20

Strings

Prefer the standard to the offbeat.
– Strunk & White

Strings — characters — ***char_traits*** — ***basic_string*** — iterators — element access — constructors — error handling — assignment — conversions — comparisons — insertion — concatenation — find and replace — size and capacity — string I/O — C-style strings — character classification — C library functions — advice — exercises.

20.1 Introduction [string.intro]

A string is a sequence of characters. The standard library ***string*** provides string manipulation operations such as subscripting (§20.3.3), assignment (§20.3.6), comparison (§20.3.8), appending (§20.3.9), concatenation (§20.3.10), and searching for substrings (§20.3.11). No general substring facility is provided by the standard, so one is provided here as an example of standard string use (§20.3.11). A standard string can be a string of essentially any kind of character (§20.2).

Experience shows that it is impossible to design the perfect ***string***. People's taste, expectations, and needs differ too much for that. So, the standard library ***string*** isn't ideal. I would have made some design decisions differently, and so would you. However, it serves many needs well, auxiliary functions to serve further needs are easily provided, and ***std*::*string*** is generally known and available. In most cases, these factors are more important than any minor improvement we could provide. Writing string classes has great educational value (§11.12, §13.2), but for code meant to be widely used, the standard library ***string*** is the one to use.

From C, C++ inherited the notion of strings as zero-terminated arrays of ***char*** and a set of functions for manipulating such C-style strings (§20.4.1).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

20.2 Characters [string.char]

''Character'' is itself an interesting concept. Consider the character *C*. The *C* that you see as a curved line on the page (or screen), I typed into my computer many months ago. There, it lives as the numeric value *67* in an 8-bit byte. It is the third letter in the Latin alphabet, the usual abbreviation for the sixth atom (Carbon), and, incidentally, the name of a programming language (§1.6). What matters in the context of programming with strings is that there is a correspondence between squiggles with conventional meaning, called characters, and numeric values. To complicate matters, the same character can have different numeric values in different character sets, not every character set has values for every character, and many different character sets are in common use. A character set is a mapping between a character (some conventional symbol) and an integer value.

C++ programmers usually assume that the standard American character set (ASCII) is available, but C++ makes allowances for the possibility that some characters may be missing in a programmer's environment. For example, in the absence of characters such as [and {, keywords and digraphs can be used (§C.3.1).

Character sets with characters not in ASCII offer a greater challenge. Languages such as Chinese, Danish, French, Icelandic, and Japanese cannot be written properly using ASCII only. Worse, the character sets used for these languages can be mutually incompatible. For example, the characters used for European languages using Latin alphabets *almost* fit into a 256-character character set. Unfortunately, different sets are still used for different languages and some different characters have ended up with the same integer value. For example, French (using Latin1) doesn't coexist well with Icelandic (which therefore requires Latin2). Ambitious attempts to present every character known to man in a single character set have helped a lot, but even 16-bit character sets – such as Unicode – are not enough to satisfy everyone. The 32-bit character sets that could – as far as I know – hold every character are not widely used.

Basically, the C++ approach is to allow a programmer to use any character set as the character type in strings. An extended character set or a portable numeric encoding can be used (§C.3.3).

20.2.1 Character Traits [string.traits]

As shown in §13.2, a string can, in principle, use any type with proper copy operations as its character type. However, efficiency can be improved and implementations can be simplified for types that don't have user-defined copy operations. Consequently, the standard ***string*** requires that a type used as its character type does not have user-defined copy operations. This also helps to make I/O of strings simple and efficient.

The properties of a character type are defined by its ***char_traits***. A ***char_traits*** is a specialization of the template:

```
template<class Ch> struct char_traits { };
```

All ***char_traits*** are defined in ***std***, and the standard ones are presented in ***<string>***. The general ***char_traits*** itself has no properties; only ***char_traits*** specializations for a particular character type have. Consider ***char_traits<char>***:

```
template<> struct char_traits<char> {
    typedef char char_type;          // type of character
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        static void assign(char_type&, const char_type&);       // = for char_type

        // integer representation of characters:

        typedef int int_type;           // type of integer value of character

        static char_type to_char_type(const int_type&);         // int to char conversion
        static int_type to_int_type(const char_type&);          // char to int conversion
        static bool eq_int_type(const int_type&, const int_type&); // ==

        // char_type comparisons:

        static bool eq(const char_type&, const char_type&);     // ==
        static bool lt(const char_type&, const char_type&);     // <

        // operations on s[n] arrays:

        static char_type* move(char_type* s, const char_type* s2, size_t n);
        static char_type* copy(char_type* s, const char_type* s2, size_t n);
        static char_type* assign(char_type* s, size_t n, char_type a);

        static int compare(const char_type* s, const char_type* s2, size_t n);
        static size_t length(const char_type*);
        static const char_type* find(const char_type* s, int n, const char_type&);

        // I/O related:

        typedef streamoff off_type;     // offset in stream
        typedef streampos pos_type;     // position in stream
        typedef mbstate_t state_type;   // multi-byte stream state

        static int_type eof();                      // end-of-file
        static int_type not_eof(const int_type& i); // i unless i equals eof(); if not any value!=eof()
        static state_type get_state(pos_type p);    // multibyte conversion state of character in p
};
```

The implementation of the standard string template, ***basic_string*** (§20.3), relies on these types and functions. A type used as a character type for ***basic_string*** must provide a ***char_traits*** specialization that supplies them all.

For a type to be a ***char_type***, it must be possible to obtain an integer value corresponding to each character. The type of that integer is ***int_type***, and the conversion between it and the ***char_type*** is done by ***to_char_type()*** and ***to_int_type()***. For a ***char***, this conversion is trivial.

Both ***move(s,s2,n)*** and ***copy(s,s2,n)*** copy ***n*** characters from ***s2*** to ***s*** using ***assign(s[i],s2[i])***. The difference is that ***move()*** works correctly even if ***s2*** is in the ***[s,s+n[*** range. Thus, ***copy()*** can be faster. This mirrors the standard C library functions ***memcpy()*** and ***memmove()*** (§19.4.6). A call ***assign(s,n,x)*** assigns ***n*** copies of ***x*** into ***s*** using ***assign(s[i],x)***.

The ***compare()*** function uses ***lt()*** and ***eq()*** to compare characters. It returns an ***int***, where ***0*** represents an exact match, a negative number means that its first argument comes lexicographically before the second, and a positive number means that its first argument comes after its second. This mirrors the standard C library function ***strcmp()*** (§20.4.1).

The I/O-related functions are used by the implementation of low-level I/O (§21.6.4).

A wide character – that is, an object of type ***wchar_t*** (§4.3) – is like a ***char***, except that it takes up two or more bytes. The properties of a ***wchar_t*** are described by ***char_traits<wchar_t>***:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
template<> struct char_traits<wchar_t> {
    typedef wchar_t char_type;
    typedef wint_t int_type;
    typedef wstreamoff off_type;
    typedef wstreampos pos_type;

    // like char_traits<char>
};
```

A ***wchar_t*** is typically used to hold characters of a 16-bit character set such as Unicode.

20.3 Basic_string [string.string]

The standard library string facilities are based on the template ***basic_string*** that provides member types and operations similar to those provided by standard containers (§16.3):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class std::basic_string {
public:
    // ...
};
```

This template and its associated facilities are defined in namespace ***std*** and presented by ***<string>***.

Two ***typedef***s provide conventional names for common string types:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

The ***basic_string*** is similar to ***vector*** (§16.3), except that ***basic_string*** provides some typical string operations, such as searching for substrings, instead of the complete set of operations offered by ***vector***. A ***string*** is unlikely to be implemented by a simple array or ***vector***. Many common uses of strings are better served by implementations that minimize copying, use no free store for short strings, allow for simple modification of longer strings, etc. (see §20.6[12]). The number of ***string*** functions reflects the importance of string manipulation and also the fact that some machines provide specialized hardware instructions for string manipulation. Such functions are most easily utilized by a library implementer if there is a standard library function with similar semantics.

Like other standard library types, a ***basic_string<T>*** is a concrete type (§2.5.3, §10.3) without virtual functions. It can be used as a member when designing more sophisticated text manipulation classes, but it is not intended to be a base for derived classes (§25.2.1; see also §20.6[10]).

20.3.1 Types [string.types]

Like ***vector***, ***basic_string*** makes its related types available through a set of member type names:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // types (much like vector, list, etc.: §16.3.1):
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
    typedef Tr traits_type;          // specific to basic_string

    typedef typename Tr::char_type value_type;
    typedef A allocator_type;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;

    typedef implementation_defined iterator;
    typedef implementation_defined const_iterator;

    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // ...
};
```

The ***basic_string*** notion supports strings of many kinds of characters in addition to the simple ***basic_string<char>*** known as ***string***. For example:

```
typedef basic_string<unsigned char> Ustring;

struct Jchar { /* ... */ };        // Japanese character type
typedef basic_string<Jchar> Jstring;
```

Strings of such characters can be used just like strings of ***char*** as far as the semantics of the characters allows. For example:

```
Ustring first_word(const Ustring& us)
{
    Ustring::size_type pos = us.find(' ');      // see §20.3.11
    return Ustring(us,0,pos);                   // see §20.3.4
}

Jstring first_word(const Jstring& js)
{
    Jstring::size_type pos = js.find(' ');      // see §20.3.11
    return Jstring(js,0,pos);                   // see §20.3.4
}
```

Naturally, templates that take string arguments can also be used:

```
template<class S> S first_word(const S& s)
{
    typename S::size_type pos = s.find(' '); // see §20.3.11
    return S(s,0,pos);                       // see §20.3.4
}
```

A ***basic_string<Ch>*** can contain any character of the set ***Ch***. In particular, ***string*** can contain a ***0*** (zero).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

20.3.2 Iterators [string.begin]

Like other containers, a ***string*** provides iterators for ordinary and reverse iteration:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // iterators (like vector, list, etc.: §16.3.2):

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // ...
};
```

Because ***string*** has the required member types and the functions for obtaining iterators, ***string***s can be used together with the standard algorithms (Chapter 18). For example:

```
void f(string& s)
{
    string::iterator p = find(s.begin(),s.end(),'a');
    // ...
}
```

The most common operations on ***string***s are supplied directly by ***string***. Hopefully, these versions will be optimized for ***string***s beyond what would be easy to do for general algorithms.

The standard algorithms (Chapter 18) are not as useful for strings as one might think. General algorithms tend to assume that the elements of a container are meaningful in isolation. This is typically not the case for a string. The meaning of a string is encoded in its exact sequence of characters. Thus, sorting a string (that is, sorting the characters in a string) destroys its meaning, whereas sorting a general container typically makes it more useful.

The ***string*** iterators are not range checked.

20.3.3 Element Access [string.elem]

Individual characters of a ***string*** can be accessed through subscripting:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // element access (like vector: §16.3.3):
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
    const_reference operator[](size_type n) const; // unchecked access
    reference operator[](size_type n);

    const_reference at(size_type n) const;          // checked access
    reference at(size_type n);

    // ...
};
```

Out-of-range access causes ***at()*** to throw an ***out_of_range***.

Compared to ***vector***, ***string*** lacks ***front()*** and ***back()***. To refer to the first and the last character of a ***string***, we must say ***s[0]*** and ***s[s.length()-1]***, respectively. The pointer/array equivalence (§5.3) doesn't hold for ***strings***. If ***s*** is a ***string***, ***&s[0]*** is not the same as ***s***.

20.3.4 Constructors [string.ctor]

The set of initialization and copy operations for a ***string*** differs from what is provided for other containers (§16.3.4) in many details:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // constructors, etc. (a bit like vector and list: §16.3.4):

    explicit basic_string(const A& a = A());
    basic_string(const basic_string& s,
                 size_type pos = 0, size_type n = npos, const A& a = A());
    basic_string(const Ch* p, size_type n, const A& a = A());
    basic_string(const Ch* p, const A& a = A());
    basic_string(size_type n, Ch c, const A& a = A());
    template<class In> basic_string(In first, In last, const A& a = A());

    ~basic_string();

    static const size_type npos;     // ''all characters'' marker

    // ...
};
```

