Fundamental Data Types

- derived from the slides of the book “C++ For Everyone” by Cay Horstmann
- slides by Evan Gallagher

Goals

- define and initialize variables and constants
- understand the properties and limitations of integer and floating-point numbers
- write arithmetic expressions and assignment statements
- appreciate the importance of comments and good code layout
- create programs that read and process input, and display the results

Variables

- variable: used to store information
- can contain one piece of information at a time
- has an identifier

Variable Analogy

- parking garages

  store cars

- each parking space is identified (J-053)
- like a variable’s identifier

- each parking space “contains” a car
- like a variable’s current contents

Variable Analogy

- each space can contain only one car
- and only cars, not buses or trucks

Variable Definitions

- when creating variables, programmer specifies:

- identifier (name)
- type of information to be stored
- initial value (optional)

Variable Definition Example

```
int cans_per_pack = 6;
```

- `cans_per_pack`: variable’s name

- `int`: variable’s type
- indicates that the variable will hold integers

- 6: initial value

Name Rules

- can contain only letters, numbers, or underscores
- must start with letter or underscore
- no other symbols such as `$` or `%`
- spaces are not permitted

- you cannot use reserved words such as `return`

Case Sensitivity

- names are case-sensitive
- it is a good idea to use only lowercase letters

Name Examples

- `can_volume1`: valid
- `x`: valid
- `Can_volume`: valid but not same as `can_volume`

- `6pack`: invalid, starts with digit
- `can volume`: invalid, contains space
- `int`: invalid, reserved
- `ltr/fl.oz`: invalid, contains `/` and `.`

Good Names

- pick a name that explains the purpose

- pick a descriptive name, such as `can_volume`
- rather than a terse name, such as `cv`

Comments

- explanations for human readers of the code
- compiler ignores comments completely

```
double can_volume = 0.355;  // liters in a 12-ounce can
```

- compiler ignores everything after // to the end of line

Multiline Comments

- multiline for longer comments
- between /* and */
- not nestable

```
/*
   This program computes the volume (in liters)
   of a six-pack of soda cans.
*/

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

Numeric Types

- integer: `int`
- floating point: `double`

- floating point with lower precision: `float`
- rarely used

Value Ranges

- variants on integers: `short`, `long`
- `unsigned` specifier
- these can represent different ranges of values
- not set by the C standard

- floating point with higher precision: `long double`

Initial Values

- initialization is not required

```
int bottles;
```

- causes a warning
- giving initial values are recommended

Initial Values

- initial value can contain previously defined variables

```
int cans = 6;
int bottles = 10;
int total = cans + bottles;
```

Multiple Definitions

- multiple variables can be defined in the same statement

