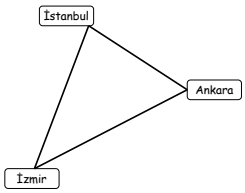
Introduction to Programming

H. Turgut Uyar
uyar@cs.itu.edu.tr

Computer Programs

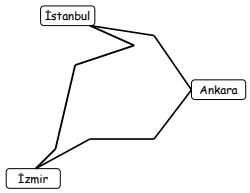
- how to represent the problem?
  - computers work on numbers
  - program about the highways in Turkey
  - entities: cities and roads
  - representing a city: name, latitude, longitude
- how to express the solution?

Representing the Problem



- representing a road: line
  - assume the road is straight
  - start and end cities
- very easy
- very inaccurate

Representing the Problem



- representing a road: consecutive lines
- very hard
- more accurate → more complicated

Representation: Model

- FIRST STEP: build a correct / accurate (and feasible) model
- what you are solving is the model, not the problem itself
  - incorrect model → incorrect solution
  - inaccurate model → meaningless solution
  - infeasible model → expensive implementation

Representing the Problem

- representing highways:
  - if you are only interested in total distances, you can use lines
  - if you will talk about "the 274th km of the İstanbul–Ankara highway", you should use consecutive lines

Data

- data that model an entity are represented by *variables*
  - symbolic name for the data
  - variables take *values*
  - city variables: name latitude longitude

    to represent İstanbul:

    name: "İstanbul"

    latitude: 41

    longitude: 29

Variables

- variables are kept in memory
  - variable is the name of the memory cell
  - value is the content of the memory cell

name	latitude	longitude
"İstanbul"	41	29

Assignment

- block structured programs proceed by assigning values to variables
- notation: latitude ← 41
  - "store the value 41 in the memory cell named latitude"
- left hand side is a variable
- right hand side is an *expression*
  - a computation that yields a value

Expressions

- can be a single value or variable:
  - 41
  - latitude
- can be combinations of values and variables connected via *operators*:
  - 4 * longitude — multiplication operator
  - latitude + longitude — addition operator

Assignment

- ASSIGNMENT IS NOT EQUALITY!!!
- 41 ← latitude doesn't make sense
- i ← i + 1 means: increment the value of i by 1
  - if i was 5 before the operation, it will become 6 after the operation
- mathematically it would be incorrect:

  0 = 1

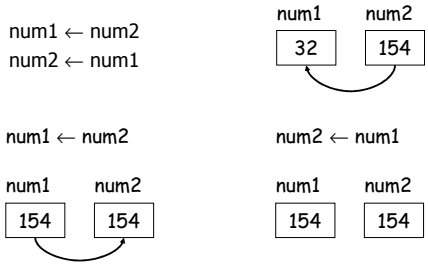
Swap

- swap the values of two variables:

before the operation		after the operation	
num1	num2	num1	num2
32	154	154	32

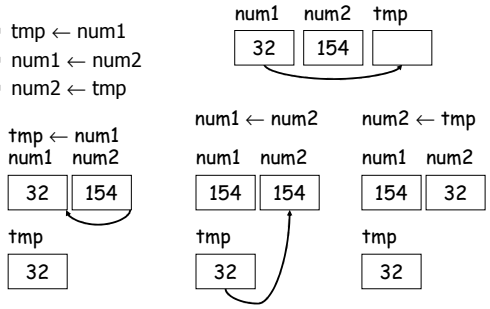
Swap: Incorrect

- num1 ← num2
- num2 ← num1



Swap

- tmp ← num1
- num1 ← num2
- num2 ← tmp



Data Types

- basic data types:
  - integer
  - real number
  - logical
  - character
  - string
- composite data types: record
- vector data types: array

Basic Data Types

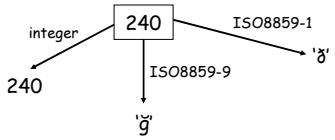
- integer
  - birthyear, number of letters in the surname, height in cm
- real numbers
  - height in m, average of several exam scores, square root of a number
- logical: values can be *true* or *false*
  - student successful, older than 18 years

Character

- any symbol: letter, digit, punctuation mark, ...
  - first letter of surname, the key the user pressed
- mostly written between single quotes:
  - 'Y', '4', '?'

Encoding

- numbers correspond to symbols
- ASCII, ISO8859-X, Unicode

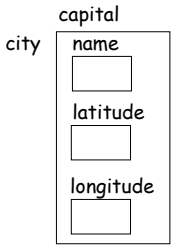


Strings

- name, word, sentence, ISBN number, ...
- mostly written between double quotes:
  - "Dennis Ritchie", "ISBN 0 13 110362 8"
- use numbers if you plan to make arithmetic operations on it:
  - student numbers at ITU: 9 digit numbers
  - will you add/multiply/... student numbers?
  - no sense in using integers, use strings

Composite Data Types

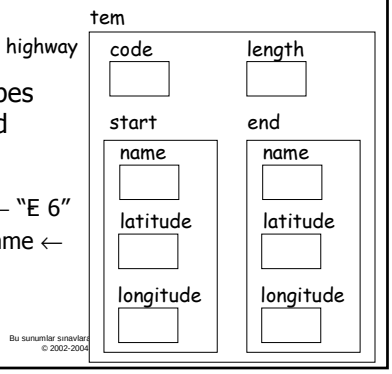
- grouping types to form a new type



- access:
  - capital.name ← "Ankara"
  - NOT city.name ← "Ankara"

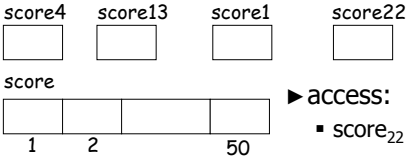
Composite Data Types

- composite types can be nested
  - access:
    - tem.code ← "E 6"
    - tem.end.name ← "Ankara"



Vector Data Types

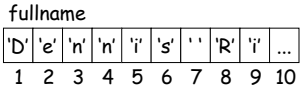
- grouping elements of the same type
  - exam scores for a 50 student class:
    - 50 integer variables: score1, score2, ..., score50
    - an integer array with 50 elements: score



- access:
  - $score_{22} \leftarrow 95$

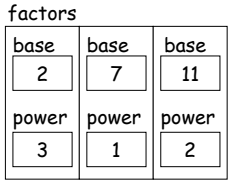
Strings

- strings are usually arrays of characters



Arrays of Records

- represent the factors of a number:
  - an array where each member is a factor
  - a factor is a record of a base and a power



- access:
  - $factors_2$.base ← 7

Expressing the Solution

- step by step guide to the solution: algorithm
- recipe for Jamaican rice and peas:
  - put 1 1/2 can of beans in 4-5 cups of water
  - add 1/4 can of coconut milk, 1 sprig of thyme and salt and pepper to taste
  - cook until beans are soft
  - smash the bottom of an escallion and add it to the pot along with 2 cups of rice and 1/4 can of coconut milk and 2 sprigs of thyme
  - remove any access water over 2cm above the rice
  - bring to a boil for 5 min
  - continue to cook covered until rice is tender

Algorithm

- there must be no room for judgement
  - 4-5 cups? sprig?
  - salt and pepper to taste?
  - beans are soft? rice is tender?
- this cooking recipe is NOT an algorithm
- must be finite
- in a finite number of steps:
  - either find the correct solution
  - or report failure to find a solution
- must not run forever

Flowcharts

- describe algorithms
- elements:
  - box: an operation
  - arrow: flow direction
  - diamond: decision point
  - parallelogram: input/output operation

Flowcharts

- find the maximum score in an exam with 50 students
  - represent exam scores by a 50 element integer array (variable: score)
  - represent maximum score by an integer (variable: max)

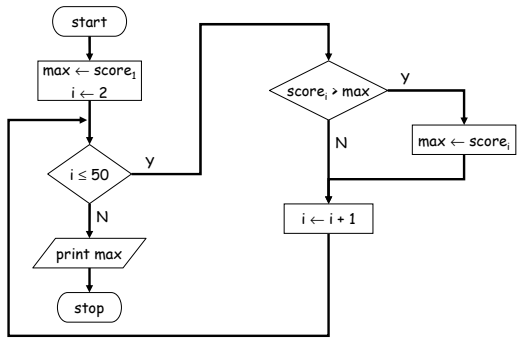
1. choose first score as maximum
2. if there are more students go to step 3, else go to step 5
3. if next score is higher than maximum, choose it as maximum
4. proceed to next student
5. print the maximum

Flowcharts

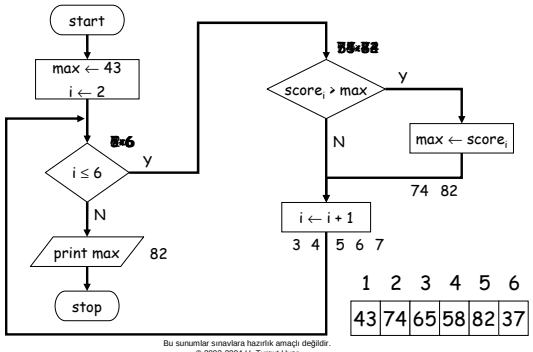
- representation problem:
  - more students? next score?
  - counter variable: i

1. $max \leftarrow score_1$, $i \leftarrow 2$
2. if $i \leq 50$ go to step 3 else go to step 5
3. if $score_i > max$ then $max \leftarrow score_i$
4. $i \leftarrow i + 1$ and go to step 2
5. print max

Flowcharts



Flowcharts



Flowcharts

- using tables to represent flow:

max	i	i ≤ 6	$score_i$ > max
43	2	T (2 < 6)	T (74 > 43)
74	3	T (3 < 6)	F (65 < 74)
	4	T (4 < 6)	F (58 < 74)
	5	T (5 < 6)	T (82 > 74)
82	6	T (6 = 6)	F (37 < 82)
	7	F (7 > 6)	

