

# Lab information

- Labs start this week
- Lab - 01
  - On zoom → **this week only**
  - Lab work will be handed in on BrightSpace → **this week only.**
- Only attend registered/waitlisted lab section

# Types – Review

- int – holds an integer (whole number). Ex int demo\_int = 5;
- float – holds a number with decimals. Ex: float demo\_float = 3.14;
- double – holds a number with decimals (can hold a larger number than a float). Ex: double demo\_double = 3.14;
- char – holds a character. Assign in single quotes. Ex: char demo = 'd';

# Arithmetic Operators and Functions

# Unary and binary math operators

Operator Symbol	Operation	Description
+ , -	<u>Unary</u> +, -	Applies sign to one number
+	Binary Addition	Adds 2 numbers
-	Binary Subtraction	Subtracts one number from another
*	Binary Multiplication	Multiplies one number by another
/	Binary Division	Divides one number by another
%	Binary Modulo	Divides one number by another and gives the remainder of the division

# operators and types effect on expression evaluation

Operator	Applied to	Results in
<code>+, -, *, /, %</code>	<code>int, int</code>	<code>int</code>
<code>+, -, *, /</code>	<code>float, int</code> <code>int, float</code> <code>float, float</code>	<code>float</code>
<code>%</code>	<code>float, float</code> <code>float, int</code> <code>int, float</code>	Compile ERROR

It basically means we cannot use modulus (%) for any other variable types except integer types. This is important, as compilers will give compile error if we do that.

# Compound Assignments Operators


Operator	Example	Equivalent example
<code>--</code>	<code>a -= 5</code>	<code>a = a - 5</code>
<code>+=</code>	<code>b += 2</code>	<code>b = b + 2</code>
<code>*=</code>	<code>c *= 6</code>	<code>c = c * 6</code>
<code>/=</code>	<code>d /= 3</code>	<code>d = d / 3</code>
<code>%=</code>	<code>e %= 4</code>	<code>e = e % 4</code>

# Other Unary Operators

Operator	Example	Expanded equivalent version
<code>++ postfix</code>	<code>a++</code>	<code>a = a + 1</code>
<code>++ prefix</code>	<code>++a</code>	<code>a = a + 1</code>
<code>-- postfix</code>	<code>a--</code>	<code>a = a - 1</code>
<code>-- prefix</code>	<code>--a</code>	<code>a = a - 1</code>

# Adding operators to precedence table

Precedence	Description	Associativity
Highest	Operations enclosed in brackets ( ), <b>++/-- postfix</b>	left to right
	<b>+/- unary operator, ++/-- prefix</b>	right to left
	<b>*, /, %</b>	left to right
	<b>+, -</b>	left to right
Lowest	<b>=, +=, -=, *=, /=, %=</b>	right to left





# casting

- Forcing a value to be a specified type

- explicit casting:

```
int i = 5;  
(double)i → 5.0
```

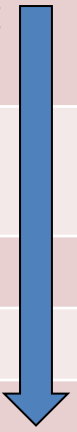
```
double d = 7.9;  
(int)d → 7
```

- implicit casting:

```
int i = 5.7;    // value of i is 5  
double d = 7;  // value of d is 7.0
```

# Adding **cast** to precedence table

Precedence	Description	Associativity
Highest	Operations enclosed in brackets ( ), ++/-- postfix	right to left
	+/- unary operator, ++/-- prefix, <b>(type) cast</b>	right to left
	<b>*</b> , <b>/</b> , <b>%</b>	<b>left to right</b>
	+, -	left to right
Lowest	=, +=, -=, *=, /=, %=	right to left



# main is a function

General form:

```
int main ( void ) {  
    C statement 1;  
    C statement 2;  
    return 0;  
}
```

optional



# main is a function

It can call other functions from other libraries:

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

int main ( ) {
    printf("hi");

    double d = 3.5;
    double sin_of_d = sin(d);

    return 0;
}
```

# 2 pieces to define your own function

General form:

Concrete example:

Function prototype

```
void fn_name();
```

```
void print_number();
```

```
void fn_name () {  
    C statement 1;  
    C statement 2;  
}
```

```
void print_number () {  
    int num = 10;  
    printf("%d\n", num);  
}
```

Function definition

# Calling a function

```
void fn_name();
```

Function prototype



```
fn_name();
```

Function call



Must be called from a reachable  
point in your program

# Putting it all together

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

```
void print_number();
```

← Function PROTOTYPE

```
int main ( ) {  
    print_number();  
    return 0;  
}
```

← Function CALL

```
void print_number ( ) {  
    int num = 10;  
    printf("%d\n", num);  
}
```

} Function DEFINITION

# Function naming rules

- Rules
  - Cannot use key words as a function name
  - Cannot contain spaces
  - First character must be a letter or underscore
  - All other characters must be a letter, number or underscore
  - Uppercase and lowercase characters are distinct
- We will use typical C function naming conventions:
  - Begin with a lowercase letter
  - Typically will contain a verb (action word)
  - Multi-word variable names have words separated with ‘\_’
- Examples:

```
void calculate_area();  
...
```

```
void print_num();  
...
```