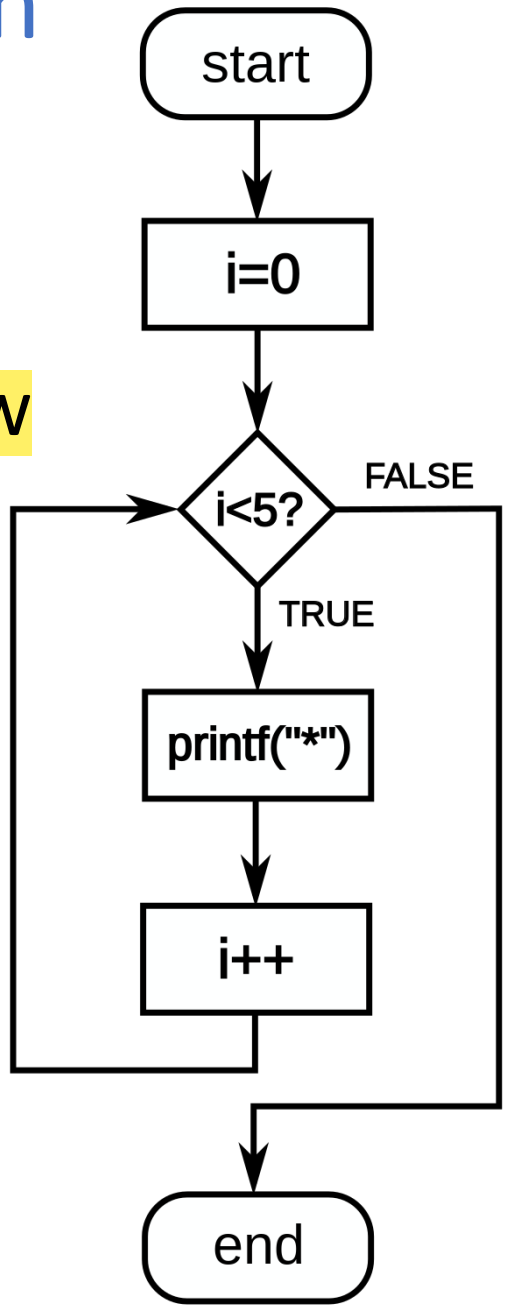
CS 101
Computer Programming and Utilization

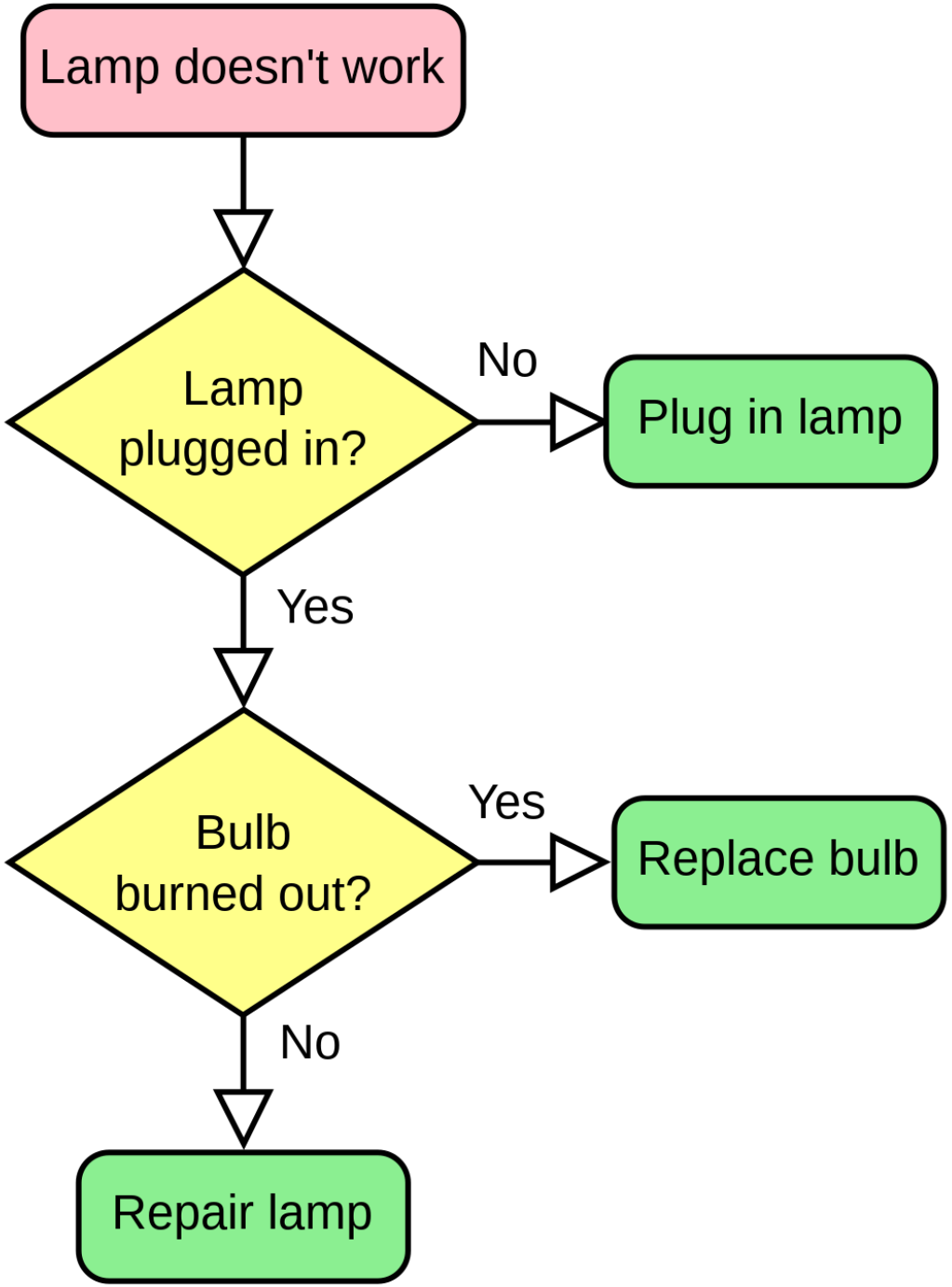
Conditions, Loops

Suyash P. Awate

Conditional Execution

- Flowcharts are often based on conditions that direct/change control flow leading to conditional execution of instructions/code

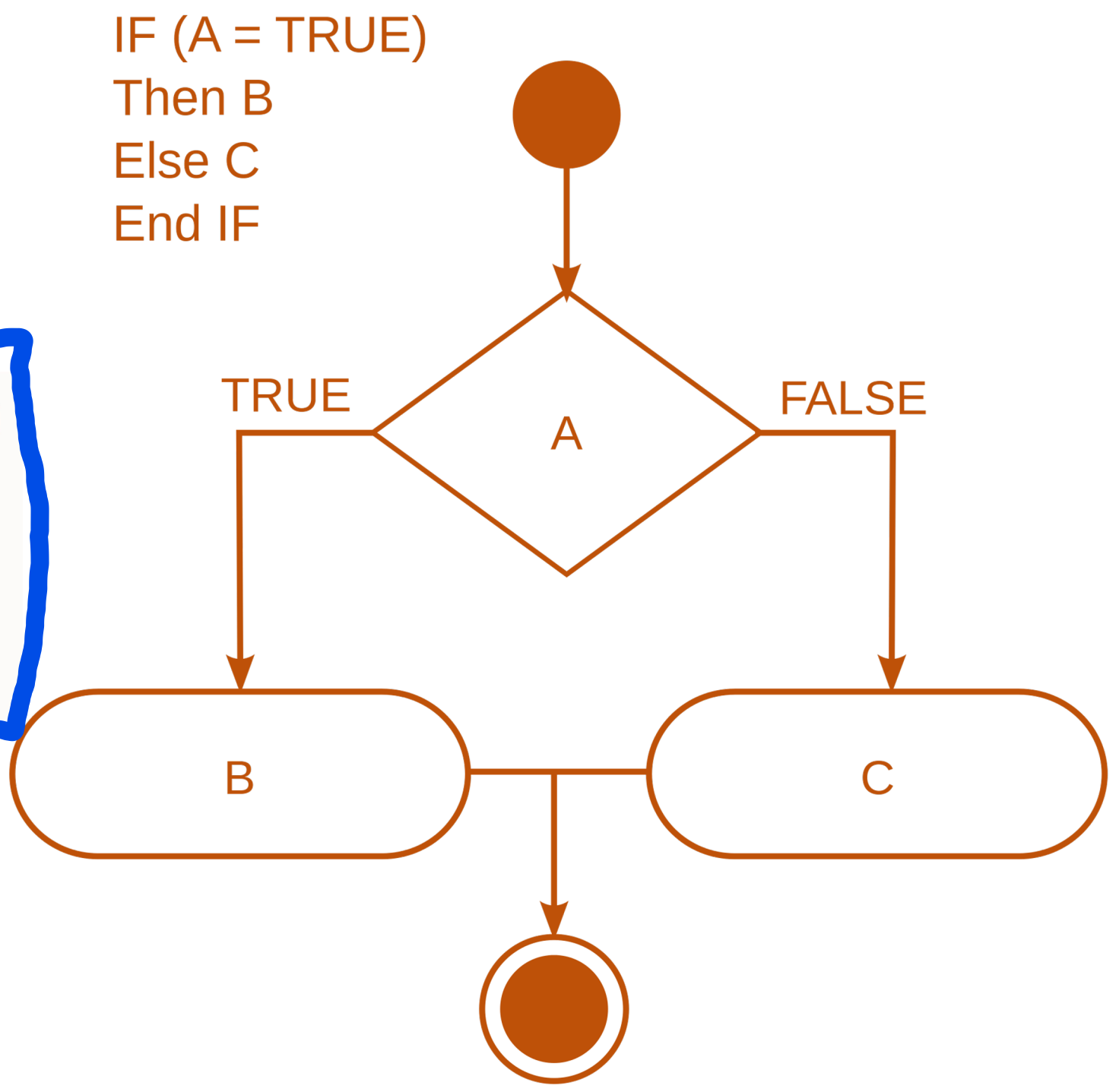




Conditional Execution

- Pseudocode describing an "if statement" in typical programming languages

```
If (boolean condition) Then
   (consequent)
Else
   (alternative)
End If
```



Conditional Execution

- In Simplecpp
  - if (condition) consequent
  - if (condition) consequent else alternative
- Condition: a simple form
  - Expression1 relationalOperator Expression2
- Relational operators: <, >, <=, >=, ==, !=
- Condition: examples
  - “A == 5”
  - “A < 5”
  - “A != 5”
  - If value of A is 5, then first condition **evaluates** to boolean “true”, and other conditions evaluate to boolean “false”

Conditional Execution

- Condition: more complex forms
  - Conjunction (“and”) of multiple conditions
    - e.g., condition1 && condition2
    - && = logical AND operator; applies to Boolean variables
  - Disjunction (“or”) of multiple conditions
    - e.g., condition1 || condition2 || condition3
    - || = logical OR operator; applies to Boolean variables
  - Negation
    - !condition
    - ! = negation operator; applies to Boolean variables

Conditional Execution

- Consequent and alternative can be single statements or blocks
- Example
  - if (a == 5)
    {
      statement 1;
      statement 2;
    }

```
if (condition)
{
  consequent
}

else {
alternative
}
```

Conditional Exec

- Example1
  - Income tax calculation

```
main_program{
  float income; // in rupees.
  float tax;    // in rupees.

  cout << "What is your income in rupees? ";
  cin >> income;

  if(income <= 180000) tax = 0;                          //
  if((income >  180000) && (income <= 500000))           //
    tax = (income - 180000)* 0.1;
  if((income >  500000) && (income <= 800000))           //
    tax = 32000+(income - 500000)* 0.2;
  if(income > 800000)                                    //
    tax = 92000+(income - 800000)* 0.3;

  cout << "Tax is: " << tax << endl;
}
```

Conditional Execution

- Simplecpp if statement: complex form

```
if (condition1) consequent1
else if (condition2) consequent2
else if (condition3) consequent3
...
else if (conditionn) consequentn
else alternate
```

Else if remembers previous if conditions.

