Data Types in C (a deeper dive)

ESC101: Fundamentals of Computing

Nisheeth

Mixing Types in C Expressions

- We can have C expression with variables/constants of several types
- Certain rules exist that decide the type of the final value computed
- Demotion and Promotion are two common rules

```
int a = 2/3;          // a will be 0 (no demotion/promotion)
float a = 2/3;        // a will be 0.0 (RHS is int with value 0, promoted to float with value 0.0)
int a = 2/3.0;        // a will be 0 (RHS is float with value 0.66, becomes int with value 0)
float a = 2/3.0;      // a will be 0.66 (RHS is float with value 0.66, no demotion/promotion)
int a = 9/2;          // a will be 4 (RHS is int with value 4, no demotion/promotion)
float a = 9/2;        // a will be 4.0 (RHS is int with value 4, becomes float with value 4.0)
```

- During demotion/promotion, the RHS value doesn’t change, only the data type of the RHS value changes to the data type of LHS variable

Type Casting or Typecasting

Also remember: When assigning values, I always compute the RHS first

- Converting values of one type to other.
  - Example: int to float and float to int (also applies to other types)
- Conversion can be implicit or explicit. Typecasting is the explicit way

Automatic (compiler)

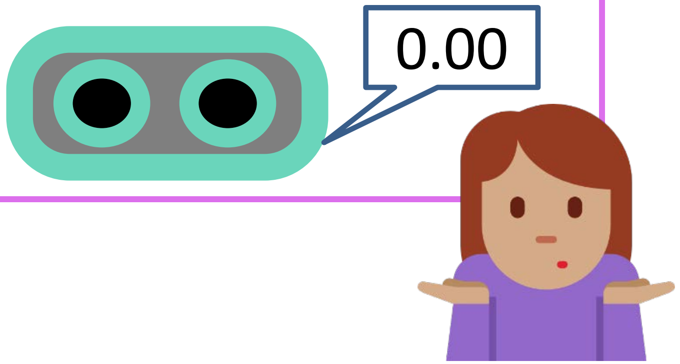
By us

```
int k =5;
float x = k;            // good implicit conversion, x gets 5.0
float y = k/10;         // poor implicit conversion, y gets 0.0
float z = ((float) k)/10; // Explicit conversion by typecasting, z gets 0.5
float z = k/10.0;       // this works too (explicit without typecasting), z gets 0.5
```

Typecasting: An Example Program

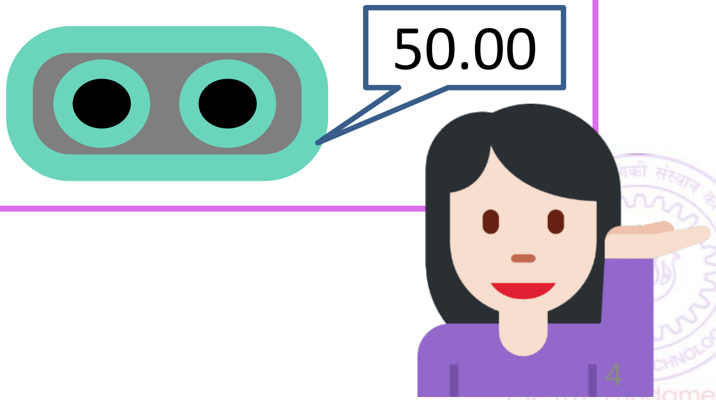
```c
#include <stdio.h>
int main(){
    int total = 100, marks = 50;
    float percentage;
    percentage = (marks/total)*100;
    printf("%.2f",percentage);
    return 0;
}
```

Equals 0



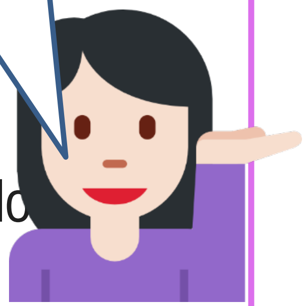

```c
#include <stdio.h>
int main(){
    int total = 100, marks=50;
    float percentage;
    percentage = (float)marks/total*100;
    printf("%.2f",percentage);
    return 0;
}
```

