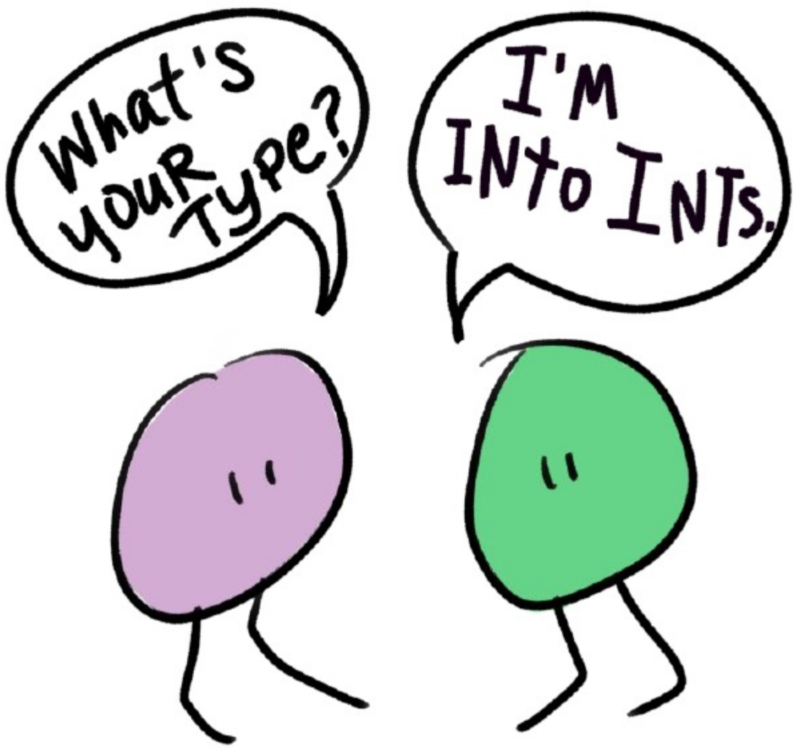


Variables and Arithmetic

Due this week

- **Recitation 1**
  - Write pseudocode
- **Homework 1**
  - Submit pdf file on Canvas. PDF
- Check the due date! **No late submissions!!**

Today

- Variables
- Arithmetic

Variables

Variables

A **variable:**

- is used to **store** information (the **value**/**contents** of the variable)
  - can contain one piece of information at a time.
- has an **identifier** (the name of the variable)

- The programmer picks a good name
  - A good name describes the contents of the variable or what the variable will be used for
  - has a **type** (more about this very soon)

Variables: Like a parking garage

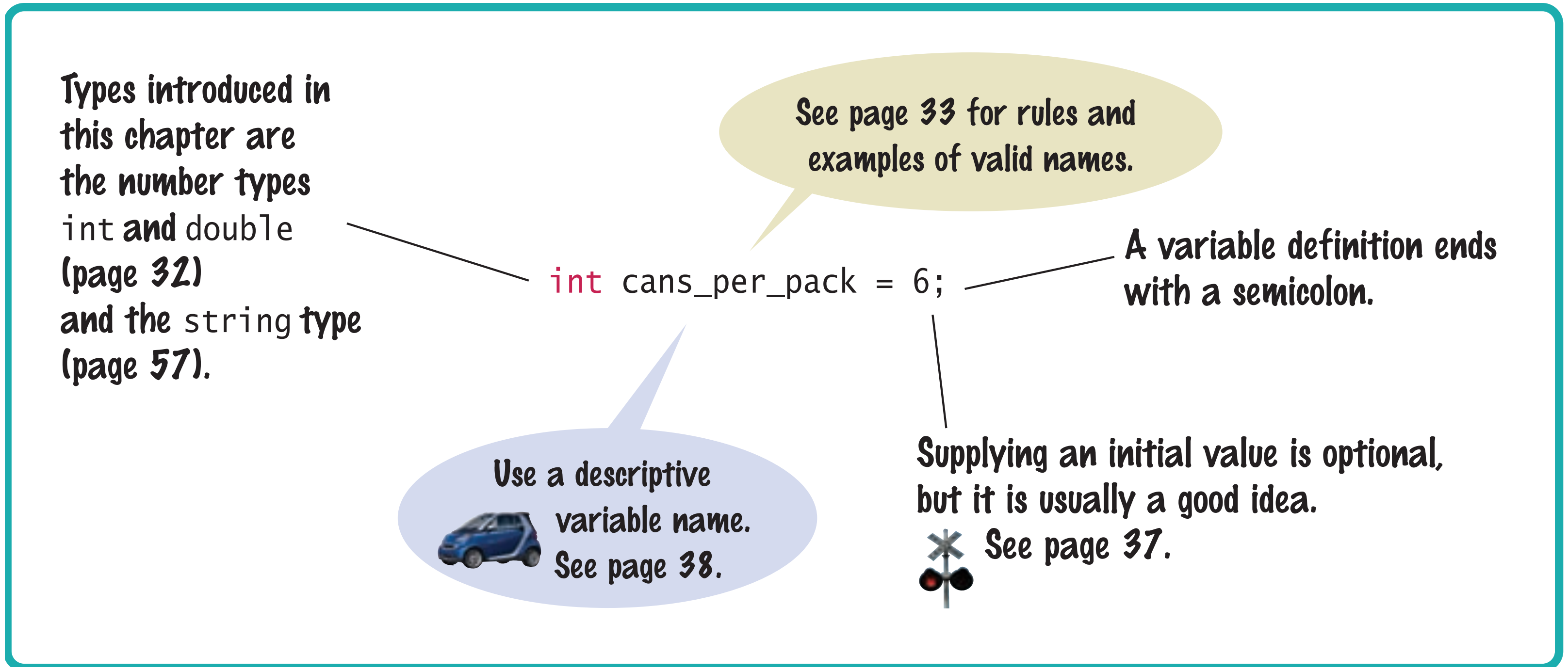
- Parking garages store cars.