Conditional Execution

- Example2
  - Income tax calculation
- Which one executes faster: example1 or example2 ?

```
main_program{
  float income, tax;

  cout << "What is your income? ";
  cin >> income;

  if(income <= 180000) tax = 0;          //
  else if(income <= 500000)              //
    tax = (income - 180000)* 0.1;
  else if(income <= 800000)              //
    tax = 32000+(income - 500000)* 0.2;
  else
    tax = 92000+(income - 800000)* 0.3;


  cout << "Tax is: " << tax << endl;
}
```

Conditional Execution

- Turtle controller

```
main_program{
  char command;
  turtleSim();

  repeat(100){
    cin >> command;
    if (command == 'f') forward(100);
    else if (command == 'r') right(90);
    else if (command == 'l') left(90);
    else cout << "Not a proper command, " << command << endl;
  }
}
```

Conditional Execution

- Building buttons
- Detecting clicks on a button

```
main_program{
  initCanvas();

  const float bFx=150,bFy=100, bLx=400,bLy=100, bWidth=150,bHeight=50;
  Rectangle buttonF(bFx,bFy,bWidth,bHeight), buttonL(bLx,bLy,bWidth,bHeight);

  Text tF(bFx,bFy,"Forward"), tL(bLx,bLy,"Left Turn");
  Turtle t;

  repeat(100){
    int clickPos = getClick();
    int cx = clickPos/65536;
    int cy = clickPos % 65536;

    if(bFx-bWidth/2<= cx && cx<= bFx+bWidth/2 &&
       bFy-bHeight/2 <= cy && cy <= bFy+bHeight/2) t.forward(100);

    if(bLx-bWidth/2<= cx && cx<= bLx+bWidth/2 &&
       bLy-bHeight/2 <= cy && cy <= bLy+bHeight/2) t.left(10);
  }
}
```

Conditional Execution

- Switch statement (helps avoid too many if-then-else statements)
  - Intended to be easier to read, understand, maintain, debug
  - "expression", constant1, constant2, … must be of type int
  - First evaluate expression
  - Second
    If value = constantX, then control passes to statements group(X)
    - If "break" not present, execution falls through the next group
    - Not having a "break" can make code complex/confusing; try to avoid
  - If value doesn't match any constant, default group executed

```
switch (expression){
   case constant1:
      group(1) of statements usually ending with ‘‘break;’
   case constant2:
      group(2) of statements usually ending with ‘‘break;’
   ...
   default:
      default-group of statements
}
```

Conditional Execution

- Switch

```
main_program{
  char command;
  turtleSim();

  repeat(100){
    cin >> command;
    switch(command){
      case 'f': forward(100);
                break;
      case 'r': right(90);
                break;
      case 'l': left(90);
                break;
      default: cout << "Not a proper command, " << command << endl;
    }
  }
}
```

Conditional Execution

- Switch without break statements in some cases

```
main_program{
  int month;
  cin >> month;
  switch(month){
    case 1:  // January
    case 3:  // March
    case 5:  // May
    case 7:  // July
    case 8:  // August
    case 10: // October
    case 12: // December
             cout << ‘‘This month has 31 days.\n’’;
             break;
    case 2:  // February
             cout << ‘‘This month has 28 or 29 days.\n’’;
             break;
    case 4:  // April
    case 6:  // June
    case 9:  // September
    case 11: // November
             cout << ‘‘This month has 30 days.\n’’;
             break;
    default: cout << ‘‘Invalid input.\n’’;
  }
}
```

Conditional Execution

- Conditional expression

```
condition ? consequent-expression : alternate-expression
```

  - First evaluates condition
  - If condition evaluates to true, then consequent-expression is evaluated
    - Value of consequent-expression becomes value of conditional expression
  - If condition evaluates to false, then alternate-expression is evaluated
    - Value of alternate-expression becomes value of conditional expression
  - Example

```
int marks; cin >> marks;
int actualmarks = (marks > 100) ? 100 : marks;
char grade = (marks >= 35) ? 'p' : 'f';
```

?

Conditional Execution

- Conditional expressions can be nested
  - Difficult to read and interpret; avoid

```
if(income <= 180000) tax = 0;          //
else if(income <= 500000)              //
  tax = (income - 180000)* 0.1;
else if(income <= 800000)              //
  tax = 32000+(income - 500000)* 0.2;
else
  tax = 92000+(income - 800000)* 0.3;
```

```
main_program{
  float income; cin >> income;

  cout << (                                  //  **
          income <= 180000 ? 0 :
          income <= 500000 ? (income - 180000) * 0.1 :
          income <= 800000 ? 32000 + (income - 500000) * 0.2 :
          92000 + (income - 800000) * 0.3
         )                                   //  **
       << endl;
}
```

Conditional Execution

- Logical variables and "data"
  - Data can have Boolean (categorical) values
  - Example
    - Logical variable indicating low/mid/high income

```
float income; cin >> income;
bool lowIncome, midIncome, highIncome;
lowIncome = (income <= 180000);
midIncome = (income > 180000) && (income <= 800000);
highIncome = (income > 800000);
```

    - Recall: conditions evaluate to boolean true or false
    - Suppose income had value 200,000. Then what would be values for bool variables ?
  - Example
    - Logical variable indicating letter case

```
char in_ch;
bool lowerCase;
cin >> in_ch;
lowerCase = (in_ch >= 'a') && (in_ch <= 'z');
```

    - "cout << true;" outputs "1" on the screen; "cout << false;" outputs "0" on the screen.

Conditional Execution

- For any logical value b
  - b && true = b
  - b || false = b
- Distributivity of && (conjunction) over || (disjunction)
  - Similar to those in logic
    - https://en.wikipedia.org/wiki/Distributive_property#Propositional_logic

$$
\begin{array}{lll}
(P \wedge (Q \vee R)) & \Leftrightarrow ((P \wedge Q) \vee (P \wedge R)) & \text{Distribution of conjunction over disjunction} \\
(P \vee (Q \wedge R)) & \Leftrightarrow ((P \vee Q) \wedge (P \vee R)) & \text{Distribution of disjunction over conjunction} \\
(P \wedge (Q \wedge R)) & \Leftrightarrow ((P \wedge Q) \wedge (P \wedge R)) & \text{Distribution of conjunction over conjunction} \\
(P \vee (Q \vee R)) & \Leftrightarrow ((P \vee Q) \vee (P \vee R)) & \text{Distribution of disjunction over disjunction}
\end{array}
$$

- De Morgan's laws
  - not (A or B) = (not A) and (not B)
  - not (A and B) = (not A) or (not B)

Conditional Execution

- Augustus De Morgan
  - British mathematician, logician
  - Born in Madurai
  - Introduced mathematical induction
  - In 1823, at age 16, entered Cambridge Univ.
  - At age 22, professor of math at London Univ.
- London Univ. established on ideas on religious neutrality
  - “Oxford and Cambridge were so guarded by theological tests that no Jew or Dissenter outside the Church of England could enter as a student, still less be appointed to any office”
    - en.wikipedia.org/wiki/Universities_Tests_Act_1871
    - ox.ac.uk/about/oxford-people/women-at-oxford/centenary-womens-timeline

Conditional Execution

- Is a number prime
- How can you improve this code ?
  - Repeating (x-2) times is redundant
  - Need a way to stop repeat loop when factor is found

```
main_program{  //Decide if x is prime.
   int x; cin >> x;

   int i=2;
   bool found = false;
   repeat(x-2){
      found = found || (x % i) == 0;
      i = i+1;
   }

   if (found) cout << x << " is composite." << endl;
   else cout << x << " is prime." << endl;
}
```

Conditional Execution

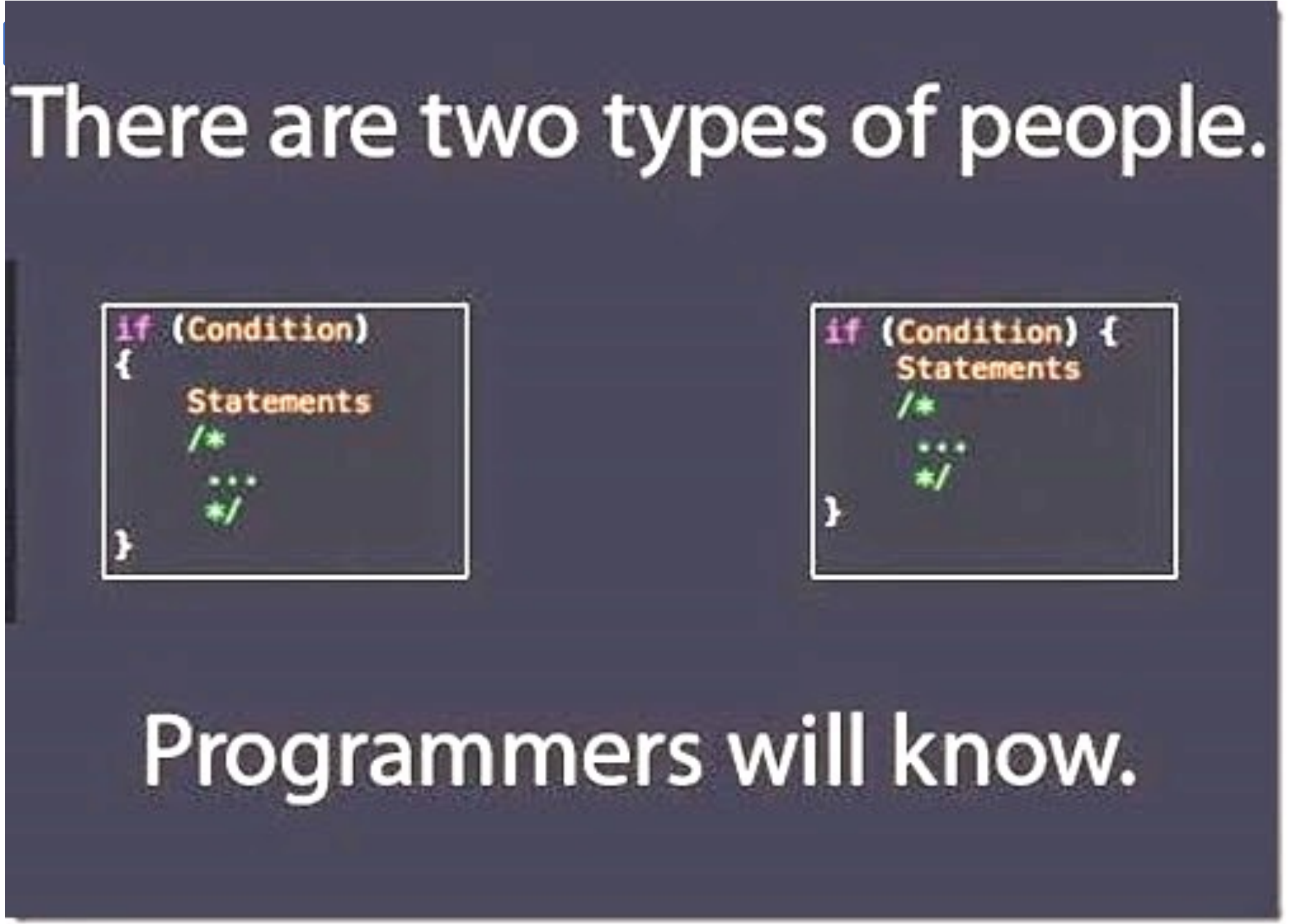
- Be careful with “=” and “==”
  - Consider
    - if (a == 5) cout << “found”;
    - if (a = 5) cout << “found”;
  - What will be the result of executing both statements ?
  - Expression “a=5” evaluates to 5, which will then be converted to Boolean
    - Rule for conversion:
      any non-zero integer value converts to Boolean true value;
      a zero integer value converts to Boolean false value

Conditional Execution

- Be careful about nested if statements
  - Consider `if(a > 0) if(b > 0) c = 5; else c = 6;`
  - Problem: the “else” block associated with which “if” ?
    - Rule: else joins with the innermost if
    - Can easily lead to confusion in human reading and interpretation
  - Always be unambiguous and explicit, by using braces

```
if(a > 0) {if(b > 0) c = 5; else c = 6;}

if(a > 0) {if(b > 0) c = 5;} else c = 6;
```

Conditional



Practice Examples for Lab: Set 5

- 1

Write a program that reads 3 numbers and prints them in non-decreasing order.

- 2

Write a program which takes as input a number denoting the `year`, and says whether the year is a leap year or not a leap year.

- 3

Write a program that takes as input 3 numbers a, b, c and prints out the roots of the quadratic equation $ax^2 + bx + c = 0$. Make sure that you handle all possible values of a, b, c without running into a division by zero or having to take the square root of a negative number. Even if the roots are complex, you should print them out suitably.

- 4

Write a program that reads in 3 characters. If the three characters consist of two digits with a '.' between them, then your program should print the square of the decimal number represented by the characters. Otherwise your program should print a message saying that the input given is invalid.

Practice Examples for Lab: Set 5

- 5

Write a program which prints all the prime numbers smaller than n, where n is to be read from the keyboard.

- 6

en.wikipedia.org/wiki/Perfect_number

A number is said to be perfect if it is equal to the sum of all numbers which are its factors (excluding itself). So for example, 6 is perfect, because it is the sum of its factors 1, 2, 3. Write a program which determines if a number is perfect. It should also print its factors.

