Fundamental Data Types

Outline

1. Motivation

- After our previous lecture, you should be able to have intuitive understanding of the following simple C program

```c
#include <stdio.h>

int main(void)
{
  int area, h, b;

  h = 3;
  b = 4;
  printf("Height and base of triangle: %d, %d\n", h,
b);
  area = (1/2) * h * b;
  printf("Area of triangle: %d\n", area);

  return 0;
}
```

1. Motivation

- This C program, however, do not produce the expected result. Why?! All the formula looks alright to you, right?

```c
#include <stdio.h>

int main(void)
{
  int area, h, b;

  h = 3;
  b = 4;
  printf("Height and base of triangle: %d, %d\n", h,
b);
  area = (1/2) * h * b;
  printf("Area of triangle: %d\n", area);

  return 0;
}
```

1. Motivation

- To understand the problem of the above program, you need to learn more about data types
- In the previous lecture, we have introduced the *integer* data type:

```
int x = 0;
```

- You may also recall a *double* data type:

```
double pi = 3.14;
```

- How do they differ? Why do we need to have different data types?
- Understanding data types thoroughly is a crucial first step of being a programmer in C

2. Integral (Integer) Types

Key concepts

- The smallest/largest integer values of type `int`

- Variation of integer types
  - Integer types of different sizes
  - Unsigned and signed integers

- Integer overflow

2.1. Basic Building Block: Bits

- Ultimately, a computer represent data in *bits*.
- A *bit* (binary digit) is the smallest unit data.
- A bit can be either 0 or 1.
- 1 *byte* == 8 bits
- Computers use combination of bits to represent all sorts of data.

2.1. From Bits to Integers

- Binary number system uses *N* bits to represent integers.
  - Typically, *N* = 8, 16, 32, 64 (corresponding to 1, 2, 4 and 8 bytes)

00000000, 00000001, 00000010, 00000011, ..., 11111110, 11111111
Some people assign each a *non-negative* integer value, 0, 1, 2, ..., 254, 255.

Some people rearrange the patterns in a different order:

10000000, 10000001, 10000010, 10000011, 10000100, 1000101, ...,
11111110, 11111111, 00000000, 00000001, 00000010, ..., 01111111

And call them -128, -127, -126, -125, ..., -2, -1, 0, 1, 2,3, ..., 125, 126, 127
The patterns are used for representing both –ve, 0 and +ve integers.

2.1. From Bits to Integers

- With N bits, we can represent 2^N distinct values.
  - Half for negative integers, and half for non-negative integers:

    $-(2^{N-1}), -(2^{N-1} - 1), \ldots, -2, -1, \quad 0, +1, +2, \ldots, +(2^{N-1} - 1)$

- Type `int` is typically 32 bits in size *nowadays*. As such, it can represent integers in the following range

$-(2^{31}), -(2^{31} - 1), \ldots , -2, -1, \quad 0, +1, +2, \ldots, +(2^{31} - 1)$

or

-2147483648, ..., -2, -1, 0, 1, 2, ..., 2147483647

2.2. Variations of Integral Types

Type	Size in bytes [Replit (2021)]	Range
`signed char`	1	-128 to 127 (a signed byte)
`short (or short int)`	2	-2^{15} to $2^{15}-1$ (-32768 to 32767)
`int`	4	-2^{31} to $2^{31}-1$ (if 4 bytes)
`long (or long int)`	8	-2^{63} to $2^{63}-1$ (if 8 bytes)
`unsigned char`	1	0 to 255 (an unsigned byte)
`unsigned short`	2	0 to $2^{16}-1$ (0 to 65535)
`unsigned int` `(or unsigned)`	4	0 to $2^{32}-1$ (if 4 bytes)
`unsigned long`	8	0 to $2^{64}-1$ (if 8 bytes)

Why are there so many different types of integers?

2.2. Variations of Integral Types

- What is the appropriate type to represent integers in this program?
  - When the amount of data to be processed is large and the memory space is scarce, we have to be *mean*.

- For now it suffices to know that these variations of integral types exist. For most applications, using `int` is adequate.

2.3. Integer Overflow

- *Integer overflow* occurs when the result of an arithmetic operation is too large to be represented by the underlying integer representation.

- e.g.: assume integers are 32 bits in size
  - Add one to the largest signed positive integer:

    2147483647 + 1 ☾ –2147483648

  - The correct result +2147483648 is NOT representable in 32-bit signed integer representation, i.e. out of range.
    - We have to use another data type to represent such a number

3. Floating Point Numbers

Key Concepts

- Floating point numbers representation and arithmetic are not exact.

- [Advanced and Optional] Further Reading and Reference

  IEEE 754 Standard for Floating-Point Arithmetic, Wikipedia (https://en.wikipedia.org/wiki/IEEE_754)

  IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic (https://ieeexplore.ieee.org/document/8766229)

3.1. Floating Point Number Representation

- Floating point numbers and integers have different representations.
- Not all real numbers are representable.
  - Finite number of bits vs. infinitely many real numbers
  - Decimal to/from binary conversion error, e.g. convert 0.1 to binary
- Floating-point number representation and arithmetic operations need not be exact.

  e.g., `printf("%.20f; %.20f", 3.3, 2.1 - 2.0 - 0.1);`

  yields `3.29999999999999982236; 0.00000000000000008327`
- For very large computations, rounding errors may accumulate and become significant.

3.2. C Language Floating-Point Types

Type	Size [Replit (2021)]	Range and Precision
`float`	4 bytes 32-bit *single*-precision	Range: ±3.4 x $10^{\pm 38}$ Precision: 6 significant decimal places
`double`	8 bytes 64-bit *double*-precision	Range: ±1.7 x $10^{\pm 308}$ Precision: 15 significant decimal places
`long double`	≥ 8 bytes [16 bytes]	System dependent; *may provide* wider range and more precision

- 123.451234512345 is more precise than 123.451
- **For most applications, using `double` is recommended.**
- If possible, avoid using `float` which is imprecise in modern day standard, thus leading to loss of precision.

3.3. Using floating point with `printf()`

- Remember our example last time?

```c
#include <stdio.h>

int main(void)
{
  double pi = 3.1415927;


  printf("A) %f\n", pi);
  printf("B) %.2f\n", pi)
  printf("C) %.7f\n", pi)

  return 0;
}

```

```
A) 3.141593
B) 3.14
C) 3.1415927
```

The format specifier, %.xf, tells `printf()` to format the corresponding floating point number with x decimal places.

3.3. Using floating point with `printf()`

- You should now realize that integers and floating point numbers are completely different things in C language
- You must specify correctly what data type you are supplying or you will see meaningless results
- What do you think the result will be in the following example?

e.g. `printf("%d", 3.14);`

3.4. Using floating point with `scanf()`

```c
#include <stdio.h>

int main(void)
{
  double r1, r2;

  printf("Enter two real numbers:\n");
  scanf("%lf%lf", &r1, &r2

  printf("r1 = %f\n", r1);
  printf("r2 = %f\n", r2);

  return 0;
}
```

Variables **r1** and **r2** are of type `double`

Concerning `double`-typed values,
`scanf()` uses `%lf` ('ell' f);
`printf()` uses `%f`.
Look similar but *slightly different*!

```
Enter two real numbers:
123 456.125↵
r1 = 123.000000
r2 = 456.125000
```

`printf()`, by default, prints decimal numbers (floating point numbers) with 6 decimal places.

4. Type Conversion

<u>Key Concepts</u>

- How types are converted in an expression with mixed types of numbers

  e.g., `2.5 + 5 / 2 = ?`

- How a `double` type value is converted to an integral type value

- Explicit Type Conversion (Type Casting)

4.1. Expressions with mixed types of data

- HK$1000 + US$100 = ?
- `3.1 + 2` = ?
- `double d = 4;` What value will **d** hold?
- `int x = 4.1;` What value will **x** hold?

- Some kind of conversion is needed to ensure the type of both operands are compatible before the computer can evaluate the expressions.

4.1. Implicit Type Conversion

- C language has a set of conversion rules to resolve certain mismatched operand types.

- As a convenient to programmers, compilers automatically convert the value of an operand from one type to another based on these rules whenever possible.

- Sometimes called *coercion*.

4.1. Arithmetic Conversions (Simplified Rules)

- If either operand is a `double`, the other is converted to `double`. The result type is also `double`.

  e.g.:

  ```
    3.1 + 2     (3.1 is of type double)
  = 3.1 + 2.0   (Therefore, 2 is converted to 2.0)
  = 5.1         (Result is of type double)
  ```

- If both operands are of one of the integral types `char`, `short` and `int`, then both operands are converted to `int`. The result type is also `int`.

4.2. Converting Integral Type to `double`

- Converting integral type to `double` is safe.
  - No warning is given at compile time

```
double d;
d = 4;
   /* 4 is converted to 4.0,
            and then 4.0 is assigned to d */
```

4.2. Converting `double` to Integral Type

- Converting a `double` to an integral type may result in *loss of data*.
  - If the number is within the range of the integral type, the fractional part is *truncated*, i.e. *discarded*.
  - Compilers *usually* warn at compile time (but NOT guaranteed.)

```
int x = 4.1;    /* 4.1 is converted to 4 */
x = 2 * 4.1;    /* 8.2 is converted to 8 */
x = 92345678901.2;  /* behaviour is undefined
                         when the value is too
                         large to hold in x */
```

4.3. Explicit Type Conversion (Casting)

Syntax: `(new_type) operand`

- Converts the *value of* `operand` to the equivalent value of type `new_type`.
  - `(new_type)` is called the type casting operator
  - Note that not every type conversion is possible, however.

e.g.,

```
double d = 4.2;
int y = (int) d;  // y becomes 4, no warning
int x = d;        // x becomes 4, compiler
warns
// because of missing type casting
```

4.4. Type Conversion (Examples)

```
int x = 5, y = 2;
double a, b;
a = 2.5 + (x / y);           /*    R.H.S. is evaluated
as
                                   2.5 + (5 / 2)
                                => 2.5 + 2
                                => 2.5 + 2.0
                                => 4.5
                             */

b = 2.5 + ((double)x / y);   /*    R.H.S. is evaluated
as
                                   2.5 + (5.0 / 2)
                                => 2.5 + (5.0 / 2.0)
                                => 2.5 + 2.5
                                => 5.0
                             */
```

Example 4.1. Expression with mixed data types

4.4. Type Conversion (Examples)

```c
int x, y;
double a = 2.6, b = 2.4;
x = (int)(a + 0.5);       // x is assigned 3
y = (int)(b + 0.5);       // y is assigned 2
```

Example 4.2. Rounding floating point numbers to nearest integer

4.4. Using Type Casting Operators (Exercise)

- Average of *N* integers

```
//  Consider the following declaration
int total, N;
double avg;
…
//  Suppose we have obtained the value of N and
// calculated the total of  N integers.
//  Which of these will correctly calculate the average?
avg = total / N;              // A
avg = (double)total / N;   // B
avg = total / (double)N;   // C
avg = (double)(total / N);   // D
avg = (double)total / (double)N;  // E
```

4.5. How are numbers converted?

(Apply to both implicit and explicit conversions)

- `double` to integral types
  - Only retain the integer part (no rounding)
  - e.g.:

```
    int x = (int) 4.9;          // x gets 4
    int y = (int) -3.99;    // y gets -3
```

- "Larger" integral types to "smaller" integral types
  - Retain only the ***least significant bits (LS-bit)***
  - e.g.: 32-bit integer (`int`) to 16-bit integer (`short`)

$00000000000000100000000000000011 = 131075_{10}$

$0000000000000011 = 3_{10}$

```
short x = (short)131075;      // x becomes 3
```

5. Back to the Original Example

- Given your understanding of data types now, what is wrong with our original example?

```c
#include <stdio.h>

int main(void)
{
  int area, h, b;

  h = 3;
  b = 4;
  printf("Height and base of triangle: %d, %d\n", h,
b);
  area = (1/2) * h * b;
  printf("Area of triangle: %d\n", area);

  return 0;
}
```

5. Back to the Original Example

- Given your understanding of data types now, what is wrong with our original example?

```c
#include <stdio.h>

int main(void)
{
   int area, h, b;

   h = 3;
   b = 4;
   printf("Height and base of triangle: %d, %d\n", h,
b);
   area = (1/2) * h * b;
   printf("Area of triangle: %d\n", area);

   return 0;
}
```

Are we using the proper type? Are these variables always integers?

How about this? What are we telling `printf`?

Does 1/2 give integer as result, or floating point? Should we use, say, 0.5 instead?

Summary

- All number types have an intended purpose, precision, and range.
  - Choose a proper data type to represent data
  - Beware of and prevent overflow

- Floating-point representation and arithmetic may not be exact.

- Expressions with mixed types of data
  - Automatic and explicit type conversion
  - Number conversion (`double` to `int`)

Appendix: Finding out the size of an integer

```c
#include <stdio.h>

int main(void) {
    printf("size of int = %d bytes\n",
(int) sizeof(int) );
    return  0;
}
```

```
size of int = 4 bytes
```

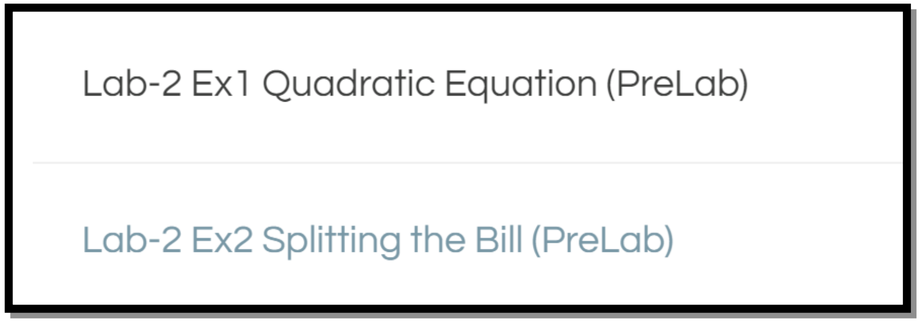
- `sizeof(`*`data_type`*`)` yields the number of bytes used to represent a value of type *`data_type`* (as unsigned long).
- `(int)` explicitly converts the value to int.

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Appendix C Number Systems

Reminder: PreLabs are Ready!

- Every Mon afternoon we will release the **PreLabs**
  - Meant to help you prepare for the lab
  - Due **Wed 9:30am** – Please try it after the lecture and submit before Wed!
  - Don't worry – it's super easy (takes < 30 min) and it's very easy marks to get! Don't forget!


Lab-2 Ex1 Quadratic Equation (PreLab)

Lab-2 Ex2 Splitting the Bill (PreLab)

PreLabs are marked "(PreLab)" on repl.it