Expressions, Statements and Functions

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- New-ish operators in C
- Type Conversion
  - Type Casts
  - The Usual Arithmetic Conversions
- Flow-of-control
- Functions

C Expressions

- C and Java have many of the same operators
- Here are the first ones we'll talk about, in order of precedence.

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

More C Operators

Operator	Description
== !=	Equals, not equals
&&	Logical and
||	Logical or
? :	Ternary operator
= += -= *= …	Assignment, and its friends
,	Comma operator

Meet Sizeof

- sizeof is an operator that tells you how much memory it takes to store something.
- It works for types:

```
sizeof( int )
```

- Or variables:

```
sizeof( a )
```

The parentheses aren't required, but you have to think about precedence.

- Or other values:

```
sizeof( x * y )
```

- We're going to use this operator **a lot**.

How Much Memory : size_t

- C has a type for talking about how much memory you need
- It's called **size_t**
- In practice, this type is aliased to one of the types you've already seen.
  - Like **unsigned long** on the common platform.
- There's a conversion specification for this type:

Get it? Size – type?

```
printf( "%zd\n", sizeof( x ) );
```

```
size_t s;
scanf( "%zd", &s );
```

Meet The Comma Operator

- In C, you can chain together multiple expressions with a comma
- Sub-expressions get evaluated in order, left-to-right
- The whole expression evaluates to the value of the last sub-expression.

```
int a = 5, b = 10;
int c;

c = a, b;
c = ( a, b );
```

This won't do what you might think.

There, that's better
(but, really, why would you do this?)

- Mostly useful for packing multiple expressions in in a space made for one expression.

Meet The Ternary Operator

- We had this in Java, but sometimes students don't get to see it until this class.
- It's an expression that works like an if/else statement.

$$test \; ? \; expr_1 \; : \; expr_2$$

```
// Choose the larger of a and b.
c = a > b ? a : b;

// Choose the right word to print.
printf( "that'll be %d %s\n", x,
        x == 1 ? "dollar" : "dollars" );
```

Type Conversion

- C can convert between its various basic types (int, float, unsigned short, long, bool, etc.)
  - This can be specified explicitly, via a type cast
  - Or implicitly, by combining types in an expression, or providing one type where the compiler expects another

```
float f;
unsigned char uc;
short s;
double d;
…
s = (short) d;
d = f + ( uc * s );
```

Here, we're explicitly asking for a type conversion.

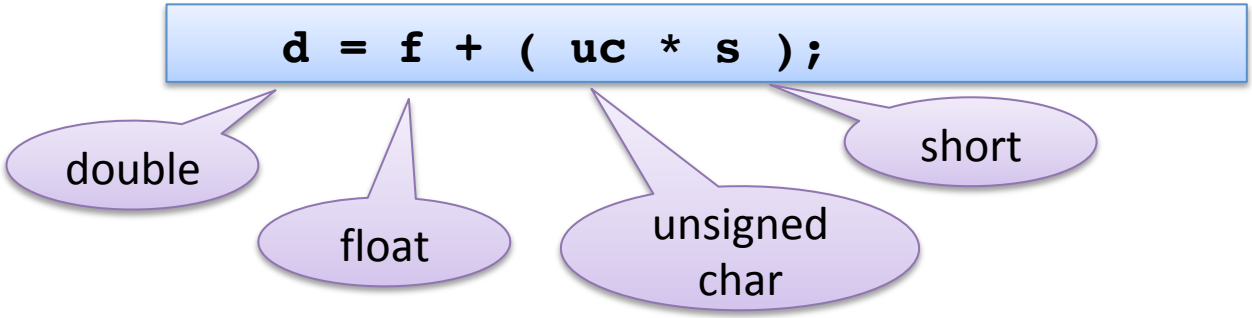
Here, we're implicitly asking for lots of conversions.

The Type Cast

- A type cast is a kind of operator
  - It lets the programmer specify a type conversion
- Syntax: `(`*`destination_type`*`)` *`expression`*
- Example: `c = (unsigned char) d;`
- Remember, the type cast has very high precedence
- A special case, casting to void
  - We can tell the compiler we don't need the result of an expression
  - Example: `(void) f( x );`
  - Why? Mostly to suppress compiler warnings.

Implicit Conversions

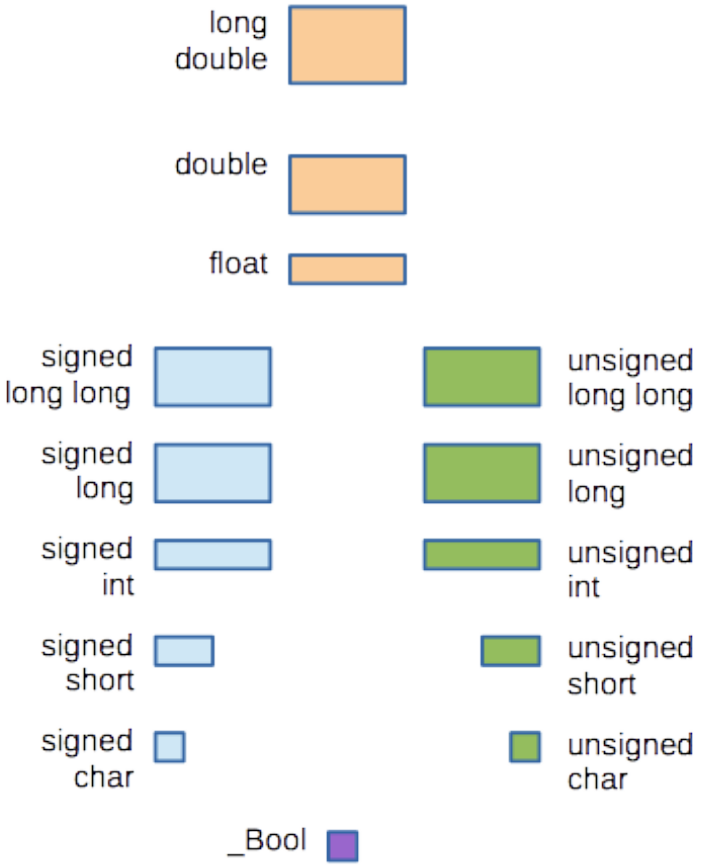
- Internally, the hardware typically can't perform arithmetic on mixed types
- C will automatically insert numeric conversions in order to evaluate an expression
- Lots more than Java would perform automatically.



- The standard defines a set of *Usual Arithmetic Conversions*

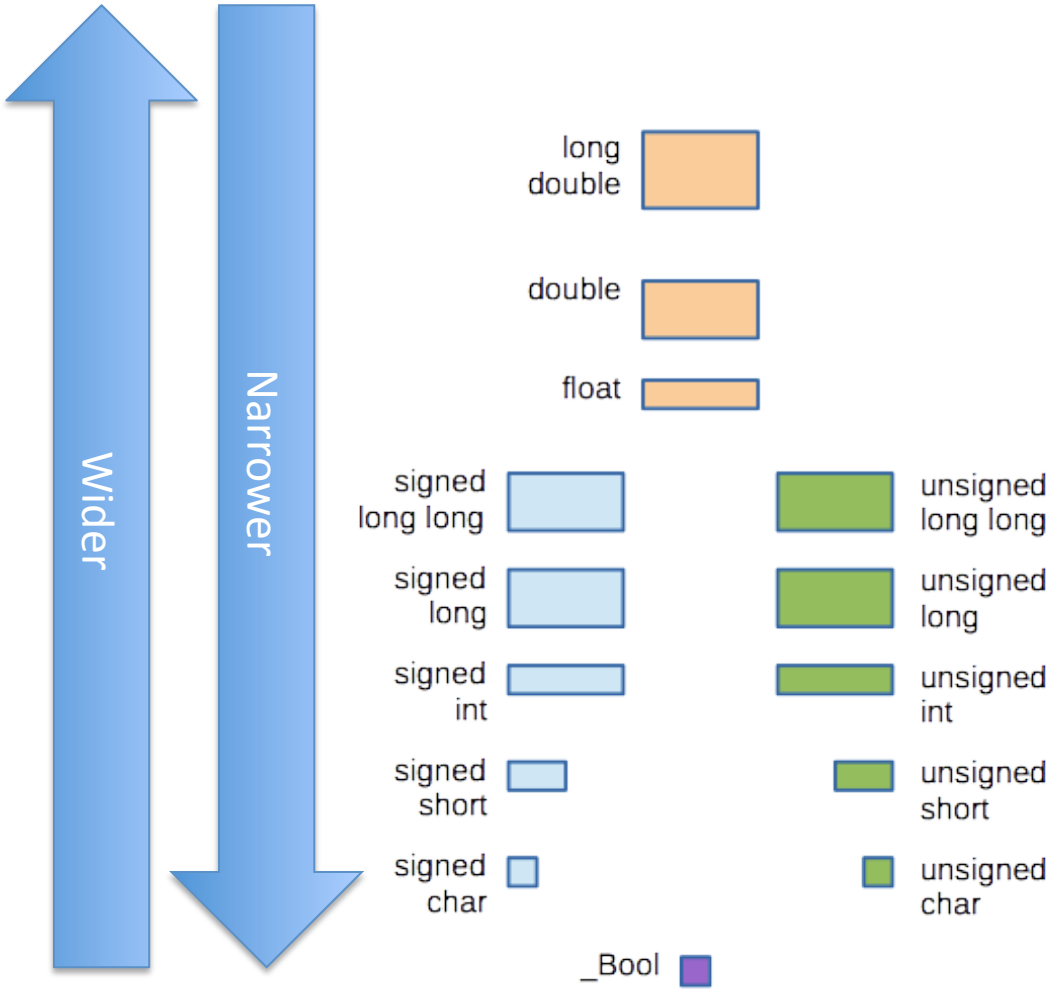
Usual Arithmetic Conversions

- The capacity of each type can depend on the compiler and the platform.
- C describes rules general rules that can be applied to any platform.
  - We will review these in general.
  - And, for the common platform.



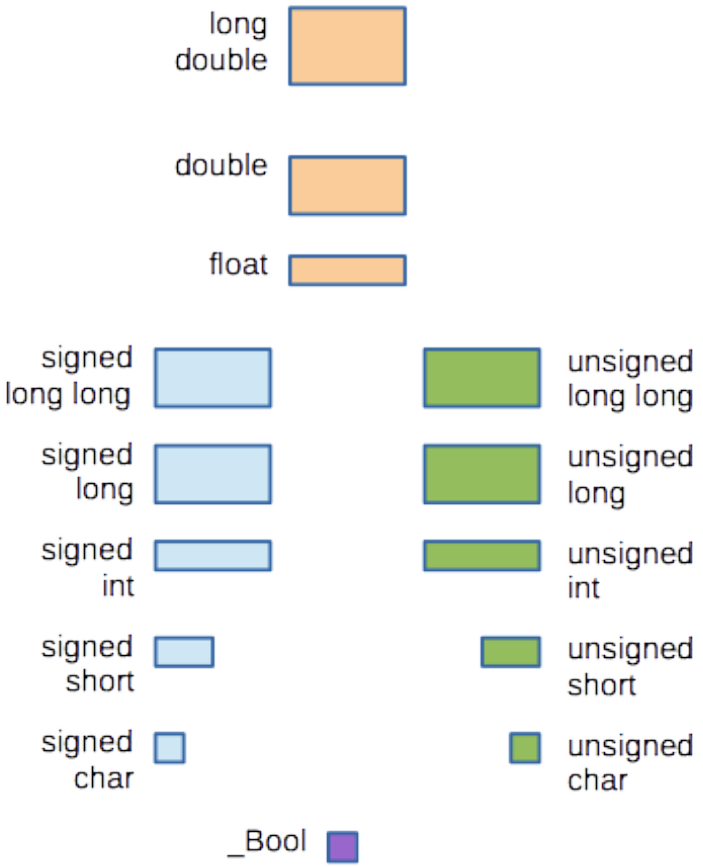
Usual Arithmetic Conversions

- As we move up this chart, we tend to get greater capacity.
  - We can think of some types as *wider* or *narrower* than others.



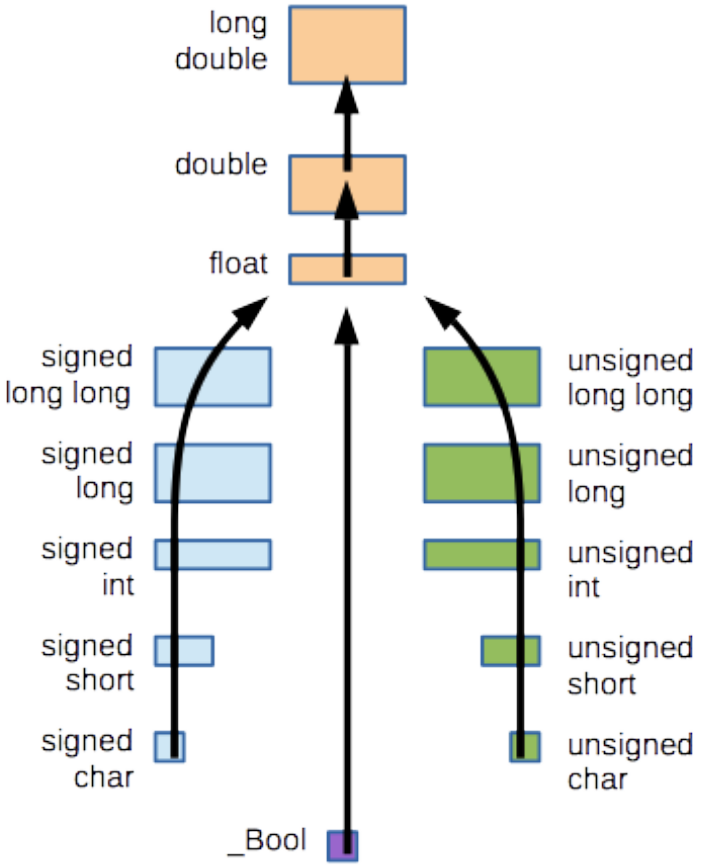
Usual Arithmetic Conversions

- There are several specific rules … but they reflect a few general ideas.
  1. Use the wider of the two types if it's good enough.
  2. Don't use types smaller than int.
  3. If operands are the same size but mixed signed-ness, use unsigned.



Usual Arithmetic Conversions

- If either value is a real ...
  - If just one operand is real → convert to that type.
  - If both are real types → convert to the wider one.



Examples

```
int * float
```

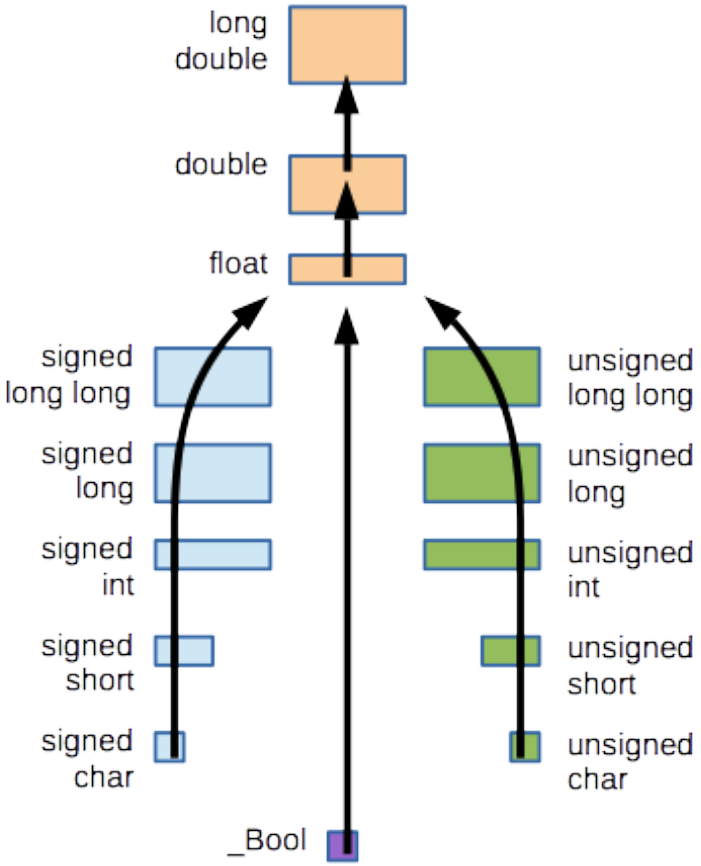
```
float * float
```

```
double + unsigned char
```

```
double + double
```

