12.5. Bitsets

Bitsets model fixed-sized arrays of bits or Boolean values. They are useful to manage sets of flags, where variables may represent any combination of flags. C and old C++ programs usually use type `long` for arrays of bits and manipulate them with the bit operators, such as `&`, `|`, and `~`. The class `bitset` has the advantage that bitsets may contain any number of bits, and additional operations are provided. For example, you can assign single bits and can read and write bitsets as a sequence of `0`s and `1`s.

Note that you can't change the number of bits in a bitset. The number of bits is the template parameter. If you need a container for a variable number of bits or Boolean values, you can use the class `vector<bool>` (described in Section 7.3.6, page 281).

The class `bitset` is defined in the header file `<bitset>`:

```
#include <bitset>
```

In `<bitset>`, the class `bitset` is defined as a class template, with the number of bits as the template parameter:

```
namespace std {
    template <size_t Bits>
    class bitset;
}
```

In this case, the template parameter is not a type but an unsigned integral value (see Section 3.2, page 33, for details about this language feature).

Templates with different template arguments are different types. You can compare and combine bitsets only with the same number of bits.

Recent Changes with C++11

C++98 specified almost all features of bitsets. Here is a list of the most important features added with C++11:

- Bitsets now can be initialized by string literals (see Section 12.5.1, page 653).
- Conversions to and from numeric values now support type `unsigned long long`. For this, `to_ullong()` was introduced (see Section 12.5.1, page 653).
- Conversions to and from strings now allow you to specify the character interpreted as set and unset bit.
- Member `all()` is now provided to check whether all bits are set.
- To use bitsets in unordered containers, a default hash function is provided (see Section 7.9.2, page 363).

12.5.1. Examples of Using Bitsets

Using Bitsets as Sets of Flags

The first example demonstrates how to use bitsets to manage a set of flags. Each flag has a value that is defined by an enumeration type. The value of the enumeration type is used as the position of the bit in the bitset. In particular, the bits represent colors. Thus, each enumeration value defines one color. By using a bitset, you can manage variables that might contain any combination of colors:

Click here to view code image

```
// contadapt/bitset1.cpp

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

int main()
{
    // enumeration type for the bits
    // - each bit represents a color
    enum Color { red, yellow, green, blue, white, black, …,
                 numColors };

    // create bitset for all bits/colors
    bitset<numColors> usedColors;

    // set bits for two colors
    usedColors.set(red);
    usedColors.set(blue);

    // print some bitset data
```

```
        cout << "bitfield of used colors:   " << usedColors << endl;
        cout << "number   of used colors:   " << usedColors.count() << endl;
        cout << "bitfield of unused colors: " << ~usedColors << endl;

        // if any color is used
        if (usedColors.any()) {
            // loop over all colors
            for (int c = 0; c < numColors; ++c) {
                // if the actual color is used
                if (usedColors[(Color)c]) {
                    ...
                }
            }
        }
    }
```

Using Bitsets for I/O with Binary Representation

A useful feature of bitsets is the ability to convert integral values into a sequence of bits, and vice versa. This is done simply by creating a temporary bitset:

Click here to view code image

```
   // contadapt/bitset2.cpp

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

   int main()
   {
       // print some numbers in binary representation
       cout << "267 as binary short:     "
            << bitset<numeric_limits<unsigned short>::digits>(267)
            << endl;

       cout << "267 as binary long:      "
            << bitset<numeric_limits<unsigned long>::digits>(267)
            << endl;

       cout << "10,000,000 with 24 bits: "
            << bitset<24>(1e7) << endl;

       // write binary representation into string
       string s = bitset<42>(12345678).to_string();
       cout << "12,345,678 with 42 bits: " << s << endl;

       // transform binary representation into integral number
       cout << "\"1000101011\" as number:   "
            << bitset<100>("1000101011").to_ullong() << endl;
   }
```

Depending on the number of bits for `short` and `long long`, the program might produce the following output:

Click here to view code image

```
   267 as binary short:     0000000100001011
   267 as binary long:      00000000000000000000000100001011
   10,000,000 with 24 bits: 100110001001011010000000
   12,345,678 with 42 bits: 000000000000000000101111000110000101001110
   "1000101011" as number:  555
```

In this example, the following expression converts `267` into a bitset with the number of bits of type `unsigned short` (see Section 5.3, page 116, for a discussion of numeric limits):

```
   bitset<numeric_limits<unsigned short>::digits>(267)
```

The output operator for `bitset` prints the bits as a sequence of characters `0` and `1`.

You can output bitsets directly or use their value as a string:

```
   string s = bitset<42>(12345678).to_string();
```

Note that before C++11, you had to write

```
   string s = bitset<42>(12345678).to_string<char,char_traits<char>,
                                                allocator<char> >();
```

here because `to_string()` is a member template, and there were no default values for the template arguments defined.

Similarly, the following expression converts a sequence of binary characters into a bitset, for which `to_ullong()` yields the integral value:

```
bitset<100>("1000101011")
```

Note that the number of bits in the bitset should be smaller than `sizeof(unsigned long long)`. The reason is that you get an exception when the value of the bitset can't be represented as `unsigned long long`.[1]

[1] Before C++11, type `unsigned long long` was not provided, so you could call only `to_ulong()` here. `to_ulong()` is still callable and works fine if the number of bits is smaller than `sizeof(unsigned long)`.

Note also that before C++11, you had to convert the initial value to type `string` explicitly:

```
bitset<100>(string("1000101011"))
```

12.5.2. Class `bitset` in Detail

Due to the thickness of this book, the subsection that presents the members of class `bitset<>` in detail is provided as a supplementary chapter on the Web site of this book at http://www.cppstdlib.com.