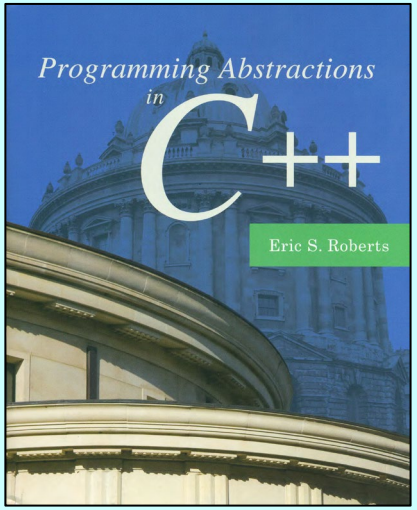
CHAPTER 17

Sets

We are an ambitious set, aren't we?

—Louisa May Alcott, *Little Women,* 1868



Methods in the `Set<type>` Class

Flashback

`set.size()` Returns the number of elements in the set.
`set.isEmpty()` Returns **`true`** if the set is empty.
`set.add(value)` Adds **`value`** to the set.
`set.remove(value)` Removes **`value`** from the set.
`set.contains(value)` Returns **`true`** if the set contains the specified value.
`set.clear()` Removes all words from the set.
`s1.isSubsetOf(s2)` Returns **`true`** if **`s1`** is a subset of **`s2`**.
`set.first()` Returns the first element of the set in the ordering specified by the value type.

The Easy Implementation

- As is so often the case, the easy way to implement the **`Set`** class is to build it out of data structures that you already have. In this case, it make sense to build **`Set`** on top of the **`Map`** class.
- The private section looks like this:

```
private:

/* Instance variables */

   Map<ValueType, bool> map;          /* The bool is unused */
```

High-Level Set Operations

- Up to this point in both the text and the assignments, the examples that used the **`Set`** class have focused on the low-level operations of adding, removing, and testing the presence of an element.
- Particularly when we introduce the various fundamental graph algorithms next week, it will be important to consider the high-level operations of *union*, *intersection*, *difference*, *subset*, and *equality* as well.
- The primary goal of today's lecture is to write the code that implements those operations.

The `Set<type>` Class

- This class is used to model the mathematical abstraction of a ***set***, which is a collection in which the elements are unordered and in which each value appears only once.

Operators

s_1 **+** s_2	Returns the ***union*** of s_1 and s_2, which consists of the elements in either or both of the original sets.
s_1 ***** s_2	Returns the ***intersection*** of s_1 and s_2, which consists of the elements common to both of the original sets.
s_1 **−** s_2	Returns the ***set difference*** of s_1 and s_2, which consists of the all elements in s_1 that are not present in s_2.
s_1 **+=** s_2 s_1 **−=** s_2 s_1 ***=** s_2	The **+**, **−**, and ***** operators can be combined with assignment just as they can with numeric values. For **+=** and **−=**, the right hand value can be a set, a single value, or a list of values separated by commas.

High-Level Operators in `set.h`

```
/*
 * Method: isSubsetOf
 * Usage: if (set.isSubsetOf(set2)) . . .
 * --------------------------------------
 * Implements the subset relation for sets.  This method returns true
 * if every element of this set is contained in set2.
 */

   bool isSubsetOf(const Set & set2) const;

/*
 * Operator: ==
 * Usage: set1 == set2
 * -------------------
 * Returns true if set1 and set2 contain the same elements.
 */

   bool operator==(const Set & set2) const;

/*
 * Operator: !=
 * Usage: set1 != set2
 * -------------------
 * Returns true if set1 and set2 are different.
 */

   bool operator!=(const Set & set2) const;
```

High-Level Operators in `set.h`

```
/*
 * Operator: +
 * Usage: set1 + set2
 *        set1 + element
 * --------------------
 * Returns the union of sets set1 and set2, which is the set of elements
 * that appear in at least one of the two sets.  The right hand set can be
 * replaced by an element of the value type, in which case the operator
 * returns a new set formed by adding that element.
 */

   Set operator+(const Set & set2) const;
   Set operator+(const ValueType & value) const;

/*
 * Operator: +=
 * Usage: set1 += set2;
 *        set1 += value;
 * --------------------
 * Adds all elements from set2 (or the single specified value) to set1.
 */

   Set & operator+=(const Set & set2);
   Set & operator+=(const ValueType & value);
```