Loops

- A motivating example

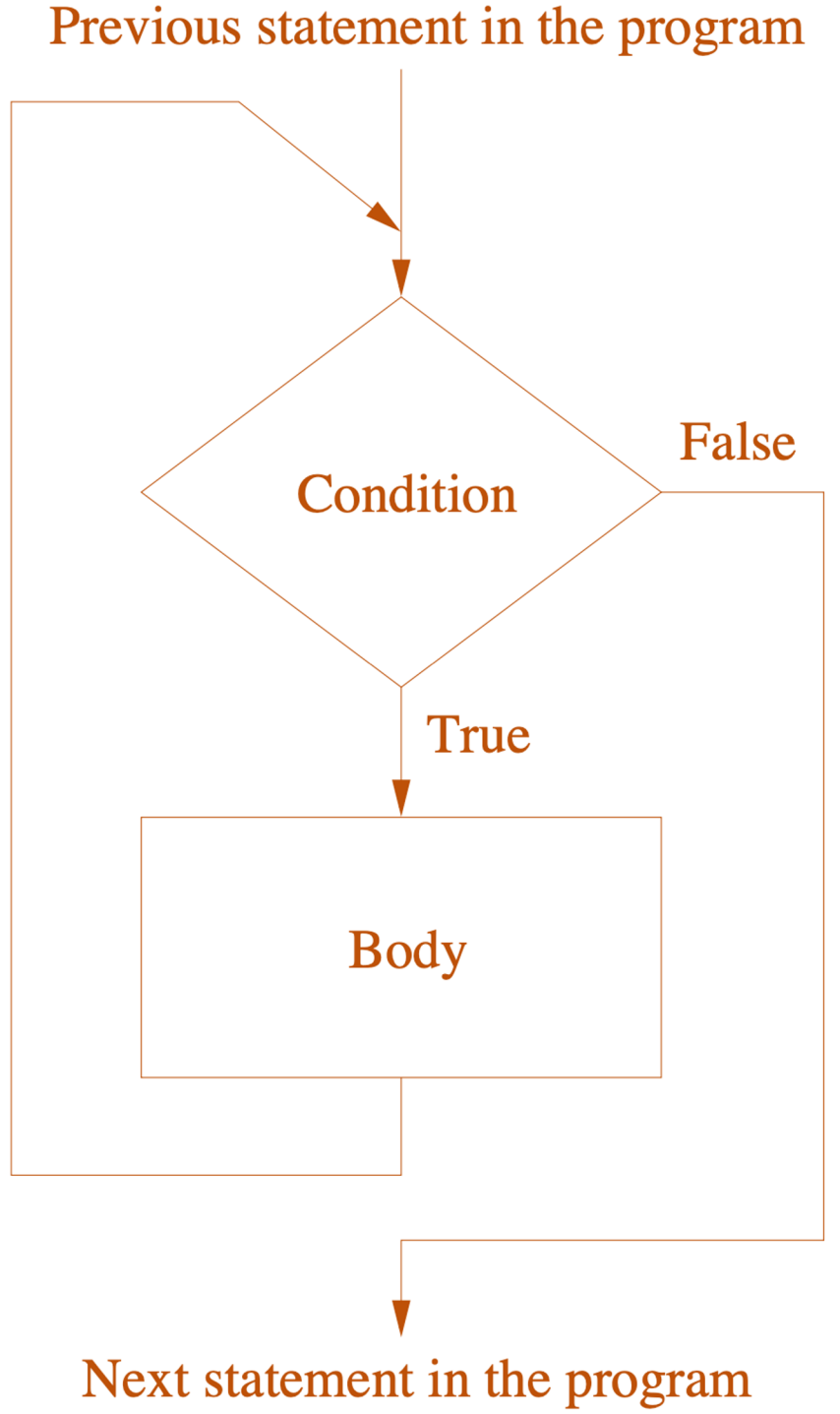
From the keyboard, read in a sequence of numbers, each denoting the marks obtained by students in a class. The marks are known to be in the range 0 to 100. The number of students is not told explicitly. If any negative number is entered, it is not to be considered the marks of any student, but merely a signal that all the marks have been entered. Upon reading a negative number, the program should print the average mark obtained by the students and stop.

- "Repeat" loop isn't great at solving this problem

```
repeat(count){
  statements
}
```

Loops

- Looping using “while”
  - while (condition) body
  1. Evaluate condition.
  2. If condition evaluates to true, then continue iterations, i.e., executing body once and return to step 1.
  3. If condition evaluates to false, then stop iterating.
- “body” needs to update some variable that has an effect on “condition”



Loops

- Any “repeat-count” loop can be replaced by an equivalent “while-condition” loop
  - unsigned int counter = 1;
    while (counter <= count)
    {
      statements;
      counter = counter + 1;
    }

```
repeat(count){
   statements
}
```

Loops

- While loop
  - Example: cubes of integers from 1 to 100

```
main_program{
   int i=1;
   while(i <= 100){
     cout << ‘‘The cube of ‘‘ << i << ‘‘ is ‘‘ << i*i*i << endl;
     i = i + 1;
   }
   cout << ‘‘Done!’’ << endl;
}
```

Loops

- While loop
  - Example: counting number of digits in an integer

```
main_program{
  int n;  cout << "Type a number: ";  cin >> n;

  int d = 1, ten_power_d=10;
          // ten_power_d will always be set to 10 raised to d

  while(n >= ten_power_d){  // if loop entered,
                            // number of digits in n must be > d
    d++;                    // so we try next choice for d
    ten_power_d *= 10;
  }

  cout << "The number has " << d << " digits." << endl;
}
```

Loops

- While loop
  - Example: averaging marks
    - Need to keep track of:
      (1) count of non-negative marks input so far
      (2) sum of non-negative marks input so far
    - Within each iteration, need to input marks and check if it is negative.
    - If input marks negative, then terminate loop; otherwise keep iterating
    - After exiting loop, compute average
    - Code assumes at least one non-negative mark entered (can modify to handle that case)

```
main_program{
  float nextmark, sum=0;
  int count=0;

  cin >> nextmark;

  while(nextmark >= 0){
    sum = sum + nextmark;
    count = count + 1;

    cin >> nextmark;
  }

  cout << "The average is: " << sum/count
}
```

Loops

- Example: averaging marks
- "break" inside body of a "while"
  - "condition" is always true
  - Must have a break inside body; otherwise we will have an "infinite loop"
    - Earlier version was easier to understand because terminating condition appeared at start of loop
    - Here, we need to search for a break statement inside body to figure out the logic
  - break gets control out of the (single) while within the body of which the break appeared
    - Be careful in case of nested while blocks

```
float nextmark, sum=0;
int count=0;

while(true){
   cin >> nextmark;
   if(nextmark < 0) break;
   sum = sum + nextmark;
   count = count + 1;
}
cout << sum/count << endl;
```

Loops

- What will this code do ?
  - int a = 0;
    while (a < 10)
    {
      cout << “a: “ << a << endl;
      if (a = 5) cout << “a equals 5” << endl;
      a++;
    }

a == 5

Loops

- What will this code do ?
  - float x = 0.1;
    while (x != 1.1)
    {
      cout << “x: ” << x << endl;
      x += 0.1;
    }
  - Infinite loop can be caused by rounding errors in floating-point arithmetic
    - Some decimal numbers cannot be represented exactly in finite-length binary

C output on an AMD Turion processor:

x = 0.10000000149011611938

x = 0.20000000298023223877

x = 0.30000001192092895508

x = 0.40000000596046447754

x = 0.50000000000000000000

x = 0.60000002384185791016

x = 0.70000004768371582031

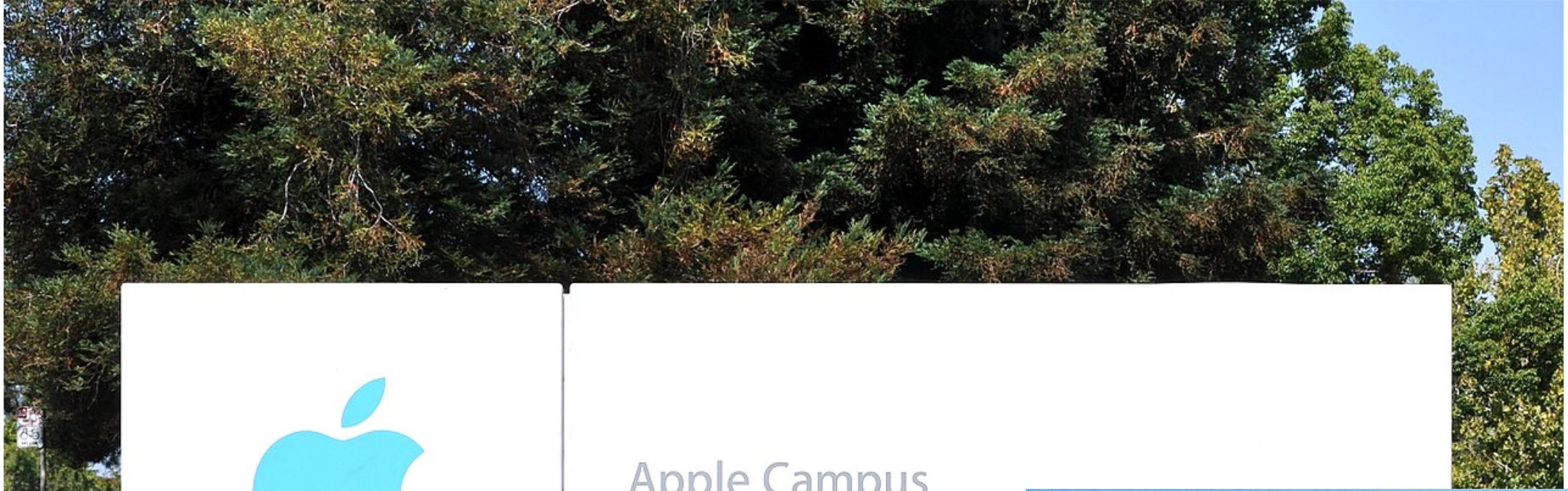
x = 0.80000007152557373047

x = 0.90000009536743164062

x = 1.00000011920928955078

x = 1.10000014305114746094

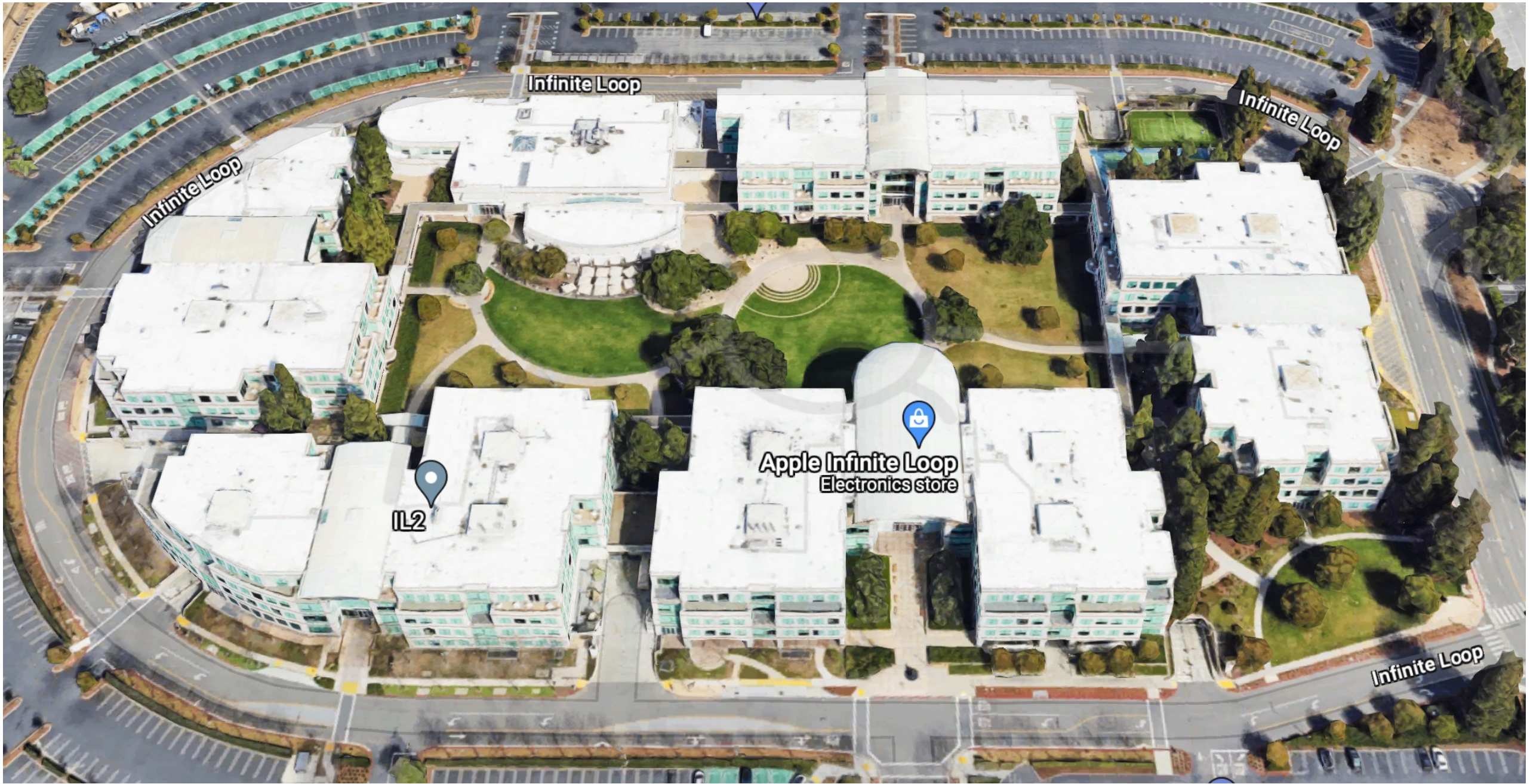
x = 1.20000016689300537109




Apple Campus
One Infinite Loop


INFINITE LOOP

Loops


Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
IL2
Apple Infinite Loop
Electronics store

Loops

- "continue" inside body of a "while"
  - When a "continue" executes inside while loop, control goes back to start of loop

```
while(true){
   cin >> nextmark;
   if(nextmark > 100){
      cout << "Larger than 100, ignoring." << endl;
      continue;
   }
   if(nextmark < 0) break;
   sum = sum + nextmark;
   count = count + 1;
}
```

Loops

- do-while loop
  - do body
    while (condition);
    - Control flow enters loop
    - Execute body
    - Evaluate condition
    - If condition true, then control goes back to start of loop
    - If condition false, then control comes out of the loop
  - When we don't want to check terminating condition upon first entry into loop

```
main_program{
  float x;
  char response;

do{
  cout << ''Type the number whose square root you want: '';
  cin >> x;
  cout << ''The square root is: '' << sqrt(x) << endl;

  cout << ''Type y to repeat: '';
  cin >> response;
  }
while(response == 'y');
}
```

Loops

- "for" loop
  - Motivation:
    When execution inside body of loop relies on value of a (typically, int) variable, whose values across iterations follow a certain sequence
  - Instead of this:

```
int i = 1;
repeat(100){
   cout << i << ‘ ‘ << i*i*i << endl;
   i = i + 1;
}
```