- Each parking space is identified
  – like a variable’s identifier
- Each parking space “contains” a car
  – like a variable’s current contents
- Each space can contain only one car
- and not trucks or buses, just a car

Variable Definitions

- When creating variables, the programmer specifies the **type** of information to be stored.

- Unlike a parking space, a variable is often given an initial value.
    - ***Initialization*** is putting a value into a variable when the variable is created.
    - Initialization is **not required**.



Types introduced in this chapter are the number types `int` and `double` (page 32) and the `string` type (page 57).

See page 33 for rules and examples of valid names.

```
int cans_per_pack = 6;
```

A variable definition ends with a semicolon.

Use a descriptive variable name. See page 38.

Supplying an initial value is optional, but it is usually a good idea. See page 37.

Variable Definitions

Variable Definitions: example

The following statement defines a variable:

```
int cans_per_pack = 6;
```

`cans_per_pack` is the variable's name.

`int` indicates that the variable **cans_per_pack** will hold integers. Other variable types covered later will hold *strings* and *floating-point numbers*.

`= 6` indicates that the variable **`cans_per_pack`** will initially contain the value 6.

Like all statements, it must end with a semicolon.

The Assignment Statement

- The contents in variables can “vary” over time (hence the name!).
- Variables can be changed by
    - assigning to them
        - **The assignment statement** (”=“)
    - using the increment or decrement operator (++, --)
    - inputting into them
        - The input statement (“cin”)

Assignment Statement Example

- An *assignment statement* stores a new value in a variable, replacing the previously stored value.

```
cans_per_pack = 8;
```

- This assignment statement changes the value stored in **`cans_per_pack`** to be 8.

- The previous value is replaced.

The Meaning of the Assignment = Symbol

- The = in an assignment does ***not*** mean the left hand side is equal to the right hand side as it does in math.
- = is an instruction to do something:

  copy the value of the expression on the right
  into the variable on the left.

- Consider what it would mean, mathematically, to state:

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

counter *EQUALS* counter + 2

Assignment Statement: defining vs. assigning

- There is an important difference between a variable definition and an assignment statement:

```
int cans_per_pack = 6; // Variable definition
...
cans_per_pack = 8; // Assignment statement
```

- The first statement is the *definition* of **`cans_per_pack`**.
- The second statement is an *assignment statement.*
  - An *existing* variable's contents are replaced.
- A variable's definition must occur ***<u>only once</u>*** in a program. The same variable may be in several assignment statements in a program.