High-Level Operators in `set.h`

```
/*
 * Operator: *
 * Usage: set1 * set2
 * ------------------
 * Returns the intersection of sets set1 and set2, which is the set of all
 * elements that appear in both.
 */

   Set operator*(const Set & set2) const;

/*
 * Operator: *=
 * Usage: set1 *= set2;
 * --------------------
 * Removes any elements from set1 that are not present in set2.
 */

   Set & operator*=(const Set & set2);
```

High-Level Operators in `set.h`

```
/*
 * Operator: -
 * Usage: set1 - set2
 *        set1 - element
 * ---------------------
 * Returns the difference of sets set1 and set2, which is all of the
 * elements that appear in set1 but not set2.  The right hand set can be
 * replaced by an element of the value type, in which case the operator
 * returns a new set formed by removing that element.
 */

   Set operator-(const Set & set2) const;
   Set operator-(const ValueType & value) const;

/*
 * Operator: -=
 * Usage: set1 -= set2;
 *        set1 -= value;
 * ---------------------
 * Removes all elements from set2 (or a single value) from set1.
 */

   Set & operator-=(const Set & set2);
   Set & operator-=(const ValueType & value);
```

The Implementation Section

```
template <typename ValueType>
Set<ValueType>::Set() {
   /* Empty */
}

template <typename ValueType>
Set<ValueType>::~Set() {
   /* Empty */
}

template <typename ValueType>
int Set<ValueType>::size() const {
   return map.size();
}

template <typename ValueType>
bool Set<ValueType>::isEmpty() const {
   return map.isEmpty();
}

template <typename ValueType>
void Set<ValueType>::add(const ValueType & value) {
   map.put(value, true);
}
```

The Implementation Section

```
template <typename ValueType>
void Set<ValueType>::remove(const ValueType & value) {
   map.remove(value);
}

template <typename ValueType>
bool Set<ValueType>::contains(const ValueType & value) const {
   return map.containsKey(value);
}

template <typename ValueType>
void Set<ValueType>::clear() {
   map.clear();
}

/*
 * Method: first
 * Usage: ValueType value = set.first();
 * -----------------------------
 * Returns the first element of the.
 */
template <typename ValueType>
ValueType Set<ValueType>::first() {
/* Iterator needed */
}
```

The Implementation Section

```
/*
 * Implementation notes: isSubsetOf
 * --------------------------------
 * The implementation of the high-level functions does not require knowledge
 * of the underlying representation
 */

template <typename ValueType>
bool Set<ValueType>::isSubsetOf(const Set & set2) const {
   for (ValueType value : map) {
      if (!set2.contains(value)) return false;
   }
   return true;
}

/*
 * Implementation notes: ==
 * ------------------------
 * Two sets are equal if they are subsets of each other.
 */

template <typename ValueType>
bool Set<ValueType>::operator==(const Set & set2) const {
   return isSubsetOf(set2) && set2.isSubsetOf(*this);
}
```

Exercise: Implementing Set Methods

```
/*
 * Operator: +
 * Usage: set1 + set2
 *        set1 + value
 * --------------------
 * Returns the union of sets set1 and set2, which is the set of elements
 * that appear in at least one of the two sets.  The right hand set can be
 * replaced by an element of the value type, in which case the operator
 * returns a new set formed by adding that element.
 */

template <typename ValueType>
Set<ValueType> Set<ValueType>::operator+(const Set & set2) const {
   Set<ValueType> set = *this;
   for (ValueType value : set2.map) {
      set.add(value);
   }
   return set;
}

template <typename ValueType>
Set<ValueType> Set<ValueType>::operator+(const ValueType & value) const {
   Set<ValueType> set = *this;
   set.add(value);
   return set;
}
```

Implementing Sets

- Modern library systems adopt either of two strategies for implementing sets:

  1. ***Hash tables***. Sets implemented as hash tables are extremely efficient, offering average $O(1)$ performance for adding a new element or testing for membership. The primary disadvantage is that hash tables do not support ordered iteration.

  2. ***Balanced binary trees***. Sets implemented using balanced binary trees offer $O(\log N)$ performance on the fundamental operations, but do make it possible to write an ordered iterator.

- The **`set`** class in standard C++ uses the latter approach. The **`unordered_set`** class (from C++11) uses the former.
- One of the implications of using the BST representation is that the underlying value type must support the comparison operators **`==`** and **`<`**. In Chapter 20, you'll learn how to relax that restriction by specifying a comparison function as an argument to the **`Set`** constructor.

