Types and Variables

CSC 230 : C and Software Tools

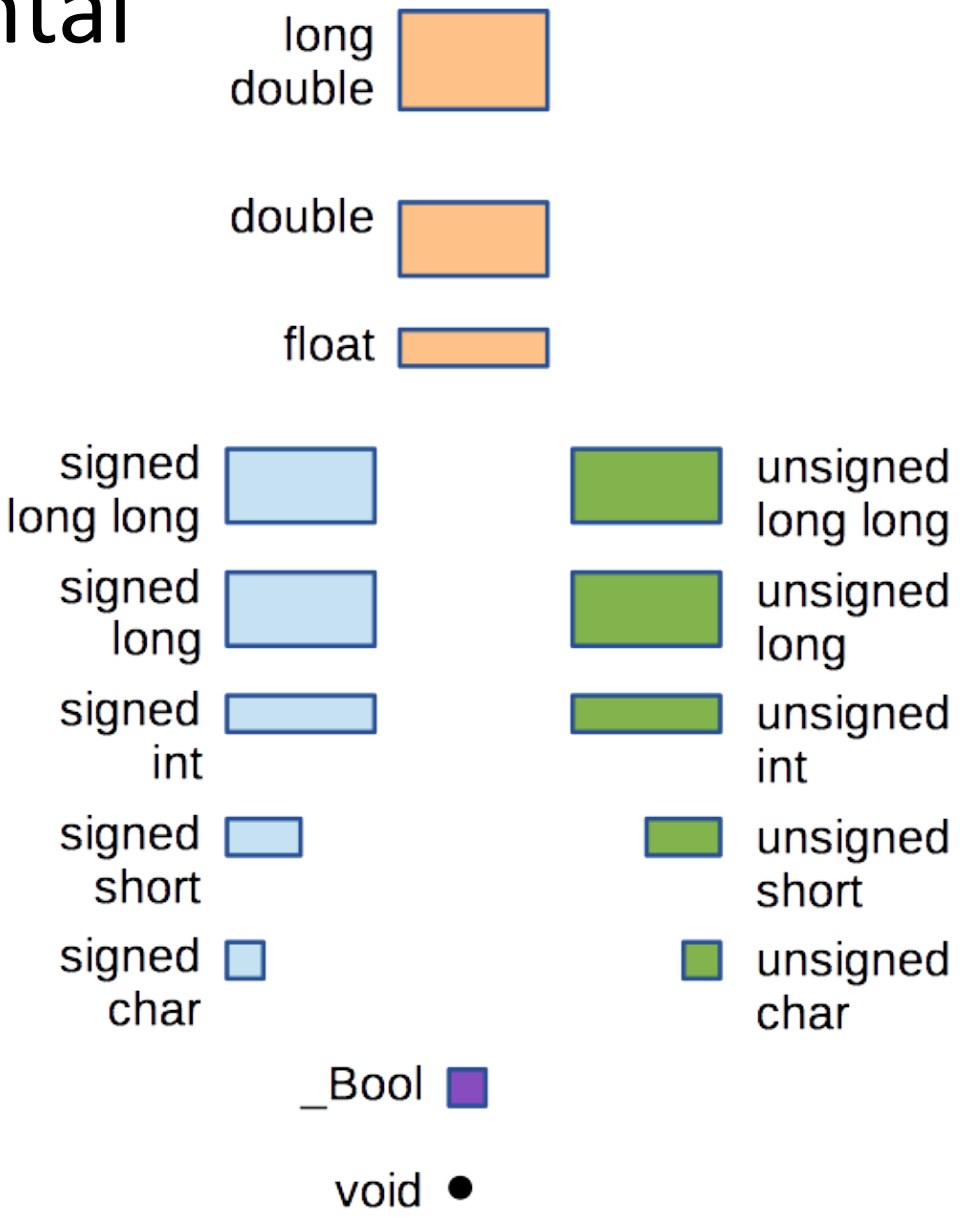
NC State Department of Computer Science

Topics for Today

- Basic Types
- Variables and Scope
- Characters and Strings
- Literal values
- Representation and Overflow / Underflow.

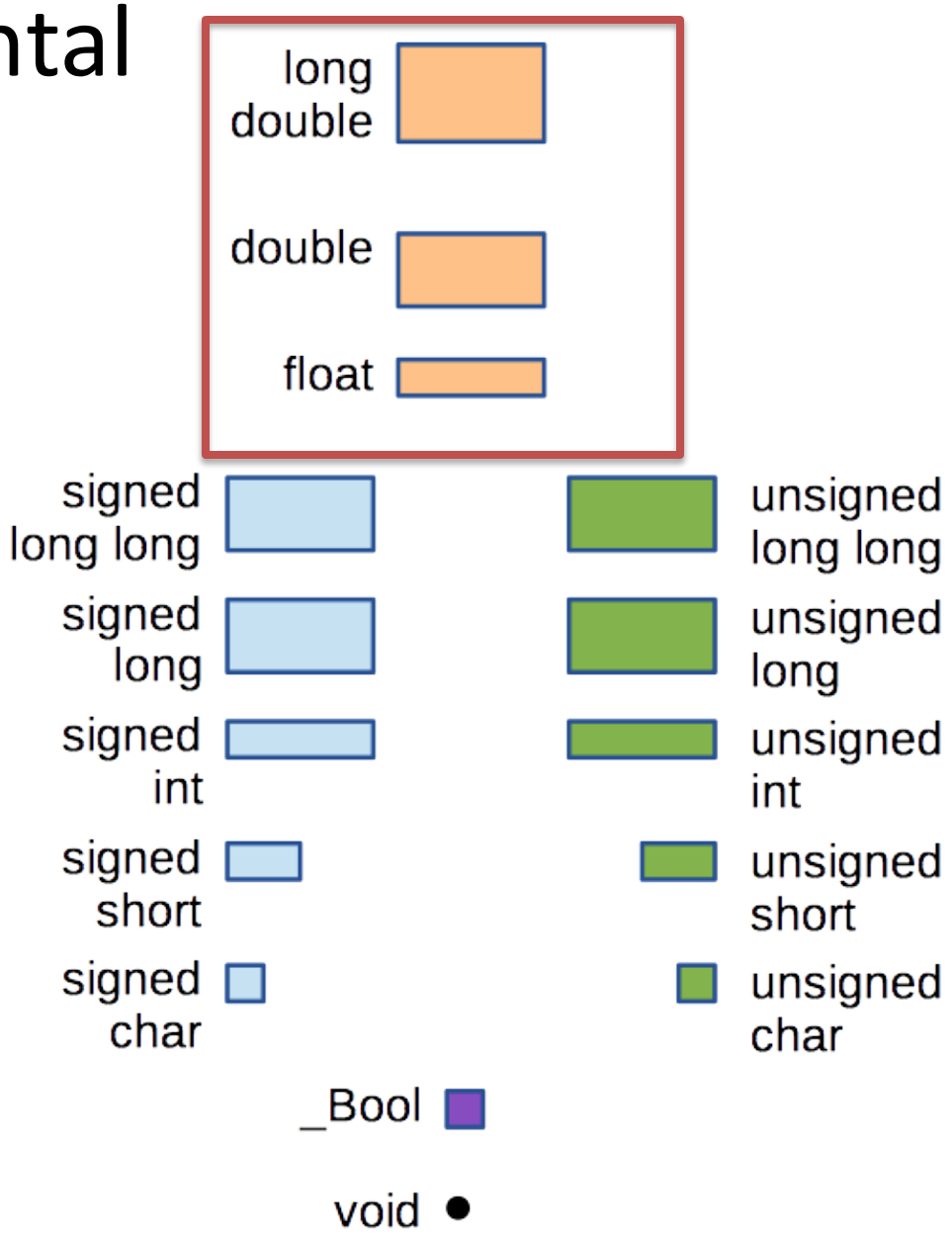
All Fundamental Types

- This figure shows all the basic types we have.
- … a few more than we have in Java.



All Fundamental Types

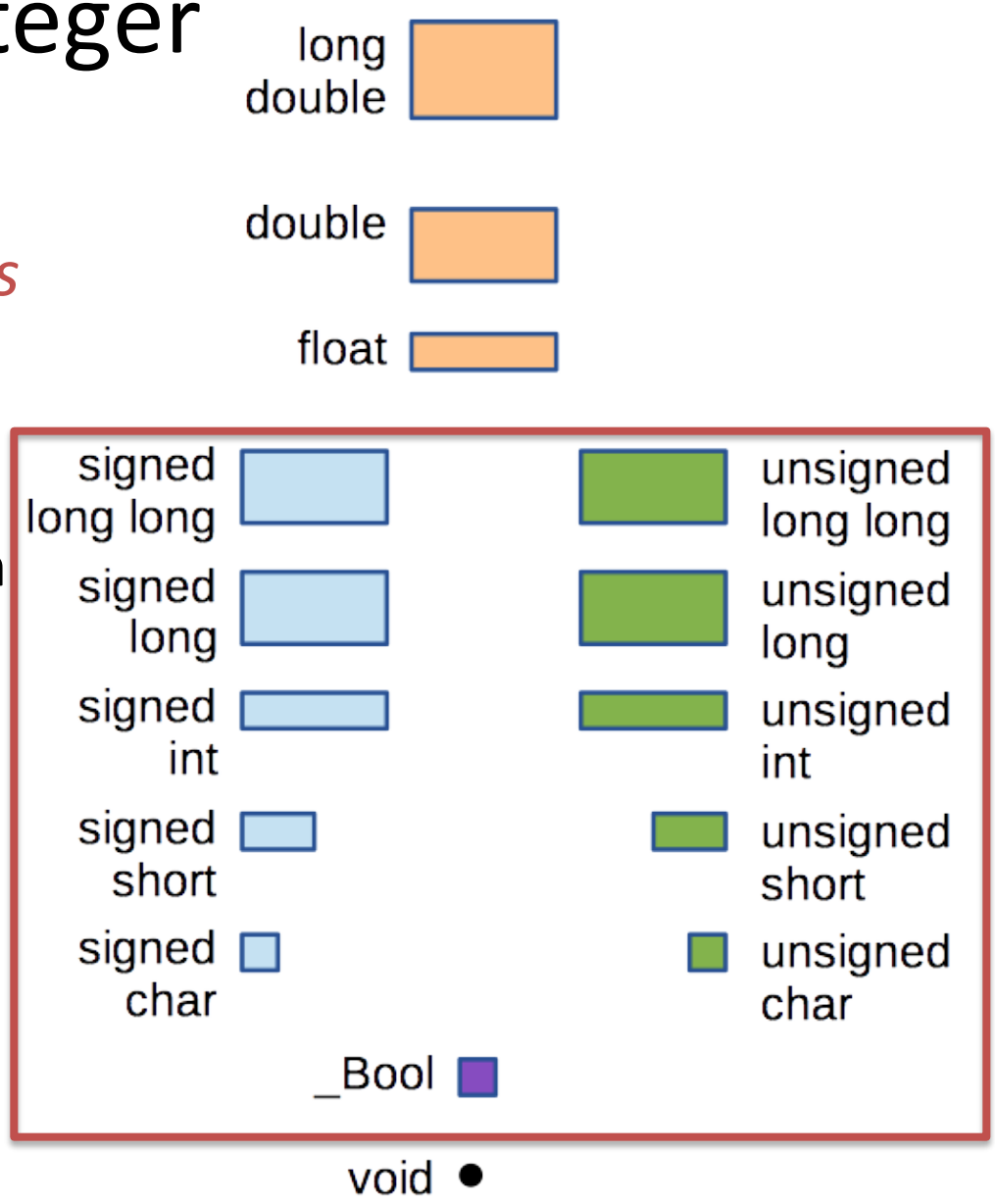
- You have types for real (floating point) values.
  - Including long double, which we don't have in Java



Specialized Integer Types

- You have *specializations* of the integer types.
- Different sizes
  - Including long long, which we don't have in Java
- Integer types can have different signed-ness
  - Mostly, signed is the default.
  - Except for char

Its signed-ness is platform dependent.



What's in a Name

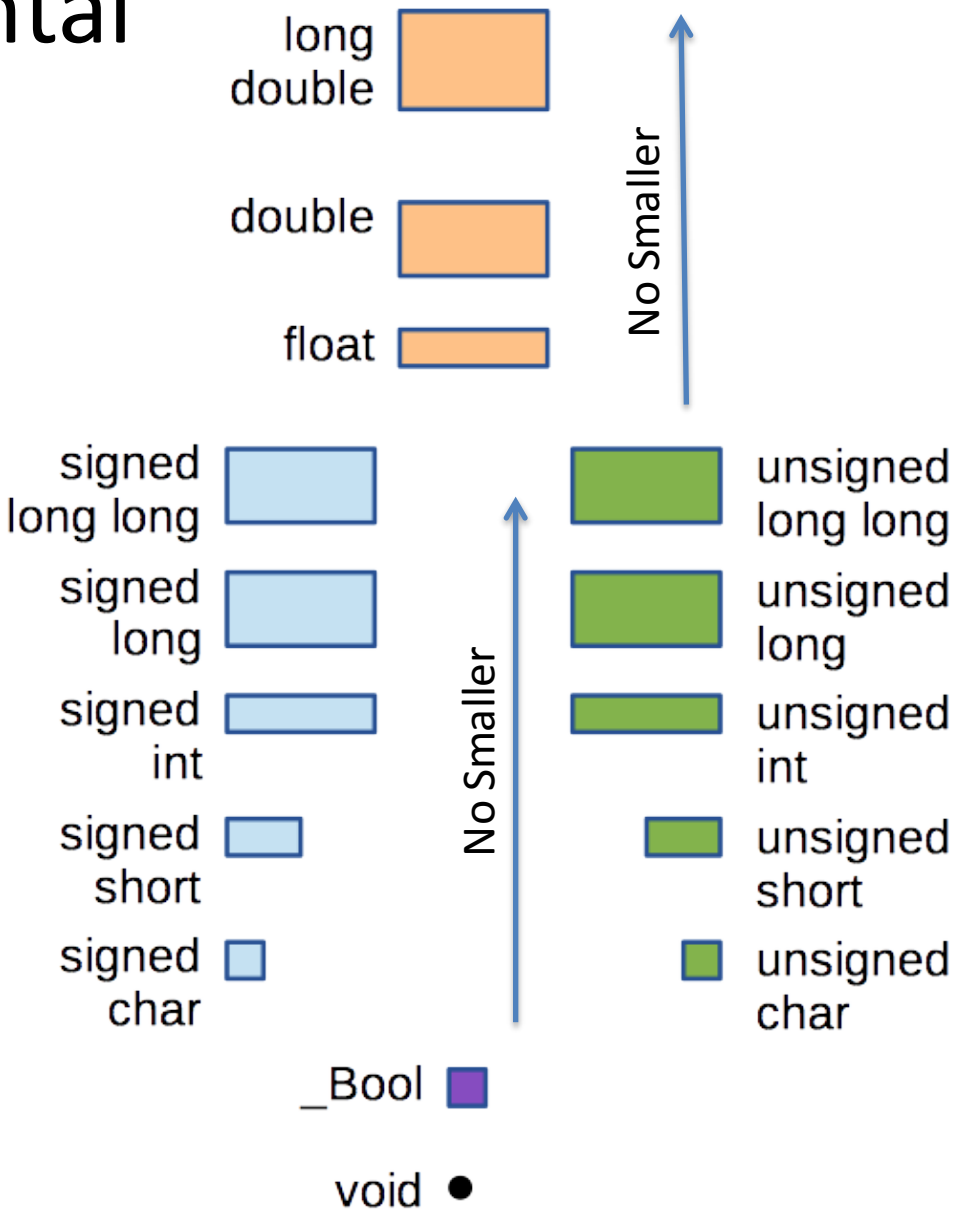
- So, there are lots of different ways to name the same integer type.
- Signed is usually the default, so
  - int is the same as signed int
  - long is the same as signed long
  - short is the same as signed short
  - …

What's in a Name

- The word int is optional for short, long and long long, so:
  - long int is the same as long
  - long long int is the same as long long
  - short int is the same as short
  - …
- Combining these:
  - signed long int is the same as long
  - unsigned short int is the same as unsigned short
  - …

All Fundamental Types

- The standard makes few guarantees about the sizes of these types.
  - For example, long may or may not have more capacity than int.
  - … but it must not have less capacity.



Integer Ranges

- The exact range of a type is platform dependent
- We could write code to figure these out for the common platform
- Or, we can just look in the `limits.h` header
  - It's a file: `/usr/include/limits.h`
- With constant names:
  - SHRT_MIN, SHRT_MAX, USHRT_MAX
  - INT_MIN, INT_MAX, UINT_MAX
  - LONG_MIN, LONG_MAX, ULONG_MAX

Signed range

Unsigned range

Integer Ranges on the Common Platform

Type	Bits	Minimum	Maximum
signed char	8	-128	127
unsigned char	8	0	255
signed short	16	-32,768	32,767
unsigned short	16	0	65,535
signed int	32	-2,147,483,648	2,147,483,647
unsigned int	32	0	4,294,967,295
signed long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long	64	0	18,446,744,073,709,551,615
signed long long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	64	0	18,446,744,073,709,551,615

Integer Ranges on the Common Platform

Type	Bits	Minimum	Maximum
signed char	8	-128	127
unsigned char	8	0	255
signed short	16	-32,768	32,767
unsigned short	16	0	65,535
signed int	32	-2,147,483,648	2,147,483,647
unsigned int	32	0	4,294,967,295
signed long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long	64	0	18,446,744,073,709,551,615
signed long long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	64	0	18,446,744,073,709,551,615

See, no more capacity than long, but no less.

Integer Ranges on the Common Platform

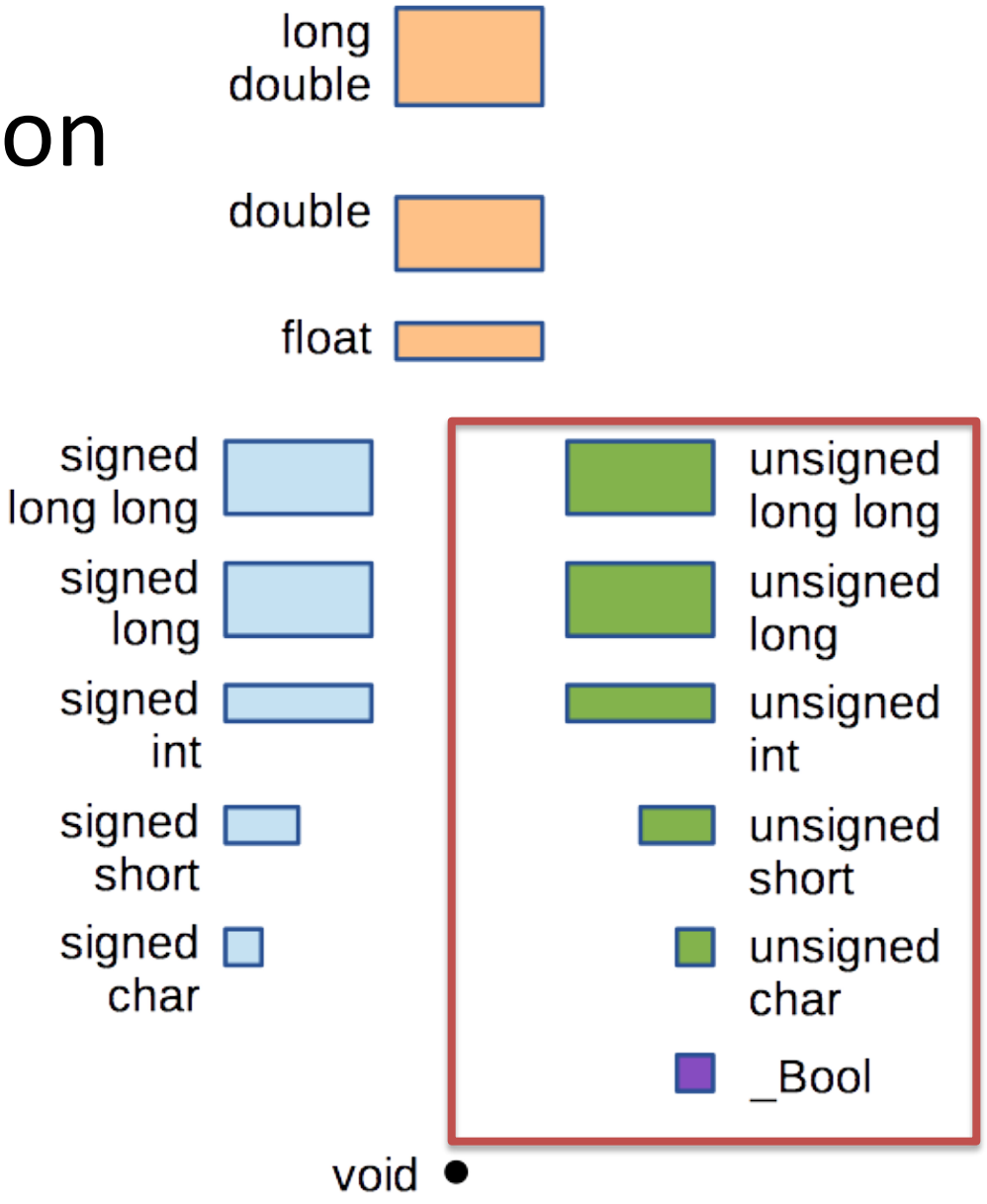
Type	Bits	Minimum	Maximum
signed ch	8	-128	127
u		0	255
signed	16	-32,768	32,767
unsigned short	16	0	65,535
signed int	32	-2,147,483,648	2,147,483,647
unsigned int	32	0	4,294,967,295
signed long		-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsig		0	18,446,744,073,709,551,615
sign		-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigne	4	0	18,446,744,073,709,551,615

Always zero

Typically, one extra negative value.

Type Representation

- The standard makes few guarantees about how these types should be represented.
- It does for unsigned types
- And, in practice, all systems represent integer types the same way.



Making Variables

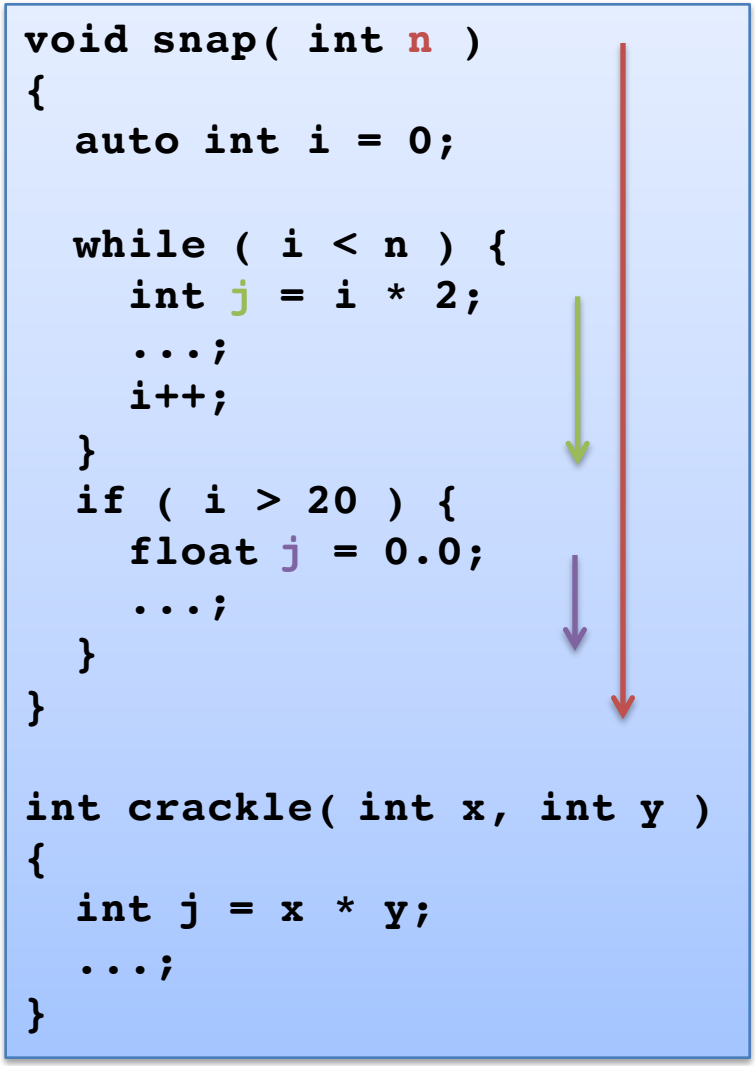
- A *variable* has:
  - A name : that's a legal identifier name
  - A type : that's how the compiler interprets the contents of memory representing the variable
  - A scope : that's the section of code that can access the variable
  - A storage class : that's how it's stored and initialized

Static vs Dynamic Types

- In C (and Java), all variables are *statically typed*
  - The type of a variable type can't change as the program runs
- It wouldn't have to be this way
- Lots of languages have *dynamic typing* (e.g., JavaScript, PHP, Python)
- It's a tradeoff in:
  - flexibility
  - performance
  - ability to detect errors at compile time

Auto (Local) Variable Scope

- Declared inside a function
  - … including function parameters.
- In scope from the declaration to the end of the surrounding block
- These are called *auto* variables
  - Their storage class is called *auto*
  - In fact, auto is a keyword in C



Global Variable Scope

- A variable defined outside any function has *global scope*
- Its lifetime is the whole execution of the program
  - Their storage class is called *static*
- Any code can access this variable.
- There's a potential for name collision

```
int x = 25;

int main(void)
{
    … x …
}

int length = 15;

int f(…)
{
    … x … length …
}
```

Shadowing

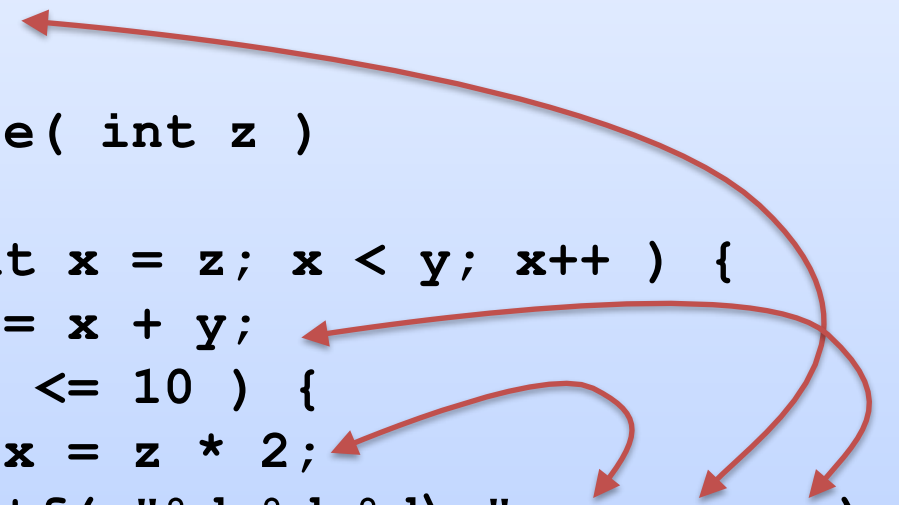
- Two variables can have the same name … as long as they are declared in different scopes.
  - They are independent variables, just with the same name.
- How does the compiler decide which one you want?
  - You get the one in the one declared in the narrowest surrounding scope.
- We say the one in the narrower scope shadows the one in the wider scope

Shadowing Example

```
int x = 6;
int y = 8;

void waffle( int z )
{
  for ( int x = z; x < y; x++ ) {
    int z = x + y;
    if ( z <= 10 ) {
      int x = z * 2;
      printf( "%d %d %d\n", x, y, z );
    }
  }

  int y = x - 1;
  printf( "%d %d %d\n", x, y, z );
}
```

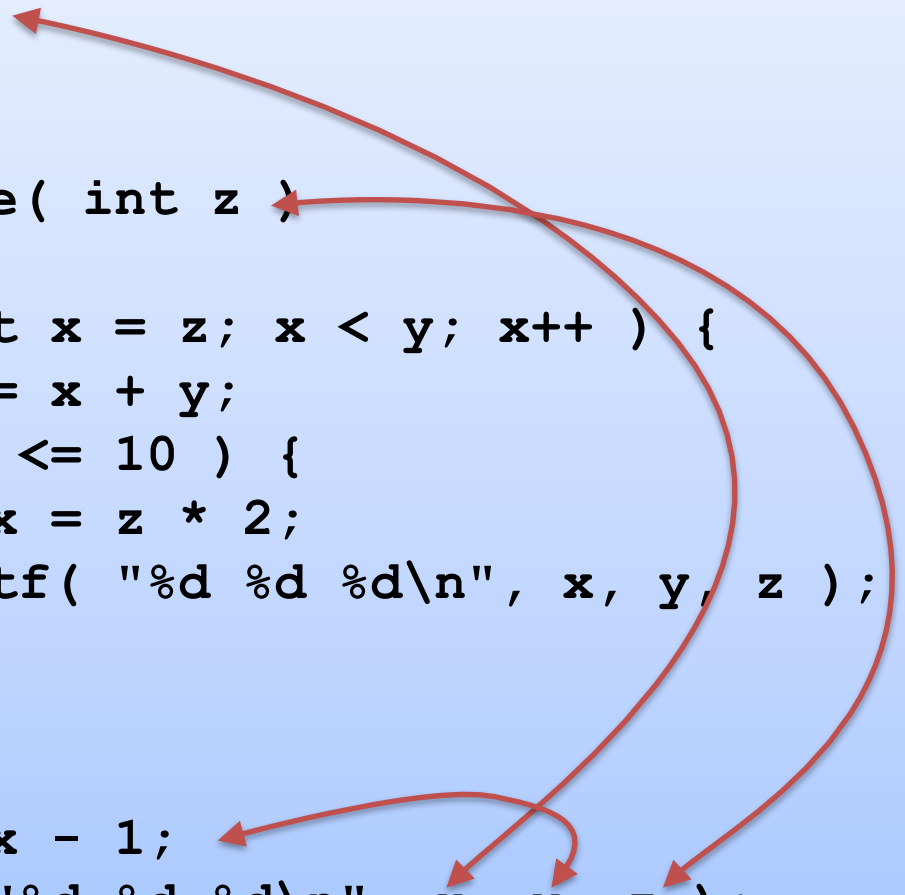
Shadowing Example

```
int x = 6;
int y = 8;

void waffle( int z )
{
  for ( int x = z; x < y; x++ ) {
    int z = x + y;
    if ( z <= 10 ) {
      int x = z * 2;
      printf( "%d %d %d\n", x, y, z );
    }
  }

  int y = x - 1;
  printf( "%d %d %d\n", x, y, z );
}
```



Shadowing Example

```
int x = 6;
int y = 8;

void waffle( int z )
{
  for ( int x = z; x < y; x++ ) {
    int z = x + y;
    if ( z <= 10 ) {
      int x = z * 2;
      printf( "%d %d %d\n", x, y, z );
    }
  }

  int y = x - 1;
  printf( "%d %d %d\n", x, y, z );
}
```



Shadowing

- The ability to shadow a variable is common to lots of languages
  - In Java, a local variable can shadow a field.
- This is a clever trick … that you should probably avoid using too cleverly.
- There's at least one place where it's really nice to have.
  - In macro definitions … later.

About _Bool

- Originally, C had no specific boolean type
- We just used an integer type
  - With 0 being interpreted as false
  - And anything else being true.
- This still works just fine.

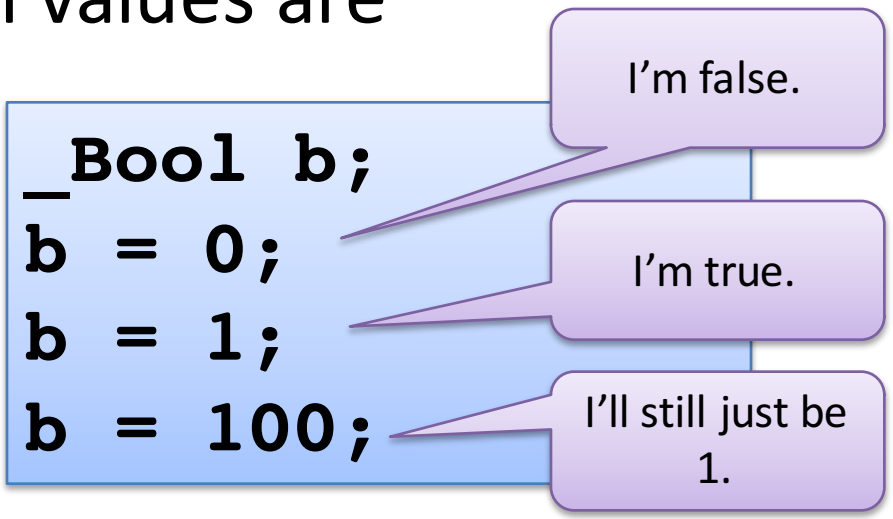
```
int i = 100;
while ( i ) {
  i -= 1;
}
```

```
double f = 10;
while ( f ) {
  f -= 0.1;
}
```

It even works with floating-point types … but this is probably a bad idea.

About _Bool

- In C99, they added a type for boolean values.
- They had to give it a goofy name, **`_Bool`**, a name nobody would have been using for anything.
- Still, internally, boolean values are
  - 0 for false
  - Anything else for true
  - With 1 as the typical value for true.



Meet stdbool.h

- _Bool works, but it looks like a strange type name.
- You can include stdbool.h
  - You get the name the name **`bool`** as an alias for _Bool
  - But wait, there's more. You also get
  - **`false`** : a preprocessor constant for 0
  - **`true`** : a preprocessor constant for 1
- So, you can code as if bool is a built-in type

Character Representation

- A char variable a small number
  - Just what you can fit in one byte.
  - (the standard doesn't absolutely guarantee that it's a byte … but, it's a byte).
- How does it store a symbol?
  - Simple, we just use a numeric code for each symbol
- That' what **ASCII** is
  - the **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
  - It's a rule giving the code for common, western characters.
- The C standard doesn't require ASCII
  - But that's what you'll get on the common platform.

ASCII

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

- Don't memorize this table, but remember a few things:
  - Letters and digits are consecutive (good)
  - Capital and lower case letters are in different parts of the table (with a little gap in between)
  - Not every code represents a visible character.

ASCII

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

- More things to notice
  - Only half the codes are used, the rest are application/platform dependent.
  - The symbol ‘0’ isn’t represented by the code zero … that code has a special job in C.

Character Literals

- You could memorize the ASCII character table … but why bother.
- Need the code for a character like **z**?
  - That's what **'z'** means.
- The code for a character is just that character inside single quotes.
- How about codes for characters you can't type?

Special Characters

- Some character must be entered with *escape sequences:*

Escape Sequence	Character
\0	A null (more later)
\’	Single quote
\”	Double quote
\\	Backslash
\n	Newline
\t	Horizontal tab
\nnn	Any code you want, in octal
\xnn	Any code you want, in hexadecimal

Meet Hexdump

- Hexdump is a shell command that can show you the exact sequence of bytes in a file.
  - It has options to show the contents in various formats.

```
$ hexdump -C someFile.txt
```

Show numeric codes and characters.

File you want to see.

Text Files

- In a file, text is just a sequence of codes:

h_good.c

```
#include <stdio.h>

int main()
{
  printf( "Hello World\n" );
  return 0;
}
```

```
$ hexdump -C h_good.c
00000000  23 69 6e 63 6c 75 64 65  20 3c 73 74 64 69 6f 2e  |#include <stdio.|
00000010  68 3e 0a 0a 69 6e 74 20  6d 61 69 6e 28 29 0a 7b  |h>..int main().{|
00000020  0a 20 20 70 72 69 6e 74  66 28 20 22 48 65 6c 6c  |.  printf( "Hell|
00000030  6f 20 57 6f 72 6c 64 5c  6e 22 20 29 3b 0a 20 20  |o World\n" );.  |
00000040  72 65 74 75 72 6e 20 30  3b 0a 7d 0a              |return 0;.}.|
```

Code for #, then i …

There's an end-of-line.

There's the end-of-line for the last line.

See, no end-of-file character.

Text Files

- Looking at these codes can show what's really there.

h_bad.c

```
#include <stdio.h>

int main()
{
    printf( "Hello World\n" );
    return 0;
}
```

There's a CR LF.

and there's a hard tab.

```
$ hexdump -C h_bad.c
00000000  23 69 6e 63 6c 75 64 65  20 3c 73 74 64 69 6f 2e  |#include <stdio.|
00000010  68 3e 0d 0a 0d 0a 69 6e  74 20 6d 61 69 6e 28 29  |h>....int main()|
00000020  0d 0a 7b 0d 0a 09 70 72  69 6e 74 66 28 20 22 48  |..{...printf( "H|
00000030  65 6c 6c 6f 20 57 6f 72  6c 64 5c 6e 22 20 29 3b  |ello World\n" );|
00000040  0d 0a 09 72 65 74 75 72  6e 20 30 3b 0d 0a 7d 0d  |...return 0;..}.|
00000050  0a                                                |.|
```

Wide Characters

- One-byte codes aren't sufficient for all character sets.
- C supports a wide character type, **wchar_t**
- In a file, a format like UTF-8 supports unicode
  - Using 1-byte codes for ASCII characters
  - … and multi-byte codes for other characters.

```
$ hexdump -C uni.txt
00000000   61 62 63 ce 91 ce 92 ce  93 0a                    |abc.......|
```

One-byte codes for a, b, c

Two-byte codes for Greek A, B, Γ

String Literals

- A string literal is a sequence of characters surrounded by double quotes: **“I’m A String!”**
- Internally stored as:
  - A sequence (an array) of character codes
  - With a null character at the end
  - So, “abc” really takes four bytes to store
  - More about this later
- Strings can’t contain line breaks
  - But adjacent literals are implicitly concatenated:
    `“I” “ am just ” “one long” “ string”`

Integer Values

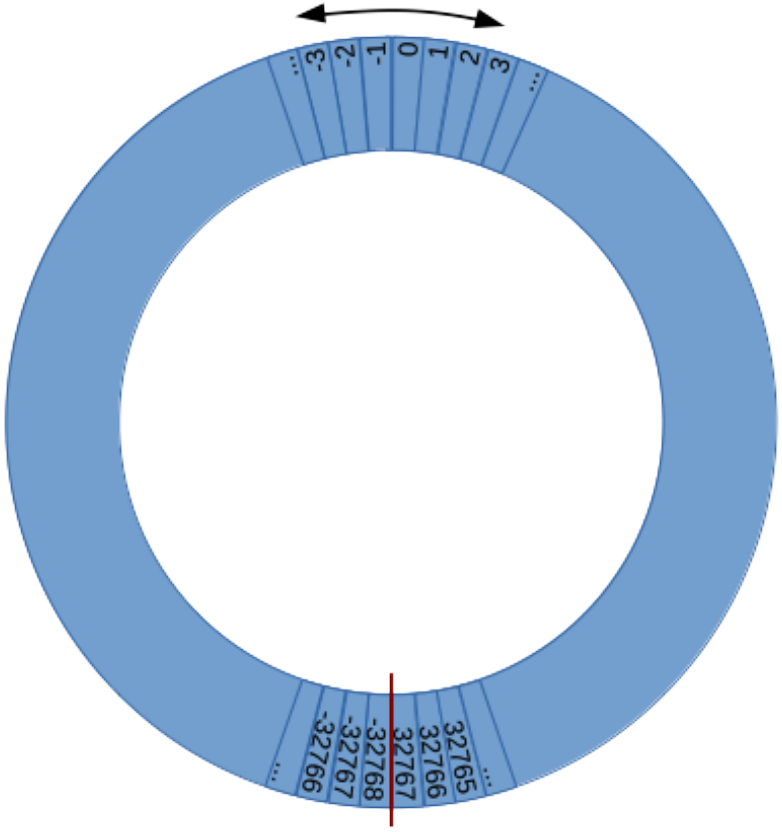
- If your program says an integer, like: **75**
- … the compiler assumes you want the **int** type
  - Unless the value is too large to fit in an int:
  - Then, it assumes you want the smallest integer type that's big enough:
    signed int → unsigned int → signed long → …

Integer Values

- You can tell the compiler exactly what you want
  - For unsigned, put a `U` at the end (e.g., 351**U**)
  - For long, put an `L` at the end (e.g., 351**L**)
  - For long long, put LL at the end (e.g., 351**LL**)
  - You can combine these, in any order (e.g., 351**ULL**)
- Numbers starting with **0** are implicitly in **octal**
  - So, **0234** really means 156 … more about this later.
- Numbers starting with **0x** are implicitly in **hexadecimal**
  - So, **0x234** really means 564 … more later.

Overflow

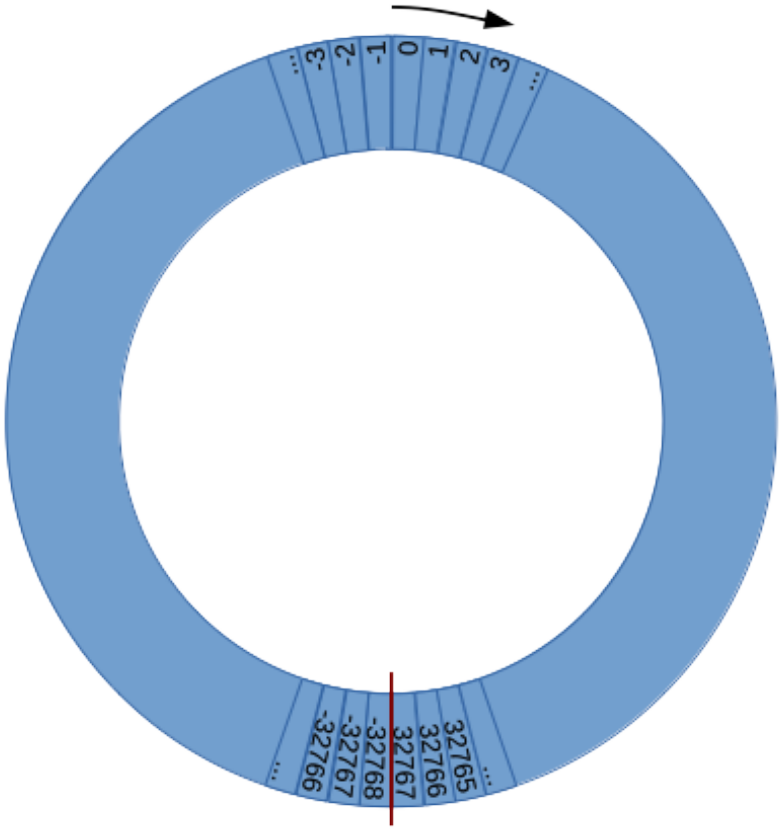
- Integer types have limited capacity
- What happens if you exceed that capacity?
  - Depends on how values are represented
  - The C standard doesn't require a particular **signed** representation
  - … but modern computer systems all use two's complement
- Think of values forming a circle rather than a line



Overflow

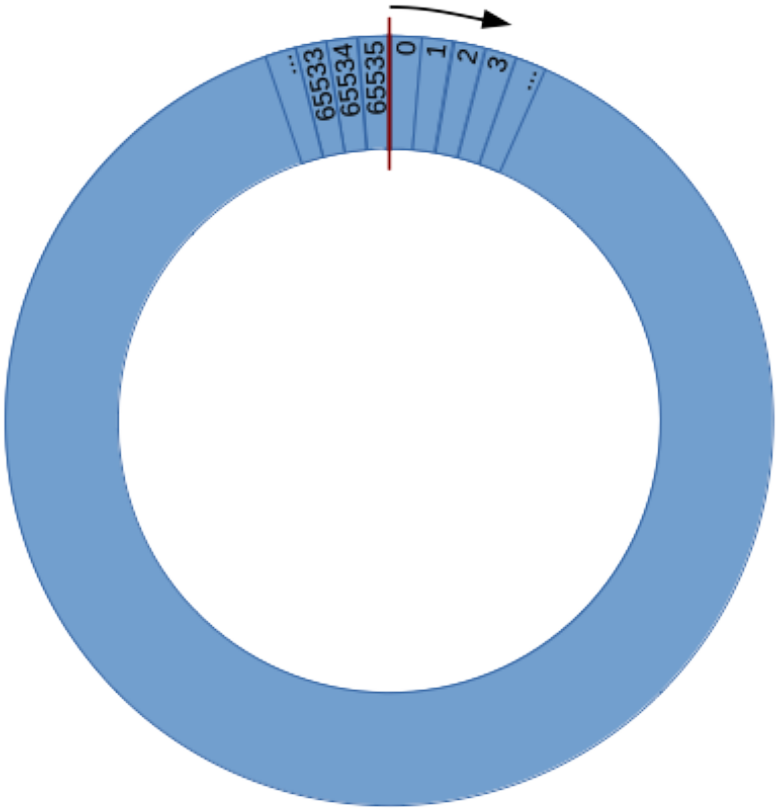
```
// So, what will happen?
short x = 0;
while ( true ) {
  x = x + 1;
  printf( "%d\n", x );
}
```

- Officially, behavior is implementation defined
- … but, in practice, you will get wrap-around



Overflow

- Unsigned numbers work the same way
- The wrap-around just occurs in a different place.
- And, the standard guarantees wrap-around behavior here.




```
unsigned short x = 0;
while ( true ) {
  x = x + 1;
  printf( "%u\n", x );
}
```

Overflow Happens

- Overflow can happen when you …
  - Compute a value that's outside the range of the type that's holding it.

```
int x = 65536;
x = x * x;
```

Ouch

  - Copy from a wider type to a narrower one.

```
int x = 123456789;
short y = x;
```

Ouch

Overflow Happens

- Overflow can happen when you …
  - Copy between signed and unsigned types of the same width.
  - From signed to unsigned:

```
short x = -1;
unsigned short y = x;
```

Ouch

  - Or, the other way around:

```
unsigned short x = 40000;
short y = x;
```

Ouch

Overflow Happens

- Even just changing the sign of a value can cause overflow … in one, special case.

```
int x = -2147483648;
x = -x;
```

I just barely fit in an int.

But not me.

Real-Valued Types

- Floating point numbers can be
  - Regular precision: `float`
  - Double precision: `double`
  - Extended double: `long double`
- Examples:
  - float x;
  - double y;
  - long double z;

We don't have unsigned versions of these types.

Floating Point Numbers

- Floating point representation is platform specific
- Typical representation:

$+7.03 * 10^{36}$

A sign for the mantissa.

A magnitude for the mantissa (normalized)

An exponent

- All these values packed into one block of memory

Floating Point Numbers

- Capacity of the exponent determines range of values
- Capacity of the mantissa determines the precision
- And, these values are typically **binary**
  - So, internally, it's more like:

$$+1.01101 * 2^{10100}$$

- Fortunately, we can usually write our code without having to think about this … mostly.

Floating Point Representation

- Floating point types can't represent every value exactly.
  - Values like 8.5 or 5.25, no problem.
  - But values like 8.1 or 5.3, we have to approximate.
- So, code like the following may not behave the way you want.

```
for ( double x = 0; x != 1; x += 0.1 ) {
   …;
}
```

This will be an infinite loop.

```
for ( double x = 0; x < 1; x += 0.1 ) {
   …;
}
```

Even this will loop 11 iterations.

Floating Point Numbers

- There's an IEEE standard for floating point representation
  - C doesn't require compilers to follow it, but many of them do.
- IEEE single precision:
  - 1 bit for the sign
  - 23 (24) bits for the mantissa
  - 8 bits for the exponent
- IEEE double precision:
  - 1 bit for the sign
  - 52 (53) bits for the mantissa
  - 11 bits for the exponent

$+1.01101 * 2^{10100}$

Normalization gives us one bit for free.

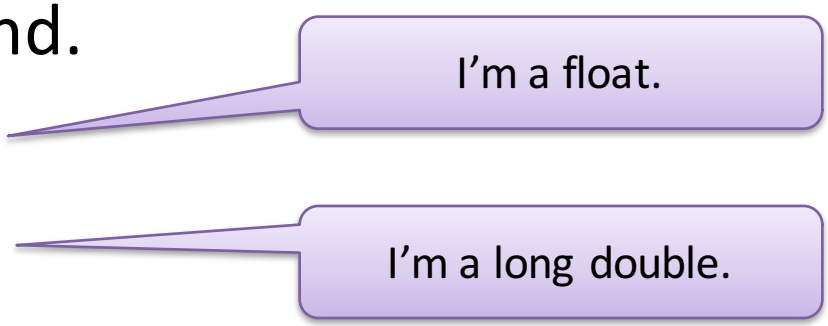
Floating Point Range

- long double on the common platform
  - 80-bit format (stored in 16 bytes)
  - 64 bits for the mantissa
  - 15 for the exponent
- So, on the common platform

Type	Magnitude range	Precision
float	About 10^{-38} .. 10^{38}	About 7 decimal digits
double	About 10^{-308} .. 10^{308}	About 15 decimal digits
long double	About 10^{-4930}.. 10^{4930}	About 19 decimal digits

Literally

- You can give literal floating point values.
  - Like 3.14
  - Or, in scientific notation like: 6.022E23
- The compiler will assume it's a double
  - Unless you say otherwise, via an extra character at the end.
  - 3.14F
  - 3.14L



Floating-Point Overflow

- Floating point values have a wide range … but they can still overflow.

```
double z = 1.0;
while ( true ) {
    printf( "%f\n", z);
    z = z * 2;
}
```

Eventually, we're going to hit the limit of this representation.

- And, there's another floating point behavior to keep in mind …

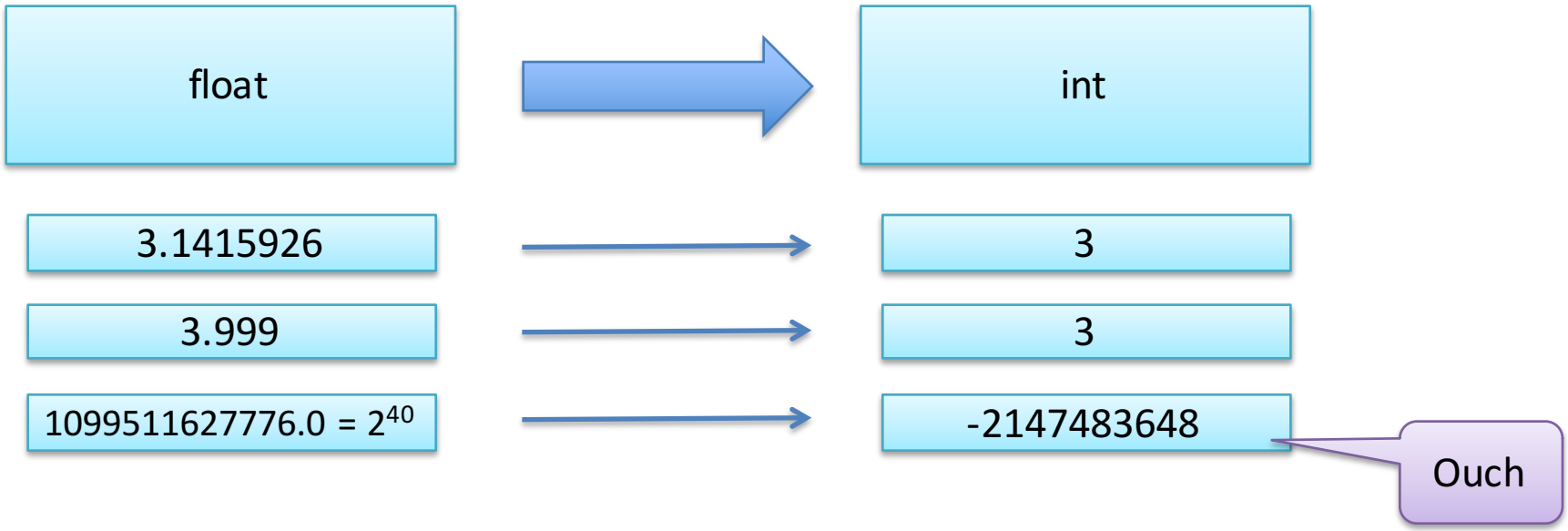
Underflow

- Floating point numbers can represent magnitudes only so small
- double and long double have extra capacity, but, still, there's a limit
- Falling below this low-magnitude limit is called *underflow*

```
double z = 1.0;
while ( z != 0.0 ) {
    printf( "%f\n", z);
    z = z * 0.5;
}
```

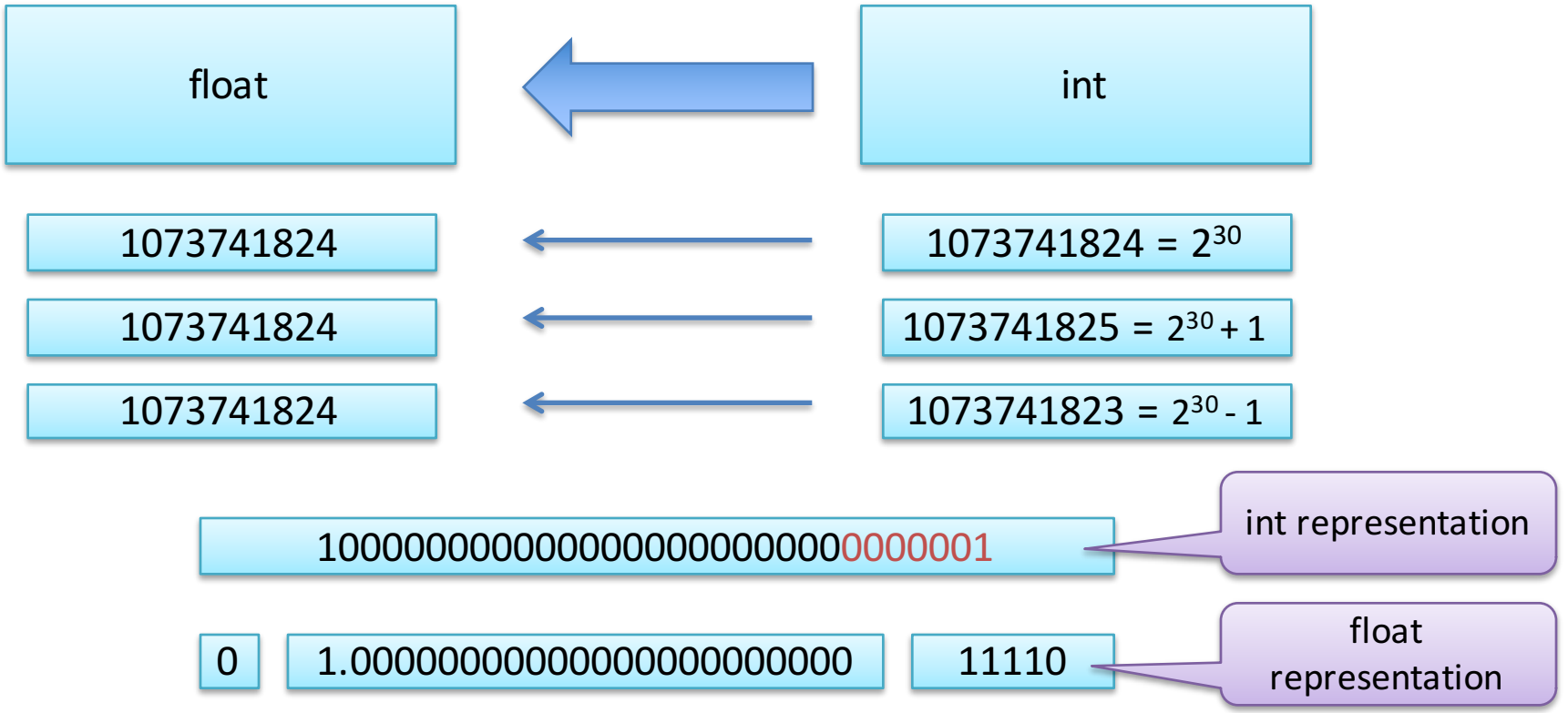
Floating-Point / Integer Conversion

- C has simple rules for converting from real to integer types.
  - Truncate fractional part.
  - The integer gets the remaining, whole number part.
  - There could definitely be some overflow here.



Floating-Point / Integer Conversion

- Integer to Floating Point
  - Approximate with the closest floating point value



Floating-Point / Integer Conversion

- In general, floating point types can exactly represent any small integer values exactly …
- … but not all really big values.
- On the common platform there are potential rounding errors: