Type Conversions

CSC230: C and Software Tools

N.C. State Department of Computer Science

Computer Science
NC STATE UNIVERSITY

Outline

- Type Conversions
  - Explicit
  - Overflow and Underflow
  - Implicit
- More I/O in C
  - `scanf` and conversions

Type Conversions

- Data type conversions occur in two ways
  - explicitly (e.g., programmer deliberately *casts* from one type to another)
  - or implicitly (e.g., variables of different types are combined in a single expression, compiler casts from one type to another)

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

Explicit

Implicit

Casting (Explicit Conversion)

- Force a type conversion in the way specified
- Syntax: `(typename) expression`
- Ex.:

```
d = (double) c;
```

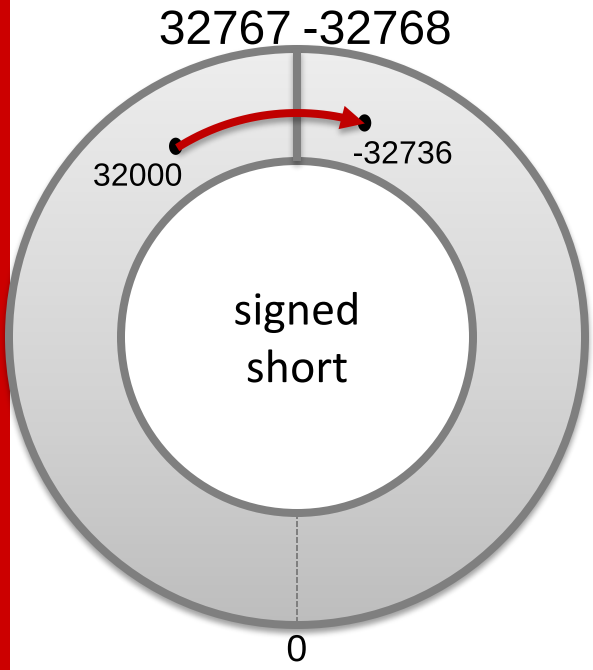
- Can the programmer get higher precision results by explicitly casting?
- A special case:

```
(void) expression;
```

  - means value of expression must not be used in any way
  - Q: how could that possibly be useful?
  - A: Prevent mistakes! Don’t let users set variables to void values.

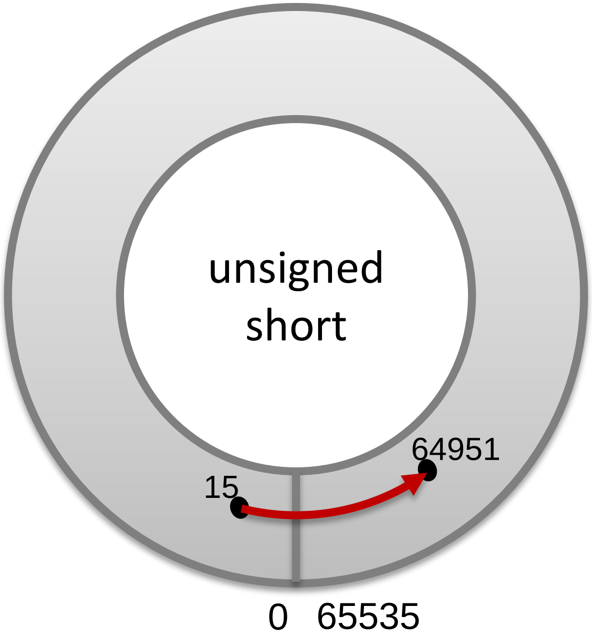
Overflow and Underflow

- Think of number ranges as a circle rather than a line
  - Example: **signed** and **unsigned short**
    - Shorts hold 16 bits on most machine
    - Signed Range: -((2^{16}) / 2) to (((2^{16}) / 2) – 1) or [-32768, 32767]
    - Unsigned Range: 0 to (2^{16} – 1) [0, 65535]