```
int cans, bottles;
```

```
int cans = 6, bottles = 10;
```

Numeric Literals

- type inferred from value

`int`

- `6`
- `-6`
- `0`

`double`

- `0.5`
- `1.0`
- `1E6`
- `2.96E-2`

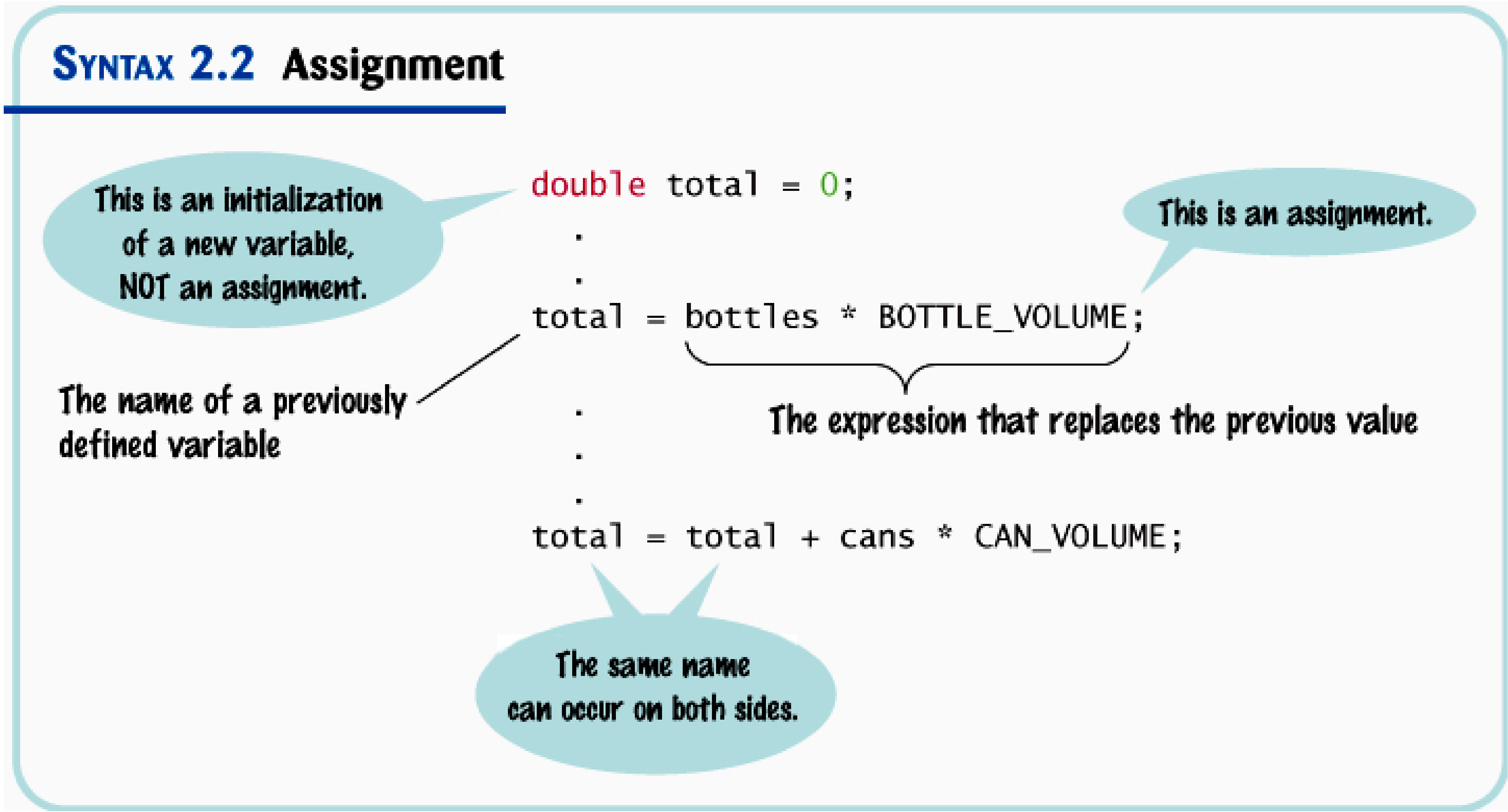
Assignment

- contents in variables can “vary” over time

- assignment statement
- stores a new value in a variable
- replace the previously stored value

```
cans_per_pack = 8;
```

Assignment Example



Syntax 2.2 Assignment

```
double total = 0;
   .
   .
total = bottles * BOTTLE_VOLUME;
   .
   .
   .
total = total + cans * CAN_VOLUME;
```

This is an initialization of a new variable, NOT an assignment.

This is an assignment.

The name of a previously defined variable

The expression that replaces the previous value

The same name can occur on both sides.

Definition and Assignment

- difference between variable definition and assignment

```
1  int cans_per_pack = 6;
2  ...
3  cans_per_pack = 8;
```

- line 1: definition of `cans_per_pack`
- line 3: assignment - **existing** variable’s contents are replaced

Assignment and Equality

- The = in an assignment does not mean that the left hand side is equal to the right hand side as it does in math.
- It’s an instruction to do something.
- Copy the value of the expression on the right into the variable on the left.

Assignment and Equality

- what would the below statement mean, mathematically?

