same manner without distinguishing between them. The only difference is that built-in types and their operations are directly implemented by the compiler and directly map to how machines store and manipulate data. In contrast, classes are implemented by programmers using C++ code and reflect abstractions of entities existing in the world being simulated.

In class terminology, functions that perform the tasks of defining and initializing objects are called *constructor functions*, or *constructors*, for short. A major source of bugs in C is the use of uninitialized variables that will generate intermittent, undefined behavior. Constructors are C++'s way of always initializing objects of user-defined types to a safe initial state. The `string` class has several constructors that may be explicitly or implicitly called to create and initialize a `string` object. The following examples list the constructors provided by the `string` class for defining and initializing a string object. The examples require the inclusion of `<string>` header file.

Default construction

Constructors initialize objects. A default constructor is a constructor that can be called with no arguments. So a default constructor is the C++ way of initializing an object without any information from the place where the object is being created. This only makes sense if the object can be initialized to a safe state. Default construction for string objects means to initialize the string object to encapsulate the empty string `""` having zero length. This constructor has the forms `string str_obj` and `string str_obj {}`. The following code fragment illustrates these two syntaxes for default construction of string objects:

```
std::string msg;     // default construction to a safe state
std::string msg2 {}; // same as above
```

In contrast, notice how the elements of an uninitialized `char` array will take up junk values and reading such values can cause undefined behavior:

```
char cstr[10]; // uninitialized array elements
std::cout << cstr; // prints junk
```

String literal and C-string construction

The purpose of the next set of constructors is to initialize string objects with a string literal and they have the general form `string str_obj = str_literal` or `string str_obj(str_literal)` or `string str_obj {str_literal}`. Some examples of initializing C++ string objects using string literals or C-strings:

```
std::string s1 = "Hello";  // this is called copy initialization
std::string s2("World");   // this is called direct initialization
string s3 {"Hello World"}; // this is called list initialization
```

C++ string objects can also be initialized using a pointer to [the first element of a] C-string using the general form `string str_obj = cstr_obj` or `string str_obj(cstr_obj)` or `string str_obj {cstr_obj}`:

```
char const *ps {"Hello"}; // define and initialize a C-string variable
std::string s4 = ps; // this is called copy initialization
std::string s5(ps);  // this is direct initialization
std::string s6 {ps}; // this is called list initialization
```

We can also initialize string objects with the substring of a string literal or C-string. This type of constructor has the general form `string(str_literal, n)` or `string(cstr_obj, n)` where a string object is initialized with the first `n` characters of string literal `str_literal` or C-string pointed to by `cstr_obj`:

```
std::string s7("Bart Simpson", 4);  // s7 encapsulates "Bart"
std::string s8 {"Bart Simpson", 4}; // s8 encapsulates "Bart"
char const *ps1 {"Hello World"};
std::string s9 {ps1, 5};            // s9 encapsulates "Hello"
std::string s10(ps1, 5);            // s10 encapsulates "Hello"
```

Copy construction

Copy construction is used to define and initialize a string object to a value that is a previously declared string object with the general form `string str_obj = a_str_obj` or `string str_obj(a_str_obj)` or `string str_obj {a_str_obj}`:

```cpp
std::string s0 {"Bart"}; // construct string object using string literal
std::string s1(s0);  // direct initialization so that s1 is copy of s0
std::string s2 = s0; // copy initialization so that s2 is copy of s0
std::string s3 {s0}; // list initialization so that s3 is copy of s0
```

Substring copy construction

This constructer has the general form `string name(str_obj, pos, n)` and it defines and initializes a string object using a substring starting at subscript `pos` of `str_obj` and containing `n` characters. The third argument has a default parameter so that if `n` is not specified, then it means *until end of string* abstracted by `str_obj`:

```cpp
std::string s0 {"Bart Simpson"}; // s0 encapsulates "Bart Simpson"
std::string s1(s0, 5, 3);  // s1 encapsulates "Sim"
std::string s2 {s0, 5, 3}; // s2 encapsulates "Sim"
std::string s3(s0, 5);  // s3 encapsulates "Simpson"
std::string s4 {s0, 5}; // s4 encapsulates "Simpson"
std::string s5(s0, 0);  // s5 encapsulates "Bart Simpson"
std::string s6 {s0, 0}; // s6 encapsulates "Bart Simpson"
```

Fill construction

This method has the general form `string str_obj(n, ch)` to define and initialize a string object `str_obj` with `n` copies of character `ch`:

```cpp
std::string s1(3, '+'); // s1 encapsulates "+++"
std::string s2(5, 'z'); // s2 encapsulates "zzzzz"
```

Recommended way of initializing `string`s

The following table lists the recommended way of initializing string objects described above:

Constructor	Examples
`string name;`	`std::string s0;` or `std::string s0 {};`

`string name(str_literal);`
`string name(str_literal, n);`
	`std::string s1 {"Bart Simpson"};`
	`std::string s2("Bart Simpson", 4); // "Bart"`

`string name(cstr_variable);`
`string name(cstr_variable, n);`
	`char const *ps {"Hello World"};`
	`std::string s3 {ps};`
	`std::string s4(ps, 5); // "Hello"`

`string name(str_obj);`	`std::string s5 {s1}; // "Bart Simpson"`
`string name(str_obj, pos, n);`	`std::string s6(s5, 5, 3); // "Sim"`
`string name(n, ch);`	`std::string s7(3, '+'); // "+++"`

How are strings stored?

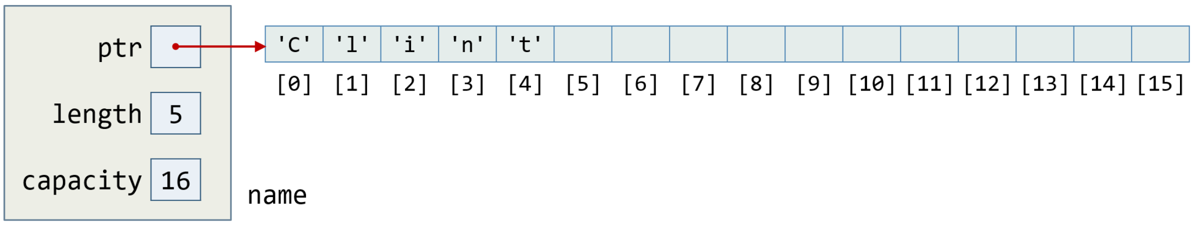
A possible definition of class `std::string` could be something like this:

```cpp
class string {
  char *ptr;
  size_t length;
  size_t capacity;
};
```

Consider the definition of a `string` object:

```cpp
std::string name {"Clint"};
```

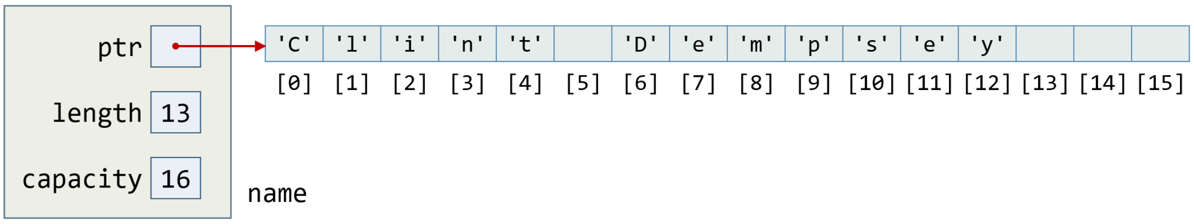
The memory layout of object `name` will be something like this:



`name` is a `string` object with several data members. Data member `ptr` is a pointer to the first byte in a dynamically allocated array of `char`s. The data member `length` contains the number of characters or the length of the string. To avoid frequent and costly heap memory allocations and deallocations that arise whenever the encapsulated string is altered, the authors of class `string` have chosen to allocate more bytes than required to store the characters comprising the string `"Clint"`. Data member `capacity` keeps track of the number of valid characters that may be currently stored by the dynamically allocated array. For example, the following statement changes the string encapsulated by object `name` from `"Clint"` to `"Clint Dempsey"`:

```cpp
name += " Dempsey";
```

However, because the current capacity of the container can store the updated string, a memory allocation to store the new string followed by a deallocation of the old string is not necessary.



Also notice that unlike string literals and C-style strings, this example implementation of class `string` doesn't null terminate the sequence of characters `'C'`, `'l'`, `'i'`, `'n'`, `'t'`.

> ***The C++ standard doesn't require `string` to provide special meaning for character `'\0'` - it may be a part of the `string` just like any other character. But when queried for one-past the last character, C++11 requires the `string` object to return `'\0'`.***

String input and output

In addition to a string being initialized with the constructors listed in the earlier section, strings can be input from the keyboard and displayed onscreen. The following table lists the basic functions and objects for input and output of string values:

C++ classes or function	Description
`std::istream`	Extract word of characters from an input stream
`std::ostream`	Insert characters encapsulated by string into output stream
`std::getline(std::istream, string_object)`	Extract all characters from an input stream up to but not including newline `'\n'` and stores them in the string object

Class `std::istream` overloads bitwise right-shift operator `>>` to *extract* a *word* of characters from an input stream. Likewise, class `std::ostream` overloads bitwise left-shift operator `<<` to *insert* characters encapsulated by string into an output stream.

```cpp
std::cout << "Enter name: "
std::string name;
std::cin >> name;
std::cout << "Hello " << name << '\n';
```

As in the case of other types, strings can be positioned within output fields (declared in `<iomanip>`)

```cpp
std::string s("Spring flowers ");
std::cout << std::left           // left-aligned
          << std::setfill('?')   // fill character ?
          << std::setw(30) << s ; // field width 30
```

This example outputs `Spring flowers ???????????????`. Manipulator `std::right` can be used to right-justify the output within the field:

```cpp
std::string s("Spring flowers ");
std::cout << std::right          // right-aligned
          << std::setfill('?')   // fill character ?
          << std::setw(30) << s ; // field width 30
```

and will output `???????????????Spring flowers`. Note that there is a space after the terminating character `s` but the document format is unable to show it correctly.

Reading line of text using `getline`

Function `operator>>` reads a string word-wise. That is, when a string is being read, leading whitespace is skipped by default, and the string is read until another whitespace character or end-of-file is encountered. Consider the following code fragment:

```cpp
std::cout << "Enter name: "
std::string name;
std::cin >> name;
std::cout << "Hello " << name << '\n';
```

and suppose the user types

```
Clint Dempsey
```

as the response to the prompt, then only `Clint` is stored in `name`. To read the second string, another input statement must be used. This constraint makes it tricky to write an input statement that deals properly with user responses. Some users might type just their first names, others might type their first and last names, and others might even supply their middle initials. To handle such a situation, use function `getline` in namespace `std` with the declaration:

```
std::istream& getline(std::istream& is, std::string& str);
```

Replacing the statement on line 3 in the above code fragment with this statement

```
getline(std::cin, name);
```

reads all keystrokes until the Enter key is pressed which creates a newline character `\n`, makes a string containing all of the keystrokes except the `\n`, and encapsulates the string into object `name`. With the preceding input example, `name` will now encapsulate string `"Clint Dempsey"`. This is a string containing 13 characters, one of which is a space. You should always use function `getline` if you are not sure that the user input consists of a single word.

