Introduction to Computing and Algorithms

Basic Data, Operations and Decisions

Chapter 3

Basic Tools for Building Algorithms

- Atomic Data
  - Numbers
  - Characters
  - Booleans
  - Pointers
- Operators
  - Assignment
  - Arithmetic
  - Input / Output
  - Relational
  - Boolean

Data

Data Structures are containers for data.

The simplest of them are called "**atomic**" because they hold only a single value; the data they hold cannot be subdivided into lower-level components.

"**Complex**" data structures may hold multiple pieces of data, are constructed from atomic types.

Four atomic types:

- Character
- Number
- Boolean
- Pointer

One built-in complex type:

- String

Built-in Data Types

Variables are data structures whose value *may be changed* by the algorithm.

- Each variable must be *declared* to
  - have an *identifier* (i.e., a *name*)
  - be of some *data type*:

  For example: to declare a variable named "some number" that can hold a numeric value:

  ```
  some_number isoftype Num
  ```

- Each variable must be *assigned* a value of the appropriate type:

  For example: to assign the value *4* to the variable *some_number*:

  ```
  some_number <- 4
  ```

Built-in Data Types

- ***Numbers***, declared as:

```
my_age isoftype Num
```

my_age can now store one number of any magnitude

For example:

```
my_age <- 1      // this is OK
my_age <- 65     // so is this
my_age <- 14.5   // and so is this

my_age <- "old" // an ERROR
```

Built-in Data Types (cont.)

- ***Characters***, declared as:

```
your_grade isoftype Char
```

your_grade can now store any one alphanumeric character single-quote marks are used to distinguish Char values from identifiers.

For example:

```
your_grade <- 'A'  // this is OK
your_grade <- 'B'  // so is this

your_grade <- 'A-' // an ERROR


your_grade <- '9'  // this is OK
your_grade <- 9    // an ERROR
your_grade <- '10' // an ERROR
```

Built-in Data Types (cont.)

- ***Booleans***, declared as:

```
this_test isoftype Boolean
```

this_test can now hold one boolean (*TRUE* or *FALSE*) value

For example:

```
this_test <- TRUE        //  this is OK
this_test <- FALSE       //  so is this

this_test <- "true"      //  an ERROR
```

Built-in Data Types (cont.)

- ***Pointers***, declared as:

```
this_num_ptr isoftype ptr toa Num
```

`this_num_ptr` can now "know" where a *Num* variable "lives" in memory

(will discuss pointers later)

Built-in Data Types (cont.)

- ***String*** is a *built-in complex* (non-atomic) type which can holds a collection of characters.

  A string is declared as:

  this_string isoftype String

  this_string can now hold a "string" of any number of alphanumeric characters

For example:

this_string <- "a group of Chars" // this is OK
this_string <- a group of Chars // an ERROR

this_string <- "45" // this is OK
this_string <- 45 // an ERROR

The Assignment Operator

The Assignment Operator stores a particular value in a variable.

```
my_age <- 43
```

(any integer or real number)

```
your_grade <- 'A'
```

(quote marks needed to distinguish Chars from variable names)

```
this_test <- TRUE
```

(*TRUE* and *FALSE* are each *single* boolean values)

Assignment Operator (cont.)

For pointers, assignment means one of three things, depending on how used ...

1. Creating a new variable. Given:

```
this_ptr isoftype ptr toa Num,
that_ptr isoftype ptr toa Num ,
other_ptr isoftype ptr toa Num
```

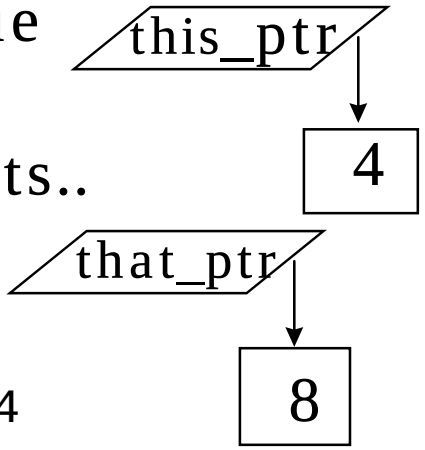
then:

```
this_ptr <- new(Num)
that_ptr <- new(Num)
```

creates a *new* Number variable for each pointer and makes the specified *pointer variable* refer to it, e.g.:

Assignment Operator (cont.)