Sets and Efficiency

- After you release the **`Set`** package, you might discover that clients use them often for particular types for which there are much more efficient data structures than binary trees.
- One thing you could do easily is check to see whether the element type was **`string`** and then use a **`Lexicon`** (DAWG) instead of a binary search tree. The resulting implementation would be far more efficient. This change, however, would be valuable only if clients used **`Set<string>`** often enough to make it worth adding the complexity.
- One type of sets that do tend to occur in certain types of programming is **`Set<char>`**, which comes up, for example, if you want to specify a set of delimiter characters for a scanner. These sets can be made astonishingly efficient as described on the next few slides.

Character Sets

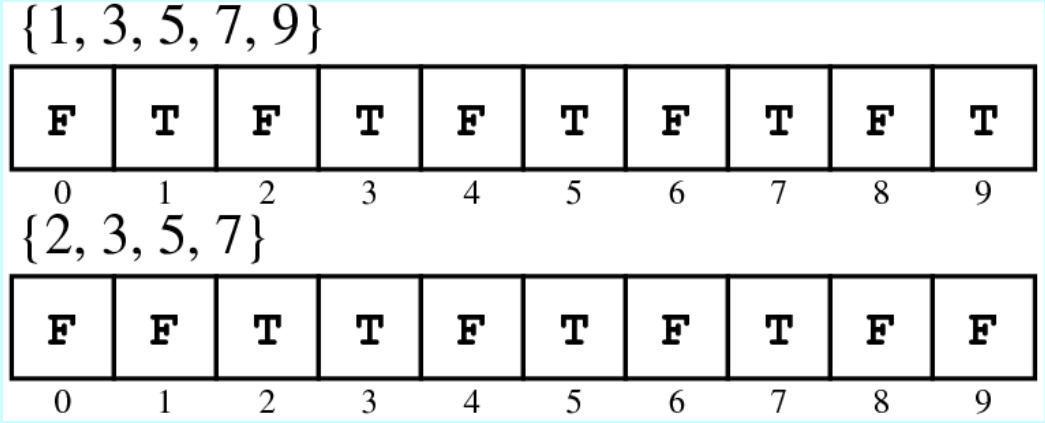
- The key insight needed to make efficient ***character sets*** (or, equivalently, ***sets of small integers***) is that you can represent the inclusion or exclusion of a character using a single bit. If the bit is a 1, then that element is in the set; if it is a 0, it is not in the set.
- You can tell what character value you're talking about by creating what is essentially an array of bits, with one bit for each of the ASCII codes. That array is called a ***characteristic vector/indicator vector***.
- What makes this representation so efficient is that you can pack the bits for a characteristic vector into a small number of words inside the machine and then operate on the bits in large chunks.
- The efficiency gain is enormous. Using this strategy, most set operations can be implemented in just a few instructions.

Example: Characteristic Vectors

- Any subsets of the following set:

  digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

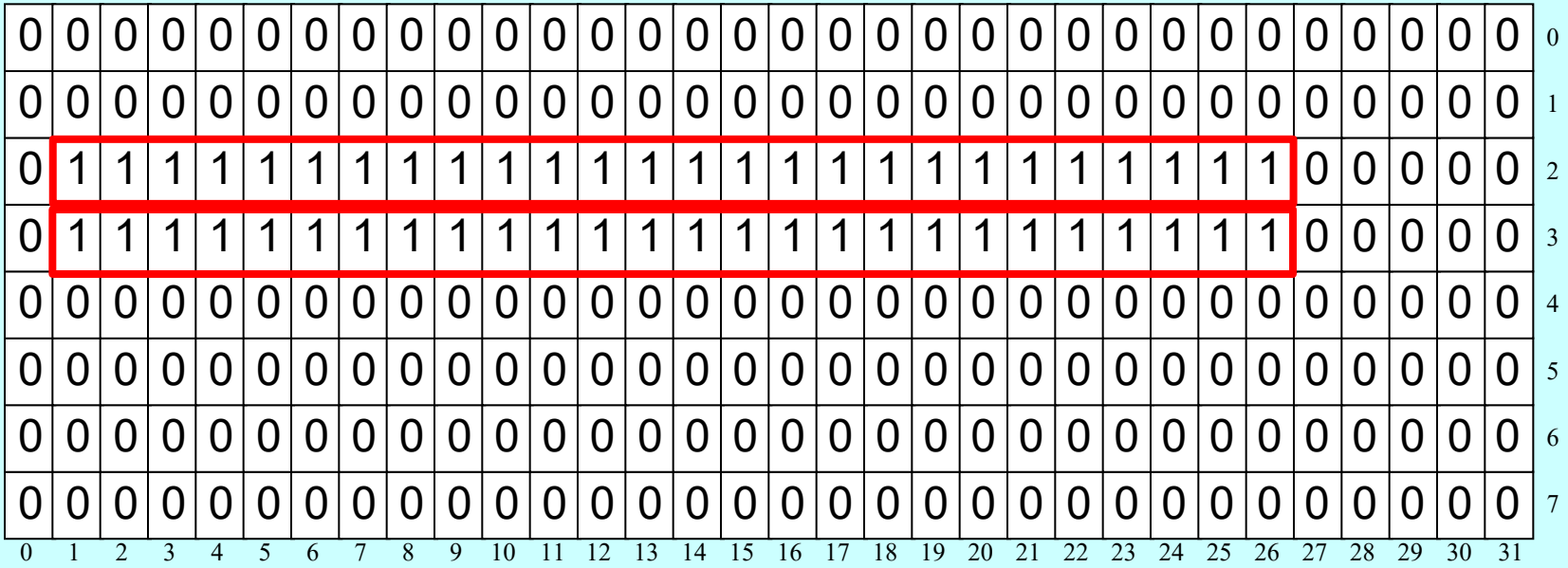
  can be indicated by a characteristic vector of 10 bits:



- The advantage of using characteristic vectors is that doing so makes it possible to implement the operations **`add`**, **`remove`**, and **`contains`** in any subsets in constant time. For example, to add the element *k* to a set, all you have to do is set the element at index position *k* in the characteristic vector to true. Similarly, testing membership is simply a matter of selecting the appropriate element in the array.

Bit Vectors and Character Sets

- This picture shows a characteristic vector representation for the set containing the uppercase and lowercase letters:



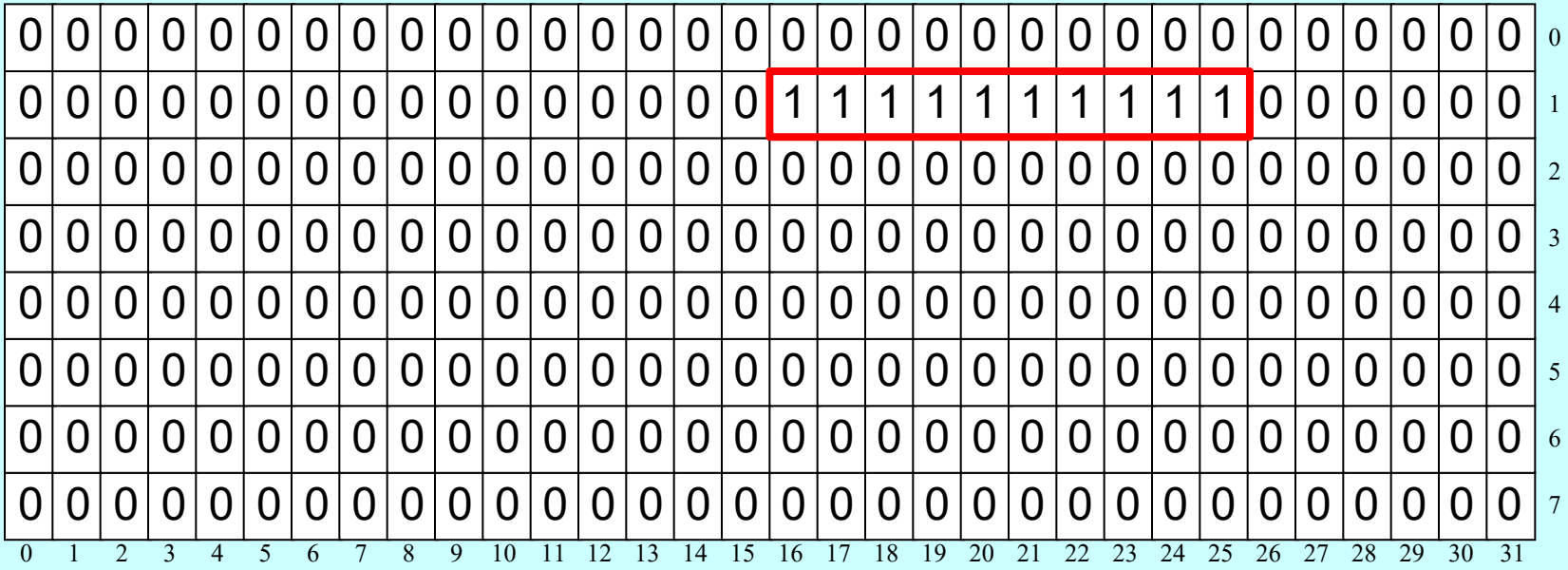
- Storing characteristic vectors as explicit arrays can require a large amount of memory, particularly if the set is large. To reduce the storage requirements, you can pack the elements of the characteristic vector into machine words so that the representation uses every bit in the underlying representation.