Typecasting makes it 50.0/100 which equals 0.5

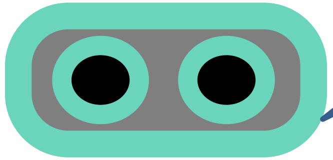


Typecasting is Nice. But Take Care..

```
#include <stdio.h>
int main(){
    float x; int y;
    x = 5.67;
    y = (int) x; // typecast (convert) flo
    printf(“%d”,y);
    return 0;
}
```

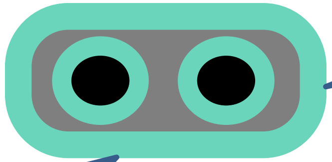


Expected conversion



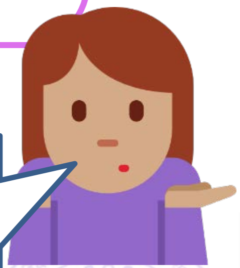
5

```
#include <stdio.h>
int main(){
    float x; int y;
    x = 1.0e50; // 10^50
    y = (int) x; // typecast (convert) float to int
    printf(“%d”,y);
    return 0;
}
```



-2147483648

No. 1.0e50 is too big to be cast as an int (or even long – try yourself)



Are you kidding? Unexpected!

Reverse typecasting error can happen too: Sometimes converting a smaller data type (say int) to larger data type (say float) can also give unexpected results (more on this later in the semester)

int and long

Very good friends since both store integers

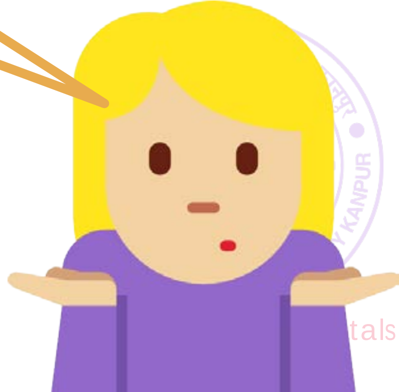
Can add/subtract/multiply/divide/remainder two ints, two longs, as well as an int and a long

In fact, even if we try to print an int using %ld or print a long using %d, Prutor will only warn us, not throw an error (but results at run-time may be unexpected sometimes)

long can store much larger integers than int

long can store smaller integers too ☺

So I don't have to be careful about anything?



int and long

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

Often, you don't have control over the kind of data you receive. Typecasting helps convert data to a form your like to work with

your requests nicely

}

Dotted line

I create th

variables r

will be e

Thankfully, we now know typecasting. It can save us here. ☺

Why not just define a long variable? No need for typecasting!

I should try this too:

long b = 2*(long)a;

long b = (long)a + a;

☺

4000000000

b = (long)a + (long)a;

printf("%ld",b);

}

40000 00000

a		b
20000 00000	40000 00000 / -2949 67296	

a			b
20000 00000			

Mixed Type Operations (Already Saw Some Cases)

What if we ha

c = a * b;

Can typecast int to long

b = (long) a;

Can typecast long to int

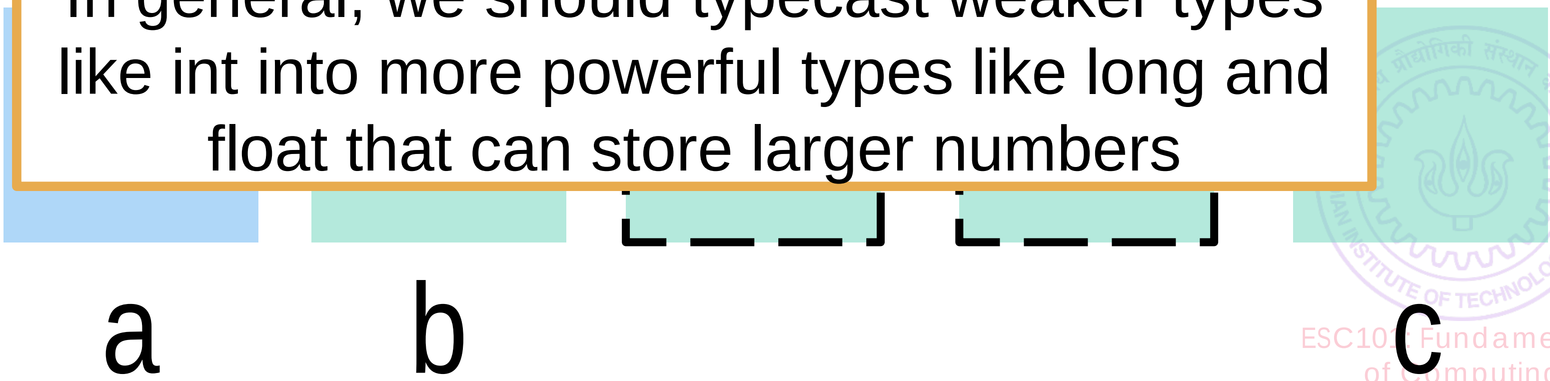
a = (int) b;

```
int a = 2;
long c, b = 5;
```

Hmm … An int being multiplied to a long. Let me take care to convert the int to a long before performing the operation ☺

Be careful! If b was storing a very large integer that won't fit into int, this typecast will cause errors

In general, we should typecast weaker types like int into more powerful types like long and float that can store larger numbers



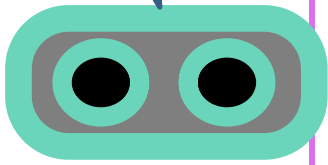
Arithmetic on char data type

Note: When printing a char using printf, the quote symbols ‘ ‘ are not shown

Note: When giving char input for scanf, we don’t type the quote symbols ‘ ‘

- Recall that each char is associated with an integer value
  - Example: char ‘A’ to ’Z’ are associated with integers 65 to 90
  - Refer to the ASCII table shown in last lecture’s slides
  - Note: signed char range is -128 to 127, unsigned char range is 0 to 255

Try in Prutor and see yourself



```
#include <stdio.h>
int main(){
    int x = ‘B’ - ‘A’ + 2;
    printf(“x = %d\n”, a);
    char y = 68;
    printf(“y = %c”,y);
    return 0;
}
```

3

D

```
#include <stdio.h>
int main(){
    char x = 128;
    printf(“x = %d\n”, x);
    char y = -130;
    printf(“y = %d\n”,y);
    return 0;
}
```

-128

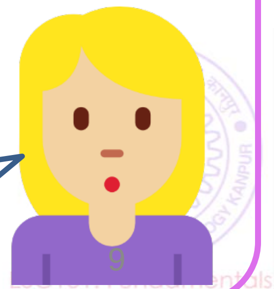
First number from the negative side

126

Second number from the positive side

128 and -130 are out of the range of signed char

What if x and y are unsigned char ?



Arithmetic on char data type: More Examples

- Keep in mind that char and int are inter-convertible

```
printf("%d\n", 'A');
printf("%d\n", '7');
printf("%c\n", 70);
printf("%c\n", 321);
```

Output:
65
55
F

So if you want, I can use/print a char as int and int as char (within char limits of course ☺)

321 is out of range of signed char (and even unsigned char)

Try in Prutor and see what happens

```
printf("%c\n", ‘C’+5);
printf("%c\n", ‘D’ - ‘A’ + ‘a’ );
printf("%d\n", ‘3’ + 2);
```

Output:
H
d
53

* and / are also valid but should avoid with char

Representing Negative Integers

- Mainly three ways
- \- Signed Magnitude
- \- One’s Complement
- \- Two’s Complement (used in modern computers)

