Bit Operations

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Leftover topic: Linked lists with pointer-to-pointer
- Bitwise operator review
- Bitwise tasks
  - Setting / clearing / testing selected bits
  - Copying selected bits
  - Movable masks
  - Manipulating a bit field
- Bit fields in structs

Typedef and 2D Arrays

- This can also simplify describing multi-dimensional arrays.

I'm a pointer to a Row, so you can use me like an array of Rows.

Now, I'm an array of 50 rows.

```
typedef int Row[ 20 ];

Row *table;

table = (Row *) malloc( 50 * sizeof( Row ) );
```

- We could do without this typedef, but it makes the types harder to describe.

Oops. This is just the same thing without typedef.

```
int (*table)[ 20 ];

table = (int (*)[20]) malloc( 50 * 20 * sizoef( int ) );
```

Linked List Representation

- Same representation as before.

```
struct NodeTag {
  int value;
  struct NodeTag *next;
};

typedef struct NodeTag Node;
```

Representation for a Node.

```
typedef struct {
  Node *head;

} List;
```

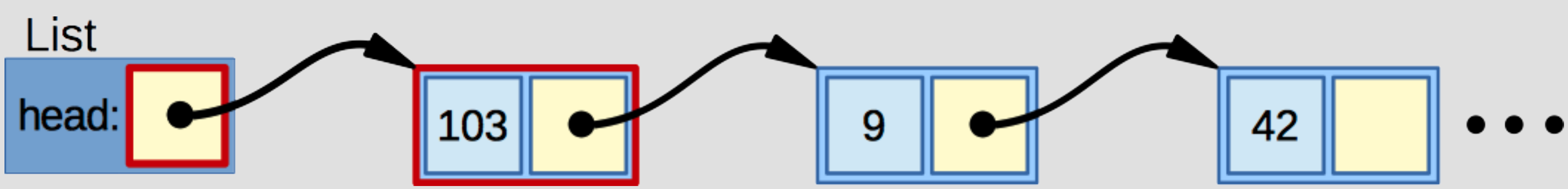
Representation for a whole list.

Removing Nodes, Classic

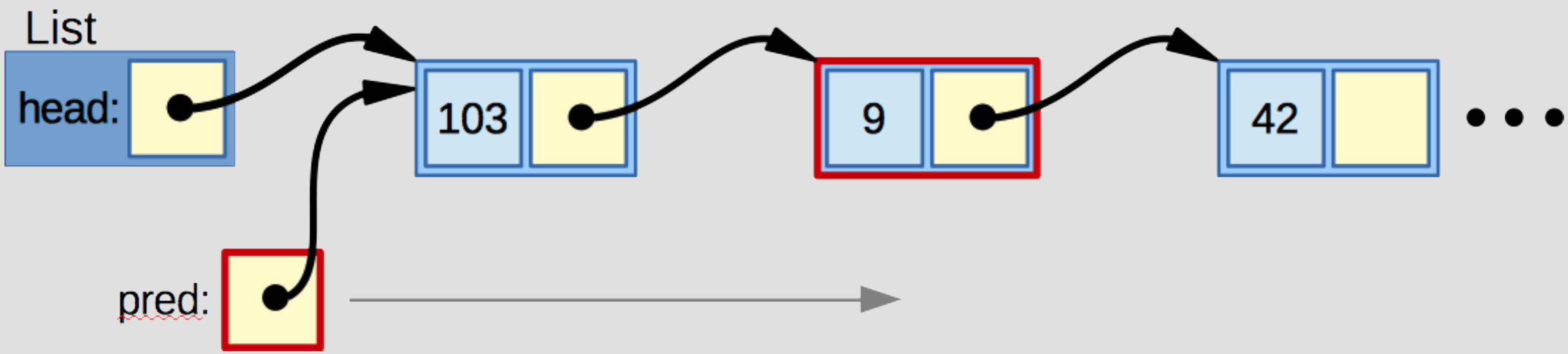
- Normally, to remove a node we might handle the head of the list as a special case:

```
bool removeValue( List *list, int val )
{
  if ( list->head && list->head->value == val ) {
    Node *n = list->head;
    list->head = list->head->next;

    free( n );
    return true;
  }
  …
```

First node as a special case.