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

Bit Vectors and Character Sets

- Exercise: What is a characteristic vector representation for the set containing the digit letters:



Bitwise Operators

- If you know your client is working with sets of characters, you can implement the set operators extremely efficiently by storing the set as an array of bits and then manipulating the bits all at once using C++'s ***bitwise operators***.
- The bitwise operators are summarized in the following table and then described in more detail on the next few slides:

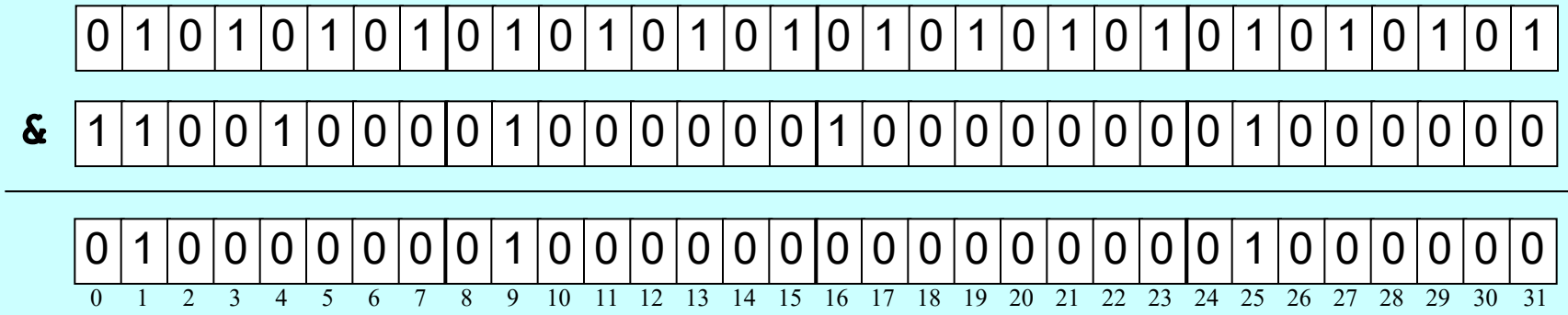
x **&** *y*	Bitwise AND. The result has a 1 bit wherever both *x* and *y* have 1s.
x **|** *y*	Bitwise OR. The result has a 1 bit wherever either *x* or *y* have 1s.
x **^** *y*	Exclusive OR. The result has a 1 bit wherever *x* and *y* differ.
~*x*	Bitwise NOT. The result has a 1 bit wherever *x* has a 0.
x **<<** *n*	Left shift. Shift the bits in *x* left *n* positions, shifting in 0s.
x **>>** *n*	Right shift. Shift *x* right *n* bits (logical shift if *x* is unsigned).

The Bitwise AND Operator

- The bitwise AND operator (`&`) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position in which both *x* and *y* have 1 bits. A table for the `&` operator appears to the right.

	0	1
0	0	0
1	0	1

- The primary application of the `&` operator is to ***select*** certain bits in an integer, clearing the unwanted bits to 0. This operation is called ***masking***.
- In the context of sets, the `&` operator performs an intersection operation, as in the following calculation of **odds** ∩ **squares**:

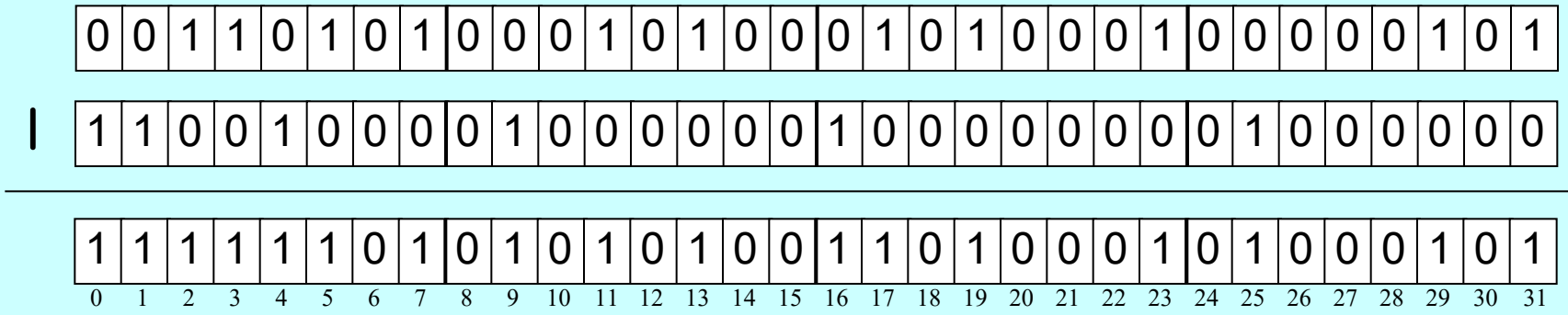


The Bitwise OR Operator

- The bitwise OR operator (`|`) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position which either *x* or *y* has a 1 bit (or if both do, i.e., non-exclusive), as shown in the table on the right.

	0	1
0	0	1
1	1	1

- The primary use of the `|` operator is to ***assemble*** a single integer value from other values, each of which contains a subset of the desired bits.
- In the context of sets, the `|` operator performs a union, as in the following calculation of **primes** ∪ **squares**:



The Exclusive OR Operator

or but not both

- The exclusive OR or XOR operator (^) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position in which *x* and *y* have different bit values, as shown on the right.

	0	1
0	0	1
1	1	0

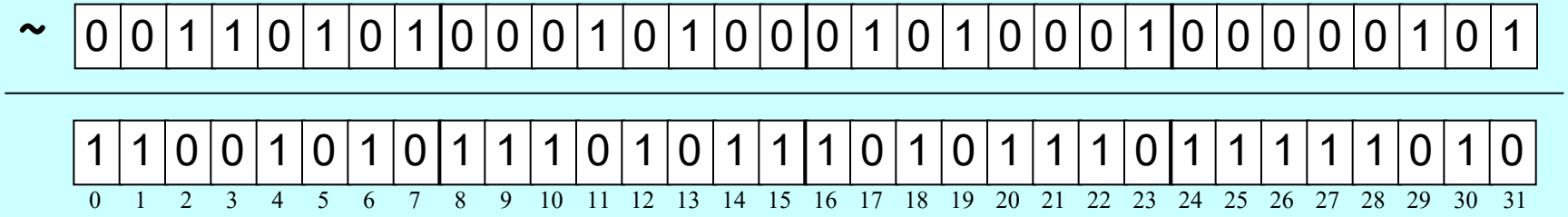
- The XOR operator has many applications in programming, most of which are beyond the scope of this text.
- The following example ***flips*** all the bits in the rightmost three bytes of a word:

	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1
^	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

- Question: What if we apply the same XOR mask twice?
  See "XOR drawing mode" in computer graphics if interested.

The Bitwise NOT Operator

- The bitwise NOT operator (**~**) takes a single operand *x* and returns a value that has a 1 wherever *x* has a 0, and vice versa.
- You can use the bitwise NOT operator to create a mask in which you mark the bits you want to eliminate as opposed to the ones you want to preserve.
- The **~** operator creates the ***complement*** of a set, as shown with the following diagram of **~primes**:



- Question: Can you use the **~** operator to compute the set difference operation?

The Shift Operators

- C++ defines two operators that have the effect of shifting the bits in a word by a given number of bit positions.
- The expression *x* **<<** *n* shifts the bits in the integer *x* leftward *n* positions. Spaces appearing on the right are filled with 0s.
- The expression *x* **>>** *n* shifts the bits in the integer *x* rightward *n* positions. The question as to what bits are shifted in on the left depend on whether *x* is a signed or unsigned type:
  - If *x* is an unsigned type, the **>>** operator performs a ***logical shift*** in which missing digits are always filled with 0s.
  - If *x* is a signed type, the **>>** operator performs what computer scientists call an ***arithmetic shift*** in which the leading bit in the value of *x* never changes. Thus, if the first bit is a 1, the **>>** operator fills in 1s; if it is a 0, those spaces are filled with 0s.
- Arithmetic shifts are efficient ways to perform multiplication or division of signed integers by powers of two.

Two's Complement

- Two's complement is a mathematical operation on binary numbers, and a method of signed number representation.
- The two's complement of an N-bit number is defined as its complement with respect to 2^N, i.e., the sum of a number and its two's complement is 2^N.
- Conveniently, another way of finding the two's complement is inverting the digits and adding one.
- 3-bit and 8-bit signed integers:

0	000
1	001
2	010
3	011
−4	100
−3	101
−2	110
−1	111

0	0000 0000
1	0000 0001
2	0000 0010
126	0111 1110
127	0111 1111
−128	1000 0000
−127	1000 0001
−126	1000 0010
−2	1111 1110
−1	1111 1111

Exercise: Bitwise Operators

- What are the outputs:

```
int a = 5;
int b = 10;
cout << (a&&b) << ' ' << (a&b) << endl;
```

```
1 0
```

```
cout << hex << -1 << ' '
<< (-1 << 1) << ' '
<< (-1 >> 1) << ' '
<< unsigned(-1) << ' '
<< (unsigned(-1) << 1) << ' '
<< (unsigned(-1) >> 1) << endl;
```

Largest unsigned integer in 32-bit machines, 2^{32}-1.

Largest positive integer in 32-bit machines, 2^{31}-1.

```
ffffffff fffffffe ffffffff ffffffff ffffffe 7fffffff
```

```
-1 -2 -1 4294967295 4294967294 2147483647
```

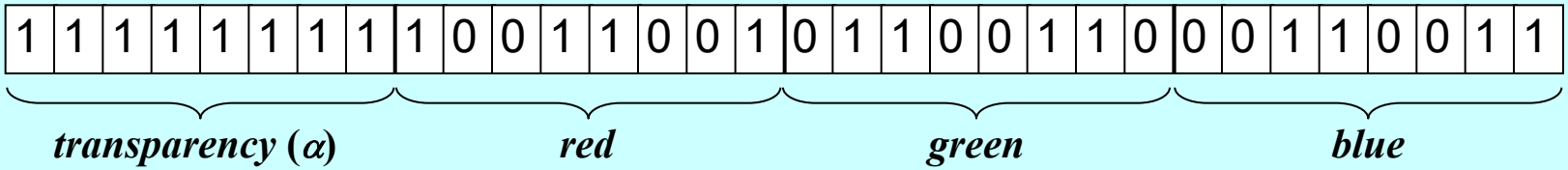
Example: Function `hashCode` for Strings

```
const int HASH_SEED = 5381; /* Starting point for first cycle */
const int HASH_MULTIPLIER = 33; /* Multiplier for each cycle */
const int HASH_MASK = unsigned(-1) >> 1; /* Largest positive integer */
/*                          FFFFFFFF   7FFFFFFF                      */
/*
 * Function: hashCode
 * Usage: int code = hashCode(key);
 * --------------------------------
 * This function takes a string key and uses it to derive a hash code,
 * which is nonnegative integer related to the key by a deterministic
 * function that distributes keys well across the space of integers.
 * The specific algorithm used here is called djb2 after the initials
 * of its inventor, Daniel J. Bernstein, Professor of Mathematics at
 * the University of Illinois at Chicago.
 */

int hashCode(const string & str) {
   unsigned hash = HASH_SEED;
   int nchars = str.length();
   for (int i = 0; i < nchars; i++) {
      hash = HASH_MULTIPLIER * hash + str[i];
   }
   return (hash & HASH_MASK);
}
```

Exercise: Manipulating Pixels

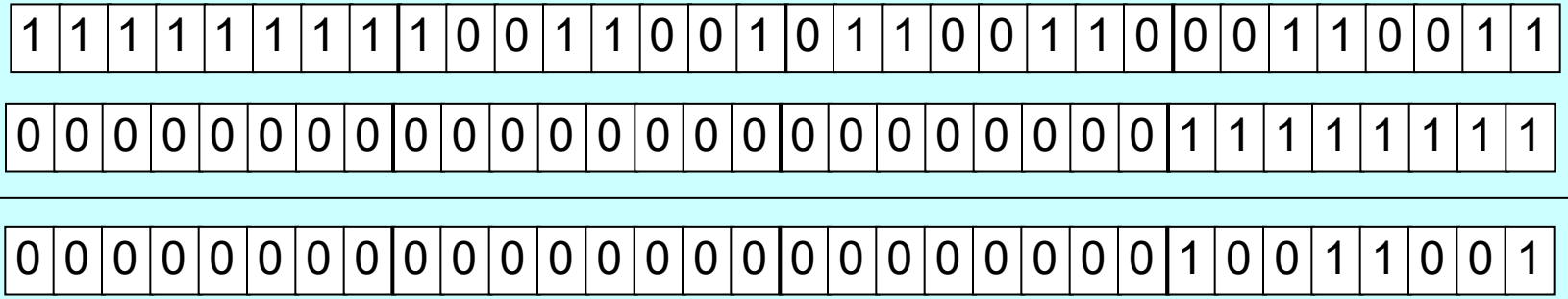
- Computers typically represent each pixel in an image as an **`int`** in which the 32 bits are interpreted as follows:



  This color, for example, would be represented in hexadecimal as **`0xFF996633`**, which indicates the color **brown**.

- Write the code to isolate the red component of a color stored in the integer variable **`pixel`**.

```
int red = (pixel >> 16) & 0xFF;
```



Implementing Characteristic Vectors

```
const int BITS_PER_BYTE = 8;
const int BITS_PER_LONG = BITS_PER_BYTE * sizeof(long);
const int CVEC_WORDS = (RANGE_SIZE + BITS_PER_LONG - 1) / BITS_PER_LONG;
struct CharacteristicVector {
   unsigned long words[CVEC_WORDS];
};
unsigned long createMask(int k) {
   return unsigned long(1) << k % BITS_PER_LONG;
}
bool testBit(CharacteristicVector & cv, int k) {
    if (k < 0 || k >= RANGE_SIZE) {
       error("testBit: Bit index is out of range");
    }
    return cv.words[k / BITS_PER_LONG] & createMask(k);
}
void setBit(CharacteristicVector & cv, int k) {
   if (k < 0 || k >= RANGE_SIZE) {
      error("setBit: Bit index is out of range");
   }
   cv.words[k / BITS_PER_LONG] |= createMask(k);
}
void clearBit(CharacteristicVector & cv, int k) {
   if (k < 0 || k >= RANGE_SIZE) {
      error("setBit: Bit index is out of range");
   }
   cv.words[k / BITS_PER_LONG] &= ~createMask(k);
}
```

How many words do we need for the characteristic vector?

*column index - note here the bits are stored **from right to left** to simplify the calculation.*

row index

Abstract Data Type (ADT)

- The atomic/primitive data types like **`bool`**, **`char`**, **`int`**, **`double`**, and the enumerated types (**`enum`**) occupy the lowest level in the data structure hierarchy.
- To represent more complex information, you combine the atomic types to form larger structures.
- It is usually far more important to know how to use those structures effectively than to understand their underlying representation, e.g., using strings as abstract values (Ch. 3).
- A type defined in terms of its behavior rather than its representation is called an ***Abstract Data Type (ADT)***.
- ADTs are central to the object-oriented style of programming, which encourages programmers to think about data structures in a holistic way.

Abstract Data Type (ADT)

- An ***Abstract Data Type (ADT)*** is defined in terms of its behavior rather than its representation, and the subtle difference in how it is used may require different underlying representations.
- In turn, different underlying representations may result in subtly different ADTs.
- In Stanford C++ Library
  - **`Map`** vs. **`HashMap`**
  - **`Set`** vs. **`HashSet`** vs. **`Lexicon`**
- In Standard C++ Library
  - **`vector`** vs. **`list`**
  - **`map`** (**`set`**) vs. **`unordered_map`** (**`unordered_set`**)

Initial Versions Should Be Simple

Premature optimization is the root of all evil.
—Don Knuth

- When you are developing an implementation of a public interface, it is best to begin with the simplest possible code that satisfies the requirements of the interface.
- This approach has several advantages:
  - You can get the package out to clients much more quickly.
  - Simple implementations are much easier to get right.
  - You often won't have any idea what optimizations are needed until you have actual data from clients of that interface. In terms of overall efficiency, some optimizations are much more important than others.

Efficiency vs. Readability

Coding when alone

```
i *= 2;
i *= -1;
```

Coding when somebody is watching

```
i <<= 1;
i = ~i + 1;
```

The End