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

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



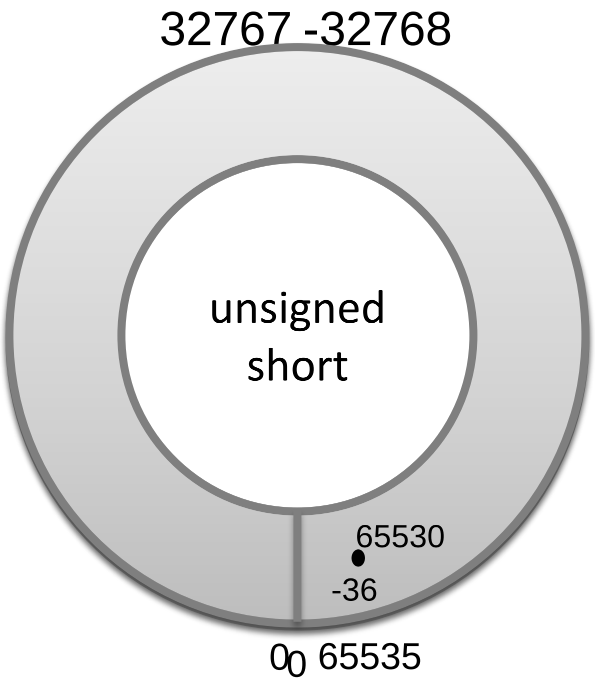
Converting **signed** to **unsigned**

- This only makes sense if you are ***sure*** the value stored in the **signed** operand is positive

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

Result when output:

```
b = 65500
a = -36
```



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

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

```
short a;
unsigned char b;
a = -36;
b = (unsigned char) a;
a = (signed) b;
```

Result when output:
```
b = 65500
a = -36
```

What happened?

Result when output:
```
b = 220
a = 220
```

We can't describe this effect using the number circle alone… Have to look at the bits!

Converting

- Extend bits with ones source is negative, extend with zeroes if source is positive.

```
short a;
unsigned char b;
a = -36;
b = (unsigned char) a;
a = (signed) b;
```

Result when output:

```
b = 220
a = 220
```

Variable	Decimal	Binary
a	-36	Strip high bits, treat as unsigned ~~11111111~~11011100
b	220	11011100
a	220	0000000011011100 Extend with zeroes, since source number is positive

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

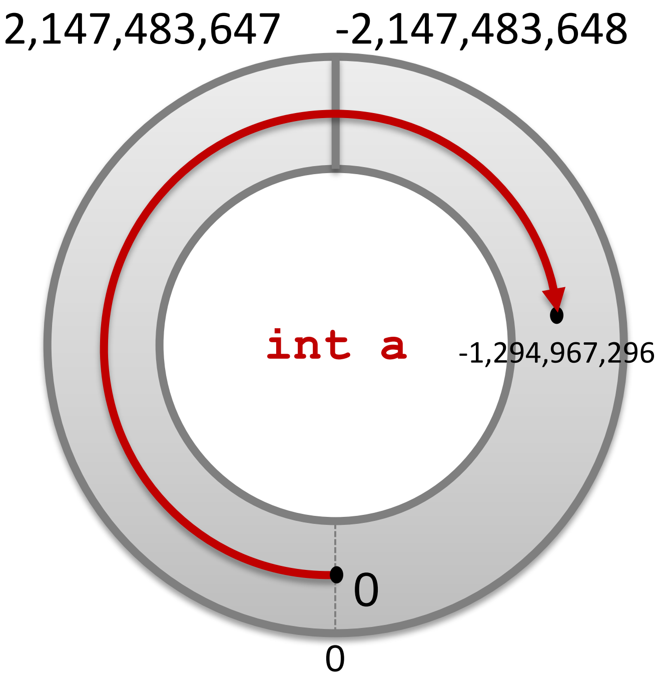
- If **signed** is large enough to store the correct value, no problems
  - otherwise, will definitely be an error (overflow)!

```
int a;
unsigned int b;


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

Result when output:

```
b = 3000000000
a = -1294967296
```



Exercise 04a

Conversions

- Given:
  - short a = -1;
  - int b = -2;
  - unsigned int c = 2147483648;
- State what the results of the following conversions would be if the variable is printed to the console.
  - `unsigned short d = (unsigned short) a;`
  - `unsigned int e = (unsigned int) b;`
  - `short f = (short) d;`
  - `int g = (int) e;`
  - `short h = (short) a;`
  - `int i = (int) a;`

Answer format:
d=<*blah*>
e=<*blah*>
etc...

Reminder: Go to course web page for link to exercise form. Paste code into ideone.com and submit the link.

Converting Floating to Integer

- Round towards zero (“truncate”) to get the integer part, and discard the fractional part
  - +3.999 → 3
  - -3.999 → -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:

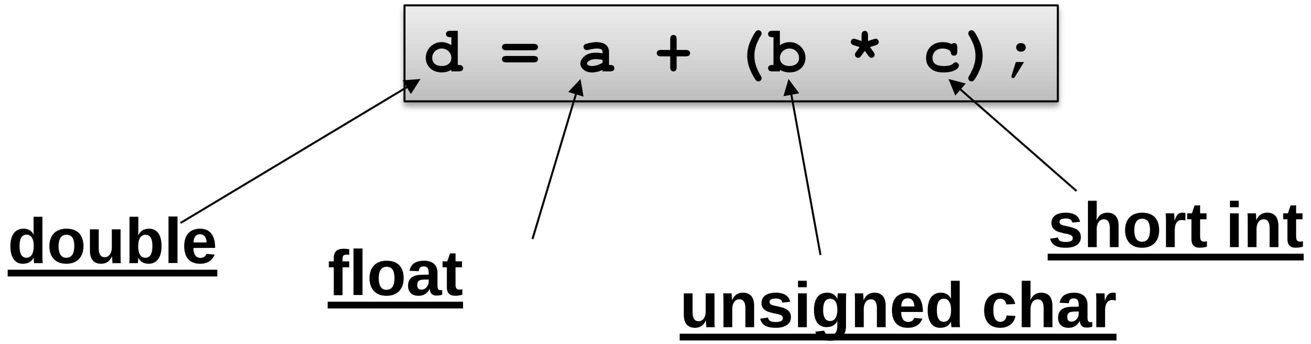
```
f = 10000000000.0
i = -2147483648
```

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.,



```
d = a + (b * c);
```

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

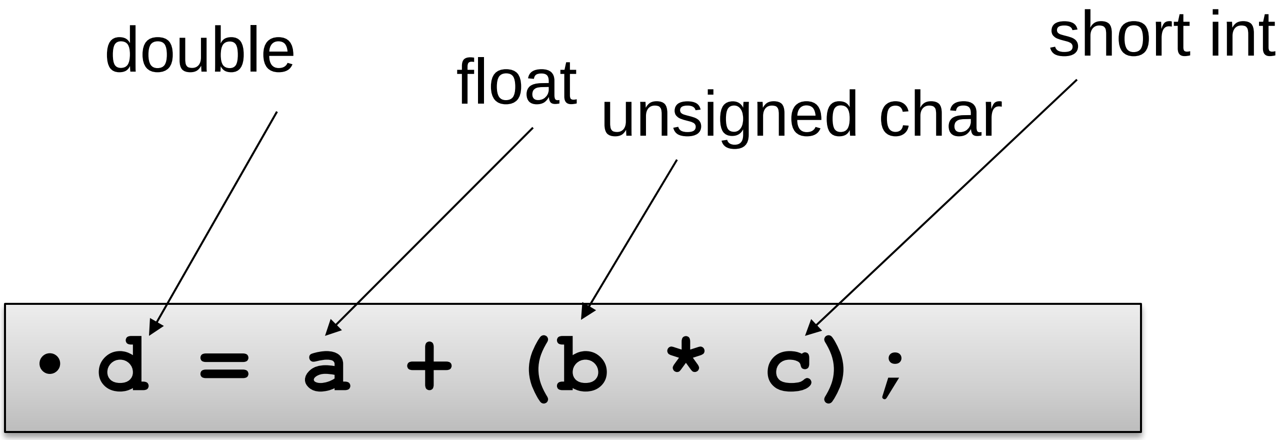
The "Usual Conversions" For Arithmetic Operations

- 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



- `d = a + (b * c);`

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)`**

The `scanf()` function

- `getchar()` is 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…
```

Parts of the Format Specifier

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

Some Types of Conversions

Convert input to Type…	Specifier
char	%c
unsigned int	%u (in decimal) %o (in octal) %x, %X (in hex) %lx, %lu, etc. for long
signed int	%d, %i (in decimal) %ld for long
float	%f (%lf for double)
float	%e, %E (use scientific notation) (%le for double)
(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
- Ex.:

```
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 format specifier affects results
  - The textbook is an excellent resource on different input strings are processed
- Always check return value to see if you read the number of values you were expecting
  - If statements soon…

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

Results with input

- **12345678912345678?**
- **1 2 345678912345 1234?**

Example: sum numbers on stdin

sum.c (simple)

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Input numbers...\n");
    int num_read;
    double value_read;
    double sum=0;
    while (1) {
        num_read = scanf("%lf", &value_read);
        if (num_read == 0) {
            break;
        }
        sum = sum + value_read;
    }
    printf("Sum: %f\n",sum);
    return EXIT_SUCCESS;
}
```

```
Input numbers...
3.14159
20
x
Sum: 23.141590
```

sum2.c (shorter)

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Input numbers...\n");

    double value_read, sum=0;

    while (scanf("%lf", &value_read)) {




        sum += value_read;
    }
    printf("Sum: %f\n",sum);
    return EXIT_SUCCESS;
}
```

Exercise 04b

Using scanf

- Write a program that uses scanf to read 3 integers from stdin, then print them in reverse order.

```
$ gcc reverse3.c && ./a.out
3
4
6
6
4
3
```

Reminder: Go to course web page for link to exercise form.
Paste code into ideone.com and submit the link.

Any Questions?