Sets

Eric Roberts
CS 106B
February 20, 2015

Outline

1. Sets in mathematics
2. Venn diagrams
3. High-level set operations
4. Implementing the **`Set`** class
5. Sets and efficiency
6. Bitwise operators

Sets in Mathematics

- A ***set*** is an unordered collection of distinct values.

  digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
  evens = { 0, 2, 4, 6, 8 }
  odds = { 1, 3, 5, 7, 9 }
  primes = { 2, 3, 5, 7 }
  squares = { 0, 1, 4, 9 }

  colors = { *red*, *yellow*, *green*, *cyan*, *blue*, *magenta* }
  primary = { *red*, *green*, *blue* }
  secondary = { *yellow*, *cyan*, *magenta* }

  R = { x | x is a real number }
  Z = { x | x is an integer }
  N = { x | x is an integer and $x \geq 0$ }

- The set with no elements is called the ***empty set*** ($\varnothing$).

Set Operations

- The fundamental set operation is ***membership*** ($\in$).

$3 \in$ **primes**	$3 \notin$ **evens**
red $\in$ **primary**	*red* $\notin$ **secondary**
$-1 \in$ **Z**	$-1 \notin$ **N**

- The ***union*** of two sets A and B ($A \cup B$) consists of all elements in either A or B or both.
- The ***intersection*** of A and B ($A \cap B$) consists of all elements in both A or B.
- The ***set difference*** of A and B ($A - B$) consists of all elements in A but not in B.
- Set A is a ***subset*** of B ($A \subseteq B$) if all elements in A are also in B.
- Sets A and B are ***equal*** ($A = B$) if they have the same elements.

Exercise: Set Operations

Suppose that you have the following sets:

digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
evens = { 0, 2, 4, 6, 8 }
odds = { 1, 3, 5, 7, 9 }
primes = { 2, 3, 5, 7 }
squares = { 0, 1, 4, 9 }

What is the value of each of the following expressions:

a) **evens** ∪ **squares** { 0, 1, 2, 4, 6, 8, 9 }

b) **odds** ∩ **squares** { 1, 9 }

c) **squares** ∩ **primes** ∅

d) **primes** – **evens** { 3, 5, 7 }

Given only these sets, can you produce the set { 1, 2, 9 }?

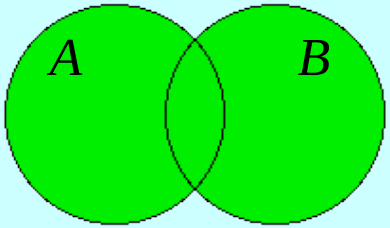
(**primes** ∩ **evens**) ∪ (**odds** ∩ **squares**)

Fundamental Set Identities

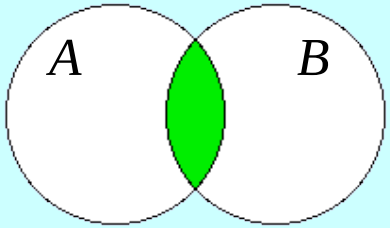
$S \cup S \equiv S$ $S \cap S \equiv S$	*Idempotence*
$A \cap (A \cup B) \equiv A$ $A \cup (A \cap B) \equiv A$	*Absorption*
$A \cup B \equiv B \cup A$ $A \cap B \equiv B \cap A$	*Commutative laws*
$A \cup (B \cup C) \equiv (A \cup B) \cup C$ $A \cap (B \cap C) \equiv (A \cap B) \cap C$	*Associative laws*
$A \cap (B \cup C) \equiv (A \cap B) \cup (A \cap C)$ $A \cup (B \cap C) \equiv (A \cup B) \cap (A \cup C)$	*Distributive laws*
$A - (B \cup C) \equiv (A - B) \cap (A - C)$ $A - (B \cap C) \equiv (A - B) \cup (A - C)$	*DeMorgan's laws*

Venn Diagrams

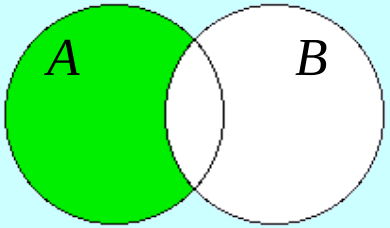
- A ***Venn diagram*** is a graphical representation of a set in that indicates common elements as overlapping areas.
- The following Venn diagrams illustrate the effect of the union, intersection, and set-difference operators:



$A \cup B$



$A \cap B$



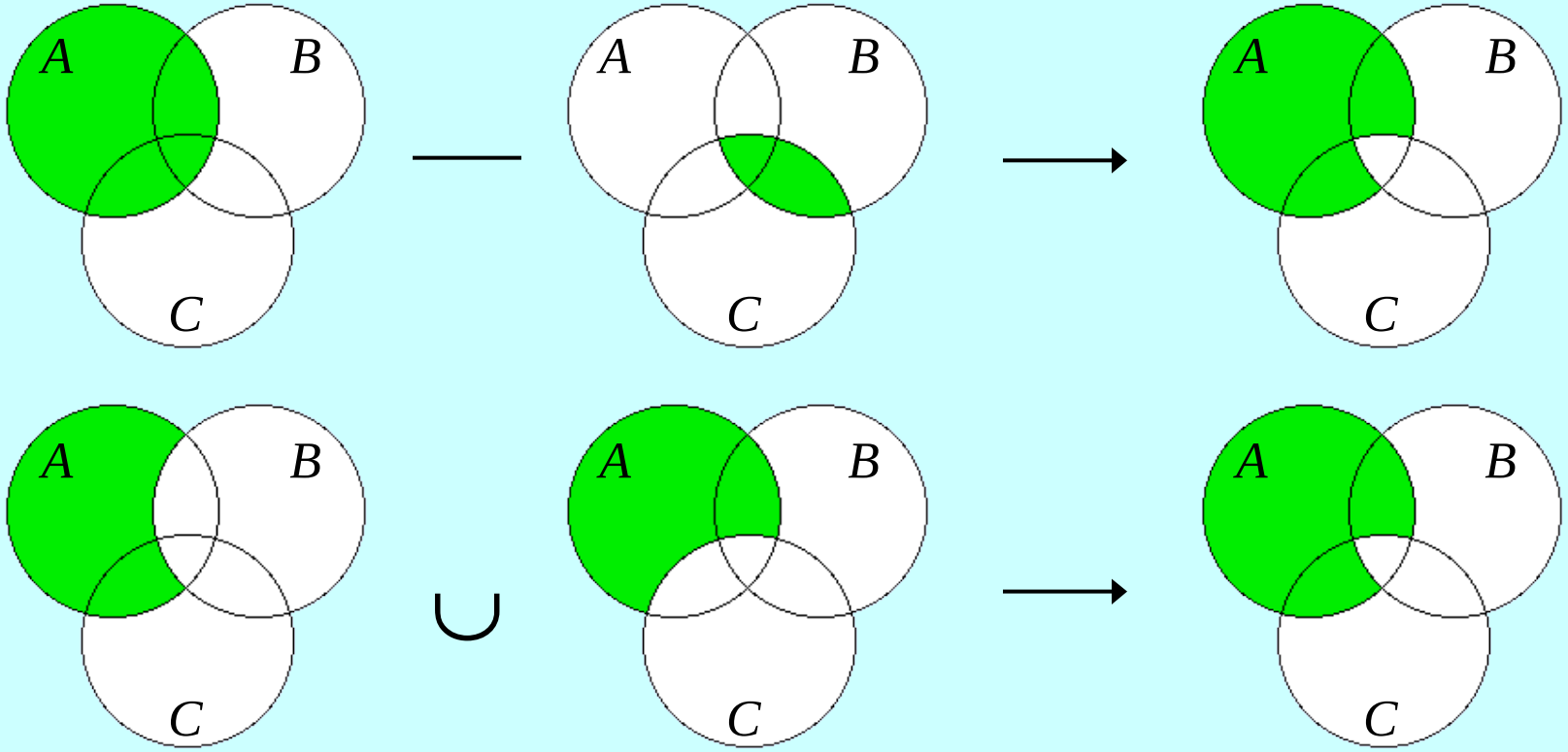
$A - B$

- If $A \subseteq B$, then the circle for A in the Venn diagram lies entirely within (or possibly coincident with) the circle for B.
- If $A = B$, then the circles for A and B are the same.

Venn Diagrams as Informal Proofs

- You can also use Venn diagrams to justify set identities. Suppose, for example, that you wanted to prove

$$A - (B \cap C) \equiv (A - B) \cup (A - C)$$



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*, *set difference*, *subset*, and *equality* as well.
- The primary goal of today's lecture is to write the code that implements those operations.

High-Level Operators in `set.h`

```
/*
 * 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 & element) 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 & element) const;

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

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

Implementing Sets

- Modern library systems adopt either of two strategies for implementing sets:
  - ***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.
  - ***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.
- For the **`Set`** class itself, C++ uses the latter approach.
- 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.

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,char> map;          /* The char is unused */
```

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(ValueType element) {
   map.put(element, ' ');
}
```

. . . and so on . . .

The Implementation Section

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

template <typename ValueType>
bool Set<ValueType>::operator==(const Set & s2) const {
   return isSubsetOf(s2) && s2.isSubsetOf(*this);
}

/*
 * 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(Set & s2) {
   for (ValueType value : *this) {
      if (!s2.contains(value)) return false;
   }
   return true;
}
```

Exercise: Implementing Set Methods

```
template <typename ValueType>
Set<ValueType> Set<ValueType>::operator+(const Set & s2) const {









}

template <typename ValueType>
ValueType Set<ValueType>::first() {









}
```

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.

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`** 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***.
- 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.

Bit Vectors and Character Sets

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

	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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	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	0	0	0	0	0
3	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	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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).

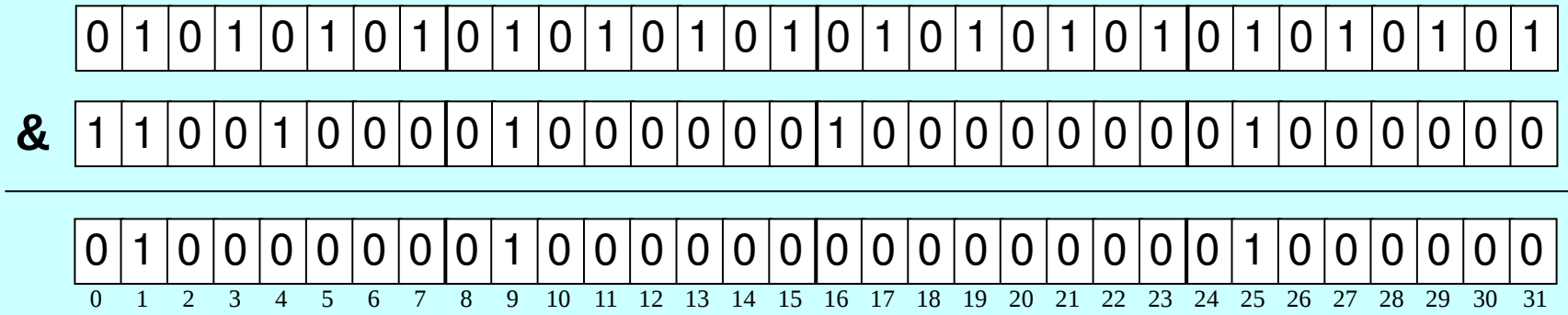
The Bitwise AND Operator

- 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**:

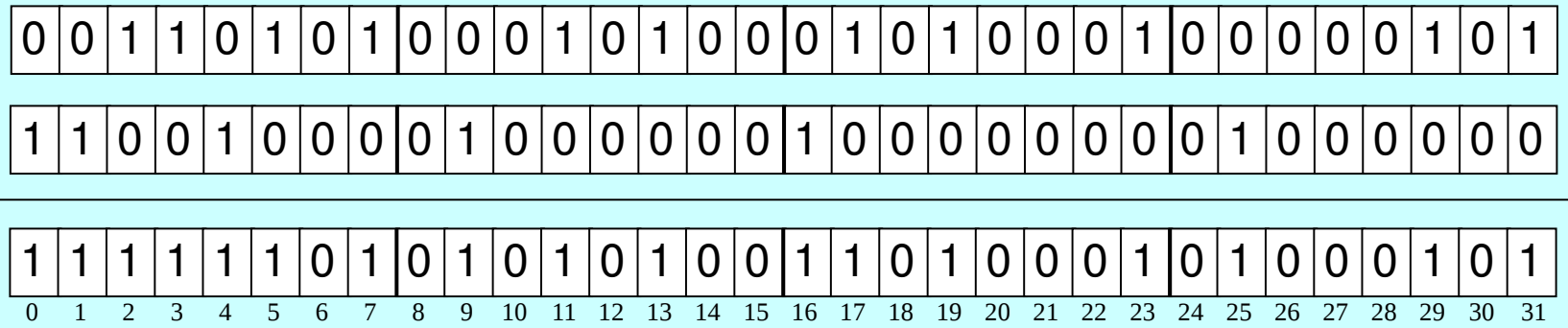


The Bitwise OR Operator

- 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), 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**:

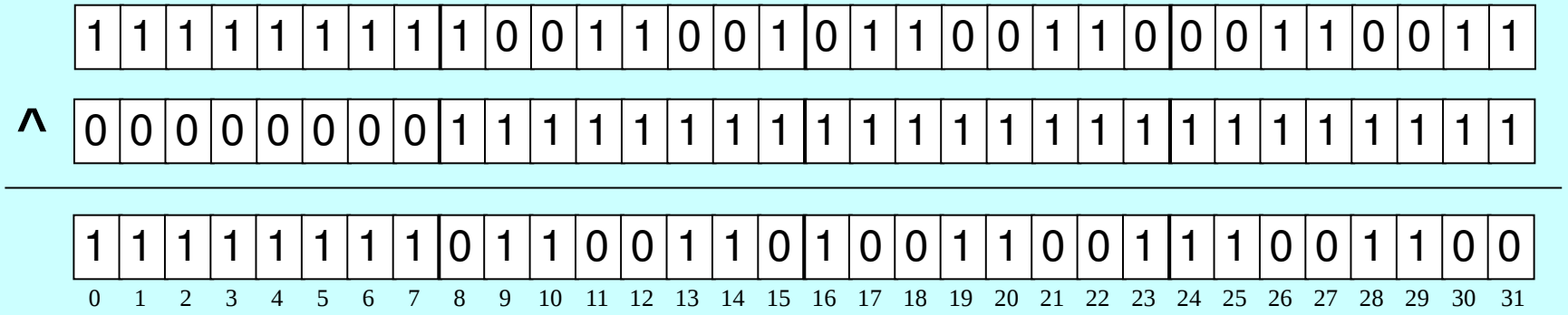


The Exclusive OR Operator

- 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.

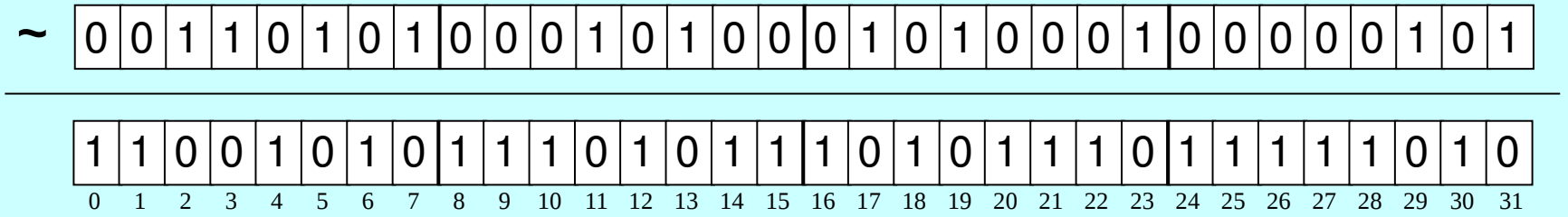
	0	1
0	0	1
1	1	0

- 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:



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:**



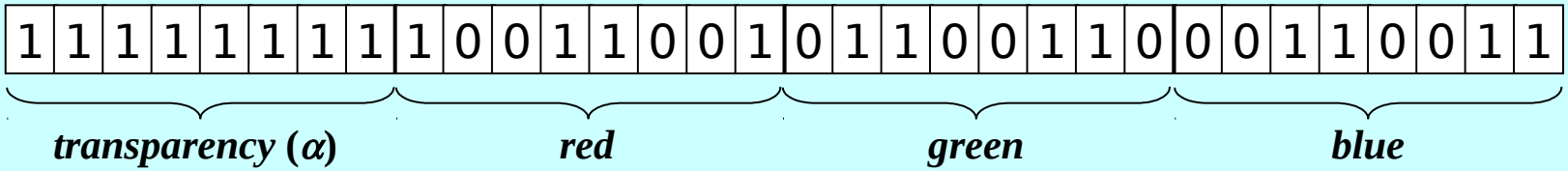
- Question: How could you use the ~ operator to compute the set difference operation?

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 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.
  - If *x* is an unsigned type, the **>>** operator performs a ***logical shift*** in which missing digits are always filled with 0s.

Exercise: Manipulating Pixels

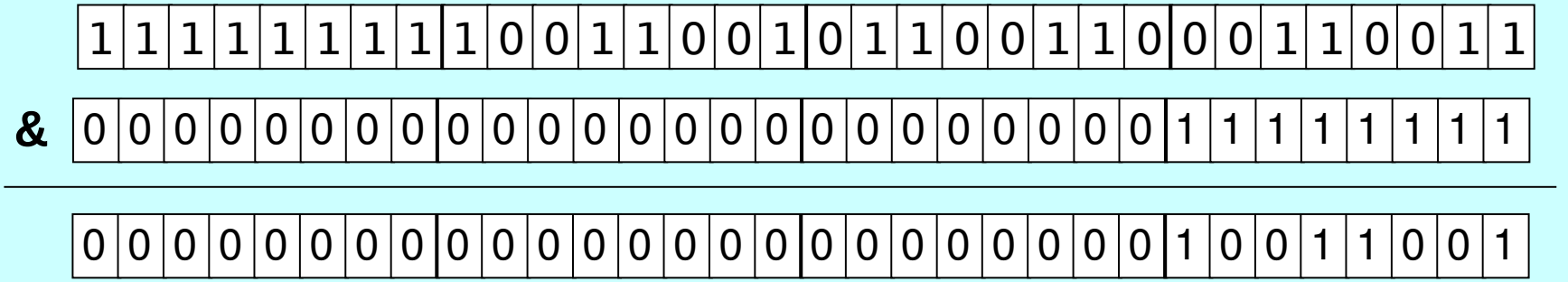
- Computers typically represent each pixel in an image as an **`int`** in which the 32 bits are interpreted as follows:



  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;
```



The End

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**: