Listing 5

1. A C++ String Class. To demonstrate the use of `CWEB` for C++ programming, we adapt the **string** class described by Stroustrup [1, pages 248–251]. Explanations in *slanted type* (including inline comments, when possible) are direct quotes from the original. We make a few minor changes along the way, but on the whole, we stick to Stroustrup's design.

2. We put the interface part of our class in the header file `xstring.h`. We call our class "**Xstring**" rather than "**string**" to avoid confusion with the original and other (more useful) string classes. We restrict ourselves to a lowercase file name to maintain portability among operating systems with case-insensitive file names.

```
⟨ xstring.h 2 ⟩ ≡
#ifndef XSTRING_H
#define XSTRING_H      // prevent multiple inclusions
  class Xstring {
    ⟨ Private Xstring members 4 ⟩
  public:
    ⟨ Public Xstring members 5 ⟩
  };
#endif
```

This code is cited in section 3.

This code is used in section 3.

3. We implement the class members in a single "unnamed chunk" that will be tangled to `xstring.c` (or `xstring.cc` or `xstring.cpp`, depending on your compiler's preference). We include the contents of ⟨ `xstring.h` 2 ⟩ directly, rather than relying on **#include**, because we can.

```
  ⟨ Header files 8 ⟩
  ⟨ xstring.h 2 ⟩
  ⟨ Xstring members and friends 6 ⟩
```

Listing 5 (continued)

4. Representing an Xstring. The internal representation of an **Xstring** is simple. *It counts the references to a string to minimize copying and uses standard C++ character strings as constants.*

⟨ Private **Xstring** members 4 ⟩ ≡

```
struct srep {
    char *s;    // pointer to data
    int n;      // reference count
    srep() { n ← 1; }
};
srep *p;
```

See also section 16.

This code is used in section 2.

---

5. Construction and Destruction. *The constructors and the destructor are trivial.* We use the null string as a default constructor argument rather than a null pointer to protect against possible `string.h` function anomalies.

⟨ Public **Xstring** members 5 ⟩ ≡

```
Xstring(const char *s ← "");      // Xstring x ← "abc"
Xstring(const Xstring &);        // Xstring x ← Xstring ...
~Xstring();
```

See also sections 12, 14, and 15.

This code is used in section 2.

6. An **Xstring** constructed from a standard string needs space to hold the characters:

⟨ **Xstring** members and friends 6 ⟩ ≡

```
Xstring::Xstring(const char *s)
{
    p ← new srep;
    ⟨ Allocate space for the string and put a copy of s there 7 ⟩;
}
```

See also sections 9, 10, 13, and 17.

This code is used in section 3.

7. There is always the possibility that a client will try something like "**Xstring** $x \leftarrow \Lambda$." We substitute the null string whenever we are given a null pointer.

⟨ Allocate space for the string and put a copy of s there 7 ⟩ ≡

```
if (s ≡ Λ) s ← "";
p→s ← new char [strlen(s) + 1];
strcpy(p→s, s);
```

This code is used in sections 6 and 13.

8. ⟨ Header files 8 ⟩ ≡

```
#include <string.h>     // Standard C header for strcpy
```

This code is used in section 3.

9. On the other hand, to build an **Xstring** from another **Xstring**, we only have to increment the reference count:

Listing 5 (continued)

⟨ **Xstring** members and friends 6 ⟩ +≡

```
Xstring::Xstring(const Xstring &x)
{
  x.p→n++;
  p ← x.p;
}
```

10. The destructor also has to worry about the reference count:

⟨ **Xstring** members and friends 6 ⟩ +≡

```
Xstring::~Xstring()
{
  ⟨ Decrement reference count, and remove p if necessary 11 ⟩;
}
```

11. ⟨ Decrement reference count, and remove p if necessary 11 ⟩ ≡

```
if (−−p→n ≡ 0) {
  delete []p→s;
  delete p;
}
```

This code is used in sections 10 and 13.

---

12. Assignment. *As usual, the assignment operators are similar to the constructors. They must handle cleanup of their first (left-hand) operand:*

⟨ Public **Xstring** members 5 ⟩ +≡

```
Xstring &operator←(const char *);
Xstring &operator←(const Xstring &);
```

13. ⟨ **Xstring** members and friends 6 ⟩ +≡

```
Xstring &Xstring::operator←(const char *s)
{
  if (p→n > 1) {      // disconnect self
    p→n−−;
    p ← new srep;
  } else      // free old string
    delete []p→s;
  ⟨ Allocate space for the string and put a copy of s there 7 ⟩;
  return *this;
}
Xstring &Xstring::operator←(const Xstring &x)
{
  x.p→n++;      // protect against "st ← st"
  ⟨ Decrement reference count, and remove p if necessary 11 ⟩;
  p ← x.p;
  return *this;
}
```

---

14. Miscellaneous Operations. We provide a conversion operator to translate **Xstring**'s into ordinary strings. This allows us to pass them to standard functions like *strlen* (and gives us an output operator for free). We convert to **const** strings to prevent strange things from happening if a client should try to use a standard function like *strcat* to modify an **Xstring**.

⟨ Public **Xstring** members 5 ⟩ +≡
  operator const char $*()$ { **return** $p\rightarrow s$; }

15. *The subscript operator is provided for access to individual characters. The index is checked.* However, we depart from the original design by returning a dummy element when the index is out of bounds rather than generating an error message (or an exception).

⟨ Public **Xstring** members 5 ⟩ +≡
  char &**operator** [ ](**int** i) {
    return $((i < 0) \lor (strlen(p\rightarrow s) < i)\ ?\ dummy : p\rightarrow s[i])$; }

16. ⟨ Private **Xstring** members 4 ⟩ +≡
  static char *dummy*;

17. ⟨ **Xstring** members and friends 6 ⟩ +≡
  char Xstring :: *dummy*;

---

18. References.

[1] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

---

19. Index.