The following program shows the use of function `getline` and standard output stream object `std::cout` to input and output a string containing spaces that's entered at the user's terminal:

```
#include <iostream>
#include <string>
int main() {
  std::cout << "Enter a greeting: ";
  std::string greeting;
  std::getline(std::cin, greeting);
  std::cout << "Your greeting is: " << greeting << "\n";
}
```

The following is a sample run of the program:

```
Enter a greeting: Testing to see if whitespace is read!!!
Your greeting is: Testing to see if whitespace is read!!!
```

Function `getline` can be used to implement a simple file copy program `fc.out`:

```
#include <iostream>
#include <string>

int main() {
  std::string line;
  while (getline(std::cin, line)) {
    std::cout << line << "\n";
  }
}
```

To make a copy of file `lines.txt` called `lines-copy.txt`, you'd run the program like this:

```
$ ./fc.out < lines.txt > lines-copy.txt
```

A more general overload of function `getline` that allows programmers to specify a line delimiter other than `'\n'`:

```
std::istream& getline(std::istream& is, std::string& str, char delim);
```

`str` is the name of a string object and `delim` is a character constant or variable specifying the terminating character for a line of text. For example, the expression `getline(cin, name, '!')` accepts all characters entered at the keyboard, including a newline character, until an exclamation point is entered. The exclamation point isn't stored as part of the string. If the optional third argument, `delim`, is omitted when `getline` is called, the default terminating character is the newline `'\n'` character. Therefore, the statement `getline(cin, name, '\n');` can be used in place of the statement `getline(cin, name);`. The following code fragment extract a *word* from a line of text:

```
#include <iostream>
#include <string>

int main() {
  int word_cnt {0};
  std::string line;
  while (getline(std::cin, line, ' ')) {
    std::cout << line << "\n";
    ++word_cnt;
  }
  std::cout << "word count: " << word_cnt << "\n";
}
```

When you run the program and type the line `today is a good line` followed by a newline, the program will print an individual word on a newline followed by the word count.

Reading an unknown number of `string`s

During the discussion on streams, we learnt an useful idiom for recognizing the end of valid values in an input stream and for recognizing the end of the input file that the stream represents. The idiom was illustrated using the following program that reads an unknown of integer values from standard input and computes the average of these unknown count of values:

```
#include <iostream>
#include <string>

int main() {
  int sum = 0, count = 0, value;
  // continue reading integers until non-integer value is encountered
  while (std::cin >> value) {
    sum += value; // accumulate sum of integer values
    ++count;      // keep track of how many integer values were encountered
  }
  double average = static_cast<double>(sum)/count;
  std::cout << "Average of integer values: " << average << '\n';
}
```

Class `std::istream` overloads function `operator>>` for a variety of built-in types. The declaration of the function to extract values of type `int` looks like this:

```cpp
std::istream& operator>> (int& val);
```

Start by remembering that expression `std::cin >> value` evaluates to a reference to `std::cin`, so that asking for the value of `std::cin >> value` in the `while` statement's condition is equivalent to executing `std::cin >> value` and then asking for the value of `std::cin`. Because `std::cin` has type `std::istream`, we must look to the definition of class `std::istream` for the meaning of `while (cin)`. The details of the member function that implements this behavior is complicated enough that this discussion is delayed until we study class design later in the course. For now, what we need to know is that the `std::istream` class provides a conversion function that can be used to convert `std::cin` into a `bool` value that can be used in a conditional expression. Whether the `bool` value is `true` or `false` depends on the internal state of the `std::istream` object, which will remember whether the last attempt to read worked. Thus, using `std::cin` as a condition is equivalent to testing whether the last attempt to read from `std::cin` was successful.

You can signal to the program to terminate the `while` loop by providing a non-integer value as input. Why? Because (as explained earlier), the `while` statement's condition `std::cin >> value` will evaluate to `std::cin` which in turn will evaluate to `bool` value `false` since `std::cin` will be in a failed state (because the input stream contains a character that is not compatible with an integer causing the read to fail).

Recall that you could terminate the `while` loop by

- Explicitly providing a non-integer value as input.
- By using the $\mathrm{CTRL+D}$ combo to simulate EOF (end-of-file) condition to signal the program that no further input is to be expected.
- Collecting integer values to be processed in a text file and redirecting the contents of this file to the input file stream with input indirection symbol `<`. After reading all the integer values in the file, EOF is signaled to the program.

Can a similar program be authored to read an unknown number of `string`s? First, notice that operator>> is overloaded to extract a `string` value:

```cpp
std::istream& operator>> (std::istream& is, std::string& str);
```

Just like the `int` overload, the `string` overload also returns the `istream` object that invoked the function. This means we can the same idiom for reading unknown number of `int`s to read an unknown number of `string`s. The following program `words.cpp` reads an unknown number of `string`s and returns a count of the number of words that were read:

```
#include <iostream>
#include <string>
int main() {
  int count = 0;
  std::string word;
  // continue reading words until EOF is encountered
  while (std::cin >> word) {
    ++count; // keep track of how many words were encountered
  }
  std::cout << "Count of words: " << count << '\n';
}
```

The following is a sample run of the program `words.out`:

```
$ ./words.out
today is a good day
tomorrow will be a better day
CTRL+D
Count of words: 11
$
```

Since a `string` object can encapsulate any character, the only way to terminate the program is to type the $\mathrm{CTRL}+\mathrm{D}$ combo to simulate EOF or type the input provided to the program in a text and redirect the file contents to the standard input stream.

Using `getline` to read unknown number of lines

Notice that the overload of operator>> for `std::string` uses whitespaces as separators to extract `string`s. Therefore, the function can only extract words from the stream. We've earlier studied function `getline` to read entire lines of text. This function has the declaration:

```
std::istream& getline (std::istream& is, std::string& str);
```

Notice that this function returns a reference to the `istream` object from which the characters are extracted. This means expression `getline(std::cin, str)` where `str` is of type `std::string` evaluates to `std::cin`. In a `while` statement `while(getline(std::cin, str))`, asking for the value of `getline(std::cin, str)` in the `while` statement's condition is equivalent to asking for the value of `std::cin`. All of this means that the idiom for reading an unknown number of `string`s can be adopted to read an unknown number of lines using function `getline`. The following program `lines.cpp` reads an unknown number of lines and returns a count of the number of lines that were read:

```cpp
#include <iostream>
#include <string>

int main() {
  int count = 0;
  std::string line;
  // continue reading lines until EOF is encountered
  while (std::getline(std::cin, line)) {
    ++count; // keep track of how many newlines were encountered
  }
  std::cout << "Count of lines: " << count << '\n';
}
```

The following is a sample run of the program:

```
$ ./words.out
today is a good day
tomorrow will be a better day
CTRL+D
Count of lines: 2
$
```

Mixing `>>` and `getline` input

It is tricky to mix `>>` and `getline` input. Consider the problem of reading information from the standard input stream about a typhoon consisting of (in this order) the name (for example, Maria), maximum wave height (for example, 9.1 meters), number of fatalities (for example, 5 deaths), and the location (for example, Ryuku Islands):

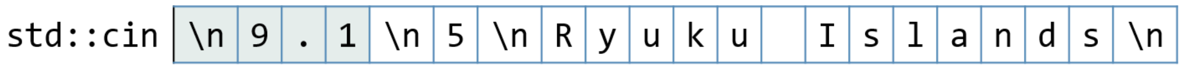
```cpp
 1  std::string name, location;
 2  double wave_ht;
 3  int fatalities;
 4  std::cout << "Enter tsunami name: ";
 5  std::cin >> name;
 6  std::cout << "Enter maximum wave height: ";
 7  std::cin >> wave_ht;
 8  std::cout << "Enter fatalities: ";
 9  std::cin >> fatalities;
10  std::cout << "Enter tsunami location: ";
11  getline(std::cin, location);
```

Typhoons are given names formed from a single word. Therefore, the `operator>>` function on line 5 is appropriate for reading the typhoon's name. This function will read the name until it reaches a newline. It does not consume the newline. This is a problem when a call to `getline` immediately follows a call to `operator>>`. For example, the call to `getline` on line 11 reads only the newline, considering it as the end of an empty line of text.

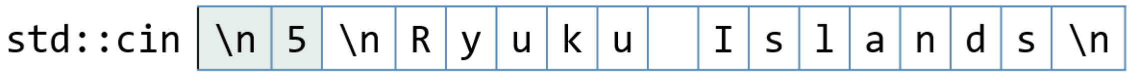
Perhaps an example will make this clearer. Consider the text entered by the user. Calling function `operator>>` on line 5 will consume the following colored characters.

std::cin	M	a	r	i	a	\n	9	.	1	\n	5	\n	R	y	u	k	u		I	s	l	a	n	d	s	\n

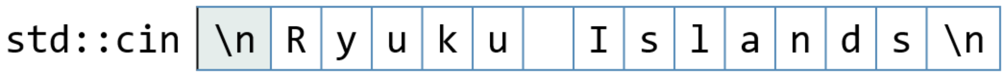
After the call to `operator>>` on line 5, a second call is made to `operator>>` on line 7 to read all characters involving a floating-point value. The initial newline `\n` character is consumed, characters `9`, `.`, and `1` are read until the newline character is encountered. The newline character is read but not consumed. The colored characters are the actual characters consumed and removed from the input stream.



A third call to `operator>>` is made on line 9 to read the number of fatalities caused by the typhoon. The color characters are consumed. This call will consume the initial newline `\n` character followed by character `5` until the newline character is encountered. The newline character is read but not consumed.



This call is followed by a call to function `getline` on line 11 which will read only the newline and set `location` to an empty string. We're required to use function `getline` rather than `operator>>` for the typhoon's location because the location may consist of multiple words, as in Ryuku Islands.



This is a problem whenever an input with `operator>>` is followed by a call to `getline`. Two solutions exist to skip the newline and have `getline` read the typhoon's location. The first solution is to use the `ignore` member function of class `std::istream`. The following statements must be inserted after the call to `operator>>` on line 9 and before the call to `getline` on line 11:

```
std::cin.ignore(1, '\n');
```

The second solution is to accept the newline represented by the Enter key and then discard it. This purpose is achieved by the following statements, which must be inserted after the call to `operator>>` on line 9 and before the call to `getline` on line 11:

```
std::string newline;
getline(std::cin, newline); // consume newline character
// now you're ready to call getline again
```

String operations

`string` is a user-defined class type composed of built-in types, possibly other user-defined types, and functions. The parts used to define the class are called *members*. Members can be of various kinds. Most are either *data members* which define the representation of an object of the class, or *member functions* (also known as *methods*) which provide the operations on such objects. Usually, we think of a class as having an *interface* plus an *implementation*. The interface specifies *what* objects instantiated from the class can do. It declares the identifiers, types, and functions that are available to users of these objects. An implementation specifies *how* these objects accomplish the purpose advertised by their class interface. Users should not see any part of the internal implementation; instead, they'll interact with an object solely through its class interface.

Our objective of using a user-defined type such as `string` is that we don't care how it has been implemented; instead, we're only interested in doing useful work only by accessing its member functions through its interface. We access a member function using the `object.member` notation.

Size

The `string` class has the `empty`, `size`, and `length` member functions. The `empty` member function returns a `bool` indicating whether the `string` object is empty. Both `size` and `length` member functions return the number of characters in the string that an object encapsulates. For example:

```
std::string s0; // empty string
std::cout << "s0 is " << (s0.empty() ? "" : "not ") << "empty string.\n";
std::string s1 {"Bart"};
std::cout << s1 << " has " << s1.length() << " characters.\n";
s1 += " Simpson";
std::cout << s1 << " has " << s1.size() << " characters.\n";
```

The `string::size_type` type

Both `size` and `length` member functions return a `string::size_type` value. What is this type? The `string` class - and most other library types - defines several companion types so that the library types can be used in a machine-independent manner. The type `size_type` is one of these companion types. It is unsigned type `size_t` [declared in the C++ standard library header `<cstddef>`] which is a type big enough to hold the size of any `string`. Any variable used to store the result from the `size` or `length` should be of type `string::size_type`.

Because `size` returns an unsigned type, it is essential to remember that expressions that mix signed and unsigned data can have surprising results. For example, if `n` is an `int` that holds a negative value, then `s.size() < n` will almost surely evaluate as `true`. It yields `true` because the negative value in `n` will convert to a large unsigned value.

> ***You can avoid problems due to conversion between `unsigned` and `int` by not using `int`s in expressions involving member functions that return a value of `size_type`. Such functions include `size`, `length`, and search functions including the name `find`.***

Element access

Because the `string` class overloads subscript operator `[]`, you can access characters in a `string` object in the same way that you access array elements. A `string` knows its size, so we can print the elements of a `string` like this:

```
std::string s {"Lisa Simpson"};
for (std::string::size_type i{0}; i < s.length(); ++i) {
  std::cout << s[i];
}
```

The call `s.length()` gives the number of elements of `string` variable `s`. In general, `s.length()` gives us the ability to access elements of a `string` without accidentally referring to an element outside the `string`'s range. The range of elements for `s` is $[$`0:s.length()`$)$. That's the mathematical notation for a half-open sequence of elements. The first character of the string encapsulated by `s` is `s[0]` and the last character is `s[s.length()-1]`. If `s.length()==0`, `s` has no elements, that is, `s` is an empty `string`. This notion of half-open sequences is used throughout C++ and the C++ standard library.

Iterators

In the previous code fragment, we sequentially accessed the elements of the string encapsulated by object `s`. When we write the expression `s[i]` to access an element of `s`, we're implicitly saying that we might access `s`'s elements in any order, not just sequentially. The reason we care about the sequence in which we access elements of a `string` is that different types have different performance characteristics and support different operation. If we know that our program uses only those operations that a particular type supports efficiently, then we can make our program more efficient by using that type. In other words, because the above code fragment requires only sequential access, we don't need to use indices, which provide the ability to access any element randomly. Instead, we'd like to rewrite the code fragment so as to restrict access to the elements to operations that only support sequential access. To that end, the C++ standard library supplies an assortment of types called iterators, which allow access to data structures in ways that the library can control. This control lets the library ensure efficient implementation.

We know that we access the elements of `s` sequentially, because we access those elements only by using `i` as an index, and the only operations we ever perform on `i` are to read it in order to compare it with the size of the `string`, and to increment it:

```
std::string s {"Lisa Simpson"};
for (std::string::size_type i{0}; i < s.length(); ++i) {
  std::cout << s[i];
}
```

From this usage, it is clear that we use `i` only sequentially. Unfortunately, even though we know this fact, the compiler has no way to know it. By using iterators instead of indices, we can make that knowledge available to the compiler. An ***iterator*** is a value that

- Identifies a container type such as `string` and an element in the container
- Lets us examine the value stored in that element
- Provides operations for moving between elements in the container
- Restricts the available operations in ways that correspond to what the type can handle efficiently

Because iterators behave analogously to indices, we can often rewrite code that uses indices to make them use iterators instead. As an example, we can rewrite the above code fragment by replacing the index `i` with an iterator:

```
std::string s {"Lisa Simpson"};
for (std::string::const_iterator it{s.begin()}; it != s.end(); ++it) {
  std::cout << *it;
}
```

There's a lot going on in this rewrite, so let's pick it apart a bit at a time.

Every container data type, such as `string`, defines two associated iterator types: `type::const_iterator` and `type::iterator` where `type` is the container type such as `string`. When we want to use an iterator to change the values stored in the container, we use the `iterator` type:

```
std::string s {"Lisa is number 1!!!"};
for (std::string::iterator it{s.begin()}; it != s.end(); ++it) {
  *it = std::islower(*it) ? std::toupper(*it) : *it;
  std::cout << *it;
}
```

If we need only read access, then we use the `const_iterator` type:

```
std::string s {"Lisa Simpson"};
for (std::string::const_iterator it{s.begin()}; it != s.end(); ++it) {
  std::cout << *it;
}
```

Abstraction is selective ignorance. The details of what particular type an iterator has may be complicated, but we don't need to understand these details. All we need to know is how to refer to the iterator type, and what operation the iterator allows. We need to know the type so that we can create variables that are iterators. We don't need to know anything about that type's implementation. For example, our definition of `it`: `std::string::const_iterator it{s.begin()}` says that `it` is of type `std::string::iterator`. We don't know the actual type of `it` - that's an implementation detail of `string` - nor do we need to know. All we need to know is that `std::string` has a member named `iterator` that defines a type that we can use to obtain read-only access to elements of the `string`.

The other thing we need to know is that there is an automatic conversion from type `iterator` to type `const_iterator`. As we're about to learn, `s.begin()` returns an `iterator`, but we said that `it` is a `const_iterator`. In order to initialize `it` with the value of `s.begin()`, the implementation converts the `iterator` value into the corresponding `const_iterator`. This conversion is one way, meaning that we can convert an `iterator` to a `const_iterator` but not vice versa.

Having defined `it`, we set it to the value of `s.begin()` and every iteration we compare `it`s value with `s.end()`. The `begin` and `end` functions return a value that denotes the beginning and one past the end of the sequence of characters in `s`, respectively. What's useful to know is that `begin` and `end` functions return a value of the iterator type for `std::string`. Thus, `begin` returns a `std::string::iterator` positioned at the initial character of `s`, so `it` initially refers to the first character in `s`.

The condition in the `for` statement, `it != s.end()` checks whether we've reached the end of `s`. Recall that `end` returns a value that denotes (one past) the end of the sequence of characters in `s`. As with `begin`, the type of this value is `std::string::iterator`. We can compare two iterators, `const` or not, for inequality (or equality). If `it` is equal to the value returned by `s.end()`, then we're through.

The last expression in the `for` statement, `++it`, increments the iterator so that it refers to the next element in `s` on the next trip through the `for`. The expression `++it` uses the increment operator, overloaded for the iterator type. The increment operator has the effect of advancing the iterator to the next element in the container. We don't know, and shouldn't care, how the increment operator works. All that we need to know is that afterward, the iterator denotes the next element in the container.

In the body of the `for`, `it` is positioned on an element in `s`, whose value we need to read and then write to standard output. We access that element by calling the dereference operator `*`. When applied to an iterator, the `*` operator returns an lvalue that is the element to which the iterator refers. Therefore, the output operation `std::cout << *it` has the effect of writing the current element's value on the standard output.

One last thing about iterators. Since C++11, the compiler provides global functions `std::begin` and `std::end` [declared in `<iterator>`] that return an iterator to the first and an iterator pointing to one past the end of the sequence of elements in a container, respectively. These functions return a value of type `iterator` [belonging to the appropriate container]. If you want `const_iterator`, C++11 provides the global functions `std::cbegin` and `std::cend`:

```cpp
std::string s {"Lisa Simpson"};
for (std::string::const_iterator it{std::begin(s)};
     it != std::end(s); ++it) {
  std::cout << *it << ' ';
}
```

Range-`for` Statements

C++ takes advantage of the notion of a half-open sequence to provide a simple loop over all the elements of a sequence. For example, we can eschew the use of indexes or iterators to iterate through the elements of a `string` variable:

```cpp
std::string s {"Lisa Simpson"};
for (char ch : s) {
  std::cout << ch << ' ';
}
```

This is called a range-`for`-loop because the word range is often used to mean the same as "sequence of elements." We read `for (char ch : s)` as "for each `char ch in s`" and the meaning of the loop is exactly the equivalent loop over the subscripts [`0:s.length()`). We use range-`for`-loop as a safer alternative to a `for` loop that uses the subscript operator `[]` to inspect the elements of a sequence one element at a time. More complicated loops, such as looking at every third element of a `string`, or looking at only the second half of a `string`, or comparing elements of two `string`s, are usually better done using the more complicated and more general traditional `for` statement. Still, we can do useful things with the range-`for`-loop such as make all Latin characters in a `string` uppercase:

```cpp
std::string s ("Lisa is number 1!!!");
for (char& ch : s) {
  ch = std::islower(ch) ? std::toupper(ch) : ch;
}
```

Because we want to change the value of the elements in `s`, we declare `ch` as a reference. When we assign to `ch` inside the loop, that assignment changes the element to which `ch` is bound.

We can use the fact that one of the overloads of `operator[]` returns a `char&` to rewrite the above code snippet using the traditional `for` statement:

```cpp
std::string s {"Lisa is number 1!!!"};
for (std::string::size_type i{0}; i < s.length(); ++i) {
  s[i] = std::islower(s[i]) ? std::toupper(s[i]) : s[i];
}
```

While operator `[]` doesn't check whether the index passed as an argument is valid; member function `at` does. If called with an invalid index, `at` throws an exception of type `std::out_of_range`. If operator `[]` is called with an invalid index, program behavior is undefined. In general, the position after the last character is valid and returns `\0`. That is, `s[s.length()]` will evaluate to `'\0'`.

Comparisons

The `string` class has overloads for relational operators `==`, `!=`, `<`, `<=`, `>`, and `>=` that allow lexicographic comparisons between `string`s and C-style strings, as long as both operands are not C-style strings:

```cpp
std::string first {"aaa"};
std::string second {"aaaa"};
std::cout << (first < first) << "\n";  // false
std::cout << (first <= first) << "\n"; // true
std::cout << (first < second) << "\n"; // true
```

When two strings are lexicographically compared, their characters are compared a pair at a time (both first characters, then both second characters, and so on). If no differences are found, the strings are equal; if a difference is found, the string with the first lower character is considered the smaller string, as shown in these examples:

- `"Hello"` is greater than `"Good Bye"` because `'H'` in `"Hello"` is greater than `'G'` in `"Good Bye"`.
- `"Hello"` is less than `"hello"` because `'H'` in `"Hello"` is less than `'h'` in `"hello"`.
- `"SMITH"` is greater than `"JONES"` because `'S'` in `"SMITH"` is greater than `'J'` in `"JONES"`.
- `"123"` is greater than `"1227"` because the third character `'3'` in `"123"` is greater than the third character `'2'` in `"1227"`.
- `"Behop"` is greater than `"Beehive"` because the third character `'h'` in `"Behop"` is greater than the third character `'e'` in `"Beehive"`.

> ***The C++ type system permits it to compare `string`s and C-style strings to `string`s but cannot compare C-style strings to C-style strings.***

Assignments

To modify a string, you can use operator `=` to assign a new value. The assigned value may be a `string`, a C-style string, or a single character:

```cpp
std::string one {"othello"};
std::string two;     // empty string
two = one;           // assign "othello"
two = "two\nlines"; // assign a C-style string
two = 'x';           // assign a single character
```

Substrings and concatenation

You can extract substrings, and you can glue smaller `string`s together to form larger ones. You can extract a substring from any `string` by using the `substr` member function. Given a `string` object `s`, the call `s.substr(pos, len)` returns a `string` that is made from the characters in `s`, starting at index `pos`, and containing `len` characters. Don't forget that index numbers start at `0`. The first character in a sequence is at index `0`, the second one `1`, and so on. This function is declared to have default value `0` for `pos` and a default value indicating all characters until the end of the string. For example:

```
std::string s("interchangeability");
s.substr(5, 6); // returns string("change")
s.substr(0, 5); // returns string("inter")
s.substr(11);   // returns string("ability")
s.substr();     // returns string("interchangeability")
```

Now that you know how to take strings apart, let’s see how to put them back together. You can use operator `+` to concatenate two `string`s or a `string` and a C-style string or a `string` and a value of type `char`. In addition, operator `+=` is overloaded for `string`s so that a `string` can be extended with the characters in a `string` or C-style string or a `char`. For example, the following code fragment:

```
std::string s1("enter"), s2("nation");
std::string res = 'i' + s1.substr(1) + s2 + "aliz" + s2.substr(1);
std::cout << "res: " << res << "\n";
```

will have the following output:

```
res: internationalization
```

The earlier concatenation can be written using operator `+=`:

```
std::string res {'i'};
res += s1.substr(1);
res += s2;
res += "aliz";
res += s2.substr(1);
```

The following example incorporates both substrings and concatenation concepts:

```
std::cout << "Enter your full name (first middle last): ";
std::string first, middle, last;
std::cin >> first >> middle >> last;
std::string initials = first.substr(0, 1);
initials += middle.substr(0, 1) + last.substr(0, 1);
std::cout << "Your initials are " << initials << "\n";
```

Searching and finding

C++ offers the ability to search in a `string` object in many variations with each variation existing in various overloaded forms. All search functions in the `string` interface have the word `find` inside their name. These functions try to find a character position given a `value` that is passed as an argument. How the search proceeds depends on the exact name of the `find` function:

`string` **member function**	**Effect**
`s.find`	Finds *first* occurrence of `value` in `s`
`s.rfind`	Finds *last* occurrence of `value` in `s`
`s.find_first_of`	Finds *first* character that is part of `value` in `s`
`s.find_last_of`	Finds *last* character that is part of `value` in `s`
`s.find_first_not_of`	Finds *first* character that is not part of `value` in `s`
`s.find_last_not_of`	Finds *last* character that is not part of `value` in `s`

All search functions return the index of the first character of the character that matches the search. If the search fails, they return `string::npos`. The search functions use the following argument scheme:

- The first argument is always the value that is searched for.
- The second optional value indicates an index at which to start the search in the `string`.
- The optional third argument is the number of characters of the value to search.

In particular, each search function is overloaded with the following set of arguments:

- `string const& value`: searches against the characters of the `string` `value`.
- `string const& value, size_type idx`: searches against the characters of `value`, starting with index `idx` in the `string` object calling this function.
- `char const* value`: searches against the characters of the C-style string `value`.
- `char const* value, size_type idx`: searches against the characters of the C-style string `value`, starting with index `idx` in the `string` object calling this function.
- `char const* value, size_type idx, size_type value_len`: searches against the `value_len` characters of the character array `value`, starting with index `idx` in the `string` object calling this function. The null character `'\0'` has no special meaning here inside `value`.
- `char const value`: searches against the character `value`.
- `char const value, size_type idx`: searches against the characters `value`, starting with index `idx` in the `string` object calling this function.

For example:

```cpp
std::string s{"Hi Bill, I'm ill, so please pay the bill"};
s.find("il"); // returns 4 (first substring "il")
s.find("il", 10); // returns 13 (first substring "il" starting from s[10])
s.rfind("il"); // return 37 (last substring "il")
s.find_first_of("il"); // returns 1 (first char 'i' or 'l')
s.find_last_of("il"); // return 39 (last char 'i' or 'l')
s.find_first_not_of("il"); // returns 0 (first char neither 'i' nor 'l')
s.find_last_not_of("il"); // returns 36 (lost char neither 'i' nor 'l')
s.find("hi"); // returns npos
```

Another example:

```cpp
std::string str;
std::string::size_type idx = str.find("no");
if (idx == std::string::npos) std::cout << "not found";

str = {"dkeu84kf8k48kdj39kdj74945du942"};
std::string str2 {"84"};
std::cout << str.find(str2); // 4
std::cout << str.rfind(str2); // 4
std::cout << str.find(str2, 10); // 18446744073709551615
std::cout << str.find('8'); // 4
std::cout << str.rfind('8'); // 11
std::cout << str.find('8', 10); // 11

str2 = "0123456789";
std::cout << str.find_first_of("678"); // 4
std::cout << str.find_last_of("678"); // 20
std::cout << str.find_first_of("678", 10); // 11
std::cout << str.find_first_of(str2); // 4
std::cout << str.find_last_of(str2); // 29
std::cout << str.find_first_of(str2, 10); // 10
std::cout << str.find_first_not_of("678"); // 0
std::cout << str.find_last_not_of("678"); // 29
std::cout << str.find_first_not_of("678", 10); // 10
std::cout << str.find_first_not_of(str2); // 0
std::cout << str.find_last_not_of(str2); // 26
std::cout << str.find_first_not_of(str2, 10); // 12
```

The value `string::npos`

If a search function fails, it returns `string::npos`. Consider the following example:

```cpp
std::string s{"Hi Bill, I'm ill, so please pay the bill"};
std::string s2{"Fill"};
std::string::size_type idx = s.find(s2);
if (idx == std::string::npos) {
  std::cout << "Could not find <\"" << s2 << "\"> in <\"" << s << "\">.\n";
} else {
  std::cout << "<\"" << s2 << "\"> is at position "
            << idx << " in <\"" << s << "\">.\n";
}
```

The `if` statement condition yields `true` if and only if `s2` is not part of `s`. Be very careful when using `npos`. When you want to check the return value, always use `string::size_type`, not `int` or `unsigned` for the type of the return value; otherwise, the comparison of the return value with `npos` might not work. This behavior is the result of the design decision that `string::npos` is defined as having type `size_type` and is initialized with value `-1`. Because `size_type` has unsigned type, `-1` is converted into an unsigned integral type, and therefore `npos` is the maximum unsigned value of its type. However, the exact value depends on the exact definition of type `size_type`. Unfortunately, maximum values differ for different unsigned types. For example, `(unsigned long)-1` differs from `(unsigned short)-1` if the sizes of the types differ. Therefore, always use `size_type` as the type of the return value.

> ***Always use `string::size_type`, not `int` or `unsigned` for the type of the return value of search functions; otherwise, the comparison of the return value with `string::npos` might not work.***

Modifying operations

There are several member functions to insert, remove, replace, and erase characters of a `string`:

Modifying member function	Effect
`s.assign`	Assigns to `s` a new `string`
`s.swap(s2)`	Swaps `s` and `s2`
`s.pop_back`	Removes last character from `s`
`s.erase`	Removes characters from `s`
`s.clear`	Clears (removes) characters from `s`
`s.append`	Appends characters to `s`
`s.push_back(c)`	Appends character `c` to `s`
`s.insert(pos, ...)`	Inserts characters to `s` starting at `pos`
`s.replace(pos, len, ...)`	Replaces `len` characters from `s` starting at `pos`

These functions are available in many overloaded versions. Functions `s.assign`, `s.append`, `s.insert`, and `s.replace` are very similar. All four can be invoked with `string`s, substrings, characters, C-style strings, ranges and initializer lists. `s.erase` can erase a single character, characters starting at given position, and ranges. The following code fragments show many of the variations. For lack of space, only the effects of the modifications are shown:

```cpp
std::string str  {"New String"};
std::string str2 {"Other String"};
str.assign(str2, 4, std::string::npos); // "r String"

str.assign(5, '-'); // "-----"

str = {"0123456789"};
str.erase(7, 2);                       // "01234569"
str.erase(str.begin()+2, str.end()-2); // "012"
str.erase(str.begin()+2, str.end());   // "01"
str.pop_back();                         // "0"
str.erase();                            // ""

str = "01234";
str += "56";              // "0123456"
str += '7';               // "01234567"
str += {'8', '9'};        // "0123456789"
str.append(str);          // "01234567890123456789"
str.append(str, 2, 4);    // "012345678901234567892345"
str.append(3, '0');       // "012345678901234567892345000"
str.append(str, 10, 10); // "0123456789012345678923450000001234567989"
str.push_back('9');       // "01234567890123456789234500001234567989 9"

str = {"345"};
str.insert(3, "6789"); // 3456789
str.insert(0, "012");  // 0123456789

str = {"only for testing purpose."};
str.replace(0, 0, "O"); // Only for testing purpose.
str.replace(0, 5, "Only", 0, 4); // Only for testing purpose.
str.replace(16, 8, ""); // Only for testing.
str.replace(4, 0, 5, 'y'); // Onlyyyyyy for testing.
str.replace(str.begin(), str.end(), "Only for testing purpose.");
str.replace(str.begin()+4,str.end()-8,10,'#'); // Only##########purpose.
```

Conversion between `string`s and C-style strings

There are three possible ways to convert the contents of the `string` into a raw array of characters or C-style string:

1. Member functions `data` and `c_str` are equivalent in behavior and both return the contents of the `string` as a null-terminated array of characters. The returned array is owned by the `string` and therefore the caller must not modify or free the memory.
2. Member function `copy` copies the specified number of characters from the `string` into a character array provided by the caller. Note that the `copy` member function doesn't append a `'\0'` character to the copied sequence of characters.

```cpp
std::string str {"C++ String and C String"};
std::cout << str << '\n';
char const *cstr = str.c_str();
std::cout << cstr << '\n';
// c_str and data are equivalent functions
cstr = str.data();
std::cout << cstr << '\n';

char buffer[11];
str.copy(buffer, 10);
//std::cout << buffer << '\n'; // ERROR: '\0' not appended
buffer[10] = '\0';
std::cout << buffer << '\n'; // ok
```

Conversion between `string`s and numeric values

The header `<string>` defines a number of overloaded global functions `to_string` that facilitate the conversion of arithmetic values of all built-in types to `string`s. These functions return a `string` that encapsulates the numeric value as a sequence of characters.

The reverse conversion of a sequence of characters encapsulated by a `string` object into a numeric value is accomplished by a variety of functions such as `stoi`, `stol`, `stof`, `stod` and other. These global functions are declared in `<string>`.