

Type Conversions

ITCS 2116: C Programming
College of Computing and Informatics
Department of Computer Science

Outline

- Type Conversions
 - Explicit
 - Overflow and Underflow
 - Implicit
- More Input/Output (I/O) in C
 - **scanf** and conversions

Type Conversions

Type Conversions

- Data type conversions occur in two ways
 - **Explicitly**: the **programmer** deliberately *casts* from one type to another)
 - **Implicitly**: variables of different types are combined in a single expression, the **compiler** casts from one type to another)

```
unsigned char a;  
int b;  
float c;  
double d;  
  
...  
c = (float) b;  
d = a + (b * c);
```

Casting: Explicit Conversion

- **Forces** a type conversion in the way specified
- Syntax: **(typename) expression**
- Example: `d = (double) c;`

This code will attempt to convert the value stored in **c** into a double and store it in **d**

- Question to consider: Can the programmer get **better** quality results by explicitly casting?

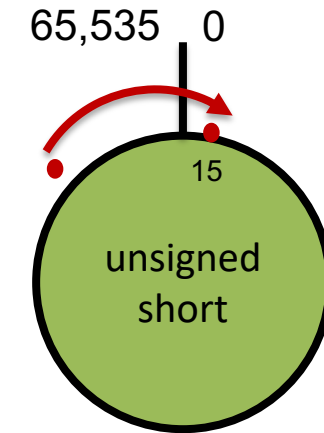
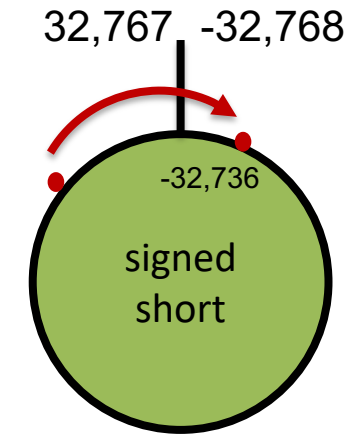
Casting: Explicit Conversion

- A special case: **(void) expression;**
 - Means that the value stored in **expression** must not be used in any way
 - Question to consider: how could that possibly be useful?

Overflow and Underflow

- Recall that numeric data types can only represent values within a given range
 - For example, in 32-bit computers a **signed short int** variable can store numbers between -32,768 and 32,767, whereas an **unsigned short int** variable in the same computer can store numbers between 0 and 65,535
- We need to take this into account when converting data from one data type to another

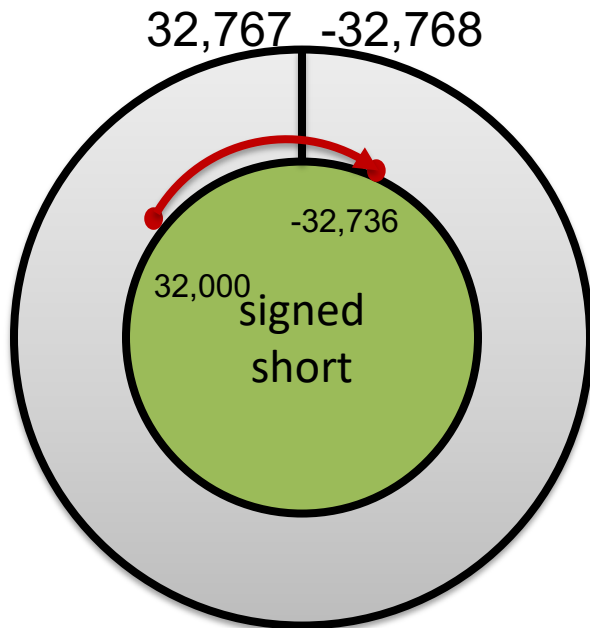
(see `number_ranges.c` and `overflow_underflow.c` in *Code samples and Demonstrations in Canvas*)



Overflow and Underflow (cont'd)

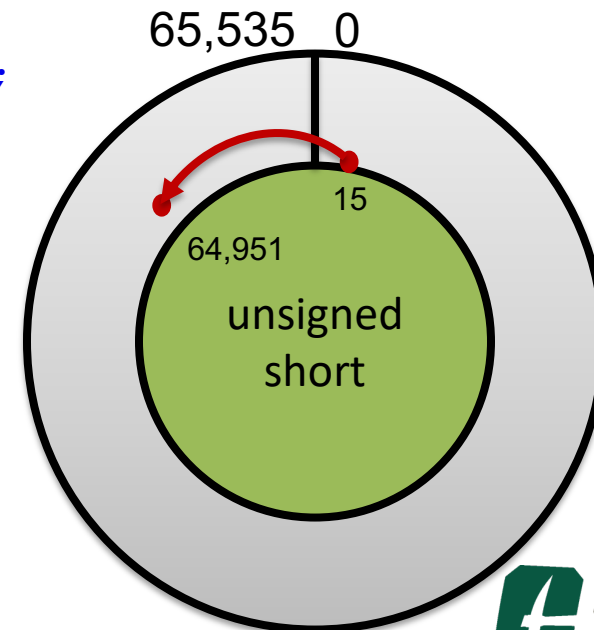
- Think of number ranges as a circle rather than a line
 - Example: **signed** and **unsigned short**
 - Shorts hold 16 bits on most machines
 - Signed Range: $-\left(\frac{2^{16}}{2}\right)$ to $\left(\left(\frac{2^{16}}{2}\right) - 1\right)$ or [-32,768 to 32,767]
 - Unsigned Range: 0 to $(2^{16} - 1)$ [0 to 65,535]

(see [number_ranges.c](#) and [overflow_underflow.c](#) in *Code samples and Demonstrations in Canvas*)



```
//overflow  
signed short x = 32000;  
x += 800;  
printf("%d\n", x);
```

```
//underflow  
unsigned short y = 15;  
y -= 600;  
printf("%d\n", y);
```



Converting **signed** to **unsigned**

- This only makes sense if you are *sure* the value stored in the **signed** operand is **positive**
- If **signed** is the shorter operand, extend it

```
int a;  
unsigned b;  
a = -36;  
b = (unsigned) a;  
a = (int) b;
```

```
Result when output:  
b = 4294967260  
a = -36
```

```
int a;  
unsigned short b;  
a = -36;  
b = (unsigned short) a;  
a = (int) b;
```

```
Result when output:  
b = 65500  
a = 65500
```

What happened???

Converting

- If **signed** number is negative, go counter-clockwise
 - Starting at 0
 - Counting 0

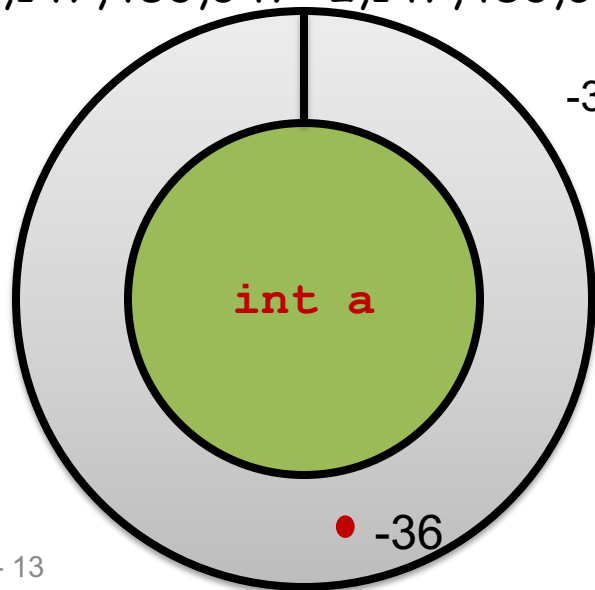
```
int a;  
unsigned short b;  
a = -36;  
b = (unsigned short) a;  
a = (int) b;
```

Result when output:

```
b = 65500  
a = 65500
```

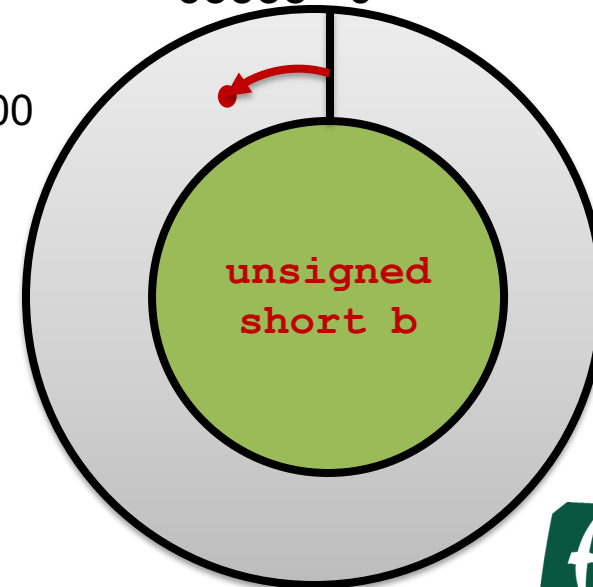
(see
[signed_to_unsigned.c](#)
in Code samples and
Demonstrations in Canvas)

2,147,483,647 -2,147,483,648



-36 -> 65,536 - 36 = 65,500

65535 0



Converting unsigned to signed

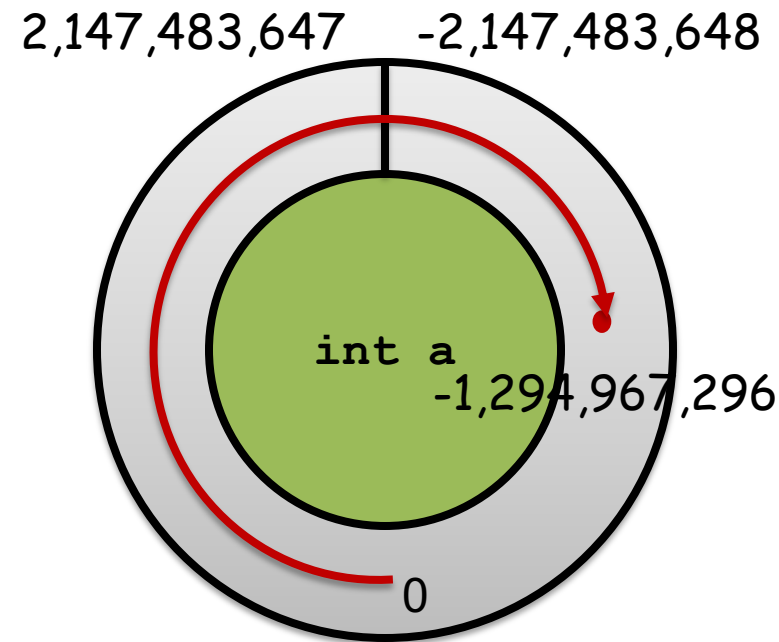
- If **signed** is large enough to store the correct value, no problems
 - otherwise, will definitely be an error (**overflow**)!

(see [unsigned_to_signed.c](#) in *Code samples and Demonstrations in Canvas*)

```
int a;  
unsigned int b;
```

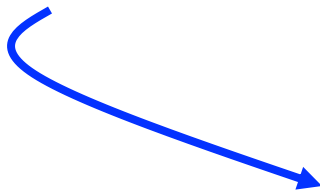
```
b = 3000000000;  
a = (int) b;
```

Result when output:
b = 3000000000
a = -1294967296



Converting Floating Point to Integer

- Round towards zero (“truncate”) to get the integer part, and discard the fractional part
 - $+3.999 \rightarrow 3$
 - $-3.999 \rightarrow -3$
 - Obviously some **loss of precision** can occur here
- **Overflow** if the integer variable is too small



```
float f = 1.0e10;  
int i;  
i = f;
```

```
Result when output:  
f = 10000000000.0  
i = -2147483648
```

Floating to Integer... (cont'd)

Example

```
int i;  
float g;  
g = 123456780000000012345678.0f;  
i = (int) g;
```

?????



Result:

```
i = -2147483648  
g = 123456780268340198244352.00000
```

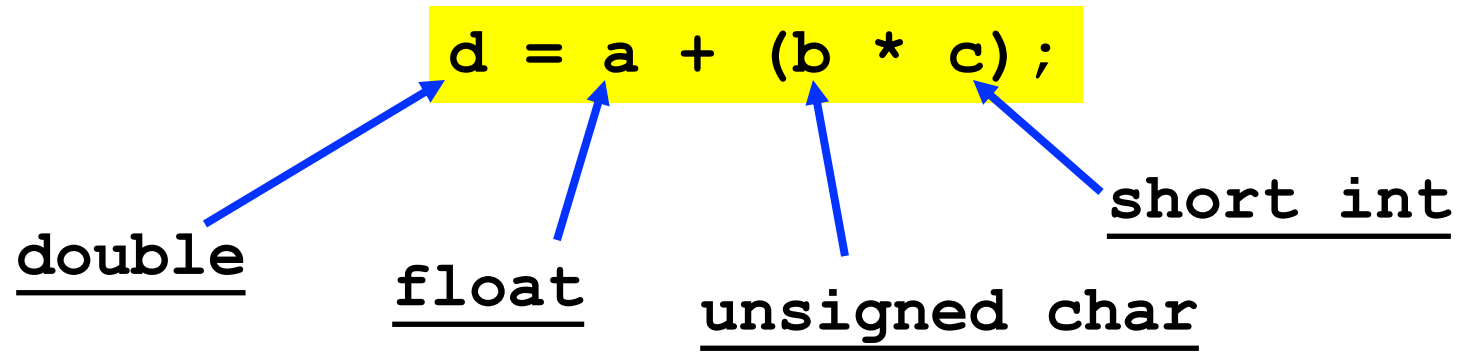
(see `float_to_int.c` in
Code samples and
Demonstrations in Canvas)

Converting to Floating

- Integer → Floating
 - If value cannot be represented exactly in floating point, convert to the **closest** value (either higher or lower) that can be represented in floating point
- Double precision → Single precision
 - If value cannot be represented exactly, convert to **closest** value (either higher or lower)
 - Can **overflow or underflow**

Implicit Conversions

- For “mixed type” expressions, e.g.,



- The compiler does “**the usual arithmetic conversions**” before evaluating the expression
- char** and **short** are always converted to **int** (or **unsigned int**) before evaluating expressions

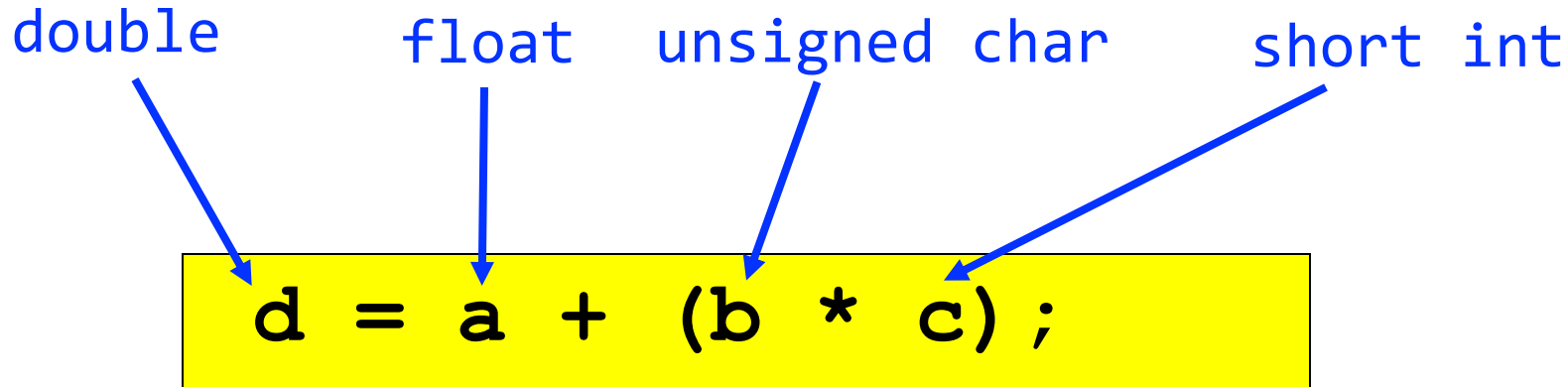
The “Usual Conversions” For Arithmetic Operations

- In a nutshell: when combining values of two numbers...
 - If either is floating point, **convert the other to floating point**, and
 - **Convert less precise** to more precise
- Order is significant in the following table!

The “Usual...” (cont’d)

Rule	If either operand is...	And other operand is...	Then convert other operand to...	
#1	long double	Anything	long double	Else...
#2	double	Anything	double	Else...
#3	float	Anything	float	Else...
#4	unsigned long int	Anything	unsigned long int	Else...
#5	long int	unsigned int	unsigned long int	Else...
#6	long int	Anything else	long int	Else...
#7	unsigned int	Anything	unsigned int	Else...
#8			(both operands have type int, no action needed)	

Example



before evaluating expression:

convert **b** to **unsigned int** and **c** to **int**

before multiplying:

convert **c** to **unsigned int** (rule #7)

before adding:

convert result of multiplying to **float** (rule #3)

when assigning:

convert result of addition to **double** (rule #2)

More on Input/Output

The `scanf()` function

- `getchar()` is a crude way to read input
- `scanf()` is a much more convenient **library function** for formatted input
 - Converts numbers to/from ASCII
 - Skips “white space” automatically
- Def: `int scanf(const char * fmt, ...)`
 - Variable number of arguments
- `fmt` specifies how input must be converted

Examples

```
char c, d;  
float f, g;  
int i, j;  
int result;
```

```
result = scanf("%c %c", &c, &d);
```

...check result to see if returned value 2...

```
result = scanf("%d %f %f", &i, &f, &g);
```

...check result to see if returned value 3...

```
result = scanf("%d", &i);
```

...check result to see if returned value 1...

(see [scanf_examples.c](#) in Code
samples and Demonstrations in Canvas)

Parts of the Format Specifier

1. % (mandatory)
2. Minimum input field width (optional, number of characters to scan)
3. Type of format conversion (mandatory)

NOTE: White space in the format string does not force white space to be present in the input stream.

(see `date.c` in *Code samples and Demonstrations in Canvas*)

Some Types of Conversions

Convert input to Type...	Specifier
char	%c
unsigned int	%u (in decimal) %o (in octal) %x, %X (in hex) (%lu, %lo, %lx for long)
signed int	%d, %i (in decimal) (%ld, %li for long)
float	%f
float	%e, %E (use scientific notation)
(string)	%s

Input Arguments to scanf()

- Must be passed using “call by reference”, so that **scanf ()** can overwrite their value
 - Pass a **pointer** to the argument using **&** operator
- Example:

```
char c;  
int j;  
double num;  
int result;
```

```
result = scanf("%c %d %lf", &c, &j, &num);
```

⚠ *common source of bugs* ⚠

**failure to use &
before arguments
to scanf**

Advice on `scanf()`

- **Experiment** with it and make sure you understand how it works, how the format specifier affects results
 - The textbook and assigned readings are excellent resources on how different input strings are processed
- Always **check return value** to see if you read the number of values you were expecting
 - Conditional (**if-else**) statements soon...

(see `scanf_examples.c` in Code samples and Demonstrations in Canvas)

scanf () Example

```
char x, y;  
int j;  
scanf ("%c%c%d", &x, &y, &j);
```

Results with input

12345678912345678?

1 2 345678912345 1234?

(see [scanf_examples.c](#) in Code
samples and Demonstrations in Canvas)

References

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.
- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.