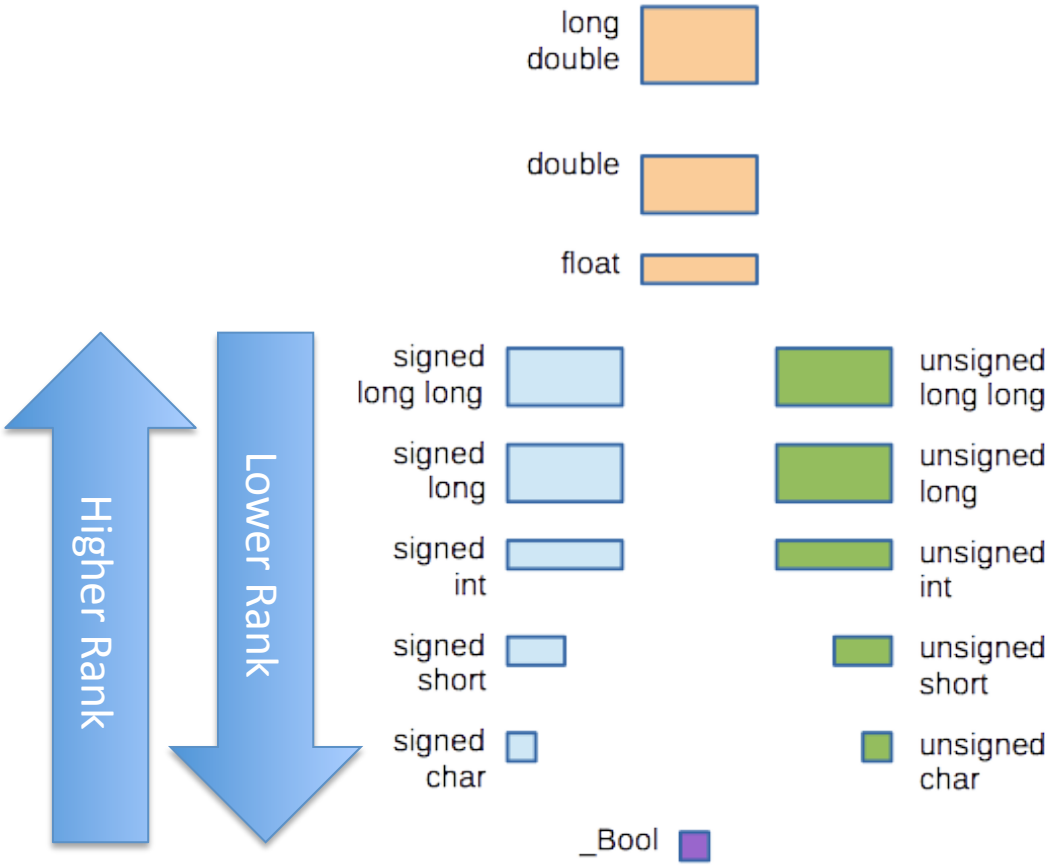
```
float - long double
```

```
long double - long double
```



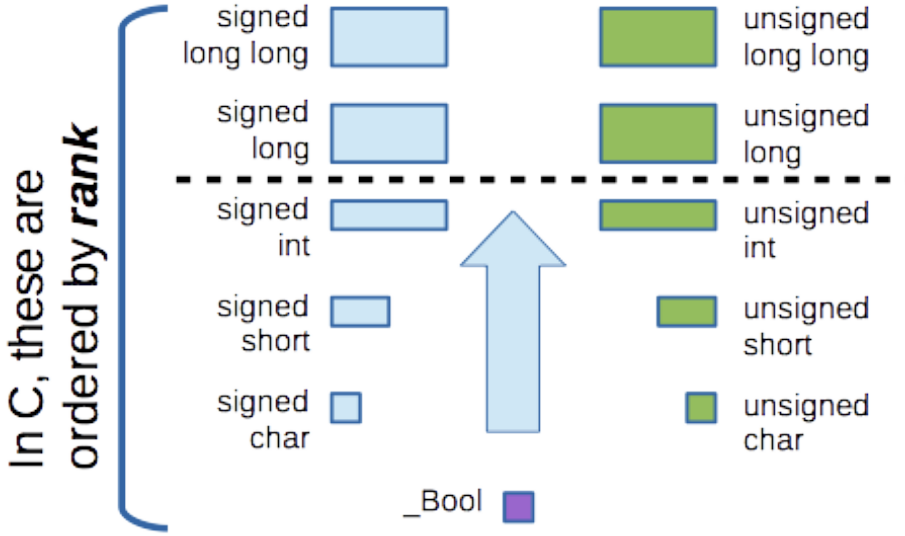
Usual Arithmetic Conversions

- OK. What if neither type is real-valued?
- C uses the *rank* to describe the relative size of its integer types.



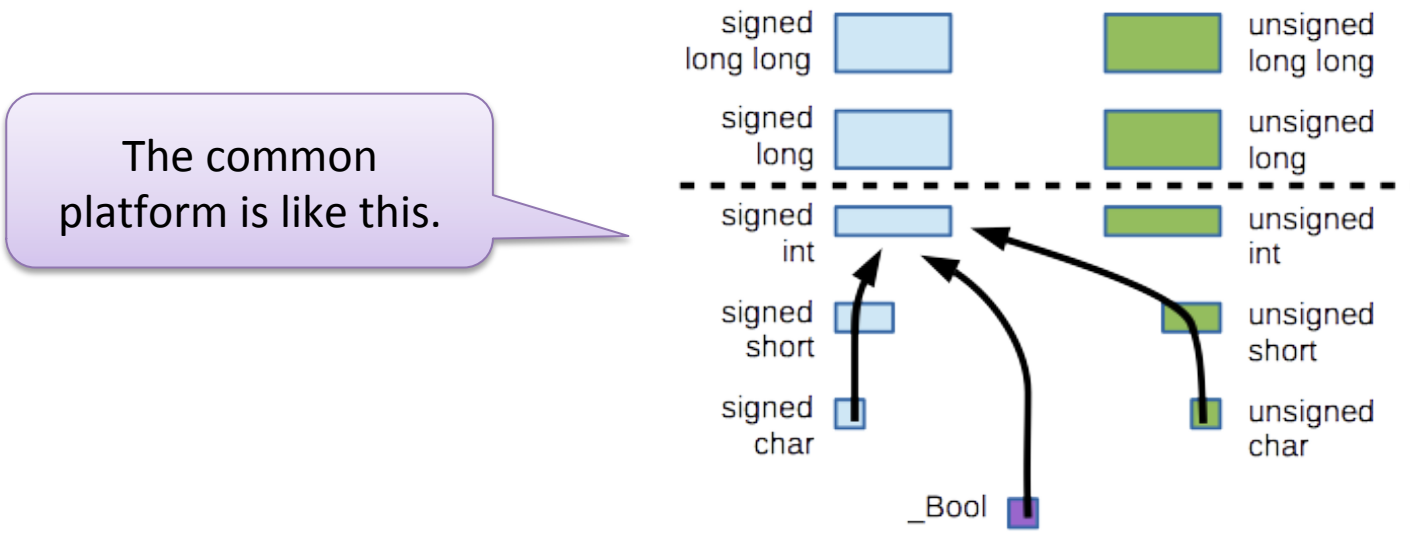
Usual Arithmetic Conversions

- Everything below the rank of **`int`** gets promoted to **`int`** or **`unsigned int`** for arithmetic.



Usual Arithmetic Conversions

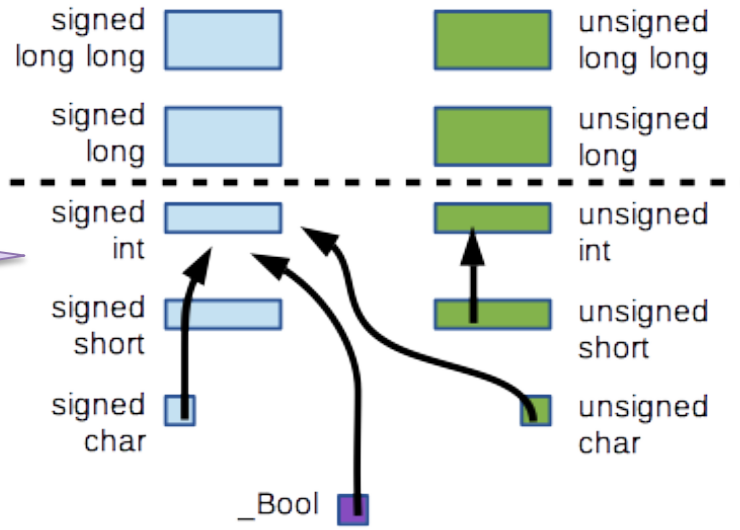
- Lower ranks are promoted to **`int`** if it's large enough to hold all values.



Usual Arithmetic Conversions

- On some platforms (not ours), unsigned short or unsigned char may be promoted to **`unsigned int`**
  - If **`int`** doesn't have enough capacity.

The common platform is **not** like this, but other platforms could be.



Examples

```
char * unsigned short
```

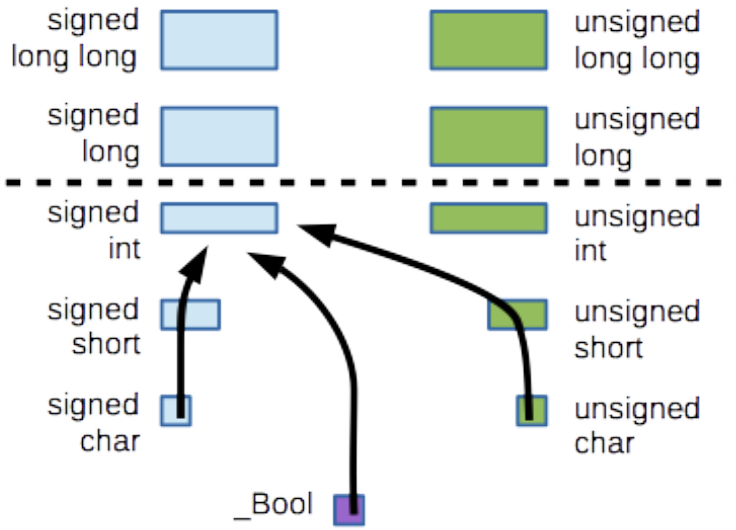
```
int * int
```

```
bool + short
```

```
int + int
```

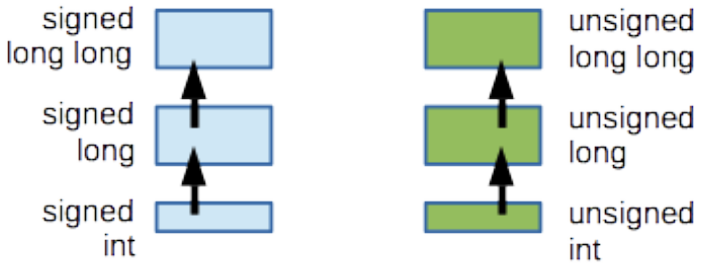
```
char – unsigned char
```

```
int - int
```



Usual Arithmetic Conversions

- Given different types with the same signed-ness, we promote to the higher-ranked (wider) type.

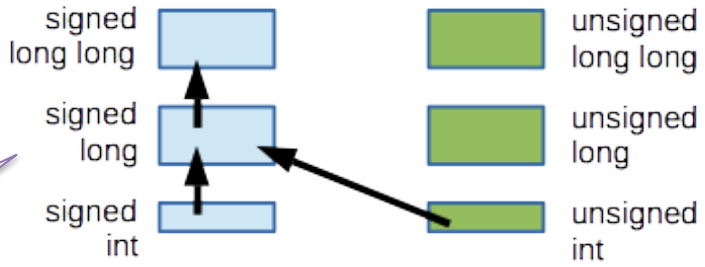


- What's left?
  Just these six types with mixed signed-ness.

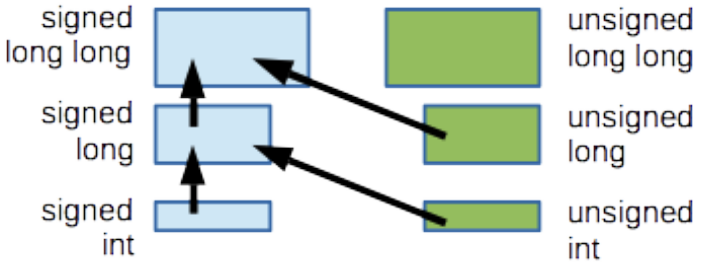
Usual Arithmetic Conversions

- So .. you have a signed parameter and an unsigned parameter.
- Promote to the signed type if can store every value the unsigned type can.

The common platform is like this.

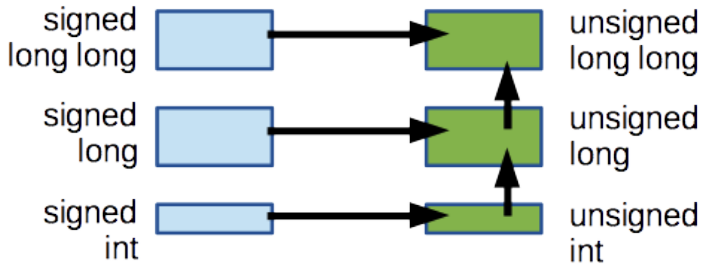


Other platforms might be like this.
(if long long was actually larger than long)



Usual Arithmetic Conversions

- **Otherwise**, use the **unsigned** version of the **higher-ranked** type.



- So, given two equal-ranked types of different signed-ness, C will use the **unsigned** type.

```
unsigned int x = 999;
while ( x ) {
   x = x - 1;
}
```

That makes sense. You probably want **unsigned** here.

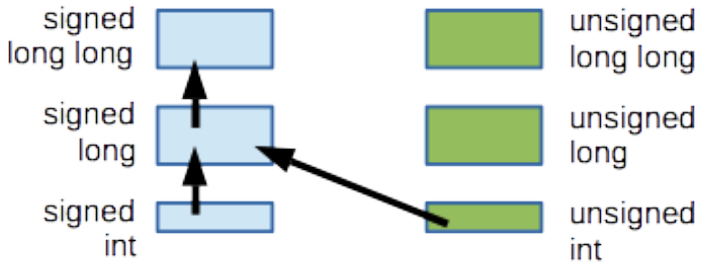
unsigned int

int

Usual Arithmetic Conversions

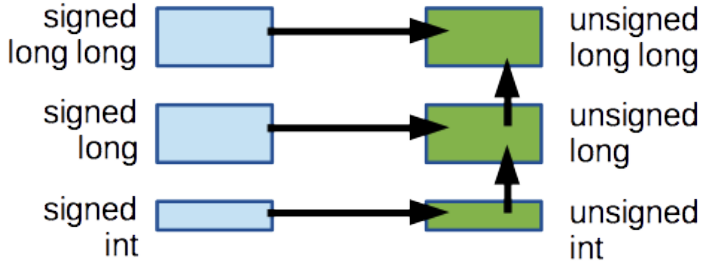
```
long * unsigned int
```

```
long * long
```




```
int + unsigned long
```

```
unsigned long + unsigned long
```




```
long long - unsigned long
```

```
unsigned long long - unsigned long long
```

Fun with Usual Arithmetic Conversions

- So, in the given code what is the type of the red operator? (what type will its operands be converted to so it can be evaluated)

```
float f;
int i;
unsigned char uc;
i + uc * f;
```

- How about here?

```
signed char sc;
long int l;
short s;
sc * s + l;
```

Fun with Usual Arithmetic Conversions

- How about here?

```
signed char sc;
unsigned int ui;
long int l;
sc / ui * l;
```

- How about each operator from our original example?

```
float f;
unsigned char uc;
short s;
double d;
d = f + ( uc * s );
```

Assignment and Implicit Conversion

- Implicit conversions can also happen in assignment
- Here, it's easy
  - the source value just gets converted to the destination type.

```
char c;
int i;
double d;
i = c * d;
```

Resulting double converted to int for assignment.

Function Parameters and Implicit Conversion

- In C, function parameters are also easy.
  - Since there's no overloading.
- Return value gets converted to the return type.
- Parameters values get converted to the parameter types expected by the function.

```
int f( double x, bool y )
{
  if ( y )
    return x + 1.0;
  return x - 1.0;
}
```

Converted to an int

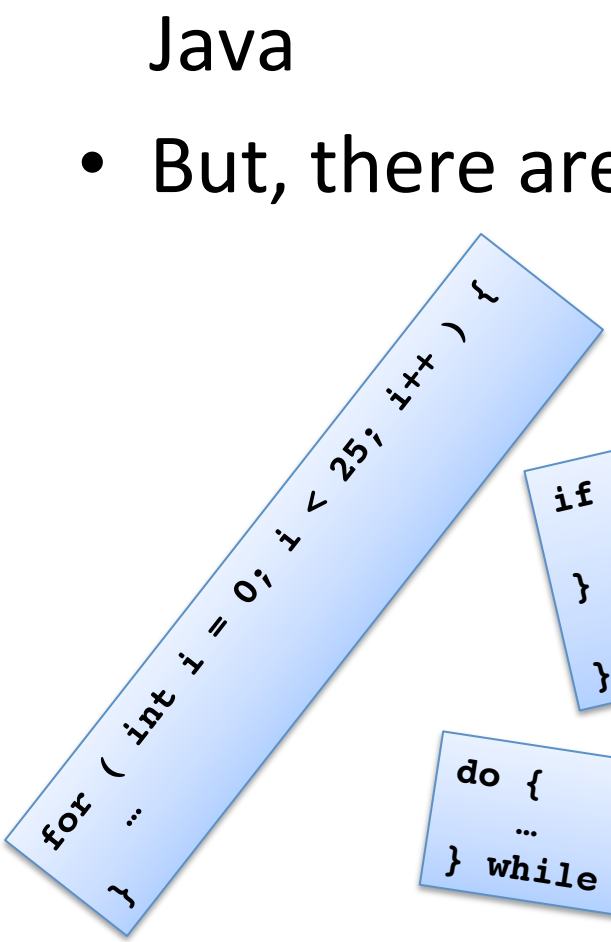
```
...

short s = f( 'a', 3.14 );

...
```

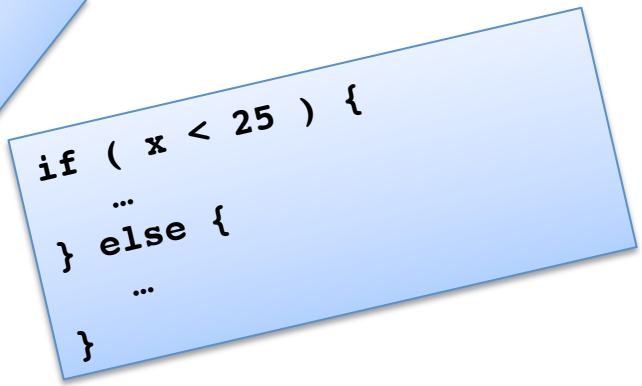
Converted to double

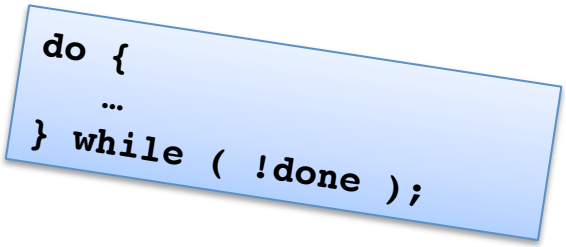
Converted to bool

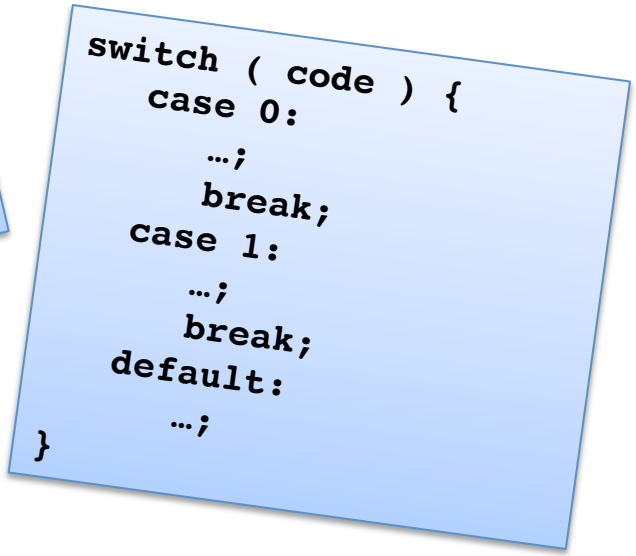
Flow-of-Control Statements

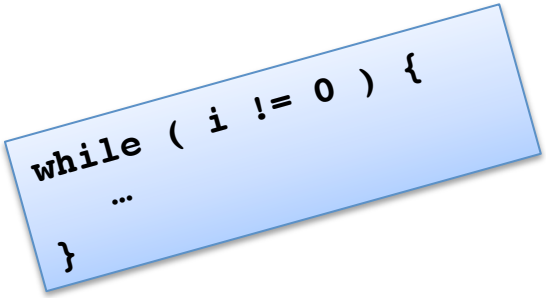
- C shares most flow-of-control statements with Java
- But, there are some differences.











Conditional Expressions

- We have a bool type … but you can get by without it.
- Any basic type can be used as a conditional.
  - If it's zero, it's false.
  - Otherwise, it's true.
- The logical operators (**!**, **&&** and **||**) work with ints.

```
int i = 0;
float f = 2.718;

    i
    i * f
  ( i * f ) && f
```

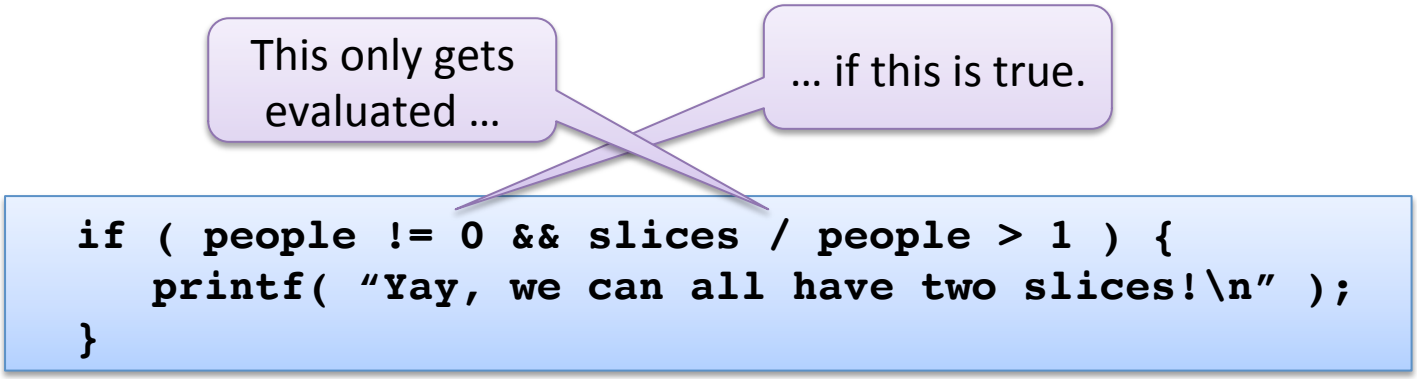
I'm an int

I'm a float

but, I'm an int

Short Circuiting

- As in Java, the && and || operators are guaranteed to be short circuiting
- This can improve performance, we stop evaluating early if the outcome is determined.
- More importantly, it lets us guard some tests with other tests that make sure they are safe.



Constant Expressions

- In some places, C requires a value that can be determined at compile time
  - e.g., in the cases of a switch statement.
  - And a bunch of other places we'll talk about later
- C will let you use an expression here
  - But **the compiler** must be able to determine the value of the expression.
  - So, it can't depend on execution-time behavior
  - These are called *constant expressions*

Constant Expressions

- The compiler will reject our code if we put some other expression where a constant expression is required.

```
int three = 3;

switch ( val ) {

case 0: // trivial
  break;

case 2 / 2: // fine
  break;

case 25/7%2 + 19 - 6*3: // OK
  break;

case three: // Nope
  break;
}
```

Changing the Flow of Control

- `break`, `continue` and `goto` let us alter the normal flow of control.
- Here's a nested loop with obvious flow-of control

```
for ( int i = 0; i < 10; i++ ) {
  for ( int j = 0; j < 10; j++ ) {
    printf( "." );
    printf( "*" );
  }
  printf( "\n" );
}
```

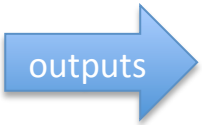
outputs

```
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
```

Changing the Flow of Control

- `continue` lets us skip the rest of the innermost loop's current iteration.

```
for ( int i = 0; i < 10; i++ ) {
   for ( int j = 0; j < 10; j++ ) {
     printf( "." );
     if ( j > i )
       continue;
     printf( "*" );
   }
   printf( "\n" );
 }
```

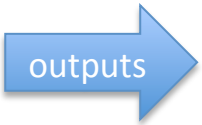



```
.*........
.*.*......
.*.*.*....
.*.*.*.*..
.*.*.*.*.*
.*.*.*.*.*.
.*.*.*.*.*.*
.*.*.*.*.*.*.
.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*
```

Changing the Flow of Control

- `break` lets us bail out of the innermost loop early (and we need it for the switch statement)

```
for ( int i = 0; i < 10; i++ ) {
  for ( int j = 0; j < 10; j++ ) {
    printf( "." );
    if ( j > i )
      break;
    printf( "*" );
  }
  printf( "\n" );
}
```




```
.*.
.*.*.
.*.*.*.
.*.*.*.*.
.*.*.*.*.*.
.*.*.*.*.*.*.
.*.*.*.*.*.*.*.
.*.*.*.*.*.*.*.*.
.*.*.*.*.*.*.*.*.*.
```

- Unlike Java, we can't make a labeled break.

Changing the Flow of Control

- But, that's OK, … err … we have `goto`
- We can put a *label* before any statement in a function.

```
myLabel:
```

- Then, we can jump to that label when we need to.

```
goto myLabel;
```

- We can even jump in and out of loops.

Changing the Flow of Control

- With `goto`, we can jump out of loops of any depth

```
for ( int i = 0; i < 10; i++ ) {
  for ( int j = 0; j < 10; j++ ) {
    printf( "." );
    if ( j > i )
      goto done;
    printf( "*" );
  }
done:
  printf( "\n" );
}
```

This is a destination for a goto.

This says jump to the done label right now.

outputs

```
.*.
.*.*.
.*.*.*.
.*.*.*.*.
.*.*.*.*.*.
.*.*.*.*.*.*.
.*.*.*.*.*.*.*.
.*.*.*.*.*.*.*.*.
.*.*.*.*.*.*.*.*.*.
```

Changing the Flow of Control

- With goto, we don't need any structured flow of control statements at all.
  - But, then we have no … well … structure.

```
int i = 0;
outer_top:
 if ( i >= 10 )
   goto finished;
 int j = 0;
inner_top:
 if ( j >= 10 )
   goto done_2;
 printf( "." );
 printf( "*" );
 j++;
 goto inner_top;
done_2:
 printf( "\n" );
 i++;
 goto outer_top;
finished:
```

outputs

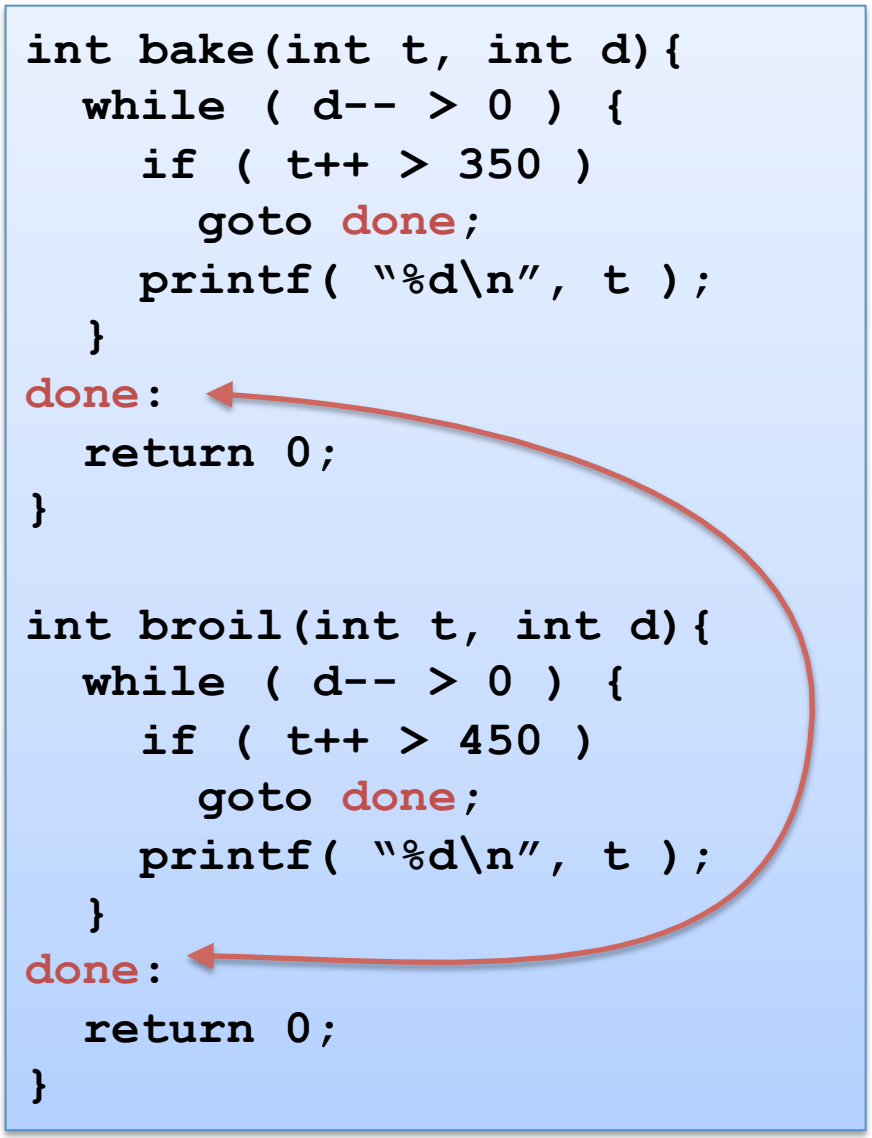
```
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*
```

Nested looping with just goto.

Labels and Scope

- Labels used by goto don't obey block-structured scoping rules.
- You can goto a label that hasn't occurred yet.
- You can goto a label in a more deeply nested block.
- But, you can't goto a label in another function.

```
int bake(int t, int d){
  while ( d-- > 0 ) {
    if ( t++ > 350 )
      goto done;
    printf( "%d\n", t );
  }
done:
  return 0;
}

int broil(int t, int d){
  while ( d-- > 0 ) {
    if ( t++ > 450 )
      goto done;
    printf( "%d\n", t );
  }
done:
  return 0;
}
```



Using and Not Using goto

- You should avoid using goto.
  - You'll just make other developers mad at you
  - And it'll look like you don't know what you are doing.
  - Some coding standards prohibit it
- Exceptions?
  - In some cases, it can simplify error handling.
  - In some cases, it can give you better performance.

```
  if ( … ) {
     for ( … ) {
        while ( … ) {
           if ( … ) {
              …;
              if ( disaster )
                 goto cleanup;
              …;
           }
        }
     }
  }
cleanup:
  /** Clean up partial computation
      and return */
```

Side Effects

- Other than just computing a value, expressions can have side effects.
  - Maybe you use post/pre increment/decrement
  - Maybe you call a function that
    - Reads some input
    - Changes some global variables
- Expressions with effects can be tricky.
- Don’t believe me? Just watch.

Expression Evaluation Order

- The C standard leaves some details of expression evaluation up to the compiler
  - Why? Maybe the compiler can write faster code if we give it some flexibility.
- What if part of an expression has side effects?

```
a = 2;
b = 3;
c = f( &a ) + g( &b ) + h( &a, &b )
```

What if these functions modify a and b?

```
a = 2;
b = ++a + a;
```

```
x = 1;
b = x-- && x;
```

Are we guaranteed anything here?

```
a = 2;
b = a * a + a++ + ++a + a / a;
```

Or here?

Evaluation Order

- C defines a surprisingly small number of *sequence points*
  - Places where know side effects to the left will occur before evaluation continues
- Here they are:
  - statement termination `;`
  - closing parenthesis in a condition evaluation `)`
  - the following operators:
    `a && b` `a || b` `a ? b : c` `a, b`
  - after evaluation of all arguments to a function call
  - after returning a value from a function
- In general, avoid cases where this matters
  - Don't mix side effects with multiple evaluations in the same expression.

But not between function parameters.

Designing with Functions

- While a Java program is a collection of cooperating objects
- … a C program is a collection of cooperating functions.
- Functions are our best building block for a good design.
  - It's how we can share code across multiple parts of an application.
  - It's how we re-use code across multiple projects.
  - A library is mostly a collection of functions we can call.

Function Parameters

- Functions can take any number of parameters
- Including no parameters
  - Like

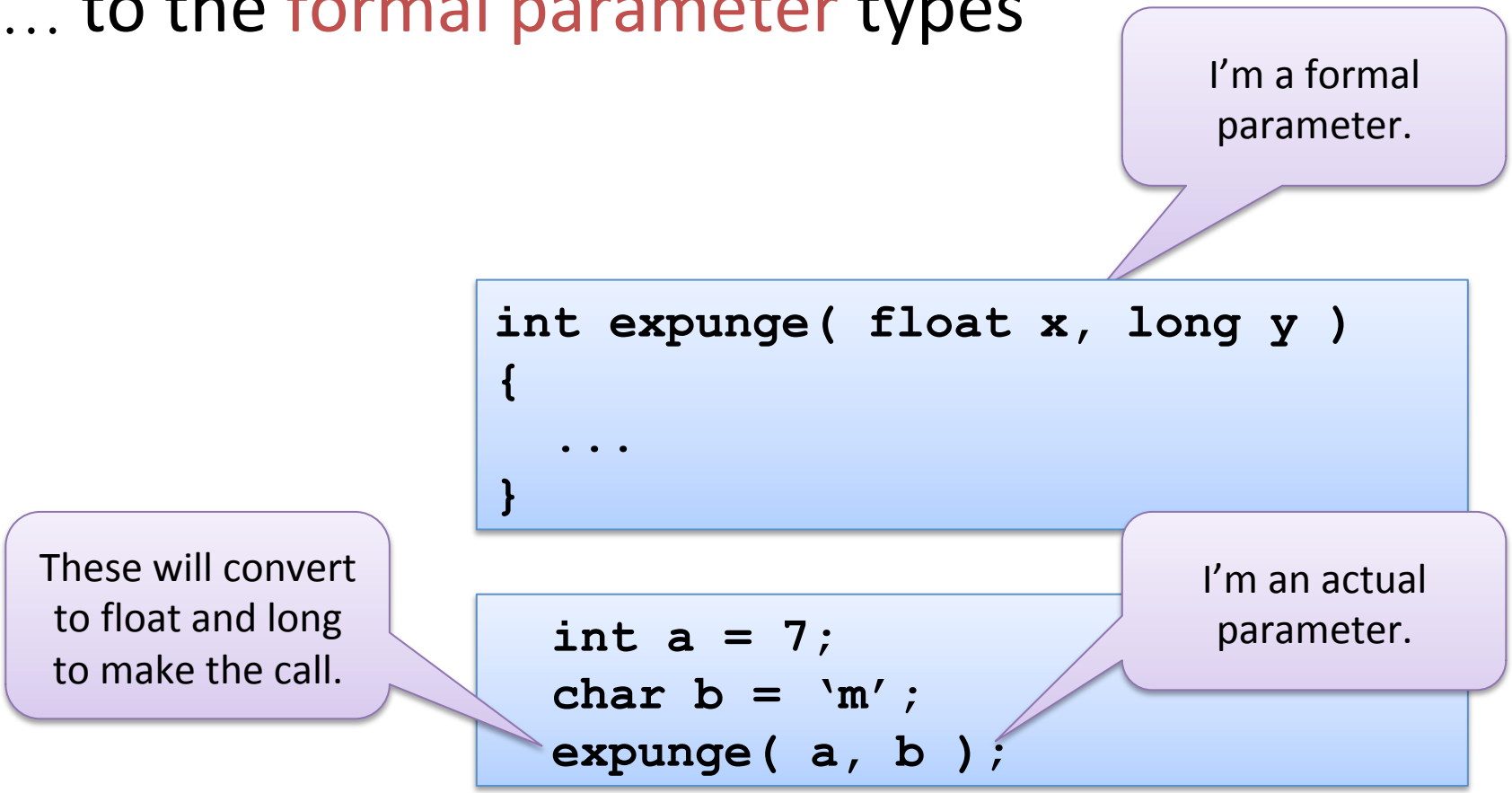
```
void abort( void )
```

A function to immediately exit, unsuccessfully.

This says “I don’t take any parameters”.

Parameter Conversion

- C will implicitly convert *actual parameters*
- … to the formal parameter types



Parameter Conversion

- This is why the compiler wants to know about a function before it gets called.
  - It needs to know what to convert the parameters to.

I guess maybe this function takes two ints?

```
int a = 7;
char b = 'm';
expunge( a, b );
```

Oops. I was wrong. Too late to fix it now.

```
int expunge( float x, long y )
{
   ...
}
```

Knowing a Function's Type

- The compiler can learn about a function's type (return and parameter types) without seeing the whole function definition.

```
int expunge( float x, long y )
{
   ...
}
```

I'm a definition.

- This is done with a *function prototype*

```
int expunge( float x, long y );
```

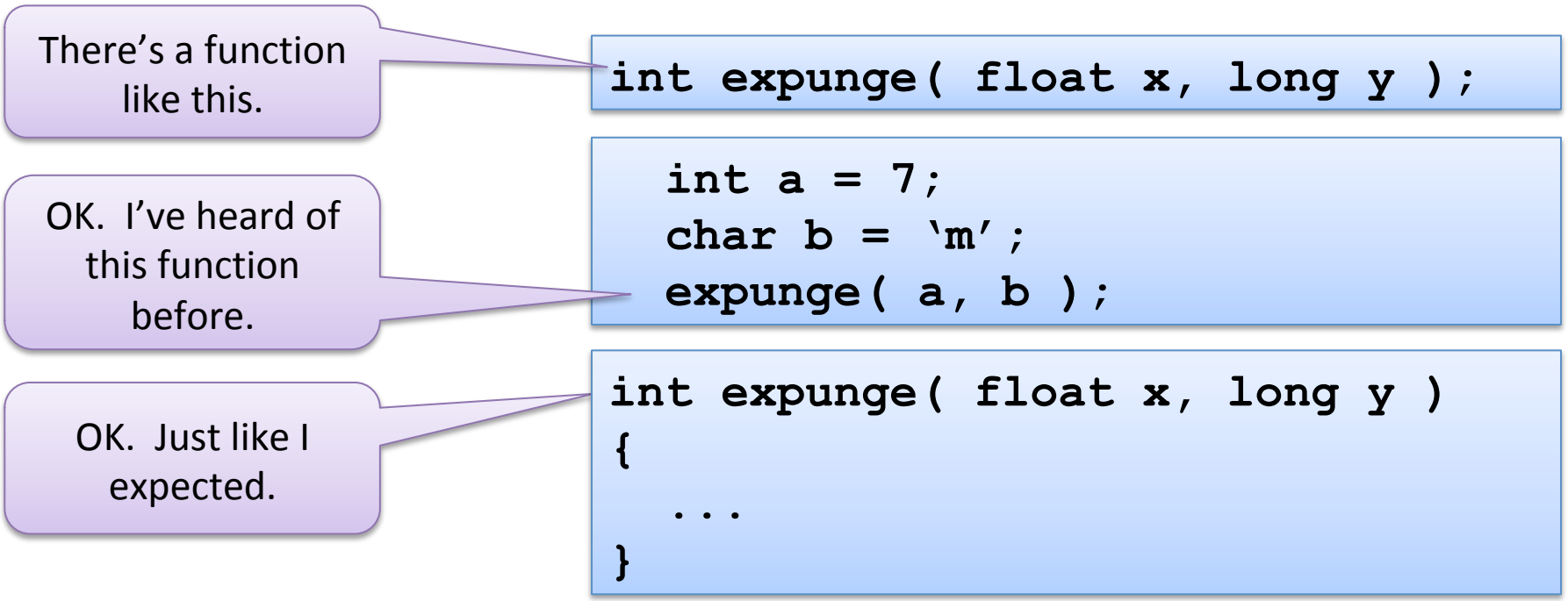
I'm just a prototype.