2. Assigning a value to a variable to which a ptr points.. Given previous, then:

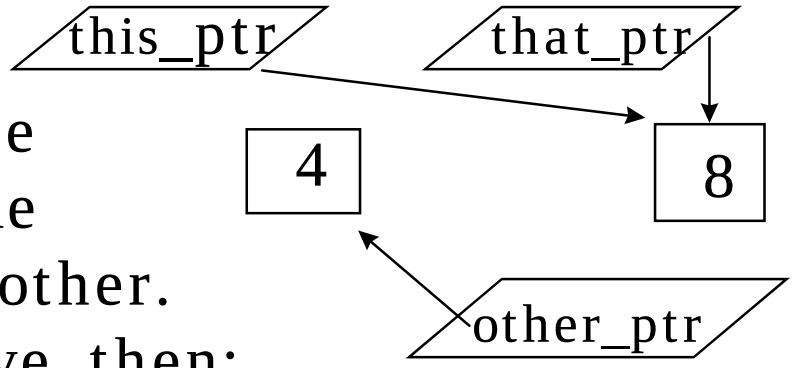



```
this_ptr^ <- 4
that_ptr^ <- 8
```

assigns those values to the Num var to which the pointer var refers.

---

3. Making one ptr variable refer to another. Given above, then:




```
other_ptr <- this_ptr
this_ptr <- that_ptr
```

assigns/reassigns pointers variables to reference other variables

Arithmetic Operators

Arithmetic Operators allow us to express mathematical operations:

- Addition +
- Subtraction -
- Multiplication *
- Division /

Using the Arithmetic Operators

- To add the values found in variables *num_one* and *num_two*, and store the result in variable *num_three*:

```
num_three <- num_one + num_two
```

Use parentheses to alter the standard order of operations:

- To add thevalues found in *num_one* and *num_two*, multiply the result by two, and store that result in variable *num_three*:

```
num_three <- (num_one + num_two) * 2
```

Input and Output Operators

I/O Operators:

allow us to communicate with the "outside world," i.e, the real world beyond the algorithm

read **-** obtains an input value

each *read* operation does 2 things:

1. obtains next value from "outside"
2. stores it in the specified variable

```
// get next value, store it in var num_one
read (num_one)
```

print **-** sends out an output value

```
// output the current value of
// variable num_two
print (num_two)

// output a literal text message
print ("Hello, World!")
```

Input and Output Operators

read and *print* may take *any number* of arguments of type Num, Boolean, Char, or String.

For example,

```
read(this_var)
read(that_var)
```

is equivalent to:

```
read (this_var, that_var)
```

- Don't worry about the exact details of the format of input or about how to format output for printing.
- We intentionally leave that vague for now. Assume that it works by magic.
- Such things are unimportant details covered in later programming courses.

A Simple Example

```
algorighm Simple_Example

   // declare the variables
   num_one,
   num_two,
   num_three isoftype Num

   // get the numbers from “outside”
   read (num_one, num_two)

   // perform addition and show result
   num_three <- num_one + num_two
   print(“The addition result is: ”, num_three)

   //perform multiplication and show result
   num_three <- num_one * num_two
   print(“This multiplication results is: ”, num_three)

endalgorithm // Simple_Example
```