Removing Nodes, Classic




```
…;
Node *pred = list->head;
while ( pred->next && pred->next->value != val )
  pred = pred->next;

if ( pred->next ) {
  Node *n = pred->next;
  pred->next = pred->next->next;

  free( n );
  return true;
}

return false;
}
```

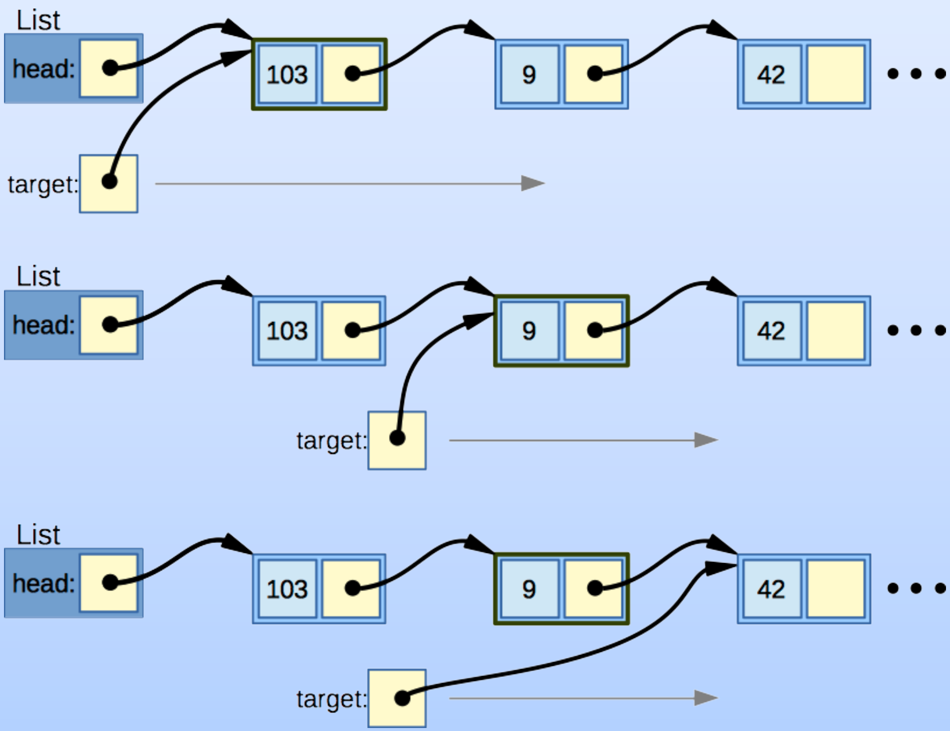
Look for the node **before** the one you want to remove.

Unlink (and free) it.

An Idea that Almost Works

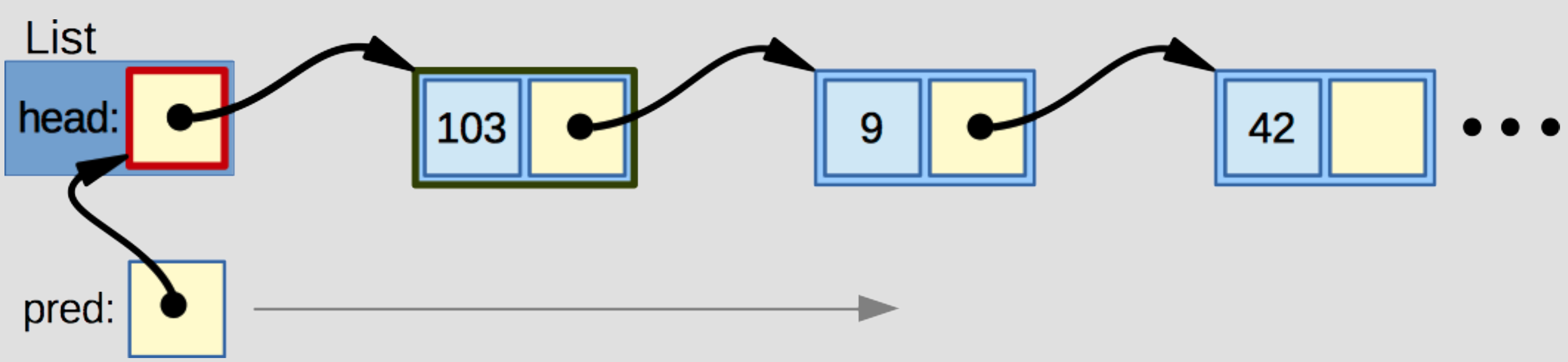
- How about this technique?

```
bool removeValue( List *list, int val )
{
  Node *target = list->head;
  while (target &&
         target->value != val)
    target = target->next;

  if ( target ) {
    Node *n = target;
    target = target->next;

    free( n );
    return true;
  }


  return false;
}
```