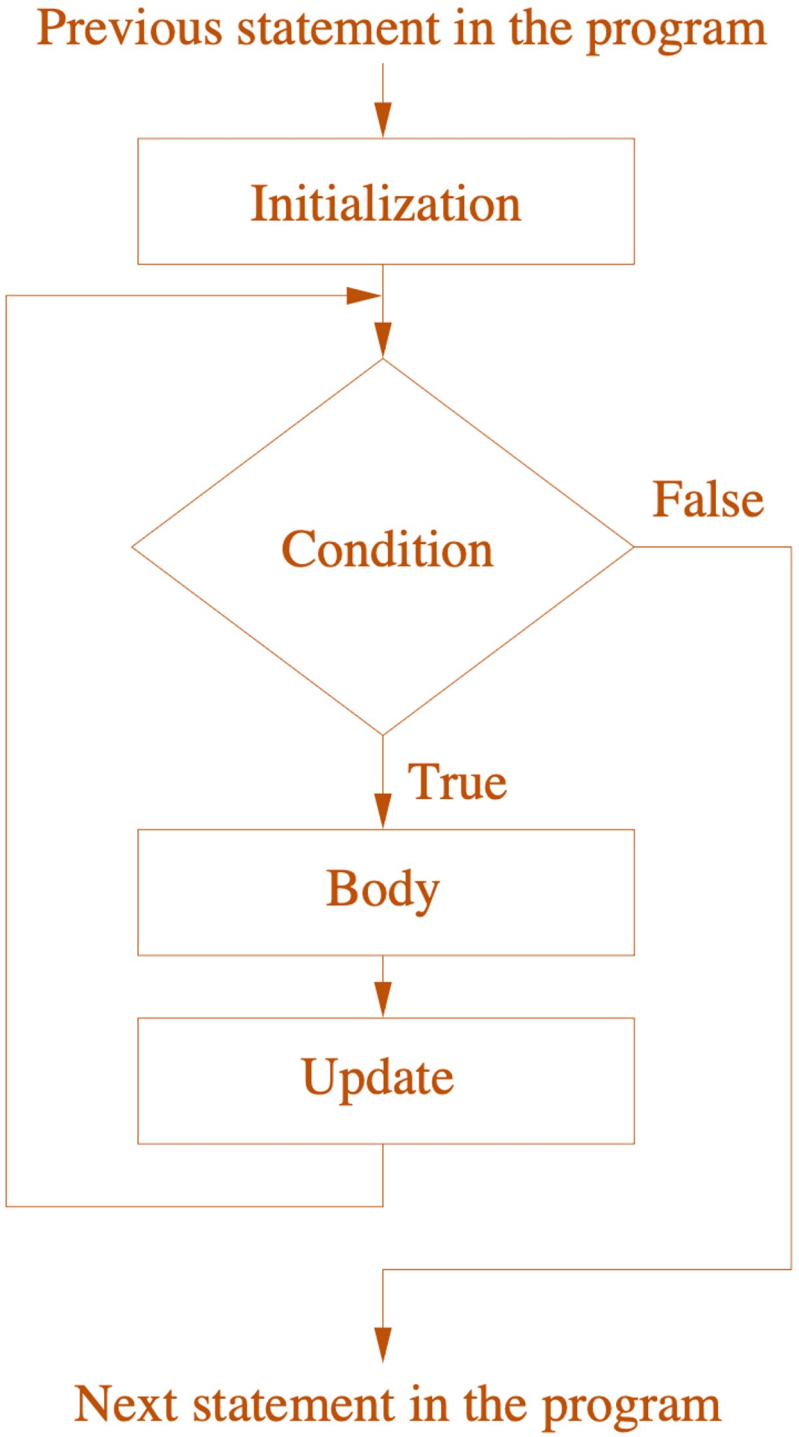
  - We do this:

```
for(int i=1; i <= 100; i = i + 1)
   cout << i << ‘ ‘ << i*i*i << endl;
```

  - Variable "i" called "control variable"
  - We iteratively execute body for different values of control variable i

Loops

- "for" loop
  - General form:
    for (initialization; condition; update) body
  - Execution:
    1. Execute initialization statement
    2. Evaluate condition.
    3. If false, then control goes out of loop
    4. If true then:
       - Execute body
       - Execute update statement
    5. Go back to Step 2
  - Special cases
    - Initialization statement can be empty
    - Update statement can be empty
    - Condition can be empty (then taken as true)



Loops

- “for” loop: scope of variables defined within “initialization”
  - Such variables:
    - Will get created within ‘for’ block
    - Will shadow other same-name variables defined earlier and in scope until ‘for’ block
    - Will get destroyed when control comes out of for block

```
int i=10;

for(int i=1; i<=100; i = i + 1) cout << i*i*i << endl;

cout << i << endl;
```

Loops

- “for” loop
  - for (initialization; condition; update) body
  - “break” statement inside body will terminate iterations; control comes out of for block
  - “continue” statement inside body will:
    - Immediately execute “update”
    - Return control to start of body

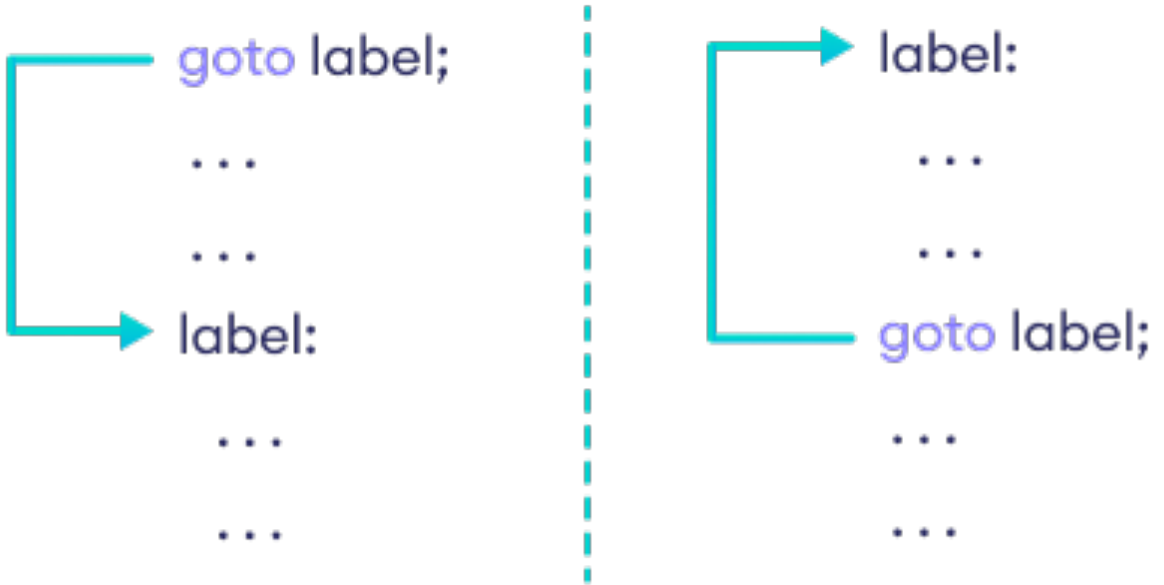
Loops

- “for” loop
  - Example of break statement inside for loop

```
main_program{
   int x; cin >> x;

   bool found = false;
   for(int i=2; i < x; i++){
                                  // x is not divisible by 2...i-1
     if(x % i == 0){ found = true; break;}
   }

   if(found) cout << "Composite.\n";
   else cout << "Prime.\n";
}
```

Loops

- “break” and “continue” aren’t great coding practices
- Like “goto” statement in C/C++, in disguise
  - In C, one could “label” any statement and then have a “goto label” command to jump control to that statement



- Abrupt jumps in control flow make code more difficult to understand, maintain, debug
- Avoid as much as possible in while/for loops
  - One justified use of “break” is within switch statement
- Java programming language doesn’t have a goto statement

Practice Examples for Lab: Set 6

- 1

Write a program that prints a conversion table from Centigrade to Fahrenheit, say between 0° C to 100° C. Write using `while` and also using `for`.

- 2

Suppose we are given n points in the plane: $(x_1, y_1), \ldots, (x_n, y_n)$. Suppose the points are the vertices of a polygon, and are given in the counterclockwise direction around the polygon. Write a program using a `while` loop to calculate the perimeter of the polygon. Also do this using a `for` loop.

- 3

Write a program that returns the approximate square root of a non-negative integer. For this exercise define the approximate square root to be the largest integer smaller than the exact square root. Your are expected to not use the built-in `sqrt` or `pow` commands, of course. Your program is expected to do something simple, e.g. check integers in order $1, 2, 3, \ldots$ to see if it qualifies to be an approximate square root.

Practice Examples for Lab: Set 6

- 4

Write a program that prints out the digits of a number starting with the least significant digit, going on to the most significant. Note that the least significant digit of a number `n` is simply `n % 10`.

- 5

Write a program that takes a number `n` and prints out a number `m` which has the same digits as `m`, but in reverse order.

- 6

A natural number is said to be a palindrome if the sequence its digits is the same whether read left to right or right to left. Write a program to determine if a given number is a palindrome.

Goto

- Spaghetti code
  - en.wikipedia.org/wiki/Spaghetti_code
  - “Code has a complex and tangled control structure, resulting in a program flow that is conceptually like a bowl of spaghetti, twisted and tangled”
  - “Can be caused by several factors, such as volatile project requirements, lack of programming style rules, and software engineers with insufficient ability or experience.”

```
#include <stdio.h>

int main()
{
    puts("This is Line 1");
    goto this;
that:
    puts("This is Line 3");
    goto theother;
backhere:
    puts("This is Line 5");
    goto end;
this:
    puts("This is Line 2");
    goto that;
theother:
    puts("This is Line 4");
    goto backhere;
end:

    return(0);
}
```

Goto

- The C Programming Language – Kernighan and Ritchie

3.8 Goto and Labels

C provides the infinitely-abusable `goto` statement, and labels to branch to. Formally, the `goto` is never necessary, and in practice it is almost always easy to write code without it. We have not used `goto` in this book.

Nevertheless, there are a few situations where `goto`s may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The `break` statement cannot be used directly since it only exits from the innermost loop. Thus:

```
for ( ... )
    for ( ... ) {
        ...
        if (disaster)
            goto error;
    }
...

error:
    clean up the mess
```

Goto

dl.acm.org/doi/10.1145/362929.362947

ARTICLE FREE ACCESS

Letters to the editor: go to statement considered harmful

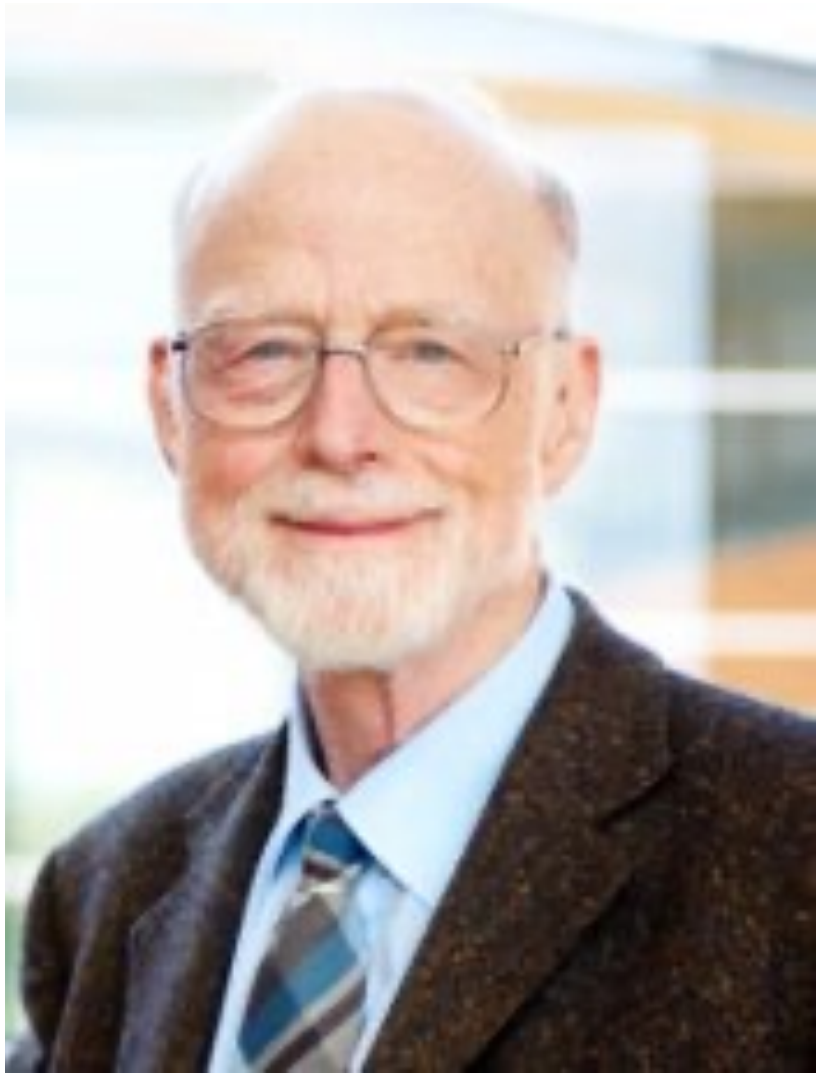
Author:  Edsger W. Dijkstra Authors Info & Claims

