Numeric Data Types Outline

1. Numeric Data Types Outline
2. Basic Data Types
3. Integers in Mathematics
4. Integers Are Also Called Fixed Point Numbers
5. Declaring `int` Variables in C
6. `int` Data Don’t Have to Be 4 Bytes Long, But They Usually Are
7. `int` Declaration: Example
8. `int` Literal Constants
9. Declaring & Using `int` Named Constants: Example
10. Real Numbers in Mathematics
11. Representing Real Numbers in a Computer
12. Declaring `float` Variables in C
13. `float` Declaration: Example
14. Scientific Notation
15. Why Can C `floats` Only Approximate Mathematical Real Numbers?
16. Floating Point Approximation
17. `float` Literal Constants
18. Declaring & Using `float` Named Constants: Example
19. Why Have Both Real Numbers & Integers?

Basic Data Types

- Numeric
  - `int`
  - `float`
- Non-numeric
  - `char`

```c
#include <stdio.h>

int main ()
{
    /* main */
    float standard_deviation, relative_humidity;
    int count, number_of_silly_people;
    char middle_initial, hometown[30];
} /* main */
```
Integers in Mathematics
Mathematically, an integer is any number (positive, negative or zero) that has nothing but zeros to the right of the decimal point:

-3984.00000000...
  0.00000000...
  23085.00000000...

Another way to think of integers is as
- the counting numbers
  1, 2, 3, 4, 5, 6, ...
- their negatives (additive inverses)
  ..., -6, -5, -4, -3, -2, -1
- and 0.

In computing, an integer has these mathematical properties; it also has a particular way of being represented in memory (which we’ll see later in the course) and a particular way of being operated on.

Integers Are Also Called Fixed Point Numbers
Integers are sometimes called fixed point numbers, because they have an invisible decimal point in a fixed (i.e., unchanging) position.

Specifically, every integer’s invisible decimal point is to the right of the rightmost digit (the “ones” digit):

-3984
  ↑
  invisible decimal point

 0
  ↑
  invisible decimal point

23085
  ↑
  invisible decimal point

In C (and in most computer languages), int literal constants are expressed **without a decimal point**:

-3984
  0
  23085
Declaring `int` Variables in C

```c
int x;
```

This declaration tells the compiler to grab a group of bytes, name them `x`, and think of them as storing an integer.

**How many bytes?**

That depends on the platform and the compiler, but these days the typical answer is that an `int` takes 4 bytes (32 bits) in most cases:

```c
x:
```

For example, on Pentium-based Linux PCs such as roosevelt, kennedy and lincoln, using the gcc compiler from gnu.org, which is the compiler that we’re using in this course, the size of an `int` is 4 bytes.

---

`int` Data Don’t Have to Be 4 Bytes Long, But They Usually Are

On some platforms (combinations of hardware family and operating system), under some compilers, **all** `int` data are 4 bytes in size.

On other platforms the default `int` size is 4 bytes, but the size of an `int` can be changed by using a compiler option.

A notable exception is the Cray family of supercomputers, whose default `int` size is 8 bytes; to get 4 byte (32 bit) `int` data, you have to use the compiler option `-i 32`.

Notice that different compilers for the same language can have different names, different defaults and different options. While there are many common features, compiler vendors are under no compulsion to follow them.
**int Declaration: Example**

```c
#include <stdio.h>

int main ()
{
    int height_in_cm;

    height_in_cm = 160;

    printf("My height is %d cm.
    
", height_in_cm);
}
```

**int Literal Constants**

An *int literal constant* is a sequence of digits, possibly preceded by an optional sign:

**CORRECT:**

- `0`
- `-345`
- `768`
- `+12345`

**INCORECT:**

- `1,234,567`
  
  No commas allowed.
- `12.0`
  
  No decimal point allowed.
- `--4`  `++3`
  
  Only one sign per int literal constant.
- `5--7`
  
  The sign must come before the digits, not after.

We can use int literal constants in several ways:

- In declaring and initializing a named constant:
  ```c
  const int w = 0;
  
  /* 0 is an int literal constant */
  ```
- In initializing a variable (within a declaration):
  ```c
  int x = -19;
  
  /* -19 is an int literal constant */
  ```
- In an assignment:
  ```c
  y = +7;
  
  /* +7 is an int literal constant */
  ```
- In an expression (which we’ll learn more about shortly):
  ```c
  z = y + 9;
  
  /* 9 is an int literal constant */
  ```

