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

Computer Science
NC STATE UNIVERSITY

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)

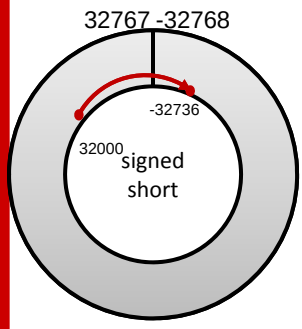
```
unsigned char a;
int b;
float c;
double d;
…
c = (float) b;
d = a + (b * c);
```


Casting (Explicit Conversion)

- Force a type conversion in the way specified
- Syntax: **`(typename) expression`**
- Ex.: `d = (double) c;`
- Q: Can the programmer get better quality 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?

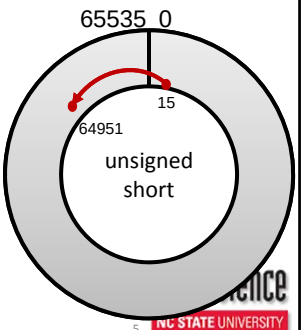
Overflow and Underflow

- 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
- 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

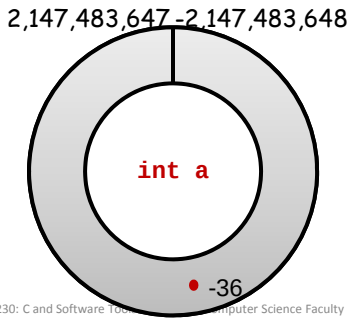
What happened???

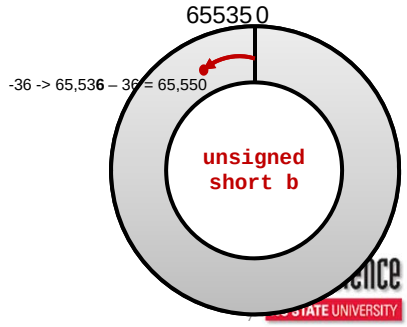
Converting

- If signed # 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
```





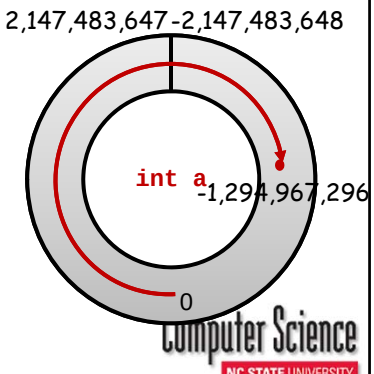
Converting **unsigned** to **signed**

- 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
```



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
```


Floating to Integer… (cont'd)

- Example

```
int i;
float g;
g = 12345678000000012345678.0;
i = (int) g;
```

?????

Result:

```
i = -2147483648
g = 12345678026834019824435 2.00000
```

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);
```

double **float** **unsigned char** **short int**

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

double float unsigned char short int

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

before evaluating expression:
  convert **`b`** to **`unsigned 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) (`%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
- 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 text book 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?**