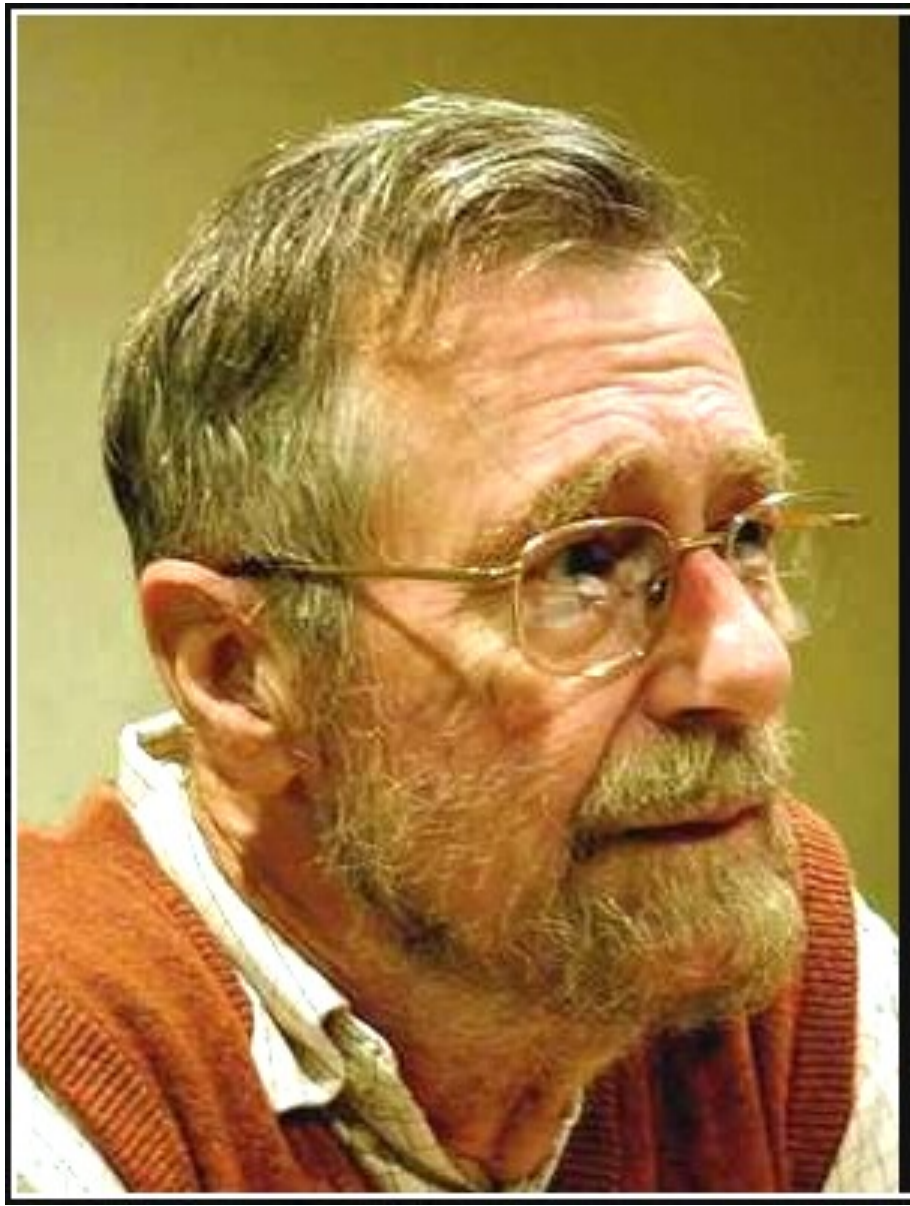
Communications of the ACM, Volume 11, Issue 3 • 01 March 1968 • pp 147–148 • https://doi.org/10.1145/362929.362947

Published: 01 March 1968 Publication History

Goto

- Edsger W. Dijkstra
- Dutch computer scientist, programmer, software engineer, systems scientist, and science essayist
- 1972 Turing Award
  for fundamental contributions to
  developing structured programming languages
- PhD thesis (1959) title:
  Communication with an Automatic Computer





Program testing can be used to show the presence of bugs, but never to show their absence!

— Edsger Dijkstra —

AZ QUOTES

Turing Award

- Annual prize given by Association for Computing Machinery (ACM) for contributions of lasting and major technical importance to computer science
  - en.wikipedia.org/wiki/Turing_Award
  - Highest distinction in computer science
  - "Nobel Prize of Computing"
- Named after Alan Turing
  - en.wikipedia.org/wiki/Alan_Turing
  - British mathematician, computer scientist, logician, cryptanalyst, philosopher, and theoretical biologist
  - Father of theoretical computer science and artificial intelligence

Association for Computing Machinery (ACM)

- US-based international learned society for computing
- en.wikipedia.org/wiki/Association_for_Computing_Machinery
- Founded in 1947
- World's largest scientific and educational computing society
- Several journals published by ACM
  - *ACM Transactions on Algorithms* (TALG)
  - *ACM Transactions on Embedded Computing Systems* (TECS)
  - *ACM Transactions on Computer Systems* (TOCS)
  - *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (TCBB)
  - *ACM Transactions on Computational Logic* (TOCL)
  - *ACM Transactions on Computer-Human Interaction* (TOCHI)
  - *ACM Transactions on Database Systems* (TODS)
  - *ACM Transactions on Graphics* (TOG)
  - *ACM Transactions on Mathematical Software* (TOMS)
  - *ACM Transactions on Multimedia Computing, Communications, and Applications* (TOMM)
  - *IEEE/ACM Transactions on Networking* (TON)
  - *ACM Transactions on Programming Languages and Systems* (TOPLAS)

Goto

- In 1974, at the peak of an international debate that followed Dijkstra's article, "Go To Statement Considered Harmful," Knuth wrote,

1. ELIMINATION OF go to STATEMENTS

Historical Background

At the IFIP Congress in 1971 I had the pleasure of meeting Dr. Eiichi Goto of Japan, who cheerfully complained that he was always being eliminated. Here is the history of the subject, as far as I have been able to trace it.

  - dl.acm.org/doi/pdf/10.1145/356635.356640

- Eiichi Goto
  - Japanese computer scientist, the builder of one of the first general-purpose computers in Japan

Goto

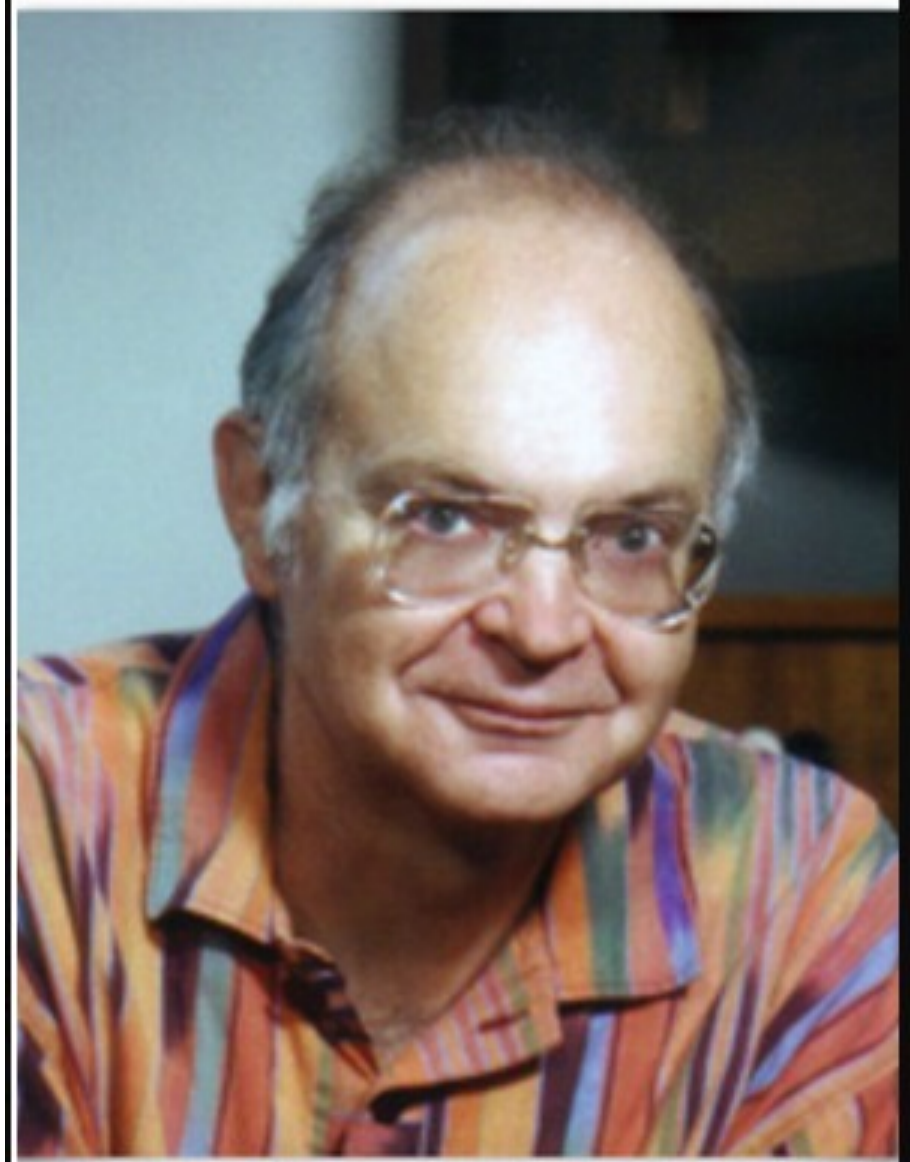
- Donald Knuth
  - American computer scientist, mathematician
  - 1974 recipient of the ACM Turing Award
  - “Father of the analysis of algorithms”
  - Author of “The Art of Computer Programming”
    - Opus in progress over 50 years

Completed [edit]

- Volume 1 – Fundamental Algorithms
  - Chapter 1 – Basic concepts
  - Chapter 2 – Information structures
- Volume 2 – Seminumerical Algorithms
  - Chapter 3 – Random numbers
  - Chapter 4 – Arithmetic
- Volume 3 – Sorting and Searching
  - Chapter 5 – Sorting
  - Chapter 6 – Searching
- Volume 4A – Combinatorial Algorithms
  - Chapter 7 – Combinatorial searching (part 1)
- Volume 4B – Combinatorial Algorithms
  - Chapter 7 – Combinatorial searching (part 2)

Planned [edit]

- Volume 4C... – Combinatorial Algorithms (chapters 7 & 8 released in several subvolumes)
  - Chapter 7 – Combinatorial searching (continued)
  - Chapter 8 – Recursion
- Volume 5 – Syntactic Algorithms
  - Chapter 9 – Lexical scanning (also includes string search and data compression)
  - Chapter 10 – Parsing techniques
- Volume 6 – The Theory of Context-Free Languages
- Volume 7 – Compiler Techniques

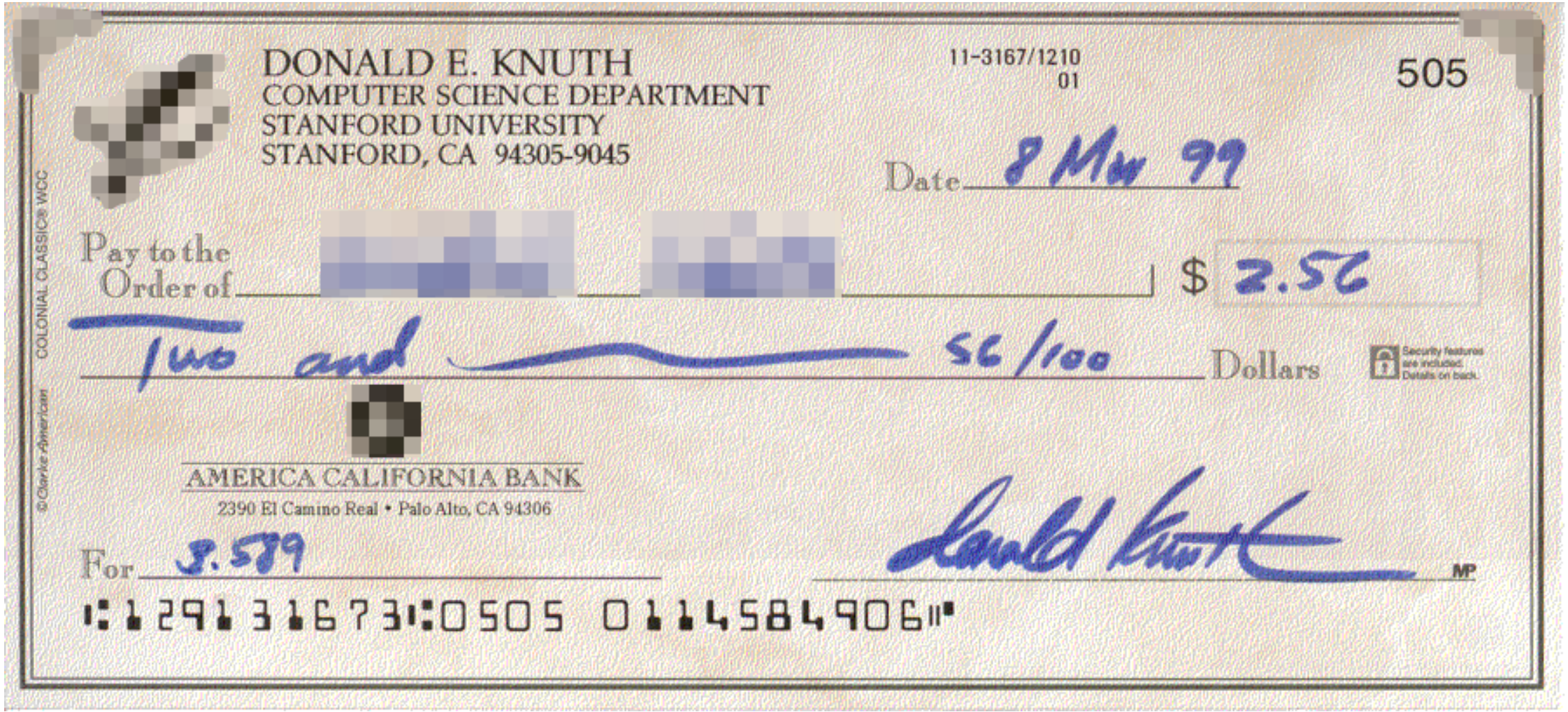


Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.