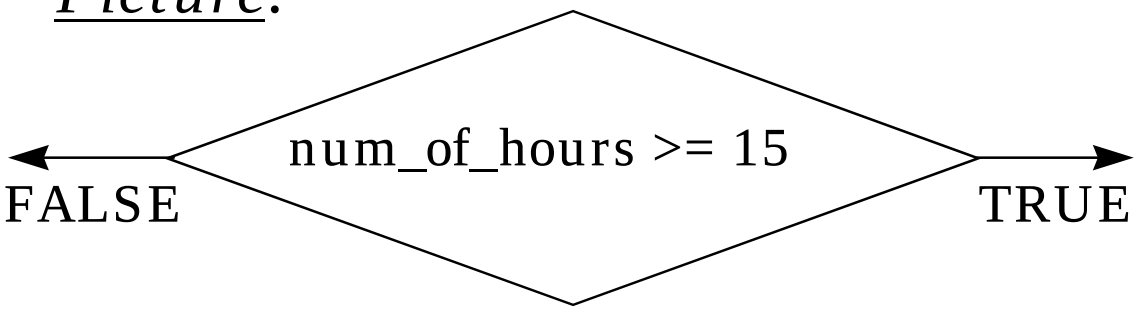
Decisions Statements (Conditionals)

Conditionals allow us to make decisions about data. Results of conditionals are always *TRUE* or *FALSE*, i.e., *boolean values*.

Natural Language:

"Is it true that the number of hours is greater than or equal to 15 ?"

Picture:



Code:

```
if (num_of_hours >= 15) then...
```

Decisions Statements (Conditionals)

Conditionals can have three parts:

- A *conditional expression*
- A *then* clause
- An (optional) *else* clause

For example:

```
if (num_one < num_two) then
   highest <- num_two
else
   highest <- num_one
endif
```

The "else" clause is optional:

```
if ( salary > MONEY_NEEDED ) then
   save some of it
   give some to charity
endif
```

Relational Operators

Conditionals produce boolean (*TRUE/FALSE*) results based on comparisons specified by the *relational operators:*

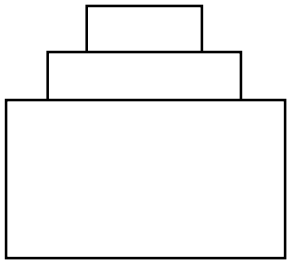
- Greater than >
- Greater or equal >=
- Equal to =
- Less than or equal <=
- Less than <
- Not equal to < >

Decisions Statements (Conditionals)

- These simple *if-then-else* constructs enable computers to do complex things.
- The difference between a *calculator* and a *computer* is "the ability to make decisions and act on them"
- *All decisions* made by *any* computer is governed by the simple mechanism of:
  - compare two values according to the specified relational operator
  - if the answer is TRUE take one course of action, else take another course of action.
- Computer-made decisions that appear to be complex are all constructed from simple *if-then-else* logic . . .

Nested Conditionals
Plain English

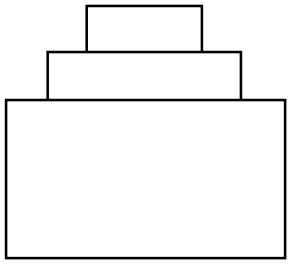
Conditionals can reside *inside* other conditionals:



check if Today is Sunday
if it is Sunday,
    if it's not Raining go to the park, otherwise stay home
if its NOT Sunday,
    go to work

Nested Conditionals Pseudocode

Conditionals can reside inside other conditionals:



```
if (Today = Sunday) then
    if (NOT Raining) then
        go to the park
    else
        stay home
    endif
else
   go to work
endif
```

- Note use indentation and *endif*s to make nesting clear, unambiguous

Boolean Operators

Decisions may have multiple parts:

> if the class is not full AND
> it does not conflict with a
> currently scheduled class. . .

Boolean operators allow us to express compound conditionals:

- AND

  // both conditions must be true
  ((5 > 4) AND (4 > 7)) is FALSE

- OR

  // at least one condition must be true
  ((5 >4) OR (4 > 7)) is TRUE

- NOT

  // reverses the boolean value
  NOT (4 > 7) is TRUE

Using the Boolean Operators

```
if (NOT (it is raining))
then
    go to the park
else
    play video games
endif
```

- Boolean operators can be combined:

```
if ((location = Atlanta) AND
   (house_cost < 150,000) OR
   won_lottery) then
       buy_the_house
else
       keep_looking
endif
```

- What does the complex condition above really do?

Documenting Algorithms

FICTION: "the main audience for an algorithm is a computer."