A ***string*** can be initialized by a C-style string, by another ***string***, by part of a C-style string, by part of a ***string***, or from a sequence of characters. However, a ***string*** cannot be initialized by a character or an integer:

```
void f(char* p, vector<char>&v)
{
    string s0;              // the empty string
    string s00 = "";        // also the empty string

    string s1 = 'a';        // error: no conversion from char to string
    string s2 = 7;          // error: no conversion from int to string
    string s3(7);           // error: no constructor taking one int argument
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
    string s4(7,´a´);          // 7 copies of 'a'; that is "aaaaaaa"

    string s5 = "Frodo";       // copy of "Frodo"
    string s6 = s5;            // copy of s5

    string s7(s5,3,2);         // s5[3] and s5[4]; that is "do"
    string s8(p+7,3);          // p[7], p[8], and p[9]
    string s9(p,7,3);          // string(string(p),7,3), possibly expensive

    string s10(v.begin(),v.end());     // copy all characters from v
}
```

Characters are numbered starting at position ***0*** so that a string is a sequence of characters numbered ***0*** to ***length()-1***.

The ***length()*** of a string is simply a synonym for its ***size()***; both functions return the number of characters in the string. Note that they do not count a C-string-style, zero-terminator character (§20.4.1). An implementation of ***basic_string*** stores its length rather than relying on a terminator.

Substrings are expressed as a character position plus a number of characters. The default value ***npos*** is initialized to the largest possible value and used to mean ''all of the elements.''

There is no constructor for creating a string of ***n*** unspecified characters. The closest we come to that is the constructor that makes a string of ***n*** copies of a given character. The length of a string is determined by the number of characters it holds at any give time. This allows the compiler to save the programmer from silly mistakes such as the definitions of ***s2*** and ***s3*** in the previous example.

The copy constructor is the constructor taking four arguments. Three of those arguments have defaults. For efficiency, that constructor could be implemented as two separate constructors. The user wouldn't be able to tell without actually looking at the generated code.

The constructor that is a template member is the most general. It allows a string to be initialized with values from an arbitrary sequence. In particular, it allows a string to be initialized with elements of a different character type as long as a conversion exists. For example:

```
void f(string s)
{
    wstring ws(s.begin(),s.end());     // copy all characters from s
    // ...
}
```

Each ***wchar_t*** in ***ws*** is initialized by its corresponding ***char*** from ***s***.

20.3.5 Errors [string.error]

Often, strings are simply read, written, printed, stored, compared, copied, etc. This causes no problems, or, at worst, performance problems. However, once we start manipulating individual substrings and characters to compose new string values from existing ones, we sooner or later make mistakes that could cause us to write beyond the end of a string.

For explicit access to individual characters, ***at()*** checks and throws ***out_of_range()*** if we try to access beyond the end of the string; `[]` does not.

Most string operations take a character position plus a number of characters. A position larger than the size of the string throws an ***out_of_range*** exception. A ''too large'' character count is simply taken to be equivalent to ''the rest'' of the characters. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void f()
{
    string s = "Snobol4";
    string s2(s,100,2); // character position beyond end of string: throw out_of_range()
    string s3(s,2,100); // character count too large: equivalent to s3(s,2,s.size()–2)
    string s4(s,2,string::npos);    // the characters starting from s[2]
}
```

Thus, ''too large'' positions are to be avoided, but ''too large'' character counts are useful. In fact, ***npos*** is really just the largest possible value for ***size_type***.

We could try to give a negative position or character count:

```
void g(string& s)
{
    string s5(s,-2,3);  // large position!: throw out_of_range()
    string s6(s,3,-2);  // large character count!: ok
}
```

However, the ***size_type*** used to represent positions and counts is an ***unsigned*** type, so a negative number is simply a confusing way of specifying a large positive number (§16.3.4).

Note that the functions used to find substrings of a ***string*** (§20.3.11) return ***npos*** if they don't find anything. Thus, they don't throw exceptions. However, later using ***npos*** as a character position does.

A pair of iterators is another way of specifying a substring. The first iterator identifies a position, and the difference between two iterators is a character count. As usual, iterators are not range checked.

Where a C-style string is used, range checking is harder. When given a C-style string (a pointer to ***char***) as an argument, ***basic_string*** functions assume the pointer is not ***0***. When given character positions for C-style strings, they assume that the C-style string is long enough for the position to be valid. Be careful! In this case, being careful means being paranoid, except when using character literals.

All strings have ***length()<npos***. In a few cases, such as inserting one string into another (§20.3.9), it is possible (although not likely) to construct a string that is too long to be represented. In that case, a ***length_error*** is thrown. For example:

```
string s(string::npos,´a´);     // throw length_error()
```

20.3.6 Assignment [string.assign]

Naturally, assignment is provided for strings:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // assignment (a bit like vector and list: §16.3.4):
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
    basic_string& operator=(const basic_string& s);
    basic_string& operator=(const Ch* p);
    basic_string& operator=(Ch c);

    basic_string& assign(const basic_string&);
    basic_string& assign(const basic_string& s, size_type pos, size_type n);
    basic_string& assign(const Ch* p, size_type n);
    basic_string& assign(const Ch* p);
    basic_string& assign(size_type n, Ch c);
    template<class In> basic_string& assign(In first, In last);

    // ...
};
```

Like other standard containers, ***string***s have value semantics. That is, when one string is assigned to another, the assigned string is copied and two separate strings with the same value exist after the assignment. For example:

```
void g()
{
    string s1 = "Knold";
    string s2 = "Tot";

    s1 = s2;          // two copies of "Tot"
    s2[1] = 'u';      // s2 is "Tut", s1 is still "Tot"
}
```

Assignment with a single character to a string is supported even though initialization by a single character isn't:

```
void f()
{
    string s = 'a'; // error: initialization by char
    s = 'a';        // ok: assignment
    s = "a";
    s = s;
}
```

Being able to assign a ***char*** to a ***string*** isn't much use and could even be considered error-prone. However, appending a ***char*** using `+=` is at times essential (§20.3.9), and it would be odd to be able to say `s+='c'` but not `s=s+'c'`.

The name ***assign***`()` is used for the assignments, which are the counterparts to multiple argument constructors (§16.3.4, §20.3.4).

As mentioned in §11.12, it is possible to optimize a ***string*** so that copying doesn't actually take place until two copies of a ***string*** are needed. The design of the standard ***string*** encourages implementations that minimize actual copying. This makes read-only uses of strings and passing of strings as function arguments much cheaper than one could naively have assumed. However, it would be equally naive for programmers not to check their implementations before writing code that relied on ***string*** copy being optimized (§20.6[13]).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

20.3.7 Conversion to C-Style Strings [string.conv]

As shown in §20.3.4, a ***string*** can be initialized by a C-style string and C-style strings can be assigned to ***string***s. Conversely, it is possible to place a copy of the characters of a ***string*** into an array:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // conversion to C-style string:

    const Ch* c_str() const;
    const Ch* data() const;
    size_type copy(Ch* p, size_type n, size_type pos = 0) const;

    // ...
};
```

The ***data()*** function writes the characters of the string into an array and returns a pointer to that array. The array is owned by the ***string***, and the user should not try to delete it. The user also cannot rely on its value after a subsequent call on a non-***const*** function on the string. The ***c_str()*** function is like ***data()***, except that it adds a 0 (zero) at the end as a C-string-style terminator. For example:

```
void f()
{
    string s = "equinox";           // s.length()==7
    const char* p1 = s.data();      // p1 points to seven characters
    printf("p1 = %s\n",p1);         // bad: missing terminator
    p1[2] = 'a';                    // error: p1 points to a const array
    s[2] = 'a';
    char c = p1[1];                 // bad: access of s.data() after modification of s

    const char* p2 = s.c_str();     // p2 points to eight characters
    printf("p2 = %s\n",p2);         // ok: c_str() adds terminator
}
```

In other words, ***data()*** produces an array of characters, whereas ***c_str()*** produces a C-style string. These functions are primarily intended to allow simple use of functions that take C-style strings. Consequently, ***c_str()*** tends to be more useful than ***data()***. For example:

```
void f(string s)
{
    int i = atoi(s.c_str());    // get int value of digits in string (§20.4.1)
    // ...
}
```

Typically, it is best to leave characters in a ***string*** until you need them. However, if you can't use the characters immediately, you can copy them into an array rather than leave them in the buffer allocated by ***c_str()*** or ***data()***. The ***copy()*** function is provided for that. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
char* c_string(const string& s)
{
    char* p = new char[s.length()+1]; // note: +1
    s.copy(p,string::npos);
    p[s.length()] = 0;                // note: add terminator
    return p;
}
```

A call `s.copy(p,n,m)` copies at most `n` characters to `p` starting with `s[m]`. If there are fewer than `n` characters in `s` to copy, `copy()` simply copies all the characters there are.

Note that a `string` can contain the `0` character. Functions manipulating C-style strings will interprete such as `0` as a terminator. Be careful to put `0`s into a string only if you don't apply C-style functions to it or if you put the `0` there exactly to be a terminator.

Conversion to a C-style string could have been provided by an `operator const char*()` rather than `c_str()`. This would have provided the convenience of an implicit conversion at the cost of surprises in cases in which such a conversion was unexpected.

If you find `c_str()` appearing in your program with great frequency, it is probably because you rely heavily on C-style interfaces. Often, an interface that relies on `string`s rather than C-style strings is available and can be used to eliminate the conversions. Alternatively, you can avoid most of the explicit calls of `c_str()` by providing additional definitions of the functions that caused you to write the `c_str()` calls:

```
extern "C" int atoi(const char*);

int atoi(const string& s)
{
    return atoi(s.c_str());
}
```

20.3.8 Comparisons [string.compare]

Strings can be compared to strings of their own type and to arrays of characters with the same character type:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...

    int compare(const basic_string& s) const; // combined > and ==
    int compare(const Ch* p) const;

    int compare(size_type pos, size_type n, const basic_string& s) const;
    int compare(size_type pos, size_type n,
                const basic_string& s, size_type pos2, size_type n2) const;
    int compare(size_type pos, size_type n, const Ch* p, size_type n2 = npos) const;

    // ...
};
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

When an argument *n* is supplied, only the *n* first characters will be compared. The comparison criterion used is *char_traits<Ch>*'s *compare()* (§20.2.1). Thus, *s.compare(s2)* returns *0* if the strings have the same value, a negative number if *s* is lexicographically before *s2*, and a positive number otherwise.

A user cannot supply a comparison criterion the way it was done in §13.4. When that degree of flexibility is needed, we can use *lexicographical_compare()* (§18.9), define a function like the one in §13.4, or write an explicit loop. For example, the *toupper()* function (§20.4.2) allows us to write case-insensitive comparisons:

```
int cmp_nocase(const string& s, const string& s2)
{
    string::const_iterator p = s.begin();
    string::const_iterator p2 = s2.begin();

    while (p!=s.end() && p2!=s2.end()) {
        if (toupper(*p)!=toupper(*p2)) return (toupper(*p)<toupper(*p2)) ? -1 : 1;
        ++p;
        ++p2;
    }

    return (s2.size()==s.size()) ? 0 : (s.size()<s2.size()) ? -1 : 1; // size is unsigned
}

void f(const string& s, const string& s2)
{
    if (s == s2) {                    // case sensitive compare of s and s2
        // ...
    }

    if (cmp_nocase(s,s2)== 0) {       // case insensitive compare of s and s2
        // ...
    }

    // ...
}
```

The usual comparison operators ==, !=, >, <, >=, and <= are provided for *basic_string*s:

```
template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
bool operator==(const Ch*, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch,Tr,A>&, const Ch*);

// similar declarations for !=, >, <, >=, and <=
```

Comparison operators are nonmember functions so that conversions can be applied in the same way to both operands (§11.2.3). The versions taking C-style strings are provided to optimize comparisons against string literals. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
void f(const string& name)
{
    if (name =="Obelix" || "Asterix"==name) {    // use optimized ==
        // ...
    }
}
```

20.3.9 Insert [string.insert]

Once created, a string can be manipulated in many ways. Of the operations that modify the value of a string, is one of the most common is appending to it – that is, adding characters to the end. Insertion at other points of a string is rarer:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // add characters after (*this)[length()–1]:

    basic_string& operator+=(const basic_string& s);
    basic_string& operator+=(const Ch* p);
    basic_string& operator+=(Ch c);
    void push_back(Ch c);

    basic_string& append(const basic_string& s);
    basic_string& append(const basic_string& s, size_type pos, size_type n);
    basic_string& append(const Ch* p, size_type n);
    basic_string& append(const Ch* p);
    basic_string& append(size_type n, Ch c);
    template<class In> basic_string& append(In first, In last);

    // insert characters before (*this)[pos]:

    basic_string& insert(size_type pos, const basic_string& s);
    basic_string& insert(size_type pos, const basic_string& s, size_type pos2, size_type n);
    basic_string& insert(size_type pos, const Ch* p, size_type n);
    basic_string& insert(size_type pos, const Ch* p);
    basic_string& insert(size_type pos, size_type n, Ch c);

    // insert characters before p:

    iterator insert(iterator p, Ch c);
    void insert(iterator p, size_type n, Ch c);
    template<class In> void insert(iterator p, In first, In last);

    // ...
};
```

Basically, the variety of operations provided for initializing a string and assigning to a string is also available for appending and for inserting characters before some character position.

The `+=` operator is provided as the conventional notation for the most common forms of append. For example:

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
string complete_name(const string& first_name, const string& family_name)
{
    string s = first_name;
    s += ' ';
    s += family_name;
    return s;
}
```

Appending to the end can be noticeably more efficient than inserting into other positions. For example:

```
string complete_name2(const string& first_name, const string& family_name) // poor algorithm
{
    string s = family_name;
    s.insert(s.begin(), ' ');
    return s.insert(0, first_name);
}
```

Insertion usually forces the ***string*** implementation to do extra memory management and to move characters around.

Because ***string*** has a ***push_back()*** operation (§16.3.5), a ***back_inserter*** can be used for a ***string*** exactly as for general containers.

20.3.10 Concatenation [string.cat]

Appending is a special form of concatenation. *Concatenation* – constructing a string out of two strings by placing one after the other – is provided by the + operator:

```
template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A>
operator+(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const Ch*, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(Ch, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, const Ch*);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, Ch);
```

As usual, + is defined as a nonmember function. For templates with several template parameters, this implies a notational disadvantage, since the template parameters are mentioned repeatedly.

On the other hand, use of concatenation is obvious and convenient. For example:

```
string complete_name3(const string& first_name, const string& family_name)
{
    return first_name + ' ' + family_name;
}
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

This notational convenience may be bought at the cost of some run-time overhead compared to *complete_name*( ). One extra temporary (§11.3.2) is needed in *complete_name3*( ). In my experience, this is rarely important, but it is worth remembering when writing an inner loop of a program where performance matters. In that case, we might even consider avoiding a function call by making *complete_name*( ) inline and composing the result string in place using lower-level operations (§20.6[14]).

20.3.11 Find [string.find]

There is a bewildering variety of functions for finding substrings:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // find subsequence (like search() §18.5.5):

    size_type find(const basic_string& s, size_type i = 0) const;
    size_type find(const Ch* p, size_type i, size_type n) const;
    size_type find(const Ch* p, size_type i = 0) const;
    size_type find(Ch c, size_type i = 0) const;

    // find subsequence searching backwards from the end (like find_end(), §18.5.5):

    size_type rfind(const basic_string& s, size_type i = npos) const;
    size_type rfind(const Ch* p, size_type i, size_type n) const;
    size_type rfind(const Ch* p, size_type i = npos) const;
    size_type rfind(Ch c, size_type i = npos) const;

    // find character (like find_first_of() in §18.5.2):

    size_type find_first_of(const basic_string& s, size_type i = 0) const;
    size_type find_first_of(const Ch* p, size_type i, size_type n) const;
    size_type find_first_of(const Ch* p, size_type i = 0) const;
    size_type find_first_of(Ch c, size_type i = 0) const;

    // find character from argument searching backwards from the end:

    size_type find_last_of(const basic_string& s, size_type i = npos) const;
    size_type find_last_of(const Ch* p, size_type i, size_type n) const;
    size_type find_last_of(const Ch* p, size_type i = npos) const;
    size_type find_last_of(Ch c, size_type i = npos) const;

    // find character not in argument:

    size_type find_first_not_of(const basic_string& s, size_type i = 0) const;
    size_type find_first_not_of(const Ch* p, size_type i, size_type n) const;
    size_type find_first_not_of(const Ch* p, size_type i = 0) const;
    size_type find_first_not_of(Ch c, size_type i = 0) const;

    // find character not in argument searching backwards from the end:
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        size_type find_last_not_of(const basic_string& s, size_type i = npos) const;
        size_type find_last_not_of(const Ch* p, size_type i, size_type n) const;
        size_type find_last_not_of(const Ch* p, size_type i = npos) const;
        size_type find_last_not_of(Ch c, size_type i = npos) const;
        // ...
};
```

These are all ***const*** members. That is, they exist to locate a substring for some use, but they do not change the value of the string to which they are applied.

The meaning of the ***basic_string*::*find*** functions can be understood from their general algorithm equivalents. Consider an example:

```
void f()
{
        string s = "accdcde";
        string::size_type i1 = s.find("cd");                    // i1 = 2   s[2]=='c' && s[3]=='d'
        string::size_type i2 = s.rfind("cd");                   // i2 = 4   s[4]=='c' && s[5]=='d'
        string::size_type i3 = s.find_first_of("cd");           // i3 = 1   s[1] == 'c'
        string::size_type i4 = s.find_last_of("cd");            // i4 = 5   s[5] == 'd'
        string::size_type i5 = s.find_first_not_of("cd");       // i5 = 0   s[0]!='c' && s[0]!='d'
        string::size_type i6 = s.find_last_not_of("cd");        // i6 = 6   s[6]!='c' && s[6]!='d'
}
```

If a ***find*()** function fails to find anything, it returns ***npos***, which represents an illegal character position. If ***npos*** is used as a character position, ***range_error*** will be thrown (§20.3.5).

Note that result of a ***find*()** is an ***unsigned*** value.

20.3.12 Replace [string.replace]

Once a position in a string is identified, the value of individual character positions can be changed using subscripting or whole substrings can be replaced with new characters using ***replace*()**:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        // ...
        // replace [ (*this)[i], (*this)[i+n] [ with other characters:

        basic_string& replace(size_type i, size_type n, const basic_string& s);
        basic_string& replace(size_type i, size_type n,
                              const basic_string& s, size_type i2, size_type n2);
        basic_string& replace(size_type i, size_type n, const Ch* p, size_type n2);
        basic_string& replace(size_type i, size_type n, const Ch* p);
        basic_string& replace(size_type i, size_type n, size_type n2, Ch c);

        basic_string& replace(iterator i, iterator i2, const basic_string& s);
        basic_string& replace(iterator i, iterator i2, const Ch* p, size_type n);
        basic_string& replace(iterator i, iterator i2, const Ch* p);
        basic_string& replace(iterator i, iterator i2, size_type n, Ch c);
        template<class In> basic_string& replace(iterator i, iterator i2, In j, In j2);
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        // remove characters from string (''replace with nothing''):

        basic_string& erase(size_type i = 0, size_type n = npos);
        iterator erase(iterator i);
        iterator erase(iterator first, iterator last);

        // ...
};
```

Note that the number of new characters need not be the same as the number of characters previously in the string. The size of the string is changed to accommodate the new substring. In particular, ***erase***( ) simply removes a substring and adjusts its size accordingly. For example:

```
void f()
{
        string s = "but I have heard it works even if you don´t believe in it";
        s.erase(0,4);                          // erase initial "but "
        s.replace(s.find("even"),4,"only");
        s.replace(s.find("don´t"),5,"");  // erase by replacing with ""
}
```

The simple call ***erase***( ), with no argument, makes the string into an empty string. This is the operation that is called ***clear***( ) for general containers (§16.3.6).

The variety of ***replace***( ) functions matches that of assignment. After all, ***replace***( ) is an assignment to a substring.

20.3.13 Substrings [string.sub]

The ***substr***( ) function lets you specify a substring as a position plus a length:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        // ...
        // address substring:

        basic_string substr(size_type i = 0, size_type n = npos) const;
        // ...
};
```

The ***substr***( ) function is simply a way of reading a part of a string. On the other hand, ***replace***( ) lets you write to a substring. Both rely on the low-level position plus number of characters notation. However, ***find***( ) lets us find substrings by value. Together, they allow us to define a substring that can be used for both reading and writing:

```
template<class Ch> class Basic_substring {
public:
        typedef typename basic_string<Ch>::size_type size_type;

        Basic_substring(basic_string<Ch>& s, size_type i, size_type n);      // s[i]..s[i+n–1]
        Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2); // s2 in s
        Basic_substring(basic_string<Ch>& s, const Ch* p);                   // *p in s
```

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

```
        Basic_substring& operator=(const basic_string<Ch>&);          // write through to *ps
        Basic_substring& operator=(const Basic_substring<Ch>&);
        Basic_substring& operator=(const Ch*);
        Basic_substring& operator=(Ch);

        operator basic_string<Ch>() const;                            // read from *ps
        operator Ch*() const;
    private:
        basic_string<Ch>* ps;
        size_type pos;
        size_type n;
    };
```

The implementation is largely trivial. For example:

```
    template<class Ch>
    Basic_substring<Ch>::Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2)
        :ps(&s), n(s2.length())
    {
        pos = s.find(s2);
    }

    template<class Ch>
    Basic_substring<Ch>& Basic_substring<Ch>::operator=(const basic_string<Ch>& s)
    {
        ps->replace(pos,n,s);     // write through to *ps
        return *this;
    }

    template<class Ch> Basic_substring<Ch>::operator basic_string<Ch>() const
    {
        return basic_string<Ch>(*ps,pos,n);       // copy from *ps
    }
```

If *s2* isn't found in *s*, *pos* will be *npos*. Attempts to read or write it will throw *range_error* (§20.3.5).

This *Basic_substring* can be used like this:

```
    typedef Basic_substring<char> Substring;

    void f()
    {
        string s = "Mary had a little lamb";
        Substring(s,"lamb") = "fun";
        Substring(s,"a little") = "no";
        string s2 = "Joe" + Substring(s,s.find(' '),string::npos);
    }
```

Naturally, this would be much more interesting if *Substring* could do some pattern matching (§20.6[7]).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

20.3.14 Size and Capacity [string.capacity]

Memory-related issues are handled much as they are for *vector* (§16.3.8):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // size, capacity, etc. (like §16.3.8):

    size_type size() const;                          // number of characters (§20.3.4)
    size_type max_size() const;                      // largest possible string
    size_type length() const { return size(); }
    bool empty() const { return size()==0; }

    void resize(size_type n, Ch c);
    void resize(size_type n) { resize(n,Ch()); }

    size_type capacity() const;                      // like vector: §16.3.8
    void reserve(size_type res_arg = 0);             // like vector: §16.3.8

    allocator_type get_allocator() const;
};
```

A call *reserve*(*res_arg*) throws *length_error* if *res_arg*>*max_size*().

20.3.15 I/O Operations [string.io]

One of the main uses of *string*s is as the target of input and as the source of output:

```
template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&, Ch eol);

template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&);
```

The << operator writes a string to an *ostream* (§21.2.1). The >> operator reads a whitespace-terminated word (§3.6, §21.3.1) to its string, expanding the string as needed to hold the word. Initial whitespace is skipped, and the terminating whitespace character is not entered into the string.