Flowcharts

- number guessing game:
  - one player picks a number (target) between lower and upper bounds
  - the other player makes a guess:
    - if guess is bigger than target, picker says "smaller"
    - if guess is smaller than target, picker says "bigger"
    - game ends when guess = target

Flowcharts

- Algorithm II:
  - try the number in the middle
  - if "smaller" narrow the search to the bottom half
  - if "bigger" narrow the search to the top half

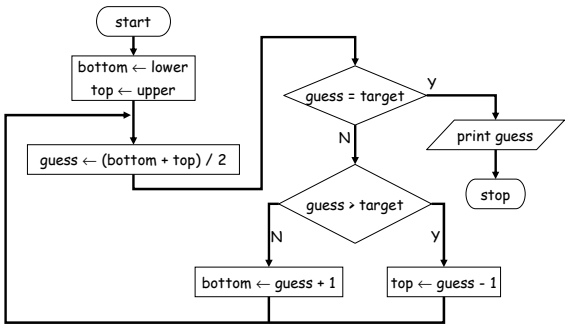
- Algorithm I:
  - start with lower, increment by 1 until found

1. guess ← lower
2. if guess = target stop
3. guess ← guess + 1 and go to step 2

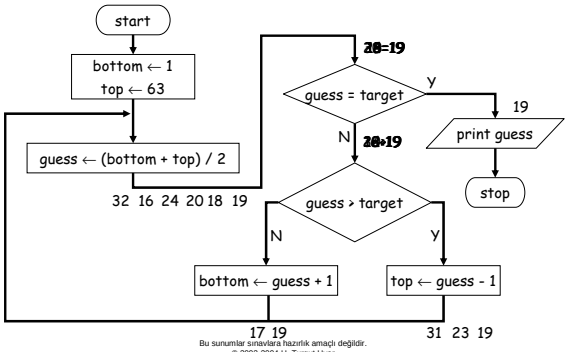
Flowcharts

1. bottom ← lower, top ← upper
2. guess ← (top + bottom) /2
3. if guess = target stop
4. if guess > target then top ← guess- 1
   otherwise bottom ← guess + 1
   and go to step 2

Flowcharts



Flowcharts



Flowcharts

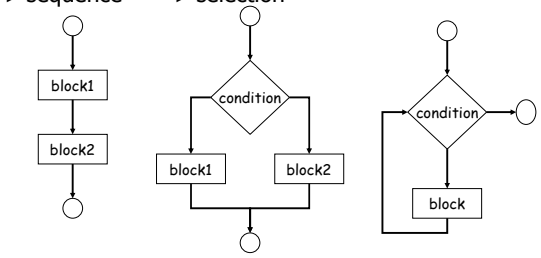
bottom	top	guess	guess=target
1	63	32	
	31	16	F (32 > 19)
17		24	F (16 < 19)
	23	20	F (24 > 19)
	19	18	F (20 > 19)
19		19	F (18 < 19)
			T (19 = 19)

Comparing Algorithms

- number guessing: which algorithm is better?
- speed:
  - worst case: first one 63, second one 6
  - average case: first one 32, second one ~5
- size: second one requires two more variables

Block Structures

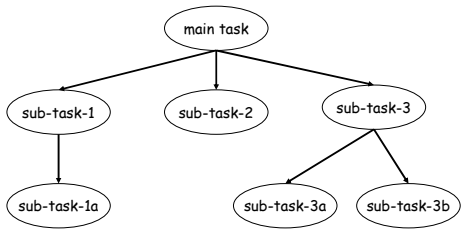
- sequence
- selection
- repetition



Abstraction

- divide the main task to sub-tasks
- consider each sub-task as a main task and divide into sub-sub-tasks, ...
  - divide and conquer
- top-down design
- each task is implemented by a procedure (in C a function)

Abstraction



Abstraction

- procedures are only interested in WHAT sub-procedures are doing, not HOW they are doing it
- smaller units are easier to manage
- maintaining is easier
  - if the HOW of the sub procedure changes, the super procedure is not affected

Abstraction

- procedures should be general:
  - instead of "find the maximum score in the final exam of BIL105E"
  - do "find the maximum of any array"
  - you can use this to find the "maximum score in the final exam of BIL105E"
  - and also to find the "maximum shoe size of the LA Lakers players"

Parameters

- which data will the procedure work on?
  - input parameter:
    - the scores in the final exam of BIL105E
    - the shoe sizes of the LA Lakers players
- what value will the procedure produce?
  - output parameter:
    - maximum score
    - maximum shoe size

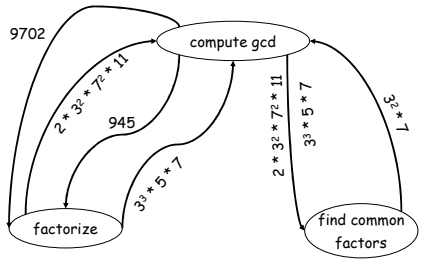
Abstraction

- find the greatest common divisor (gcd) of two numbers:
  1. decompose the first number to its prime factors
  2. decompose the second number to its prime factors
  3. find the common factors of both numbers
  4. compute the gcd from the common factors

Abstraction

- sample numbers: 9702 and 945
  1. $9702 = 2 * 3^2 * 7^2 * 11$
  2. $945 = 3^3 * 5 * 7$
  3. $3^2 * 7$
  4. 63

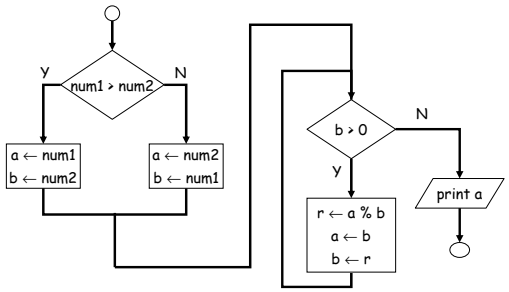
Abstraction



Euclides Algorithm

- finding the greatest common divisor (gcd) of two numbers: Euclides algorithm
  - let a be the bigger number and b the smaller number
  - the gcd of a and b is the same as the gcd of b and a % b

Euclides Algorithm



Euclides Algorithm

a	b	r
9702	945	252
945	252	189
252	189	63
189	63	0

Comparing Algorithms

- Algorithm I:
  - hard to factorize numbers
  - easier to compute the gcd/lcm of more than two numbers?
- Algorithm II (Euclides):
  - very fast
  - very easy to implement

Input

- most programs read the data from outside:
  - ask the user: get from keyboard
  - read from a file
  - read from the environment: get temperature of the room via a sensor
- input commands transfer the value read from outside to a variable

Output

- what to do with the produced results?
  - tell the user: print it on the screen
  - write it to a file or printer
  - send to the environment: control the valve of a gas pipe
- output commands send results to output units
- error messages to error unit

Program Types

- console / command line / text mode programs:
  - read inputs
  - process data and produce results
  - show outputs
- graphical programs are event driven:
  - prepare the environment (windows, buttons, ...)
  - wait for events (mouse click, key press, ...)
  - respond to events

Development Stages

- design: on "paper"
  - model, algorithm
  - which programming language?
  - software engineering
- coding: writing the program
  - source code
  - editor
- testing: does the program work as expected?
  - scenarios
- debugging: finding and correcting the errors
  - debugger

Errors

- syntax errors
  - not conforming to the rules of the language
- logical errors
  - division by zero

Evaluating Programs

- efficiency
  - speed
  - hardware requirements
- portability
  - can it run on another platform without much change?
  - source code portability
- understandibility
  - can others (or you) understand your code?
- ease of maintenance
  - can new features be added?
- robustness
  - can it tolerate user errors?

Machine Code

- executable files have to have a computer understandable format
- executable format differs between
  - hardware
  - operating systems
- programmers cannot write directly in machine code
- write source code in a high level language
- use tools to convert source code to machine code

Conversion

- *interpreted*
  - read a command from source code
  - convert
  - execute
  - repeat for next command
  - if error report (only first error) and abort
- *compiled*
  - read in the whole source code
  - convert and build an executable file
  - if error report (all errors) and no executable
  - conversion is done once $\Rightarrow$ much faster

Libraries

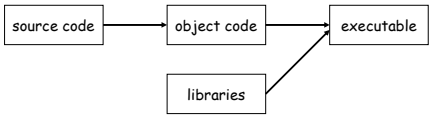
- every feature built into the language ⇒ hard and slow implementation
- small, fast core language + extensions
- no built in square root feature in C
- extra features are kept in procedure archives called *libraries*
- why use libraries?
  - thoroughly tested ⇒ reliable and efficient
  - save time

Standards

- to achieve portability, standards are needed
- ANSI C: core C language,libraries
- ISO C/C++ standards
- POSIX: how to access the operating system?
- no standard for graphics operations!

Building the Executable

- *compiler*: produces machine code for the procedures in the source code → object code
- *linker*: combines the object code with the library procedures used in the source code → executable



Introduction to the C Language

First Example

```
/* My first C program.          *
 * This program computes .. */
                    VARIABLES
#include <iostream>    // cout,cin
#include <stdlib.h>
                    INPUT
#define PI 3.14
                    PROCESSING

                    OUTPUT
```

```
int main(void)
{
  float radius;
  float circum, area;

  std::cout << "Radius: ";
  std::cin >> radius;
  circum = 2 * PI * radius;
  area = PI * radius * radius;
  std::cout << "Circumference: "
            << circum << "\n";
  std::cout << "Area: "
            << area << "\n";
  return EXIT_SUCCESS;
}
```

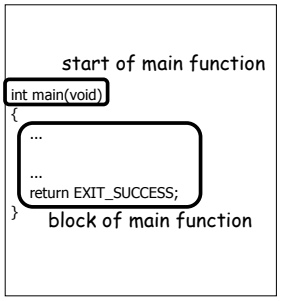

Comments

- comments improve the understandability of your code
  - anything between /* and */
  - anything from // to end of line
  - completely ignored by the compiler

```
/* My first C program.          *
 * This program computes .. */

#include <iostream>    // cout,cin
#include <stdlib.h>

#define PI 3.14

int main(void)
{
  ...
}
```

Functions

- execution starts with function main
  - each program must have one and only one main
  - starts with:
    int main(void)
  - ends with:
    return EXIT_SUCCESS;
  - program successful
- blocks are delimited using curly braces



Statements

- statements end with semicolon
- they can span across lines
- multiple blanks are regarded as one blank
  - except in strings

```
int main(void)
{
  float radius;
  float circum, area;

  cout << "Radius: ";
  cin >> radius;
  circum = 2 * PI * radius;
  area = PI * radius * radius;
  cout << "Circumference: "
       << circum << endl;
  cout << "Area: "
       << area << endl;
  return EXIT_SUCCESS;
}
```


Indentation

- all statements belonging to the same block should have the same left margin

```
int main(void)
{
  float radius;
  float circum, area;

  std::cout << "Radius: ";
  std::cin >> radius;
  circum = 2 * PI * radius;
  area = PI * radius * radius;
  std::cout << "Circumference: "
            << circum << "\n";
  std::cout << "Area: "
            << area << "\n";
  return EXIT_SUCCESS;
}
```


Assignment

- assignment is done using the = symbol
- NOT EQUALITY

```
int main(void)
{
  float radius;
  float circum, area;

  std::cout << "Radius: ";
  std::cin >> radius;
  circum = 2 * PI * radius;
  area = PI * radius * radius;
  std::cout << "Circumference: "
            << circum << "\n";
  std::cout << "Area: "
            << area << "\n";
  return EXIT_SUCCESS;
}
```


Header Files

- needed for using library entities
  - to use std::cout and std::cin, we need iostream
  - to use EXIT_SUCCESS, we need stdlib.h

```
#include <iostream>
#include <stdlib.h>
...
int main(void)
{
  ...
  std::cout << "Radius: ";
  std::cin >> radius;
  ...
  return EXIT_SUCCESS;
}
```


Rules for Names

- valid symbols: lower and uppercase letters of English, digits and underscore
  - pi, weight, weight1 and weight_1 are valid
  - π, ağırlık and weight-1 are not valid
- first symbol must not be a digit
  - 1weight is not valid
- should not be longer than 31 symbols
- names are case-sensitive:
  - weight and wEiGHt are different

Rules for Names

- C reserved words can not be chosen as names
  - int, main, void and return are not valid
- library names can be chosen as names but they will become unaccessible

Name Conventions

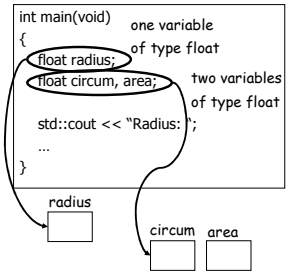
- CONVENTION:
  - start variable names with lowercase letters
  - use meaningful names
  - if name consists of more than one word use one of:
  - birth_month, birthMonth

Numbers

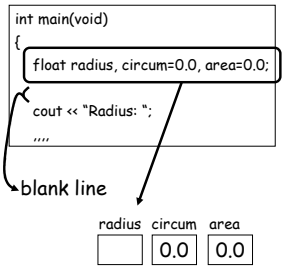
- integers:
  - decimal: 59
  - octal: start with 0: 073
  - hexadecimal: start with 0x: 0x3b
- real numbers:
  - decimal: 3.14
  - scientific: use E: 0.314E1, 31.4E−1

Variables

- variables must be defined before used
  - data type and name
- reserve a memory cell for this variable
- definitions can be grouped



Variables



- initial values can be assigned to variables
- define variables *before* statements
- CONVENTION:
  - leave one blank line between end of variable definitions and start of statements

Data Types

- integers: int
  - short / long
  - signed / unsigned
- real numbers: float
- double precision real number: double
- symbol: char
- logical: bool
- type determines value range
- if short int is 16 bits
  - signed short int can represent -32768 to 32767
  - unsigned short int can represent 0 to 65535
- sizeof

Constants

```
...
#define PI 3.14
...
int main(void)
{
   ...
   circum = 2 * PI * radius;
   area = PI * radius * radius;
   ...
}
```

- improve understandibility
  - symbolic name instead of number
- changing is easier
  - use 3.14159 instead of 3.14 → change at one point

Constants

- using #define

#define NAME value

  - as if you have written 3.14 instead of PI everywhere in the code
- CONVENTION:
  - use all uppercase names for constants
- using const:
  - put const in front of variable declaration
  - defines a read-only variable

```
float circum, area;
const float pi = 3.14;
circum = 2 * pi * radius;
```


Arithmetic Expressions

- addition: +
- subtraction:-
- multiplication: *
- division: /
- remainder: %
- math library functions:
  - sin,cos,exp,log,pow,...
- CONVENTION:
  - leave one blank space before and after operators
  - also before and after = in assignments

Precedence

- expressions in parentheses
- unary + and -: -2
- * / %
- + -
- operators of equal precedence are evaluated left to right

Precedence

a + b + c + d + e / 5

- will be evaluated as:

(a + b + c + d) + e / 5

- to divide the sum by 5:

(a + b + c + d + e) / 5

p * r % q + w / x- y

- will be evaluated as:

p * r

p * r % q

w / x

p * r % q + w / x

p * r % q + w / x- y

Typecasting

- if both operands are integers, result is integer
- if either one or both real, result real
  - 14 / 4 = 3
  - 14.0 / 4 = 3.5

```
int num1 = 14, num2 = 4;
float quotient;
quotient = num1 / num2;
```

- convert a variable to some other type:
- (type) expression

```
(float) num1
quotient = (float) num1 / num2;
```

Increment / Decrement

- can be combined with assignment:
  - a = a + 5; a += 5;
  - a *= b + 1;
  - a = a * (b + 1);
- increment: ++
- decrement: --
  - a = a + 1; a += 1; a++; ++a;
  - b = b- 1; b- =1; b- ;- b;

Input / Output

- input:
  cin >> radius;
- transfer the value that the user has typed to radius
- multiple values can be read in one statement:
  cin >> num1 >> num2;
- output:
  cout << "Area: "
  << area << endl;
- send string or value of expression to output
- endl: end line (start a new line)

Flow Control

Computing Roots

- find the roots of the second degree equation: $ax^2 + bx + c = 0$
  $x_{1,2} = (b \pm (b^2 - 4ac)^{1/2}) / (2 * a)$
- compute discriminant: $b^2 - 4ac$
  - if negative: "no real roots"
  - if zero: "two colliding roots at ..."
  - if positive: "two real roots at ..."

Computing Roots: Outline

```
...
int main(void)
{
    float a, b, c;
    float disc;
    float x1, x2;

    std::cout << "Enter coefficients: ";
    std::cin >> a >> b >> c;
    disc = b * b - 4 * a * c;
    // compute the roots
    // and report accordingly
    return EXIT_SUCCESS;
}
```

- variables:
  - a, b and c: coefficients of the polinom
  - disc: discriminant
  - x1 and x2: real roots
- note the input command:
  - 3 variables in one input command

Conditions

- logical expression:
  - comparison of two arithmetic expressions
  - either true or false
- disc < 0
  - if value of disc is less than 0, then true
  - otherwise, false
- true is represented with 1, false with 0
  - any expression not equal to 0 is true

Comparison Operators

- equal: x == y
- not equal: x != y
- less than: x < y
- greater than: x > y
- less or equal: x <= y
- greater or equal: x >= y

- Examples
  - age >= 18
  - (year % 4) == 0
- works for chars too: only for English
- not for strings
- real numbers should not be compared for equality

Logical Operators

- negating conditions:
  - NOT operator: !
  - !(age >= 18)
  - same as: age < 18
- combining several conditions
  - 18 <= x < 65 does not work
  - 18 <= x and x < 65
  - AND operator: &&
  - OR operator: ||
  - (18 <= x) && (x < 65)
  - !((18 <= x) && (x < 65))
  - (18 > x) || (x >= 65)
- precedence: ! && ||
- leap year:

  (year % 4 == 0) &&
  (year % 100 != 0) ||
  (year % 400 == 0)

Selection

```
...
if (disc == 0) {
  x1 = -b / (2 * a);
  std::cout << "Two colliding roots at "
        << x1 << "\n";
} else {
  x1 = (-b + sqrt(disc)) / (2 * a);
  x2 = (-b - sqrt(disc)) / (2 * a);
  std::cout << "Two real roots at "
      << x1 << " and " << x2 << "\n";
}
....
```

- if disc is equal to 0
  - execute first block: operations for colliding roots (2 statements)
- if not
  - execute second block: operations for distinct real roots (3 statements)
- else block is not obligatory
  - if true execute block, else do nothing

Selection

```
...
if (disc < 0)
  cout << "No real roots." << endl;
else {
  if (disc == 0) {
    ...
  } else {
    ...
  }
}
....
```

- one statement blocks don't need braces
- could also be:

```
if (disc < 0) {
    cout << "No real ..."
} else {
    ...
```

- NOTE THE INDENTATION

Conditional Operator

- choose one of two expressions

```
z = x < y ? x : y;
if (x < y)
    z = x;
else
    z = y;
```

- an operator, not a selection structure!

Multiple Selection

```
...
int main(void)
{
  int num1, num2, result;
  char op;

  cout << "Type the operation: ";
  cin >> num1 >> op >> num2;
  // carry out operation
  cout << "Result: " << result << endl;
  return EXIT_SUCCESS;
}
```

- carry out operation specified by user
- variables
  - num1, and num2: operands
  - op: operator (one of + - * / %)
  - result: result of operation

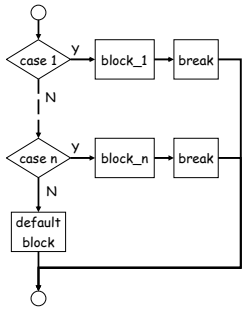
Multiple Selection

```
if (op == '+') {
   result = num1 + num2;
} else {
   if (op == '-') {
      result = num1 - num2;
   } else {
      if (op == '*') {
         ....
```

- comparing same expression with several values

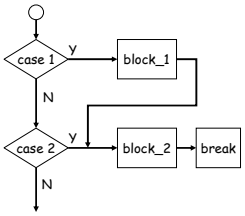
```
switch (op) {
   case '+':
      result = num1 + num2;
      break;
   ....
   case '%':
      result = num1 % num2;
      break;
   default:
      cout << "No such op" <<
   endl;
      return EXIT_FAILURE;
}
```


Multiple Selection




```
switch (expression) {
   case value_1:
      block_1;
      break;
   ....
   case value_n:
      block_n;
      break;
   default:
      default_block;
      break;
}
```


Multiple Selection



- default case not obligatory
- compare with:
  - integers and chars
  - not real numbers or strings
  - not variables
- if no break:
  - continue with block of next case!

GCD (Euclides): Outline

```
...
int main(void)
{
   int num1, num2;
   int a, b, r = 1;

   cout << "Enter the numbers: ";
   cin >> num1 >> num2;
   // compute the gcd
   cout << "The gcd of " << num1 << " and "
        << num2 << " is " << a << endl;
   return EXIT_SUCCESS;
}
```

- variables:
  - num1 and num2: the numbers for which we want to find the gcd
  - a, b and r: the numbers used in the repetition part of the algorithm

Repetition

```
...
int main(void)
{
   ...
   // determine a and b
   while (b > 0) {
      r = a % b;
      a = b;
      b = r;
   }
   ...
}
```

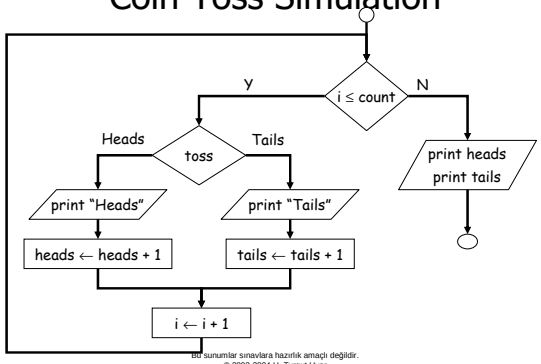
- while b is greater than 0
- make sure the loop terminates!

Repetition

```
...
while (true) {
   // prepare for check
   if (outer termination condition)
      break;
   // outer loop body
   while (true) {
      // prepare for check
      if (inner termination condition)
         break;
      // inner loop body
   }
   // outer loop body
}
...
```

- endless loop
  - check for termination condition
- if met: break
  - get out of the inmost switch, while, do-while or for
  - not out of if

Coin Toss Simulation



Outline

```
...
int main(void)
{
   int count, i;
   float number;
   int heads = 0, tails = 0;

   cout << "How many times to toss? ";
   cin >> count;
   // toss the coins
   cout << "Heads count: " << heads << endl
        << "Tails count: " << tails << endl;
   return EXIT_SUCCESS;
}
```

- variables:
  - count: total coin toss count
  - i: which toss is next?
  - number: used for simulating the toss
  - heads and tails: how many heads and how many tails?

Outline

```
...
int main(void)
{
   ...
   for (i = 1; i <= count; i++) {
      number = (float) rand() / RAND_MAX;
      if (number < 0.5) {
         cout << "Heads" << endl;
         heads++;
      } else {
         cout << "Tails" << endl;
         tails++;
      }
   }
   ...
}
```

- for: repetition block
- if: selection block 1
- else: selection block 2

Counter-Controlled Repetition

```
...
int main(void)
{
   ...
   for (i = 1; i <= count; i++) {
      number = (float) rand() / RAND_MAX;
      if (number < 0.5) {
         cout << "Heads" << endl;
         heads++;
      } else {
         cout << "Tails" << endl;
         tails++;
      }
   }
   ...
}
```

- repeating a block a number of times
- use a counter: i
  - what is the starting value? 1
  - what is the ending value? count
  - what is the increment step? 1

Counter-Controlled Repetition

```
for (start assignment;
        continue condition;
        increment) {
   block;
}

start assignment;
while (continue condition) {
   block;
   increment;
}
```

- incrementing is done AFTER the execution of the block
  - the first round is not skipped
- if condition does not hold at the start, block is not executed at all
- increment could be anything

Random Numbers

- how to toss a coin?
  - produce a random number (real) between 0 and 1
  - if smaller than 0.5 call it "heads", otherwise call it "tails"
- C's built in random number generator:
  - rand(): generates a random number (integer) between 0 and RAND_MAX
  - rand() / RAND_MAX is between 0 and 1
  - 1 + rand() % limit

Random Numbers

- random numbers are generated in a sequence
  - r0 r1 r2 r3 .....
  - each number is computed using the number that precedes is in the sequence
  - to start the sequence we need a seed (r0)
- same seed results in same sequence
  - srand(): sets seed
  - we need a different seed for each execution
  - time(): number of seconds since Jan 1, 1970
    - input parameter: NULL

Random Numbers

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
...
int main(void)
{
  ...
  srand(time(NULL));
  ...
    number = (float)rand() / RAND_MAX;
  ...
}
```

- to use srand() and rand(), we need stdlib.h
- to use time(), we need time.h

Derived Data Types

Programmer-Defined Types

- programming languages provide basic data types: int, float, bool, ...
- they also provide mechanisms for defining our own data types
  - give an existing data type a new name
  - combine existing data types to form a new data type

New Names for Types

- typedef existing_type new_name;
  - student scores in an exam:

    int midterm1, midterm2, final;
  - create a new data type score_t:

    typedef int score_t;

    score_t midterm1, midterm2, final;
- old type name is still valid

Advantages

- understanding is easier
- changing is easier
  - later we decide we need real numbers for student scores:

    typedef float score_t;
  - change at one point instead of going through the whole code

Craps Simulation

- player rolls dice:
  - if 7 or 11: player wins
  - if 2, 3 or 12: player loses
  - otherwise sum is player's "point"
- player rolls again until:
  - point: player wins
  - 7: player loses

Outline

```
...
// type definitions
...
int main(void)
{
   int die1, die2, sum, point;
   status_t game_status;

   // first roll
   // check outcome
   // continue to roll while not determined
   // report
   return EXIT_SUCCESS;
}
```

- variables:
  - die1, die2 and sum: value of first and second dice and their sum
  - point: player's point
  - game_status: continue / win / lose

Enumerated Type

- game can be in one of three states
- how to represent the states?
  - encode (assign a value to each state)

```
#define GAME_CONTINUES 0
#define PLAYER_WINS 1
#define PLAYER_LOSES 2
```

  - values don't matter as long as they are different

Enumerated Type

- defining a group of constants in one statement:
  - enum { constant definitions };

```
enum { GAME_CONTINUES = 0,
       PLAYER_WINS = 1,
       PLAYER_LOSES = 2 };
```

- same as using multiple #define statements

Enumerated Type

- if constant values are omitted:
  - first constant is 0, subsequent ones are incremented by 1

```
enum { GAME_CONTINUES,
       PLAYER_WINS, PLAYER_LOSES };
```

  - any omitted value is 1 more than the one before

```
enum { JAN = 1, FEB, ... DEC };
```


Enumerated Type

- giving a group of constants a type name:
  - enum tag { constant definitions };

```
enum status_e { GAME_CONTINUES,
                PLAYER_WINS, PLAYER_LOSES };
```

  - name of new type: enum tag
- variables can be defined of this type:

```
enum status_e game_status=GAME_CONTINUES;
```

- variables can take any value

```
game_status = 25;
```

Enumerated Type

```
#include ...
...
enum status_e { GAME_CONTINUES,
   PLAYER_WINS, PLAYER_LOSES };
typedef enum status_e status_t;
...
int main(void)
{
   ...
   status_t game_status = GAME_CONTINUES;

   ....
}
```

- enumerated type declarations are made right after the #include statements

Fall-Through Cases

```
switch (sum) {
   case 7:
   case 11:
      game_status = PLAYER_WINS;
      break;
   case 2:
   case 3:
   case 12:
      game_status = PLAYER_LOSES;
      break;
   default:
      game_status = GAME_CONTINUES;
      point = sum;
      break;
}
```

- intentionally leaving out break statements in cases
- cases are grouped

End of Program

```
// first roll
// if win or lose skip the loop below
while (game_status == GAME_CONTINUES) {
   // roll again
   if (sum == point)
      game_status = PLAYER_WINS;
   else {
      if (sum == 7)
         game_status = PLAYER_LOSES;
   }
}
// either loop was skipped: win / lose
// or we got out of loop: win / lose
```

- is it possible that this program will run forever?

Parallel Resistors

- compute the equivalent of several parallel resistors:
  - 1 / R = 1 / R1 + 1 / R2 + ... + 1 / Rn
  - R = 1 / (1 / R1 + 1 / R2 + ... + 1 / Rn)
- resistor values are input by the user
- when finished the user will input 0

Outline

```
...
// type definitions
...
int main(void)
{
   int resistor;
   rational_t sum = { 0, 1 };
   int a, b, r;
   int i = 0;

   // get resistor values from user
      // and compute the equivalent
   // report
   return EXIT_SUCCESS;
}
```

- variables:
  - resistor: resistor value typed by the user
  - sum: sum of 1/Ri so far
  - a, b and r: used to find the gcd for simplifying the fraction
  - i: counter

Input

```
...
while (true) {
   cout << "Enter resistor value: ";
   cin >> resistor;
   if (resistor == 0)
      break;
   // add the new resistor value
   // to the sum so far
   // simplify the sum
}
```

- endless loop:
  - break when user types 0
- how would you reorganize this code without using endless loop?

Structures

- combining existing data types to form a new data type:
  - struct tag { field declarations };
  - rational numbers:

```
struct rational_s {
    int nom;
    int denom;
};
```

  - name of new type: struct tag

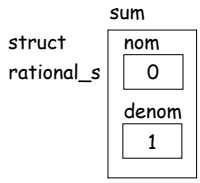
Structures

- field declarations are NOT variable definitions!!!
  - no memory is allocated until a variable of this type is defined!