FACTS: only 10-20% of $$ cost is *writing* algorithm;

80-90% is subsequent repair, extension, reuse by *people*.

TRUTH: writing algorithms for the human audience is ***critical*** and ***essential***

DOCUMENTATION: refers to brief, explanatory comments that are written into the algorithm for the purpose of making it easy for people to understand

Documenting Algorithms

- To signify documentation, use *double slash marks*.

- Whatever follows is documentation, i.e., notes to the reader, not instructions to the computer,

```
// this is a documentation comment
```

- Document:
  - purpose of algorithm and modules;
  - the role of data structures;
  - key decision points;
  - complex calculations;
  - changes in flow of control.

- Don't document the trite & trivial, e.g.,

```
// Assign the value 4 to variable a_value
a_value <- 4
```

Self-Documenting Algorithms

Whereever possible, make the algorithm *self-documenting* by:

- Using *descriptive identifiers*. Choose variable names that describe the data they will hold.

  For example,

  If declaring variables to hold values for the *radius and area of a circle* . . .

  Good:

```
radius isoftype Num
area isiftype Num
```

  Bad:

```
x isoftype Num   // used to store radius
y isoftype Num   // used to store area
```

- Use *constants* instead of *literals* . . .

Literals

Literals are data values that are "hardwired" into the algorithm:

```
algorithm Literals

    num_one isoftype Num
    num_one <- 1

    your_grade isoftype Char
    your_grade <- "A"

    your_score isoftype Num
    your_score <- 42

    print("Each assignment statement here
            contains a literal value")

endalgorithm // Literals
```

Constants

A <u>better</u> way ...

- To use non-variable values in your algorithm, use ***constants,*** not *literals*:
- A constant allows us to give a name (identifier) to a fixed value, making it easier to read and understand.

<u>For example</u>:

PI is 3.1415926
TAX_RATE is .28

Then, instead of:

```
area = 3.1415926 * radius * radius
```

we have:

```
area = PI * radius * radius
```

And instead of

```
tax_due <- taxable_gain * .28
```

we have

```
tax_due <- taxable_gain * TAX_RATE
```

Another Simple Example

```
algorithm Circle_Facts

  // declare constants and variables
  PI is 3.1415926
  radius,
  diameter,
  circumference,
  area isoftype Num

  // get data
  print("Enter the radius")
  read(radius)

  // perform calculations
  diameter <- radius * 2
  circumference<- 2 * PI * radius
  area <- PI * radius * radius

  // display the results
  print("Diameter is: ", diameter)
  print("Circumference is: ",  circumference)
  ptint ("Area is: ", area)

endalgorithm // Circle_Facts
```

Tracking Stooge Boinks

```
algorithm Stooge_Boinks
//  calculates the  average number of boinks
//  performed per Stooge in a given episode

   NUM_STOOGES is 3
   larry isoftype Num
   moe  isoftype Num
   curley isoftype Num

   // obtain data from user
   print("Enter number of boinks by Larry,
          Moe and Curley, respectively")
   read (larry, moe, curley)

   // perform calculation
   average isoftype Num
   average <- (larry + moe + curley)
                / NUM_STOOGES

   // generate output
   print ("The average number of boinks
           per stooge is  ", average)

endalgorithm // Stooge_Boinks
```

Simple Finances

```
algorithm Net_Interest_Earned
// Computes simple interest earned in one year
// and associated flat-rate tax, based on data
// obtained from user.

  money_invested isoftype Num
  read(money_invested)

  interest_rate isoftype Num
  read(interest_rate)

  interest_earned isoftype Num
  interest_earned <- money_invested *
                     (INTEREST_RATE)

  TAX_RATE is .34          // 34% (28 fed+6 state)
  taxes_incurred isoftype Num
  taxes_incurred <- interest_earned * TAX_RATE

  print ("In one year at " interest_rate, " interest, ")
  print (money_invested, "earns ", interest_earned)
  print(taxes_incurred, "is owed in taxes, for a net.")
  print("gain of ", (interest_earned - taxes_incurred))

endalgorithm // Net_Interest_Earned
```