```
counter = counter + 2;
```

- counter EQUALS counter + 2?

Assignment and Equality Example

```
counter = 11;
```

```
counter = counter + 2;
```

1. look up what is currently in counter (`11`)
2. add 2 to that value (`13`)
3. copy the result of the addition expression into the variable on the left, changing counter

Type Checking

- what if type of variable and initial value don’t match?

```
int bottles = "10";
```

- not an error, only a warning

Assigning Doubles to Integers

- when a floating-point value is assigned to an integer variable, the fractional part is discarded:

```
double price = 2.55;
int dollars = price;
```

- sets `dollars` to 2

Printing Numbers

- `printf` function: format string
- %d: decimal (`int`)
- `%f`: floating point (`double`)

Printing Example

```c
#include <stdio.h>  // printf

int main()
{
    double radius = 2.5;
    double pi = 3.14159;
    double area = radius * pi * pi;
    printf("Area: %f\n", area);
    return 0;
}
```

Formatted Output

- controlling the number of digits
- e.g. displaying money values

- round off only for display
- `%.2f`: two digits after decimal point
- `%8.2f`: eight in total (including decimal point)

Formatted Output Example

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

int main()
{
    double radius = 2.5;
    double pi = 3.14159;
    double area = pi * radius * radius;
    printf("Area: %.3f\n", area);
    return 0;
}
```

Formatted Output Example

```c
double price_per_ounce_1 = 10.2372;
double price_per_ounce_2 = 117.2;
double price_per_ounce_3 = 6.9923435;

printf("%8.2f\n", price_per_ounce_1);
printf("%8.2f\n", price_per_ounce_2);
printf("%8.2f\n", price_per_ounce_3);
printf("--------\n");
```

```
   10.24
  117.20
    6.99
--------
```

Input

- sometimes the programmer doesn’t know what should be stored in a variable, but the user does

- get the value from the user

- show a prompt (output)
- read from the keyboard (input)

Input Function

- to read values from the keyboard,
  call the `scanf` function

- include `stdio.h`

- format specifiers:
  - %d: decimal (`int`)
  - `%lf`: floating point (`double`)

Input Variable

- input value will be stored in a variable
- variable has to be defined earlier
- & in front of variable name

```c
int bottles = 0;
printf("Enter the number of bottles: ");
scanf("%d", &bottles);
```

Multiple Input

- you can read more than one value in a single input

```
int bottles = 0, cans = 0;
printf("Enter the number of bottles and cans: ");
scanf("%d %d", &bottles, &cans);
```

Multiple Input

- the user can suppy both values on the same line:

```
Enter the number of bottles and cans: 2 6
```

- alternatively, the user can press the `Enter` key after each input:

```
Enter the number of bottles and cans: 2
6
```

Constants

- some values are not meant to change
- `const`: define a constant
- convention: use all capital letters for name

```
const double PI = 3.14159;
```

- value given at initialization
- assigning to a constant causes an error

Constant Example

```c
#include <stdio.h>  // printf, scanf

int main()
{
    const double PI = 3.14159;

    double radius = 0.0;
    printf("Enter the radius: ");
    scanf("%lf", &radius);

    double area = PI * radius * radius;
    printf("Area: %.3f\n", area);

    return 0;
}
```

Constants

- second way to define constants: `#define`

```
#define PI 3.14159
```

Constant Example

```c
#include <stdio.h>  // printf

#define PI 3.14159

int main()
{
    double radius = 0.0;
    printf("Enter the radius: ");
    scanf("%lf", &radius);

    double area = PI * radius * radius;
    printf("Area: %.3f\n", area);

    return 0;
}
```

Constants

- another good reason for using constants:

```
int volume = bottles * 2;
```

- what does that 2 mean?

```
const int BOTTLE_VOLUME = 2;
int volume = bottles * BOTTLE_VOLUME;
```

Constants Example

```
int bottle_volume = bottles * 2;
int can_volume = cans * 2;
```

- what does that 2 mean?
- which 2?

- these are called magic numbers
- use constants for them

Constants

- it can get even worse

- suppose that the number 2 appears hundreds of times
- throughout a five-hundred-line program

- we need to change the bottle volume to 3
- because we are now using a different bottle

- how to change only some of those 2’s?

Constants Example

```
const double BOTTLE_VOLUME = 2.23;
const double CAN_VOLUME = 2;

...

double bottle_volume = bottles * BOTTLE_VOLUME;
double can_volume = cans * CAN_VOLUME;
```

Constants Example

```
int main()
{
    int cans_per_pack = 6;
    const double CAN_VOLUME = 0.355;  // liters in a 12-ounce ca
    double total_volume = cans_per_pack * CAN_VOLUME;
    printf("A six-pack of 12-ounce cans contains %f liters.\n",
           total_volume);

    const double BOTTLE_VOLUME = 2;
    total_volume = total_volume + BOTTLE_VOLUME;
    printf("A six-pack and a two-liter bottle contain %f liters.
           total_volume);
}
```

Program Result

```c
#include <stdio.h>  // printf

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

- what does that 0 mean?

Program Result

```c
#include <stdio.h>   // printf
#include <stdlib.h>  // EXIT_SUCCESS

int main()
{
    printf("Hello, world!\n");
    return EXIT_SUCCESS;
}
```

A Complete Program for Volumes

```
#include <stdio.h>   // printf, scanf
#include <stdlib.h>  // EXIT_SUCCESS

int main()
{
    const double CANS_PER_PACK = 6;

    // Read price per pack
    // Read can volume
    // Compute pack volume
    // Compute and print price per ounce

    return EXIT_SUCCESS;
}
```

A Complete Program for Volumes

```
// Read price per pack
double pack_price = 0.0;
printf("Please enter the price for a six-pack: ");
scanf("%lf", &pack_price);

// Read can volume
double can_volume = 0.0;
printf("Please enter the volume for each can (in ounces): ");
scanf("%lf", &can_volume);

// Compute pack volume
double pack_volume = can_volume * CANS_PER_PACK;

// Compute and print price per ounce
double price_per_ounce = pack_price / pack_volume;
printf("Price per ounce: %f\n", price_per_ounce);
```

Using Undefined Variables

- you must define a variable before you use it for the first time

```
double can_volume = 12 * liter_per_ounce;
double liter_per_ounce = 0.0296;
```

- statements are compiled in top to bottom order
- compiler reports error on first line

Using Uninitialized Variables

- there is always a value in every variable
- even uninitialized ones

```
int bottles;
double bottle_volume = bottles * 2.0;
```

- result of second line is unpredictable
- compiler issues a warning

Arithmetic Operators

- addition: +
- subtraction: -
- multiplication: *
- division: /

- between integers:
- /: integer division
- %: remainder (modulus)

Precedence

- regular precedence from arithmetic
- `*` and `/` have higher precedence than `+` and `-`

- use parentheses to adjust precedence
- and to improve readability

Integer Division Example

- given a duration in minutes

  determine how many hours and minutes

```c
int duration = 0;
printf("Enter duration [min]: ");
scanf("%d", &duration);

int hours = duration / MINUTES_IN_HOUR;
int minutes = duration % MINUTES_IN_HOUR;
printf("%d hours and %d minutes\n", hours, minutes);
```

Unintended Integer Division

- `7 / 4` is `1`, not `1.75`

- but

  `7.0 / 4.0`

  `7 / 4.0`

  `7.0 / 4`

  all yield `1.75`

Common Error

