Variables & Data Types

Variables

- **Variables** are names used to refer to some location in memory - a location that holds a value.
  - Think of variables as boxes to store data in
- **Declaring** a variable brings the variable into existence
  - This amounts to creating the box to store the value in
  - But how does the computer know what size the box should be?
    * Not all data has the same size
    * The compiler uses the **type** of the data to determine how much memory is needed to store the variable
- All variables in C are typed
- **Initializing** a variable means you **assign** the variable a value when you declare it
- Some examples:

```
int a; // declars a as an integer

int anumber, anothernumber, athirdnumber; // declares three variables, all of which are integers

int b = 5; // declares b as an int an initializes its value to 5

b = 10; // assigns b to have the value of 10

anumber = b; // assigns anumber to have the value of b, which is 10

anumber = anothernumber = athirdnumber = b; // assigns anumber, anothernumber, and athirdnumber to have the value of b, which is 10
```

Naming variables

- Variable names are made up of letters (upper and lower case), digits, and the underscore character "_".
- Names cannot begin with a digit
- Some valid variable names:

```
foo
Bar
BAZ
```

```
foo_bar
_foo42
_
QuUx
```

- Some invalid variable names:

```
2foo     // must not begin with a digit
my foo   // spaces not allowed in names
$foo     // $ not allowed -- only letters, and _
while    // language keywords cannot be used as names
```

- You may only use the same variable **once** within the same variable scope

Literals

- A value, literally
- `5` is a literal. `32.3` is a literal
- These are invariant values. They can never be changed. They can never store data.
- They are literally some value.

Basic Data Types

- Four basic types:
  1. `int`
  2. `char`
  3. `float`
  4. `double`

`int`

- Stores an integer value.
- Typically stored in 32 bits (the computer uses 32 bits to represent the number)
  - If you have a set of integers centered around 0, what's the maximum and minimum integer you can represent with 32-bits?
    * 32 bits leads to 4294967296 which is 2^{32} (binary is base 2, and we have 32 bits)
    * Maximum value: `+2147483647`
    * Minimum value: `-2147483648`

- Example usage:

```
int a = 5;
```

`char`

- Capable of holding any member of the character set.
- Stored in 1 byte (8 bits).
- The underlying structure has the same type of data as an `int` (with a smaller range of data)
  - However, we the way we *should* use chars is not through integer references
  - This is all because internally a character is literally an integer to the computer
- Examples of characters:

```
'a'
'b'
'3'
'\0'   // null character
'\n'   // newline character
'\t'   // tab character
```

- A **string literal** is a collection of characters in a single string
  - `"Hello, world!"` is an example of a string literal
  - String literals are denoted by " instead of ' for their wrapping quotations

`float`

- Holds a floating point number, such as `32.2`
- All representations of floating point numbers are inexact.
- Adding `f` to the end of a number indicates it is to be interpreted as a float
- Examples of floats:

```
32.3
3223.64563f
4.0f
6.022e+23f
```

`double`

- Exact same as a **`float`**, but uses double the precision (i.e. double the computer memory) to store the data

`sizeof`

- If you need to know the exact size of a variable, you can use `sizeof` (a unary operator) to find out:

```
sizeof(type)
sizeof obj
```

- This returns the size of the underlying type specified
- The type of `sizeof` returns is `size_t`, which represents a size (unsigned value)

```
size_t size;
int i;
size = sizeof(i);
```

- In this case, we should get `size` assigned to 4, since an integer is typically 4 bytes (32 bits).

Type Modifiers

- We may want to modify the amount of storage used by a type.
- This enables data to use more or less memory depending upon the use case.
- Adding a modifier of **`long`** will make the type use more memory
- Adding a modifier of **`short`** will make the type use less memory
- Adding a modifier of `unsigned` will make the type non-negative in all cases (changes the range of possible values
- If you use **`short`** or **`long`** by itself, the **`int`** type is implied

```
unsigned short int usi;  /* fully qualified -- unsigned short int */
short si;                /* short int */
unsigned long uli;       /* unsigned long int */
```

- The **`const`** makes a particular variable constant, or unmodifiable.
  - You *must* initialize the value when you declare it.
  - What's the advantage?