- The Signed Magnitude approach is straightforward: To represent –x, take binary representation of x and make the left-most bit 1. So -7 (7 in binary = 111) will be

One’s Complement

The first bit acts as a sign bit – if the first bit is 1, it is treated as a negative number, if the first bit is 0, it is treated as a positive number

If we have n bits, then using one’s complement, we can represent numbers between $-(2^{n-1} - 1)$ and $+(2^{n-1} - 1)$

Largest positive integer is 01111111 11111111 11111111 11111111
Smallest negative integer is 1000000 00000000 00000000 00000000
Weird thing – negative 0 ☺
11111111 11111111 11111111 11111111

- Used no more. These days, computers use two’s complement to represent negative integers

Two's Complement

- Two's complement of an n-bit binary number is the number which when added to this number, gives 2^n
- $2^n = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \ldots\ldots 0$ (1 followed by n zero bits)
- This means two's complement of b is $2^n - b$
- Recall that $b + \sim b$ = all ones = $2^n - 1$ i.e. two's complement of b is $2^n - b = \sim b + 1$
- So a way of calculating two's complement – take the one's complement and add 1 to the binary string
- These days two's complement of an integer n represents its negative (that is –n)
- So for any integer n, one's complement of n will be -(n+1)

Two's Complement

The first bit acts as a sign bit – if the first bit is 1, it is treated as a negative number, if the first bit is 0, it is treated as a positive number

Largest positive number is 01111111 11111111 11111111 11111111
Smallest negative number is 1000000 00000000 00000000 00000000
11111111 11111111 11111111 11111111 now represents -1

If we have n bits, then using two's complement, we can represent numbers between -2^{n-1} and $+(2^{n-1} - 1)$

Floating Point Representation

- Have to represent three things
  - sign
  - Exponent
  - Number

- Assign some bits of memory for each
  - 1 bit for sign
  - m for exponent
  - n for mantissa

Conceptual Example

- Consider a 4 bit memory
  - What can you assign with unsigned int?
    - 0,1,.....15
  - What can you assign with signed int?
    - Use twos complement notation
    - -8,-7,.... ,7
  - What can you assign with float?