```
struct rational_s sum;
```

  - initial values can be given in curly braces:

```
struct rational_s sum = { 0, 1 };
```



Structures

```
#include ...
...
struct rational_s {
    int nom, denom;
};
typedef struct rational_s rational_t;
...
int main(void)
{
    ...
    rational_t sum = { 0, 1 };

    ....
}
```

- structure declarations are made right after the #include statements

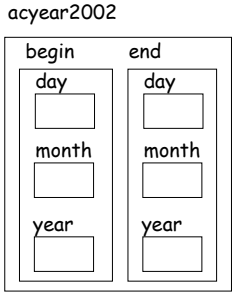
Structures

```
...
while (true) {
    // get next resistor value
    sum.nom = sum.nom * resistor
              + sum.denom;
    sum.denom *= sum.denom * resistor;
    // find gcd of sum.nom and
    // sum.denom (gcd is in a)
    sum.nom = sum.nom / a;
    sum.denom = sum.denom / a;
}
...
/// print sum.denom / sum.nom
```

- fields of a structure variable are accessed using the dotted notation

Nested Structures

```
enum month_e { JAN = 1, FEB, .. DEC };
typedef enum month_e month_t;

struct date_s {
    int day;
    month_t month;
    int year;
};
typedef struct date_s date_t;

struct acyear_s {
    date_t begin, end;
};
typedef struct acyear_s acyear_t;
```



Assigning Structures

- instead of assigning fields one-by-one:

```
acyear2002.end.day = acyear2002.begin.day;
acyear2002.end.month = acyear2002.begin.month;
acyear2002.end.year = acyear2002.begin.year;
```

- a structure can be assigned to another structure in one assignment:

```
acyear2002.end = acyear2002.begin;
acyear2003 = acyear2002;
```

Arrays

Statistical Calculations

- get student scores in an exam from user
- calculate mean, variance, standard deviation and absolute deviation

Outline: Statistics

- variables:
  - score: array for student scores
  - no_students: number of students
  - mean, variance, std_dev and abs_dev: desired results
  - total, sqr_total and abs_total: sums in the formula
  - i: loop counter

```
...
#define MAXSTUDENTS 100
...
int main(void)
{
    int score[MAXSTUDENTS];
    int no_students;
    float mean, variance,
          std_dev, abs_dev;
    float total = 0.0,
          sqr_total = 0.0,
          abs_total = 0.0;
    int i;

    // calculate and report
    return EXIT_SUCCESS;
}
```


Outline: Statistics

- to calculate the standard and absolute deviations, we need the mean
- user input and mean calculation can be done in the same loop
- both deviations can be calculated in the same loop

```
...
int main(void)
{
    // variable definitions

    // get the scores
      // and calculate the mean

    // calculate the standard
      // and absolute deviations

    return EXIT_SUCCESS;
}
```


Why Use an Array?

- to calculate the standard and absolute deviations, we also need all student scores
  - we may not "forget" after reading and adding to the sum
- imagine using separate variables for each score

Array Definition

type array_name[number];

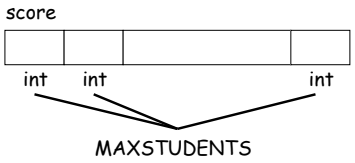
- type is the type of each element
- number is the number of elements

define an array variable score with MAXSTUDENTS elements where each element is an integer

```
int score[MAXSTUDENTS];
```

Memory Allocation

- memory space to hold this array will be allocated at the start of execution
- number of elements must be a constant



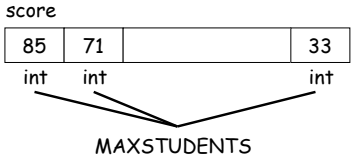
Memory Allocation

- we must estimate the maximum possible array size:
  - actual size is no_students, array size is MAXSTUDENTS
- actual size is smaller: waste memory
- actual size is larger: won't work → limitation
  - "in a class with at most 100 students"

Array Initialization

- initial value can be given in curly braces:

```
int score[MAXSTUDENTS] = { 85, 71, ..., 33 };
int score[MAXSTUDENTS] = { 0 };
```



Implicit Definition

- number of elements can be omitted if initial value is specified:

```
type array_name[] = { initial_values };
```

- compiler counts the number in the list and determines the array size
  - DANGEROUS!!!

Accessing Elements

- elements are accessed by specifying their index in square brackets:

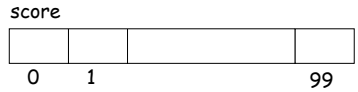
```
score[17] = 55;
x = 5 * score[43];
```

- index expressions can contain variables:

```
cin >> score[i];
```


Array Indexing

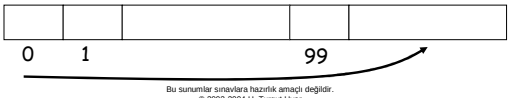
- the index of first element is always 0
- the index of last element is always 1 less than the array size
  - in an array with n elements, there is no element with index n or greater!



Bounds Checking

- the compiler does NOT check whether the index is within the bounds!!!
  - accessing an element: start from the beginning and proceed index elements
  - if not within bounds: overwrite other variable or segmentation fault

score



Array Loops

- typical loop for an array with size n:
  for (i = 0; i < n; i++)
- read into ith element, but tell the user i+1
- same counter variable for both loops (not always)

```
for (i = 0; i < no_students; i++) {
    cout << "Score of student "
         << i + 1 << ": ";
    cin >> score[i];
    total += score[i];
}

mean = total / no_students;
for (i = 0; i < no_students; i++) {
    sqr_total += sqr(score[i] - mean);
    abs_total += fabs(score[i] - mean);
}

variance = sqr_total /
           (no_students - 1);
std_dev = sqrt(variance);
abs_dev = abs_total / no_students;
```


Outline: String Reversing

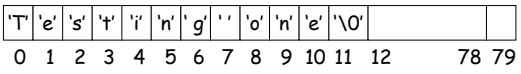
- get a sentence from the user, reverse and print
- variables:
  - sentence: sentence to be reversed
  - len: length of sentence
  - i: loop counter
  - tmp: temporary variable for swap

```
...
#define MAXLENGTH 80
...
int main(void)
{
    char sentence[MAXLENGTH];
    int len, i;
    char tmp;

    cout << "Enter the sentence: ";
    gets(sentence);
    // reverse and report
    return EXIT_SUCCESS;
}
```


Strings

- strings are character arrays
- the end of a string is marked using '\0'
  - a string can hold at most 1 less characters than the array size

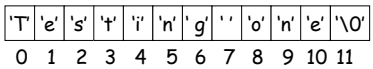
sentence



String Initialization

- strings variables can be initialized:
  char sentence[MAXLENGTH] = "Testing one";
  char sentence[] = "Testing one";
- same as:
  char sentence[12] = "Testing one"

sentence



String Input

- if string can contain spaces, cin will only take the part up to the first space
- use gets to read in a string with spaces
- gets does NOT check the size!
  - user can type a string longer than you have allocated for the string

String Library

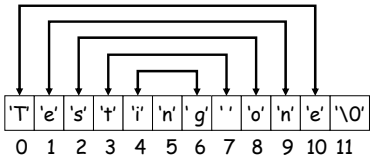
- since strings are arrays, you can not simply assign or check for equality
- use string library functions instead:
  - strlen(s): length of string s
  - strcpy(dest,src): copy content of src to dest
  - strcat(dest,src): append content of src to end of dest
    - dest must have enough memory allocated!

String Library

- strcmp(s1,s2): compare contents of s1 and s2:
  - s1 = s2: 0
  - s1 < s2: < 0
  - s1 > s2: > 0
- length checking versions of these functions: strncpy strncat strncmp

Reversing

- swap ith character from the beginning with ith character from the end
- continue until the middle



Outline: Matrix Multiplication

- variables:
  - left and right: matrices to be multiplied
  - product: result of multiplication
  - rl, cl, rr and cr: actual row and column sizes of left and right matrices
    - rr must always be equal to cl
  - i, j and k: loop counters

```
...
#define MAXSIZE 30
...
int main(void)
{
    int left[MAXSIZE][MAXSIZE],
        right[MAXSIZE][MAXSIZE],
        product[MAXSIZE][MAXSIZE] = { 0 };
    int rl, cl, cr;
    int &rr = cl;
    int i, j, k;

    // get matrices
    // multiply
    // print result
    return EXIT_SUCCESS;
}
```


Multidimensional Arrays

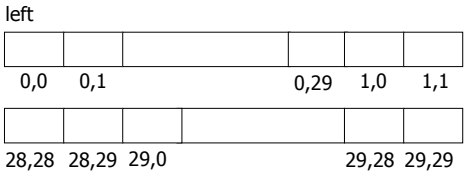
- a matrix is a two-dimensional array
- the size of each dimension is specified in square brackets
  - initial values in braces

define a two-dimensional array variable named "left", where the first dimension has size 30, the second dimension has size 30 and each element is an integer

```
int left[30][30];
```


Memory Layout

- the matrix will have rows * columns elements, placed consecutively in memory:



Multidimensional Array Indexing

- a matrix could also be considered as an array of arrays
  - left is an array with 30 elements where each element is an array of 30 integers
  - left[0] is an array of 30 integers
- each dimension is indexed separately
  - left[i][j]: element on the ith row and jth column

Multidimensional Array Loops

- nested loops:
  - i becomes 0
    - j varies from 0 to cl - 1
  - i becomes 1
    - j starts over and varies between 0 and cl - 1
  - ...
  - i becomes 29
    - j starts over

```
cin >> rl >> cl >> cr;

for (i = 0; i < rl; i++) {
    for (j = 0; j < cl; j++) {
        cout << ...
        cin >> left[i][j];
    }
}

for (i = 0; i < rr; i++) {
    for (j = 0; j < cr; j++) {
        cout << ...
        cin >> right[i][j];
    }
}
```


Matrix Addition

- both left and right matrices must have the same dimensions (rl and cl)
- the sum matrix will have the same dimensions
- for each entry in the result, add the corresponding elements of the left and right matrices

```
for (i = 0; i < rl; i++) {
    for (j = 0; j < cl; j++) {
        sum[i][j] = left[i][j] + right[i][j];
    }
}
```


Matrix Transposition

- on the left matrix
- the transposition matrix will have cl rows and rl columns

```
for (i = 0; i < cl; i++) {
    for (j = 0; j < rl; j++) {
        trans[i][j] = left[j][i];
    }
}
```

- how about storing the result in the same matrix?
- is this code correct?

```
for (i = 0; i < cl; i++) {
    for (j = 0; j < rl; j++) {
        left[i][j] = left[j][i];
    }
}
```

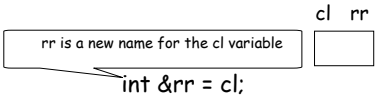

Matrix Multiplication

- the product matrix will have rl rows and cr columns
- for each entry in the result, multiply and add the ith row of the left matrix with the jth column of the right matrix
  - cl multiply / add operations

```
for (i = 0; i < rl; i++) {
    for (j = 0; j < cr; j++) {
        for (k = 0; k < cl; k++)
            product[i][j] +=
                left[i][k] * right[k][j];
    }
}
```


References

- a new name for a variable
  - same memory location
  - changing one also changes the other



Functions

Abstraction

- only one function until now: main
- for larger tasks, main task is divided into subtasks
- each (sub)task is implemented using a function

Parameters

- several input parameters
  - sqrt: one (number)
  - pow: two (base and power)
  - rand: none
  - srand: one (seed)
- at most one output parameter
  - sqrt: one (square root of number)
  - pow: one (result of exponentiation)
  - rand: one (random number)
  - srand: none

Function Call

- calling function (caller)
- called function (callee)
  - main function calls rand function
    - main is the calling function
    - rand is the called function
- after the called function is finished, the calling function resumes execution with the next statement

Parameter Passing

- calling function *passes* the input parameters to the called function
- called function *returns* the output parameter back to the calling function
- what to do with the return value?
  - assign it to a variable:
    ```
    number = rand();
    ```
  - use it in an expression:
    ```
    die = 1 + rand() % 6;
    ```
  - pass it to another function as input parameter:
    ```
    srand(time(NULL));
    ```


Type Compatibility

- type of the returned value should be compatible with its usage in the calling function:
  ```
  pow(x, y) % m
  ```
- input parameter list should also be compatible:
  - number of parameters
    ```
    pow(x, y, z)
    ```
  - types and order of parameters
    ```
    strlen(50)
    ```

Outline: Factorization

- main function gets the number from the user, factorizes it and prints the factors
- is_prime function checks whether its input parameter is prime or not
- next_prime function returns the first prime number greater than its input parameter

```
...
int next_prime(int prime);
...
int main(void)
{
   ...
}

bool is_prime(int cand)
{
   ...
}

int next_prime(int prime)
{
   ...
}
```


Outline: Main Function

- variables:
  - number: number to be factorized
  - factor: prime factor candidate
- we do not care HOW next_prime works!

```
...
int main(void)
{
   int number, factor = 2;

   cout << ...;
   cin >> number;
   while (number > 1) {
      while (number % factor == 0) {
         cout << factor << " ";
         number = number / factor;
      }
      factor = next_prime(factor);
   }
   cout << endl;
   return EXIT_SUCCESS;
}
```


Function Components

- header: WHAT does this function do?
  - name of function
  - types and order of input parameters
  - type of output parameter
  - compiler needs this information in order to do type checking
- body: HOW does this function work?
  - block of statements

Function Declaration

- specifying the function header:

  output_parameter_type function_name(input_parameter_list);
- note the semicolon!
- output parameter type can not be an array
- if no output parameter:

  void function_name(input_parameter_list);
- if no input parameter:

  output_parameter_type function_name(void);

Example Declarations

```
double sqrt(double x);
double pow(double x, double y);
int rand(void);
void srand(unsigned int seed);
int next_prime(int prime);
```

- careful: type of each parameter must be given separately
- INCORRECT:

```
double pow(double x, y);
```


Function Definition

- the first line is the header
  - without the semicolon!!!
- output parameter is passed back to the caller using *return*
  - multiple return statements allowed
- we do not care HOW is_prime works

```
...
int next_prime(int prime)
{
   int cand;

   cand = (prime == 2) ?
                3 : prime + 2;

   while (!is_prime(cand))
      cand += 2;
   return cand;
}
```

Function Definition

- 2 is prime
- any other even number is not prime
- try all odd numbers between 3 and the square root
  - if any of them divides the number, it is not prime
- otherwise it is prime

```
...
bool is_prime(int cand)
{
   int count;

   if (cand == 2)
      return true;
   if (cand % 2 == 0)
      return false;
   for (count = 3;
        count * count <= cand;
        count += 2) {
      if (cand % count == 0)
         return false;
   }
   return true;
}
```


Function Call Rule

- either the declaration
- or the definition

of the called function must be placed before the calling function

- main can call next_prime
- next_prime can call is_prime
- main can NOT call is_prime

- header files contain the function headers

```
...
int next_prime(int prime);
...
int main(void)
{
   ...
}

bool is_prime(int cand)
{
   ...
}

int next_prime(int prime)
{
   ...
}
```


Parameter Passing

- each expression listed in the function call is assigned to the corresponding input parameter
- call by value

- main:

  factor=next_prime(factor);
  - factor in main –> prime in next_prime
- factor:

  return cand;
  - cand in next_prime –> factor in main

Parameter Passing

- any expression can be passed as input parameter as long as types are compatible:

  is_prime(13)

  is_prime(cand + 1)
- same for output parameter:

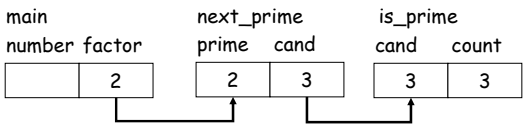
  return cand + 2;

Scope

- each variable can only be accessed in the function in which is it defined:
  - "factor in main"
- variables in different functions can have the same name
  - "cand in next_prime", "cand in is_prime"
- *scope*: the part of the program where that variable can be accessed
- variables "live" within their scope:
  - memory will be allocated when entering function
  - and freed when exiting the function

Local Variables

- any variable defined in the function body
- input parameters



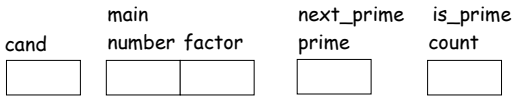
Global Variables

- any variable defined outside all functions
  - the scope is the entire program (all functions)
  - if any function changes the value, all functions will be affected

```
...
int cand = 2;
...
int main(void)
{
   ...
}

bool is_prime(void)
{
   ...
}

int next_prime(void)
{
   ...
}
```


Global Variables

- cand is accessible by main, next_prime and is_prime
  - what are the new functions of next_prime and is_prime?



Call by Reference

```
void swap(int x, int y)
{
   int tmp;

   tmp = x;
   x = y;
   y = tmp;
}

int main(void)
{
   int m, n;
   ...
   swap(m, n);
   ...
}
```

- changes to input parameters in the called function do not affect variables in the calling function
  - x and y in the swap function are swapped
  - m and n in the main function do not change
- if they should affect:
  void swap(int &x, int &y)

Outline: GCD using Factorization

- main function:
  - get numbers from the user
  - call factorize to factor the first number
  - call factorize to factor the second number
  - call gcd_factors to find the common factors
  - compute and print the gcd

```
...
void factorize(...);
void gcd_factors(...);

int main(void)
{
   ...
}

void factorize(...)
{
   ...
}

void gcd_factors(...)
{
   ...
}
```


Factor Data Structure

- a factor consists of two integer fields:
  - base, power
- factorize function is similar to main in the previous example
  - counts the power value
  - next_prime and is_prime are identical

```
#include <iostream>
...

#define MAXFACTOR 50

struct factor_s {
    int base, power;
};
typedef struct factor_s factor_t;

...

int main(void)
{

...
```


Outline: main

- variables:
  - number1, number2: numbers to be factorized
  - factors1, factors2: factors of number1 and number2
  - factors3: factors of gcd
  - n1, n2, n3: actual element counts of factors1, factors2 and factors3
- factorize should return the list of factors
  - arrays can not be returned
  - number of elements should also be returned
- same goes for gcd_factors

```
...
int main(void)
{
   int number1, number2;
   factor_t factors1[MAXFACTOR],
            factors2[MAXFACTOR],
            factors3[MAXFACTOR];
   int n1, n2, n3;
   long int gcd = 1L;
   int i;

   ...
}
...
```

Using Input Parameters for Output

- use the input parameters to send the necessary values to the calling function
- redefine factorize:
  - factors the first input parameter, stores the factors in the elements of the second input parameter (an array) and stores the number of elements in the third input parameter
  - does not return any value

```
int main(void)
{
    ...

    factorize(number1, factors1, n1);
    factorize(number2, factors2, n2);
    gcd_factors(factors1, n1,
                factors2, n2,
                factors3, n3);
    for (i = 0; i < n3; i++)
        gcd = gcd * pow(
            (double) factors3[i].base,
            (double) factors3[i].power);
     ...
}
```


Passing Arrays as Input Parameters

- to indicate that an input variable is an array: square brackets after the name
  ..., factor_t factors[], ...
- unless explicitly prohibited, elements of an input array CAN be changed
  - use const to prohibit

```
...
void factorize(int x, factor_t factors[], int &n)
{
    int factor = 2;

    n = 0;
    while (x > 1) {
        if (x % factor == 0) {
            factors[n].base = factor;
            factors[n].power = 0;
            while (x % factor == 0) {
                factors[n].power++;
                x = x / factor;
            }
            n++;
        }
        factor = next_prime(factor);
    }
}
...
```


Outline: gcd_factors

- we can assume that factors1 and factors2 are in ascending order
- i1 and i2: index of next element in factors1 and factors2

```
void gcd_factors(const factor_t factors1[], int n1,
                 const factor_t factors2[], int n2,
                 factor_t factors[], int &n)
{
    int i1 = 0, i2 = 0;

    n = 0;
    while ((i1 < n1) && (i2 < n2)) {
        // compare the factors
        // and choose one if common
    }
}
```


Finding the Common Factors

- at each step:
  - advance the one with the smaller base
  - if bases are equal, choose the one with smaller power and advance both

```
    n = 0;
    while ((i1 < n1) && (i2 < n2)) {
        if (factors1[i1].base factors[i2].base)
            i1++;
        else if (factors1[i1].base > factors[i2].base)
            i2++;
        else {
            factors[n].base = factors[i1].base;
            if (factors[i1].power factors2[i2].power)
                factors[n].power = factors1[i1].power;
            else
                factors[n].power = factors2[i2].power;
            i1++;
            i2++;
            n++;
        }
    }
```


C/C++ Differences

- functions with the same name:
  - in C: not allowed
  - in C++: allowed if input parameters are different
- default parameters:
  - assume 0 for start if not specified
  - find_char can take 2 or 3 parameters

```
int find_char(char s[], char c,
              int start = 0)
{
    int index = -1, i;

    for (i = start; s[i] != '\0'; i++) {
        if (s[i] == c) {
            index = i;
            break;
        }
    }
    return index;
}
```


Pointers

Static Variables

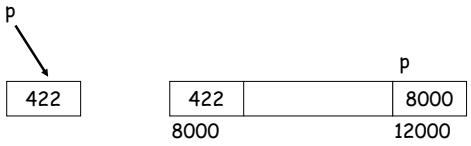
- size must be known at compile-time
  - size of array must be constant
  - waste of memory space- limitation
- memory is allocated on entering the function and released on exit
  - variables of main live throughout the whole execution
  - may not be needed all at once

Dynamic Variables

- allocate memory when needed, release when done
  - do memory management yourself
  - error- prone process, especially for beginners
- allocate as much memory as is needed:
  - determine size at run- time instead of compile time

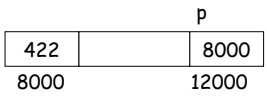
Pointers

- pointer is an ordinary variable
- its value is the address of a memory cell



Pointer Operators

- pointer: an address
- pointer's content: a value of some type
  - * operator: dereferencing
- address of a variable:
  - & operator
- special value: NULL
  - is not a valid address
  - can not be dereferenced



p: 8000
*p: 422
&p: 12000

Pointer Variables

- itself an address
  - similar to an integer
  - allocate enough memory to hold an address
- how will its content be interpreted?
- pointer variable definition:
  type *name;

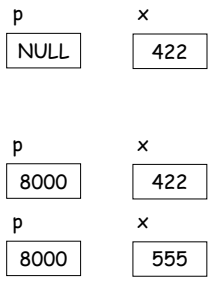
Pointer Examples

int *p = NULL;
int x = 422;

- assume that address of x is 8000

p = &x;

*p = 555;



Outline: Statistics

- same statistical calculations as in static arrays, this time using dynamic arrays
- all variables the same except:
  - score: a pointer to an integer

```
int main(void)
{
    int *score = NULL;
    int no_students;
    float mean, variance,
          std_dev, abs_dev;
    float total = 0.0,
          sqr_total = 0.0,
          abs_total = 0.0;
    int i;

    // calculate and report
    return EXIT_SUCCESS;
}
```

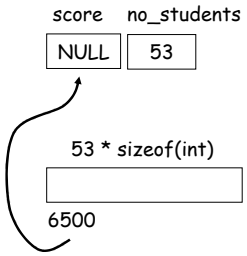

Memory Allocation

- memory MUST be allocated before used

new type[number]

score = new int[no_students];

- CAN contain variables
- allocates contigous memory area
- gives starting address of raw area



Memory Management

- to release allocated memory:

  delete name;
  - size not specified

  delete score;
- no new and delete in C:
  - allocation: malloc(requested size in bytes)
    - return address should be cast:

      score = (int *) malloc(no_students * sizeof(int));
  - release: free

    free(score);
  - need the stdlib.h header file

Dynamic Arrays

- dynamic arrays are accessed the same way as static arrays
- name of array is a pointer to its first variable:
  - score[0] is the same as *score
- pointer arithmetic:
  - score[1] is the same as *(score + 1)
  - score[n] is the same as *(score + n)

```
...
int *score = NULL;

cout << ...;
cin >> no_students;
score = new int[no_students];
for (i = 0; i < no_students; i++) {
    cout << ...;
    cin >> score[i];
    total = total + score[i];
}
...
delete score;
...
```


Pointer Parameters

- when passing arrays as input parameters to functions, always a pointer is passed
- the following are identical:

  void factorize(int number, factor_t factors[], int &n)

  void factorize(int number, factor_t *factors, int &n)
- unlike arrays, pointers can be returned:

  char *

Outline: Morse Encoding

- main function:
  - gets a word from the user
  - calls encode to get the encoded string
  - prints and exits
- encode function:
  - takes a string as input parametes
  - creates a new (encoded) string
  - returns the newly created string
  - original string is not changed: const

```
...
char *encode(const char *s);

int main(void)
{
    char word[MAXLENGTH];
    char *morse = NULL;

    cout << ...;
    cin >> word;
    morse = encode(word);
    cout << morse ..;
    delete morse;
    return EXIT_SUCCESS;
}
```

Memory Management

- encode function allocates the space for the new string
- can not release it
- main releases it

```
char *encode(const char *s)
{
   static char encoding...;
   char *morse = NULL;
   int i;

   morse = new char[MAXLENGTH];
   morse[0] = '\0';
   for (i = 0; s[i] != '\0'; i++) {
      strcat(morse,
             encoding[s[i] – 'a']);
      strcat(morse, " ");
   }
   return morse;
}
```


Static Local Variables

- the encoding array in the encode function does not need to be created and destroyed at each call
- no need to make it global: main will not use it
- make it static:
  - lives throughout the whole program
  - but only accessible locally

```
char *encode(const char *s)
{
   static char encoding[][5] = {
      ".-", "-...", ...};

   ...
}
```


Call by Address

- no references in C
- to change the value of the variable passed as input parameter in the called function:
  - caller sends the address of the variable using &
  - callee takes input parameter as pointer
  - callee changes content of pointer
- call by value, where values are addresses

Call by Address

```
void swap(int *x, int *y)
{
   int tmp;

   tmp = *x;
   *x = *y;
   *y = tmp;
}

int main(void)
{
   int m, n;
   ...
   swap(&m, &n);
  ...
}
```


Input / Output

Output

- no cout in C, use printf instead:

  cout << "Hello, world!" << endl;

  printf("Hello, world!\n");
  - \n instead of endl
- printf syntax:
  - printf(format string, expression list);
  - format string controls the output
  - print strings and/or values of expressions

Format String

- strings are transferred directly to output
- % followed by a symbol is a format specifier
  - %d: decimal, %x: hexadecimal, %f: float
  - %c: character, %s: string
- \ followed by a symbol is a special symbol
  - \n: newline
  - \t: tab
  - \\: backslash

Format Specifiers

- there must be one specifier for each to be printed
- the expression will be printed at the location its specifier is written in the format string
- the specifier controls how the expression will be printed

```
cout << "The area of a circle with radius " << radius <<
    " is " << 3.14 * radius * radius << endl;
printf("The area of a circle with radius %d is %f\n",
    radius, 3.14 * radius * radius);
```


Input

- no cin in C, use scanf instead
- syntax very similar to printf

```
cin >> radius;
scanf("%d", &radius);
```

- call by address
- when reading a string:

```
cin >> word;
scanf("%s", word);
```


Outline: Statistics Once Again

- get the exam scores from a file
- write the results to a file
- get the file names from the command line
  - not interacting with the user

```
stat3 scores.txt results.txt
```

```
...
#include <stdio.h>
...
int main(int argc, char *argv[])
{
    // variable definitions

    //  check usage
    //  get scores from input file and
    //    compute total
    //  compute statistics
    //  write results to output file

    return EXIT_SUCCESS;
}
```


Command Line Parameters

- passing parameters to the main function
  - argc: number of arguments
  - argv: array of arguments where each argument is a string
  - argv[0] is the name of the program itself: stat3
  - argv[1] is the first parameter: scores.txt
  - argv[2] is the second parameter: results.txt

```
int main(int argc, char *argv[])
{
    ...

    if (argc != 3) {
        printf("Usage: %s input_file
                output_file\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    ...
}
```


Files

- file variables are of type FILE *
  - determines where to read and write in the file
- stages of file processing:
  - open
  - read, write, ...
  - close

```
int main(int argc, char *argv[])
{
    ...
    FILE *infile, *outfile;

    // open the input file
    // read from the input file
    // close the input file

    // open the output file
    // write results to the output file
    // close the output file
}
```

Opening Files

- to open a file: fopen
  fopen(path, mode)
- path: name of file on disk
- mode:
  - "r": read-only
  - "w": write-only
  - "r+" "w+": read and write
  - "w" and "w+" create the file if it does not exist, truncate it if it does exist
- to close a file: fclose

```
int main(int argc, char *argv[])
{
   ...
   FILE *infile, *outfile;

   infile = fopen(argv[1], "r");
   // read from the input file
   fclose(infile);

   outfile = fopen(argv[2], "w");
   // write results to the output file
   fclose(outfile);
   ...
}
```


Reading from Files

- file pointer is placed at the beginning of file on open
  - each read / write advances the pointer
- to read: fscanf
  - same as scanf, just reads from file
  - moves pointer to next line: repeated reads
- to check end-of-file: feof

```
int main(int argc, char *argv[])
{
   ...
   no_students = 0;
   while (1) {
      fscanf(infile, "%d",
             &score[no_students]);
      if (feof(infile))
         break;
      total = total +
              score[no_students]
      no_students++;
   }
   ...
}
```


Writing to Files

- to write: fprintf
  - same as printf, just writes to file
  - moves pointer to next line: repeated writes

```
int main(int argc, char *argv[])
{
   ...
   fprintf(outfile,
      "Number of students: %d\n",
      no_students);
   fprintf(outfile,
      "Mean: %f\n", mean);
   fprintf(outfile,
      "Variance: %f\n", variance);
   fprintf(outfile,
      "Standard deviation: %f\n",
      std_dev);
   ...
}
```


Standard I/O Streams

- standard input: stdin
  - scanf is the same as fscanf(stdin,...)
- standard output: stdout
  - printf is the same as fprintf(stdout,...)
- standard error: stderr

Error Messages

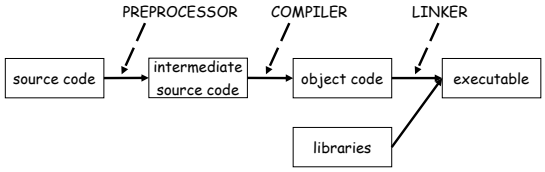
- it is better practice to send error messages to the error stream:
  cerr << message;
  fprintf(stderr, message)
- if error is critical, exit
- perror displays standard error messages

```
int main(int argc, char *argv[])
{
   ...
   infile = fopen(argv[1], "r");
   if (infile == NULL) {
      fprintf(stderr,
         "Input file could not be
          opened.\n");
      exit(EXIT_FAILURE);
   }
   ...
}
```


Preprocessor

Preprocessing

- source code goes through a preprocessing stage before compiling



Preprocessor

- removes the comments
- processes the preprocessor directives:
  - #define: define constants or macros

  ```
  #define PI 3.14
  ```
  - #include: place the file at this point in the source

  ```
  #include <stdlib.h>
  ```


Macros

- repeated code segments not worth making a function

  ```
  sqr_total = sqr_total +
        (score[i] – mean) * (score[i] – mean);
  ```
  - writing a function to compute square is overkill

    ```
    #define sqr(x) (x) * (x)
    ```
  - replace each occurrence of the macro by its expanded equivalent

    ```
    sqr_total = sqr_total + sqr(score[i] - mean)
    ```


Expanding Macros

- macros are just dumb substitutions!

  ```
  #define sqr(x) x * x
  sqr_total = sqr_total + sqr(score[i]-  mean);
  ```
  - would be:

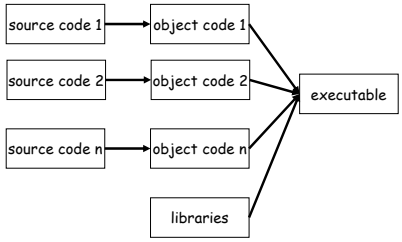
  ```
  sqr_total = sqr_total +
        score[i] – mean * score[i] – mean;
  ```


Projects

- source file gets too big:
  - hard to manage
  - takes longer to compile and link
- split into multiple source files
  - easier to manage
  - compiling takes less time
- group relevant functions in the same file
  - computations / user interface

Building Projects

- first compile separately then link



Outline: Multiple Sources

- program finds:
  - factors of a number
  - gcd of two numbers
  - lcm of two numbers
- functions splitted into two source files:
  - user interaction in file project.cpp
  - computations in ops.cpp

Outline: project.cpp

- factorize, gcd and lcm functions are called in project.cpp
- but they are defined in ops.cpp
- compiler needs the declarations
- also needs the struct definition for a factor

```
int main(void)
{
   // variables

   while (true) {
      // draw menu and get choice
      switch (choice) {
         ...
         case 3: ... factorize() ...
            ... factors[i].base ...
         case 4: ... gcd() ...
         case 5: ... lcm() ...
         ...
      }
   }
}
```


Declarations

- what if there is another source file that also calls these functions?
- the struct definition for factor is needed both in project.cpp and ops.cpp
  - write twice?

```
...
void factorize(...);
void gcd(...);
void lcm(...);
...
int main(void)
{
   ...
}
```


Header File

- put shared declarations in a header file: ops.h
- include the header file in each source file that needs the declarations

  include "ops.h"
  - "" instead of <>

```
#define MAXFACTOR 50

struct factor_s {
   int base, power;
};
typedef struct factor_s factor_t;

void factorize(int x, factor_t factors[],
               int &n);
int gcd(int number1, int number2);
int lcm(int number1, int number2);
```


Outline: ops.cpp

- declarations of is_prime, next_prime, gcd_factors and lcm_factors are not in ops.h
- they are not accessible from other source files
- header file is the interface of the source file

```
...
#include "ops.h"
...
void gcd_factors(...);
void lcm_factors(...);

int gcd() ...
int lcm() ...
bool is_prime() ...
int next_prime() ...
void factorize() ...
void gcd_factors() ...
void lcm_factors() ...
```


Header Files

- what to put in header files?
  - function declarations
  - type definitions
  - global variables: extern
  - constant and macro definitions
  - including other files
- what NOT to put in header files?
  - variable definitions
  - function definitions (bodies)

Multiple Inclusion

- to prevent an include file from being included more than once: #ifndef
- on first inclusion:
  - OPS_H is not defined
  - OPS_H gets defined (value not important)
  - make other declarations
- on second inclusion:
  - OPS_H is defined, skip the rest

```
#ifndef OPS_H
#define OPS_H

#define MAXFACTOR 50
...
int gcd(int number1, int number2);
int lcm(int number1, int number2);

#endif
```


Recursion

Recursion

- the problem is expressed in terms of itself:
  - the gcd of a and b is equal to the gcd of b and a % b
  - the factorial of x is x times the factorial of x – 1
- for this approach to work:
  - the subproblem must be of smaller size
  - there has to be a base case where the solution is known

Recursive Factorial

- x – 1 is smaller than x
  - gets smaller at each step
- the factorial of 0 is 1
- using iteration is faster and cheaper

```
int factorial(int x)
{
    if (x == 0)
        return 1;
    else
        return x * factor(x - 1);
}
```

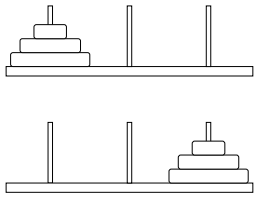

Recursive GCD

- b is smaller than a
- a % b is smaller than b
- if b is 0 the gcd is a
- using iteration is faster and cheaper

```
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

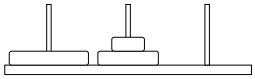

Towers of Hanoi

- 3 golden needles, 64 marble discs
- each disk is on a larger disk
- move all disks one by one from needle 1 to needle 3
- never put a disk on a smaller disk



Towers of Hanoi

- consider the largest disk:
  - all of the other disks must be on needle 2 and needle 3 must be empty
- move 64 disks from 1 to 3:
  - move 63 disks from 1 to 2
  - move a disk from 1 to 3
  - move 63 disks from 2 to 3
- same problem with 63 disks instead of 64



Towers of Hanoi

- move n disks from a to b using c:
  - move n – 1 disks from a to c using b
  - move a disk from a to b
  - move n – 1 disks from c to b using a
- base case: n = 0
- very elegant solution: no unnecessary move!

```
void move(int n, int a, int b, int c)
{
   if (n 0) {
      move(n – 1, a, c, b);
      cout << "Move a disk from "
           << a << " to " << b
           << endl;
      move(n – 1, c, b, a);
   }
}

main:
   move(64, 1, 3, 2);
```