    * You gain additional protections against a programmer making a mistake and modifying a value they shouldn't
    * Also protects against magic numbers - don't put the same literal all over your program. Use a constant to define the value once and use the constant everywhere you need that value

Simple IO

Output

- Input is the process of getting information from the user of your program
- Output is the process of presenting/saving information from the results of your program
- For now, all IO we deal with will come from the `stdio.h` Standard Library file.
- Recall our first program

```
#include <stdio.h>

int main(void)
{
  printf("Hello, World!");
  return 0;
}
```

- This will print the following on your screen:

```
Hello, world!
```

- This is a form of output to the user using the `printf()` function.
- The `printf` function takes an argument, namely the string you want to print
    - This can be a string literal or a C-style string (we'll cover these later)

Placeholders

- This is great, but what if we want to output the results of some computation?
- We can't type the result into the program directly (that would miss the whole point of having the computer compute something!).
- Instead, we can insert a **placeholder** to indicate we will place the value of a variable in the string
- Example:

```
#include <stdio.h>

int main(void)
{
  printf("19+31 is %d", 19+31);
  return 0;
}
```

- The `%d` here indicates we want to print an integer
- These placeholders are called **format specifiers**.
- Here is a list of important format specifiers:

```
%d    // int (same as %i)
%ld   // long int (same as %li)
%f    // float
%lf   // double
%c    // char
%s    // string
%x    // hexadecimal
```

- You can find a complete list of format specifiers here.

Tabs and Newlines

- We need to tell `printf` when we want to actually print whitespace
- For instance, suppose we wanted the following output:

```
 1905
312 +
-----
```

- We can insert a newline *escape character* with `\n`.
- All escaper characters begin with a \
- To get the output above, we would use the following `printf` statement

```
printf(" 1905\n312 +\n-----\n");
```

- We can also (more typically) split this over multiple lines

```
#include <stdio.h>

```

```c
int main(void)
{
  printf(" 1905\n");
  printf("312 +\n");
  printf("-----\n");
  printf("%d", 1905+312);
  return 0;
}
```

Input

- Similar to `printf`, we use a function called `scanf` to get basic input from the user.
- Placeholders are mostly similar to those of `printf`
- However, because we are getting a value from the user, we need a place to store that value
  - Where should we store this value? A variable
- Instead of directly giving `scanf` our variable, instead we'll give it a *address* to the variable
  - We'll talk more about addresses later (when we learn about pointers), but for now, think of a pointer as the memory location of a variable
- We'll get the *address* of the variable with the *address of* operator (&)
- Here's an example of getting an integer from the user:

```c
#include <stdio.h>

int main(void)
{
  int a;

  printf("Please input an integer value: ");
  scanf("%d", &a);
  printf("You entered: %d\n", a);

  return 0;
}
```

Basic Operators

- C supports basic arithmetic operators to help you do math.
- Basic operators include:
  - + addition

  - \- subtraction
  - * multiplication
  - / division (floating point and integer division depending upon type)
  - % modulo (remainder division)

Modulo (remainder division)

- Remember integer division from elementary school?
- e.g. `7/5` was `1r2` (1 with a remainder of 2) because 5 goes into 7 one time with a remainder of 2.
- When you divide two ints, you only get the quotient (number of times the denominator goes into the numerator).
- Modulo `%` gives us a way to get the remainder from the quotient division.
- Modulo is *extremely* useful.
  - It lets you add a bound to possible values.
  - For instance, suppose you want to pick a random number between 0 and 9.
  - Let's say you have a `rand()` function that returns a random number between 0 and a really, really big number (say 10000000000).
  - You can do `rand()% 10` and you are guaranteed to get a number between 0 and 9.
  - It doesn't matter how big the number is, the remainder *must* be between 0 and 9.
  - Otherwise, the quotient increments!

Exercises (for practice only)

1. Write a C program to print your name, date of birth. and mobile number.

```c
#include <stdio.h>
int main()
{
  printf("Name   : Alexandra Abramov\n");
  printf("DOB    : July 14, 1975\n");
  printf("Mobile : 99-9999999999\n");
  return 0;
}
```

2. Write a C program to compute the perimeter and area of a rectangle with a height of 7 inches. and width of 5 inches.

```c
#include <stdio.h>

```

```c
int main() {
  int width;
  int height;

  int area;
  int perimeter;

  height = 7;
  width = 5;

  perimeter = 2*(height + width);
  printf("Perimeter of the rectangle = %d inches\n", perimeter);

  area = height * width;
  printf("Area of the rectangle = %d square inches\n", area);

  return 0;
}
```

3. Write a C program that accepts two integers from the user and calculate the product of the two integers.

```c
#include <stdio.h>
int main()
{
  int x, y, result;
  printf("\nInput the first integer: ");
  scanf("%d", &x);
  printf("\nInput the second integer: ");
  scanf("%d", &y);
  result = x * y;
  printf("Product of the above two integers = %d\n", result);
}
```

4. Write a C program to convert specified days into years, weeks and days.

```c
#include <stdio.h>
int main()
{
  int days, years, weeks;

  days = 1329;

```

```c
    // Converts days to years, weeks and days
    years = days/365;
    weeks = (days % 365)/7;
    days = days - ((years*365) + (weeks*7));

    printf("Years: %d\n", years);
    printf("Weeks: %d\n", weeks);
    printf("Days: %d \n", days);

    return 0;
}
```