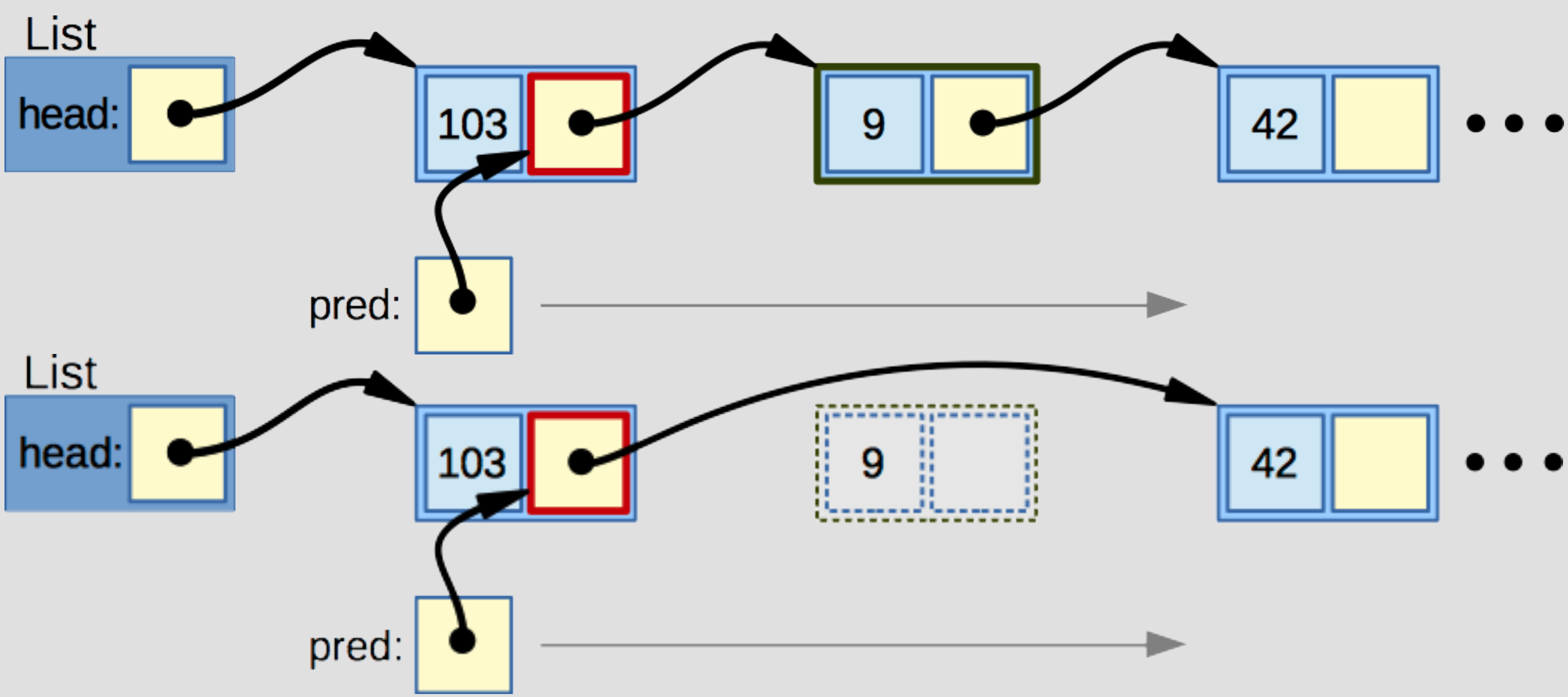
Removing Nodes, "Simplified"

- To remove a node, we must change the pointer that points to it.
  - Every node has such a pointer
  - Either **the pointer inside its predecessor**.
  - **Or, the head pointer** itself.
- Both of these pointer have the same type (pointer to Node)
- We can handle them uniformly via a pointer to pointer to Node.
- We'll start with a pointer pointing to the head pointer:



Removing Nodes, “Simplified”

- As we traverse the list, we will move the ahead to pointers within the nodes.
  - This will give us a way to remove the next node on the list.



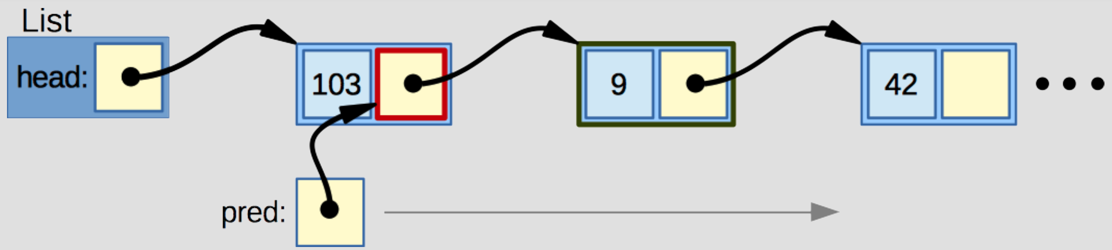
Removing Nodes, Simplified

```
bool removeValue( List *list, int val )
{
  Node **pred = &list->head;
  while ( *pred && (*pred)->value != val )
    pred = &(*pred)->next;

  if ( *pred ) {
    Node *n = (*pred);
    *pred = (*pred)->next;

    free( n );
    return true;
  }

  return false;
}
```

I'm the address of the pointer to the node you're thinking about removing.

Start at the head, then walk down through all the pointers inside the nodes.

If we find a node containing val, remove its node by changing the pointer that points to it.



Looking Inside a Value

- Internally, int and double values are represented with patterns
- A good programming language lets us ignore details of how this is done
  - We can thing of an int as … well, an integer
- But, sometimes we want to work with individual bits in a value … why?
  - Maybe we could pack more values into limited memory.
  - Maybe we need to talk to hardware or work with a file format that requires particular bit patterns
  - Maybe we need to perform compression or encryption
  - Maybe we can better exploit bit-level parallelism in the hardware

Working with Bits

- C99 provides no standard way to …
  - … give a literal constant in binary

```
i = 011001011;
```

  - … read an integer as a string of binary digits

```
scanf( "%b", &i );
```

  - … print a value as a string of binary digits

```
printf( "%b", i );
```

- What can we do?
  - Use octal or hexadecimal
  - Write our own code to work at the bit level.

C Bit-Level Operators

- We have some operators just for manipulating values at the bit level

Operator	Description
++ --	Postincrement, postdecrement
++ -- + - ! ~ (*type*) sizeof	Preincrement, predecrement Unary positive, negative Logical not Bitwise complement Type cast Well, it's the sizeof operator
* / %	Multiply, divide, mod
+ -	Add, subtract
<< >>	Bitwise left and right shift
< <= > >=	Relational operators

More C Operators

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>== !=</td><td>Equals, not equals</td></tr>
<tr><td>&</td><td>Bitwise and</td></tr>
<tr><td>^</td><td>Bitwise exclusive or</td></tr>
<tr><td>|</td><td>Bitwise (inclusive) or</td></tr>
<tr><td>&&</td><td>Logical and</td></tr>
<tr><td>||</td><td>Logical or</td></tr>
<tr><td>? :</td><td>Ternary operator</td></tr>
<tr><td>= += -= *= …</td><td>Assignment, and its friends</td></tr>
<tr><td>,</td><td>Comma operator</td></tr>
</table>

Working With Bits

- We're about to start working with the binary representation of values
- We're going to notice some behaviors we normally don't have to worry about.
  - This one is called sign extension.

```
unsigned char x;
signed char y;

x = 0xA9;  // binary 10101001 (that means 169)
y = 0xA9;  // binary 10101001 (here, that's -87)

printf( "%x\n", x );
printf( "%x\n", y );
```

This prints a9. Good

But, this prints ffffffa9

But, that's how int -87 **is** represented.

Meet ~ (a Unary Bitwise Operator)

- Unary operator ~ is a bitwise complement
- It's like **-** or **!**, but it works across all bits

```
unsigned char x, y, z;

x = 0x1A;   // binary 00011010
y = ~x;     // binary 11100101
```

Really, it's

11111111111111111111111111100101

until you assign it.

- See, ~ is different from these other operators

```
signed char x, y, z, w;

x = 0x1A;   // binary 00011010
y = ~x;     // binary 11100101
z = !x;     // binary 00000000
w = -x;     // binary 11100110
```

Same here, it's

11111111111111111111111111100110

until you assign it.

Binary Bitwise Operators

- The next three operators are binary, they take two operands
  - **&** is a a bitwise **and**
    a 1 in the result only if both corresponding input bits 1
  - **^** is bitwise **exclusive or**
    a 1 in the result only if both corresponding input bits are different
  - **|** is bitwise **(inclusive) or**
    a 1 in the result if either of the corresponding input bits are 1

Binary Bitwise Operators

- Here’s how they work:

```
  0 1 1 0 1 1 0 0
& 1 1 0 0 1 0 1 0
-----------------
  0 1 0 0 1 0 0 0
```

```
  0 1 1 0 1 1 0 0
^ 1 1 0 0 1 0 1 0
-----------------
  1 0 1 0 0 1 1 0
```

```
  0 1 1 0 1 1 0 0
| 1 1 0 0 1 0 1 0
-----------------
  1 1 1 0 1 1 1 0
```

- See, these are different from our old friends.

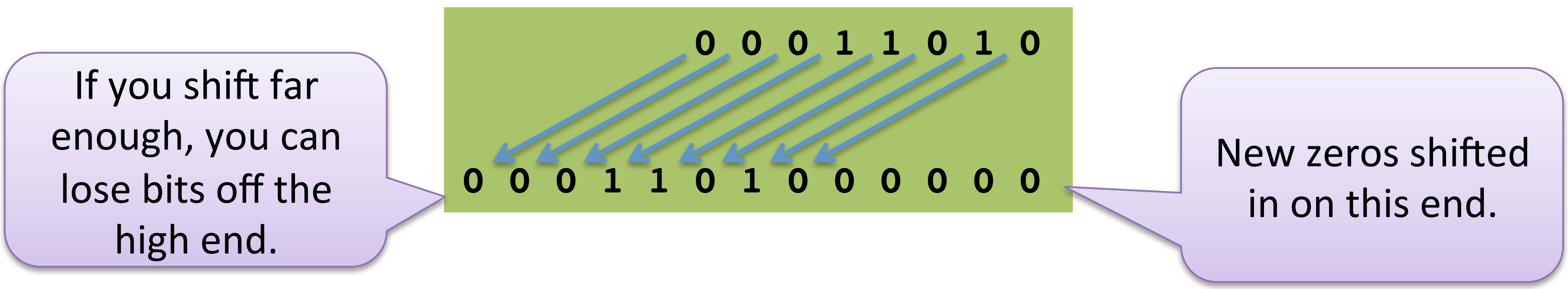
```
   0 1 1 0 1 1 0 0
&& 1 1 0 0 1 0 1 0
------------------
   0 0 0 0 0 0 0 1
```

I’m drawing these like arithmetic problems from elementary school. It’s easier to see the bits if they’re lined up.

```
   0 1 1 0 1 1 0 0
|| 1 1 0 0 1 0 1 0
------------------
   0 0 0 0 0 0 0 1
```

Shift Operators

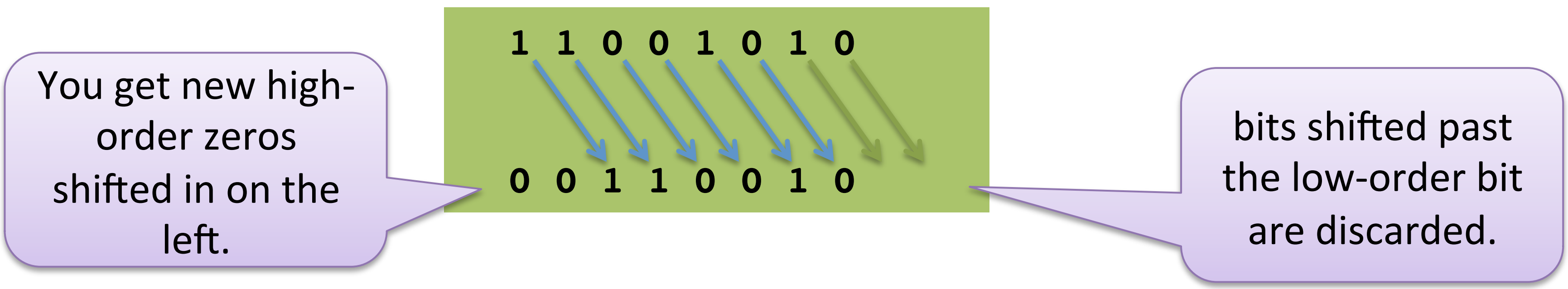
- Left logical shift of `x` by `y` bits: `x << y`
  - For example: `0x1A << 5`