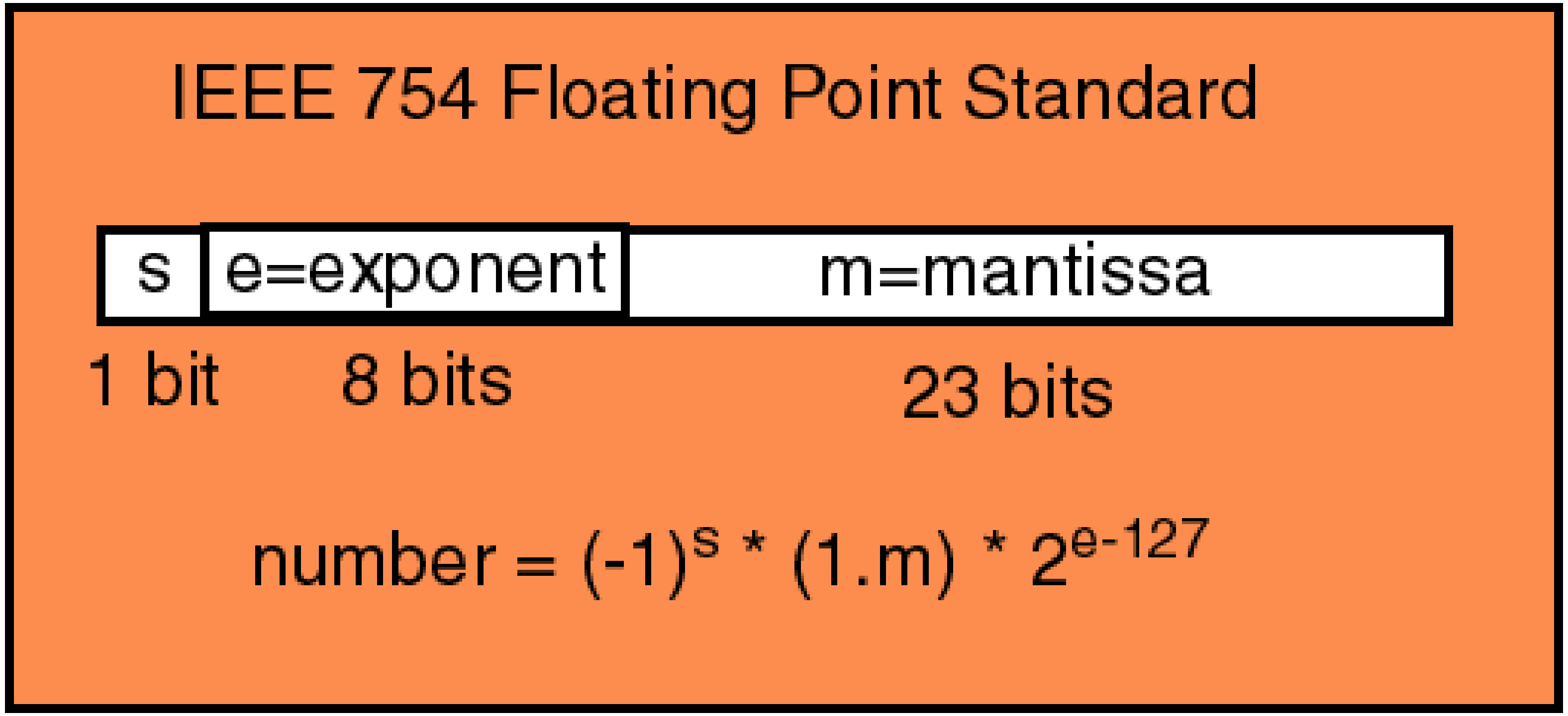
s	e	m	m

$(-1)^s * 1.m * 2^{e-0}$

1.0, 1.1, 1.2, 1.3
2.0, 2.2, 2.4, 2.6
-1.0, -1.1, -1.2, -1.3
-2.0, -2.2, -2.4, -2.6

This m is the decimal equivalent of 2 bits m m

IEEE 754 Floating Point Representation



Single-precision (float)

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign: 0 => positive
- Exponent:
  - $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: 104 - 127 = -23
- Significand:
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
    $=1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$
    $= 1.0 + 0.666115$
- Represents: $1.666115*2^{-23} \sim 1.986*10^{-7}$

This is what you're using when you are invoking *float*

Practical demonstration

- 12.375 = 12 + 0.375
- In binary = 1100 + .011 = 1100.011
- In IEEE notation = 1.100011 x 23
- So, the bias is 3, which means the exponent must be 127+3 = 130, which in binary format is 10000010
- So, the number, in IEEE single precision format will be
  - 0 – 10000010 - 10001100000000000000000

math.h

A really nice library of lots of mathematical functions

abs(x): absolute value of integer x

fabs(x): absolute value of x if x is float or double

ceil(x): ceiling function (smallest integer greater than x)

floor(x): floor function (largest integer smaller than x)

log(x): logarithm of x (do not give negative value of x)

pow(x,y): x to the power y (both doubles – typecast if int)

sqrt(x): square root of double x (typecast if not double)

cos(x), sin(x), tan(x) etc are also present – explore!

Operators

We have seen quite a few math operators till now

- +, -, *, /, %

All take two numbers and give one number as answer

- Called *binary operators* for this reason. Binary = two

Many *unary operators* also exist

- Have seen two till now:
- Unary negation int a = -21; b = -a;
- Typecasting c = (int) a;

Will see several more operators in the next class

Also will start expanding our programming power

- Conditional statements and relational operators