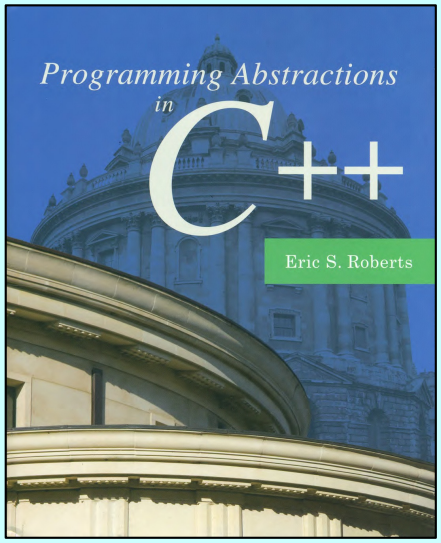
CHAPTER 3

Strings

Whisper music on those strings.
—T. S. Eliot, *The Waste Land,* 1922



Introduction to the C++ Standard Libraries

- A collection of ***classes*** and ***functions***, which are written in the core language and part of the C++ ISO Standard itself. Features of the C++ Standard Library are declared within the **`std`** ***namespace***
  - Containers: vector, queue, stack, map, set, etc.
  - General: algorithm, functional, iterator, memory, etc.
  - Strings
  - Streams and Input/Output: iostream, fstream, sstream, etc.
  - Localization
  - Language support
  - Thread support library
  - Numerics library
  - C standard library: cmath, cctype, cstring, cstdio, cstdlib, etc.

Using strings in C and C++

- Text data nowadays are as important as numeric data. Almost any program that you write in any modern language is likely to use string data at some point, even if it is only to display instructions to the user or to label results.
- Conceptually, a ***string*** is simply a sequence of ***characters***, which is precisely how strings are implemented in C.
- As a newly designed language, especially one to extend C with the object-oriented programming paradigm, C++ supports a higher-level view of strings, as ***objects***.
- The different strategies used by C and C++ on strings show the differences between different ***programming paradigms***.

Characters

- Both C and C++ use ASCII (*American Standard Code for Information Interchange*) as their encoding for character representation. The data type **`char`** therefore fits in a single eight-bit byte.
- With only 256 possible characters, ASCII is inadequate to represent the many alphabets in use throughout the world. In most modern language, ASCII has been superseded by Unicode, which permits a much larger number of characters.
- Even though the weaknesses in the ASCII encoding were clear at the time C++ was designed, changing the definition of **`char`** was impossible given the decision to keep C as a subset.
- The C++ libraries define the type **`wchar_t`** to represent "wide characters" that allow for a larger range. The details of the **`wchar_t`** type are beyond the scope of this text. We will stick with the traditional **`char`** type.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Single Characters

- To use variables of the single character type:

```
char ch;
char ch = 'a';
char ch = 97;
```

- Exercise: What is the equivalent statement of the following:

```
char ch = 32;
```

```
char ch = ' ';
```

- Some characters are special control characters:

```
char ch = 9;  // '\t' (Tab)
char ch = 10; // '\n' (New line)
char ch = 13; // '\r' (Return)
```

- To break a line, Mac uses **`\r`**, Unix uses **`\n`**, and Windows uses **`\r\n`**.

The `<cctype>` (`ctype.h`) Interface

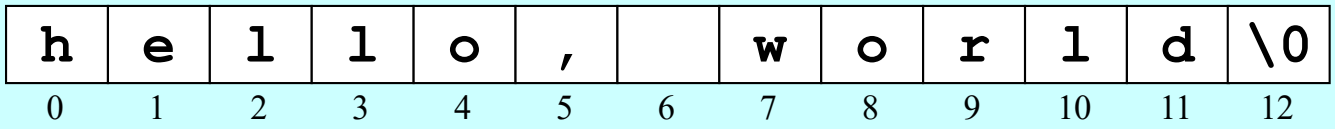
`bool isdigit(char ch)` Determines if the specified character is a digit.
`bool isalpha(char ch)` Determines if the specified character is a letter.
`bool isalnum(char ch)` Determines if the specified character is a letter or a digit.
`bool islower(char ch)` Determines if the specified character is a lowercase letter.
`bool isupper(char ch)` Determines if the specified character is an uppercase letter.
`bool isspace(char ch)` Determines if the specified character is ***whitespace*** (spaces and tabs).
`char tolower(char ch)` Converts **`ch`** to its lowercase equivalent, if any. If not, **`ch`** is returned unchanged.
`char toupper(char ch)` Converts **`ch`** to its uppercase equivalent, if any. If not, **`ch`** is returned unchanged.

The Legacy of C-Style Strings

- In C, we can represent strings, i.e., sequences of characters, as plain ***arrays*** of elements of a character type:

```
char cstr[10];
char cstr[] = "hello";
char cstr[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

- If you put double quotation marks around a sequence of characters, you get what is called a ***C string literal*** (`const char[]` type). The characters are stored in an ***array*** of bytes, terminated by a ***null byte*** whose ASCII value is 0. E.g., the characters in the C string "`hello, world`" are arranged like this:



- Character positions in a C string are identified by an ***index*** that begins at ***0*** and extends up to one less than the length of the string.

The Legacy of C-Style Strings

- Question: is `"a"` the same as `'a'`?

  `"a"` is a string literal containing an `'a'` and a null terminator `'\0'` , and therefore is a 2-char array.

Calling C String Functions

- The **<cstring>** (**string.h**) interface offers a lot of functions you can use to operate C strings. For example, if you want to determine the length of a C string **cstr**, you can use the following function **strlen**:

```
char cstr[] = "hello";
int len = strlen(cstr);
```

Does **cstr** *have enough space to hold* **"world"***?*

- If you want to assign a value to a C string **cstr**, you can use the following function **strcpy**:

```
strcpy(cstr, "world");
```

- However, you cannot directly assign a value to a C string except for initialization:

```
cstr = "world";
cstr[] = "world";
```

The <cstring> (string.h) Interface

Copying:

memcpy	Copy block of memory (function)
memmove	Move block of memory (function)
strcpy	Copy string (function)
strncpy	Copy characters from string (function)

Concatenation:

strcat	Concatenate strings (function)
strncat	Append characters from string (function)

Comparison:

memcmp	Compare two blocks of memory (function)
strcmp	Compare two strings (function)
strcoll	Compare two strings using locale (function)
strncmp	Compare characters of two strings (function)
strxfrm	Transform string using locale (function)

Searching:

memchr	Locate character in block of memory (function)
strchr	Locate first occurrence of character in string (function)
strcspn	Get span until character in string (function)
strpbrk	Locate characters in string (function)
strrchr	Locate last occurrence of character in string (function)
strspn	Get span of character set in string (function)
strstr	Locate substring (function)
strtok	Split string into tokens (function)

The <cstring> (string.h) Example

```
#include <iostream>
#include <cstring>

int main() {
    char cstr[80];
    strcpy(cstr, "these ");
    strcat(cstr, "strings ");
    strcat(cstr, "are ");
    strcat(cstr, "concatenated.");
    std::cout << cstr << " length = " << strlen(cstr);
    return 0;
}
```

The <cstring> (string.h) Exercise

Exercise: what is the output?

```
#include <iostream>
#include <cstring>
using std::cout; using std::endl;

int main() {
    char cstr[] = "hello";
    cout << cstr << endl
         << sizeof cstr << endl
         << strlen(cstr) << endl;
    strcpy(cstr, "hello world");
    cout << cstr << endl
         << sizeof cstr << endl
         << strlen(cstr) << endl;
    return 0;
}
```

sizeof x returns the actual memory size of variable **x**.

Using Strings as Abstract Values

- Ever since the very first program in the text, which displayed the message "`hello, world`" on the screen, you have been using strings to communicate with the user.
- Up to now, you have not had any idea how C++ represents strings inside the computer or how you might manipulate the characters that make up a string.
- At the same time, the fact that you don't know those things has not compromised your ability to use strings effectively because you have been able to think of strings holistically as if they were a primitive type.
- For most applications, the abstract view of strings you have held up to now is precisely the right one. On the inside, strings are surprisingly complicated objects whose details are better left hidden.

Using Strings as Abstract Values

- As a design consequence, C++ must retain the older **`char`** type and string model it inherits from its predecessor, which is achieved by including in the C++ standard library the C standard library, such as **`<cctype>`** (**`ctype.h`**); **`<cstring>`** (**`string.h`**).
- As a newly designed language, especially one to extend C with the object-oriented programming paradigm, C++ supports a higher-level view of strings, as ***objects***.
- A C++ library **`<string>`** provides a convenient ***abstraction*** for working with strings of characters by making **`string`** a ***class*** whose ***methods*** hide the underlying complexity.
- A ***class*** is the term for data types that support the object-oriented programming paradigm. The operations that apply to instances of a class are called ***methods***.

Exercise

What are the meaning of the following statements?

```
#include <cstring>
```
// the C string library, used in C++

```
#include <string.h>
```
// the C string library, used in C, acceptable in C++

```
#include "string.h"
```
// some compilers might still find the C string library, unless you defined **`string.h`** by yourself, which will cause conflict

```
#include <string>
```
// the C++ string library

```
#include "cstring.h"
```
// incorrect unless you defined **`cstring.h`** by yourself

```
#include <cstring.h>
```
// most likely an error even if you defined **`cstring.h`** by yourself

The Hello Name Program

FIGURE 3-1 An interactive version of the "Hello World" program

```
/*
 * File: HelloName.cpp
 * -------------------
 * This program extends the classic "Hello world" program by asking
 * the user for a name, which is then used as part of the greeting.
 * This version of the program reads a complete line into name and
 * not just the first word.
 */

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

int main() {
   string name;
   cout << "Enter your full name: ";
   cin >> name;
   cout << "Hello, " << name << "!" << endl;
   return 0;
}
```

Operators in the `string` Class

`str[i]` Returns the **`i`**th character of **`str`**. Assigning to **`str[i]`** changes that character.
`s1 + s2` Returns a new string consisting of **`s1`** concatenated with **`s2`**.
`s1 = s2;` Copies the character string **`s2`** into **`s1`**.
`s1 += s2;` Appends **`s2`** to the end of **`s1`**.
`s1 == s2` (and similarly for **`<`**, **`<=`**, **`>`**, **`>=`**, and **`!=`**) Compares to strings lexicographically.
`str.c_str()` Returns a C-style character array containing the same characters as **`str`**.

- Unlike most languages, C++ allows classes to redefine the meanings of the standard operators. As a result, several string operations, such as **`+`** for concatenation, are implemented as operators (overloading).
- To convert the C++ string objects into C string literals, simply apply the **`c_str`** method to the C++ string.

Concatenation and C Strings

- As those of you who have studied Python already know, the **+** operator is a convenient shorthand for ***concatenation*** for the string objects, which consists of combining two strings end to end with no intervening characters.
- In Python, the **+** operator is often used to combine items as part of a **`print`** call, as in

```
print("Hello, " + name + "!");
```

- In C++, you achieve the same result using the **<<** operator:

```
cout << "Hello, " << name << "!" << endl;
```

- Although you might imagine otherwise, you *can't* always use the **+** operator in this statement, depending on whether **`name`** is a C string (literal) or a C++ string object.

```
cout << "Hello, " + name + "!" << endl;
```

C string literal vs. C++ string object

- C++ automatically converts a C string literal to a C++ string object whenever the compiler can determine that what you want is a C++ string object:

```
string str = "hello, world";
```

- By contrast, C++ does not allow you to write the declaration:

```
string str = "hello" + ", " + "world";
```

- The + operator cannot be applied to C string literals. To get around this problem, you can explicitly convert a string literal to a string object by calling **`string`** on the literal:

```
string str = string("hello") + ", " + "world";
```

```
string str = "hello" + string(", ") + "world";
```

```
string str = "hello" + ", " + string("world");
```

C string literal vs. C++ string object

Exercise: what is the output?

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

int main() {
    char name1[] = "Ray";
    std::string name2 = "Ray";
    std::cout << "Hello, " << name1 << std::endl;
    std::cout << "Hello, " << name2 << std::endl;
    std::cout << "Hello, " + name1 << std::endl;
    std::cout << "Hello, " + name2 << std::endl;
    return 0;
}
```

Common Methods in the `string` Class

`str.length()` Returns the number of characters in the string **`str`**.
`str.at(index)` Returns the character at position **`index`**; most clients use **`str[index]`** instead.
`str.substr(pos, len)` Returns the substring of **`str`** starting at **`pos`** and continuing for **`len`** characters.
`str.find(ch, pos)` Returns the first index ≥ **`pos`** containing **`ch`**, or **`string::npos`** if not found.
`str.find(text, pos)` Similar to the previous method, but with a string instead of a character.

Calling String Methods

- Because **`string`** is a class, it is best to think of its methods in terms of sending a message to a particular object. The object to which a message is sent is called the ***receiver***, and the general syntax for sending a message looks like this:

```
receiver.name(arguments);
```

- For example, if you want to determine the length of a string **`str`**, the object-oriented version of the statement that sets **`len`** to the length of the string object **`str`** is therefore

```
int len = str.length();
```

- You might also be tempted to use the **`strlen`** function we have used before with the C string literal:

```
int len = strlen(str);
```

After all, Python does have both: **`len(str)`** or **`str.__len__()`**.

Calling String Methods

Exercise: how do you determine the length of a string **`str`**?

- The object-oriented version:

```
int len = str.length();
```

- If you must use the **`strlen`** function from C:

```
int len = strlen(str.c_str());
```

The <string> Library Example

```
#include <iostream>
#include <cstring>

int main() {
    char str[80];
    strcpy(str, "these ");
    strcat(str, "strings ");
    strcat(str, "are ");
    strcat(str, "concatenated.");
    std::cout << str << " length = " << strlen(str);
    return 0;
}
```

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

int main() {
    std::string str;
    str = "these ";
    str = str + "strings " + "are " + "concatenated.";
    std::cout << str << " length = " << str.length();
    return 0;
}
```

Programming Paradigms

- A ***programming paradigm*** is a "style" or "way" of programming.
- One of the characteristics of a programming language is its support for particular paradigms. Some languages make it easy to write in some paradigms but not others.
- A language purposely designed to allow programming in many paradigms is called a multi-paradigm programming language, e.g., C++, Python. You can write programs or libraries that are largely *procedural*, *object-oriented*, or *functional* (i.e., some typical paradigms) in these languages.
- In a large program, even different sections can be written in different paradigms.
- C++ supports the *procedural* and *object-oriented* paradigms naturally, supports the *functional* paradigm through the **`<functional>`** interface, and supports many other paradigms through various external libraries.

The Imperative Programming Paradigm

- **Imperative** programming paradigm: an explicit sequence of statements that change a program's state, specifying how to achieve the result.
  - **Structured**: Programs have clean, **`goto`**-free, nested control structures.
  - **Procedural**: Imperative programming with procedures operating on data.
  - **Object-Oriented**: Objects have/combine states/data and behavior/methods; Computation is effected by sending messages to objects (receivers).
    - **Class-based**: Objects get their states and behavior based on membership in a class.
    - **Prototype-based**: Objects get their behavior from a prototype object.

Strings as an Abstract Data Type

- Because C++ includes everything in its predecessor language, C strings are a part of C++, and you will occasionally have to recognize that this style of string exists.
- For almost every program you write, it will be far easier to use the C++'s **`string`** class, which implements strings as an ***abstract data type***, which is defined by its behavior rather than its representation. All programs that use the string class must include the **`<string>`** library interface.
- The methods C++ provides for working with strings are often subtly different from those in Python's **`String`** type. Most of these differences fall into the *accidental* category. The only *essential* difference in these models is that C++ allows clients to change the individual characters contained in a string. By contrast, Python strings are ***immutable***, which means that they never change once they are allocated.

Selecting characters in strings

- The **<string>** library offers two different mechanisms for selecting individual characters from a string:

```
str[index]
```

```
str.at(index)
```

- The only difference is that **at** checks to make sure the index is in range, **0~str.length()-1**.
- Both methods can be used to assign a new value to the character:

```
str[index] = 'H';
```

```
str.at(index) = 'H';
```

- The former has better readability while the latter has range-checking.

Iterating through the characters in strings

- When you work with strings, one of the most important patterns involves iterating through the characters in a string, which requires the following code:

```
for (int i = 0; i < str.length(); i++) {
    . . . body of loop that manipulates str[i] . . .
}
```

```
for (int i = str.length() - 1; i >= 0; i--)
```

- The following function reverses the argument string so that, calling `reverse("desserts")` returns `"stressed"`:

```
string reverse(string str) {
    string rev = "";
    for (int i = str.length() - 1; i >= 0; i--) {
        rev += str[i];
    }
    return rev;
}
```

Is this efficient?

Modifying the Contents of a String

- In many languages, including Python, Java, C#, strings are ***immutable***, which means that they never change once they are allocated.
- C++, by contrast, allows clients to change the contents of a string, both by assigning new values to selected characters and by calling string methods such as the following:

`str.erase(pos, count)` Deletes **`count`** characters from **`str`** starting at position **`pos`**.	←*Destructively changes* **`str`**
`str.insert(pos, text)` Inserts the characters from **`text`** into **`str`** starting at position **`pos`**.	←*Destructively changes* **`str`**
`str.replace(pos, count, text)` Replaces **`count`** characters in str with **`text`** starting at position **`pos`**.	←*Destructively changes* **`str`**

Modifying the Contents of a String

- As a tool for writing programs that are easier to debug and maintain, immutable strings have many advantages over their modifiable counterparts in C++. Fortunately, it is easy to secure these advantages in C++ by avoiding the use of destructive operations like **`erase`**, **`insert`**, **`replace`**, and assignment to individual characters.
- Example: Case conversion without changing the original (safe, but is it efficient?)

```
string toUpperCase(string str) {
   string result = "";
   for (int i = 0; i < str.length(); i++) {
      result += toupper(str[i]);
   }
   return result;
}
```

Avoiding the use of destructive operations

- Example: Case conversion both safely and efficiently (why?)

```
string toUpperCase(string str) {
   for (int i = 0; i < str.length(); i++) {
      str[i] = toupper(str[i]);
   }
   return str;
}
```

- Exercise: Case conversion in place (most efficient but not safe, why?)

```
void toUpperCaseInPlace(string & str) {
   for (int i = 0; i < str.length(); i++) {
      str[i] = toupper(str[i]);
   }
}
```

- Question: can we implement **`reverse`** the same way?

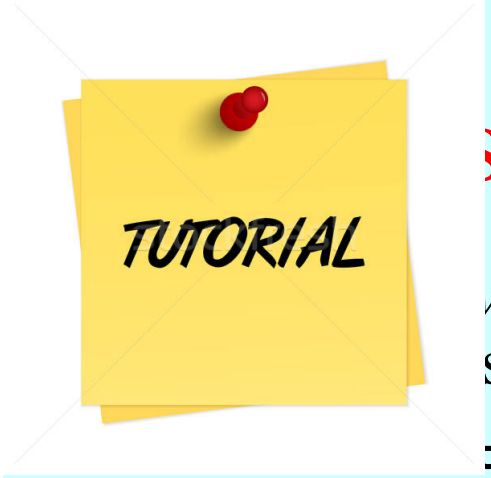
Exercise: Recognizing Palindromes

- A ***palindrome*** is a word that reads identically backward and forward, such as "level" or "noon".
- Write a C++ program **`isPalindrome`** that checks whether a string is a palindrome.