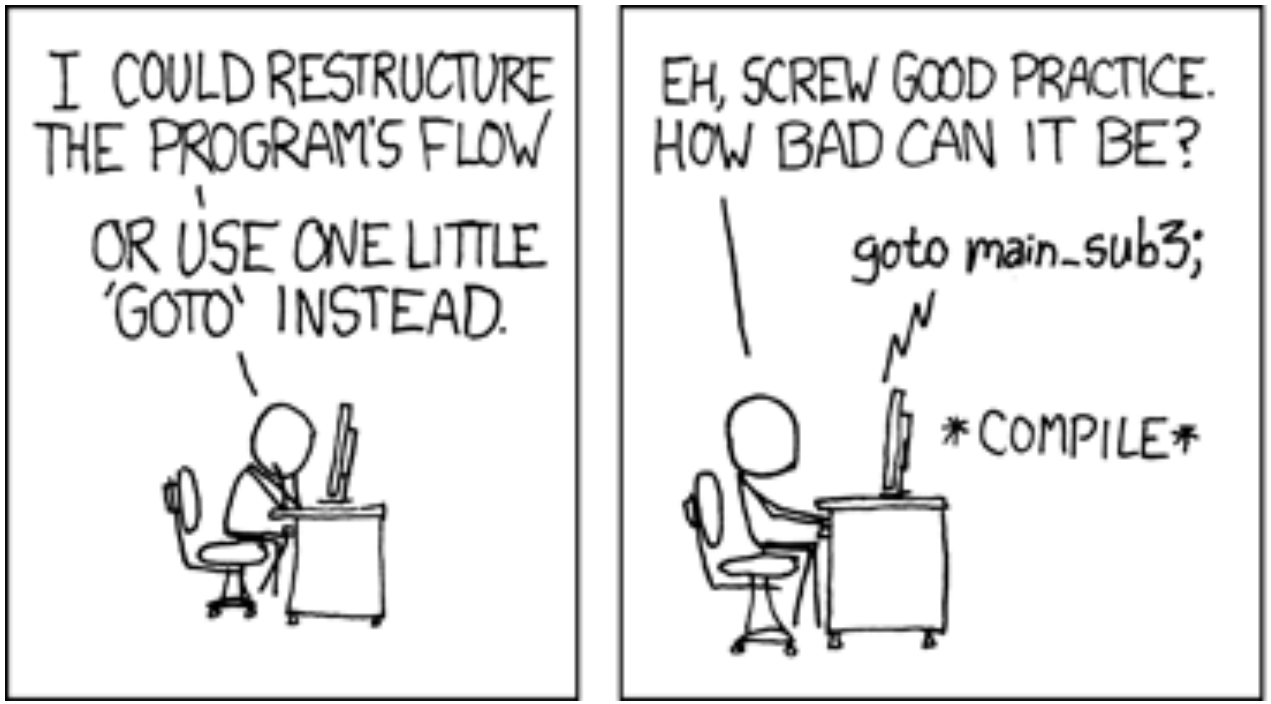
— Donald Knuth —

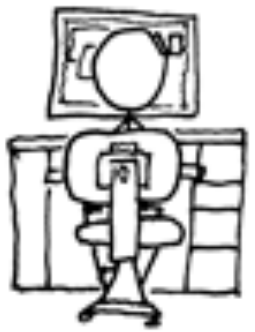
AZ QUOTES

- en.wikipedia.org/wiki/Knuth_reward_check
  - “The MIT Technology Review describes the checks as "among computerdom's most prized trophies”



Goto


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE



The pasta theory of programming

Spaghetti code

Unstructured and hard-to-maintain code caused by lack of style rules or volatile requirements. This architecture resembles a tangled pile of spaghetti in a bowl.

Lasagna code

Source code with overlapping layers, like the stacked design of lasagna. This code structure makes it difficult to change one layer without affecting others.

Ravioli code

Isolated bits of code that resemble ravioli. These are easy to understand individually but—taken as a group—add to the app's call stack and complexity.

Pizza code

A codebase with interconnected classes or functions with unclear roles or responsibilities. These choices result in a flat architecture, like toppings on a pizza.

Loops

- "for" loop (possibilities/tricks)
  - "Initialization" and "update" can have multiple assignment statements separated by commas
  - Example of digit-counting problem

```
main_program{
  int n; cin >> n;

  int d, ten_power_d;
  for(d=1, ten_power_d = 10; ten_power_d <= n; d++, ten_power_d *= 10);

  cout << "The number has " << d << " digits." << endl;
}
```

    - This for statement doesn't have a body
      - For this example, code is compact (and clever), but for loop without body isn't good coding style

Loops

- "for" loop (possibilities/tricks)
  - "Initialization" and "update" can have input statements via cin
  - Example of mark-averaging problem

```
main_program{
  float nextmark,sum=0;
  float count=0;

  for(cin >> nextmark; nextmark >= 0; cin >> nextmark){
    count++;
    sum += nextmark;
  }
  cout << sum/count;
}
```

  - Tricks usually come at the cost of readability

Loops

- Greatest common divisor (GCD)
  - Consider 2 positive integers: m,n with m > n
  - Many possible algorithms for computing GCD:
    - Check for all numbers between 1 and min(m,n); choose largest that divides both
    - Find prime factorizations; find common factors; find their product
    - Euclid's algorithm, based on the following principle:
      D divides m and n if and only if D divides m-n and n.
      Thus (when m > n) : GCD(m,n) = GCD(m-n,n)
      - Reduce problem to finding GCD of (m-n) and (n). Proof is straightforward.
    - Example
      - GCD (3977, 943)
      - = GCD (3977 - 943, 943) = GCD (3034, 943)
      - = GCD (3034 - 943, 943) = GCD (2091, 943)
      - = GCD (2091 - 943, 943) = GCD (1148, 943)
      - = GCD (1148 - 943, 943) = GCD (205, 943)
      - Observation: we subtracted 943 multiple times until result was <= 943

Loops

- Greatest common divisor (GCD)
  - Modified Euclid's principle:
    D divides m and n if and only if D divides m%n and n.
    Thus (when m % n > 0) : GCD(m,n) = GCD(m%n,n)
  - Example:

	GCD (3977, 943)		GCD(m,n)
= GCD (3977 % 943, 943)	= GCD (205, 943)	= GCD (943, 205)	GCD (m',n')
= GCD (943 % 205, 205)	= GCD (123, 205)	= GCD (205, 123)	GCD (m'',n'')
= GCD (205 % 123, 123)	= GCD (82, 123)	= GCD (123, 82)	
= GCD (123 % 82, 82)	= GCD (41, 82)	= GCD (82, 41)	
= GCD (82 % 41, 41)	→ but 82 % 41 is zero		

  - Observation: stopping criterion should be when modulo operation produces 0; then divisor is the answer

Theorem 1 (Euclid) *Suppose m, n are positive integers. If $m\%n = 0$, then $GCD(m, n) = n$. Otherwise $GCD(m, n) = GCD(m\%n, n)$.*

Loops

Theorem 1 (Euclid) *Suppose m, n are positive integers. If $m\%n = 0$, then $GCD(m, n) = n$. Otherwise $GCD(m, n) = GCD(m\%n, n)$.*

- Greatest common divisor (GCD)
  - Modified Euclid's algorithm

```
main_program{ // Compute GCD of m,n, where m > n >0.
  int m,n;
  cout << "Enter the larger number (must be > 0): ";  cin >> m;
  cout << "Enter the smaller number (must be > 0): "; cin >> n;

  while(m % n != 0){
    int Remainder = m % n;
    m = n;
    n = Remainder;
  }
  cout << "The GCD is: " << n << endl;
}
```

Loops

Theorem 1 (Euclid) *Suppose m, n are positive integers. If $m\%n = 0$, then $GCD(m, n) = n$. Otherwise $GCD(m, n) = GCD(m\%n, n)$.*

- Greatest common divisor (GCD)
  - Modified Euclid's algorithm: Analysis
  - Loop invariants
    - By Theorem 1, before and after each iteration, GCD remains unchanged
    - As per code, before and after each iteration, we have m > n > 0
  - Is termination guaranteed ?
    - At start of some iteration, if m % n = 0, then loop body isn't entered; we terminate.
    - If loop body entered
      - During each iteration, m % n always < n, and positive
      - Thus, after each iteration, max(m,n) has reduced (because m>n at start)
      - So, we will need at most max(m,n) iterations to reach termination
      - Actually it is much less:
      - Doing m % n gives m = Qn + R >= n + R (because Q >= 1; because m > n). Re-assign: m' = n and n' = R
      - [if enter body again] Doing m' % n' gives m' >= n' + R' > n'. So n > R. Re-assign: m'' = n' = R and …
      - Thus, m >= n + R = n + m'' = m' + m'' >= m'' + m'' = 2m'' (thus, in 2 iterations, m at least halved)

Computing Mathematical Functions

- Taylor series
  - In general, if x is reasonably close to x_0, then

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + f''(x_0)\frac{(x - x_0)^2}{2!} + f'''(x_0)\frac{(x - x_0)^3}{3!} + \cdots$$

    and computing first few (low-order) terms of series gives a good estimate
    - For some functions (e.g., sqrt, log, tan, arctan) it doesn't converge as x goes far from x_0
    - For some functions, (e.g., exp, sin, cos) it converges everywhere
    - en.wikipedia.org/wiki/Taylor_series#Analytic_functions
  - Another way of expressing Taylor series (define h := x - x_0)

$$f(x_0 + h) = f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2!} + f'''(x_0)\frac{h^3}{3!} + \cdots$$

  - Choosing $x_0 = 0$ gives McLaurin series

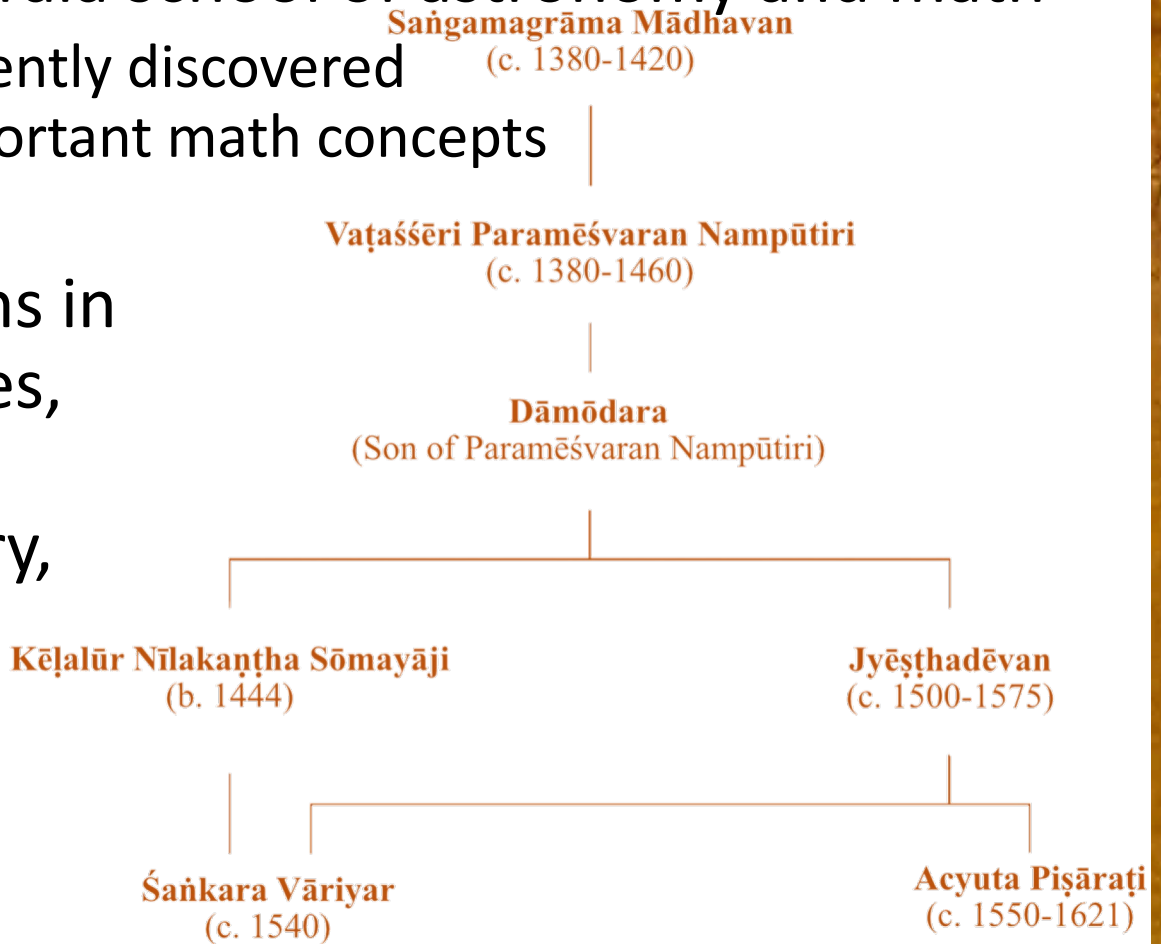
$$f(x) = f(0) + f'(0)x + f''(0)\frac{x^2}{2!} + f'''(0)\frac{x^3}{3!} + \cdots$$

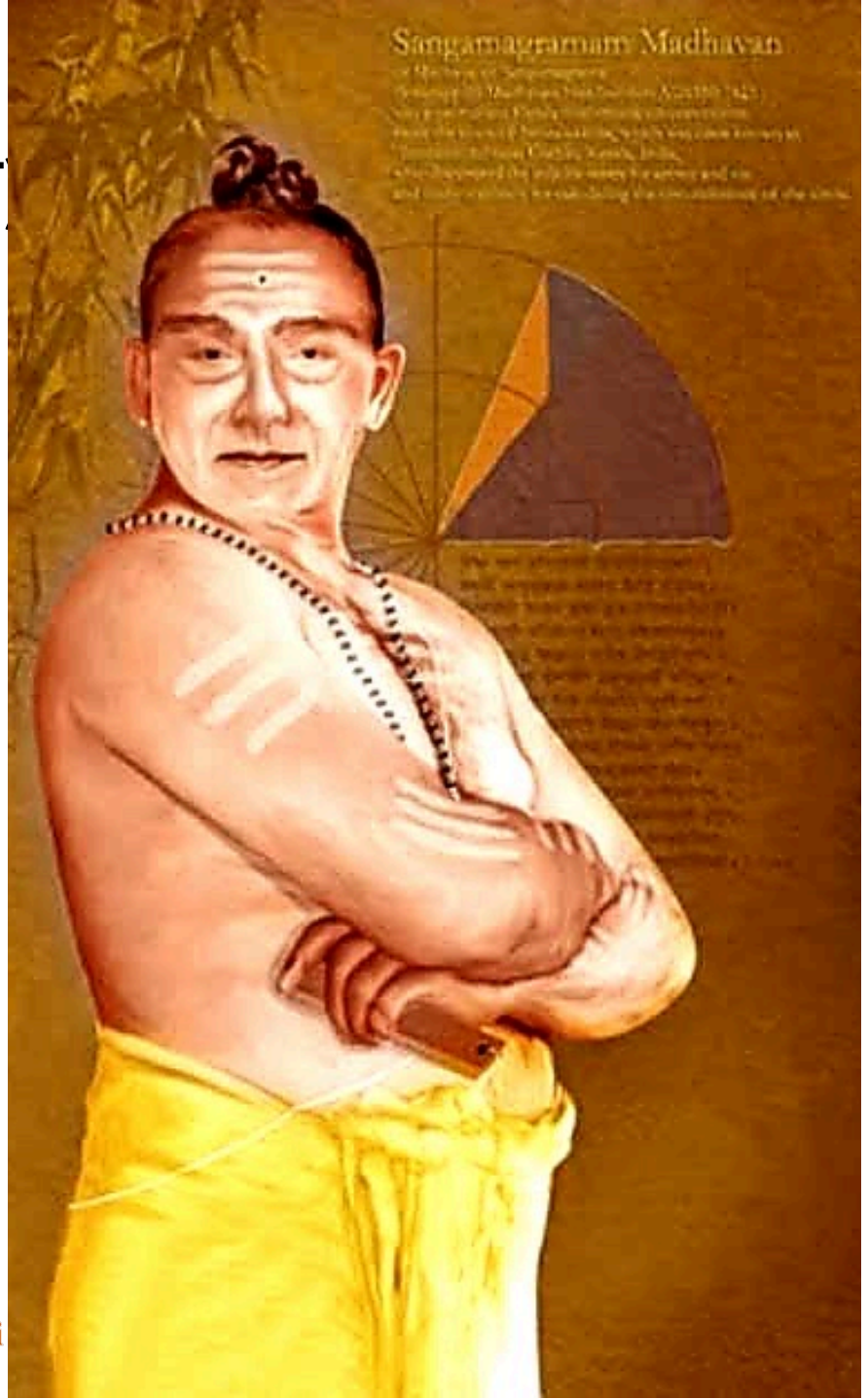
Computing Mathematical Functions

- Brook Taylor
  - British; first studied law at Cambridge Univ.: LL.B. and LL.D.
  - Proposed Taylor's theorem in 1715, which remained unrecognized until 1772, when Lagrange called it "main foundation of differential calculus"
  - On committee adjudicating some works of Newton & Leibniz
- Collin McLaurin
  - Scottish; child prodigy; entered U. Glasgow at age 11
  - Age 19 (year 1717): elected math professor at U Glasgow; was a world record for youngest prof, broken only in 2008
  - Joined U Edinburgh in 1725, where Newton, very impressed, offered to pay McLaurin's salary himself
  - McLaurin's series known before Taylor; some special cases relate to Madhava of Sangamagrama in 14th century India

Computing Mathematical Functions

- Mādhava (माधव) of Sangamagrāma (संगमग्राम)
  - One of greatest mathematician-astronomers of Middle Ages
  - Founded Kerala school of astronomy and math
    - Independently discovered many important math concepts
  - Pioneering contributions in infinite series, calculus, trigonometry, algebra, geometry, value of pi





Computing Mathematical Functions

- Choosing f(.) as sin(.) function (with angle in radians) and $x_0 = 0$ gives:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

  - To evaluate sin(x) for $x > 2\pi$, we evaluate sin(x % 2π) instead to keep x small

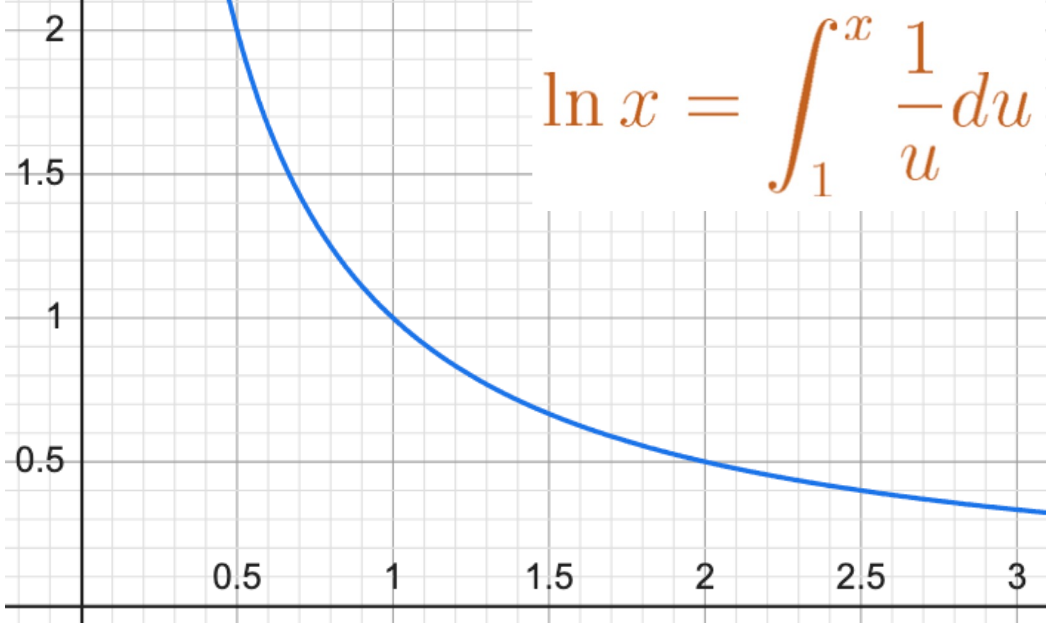
$$t_k = (-1)^{k+1}\frac{x^{2k-1}}{(2k-1)!} = t_{k-1}\left((-1)\frac{x^2}{(2k-2)(2k-1)}\right)$$

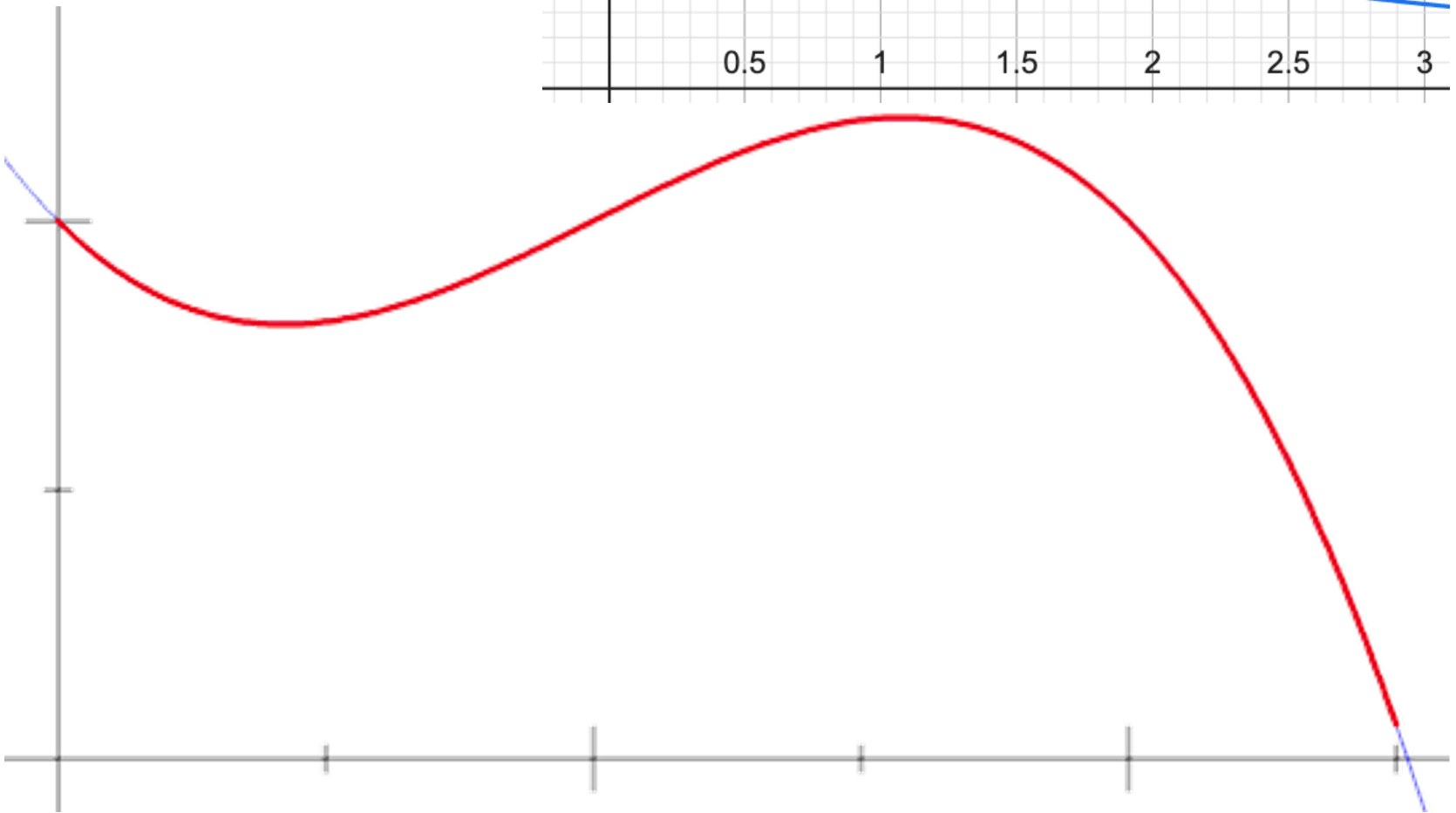
```
main_program{
  double x;  cin >> x;

  double epsilon = 1.0E-20, sum = x, term = x;

  for(int k=2; abs(term) > epsilon; k++){
              // Plan: term = t_{k-1}, sum = sum of k-1 terms
    term *= -x * x /((2*k-2)*(2*k-1));
    sum += term;
  }

  cout << sum << endl;
}
```