```
1  printf("Please enter your last three test scores: ");
2
3  int s1, s2, s3;
4  scanf("%d %d %d", &s1, &s2, &s3);
5
6  double average = (s1 + s2 + s3) / 3;
7  printf("Your average score is %f\n", average);
```

- line 6: integer division
- `s1 + s2 + s3` is an integer
- if scores add up to 14, average is computed as `4.0`

Common Error

- fix:

```
double total = s1 + s2 + s3;
double average = total / 3;
```

- or:

```
double average = (s1 + s2 + s3) / 3.0;
```

Spaces in Expressions

- style convention:

  leave one space on both sides of an operator

```
x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);
```

```
x1=(-b+sqrt(b*b-4*a*c))/(2*a);
```

Spaces in Expressions

- not after a unary minus

```
-b
```

- not:

```
-  b
```

Increment and Decrement

- incrementing/decrementing by 1 are very common operations
- special shorthand for these

```
counter++;  // add 1 to counter
counter--;  // subtract 1 from counter
```

Increment and Decrement

- also works in front of variable name

```
++counter;  // add 1 to counter
--counter;  // subtract 1 from counter
```

Augmented Assignment

- assignment can be combined with arithmetic operations

```
counter += 2;
// shortcut for: counter = counter + 2;

total *= 2;
// shortcut for: total = total * 2;

total += cans * CAN_VOLUME;
// shortcut for: total = total + cans * CAN_VOLUME;
```

Exponentiation

- no built-in exponentiation operator
- how to compute the below?

$$b \cdot (1 + \frac{r}{100})^n$$

Mathematical Functions

- standard library contains common math functions
- include `math.h`
- `sqrt(x)`
- `pow(base, exponent)`

Exponentiaton Example

```c
#include <stdio.h>   // printf, scanf
#include <stdlib.h>  // EXIT_SUCCESS
#include <math.h>    // pow

int main()
{
    // Read initial amount (double)
    // Read interest rate (double)
    // Read number of years (int)

    double final = initial * pow(1 + (rate / 100), years);
    printf("Final amount: %.2f\n", final);

    return EXIT_SUCCESS;
}
```

Spaces in Expressions

- style convention:

  no space before function opening parenthesis

```
sqrt(x)
```

- not:

```
sqrt (x)
```

Linking Libraries

- math library has to be included when linking:

```
gcc -c -std=c99 -Werror interest.c -o interest.o
gcc interest.o -lm -o interest
```

- or, in one step:

```
gcc -std=c99 -Werror interest.c -lm -o interest
```

Forgetting to Include the Header

- “include” is for the compiler
- forgetting the header line

  causes to leave the name pow undefined
- compiler warning, linker error

Overflow Errors

- if a computation that is outside the type’s range an overflow occurs
- no error is displayed
- the result is truncated

Overflow Example

```c
int one_billion = 1E9;
printf("%d\n", 3 * one_billion);
```

- prints `-1294967296`
- because the result is larger than an `int` can hold
- use `double` instead

Roundoff Errors

- floating point values cannot be represented accurately

```
double price = 4.35;

int cents = 100 * price;
// should be 100 * 4.35 = 435

printf("cent is %d \n", cents);
// prints 434
```

Roundoff Errors

- in binary, no exact representation for 4.35
- value is just a little less than 4.35
- 100 times that value is just a little less than 435
- assignment to int truncates

- remedy:

```
int cents = 100 * price + 0.5;
```

Type Conversions

- conversion from one type (such as double) to another type (such as int): cast

- not safe in general
- but if you know it to be safe in a particular circumstance, casting is the only way

- syntax:

```
(type_name) expression
```

Converting Doubles to Ints

- you are prepared to lose the fractional part
- you know that this particular floating point number is not larger than the largest possible integer

Casting Example

```c
double change = 99.98;
int cents = (int) (10 * change + 0.5);

printf("%d\n", (int) change);  // 99
printf("%d\n", cents);         // 1000
```