Chapter 9 Strings

Learning Objectives

- An Array Type for Strings
  - ✓C-Strings
- Character Manipulation Tools
  - ✓Character I/O
  - ✓get, put member functions
  - ✓putback, peek, ignore
- Standard Class string
  - ✓String processing

Introduction

- Two string types:
- C-strings
  - ✓Array with base type char
  - ✓End of string marked with null, "\0"
  - ✓"Older" method inherited from C
- String class
  - ✓Uses templates

C-Strings

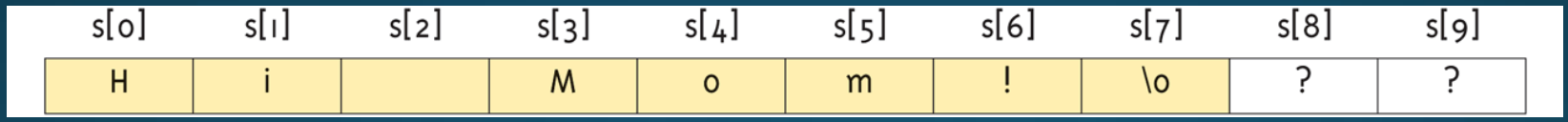
- Array with base type *char*
  - ✓One character per indexed variable
  - ✓One extra character: "\0"
    - Called "null character"
    - End marker
- We've used c-strings
  - ✓Literal "Hello" stored as c-string

C-String Variable

- Array of characters:
  char s[10];
  - ✓Declares a c-string variable to hold up to 9 characters
  - ✓+ one null character
- Typically "partially-filled" array
  - ✓Declare large enough to hold max-size string
  - ✓Indicate end with null
- Only difference from standard array:
  - ✓Must contain null character

C-String Storage

- A standard array:
  char s[10];
  - ✓If s contains string "Hi Mom", stored as:



C-String Initialization

- Can initialize c-string:
  char myMessage[20] = "Hi there.";
  - ✓Needn't fill entire array
  - ✓Initialization places "\0" at end
- Can omit array-size:
  char shortString[] = "abc";
  - ✓Automatically makes size one more than length of quoted string
  - ✓NOT same as:
    char shortString[] = {"a", "b", "c"};

C-String Indexes

- A c-string IS an array
- Can access indexed variables of:
  char ourString[5] = "Hi";
  - ✓ourString[0] is "H"
  - ✓ourString[1] is "i"
  - ✓ourString[2] is "\0"
  - ✓ourString[3] is unknown
  - ✓ourString[4] is unknown

C-String Index Manipulation

- Can manipulate indexed variables
  char happyString[7] = "DoBeDo";
  happyString[6] = "Z";
  - ✓Be careful!
  - ✓Here, "\0" (null) was overwritten by a "Z"!
- If null overwritten, c-string no longer "acts" like c-string!
  - ✓Unpredictable results!

Library

- Declaring c-strings
  - ✓Requires no C++ library
  - ✓Built into standard C++
- Manipulations
  - ✓Require library <cstring>
  - ✓Typically included when using c-strings
    - Normally want to do "fun" things with them

= and == with C-strings

- C-strings not like other variables
  - ✓Cannot assign or compare:
    char aString[10];
    aString = "Hello"; // ILLEGAL!
    - Can ONLY use "=" at declaration of c-string!
- Must use library function for assignment:
  strcpy(aString, "Hello");
  - ✓Built-in function (in <cstring>)
  - ✓Sets value of aString equal to "Hello"
  - ✓NO checks for size!
    - Up to programmer, just like other arrays!

Comparing C-strings

- Also cannot use operator ==

```
char aString[10] = "Hello";
char anotherString[10] = "Goodbye";
```

  - ✓ `aString == anotherString;  // NOT allowed!`

- Must use library function again:

```
if (strcmp(aString, anotherString))
        cout << "Strings NOT same.";
else
        cout << "Strings are same.";
```

The <cstring> Library: **Display 9.1** Some Predefined C-String Functions in <cstring> (1 of 2)

- Full of string manipulation functions

Display 9.1 Some Predefined C-String Functions in <cstring>

FUNCTION	DESCRIPTION	CAUTIONS
`strcpy(`*Target_String_Var*`,` *Src_String*`)`	Copies the C-string value *Src_String* into the C-string variable *Target_String_Var*.	Does not check to make sure *Target_String_Var* is large enough to hold the value *Src_String*.
`strcpy(`*Target_String_Var*`,` *Src_String, Limit*`)`	The same as the two-argument `strcpy` except that at most *Limit* characters are copied.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcpy`. Not implemented in all versions of C++.
`strcat(`*Target_String_Var*`,` *Src_String*`)`	Concatenates the C-string value *Src_String* onto the end of the C-string in the C-string variable *Target_String_Var*.	Does not check to see that *Target_String_Var* is large enough to hold the result of the concatenation.

(continued)

The <cstring> Library: **Display 9.1** Some Predefined C-String Functions in <cstring> (2 of 2)

Display 9.1 Some Predefined C-String Functions in `<cstring>`

FUNCTION	DESCRIPTION	CAUTIONS
`strcat(`*Target_String_Var*`,` *Src_String, Limit*`)`	The same as the two argument `strcat` except that at most *Limit* characters are appended.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcat`. Not implemented in all versions of C++.
`strlen(`*Src_String*`)`	Returns an integer equal to the length of *Src_String*. (The null character, `'\0'`, is not counted in the length.)	
`strcmp(`*String_1*`,` *String_2*`)`	Returns 0 if *String_1* and *String_2* are the same. Returns a value < 0 if *String_1* is less than *String_2*. Returns a value > 0 if *String_1* is greater than *String_2* (that is, returns a nonzero value if *String_1* and *String_2* are different). The order is lexicographic.	If *String_1* equals *String_2*, this function returns 0, which converts to `false`. Note that this is the reverse of what you might expect it to return when the strings are equal.
`strcmp(`*String_1*`,` *String_2, Limit*`)`	The same as the two-argument `strcat` except that at most *Limit* characters are compared.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcmp`. Not implemented in all versions of C++.

C-string Functions: strlen()

- "String length"
- Often useful to know string length:
  char myString[10] = "dobedo";
  cout << strlen(myString);
  - ✓Returns number of characters
    - Not including null
  - ✓Result here:
    6

C-string Functions: strcat()

- strcat()
- "String concatenate":
  char stringVar[20] = "The rain";
  strcat(stringVar, "in Spain");
  - ✓Note result:
    stringVar now contains "The rainin Spain"
  - ✓Be careful!
  - ✓Incorporate spaces as needed!

C-string Arguments and Parameters

- Recall: c-string is array
- So c-string parameter is array parameter
  - ✓C-strings passed to functions can be changed by receiving function!
- Like all arrays, typical to send size as well
  - ✓Function "could" also use "\0" to find end
  - ✓So size not necessary if function won't change c-string parameter
  - ✓Use "const" modifier to protect c-string arguments

C-String Output

- Can output with insertion operator, <<
- As we've been doing already:
  cout << news << " Wow.\n";
  - ✓Where *news* is a c-string variable
- Possible because << operator is overloaded for c-strings!

C-String Input

- Can input with extraction operator, >>
  - ✓Issues exist, however
- Whitespace is "delimiter"
  - ✓Tab, space, line breaks are "skipped"
  - ✓Input reading "stops" at delimiter
- Watch size of c-string
    - Must be large enough to hold entered string!
    - C++ gives no warnings of such issues!

C-String Input Example

- char a[80], b[80];
  cout << "Enter input: ";
  cin >> a >> b;
  cout << a << b << "END OF OUTPUT\n";
- Dialogue offered:

  Enter input: Do be do to you!
  DobeEND OF OUTPUT

  ✓Note: Underlined portion typed at keyboard
- C-string *a* receives: "do"
- C-string *b* receives: "be"

C-String Line Input

- Can receive entire line into c-string
- Use getline(), a predefined member function:

```
char a[80];
cout << "Enter input: ";
cin.getline(a, 80);
cout << a << "END OF OUTPUT\n";
```

  - ✓Dialogue:

    Enter input: Do be do to you!
    Do be do to you!END OF INPUT

Example: Command Line Arguments

- Programs invoked from the command line (e.g. a UNIX shell, DOS command prompt) can be sent arguments
  - ✓Example: COPY C:\FOO.TXT D:\FOO2.TXT
    - This runs the program named "COPY" and sends in two C-String parameters, "C:\FOO.TXT" and "D:\FOO2.TXT"
    - It is up to the COPY program to process the inputs presented to it; i.e. actually copy the files
- Arguments are passed as an array of C-Strings to the main function

Example: Command Line Arguments

- Header for main
  - ✓ int main(int argc, char *argv[])
  - ✓ argc specifies how many arguments are supplied. The name of the program counts, so argc will be at least 1.
  - ✓ argv is an array of C-Strings.
    - argv[0] holds the name of the program that is invoked
    - argv[1] holds the name of the first parameter
    - argv[2] holds the name of the second parameter
    - Etc.

Example: Command Line Arguments

```
// Echo back the input arguments
int main(int argc, char *argv[])
{
  for (int i=0; i<argc; i++)
  {
    cout << "Argument " << i << " " << argv[i] << endl;
  }
  return 0;
}
```

Sample Execution

```
> Test
Argument 0 Test
```

Invoking Test from command prompt

Sample Execution

```
> Test hello world
Argument 0 Test
Argument 1 hello
Argument 2 world
```

More getline()

- Can explicitly tell length to receive:

```
char shortString[5];
cout << "Enter input: ";
cin.getline(shortString, 5);
cout << shortString << "END OF OUTPUT\n";
```

  - ✓Results:
    Enter input: dobedowap
    dobeEND OF OUTPUT
  - ✓Forces FOUR characters only be read
    - Recall need for null character!

Character I/O

- Input and output data
  - ✓ALL treated as character data
  - ✓e.g., number 10 outputted as "1" and "0"
  - ✓Conversion done automatically
    - Uses low-level utilities
- Can use same low-level utilities ourselves as well

Member Function get()

- Reads one char at a time
- Member function of cin object:
  char nextSymbol;
  cin.get(nextSymbol);
  - ✓Reads next char & puts in variable nextSymbol
  - ✓Argument must be char type
    - Not "string"!

Member Function put()

- Outputs one character at a time
- Member function of cout object:
- Examples:

```
cout.put("a");
```

  - ✓Outputs letter "a" to screen

```
char myString[10] = "Hello";
cout.put(myString[1]);
```

  - ✓Outputs letter "e" to screen

More Member Functions

- putback()
  - ✓Once read, might need to "put back"
  - ✓cin.putback(lastChar);
- peek()
  - ✓Returns next char, but leaves it there
  - ✓peekChar = cin.peek();
- ignore()
  - ✓Skip input, up to designated character
  - ✓cin.ignore(1000, "\n");
    - Skips at most 1000 characters until "\n"

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (1 of 3)

Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE
`toupper(`*Char_Exp*`)`	Returns the uppercase version of *Char_Exp* (as a value of type `int`).	`char c = toupper('a');` `cout << c;` **Outputs:** A
`tolower(`*Char_Exp*`)`	Returns the lowercase version of *Char_Exp* (as a value of type `int`).	`char c = tolower('A');` `cout << c;` **Outputs:** *a*
`isupper(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is an uppercase letter; otherwise, returns `false`.	`if (isupper(c))` `  cout << "Is uppercase.";` `else` `  cout << "Is not uppercase.";`

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (2 of 3)

Display 9.3 Some Functions in <cctype>

FUNCTION	DESCRIPTION	EXAMPLE
`islower(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a lowercase letter; otherwise, returns `false`.	`char c = 'a';` `if (islower(c))` `  cout << c << " is lowercase.";` **Outputs:** a is lowercase.
`isalpha(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a letter of the alphabet; otherwise, returns `false`.	`char c = '$';` `if (isalpha(c))` `  cout << "Is a letter.";` `else` `  cout << "Is not a letter.";` **Outputs:** Is not a letter.
`isdigit(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is one of the digits `'0'` through `'9'`; otherwise, returns `false`.	`if (isdigit('3'))` `  cout << "It's a digit.";` `else` `  cout << "It's not a digit.";` **Outputs:** It's a digit.
`isalnum(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is either a letter or a digit; otherwise, returns `false`.	`if (isalnum('3') && isalnum('a'))` `  cout << "Both alphanumeric.";` `else` `  cout << "One or more are not.";` **Outputs:** Both alphanumeric.

Character-Manipulating Functions: **Display 9.3** Some Functions in <cctype> (3 of 3)

`isspace(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a whitespace character, such as the blank or newline character; otherwise, returns `false`.	`//Skips over one "word" and sets c` `//equal to the first whitespace` `//character after the "word":` `do` `{` `  cin.get(c);` `} while (! isspace(c));`
`ispunct(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character other than whitespace, a digit, or a letter; otherwise, returns `false`.	`if (ispunct('?'))` `  cout << "Is punctuation.";` `else` `  cout << "Not punctuation.";`
`isprint(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character; otherwise, returns `false`.	
`isgraph(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a printing character other than whitespace; otherwise, returns `false`.	
`isctrl(`*Char_Exp*`)`	Returns `true` provided *Char_Exp* is a control character; otherwise, returns `false`.	

Standard Class string

- Defined in library:

```
#include <string>
using namespace std;
```

- String variables and expressions
  - ✓Treated much like simple types
- Can assign, compare, add:

```
string s1, s2, s3;
s3 = s1 + s2;              //Concatenation
s3 = "Hello Mom!"          //Assignment
```

  - ✓Note c-string "Hello Mom!" automatically converted to string type!

Display 9.4
Program Using the Class string

Display 9.4 Program Using the Class string

```
1  //Demonstrates the standard class string.
2  #include <iostream>
3  #include <string>
4  using namespace std;

5  int main( )
6  {
7      string phrase;
8      string adjective("fried"), noun("ants");
9      string wish = "Bon appetite!";

10     phrase = "I love " + adjective + " " + noun + "!";
11     cout << phrase << endl
12          << wish << endl;

13     return 0;
14 }
```

Initialized to the empty string.

Two equivalent ways of initializing a string variable

SAMPLE DIALOGUE

```
I love fried ants!
Bon appetite!
```

String Class

- http://en.cppreference.com/w/cpp/string/basic_string

Page | Discussion | View | Edit | History

C++ / Strings library / std::basic_string

std::basic_string

Defined in header <string>

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>,
    class Allocator = std::allocator<CharT>
> class basic_string;                                          (1)

namespace pmr {
    template <class CharT, class Traits = std::char_traits<CharT>>
    using basic_string = std::basic_string< CharT, Traits,
                                            std::polymorphic_allocator<CharT>>;   (2) (since C++17)
}
```

The class template basic_string stores and manipulates sequences of char -like objects. The class is dependent neither on the character type nor on the nature of operations on that type. The definitions of the operations are supplied via the Traits template parameter - a specialization of std::char_traits or a compatible traits class.

The elements of a basic_string are stored contiguously, that is, for a basic_string s, &*(s.begin() + n) == &*s.begin() + n for any n in [0, s.size()), or, equivalently, a pointer to s[0] can be passed to functions that expect a pointer to the first element of a CharT[] array. (since C++11)

std::basic_string satisfies the requirements of AllocatorAwareContainer, SequenceContainer and ContiguousContainer (since C++17)

Several typedefs for common character types are provided:

Defined in header <string>

Type	Definition
std::string	std::basic_string<char>
std::wstring	std::basic_string<wchar_t>
std::u16string (C++11)	std::basic_string<char16_t>
std::u32string (C++11)	std::basic_string<char32_t>
std::pmr::string (C++17)	std::pmr::basic_string<char>
std::pmr::wstring (C++17)	std::pmr::basic_string<wchar_t>
std::pmr::u16string (C++17)	std::pmr::basic_string<char16_t>
std::pmr::u32string (C++17)	std::pmr::basic_string<char32_t>

String Class: member functions

- Member functions
  - ✓(constructor) // constructs a basic_string
  - ✓operator= // assigns values to the string
  - ✓…
- Member functions- Element Access
  - ✓at //access specified character with bounds checking
  - ✓operator[] // access specified character
  - ✓Front // accesses the first character
  - ✓….
  - ✓Find()
- Non-member functions
  - ✓operator +
  - ✓…
  - ✓stoi …

http://en.cppreference.com/w/cpp/string/basic_string

std::basic_string::operator=

std::basic_string::operator=

```
basic_string& operator=( const basic_string& str );                    (1)
basic_string& operator=( basic_string&& str );                         (2)  (since C++11)
basic_string& operator=( const CharT* s );                             (3)
basic_string& operator=( CharT ch );                                   (4)
basic_string& operator=( std::initializer_list<CharT> ilist );         (5)  (since C++11)
```

Replaces the contents of the string.

1) Replaces the contents with a copy of `str`. If `*this` and `str` are the same object, this function has no effect.

2) Replaces the contents with those of `str` using move semantics. Leaves `str` in valid, but unspecified state. If `*this` and `str` are the same object, the function has no effect. (until C++17)

2) Replaces the contents with those of `str` using move semantics. `str` is in a valid but unspecified state afterwards. If `std::allocator_traits<Allocator>::propagate_on_container_move_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If it is `false` and the source and the target allocators do not compare equal, the target cannot take ownership of the source memory and must assign each character individually, allocating additional memory using its own allocator as needed. Unlike other container move assignments, references, pointers, and iterators to `str` may be invalidated. (since C++17)

3) Replaces the contents with those of null-terminated character string pointed to by `s` as if by `*this = basic_string(s)`, which involves a call to `Traits::length(s)`.

4) Replaces the contents with character `ch` as if by `*this = basic_string(1,c)`

5) Replaces the contents with those of the initializer list `ilist` as if by `*this = basic_string(ilist)`

- Using line 9 at display 9.4 to see which function is invoked?

std::basic_string::operator[]

std::basic_string::operator[]

```
reference       operator[]( size_type pos );        (1)
const_reference operator[]( size_type pos ) const;  (2)
```

Returns a reference to the character at specified location pos. No bounds checking is performed.

1) If `pos == size()`, the behavior is undefined.

2) If `pos == size()`, a reference to the character with value `CharT()` (the null character) is returned. (until C++11)

If `pos == size()`, a reference to the character with value `CharT()` (the null character) is returned. (since C++11)

For the first (non-const) version, the behavior is undefined if this character is modified.

std::basic_string::operator[]

```
1)  #include <iostream>
2)  #include <string>
3)   int main()
4)  {
5)   {
6)     std::string const c("Exemplar");
7)     for (unsigned i = 7;i != 0; i/=2)
8)        std::cout << c[i];
9)   }
10)    std::cout << '\n';
11)  {
12)    std::string s("Exemplar ");
13)    s[s.size()-1] = 'y';
14)    std::cout << s;
15)  }
16) }
```

Output:
rmx
Exemplary

find()

std::basic_string::find

```
size_type find( const basic_string& str, size_type pos = 0 ) const;          (1)
size_type find( const CharT* s, size_type pos, size_type count ) const;      (2)
size_type find( const CharT* s, size_type pos = 0 ) const;                   (3)
size_type find( CharT ch, size_type pos = 0 ) const;                         (4)
```

Finds the first substring equal to the given character sequence. Search begins at pos, i.e. the found substring must not begin in a position preceding pos.

1) Finds the first substring equal to `str`.
2) Finds the first substring equal to the first `count` characters of the character string pointed to by `s`. `s` can include null characters.
3) Finds the first substring equal to the character string pointed to by `s`. The length of the string is determined by the first null character.
4) Finds the first character `ch` (treated as a single-character substring by the formal rules below).

```
1)  int main()
2)  {
3)      std::string::size_type n;
4)      std::string const s = "This is a string";
5)       // search from beginning of string
6)      n = s.find("is");     print(n, s);
7)       // search from position 5
8)      n = s.find("is", 5);     print(n, s);
9)       // find a single character
10)     n = s.find('q');
11)     print(n, s);
12) }
```

Which one is called?

```
1)  #include <string>
2)  #include <iostream>
3)   void print(std::string::size_type n, std::string const &s)
4)  {
5)      if (n == std::string::npos) {
6)          std::cout << "not found\n";
7)      } else {
8)          std::cout << "found: " << s.substr(n) << '\n';
9)      }
10) }
```

Constructor: std::basic_string::basic_string

std::basic_string::basic_string

```
explicit basic_string( const Allocator& alloc = Allocator() );                    (until C++14)
basic_string() : basic_string( Allocator() ) {}                                   (1)  (since C++14)
explicit basic_string( const Allocator& alloc );

basic_string( size_type count,
              CharT ch,
              const Allocator& alloc = Allocator() );                             (2)

basic_string( const basic_string& other,
              size_type pos,
              size_type count = std::basic_string::npos,
              const Allocator& alloc = Allocator() );                             (3)

basic_string( const CharT* s,
              size_type count,
              const Allocator& alloc = Allocator() );                             (4)

basic_string( const CharT* s,
              const Allocator& alloc = Allocator() );                             (5)

template< class InputIt >
basic_string( InputIt first, InputIt last,
              const Allocator& alloc = Allocator() );                             (6)

basic_string( const basic_string& other );                                        (7)

basic_string( const basic_string& other, const Allocator& alloc );                (7)  (since C++11)

basic_string( basic_string&& other );                                             (8)  (since C++11)

basic_string( basic_string&& other, const Allocator& alloc );                     (8)  (since C++11)

basic_string( std::initializer_list<CharT> init,
              const Allocator& alloc = Allocator() );                             (9)  (since C++11)
```

Which constructor is invoked?

```
1)    std::string s;
2)    assert(s.empty() && (s.length() == 0) && (s.size() == 0));
3)
4)    std::string s(4, '=');
5)    std::cout << s << '\n'; // "===="
6)
7)    std::string const other("Exemplary");
8)    std::string s(other, 0, other.length()-1);
9)     std::cout << s << '\n'; // "Exemplar"
10)  }
```

```
1)    // string::string()
2)    // string::string(size_type count, charT ch)
3)    (5)
4)    // string::string(string const& other, size_type pos, size_type count)
```

```
14)  std::string s("C-style string", 7);
15)  std::cout << s << '\n'; // "C-style"
16)
17)  std::string s("C-style\ostring");
18)  std::cout << s << '\n'; // "C-style"
19)
20)  char mutable_c_str[] = "another C-style string";
21)  std::string s(std::begin(mutable_c_str)+8,
                  std::end(mutable_c_str)-1);
22)  std::cout << s << '\n'; // "C-style string"
```

```
// string::string(charT const* s, size_type count)
// string::string(charT const* s)
// string::string(InputIt first, InputIt last)
```

I/O with Class string

- Just like other types!
- string s1, s2;
  cin >> s1;
  cin >> s2;
- Results:
  User types in: May the hair on your toes grow long and curly!
- Extraction still ignores whitespace:
  s1 receives value "May"
  s2 receives value "the"

getline() with Class string

- For complete lines:

```
string line;
cout << "Enter a line of input: ";
getline(cin, line);
cout << line << "END OF OUTPUT";
```

- Dialogue produced:

```
Enter a line of input: Do be do to you!
Do be do to you!END OF INPUT
```

  - ✓Similar to c-string's usage of getline()

Other getline() Versions

- Can specify "delimiter" character:

```
string line;
cout << "Enter input: ";
getline(cin, line, "?");
```

  - ✓Receives input until "?" encountered
- getline() actually returns reference
  - ✓

```
string s1, s2;
getline(cin, s1) >> s2;
```

  - ✓Results in: (cin) >> s2;

Pitfall: Mixing Input Methods

- Be careful mixing cin >> var and getline
  - ✓ int n;
    string **line**;
    cin >> n;
    getline(cin, **line**);
  - ✓ If input is: 42
    Hello hitchhiker.
    - Variable n set to 42
    - **line** set to **empty** string!
  - ✓ cin >> n skipped leading whitespace, leaving **"\n"** on stream for getline()!

getline() in C++

- std::getline
- std::fstream::getline
- std::istream::getline
- std::ifstream::getline
- std::iostream::getline
- std::wfstream::getline
- std::wistream::getline
- std::strstream::getline
- std::wifstream::getline
- std::wiostream::getline
- std::istrstream::getline
- std::stringstream::getline
- std::basic_fstream::getline
- std::basic_istream::getline
- std::istringstream::getline
- std::wstringstream::getline
- std::basic_ifstream::getline
- std::basic_iostream::getline
- std::wistringstream::getline
- std::basic_stringstream::getline
- std::basic_istringstream::getline

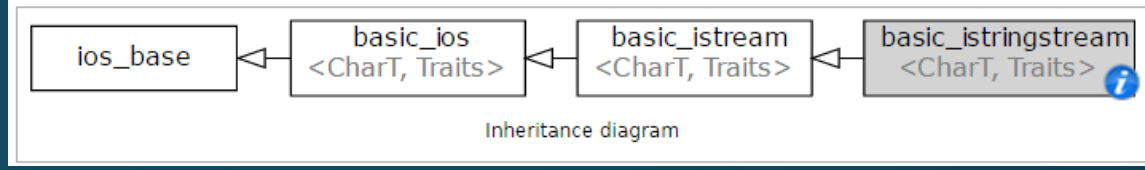
std::getline

Defined in header <string>

```
template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>& input,
                                           std::basic_string<CharT,Traits,Allocator>& str,
                                           CharT delim );                                   (1)

template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>&& input,
                                           std::basic_string<CharT,Traits,Allocator>& str,
                                           CharT delim );                                   (1) (since C++11)

template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>& input,
                                           std::basic_string<CharT,Traits,Allocator>& str );  (2)

template< class CharT, class Traits, class Allocator >
std::basic_istream<CharT,Traits>& getline( std::basic_istream<CharT,Traits>&& input,
                                           std::basic_string<CharT,Traits,Allocator>& str );  (2) (since C++11)
```

getline reads characters from an input stream and places them into a string:

1) Behaves as UnformattedInputFunction, except that input.gcount() is not affected. After constructing and checking the sentry object, performs the following:
   1) Calls `str.erase()`
   2) Extracts characters from input and appends them to str until one of the following occurs (checked in the order listed)
      a) end-of-file condition on input, in which case, getline sets eofbit.
      b) the next available input character is delim, as tested by `Traits::eq(c, delim)`, in which case the delimiter character is extracted from input, but is not appended to str.
      c) `str.max_size()` characters have been stored, in which case getline sets failbit and returns.
   3) If no characters were extracted for whatever reason (not even the discarded delimiter), getline sets failbit and returns.
2) Same as `getline(input, str, input.widen('\n'))`, that is, the default delimiter is the endline character.



http://en.cppreference.com/w/cpp/io/basic_istringstream

```
1)  int main()
2)  {
3)      // greet the user
4)      std::string name;
5)      std::cout << "What is your name? ";
6)      std::getline(std::cin, name);
7)      std::cout << "Hello " << name << ", nice to meet you.\n";
8)
9)      // read file line by line
10)     std::istringstream input;
11)     input.str("1\n2\n3\n4\n5\n6\n7\n");
12)     int sum = 0;
13)     for (std::string line; std::getline(input, line); ) {
14)         sum += std::stoi(line);
15)     }
16)     std::cout << "\nThe sum is: " << sum << "\n";
17) }
```

Possible output:

```
What is your name? John Q. Public
Hello John Q. Public, nice to meet you.

The sum is 28
```

Class string Processing

- Same operations available as c-strings
- And more!
  - ✓Over 100 members of standard string class
- Some member functions:
  - ✓.length()
    - Returns length of string variable
  - ✓.at(i)
    - Returns reference to char at position i

Display 9.7 Member Functions of the Standard Class string (1 of 2)

Display 9.7 Member Functions of the Standard Class `string`

EXAMPLE	REMARKS
Constructors	
`string str;`	Default constructor; creates empty `string` object `str`.
`string str("string");`	Creates a `string` object with data `"string"`.
`string str(aString);`	Creates a `string` object `str` that is a copy of `aString`. `aString` is an object of the class `string`.
Element access	
`str[i]`	Returns read/write reference to character in `str` at index `i`.
`str.at(i)`	Returns read/write reference to character in `str` at index `i`.
`str.substr(position, length)`	Returns the substring of the calling object starting at `position` and having `length` characters.
Assignment/Modifiers	
`str1 = str2;`	Allocates space and initializes it to `str2`'s data, releases memory allocated for `str1`, and sets `str1`'s size to that of `str2`.
`str1 += str2;`	Character data of `str2` is concatenated to the end of `str1`; the size is set appropriately.
`str.empty( )`	Returns `true` if `str` is an empty `string`; returns `false` otherwise.

(continued)

Display 9.7 Member Functions of the Standard Class string (2 of 2)

Display 9.7 Member Functions of the Standard Class `string`

EXAMPLE	REMARKS
`str1 + str2`	Returns a `string` that has `str2`'s data concatenated to the end of `str1`'s data. The size is set appropriately.
`str.insert(pos, str2)`	Inserts `str2` into `str` beginning at position `pos`.
`str.remove(pos, length)`	Removes substring of size `length`, starting at position `pos`.
Comparisons	
`str1 == str2  str1 != str2`	Compare for equality or inequality; returns a Boolean value.
`str1 < str2  str1 > str2`	Four comparisons. All are lexicographical comparisons.
`str1 <= str2  str1 >= str2`	
`str.find(str1)`	Returns index of the first occurrence of `str1` in `str`.
`str.find(str1, pos)`	Returns index of the first occurrence of string `str1` in `str`; the search starts at position `pos`.
`str.find_first_of(str1, pos)`	Returns the index of the first instance in `str` of any character in `str1`, starting the search at position `pos`.
`str.find_first_not_of (str1, pos)`	Returns the index of the first instance in `str` of any character *not* in `str1`, starting search at position `pos`.

C-string and String Object Conversions

- Automatic type conversions
  - ✓From c-string to string object:
    char aCString[] = "My C-string";
    string stringVar;
    stringVar = aCstring;
    - Perfectly legal and appropriate!
  - ✓aCString = stringVar;
    - ILLEGAL!
    - Cannot auto-convert to c-string
  - ✓Must use explicit conversion:
    strcpy(aCString, stringVar.c_str());

Summary

- C-string variable is "array of characters"
  - ✓With addition of null character, "\0"
- C-strings act like arrays
  - ✓Cannot assign, compare like simple variables
- Libraries <cctype> & <string> have useful manipulating functions
- cin.get() reads next single character
- getline() versions allow full line reading
- Class string objects are better-behaved than c-strings

Class Design Workshop

http://freeplay.game.gametower.com.tw/Game/FFPK/

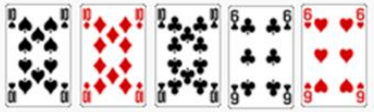
Sample Play

5 Cards Poker game (5PK)

- In a typical 5 cards poker game, each player gets a hand of 5 cards.
- The deck(一副牌) is shuffled and cards are dealt one at a time from the deck and added to the players' hands.
  - ✓In some games, cards can be removed from a hand, and new cards can be added.
- The game is won or lost depending on
  - ✓the value (ace, 2, …, king) and
  - ✓suit (spades, diamonds, clubs, hearts)

  of the cards that a player receives.
- The ranking list as for hands is shown
  - ✓See next page

5 card poker hands ranking:

Hand	Definition	Examples
ROYAL FLUSH	Ten, Jack, Queen, King, Ace of the same suit	
STRAIGHT FLUSH	Five cards in sequential order and of the same suit	
FOUR OF A KIND	Four cards of one denomination	
FULL HOUSE	Three cards of one denomination plus two cards of another denomination	
FLUSH	Any five cards of the same suit	
STRAIGHT	Five cards in sequential order	
THREE OF A KIND	Three cards of one denomination	
TWO PAIR	Two cards of one denomination plus two cards of another denomination	
PAIR	Two cards of one denomination	
HIGH CARD	A hand with no other combination, valued by the highest ranked card	

Write a program of 5PK for one player who plays with computer

HAND	PAYOFF	COMBINATIONS	PROBABILITY	EXPECTED VALUE in %
Royal Flush	1000	4	0.00%	0.15%
Straight Flush	200	36	0.00%	0.28%
Four of a Kind	100	624	0.02%	2.40%
Full House	50	3,744	0.14%	7.20%
Flush	25	5,108	0.20%	4.91%
Straight	10	10,200	0.39%	3.92%
3 of a Kind	3	54,912	2.11%	6.34%
Two Pairs	2	123,552	4.75%	9.51%
Pair, JJ-AA	1	337,920	13.00%	13.00%
Pair, 22-TT	0	760,320	29.25%	0.00%
Other	-1	1,302,540	50.12%	-50.12%
Total		2,598,960	100%	**-2.40%**

5PK type I: Five-card Stud

- Players and Cards
  - ✓52-card deck is used, and only five cards per player are dealt.
- The Play
  1. The deck is shuffled.
  2. The player places an ante (bet) in the pot.
  3. The player is dealt 5 cards (hand) face up, one at a time from the deck and added to the players' hands.
  4. The player wins if his/her hand is on the hands ranking list.
     1. Note that the banker is your program (computer)
  5. Repeat to step 1.

Let's start to design "Classes"

Our Class Design

- NOUNs in this description
  - ✓There are several candidates for objects:

    game, player, hand, card, deck, value, and **suit**.
- Card object
  - ✓the value and the suit of a card are simple values, and
  - ✓Might just be represented as instance variables in a Card object.
- In a complete program, the other five nouns might be represented by classes.
- Let's work on the ones that are most obviously reusable:

  card, **hand**, and **deck**.

Design Deck Class

- verbs in the description of 5PKgame for "Deck"
  - ✓shuffle a deck and deal a card from a deck
  - ⇒Behaviors (member functions):
    Shuffle() and DealCard()
- In detail
  - ✓When a deck of cards is first created, it contains 52 cards in some standard order
  - =>need to create a new deck, constructor() or as a member function
  - ✓Need to keep track of how many cards have used/left?
  - => CardsLeft()

Deck Class

```
1)  Class Deck
2)  {
3)  public:
4)     //  Create an unshuffled deck of cards.
5)     Deck();
6)     // Put all used cards back into the deck, and
       //       shuffle it into a random order.
7)     void Shuffle();
8)     // returns #of cards that are still left in the deck
9)     int CardsLeft();
10)    // Deals one card from the deck and returns it
11)    Card DealCard();
12)    Hand DealHand(int NoCards); // deal one hand with NoCards cards
13) private:
14)    Card deck[52];
15)    int cardUsed;
16)
17) };
```

Design Hand Class

- verbs in the description of 5PKgame for "Hand"
  - ✓Cards can be added to and removed from hands
  - ⇒AddCard() and RemoveCard()
- Like to arrange the cards in a hand so that cards of the same value are next to each other
  - =>SortByValue() and SortBySuit()
- Clear hand for each new round
  - => Clear()

Hand Class

```
1)  class Hand
2)  {
3)      Hand(); // Create a Hand object that is initially empty.

4)      // Discard all cards from the hand, making the hand empty.
5)      void clear();
6)      void addCard(Card c); // Add the card c to the hand.
7)      // same suit are grouped together, and by value.
8)      void sortBySuit();

9)      // Cards in order of increasing value, and by suit.
10)     void sortByValue();
11)   private:
12)           Card Hand[5];
13) };
```

Design Card Class

```
1)  Class Card
2)  {
3)    public:
4)          Card();
5)          ToString();
            // returns a string representation of  this card
            // including both its value and suit
6)          GetSuitAsString();
7)          GetValueAsString();
8)    private:
9)          int value;
10)         int suit;
11) }
```

Design 5PK game

Namely, write your **main** program

main()

```
1)  {
2)    Game game();
3)    game.play();
4)    return 1;
5)  }
```

Design Game Class

```
1)  class Game
2)  {
3)    public:
4)          Game();
5)          void Play();
6)          void ShowHand(Hand hand);
7)          int EvaluateHand(Hand hand);
8)          ShowCredits(int payoff);
9)    private:
10)         int gamesPlayed= 0; // number of game has played
11)         Card card;
12)         Hand hand;
13)         Deck deck;  // get a new deck of cards
14)         int payoff= 0;
15)         SetPayTable();
16) }
```

Void Play()

```
1)  {
2)     while (true)
3)     {
4)           deck.Shuffle();
5)           hand= deck.DealHand(5);
6)           ShowHand(hand);
7)           payoff= EvaluateHand(hand);
8)           ShowCreadits(payoff);
9)     }
10) }
```