  - Operands should be integer types
  - If `x` has n bits, `y` should be between 0 and n-1
  - You can't left shift by a negative amount, but …

Shift Operators

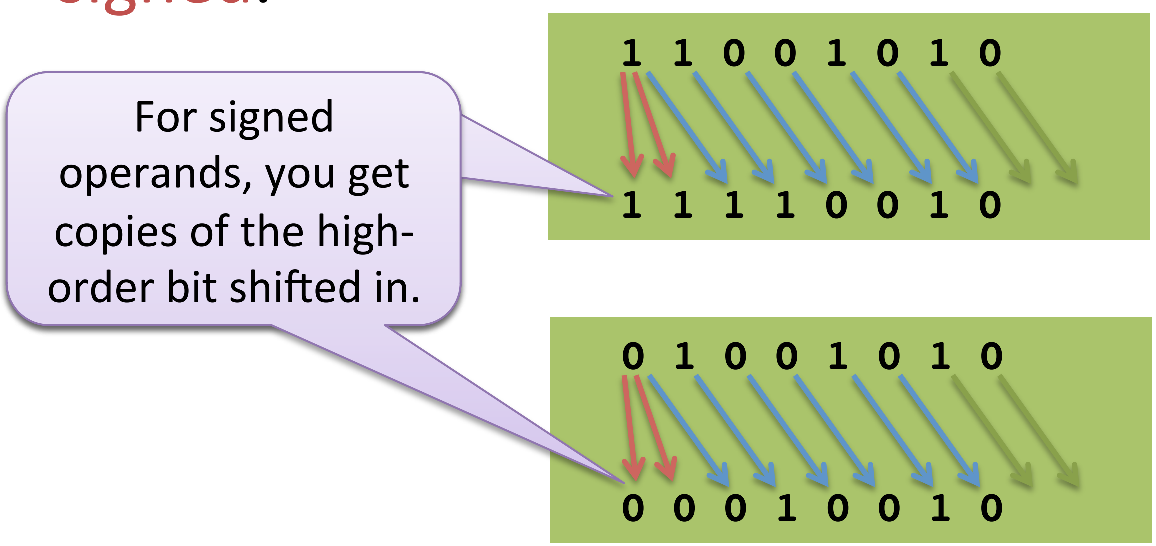
- Right logical shift of `x` by `y` bits: `x >> y`
  - For example: `0xCA >> 2`



  - Again, operands should be integer types
  - If `x` has n bits, `y` should be between 0 and n-1
  - Really, what gets shifted in on the depends ....

Shift Operators

- Right arithmetic shift of `x` by `y` bits: `x >> y`
  - Wait! This is the same operator.
  - Behavior depends on whether the left operand is signed.



Shift Operators

- Why arithmetic shift?
  - For signed values, this maintains the sign.
- We can use left shift to multiply by 2 or any power of 2
  - x << y is the same as x * 2^y
- And, we can use right shift like a divide operation
  - x >> y is the same as x / 2^y

```
unsigned int x y;

x = 75 << 2;   // that's 300
y = 75 >> 2;   // that's 18
```

But, this only works for positive numbers.

OK, How Do We Use These Operators?

- What can we do with these operators?
- Clear selected bits to zero or set them to 1
- Test if selected bits are zero or 1
- Print or count 0 and 1 bits
- Copy selected bits from x to y or to another position in x
- Access a bit field as if it was a tiny integer.

Clearing Selected Bits

- We can think of the & operator as applying a *mask*

```
  0 1 1 0 1 1 0 0
& 1 1 0 0 1 0 1 0
-----------------
  0 1 0 0 1 0 0 0
```

I'm the value you want to work with.

I'm the mask, deciding what bits to keep and discard.

- Wherever we have a 0 in the mask, a bit is cleared
- Wherever we have a 1, the original value passes through

Clearing Selected Bits

- So, we could clear just the low-order 4 bits

```
unsigned char x;
x = 0xD3;          // binary 11010011
x = x & 0xF0;      // mask   11110000
                   // now    11010000
```

- Or, we could clear every other bit.

```
x = 0xD3;          // binary 11010011
x &= 0x55;         // mask   01010101
                   // now    01010001
```

We also have these for bitwise operators.

Setting Selected Bits

- Again, it's just how you think about the bitwise operators
- We can think of | as applying a different kind of mask

```
  0 1 1 0 1 1 0 0
| 1 1 0 0 1 0 1 0
-----------------
  1 1 1 0 1 1 1 0
```

I'm the value you want to work with.

I say what bits to set and which ones remain the same.

- Wherever we have a 1, a bit is set
- Wherever we have a 0, the original value passes through

Setting Selected Bits

- So, we could set just the low-order 4 bits

```
unsigned char x;
x = 0xD3;         // binary 11010011
x = x | 0x0F;     // mask   00001111
                  // now    11011111
```

- Or, we could set just the first and last.

```
x = 0xD6;         // binary 11010110
x |= 0x81;        // mask   10000001
                  // now    11010111
```

Inverting Selected Bits

- With the ^ operator, a 1 in the mask says where to flip a bit

```
  0 1 1 0 1 1 0 0
^ 1 1 0 0 1 0 1 0
-----------------
  1 0 1 0 0 1 1 0
```

I'm the value you want to work with.

I say what bits to change.

- We could use this to flip just the low-order bit

```
unsigned char x;
x = 0xD3;        // binary 11010011
x ^= 0x01;       // mask   00000001
                 // now    11010010
```

Testing Selected Bits

- First, use & to choose the bits you want.

```
  0 1 1 0 1 1 0 0
& 1 1 0 0 1 0 1 0
-----------------
  0 1 0 0 1 0 0 0
```

- Then what?
  - If the result is zero, none of those bits were 1
  - If the result matches the mask, they were all 1

Careful here, precedence.

```
unsigned char x, mask;
x = 0xD3;                      // binary 11010011
mask = 0x0F;                   // mask   00001111
if ( ( x & mask ) != 0 )       // result 00000011
   …;
if ( ( x & mask ) == mask ) // result 00000011
   …;
```

Counting or Printing Bits

- So, if we can build a mask, we can get to any bits we want.
- What if we want to be able to access every bit … individually.
  - Say, we want to iterate over the bits in an int.
- We can build a movable mask for this
  - `0x01 << i` is a mask with a 1 just in bit position `i`
  - And, of course, `~(0x01 << i)` is the complementary mask

Is this the same as
~ 0x01 << i ?

Counting Bits

- We can iterate over the bits, then apply the mask to check each one.

```
unsigned int bill = 2348291;
int bcount = 0;
for ( int i = 0; i < 8 * sizeof( int ); i++ ) {
  if ( bill & 1 << i )
    bcount++;
}
printf( "That's %d one bits\n", bcount );
```

- We could use the same technique to print a number in binary.

Copying Selected Bits

- Say we wanted to copy the middle four bits in a character
- In the source, we can use & to mask off everything but the bits we want to copy.
- In the destination, we can use & with a complementary mask to clear out these bits
- Then, we can use | to turn on just the bits copied from the source

Source

```
  1 0 1 0 1 1 0 1
& 0 0 1 1 1 1 0 0
-----------------
  0 0 1 0 1 1 0 0
```

Destination

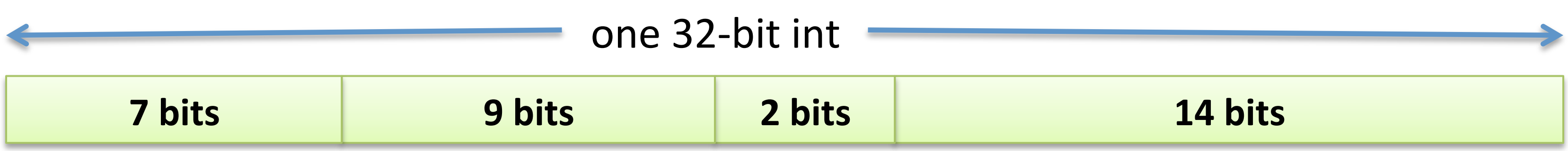
```
  0 1 0 0 1 1 1 1
& 1 1 0 0 0 0 1 1
-----------------
  0 1 0 0 0 0 1 1
```

Result

```
  0 1 0 0 0 0 1 1
| 0 0 1 0 1 1 0 0
-----------------
  0 1 1 0 1 1 1 1
```

Bit Fields

- We can pack multiple small integer values into a variable.
  - This can save some storage or reduce communication overhead.
  - It can even let us manipulate multiple values in parallel.



Extracting a Bit Field

- To extract a particular field:
  - Mask off the other bits
  - Then, shift down (so you can use the contents of the field like an int)
  - Say, we want to get just the middle 4 bits of a char.

```
   1 0 1 0 1 1 0 1
 & 0 0 1 1 1 1 0 0
 -----------------
   0 0 1 0 1 1 0 0
```

```
   0 0 1 0 1 1 0 0
>>               2
-----------------
           1 0 1 1
```

Changing a Bit Field

- Then, we can manipulate the value of that field.
- And put the result back when we're done.

```
  1 0 1 0 1 1 0 1
& 1 1 0 0 0 0 1 1
-----------------
  1 0 0 0 0 0 0 1
```

Original char, make room for the new field value.

```
        1 0 1 1
+             1
-----------------
        1 1 0 0
<<            2
-----------------
    1 1 0 0 0 0
& 0 0 1 1 1 1 0 0
-----------------
  0 0 1 1 0 0 0 0
```

Add one to the field.

Shift it back to where it goes.

Mask off other bits (in case of overflow)

```
  1 0 0 0 0 0 0 1
| 0 0 1 1 0 0 0 0
-----------------
  0 1 1 1 0 0 0 1
```

Insert the updated value

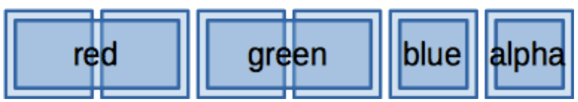
Bit Fields in Structs

- Inside a struct, we can use integer fields with variable numbers of bits
  - The compiler will worry about packing/unpacking them in words of memory.
- Why would we want this?
  - We could save memory, if we need several fields for small integer values
  - Maybe we could match the binary format of some file
  - … or the format for communicating with some hardware device

Using Bit Fields

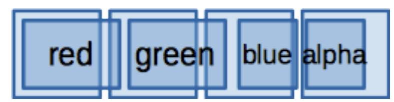
- Normally, fields occupy consecutive groups of whole bytes, maybe even with some padding.
- Internally, bit fields can use parts of individual bytes.

Here's the new syntax




```
typedef struct {
  unsigned short red;
  unsigned short green;
  unsigned char blue;
  unsigned char alpha;
} RegularColor;
```

The compiler says "6 Bytes"




```
typedef struct {
  unsigned int red:9;
  unsigned int green:9;
  unsigned int blue:6;
  unsigned int alpha:6;
} PackedColor;
```

But here just 4.

Using Bit Fields

- Mostly, you can use bit fields like any other integer field.

```
PackedColor c1 = { 511, 256, 0, 32 };

int r = c1.green;

c1.blue += 16;

c1.red++;
```

Initialize them.

Read them.

Assign and change them.

Overflow them.

- But, you can't take their addresses. A bit field may start inside a byte.

```
void *ptr = &( c1.red );
```

Wait Just a Minute!

- Didn't we just learn how to do bit fields, with bitwise operators!
- Was that a waste of time?
  - ~~Yes~~ … I mean No.
- For bit fields in a struct, the compiler determines the layout.
  - So code that depended on a particular packing of the bits wouldn't be portable.
  - And if we really wanted to use a file format or talk to hardware with bit fields in a particular format …
  - … we'd probably have to write that code ourselves

Wait Just a Minute!

- So, are bit fields in a struct useful?
- Yes
  - They can reduce the memory footprint of your program.
  - They can give you convenient access to bit fields in a file, if portability isn't important.