**Introduction to the Standard Library**

An object-oriented programming language like C++ relies significantly on libraries that support its non-library core.  These libraries implement programming solutions for much of the repetitive detail encountered in the development of modern applications.  The libraries that support the C++ language core include a broad variety of types, a framework of containers and algorithms for all user-defined types, and support for resource management and multi-threading.  Two-thirds of the C++17 standard is devoted to detailed descriptions of library support facilities.

This chapter introduces the categories of the C++ standard library and summarizes their contents.  It includes overviews of the string library, facilities added to the Standard Libraries with C++11 and the framework of containers and algorithms for user-defined types called the Standard Template Library.  The chapter entitled [Introduction to Linked Lists](https://mwatler.github.io/sep200/Week13/LinkedLists.docx) describes the technology that underlies some of the containers.

Categories of the Standard Library

The categories of the C++ standard library are listed below.  The header files for the components of these categories are listed alogside.  Examples of facilities referred to in these notes are listed below each entry.  The categories and components that are not covered in these notes are marked 'beyond scope'. :

* language support (functions and types of objects generated implicitly)
  + common definitions <cstddef> <cstdlib>  
    - size\_t ptrdiff\_t
  + start and termination <cstdlib>  
    - atoi() abort() atexit() exit()
  + dynamic memory <new>  
    - new new[] delete delete [] bad\_alloc
  + dynamic type identification <typeinfo>  
    - type\_info
  + exception handling <exception>  
    - bad\_exception terminate()
  + implementation properties (beyond scope) <limits> <climits> <cfloat>
  + integer types (beyond scope) <cstdint>
  + initializer lists (beyond scope) <initializer\_list>
  + other run-time support (beyond scope) <csignal> <csetjmp> <cstdalign> <cstdarg> <cstdbool> <cstdlib>
* diagnostics (for detecting and reporting error conditions)
  + exception classes (beyond scope) <stdexcept>
  + assertions (beyond scope) <cassert>
  + error numbers (beyond scope) <cerrno>
  + system error support (beyond scope) <system\_error>
* general utilities (generally useful)
  + components <utility> - utilities, integer sequences, pairs  
    - move
  + optional objects (see below) <optional> - may contain a value
  + variants (see below) <variant> - type-safe union
  + storage for any type <any> -
  + smart pointers (see later chapter) <memory>
  + function objects (see later chapter) <functional>  
    - ref()
  + time (see later chapter) <chrono>, <ctime>
* strings (for manipulating sequences of char-like types)
  + character traits (beyond scope) <string>
  + string classes (see below) <string>
  + string view classes (see below) <string\_view>
  + null-terminated sequence utilities <cctype>, <cwctype>, <cstring>, <cwchar> ,<cstdlib>, <cuchar> - part of the C library
* localization (for encapsulating cultural differences)
  + locales (beyond scope) <locale> - international support for text processing
  + code conversion facets (beyond scope) <codecvt> - unicode localization utilities
  + C locales (beyond scope) <clocale> - C library international support for text processing
* containers (see below) <vector, ...> - part of the STL framework
* iterators (see below) <iterator> - part of the STL framework
* algorithms (see below) <algorithms>) - part of the STL framework
* numerics (for seminumerical operations)
  + floating-point environment (beyond scope) <cfenv>
  + complex numbers (beyond scope) <complex>
  + random number generation (beyond scope) <random>
  + numeric arrays (beyond scope) <valarray> - class for representing and manipulating arrays of values
  + generalized numeric operations (see later chapter) <numerics>, <numeric, cmath> - numeric operations on values in containers
  + mathematical functions <cmath>, <ctgmath>, <cstdlib> - common math operations
* input/output (for input and output operations)
  + standard iostream objects (see later chapter) <iostream> - primary mechanism for input and output
  + iostream base classes (see later chapter) <ios> - underlying support for input and output
  + formatting and manipulation (see later chapter) <istream>, <ostream>, <iomanip> - helper functions
  + string streams (beyond scope) <sstream>
  + file streams (see later chapter) <fstream>
  + file systems (beyond scope) <filesystem>
  + C input and output (beyond scope) <cstdio>, <cinttypes> - primary C mechanism for input and output
* regular expressions (beyond scope) <regex> - matching and searching facilities
* atomic operations (beyond scope) <atomic> - concurrent access to shared data
* thread support (for multi-threading support)
  + threads (see later chapter) <thread> - manages threads and inter-thread communications
  + mutual exclusion (beyond scope) <mutex>, <shared\_mutex>
  + condition variables (beyond scope) <condition\_variable>
  + futures (see later chapter) <future> - manages threads and inter-thread communications

Most library components are declared and defined in the std namespace.  The filesystem component is declared and defined in the filesystem namespace.  The library macros are predefined in the global namespace.

The C++ Standard Library makes available the facilities of the C Standard Library.  A review of the input/output features is in the appendices entitled [The C Libraries](https://ict.senecacollege.ca/~btp305/pages/content/std_c.html) and [How C++ and C Fit Together](https://ict.senecacollege.ca/~btp305/pages/content/cpp11_p.html#std).

String Library

The String Library provides support for three general types of strings:

* the string classes
* the string\_view classes
* null terminated C-style string functions

String Classes

The string classes store and manipulate sequences of character-like objects.  The specializations of the <std::basic\_string<CharT>> template for specific character types are:

* std::string (<std::basic\_string<char>>)
* std::wstring (<std::basic\_string<wchar\_t>>)
* std::u16string <std::basic\_string<char16\_t>>)
* std::u32string <std::basic\_string<char32\_t>>)

The public member functions include:

* operator= - assigns a string to the current string
* operator[] - accesses a specified character in the string
* size() - the number of characters in the string
* substr() - returns a substring of the string
* find(char c) - find the first occurrence of c in the string
* rfind(char c) - find the last occurrence of c in the string
* find\_first\_of(char c) - find the first occurrence of c in the string
* find\_last\_of(char c) - find the last occurrence of c in the string
* find\_first\_of(basic\_string\_view(const CharT\* s)) - find the first occurrence of any character in the sequence s
* find\_last\_of(basic\_string\_view(const CharT\* s)) - find the last occurrence of any character in the sequence s
* operator+= - append a character or a string to the current string

The helper functions include:

* operator== - equality comparison
* operator!= - inequality comparison
* operator>> - extract the string from the input stream
* operator<< - insert the string into the output stream

Unlike the const char\* type (a pointer to a C-style null-terminated string), a string type keeps track of the number of characters in the string through an instance variable without using the null byte as a terminator.  A string object can include one or more null byte ('\0') characters.

|  |  |
| --- | --- |
| // String Class  // string.cpp  #include <iostream>  #include <string>  int main() {  std::string str("Hello");  str += '\0'; // adds a null byte  str += " World"; // adds a string  int i = 0;  for (const auto& c : str)  if (str[i++] == '\0')  std::cout << "Null byte at str["  << i-1 << "]\n";  std::cout << str << std::endl;  } | Null byte at str[5]  Hello World |

String View Module

C++17 introduced the <string\_view> library to augment the std::string class and avoid redundant copying operations.  The class template std::basic\_string\_view defines a lightweight non-owning, read-only view into a contiguous sequence of characters with the first element at position zero.

The string\_view classes refer to sequences of character-like objects.  The specializations of the <std::basic\_string\_view<CharT>> template are:

* std::string\_view (<std::basic\_string<char>>)
* std::wstring\_view (<std::basic\_string<wchar\_t>>)
* std::u16string\_view <std::basic\_string<char16\_t>>)
* std::u32string\_view <std::basic\_string<char32\_t>>)

The public member functions include:

* operator= - assign one view to another
* operator[] - access an element in a view
* size() - the number of characters in the view
* substr() - a substring of the view
* find(char c) - find the first occurrence of c in the view
* find\_first\_of(char c) - find the first occurrence of c in the view
* find\_last\_of(char c) - find the last occurrence of c in the view
* find\_first\_of(basic\_string\_view(const char\* s)) - find the first occurrence of any character in the sequence s
* find\_last\_of(basic\_string\_view(const char\* s)) - find the last occurrence of any character in the sequence s

The helper functions include:

* operator== - equality comparison
* operator!= - inequality comparison
* operator<< - insert into output stream

Like a string object, a string\_view object can include the null byte ('\0') or several null bytes as a non-terminating character.  Both achieve this by storing the number of characters in the string as an instance variable.

|  |  |
| --- | --- |
| // String View Class  // string\_view.cpp  #include <iostream>  #include <string>  #include <string\_view>  int main() {  std::string str("Hello");  str += '\0'; // adds a null byte  str += " World"; // adds a string  int i = 0;  std::string\_view str\_view(str);  for (const auto& c : str\_view)  if (str\_view[i++] == '\0')  std::cout << "Null byte at str\_view["  << i-1 << "]\n";  std::cout << str\_view << std::endl;  } | Null byte at str\_view[5]  Hello World |

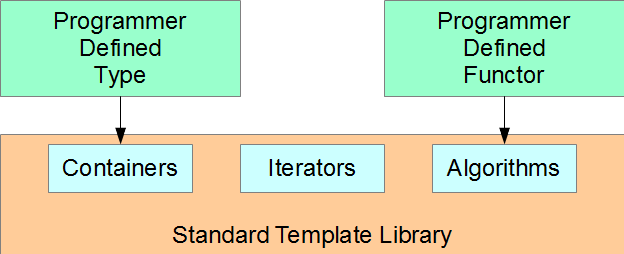
Since a string\_view object does not own the characters to which it points, it is the programmer's responsibility to ensure that the object does not outlive the pointed-to character array.

Standard Template Library

The Standard Template Library (STL) is arguably the most prominent part of the C++ Standard Library.  It provides code for managing the elements of a data structure in a generic form, hiding the complex details and allowing re-use.  The STL consists of:

* container template classes
  + sequential containers
  + container adaptors
  + associative containers
* iterators
* algorithms
* function objects

A container class represents the shell of a data structure, manages the memory associated with the elements of that structure and provides member functions to access those elements.  Iterators facilitate the traversal of the data structure and provide simple access to range of elements.  Algorithms implement solutions for sequences of elements through the use of iterators and function objects.



A complete programming solution to the implementation of a data structure requires:

* the definition of the data type of each element in the data structure
* the choice of the optimal data structure to collect the elements
* the function object for the algorithm to use on the data structure
* syntax to accesses the facilities of the STL

Recent Facilities

C++11 Facilities

lvalue and rvalue Distinction

* std::ref() (<functional>) - returns an lvalue reference to its argument
* std::move() (<utility>) - returns an rvalue reference to its argument

C++17 Facilities

C++17 introduced new library facilities.  They include:

* std::string\_view (<string\_view>) - a read-only contiguous sequence of characters (see above)
* std::variant<T, ...>() (<variant>) - represents a type-safe union
* std::optional<T> (<optional>) - may or may not contain a value (beyond scope)
* std::any (<any>) - single values of any type (beyond scope)
* std::uncaught\_exceptions (<exception>) - stack is in the process of unwinding
* Uniform container access std::size std::empty std::data - number of elements, emptiness, direct access to underlying array
* Special Mathematical Functions (<numerics>) -= (beyond scope)
* Filesystem Library filesystem:: (<filesystem>) - facilities for performing operations on file systems and their components including paths, regular files and directories (beyond scope)

Type-Safe Unions

The class template std::variant<T, ...> represents a type-safe union.  The parameter list specifies the types that may be stored in the union.  A type may be repeated.  The value of the variant can be accessed using either its type, if unique, or its index.

The public member functions include:

* operator=(v) - assigns value of variant v to the left operand
* index() - returns the index I of the variant stored in the current object

The helper functions include:

* std::get<T>(v) - returns value of variant v given the type T
* std::get<I>(v) - returns value of variant v given the index I

The helper classes include:

* std::bad\_variant\_access - exception thrown on invalid access

For example,

|  |  |
| --- | --- |
| // Variant - a type-safe union  // variant.cpp  #include <iostream>  #include <variant>  int main() {  std::variant<long, double> a, b;  a = 12l; // a contains a long  b = std::get<long>(a);  std::cout << b << std::endl;  try {  double c = std::get<double>(b);  std::cout << c << std::endl;  }  catch(std::bad\_variant\_access& bva) {  std::cout << "bad type access" << std::endl;  }  } | 12  bad type access |