The first two of these are considered GOOD programming practice; the last two of these are considered BAD programming practice, because in these cases it would be better to use named constants.
Declaring & Using Named Constants: Example

```c
#include <stdio.h>

int main ()
{
    const int number_of_people_to_tango = 2;
    const int number_of_blind_mice = 3;
    const int inches_per_foot = 12;
    const int degrees_in_a_circle = 360;
    const int US_drinking_age_in_years = 21;

    printf("It takes %d to tango.\n", number_of_people_to_tango);
    printf("\n");
    printf("%d blind mice, see how they run.\n", number_of_blind_mice);
    printf("\n");
    printf("There are %d inches in a foot.\n", inches_per_foot);
    printf("\n");
    printf("There are %d degrees in a circle.\n", degrees_in_a_circle);
    printf("\n");
    printf("In the US, you can’t legally drink until %d years old.\n", US_drinking_age_in_years);
} /* main */
```

```
gcc -o intconsts intconsts.c
```

```
It takes 2 to tango.
3 blind mice, see how they run.
There are 12 inches in a foot.
There are 360 degrees in a circle.
In the US, you can’t legally drink until you’re at least 21 years old.
```

ASIDE: notice that you can output a blank line by printing a string literal containing only the newline character \n.

---

Real Numbers in Mathematics

Mathematically, a real number is a number (positive, negative or zero) with any sequence of digits on either side of the decimal point:

- -3984.75
- 0.1111111...
- 3.1415926...

In mathematics, the string of digits to the right of the decimal point can be either:
- terminating (i.e., a finite number of nonzero digits),
- repeating (i.e., a finite sequence of digits repeated infinitely), or
- non-repeating.

There are infinitely many real numbers, and in fact infinitely many real numbers between any two real numbers; for example, there are infinitely many real numbers between 0 and 0.0000000000000001.

Notice that, in mathematics, all integers are real numbers, but not all real numbers are integers.

In particular, mathematically every integer is a real number, because it has a finite number of nonzero digits to the right of the decimal point — specifically, since it has no nonzero digits to the right of the decimal point, the finite number is zero.
Representing Real Numbers in a Computer

In a computer, a real value is stored in a finite number of bits (typically 32 or 64 bits), so a computer’s representation of real numbers can only approximate most mathematical real numbers, because only finitely many different values that can be stored in a finite number of bits (e.g., 32 bits can have only $2^{32}$ possible different values).

Like integers, real numbers have particular ways of being represented in memory (which we’ll look at later in the course) and of being operated on.

In C (and in most computer languages), float literal constants are expressed with a decimal point:

-3984.75
0.0
23085.1235

Recall that, in mathematics, all integers are reals, but not all reals are integers. Similarly, in most programming languages, some real numbers are mathematical integers (e.g., 0.0), even though they are represented in memory as reals.

Reals are often called floating point numbers. We’ll see why soon.

Declaring float Variables in C

float x;

This declaration tells the compiler to grab a group of bytes, name them x, and think of them as storing a real number.

How many bytes?

That depends on the platform and the compiler, but these days the typical answer is that real numbers take 4 bytes (32 bits) or 8 bytes (64 bits) in most cases:

x:
OR
x:

For example, on Pentium-based Linux PCs such as roosevelt, kennedy and lincoln, using the gcc compiler from gnu.org, which we’re using in this course, the default size of a float is 4 bytes (32 bits).

A notable exception to the 4 byte default rule is the Cray family of supercomputers, whose default real size is 8 bytes (64 bits).
**float Declaration: Example**

```c
#include <stdio.h>

int main ()
{
    /* main */
    /* Declaration section */
    * Local variables *
    float height_in_m;
    /* Execution section */
    * Assign the real value 1.6 to height_in_m. *
    height_in_m = 1.6;
    /* Print height_in_m to standard output. */
    printf("My height is %f m.\n", height_in_m);
} /* main */
```

**Scientific Notation**

In technical classes, we often encounter scientific notation, which is a way of writing numbers that are either very very big or very very small:

\[ 6,300,000,000,000 = 6.3 \times 10^{15} \]
\[ 0.00000000000271 = 2.71 \times 10^{-11} \]

In C, we can express such numbers in a similar way:

\[ 6,300,000,000,000 = 6.3e+15 \]
\[ 0.00000000000271 = 2.71e-11 \]

Here, the \( e \), which stands for “exponent,” indicates that the sequence of characters that follows — an optional sign followed by one or more digits — is the power of 10 that the number to the left of the \( e \) should be multiplied by.

When we express a real number in scientific notation, the decimal point is immediately to the right of the leftmost non-zero digit. So, the decimal point doesn’t have to be to the right of the “ones” digit; instead, it can be after any digit.

So, we sometimes call real numbers floating point numbers.

We recall that, similarly, integers are sometimes called fixed point numbers, because they have an implicit decimal point that is always to the right of the “ones” digit (i.e., the rightmost digit), with implied zeros to the right of the implied decimal point:

\[ 6,300,000,000,000 = 6,300,000,000,000,000,000,\ldots \]
Why Can C floats Only Approximate Mathematical Real Numbers?

In C (and in most other computer languages), real numbers are represented by a finite number of bits.

For example, on Linux PCs like roosevelt, lincoln and kennedy, the default size of a real number is 32 bits (4 bytes).

We know that 32 bits can store $2^{32} = 2^2 \times 2^{30} \approx 4,000,000,000$ possible values. And that’s a lot of possibilities.

But: there are infinitely many (mathematically) real numbers, and in fact infinitely many real numbers between any two real numbers. For example, between 1 and 10 we have:

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9</td>
<td>3.8</td>
<td>4.7</td>
<td>5.6</td>
<td>6.5</td>
<td>7.4</td>
<td>8.3</td>
<td>9.2</td>
</tr>
<tr>
<td>2.09</td>
<td>3.08</td>
<td>4.07</td>
<td>5.06</td>
<td>6.05</td>
<td>7.04</td>
<td>8.03</td>
<td>9.02</td>
</tr>
<tr>
<td>2.009</td>
<td>3.008</td>
<td>4.007</td>
<td>5.006</td>
<td>6.005</td>
<td>7.004</td>
<td>8.003</td>
<td>9.002</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, no matter how many bits we use to represent a real number, we won’t be able to exactly represent most real numbers, because we have an infinite set of real numbers to be represented in a finite number of bits.

Floating Point Approximation

No matter how many bits we use to represent a real number, we won’t be able to exactly represent most real numbers, because we have an infinite set of real numbers to be represented in a finite number of bits.

For example:
if we can exactly represent 0.125 but not 0.125000000000000000000000000000001, then we have to use 0.125 to approximate 0.125000000000000000000000000000001.

```c
#include <stdio.h>

int main ()
{
    float input_value;
    printf("What real value would you like stored?\n");
    scanf("%f", &input_value);
    printf("That real value is stored as %f.\n", input_value);
    return 0;
}
```

% cat realapprox.c
#include <stdio.h>

int main ()
{
    float input_value;
    printf("What real value would you like stored?\n");
    scanf("%f", &input_value);
    printf("That real value is stored as %f.\n", input_value);
}
% gcc -o realapprox realapprox.c
% realapprox
What real value would you like stored? 0.125000000000000000000000000000001
That real value is stored as 0.125000.
**float Literal Constants**

A *float literal constant* is an optional sign, a string of digits, a decimal point (which is optional if there is an exponent), an optional string of digits, and an optional exponent string, which consists of an *e*, an optional sign, and a string of digits.

You can tell that a numeric literal constant is a *float literal constant* by whether it has either a decimal point, or an *e*, or both.

- 0.
- -345.3847
- 7.68e+05
- +12345.434e-13
- 125.e1

We can use *float literal constants* in several ways:

- **In declaring and initializing a named constant:**
  
  const float w = 0.0;
  /* 0.0 is a float literal constant

- **In initializing a variable (within a declaration):**
  
  float x = -1e-05;
  /* -1e-05 is a float literal constant

- **In an assignment:**
  
  y = +7.24690120;
  /* +7.24690120 is a float literal constant */

- **In an expression** (which we’ll learn more about shortly):
  
  z = y + 125e3;
  /* 125e3 is a float literal constant */

The **first two** of these are considered GOOD programming practice; the **last two** of these are considered BAD programming practice, because in these cases it would be better to use named constants.

---

**Declaring & Using float Named Constants: Example**

```c
#include <stdio.h>

int main () {
  /* main */
  const float pi = 3.1415926;
  const float radians_in_a_semicircle = pi;
  const float number_of_days_in_a_solar_year = 365.242190;
  const float US_inflation_percent_in_1998 = 1.6;
  printf("pi = %f\n", pi);
  printf("There are %f radians in a semicircle.\n",
         radians_in_a_semicircle);
  printf("There are %f days in a solar year.\n",
         number_of_days_in_a_solar_year);
  printf("The US inflation rate in 1998 was %f\%\n",
         US_inflation_percent_in_1998);
} /* main */
```

```bash
$ gcc -o realconsts realconsts.c
$ realconsts
pi = 3.141593
There are 3.141593 radians in a semicircle.
There are 365.242188 days in a solar year.
The US inflation rate in 1998 was 1.600000%.
```

Again, notice that you can output a blank line by printing a string literal containing only the newline character \n.

Reference:

http://scienceworld.wolfram.com/astronomy/LeapYear.html
Why Have Both Real Numbers & Integers?

1. **Precision**: ints are exact, floats are approximate.
2. ** Appropriateness**: For some tasks, ints fit the properties of the data better.
   For example:
   - counting the number of students in a class
   - array subscripting (which we’ll see later on)
3. **Readability**: When we declare a variable to be an int, we make it obvious to anyone reading our program that the variable will contain only certain values (specifically, only integer values).
4. **Enforcement**: When we declare a variable to be an int, no one can put a non-int into it.
5. **History**: For a long time, operations on int data were much quicker than operations on float data, so anything that you could do with ints, you would. Nowadays, operations on floats can be as fast as operations on ints, so speed is no longer an issue.