Assignment Examples

```
counter = 11; // set counter to 11
counter = counter + 2; // increment
```

1. First statement assigns 11 to counter
2. Second statement looks up what is currently in the variable counter (11)
3. Then it adds 2 and copies the result of the addition
   into the variable on the left, changing counter to 13

Variable Definitions: more examples

Table 1: Variable Definitions in C++

	Comment
int cans = 6;	Defines an integer variable and initializes it with 6.
int total = cans + bottles;	The initial value need not be a constant. (Of course, cans and bottles must have been previously defined.)
int bottles = "10";	Error: You cannot initialize an int variable with a string.
int bottles;	Defines an integer variable without initializing it. This can be a cause for errors—see Common Error 2.2.
int cans, bottles;	Defines two integer variables in a single statement. In this book, we will define each variable in a separate statement.
bottles = 1;	Caution: The type is missing. This statement is not a definition but an assignment of a new value to an existing variable—see Section 2.1.4.

Table 2: Number Literals

	Type	Comment
6	int	An integer has no fractional part.
–6	int	Integers can be negative.
0	int	Zero is an integer.
0.5	double	A number with a fractional part has type double.
1.0	double	An integer with a fractional part .0 has type double.
1E6	double	A number in exponential notation: 1 × 106 or 1000000. Numbers in exponential notation always have type double.
2.96E-2	double	Negative exponent: 2.96 × 10–2 = 2.96 / 100 = 0.0296
100,000		Error: Do not use a comma as a decimal separator.
3 1/2		Error: Do not use fractions; use decimal notation: 3.5.

Table 3: Variable Names

Variable Name	Comment
can_volume1	Variable names consist of letters, numbers, and the underscore character.
x	In mathematics, you use short variable names such as x or y. This is legal in C++, but not very common, because it can make programs harder to understand (see Programming Tip 2.1)
Can_volume	Caution: Variable names are case sensitive. This variable name is different from can_volume.
6pack	Error: Variable names cannot start with a number.
can volume	Error: Variable names cannot contain spaces.
double	Error: You cannot use a reserved word as a variable name.
ltr/fl.oz	Error: You cannot use symbols such as . or /

Constants

- Sometimes the programmer knows certain values just from analyzing the problem
  - For this kind of information, use the reserved word **`const`**.
- The reserved word **`const`** is used to define a constant.
- A **`const`** is a "variable" whose contents cannot be changed and must be set when created.
  (Most programmers just call them constants, not variables.)
- Constants are commonly written using capital letters to distinguish them visually from regular variables:

```
const double BOTTLE_VOLUME = 2;
```

Constants Prevent Unclear Numbers in Code

Another good reason for using constants:

```
    double volume = bottles * 2;
```

What does that 2 mean?

If we use a constant there is no question:

```
    double volume = bottles * BOTTLE_VOLUME;
```

Constants Prevent Unclear Numbers in Code (2)

And still another good reason for using constants:

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

What does *that* 2 mean?

— *WHICH 2?*

It is not good programming practice to use magic numbers.

Use **constants**.

Constants Prevent Unclear Numbers in Code (3)

And it can get even worse …

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

Now we need to change the BOTTLE_VOLUME to 2.23 (because we are now using a bottle with a different shape)

How to change ***only*** some of those 2’s?

Constants again

Constants to the rescue!

```
const double BOTTLE_VOLUME = 2.23;
const double CAN_VOLUME = 2;
...
double bottle_volume = bottles * BOTTLE_VOLUME;
double can_volume = cans * CAN_VOLUME;
```

Comments

- *Comments* are explanations for human readers of your code (other programmers or your instructor).
- The compiler ignores comments completely.
- A leading double slash // tells the compiler the remainder of this line is a comment, to be ignored
- For example,

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

Comments: // or /*   multi-line */

Comments can be written in two styles:

- Single line:

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

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

- Multiline for longer comments, where the compiler ignores everything between /*   and   */

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

Common Error: Using Undefined Variables

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

For example, the following sequence of statements would not be legal:

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

Statements are compiled in top to bottom order.

When the compiler reaches the first statement, it does not know that **`liter_per_ounce`** will be defined in the next line, and it reports an error.

Common Error: Using Uninitialized Variables

- Initializing a variable is not required, but there is always a value in every variable, even uninitialized ones.
- Some value will be there, left over from some previous calculation or simply the random value there when the transistors in RAM were first turned on.

```
int bottles; // Forgot to initialize
int bottle_volume = bottles * 2;
```

What value would be output from the following statement?

```
cout << bottle_volume << endl;
```