Definition vs Declaration

- This demonstrates the difference between a definition and a declaration.
- *Definition* : here's some thing, allocate space or write code for it Mr. Compiler
  - Like a function definition.
- *Declaration* : there's something like this out there, but it's defined elsewhere.
  - You may see me trying to use it, so don't act surprised Mr. Compiler.
  - Like a function prototype.

Code Organization

- With prototypes, we can define our functions in any order.
- As long as we prototype them before they're used.



Types in Prototypes

- A prototype just needs the typing information.

```
int expunge( float, long );
```

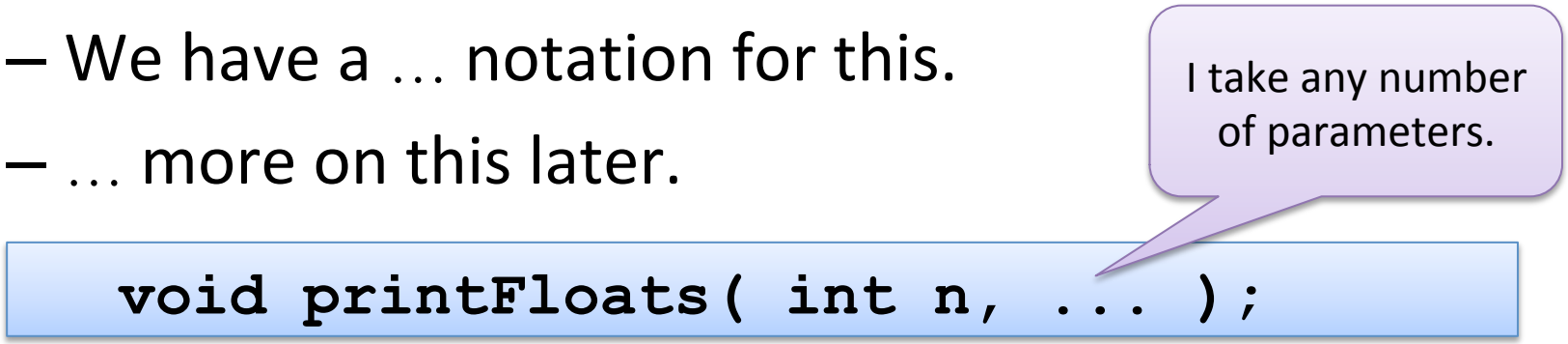
- But, you can include parameter names if you want.

```
int expunge( float x, long y );
```

- Having the names can help with documentation.

Parameter Matching

- To call a function, you need to provide
  - The right number of parameters
  - … and they have to be convert-able to the formal-parameter types
- You can define functions that take any number of parameters of various types.
  - We have a … notation for this.
  - … more on this later.



Function Return Values

- Functions can return a value, if they need to.

```
long double goFigure( int a, char b )
{
   ...
   return 'q';
}
```

Convert to long double? sure.

- The return statement will implicitly convert to the return type, if possible.

Function Return Values

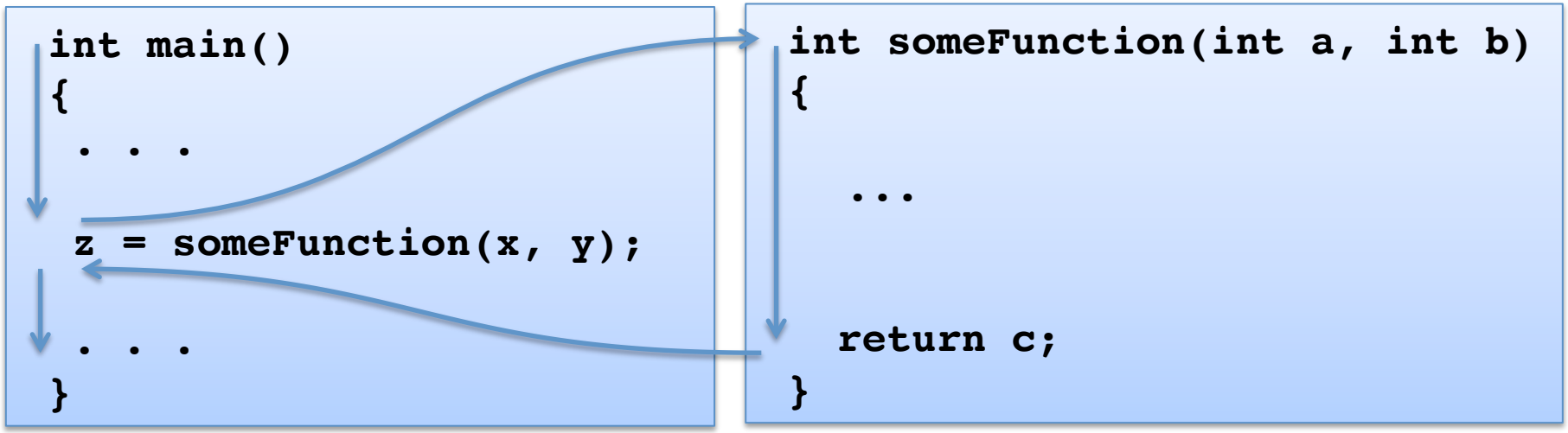
- A void return type says the function doesn't return anything.

```
void printFloat( float x );
```

- It sure would be nice to be able to return more than one value.
  - But, as in Java, each function has one precious return slot.
  - Soon, we'll learn a few tricks to get around that.

Functions aren't Free

- A lot has to go on for a function call to happen.
- Let's consider what happens automatically



- This isn't free.

Function Call Overhead

- The compiler hides a lot of detail from us
- It writes code to:
  - Compute and save the actual parameters
  - Save the return address in the caller
  - Jump to the starting address of the function
  - Allocate space for the function's local variables
  - Run the body of the function
  - Save the function's return value
  - Free space for local variables
  - Jump back to the saved return address.
  - Free space for the parameters and return address

Division of Responsibility

- It's good to have conventions when you're writing code.
- Like for error checking.
  - Who should check for errors: caller or function?

```
int divide( int x, int y ) {
    return x / y;
}
```

Function

```
scanf( "%d%d", &a, &b );
c = divide(a, b);
```

Caller

Checking for Parameter Errors

- Have the function always check?

```
int divide( int x, int y ) {
    if ( y == 0 )
        return 0;
    return x / y;
}
```

```
scanf( "%d%d", &a, &b );
c = divide(a, b);
```

More consistent behavior.

Extra overhead every time you call.

Checking for Parameter Errors

- Make the caller responsible?

```
int divide( int x, int y ) {
    return x / y;
}
```

```
scanf( "%d%d", &a, &b );

if ( b != 0 )
    c = divide(a, b);
```

Maybe some undefined behavior.

Documenting this is a good job for a comment.

Consider **"behavior is undefined when …"**

Caller can know more about context.