The *getline*() function reads a line terminated by *eol* to its string, expanding string as needed to hold the line (§3.6). If no *eol* argument is provided, a newline '\n' is used as the delimiter. The line terminator is removed from the stream but not entered into the string. Because a *string* expands to hold the input, there is no reason to leave the terminator in the stream or to provide a count of characters read in the way *get*() and *getline*() do for character arrays (§21.3.4).

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

20.3.16 Swap [string.swap]

As for *vector*s (§16.3.9), a *swap*( ) function for strings can be much more efficient than the general algorithm, so a specific version is provided:

```
template<class Ch, class Tr, class A>
void swap(basic_string<Ch,Tr,A>&, basic_string<Ch,Tr,A>&);
```

20.4 The C Standard Library [string.cstd]

The C++ standard library inherited the C-style string functions from the C standard library. This section lists some of the most useful C string functions. The description is not meant to be exhaustive; for further information, check your reference manual. Beware that implementers often add their own nonstandard functions to the standard header files, so it is easy to get confused about which functions are guaranteed to be available on every implementation.

The headers presenting the standard C library facilities are listed in §16.1.2. Memory management functions can be found in §19.4.6, C I/O functions in §21.8, and the C math library in §22.3. The functions concerned with startup and termination are described in §3.2 and §9.4.1.1, and the facilities for reading unspecified function arguments are presented in §7.6. C-style functions for wide character strings are found in <*cwchar*> and <*wchar.h*>.

20.4.1 C-Style Strings [string.c]

Functions for manipulating C-style strings are found in <*string.h*> and <*cstring*>:

```
char* strcpy(char* p, const char* q);          // copy from q into p (incl. terminator)
char* strcat(char* p, const char* q);          // append from q to p (incl. terminator)
char* strncpy(char* p, const char* q, int n);  // copy n char from q into p
char* strncat(char* p, const char* q, int n);  // append n char from q to p

size_t strlen(const char* p);    // length of p (not counting the terminator)

int strcmp(const char* p, const char* q);          // compare: p and q
int strncmp(const char* p, const char* q, int n);  // compare first n char

char* strchr(char* p, int c);                  // find first c in p
const char* strchr(const char* p, int c);
char* strrchr(char* p, int c);                 // find last c in p
const char* strrchr(const char* p, int c);
char* strstr(char* p, const char* q);          // find first q in p
const char* strstr(const char* p, const char* q);

char* strpbrk(char* p, const char* q);         // find first char from q in p
const char* strpbrk(const char* p, const char* q);

size_t strspn(const char* p, const char* q);   // number of char in p before any char in q
size_t strcspn(const char* p, const char* q);  // number of char in p before a char not in q
```

A pointer is assumed to be nonzero, and the array of *char* that it points to is assumed to be terminated by *0*. The *strn*-functions pad with *0* if there are not *n* characters to copy. String comparisons

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

return *0* if the strings are equal, a negative number if the first argument is lexicographically before the second, and a positive number otherwise.

Naturally, C doesn't provide the pairs of overloaded functions. However, they are needed in C++ for *const* safety. For example:

```
void f(const char* pcc, char* pc)    // C++
{
    *strchr(pcc,'a') = 'b';  // error: cannot assign to const char
    *strchr(pc,'a') = 'b';   // ok, but sloppy: there might not be an 'a' in pc
}
```

The C++ *strchr()* does not allow you to write to a *const*. However, a C program may ''take advantage'' of the weaker type checking in the C *strchr()*:

```
char* strchr(const char* p, int c);  /* C standard library function, not C++ */

void g(const char* pcc, char* pc)    /* C, will not compile in C++ */
{
    *strchr(pcc,'a') = 'b';  /* converts const to non-const: ok in C, error in C++ */
    *strchr(pc,'a') = 'b';   /* ok in C and C++ */
}
```

Whenever possible, C-style strings are best avoided in favor of *string*s. C-style strings and their associated standard functions can be used to produce very efficient code, but even experienced C and C++ programmers are prone to make uncaught ''silly errors'' when using them. However, no C++ programmer can avoid seeing some of these functions in old code. Here is a nonsense example illustrating the most common functions:

```
void f(char* p, char* q)
{
    if (p==q) return;            // pointers are equal
    if (strcmp(p,q)==0) {        // string values are equal
        int i = strlen(p);       // number of characters (not counting the terminator)
        // ...
    }
    char buf[200];
    strcpy(buf,p);               // copy p into buf (including the terminator)
                                 // sloppy: will overflow some day.
    strncpy(buf,p,200);          // copy 200 char from p into buf
                                 // sloppy: will fail to copy the terminator some day.
    // ...
}
```

Input and output of C-style strings are usually done using the *printf* family of functions (§21.8).

In *<stdlib.h>* and *<cstdlib>*, the standard library provides useful functions for converting strings representing numeric values into numeric values:

```
double atof(const char* p);      // convert p to double
int atoi(const char* p);         // convert p to int
long atol(const char* p);        // convert p to long
```

Leading whitespace is ignored. If the string doesn't represent a number, zero is returned. For

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

example, the value of *atoi*("*seven*") is *0*. If the string represents a number that cannot be represented in the intended result type, *errno* (§16.1.2, §22.3) is set to *ERANGE* and an appropriately huge or tiny value is returned.

20.4.2 Character Classification [string.isalpha]

In <*ctype.h*> and <*cctype*>, the standard library provides a set of useful functions for dealing with ASCII and similar character sets:

```
int isalpha(int);      // letter: 'a'..'z' 'A'..'Z' in C locale (§20.2.1, §21.7)
int isupper(int);      // upper case letter: 'A'..'Z' in C locale (§20.2.1, §21.7)
int islower(int);      // lower case letter: 'a'..'z' in C locale (§20.2.1, §21.7)
int isdigit(int);      // '0'..'9'
int isxdigit(int);     // '0'..'9' or letter
int isspace(int);      // ' ' '\t' '\v' return newline formfeed
int iscntrl(int);      // control character (ASCII 0..31 and 127)
int ispunct(int);      // punctuation: none of the above
int isalnum(int);      // isalpha() | isdigit()
int isprint(int);      // printable: ascii ' '..'~'
int isgraph(int);      // isalpha() | isdigit() | ispunct()

int toupper(int c);    // uppercase equivalent to c
int tolower(int c);    // lowercase equivalent to c
```

All are usually implemented by a simple lookup, using the character as an index into a table of character attributes. This means that constructs such as:

```
if ( ('a'<=c && c<='z') || ('A'<=c && c<='Z') ) {      // alphabetic
    // ...
}
```

are inefficient in addition to being tedious to write and error-prone (on a machine with the EBCDIC character set, this will accept nonalphabetic characters).

These functions take *int* arguments, and the integer passed must be representable as an *unsigned char* or *EOF* (which is most often *-1*). This can be a problem on systems where *char* is signed (see §20.6[11]).

Equivalent functions for wide characters are found in <*cwtype*> and <*wtype.h*>.

20.5 Advice [string.advice]

[1] Prefer *string* operations to C-style string functions; §20.4.1.
[2] Use *string*s as variables and members, rather than as base classes; §20.3, §25.2.1.
[3] You can pass *string*s as value arguments and return them by value to let the system take care of memory management; §20.3.6.
[4] Use *at*() rather than iterators or [] when you want range checking; §20.3.2, §20.3.5.
[5] Use iterators and [] rather than *at*() when you want to optimize speed; §20.3.2, §20.3.5.
[6] Directly or indirectly, use *substr*() to read substrings and *replace*() to write substrings; §20.3.12, §20.3.13.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

[7] Use the *find*() operations to localize values in a *string* (rather than writing an explicit loop); §20.3.11.
[8] Append to a *string* when you need to add characters efficiently; §20.3.9.
[9] Use *string*s as targets of non-time-critical character input; §20.3.15.
[10] Use *string*::*npos* to indicate ''the rest of the *string*;'' §20.3.5.
[11] If necessary, implement heavily-used *string*s using low-level operations (rather than using low-level data structures everywhere); §20.3.10.
[12] If you use *string*s, catch *range_error* and *out_of_range* somewhere; §20.3.5.
[13] Be careful not to pass a *char** with the value *0* to a string function; §20.3.7.
[14] Use *c_str* rather to produce a C-style string representation of a *string* only when you have to; §20.3.7.
[15] Use *isalpha*(), *isdigit*(), etc., when you need to know the classification of a character rather that writing your own tests on character values; §20.4.2.

20.6 Exercises [string.exercises]

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.

1. (∗2) Write a function that takes two *string*s and returns a *string* that is the concatenation of the strings with a dot in the middle. For example, given *file* and *write*, the function returns *file.write*. Do the same exercise with C-style strings using only C facilities such as *malloc*() and *strlen*(). Compare the two functions. What are reasonable criteria for a comparison?
2. (∗2) Make a list of differences between *vector* and *basic_string*. Which differences are important?
3. (∗2) The string facilities are not perfectly regular. For example, you can assign a *char* to a string, but you cannot initialize a *string* with a *char*. Make a list of such irregularities. Which could have been eliminated without complicating the use of strings? What other irregularities would this introduce?
4. (∗1.5) Class *basic_string* has a lot of members. Which could be made nonmember functions without loss of efficiency or notational convenience?
5. (∗1.5) Write a version of *back_inserter*() (§19.2.4) that works for *basic_string*.
6. (∗2) Complete *Basic_substring* from §20.3.13 and integrate it with a *String* type that overloads () to mean ''substring of'' and otherwise acts like *string*.
7. (∗2.5) Write a *find*() function that finds the first match for a simple regular expression in a *string*. Use ? to mean ''any character,'' * to mean any number of characters not matching the next part of the regular expression, and [*abc*] to mean any character from the set specified between the square braces (here *a*, *b*, and *c*). Other characters match themselves. For example, *find*(*s*,"*name*:") returns a pointer to the first occurrence of *name*: in *s*; *find*(*s*,"[*nN*]*ame*:") returns a pointer to the first occurrence of *name*: or *Name*: in *s*; and *find*(*s*,"[*nN*]*ame*(*)") returns a pointer to the first occurence of *Name* or *name* followed by a (possibly empty) parenthesized sequences of characters in *s*.
8. (∗2.5) What operations do you find missing from the simple regular expression function from

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

§20.6[7]? Specify and add them. Compare the expressiveness of your regular expression matcher to that of a widely distributed one. Compare the performance of your regular expression matcher to that of a widely distributed one.

9. (∗2.5) Use a regular expression library to implement pattern-matching operations on a ***String*** class that has an associated ***Substring*** class.
10. (∗2.5) Consider writing an ''ideal'' class for general text processing. Call it ***Text***. What facilities should it have? What implementation constraints and overheads are imposed by your set of ''ideal'' facilities?
11. (∗1.5) Define a set of overloaded versions for ***isalpha***( ), ***isdigit***( ), etc., so that these functions work correctly for ***char***, ***unsigned char***, and ***signed char***.
12. (∗2.5) Write a ***String*** class optimized for strings having no more than eight characters. Compare its performance to that of the ***String*** from §11.12 and your implementation's version of the standard library ***string***. Is it possible to design a string that combines the advantages of a string optimized for very short strings with the advantages of a perfectly general string?
13. (∗2) Measure the performance of copying of ***string***s. Does your implementation's implementation of ***string*** adequately optimize copying?
14. (∗2.5) Compare the performance of the three ***complete_name***( ) functions from §20.3.9 and §20.3.10. Try to write a version of ***complete_name***( ) that runs as fast as possible. Keep a record of mistakes found during its implementation and testing.
15. (∗2.5) Imagine that reading medium-long strings (most are 5 to 25 characters long) from ***cin*** is the bottleneck in your system. Write an input function that reads such strings as fast as you can think of. You can choose the interface to that function to optimize for speed rather than for convenience. Compare the result to your implementation's >> for ***string***s.
16. (∗1.5) Write a function ***itos***(***int***) that returns a ***string*** representing its ***int*** argument.

.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.