```
bool isPalindrome(string str) {
   int n = str.length();
   for (int i = 0; i < n / 2; i++) {
      if (str[i] != str[n - i - 1]) return false;
   }
   return true;
}
```

```
bool isPalindrome(string str) {
   return str == reverse(str);
}
```

- Efficiency vs. Readability

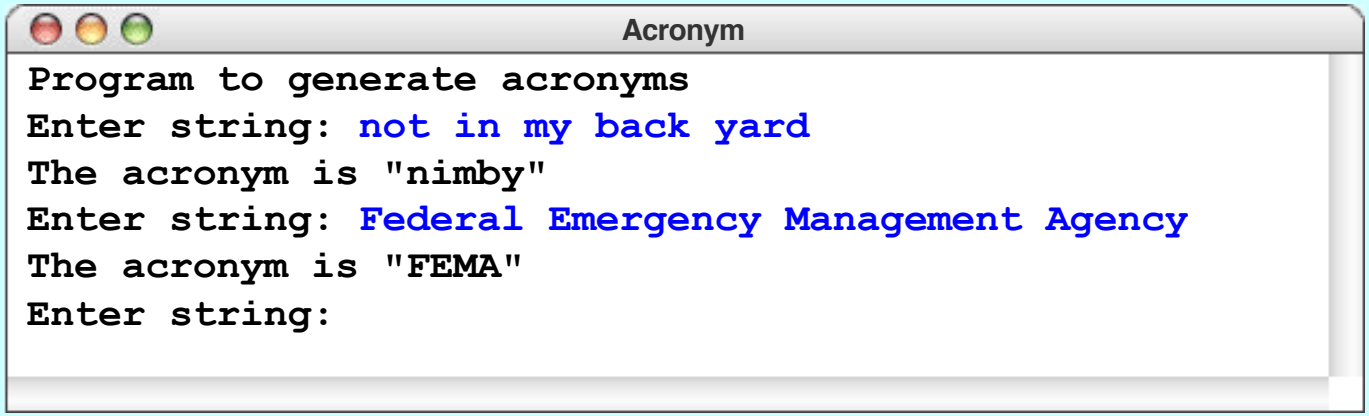


se: Writing String Applications

ym is a word formed by taking the first letter of each sequence, as in

`tained underwater breathing apparatus" → "scuba"`

- Write a C++ program that generates acronyms, as illustrated by the following sample run:



- More examples:
  - Translating English to Pig Latin

The Stanford `strlib.h` Interface

`integerToString``(`*n*`)`	Converts *n* to a C++ string.
`stringToInteger``(`*str*`)`	Converts the digits in *str* to an integer.
`realToString``(`*d*`)`	Converts *d* to a C++ string.
`stringToReal``(`*str*`)`	Converts the digits in *str* to floating point.
`toUpperCase``(`*x*`)`	Converts *str* to upper case.
`toLowerCase``(`*x*`)`	Converts *str* to lower case.
`equalsIgnoreCase``(`s_1`,` s_2`)`	Compares s_1 and s_2 without regard to case.
`startsWith``(`*str*`,` *prefix*`)`	Returns **`true`** if *str* starts with *prefix*.
`endsWith``(`*str*`,` *suffix*`)`	Returns **`true`** if *str* ends with *suffix*.
`trim``(`*str*`)`	Returns a string removing spaces from the ends.

The End