abs(term) = absolute value of term

Computing Mathematical Functio

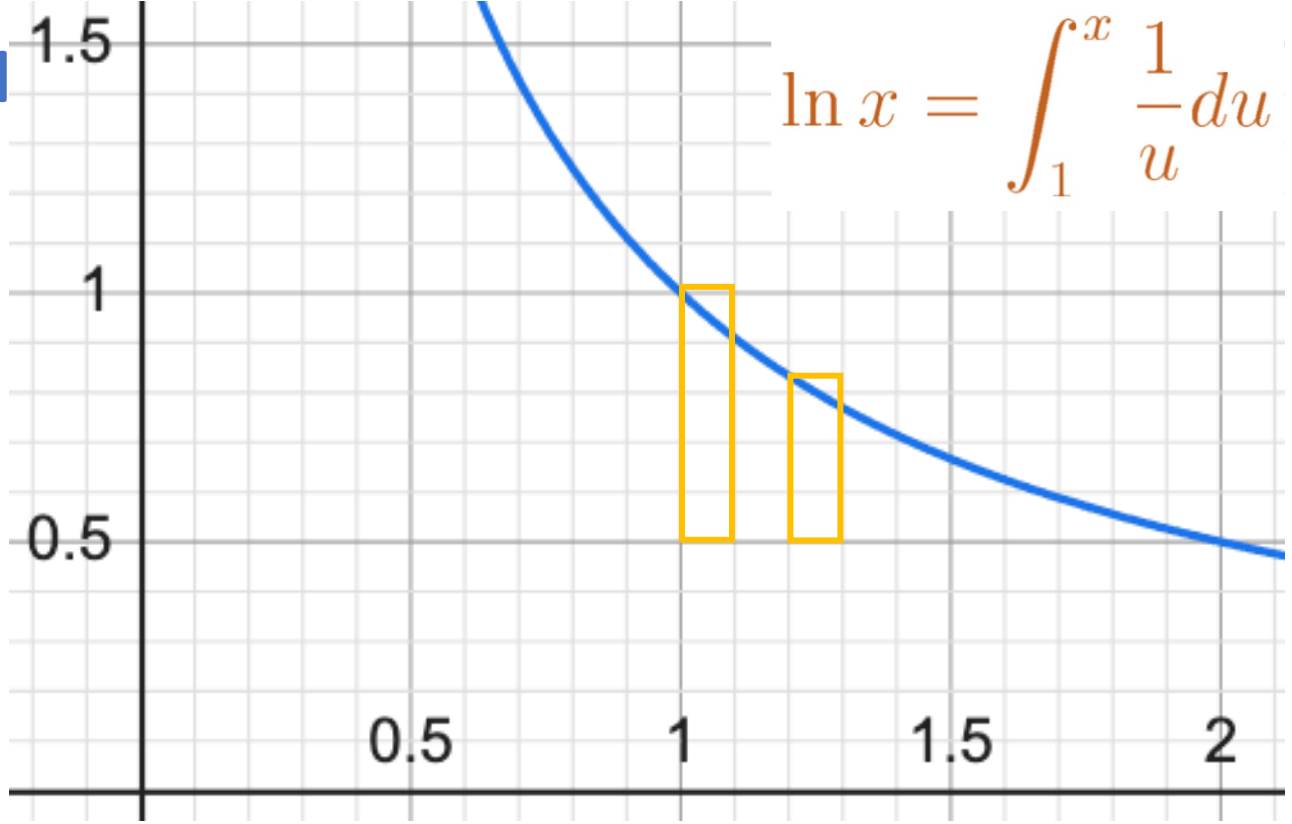
- Numerical Integration
  - ln(.) = area under part of 1/u curve between [1,x]
  - Riemann integration
  - Kinds of errors in numerical integration
    - Due to theoretical approximation
    - Due to approximation in floating-point calculation: 4-byte-floating-point representation is correct only up to 7 decimal places





Computing Mathematical Fu

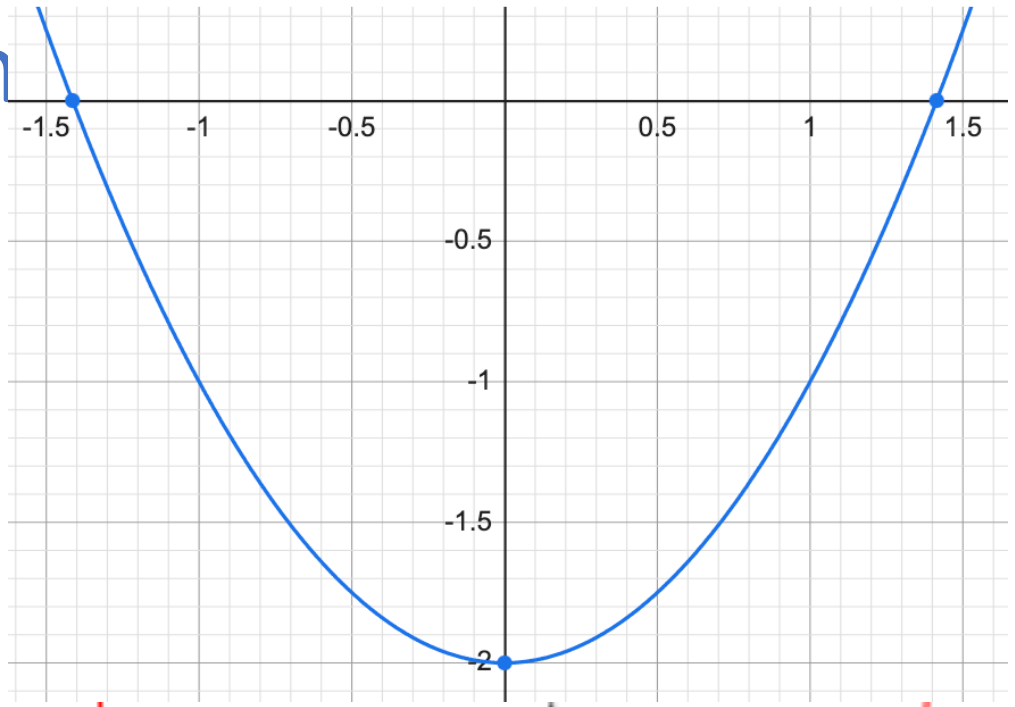
- Numerical Integration
  - Divide into n vertical strips
  - Strip width = w = (x-1)/n
  - i-th strip extends from u=1+iw to u=1+(i+1)w, where i takes values 0,1,…,n-1
  - i-th strip's height = 1/(1+iw); overestimate
  - Sum of areas of all strips = $\sum_{i=0}^{n-1} w \frac{1}{1+iw}$

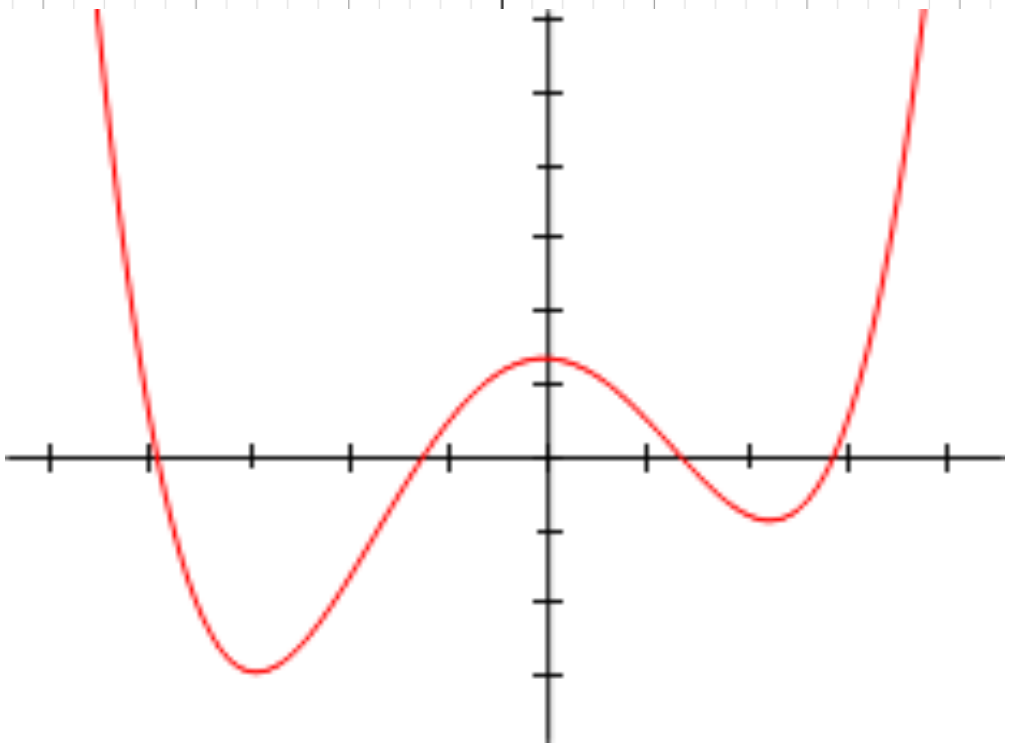



```
main_program{
   float x; cin >> x;          // will calculate ln(x)
   int n; cin >> n;            // number of rectangles to use
   float w = (x-1)/n;          // width of each rectangle
   float area = 0;             // will contain ln(x) at the end.
   for(int i=0; i < n; i++)
      area = area + w /(1+i*w);
   cout << "Natural log, from integral: "<< area << endl;
}
```

Computing Mathematical Function

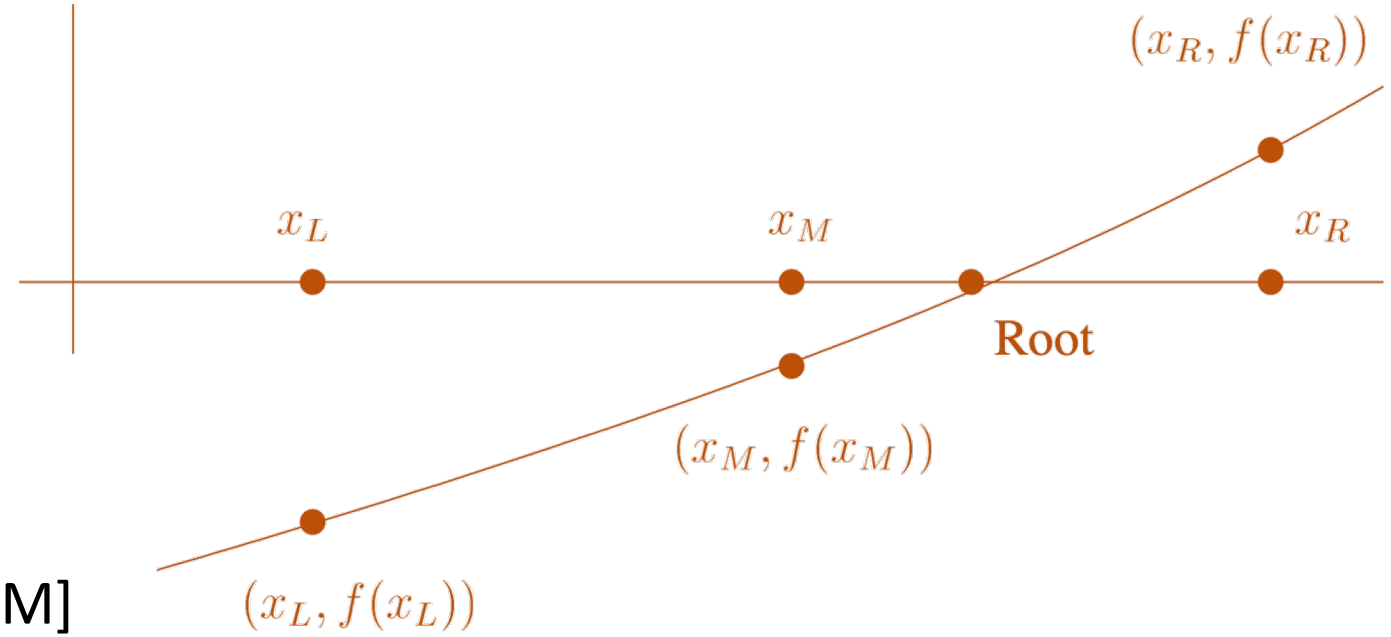
- Computing a root of a function
- For a function f(x),
  a root is defined as
  a value x' for which f(x')=0
- Example
  - Can use this to find roots of numbers, i.e., sqrt(2) equals root of function $f(x)=x^2-2$
  - Can use this to find
    local maximum/minimum of a function
    - Finding x that minimizes f(x) can be framed as first finding root of g(x) = df(x)/dx
      - Then checking for sign of $d^2f(x)/d^2x$





Computing Mathematical Functions

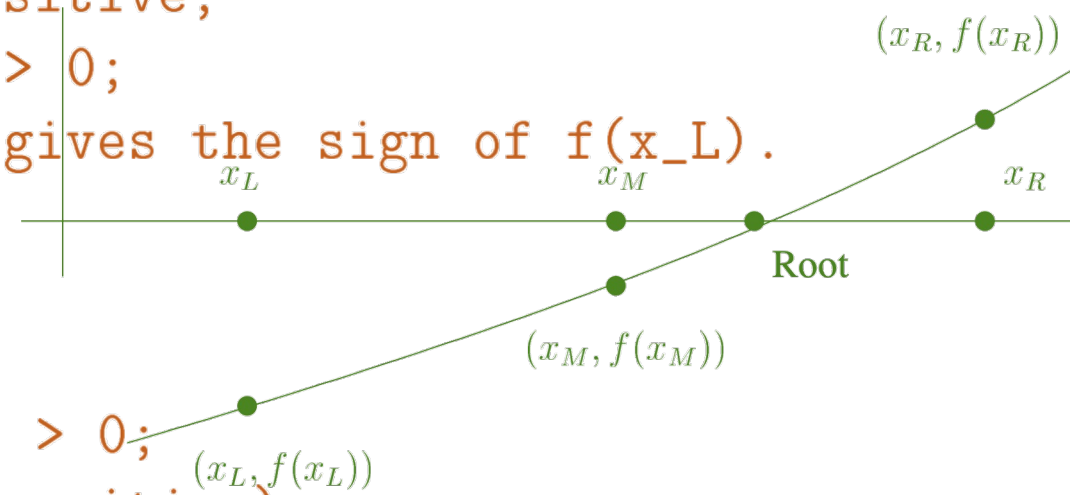
- Bisection method for computing function root
- Assumptions
  - Given: values xL & xR such that: xL < xR, and f(xL) & f(xR) have opposite signs
  - f(x) is continuous within interval [xL,xR]
- Algorithm
  - Consider interval's mid-point
    xM = (xL + xR) / 2
  - Evaluate sign of f(xM)
  - If sign(f(xL)) same as sign(f(xM)), then a root exists within [xM,xR]
    - Else, there a root exists within [xL,xM]
  - Redefine interval and repeat
  - When do we stop ? Should we check for f(xM) == 0 ?



Computing Mathematical Functions

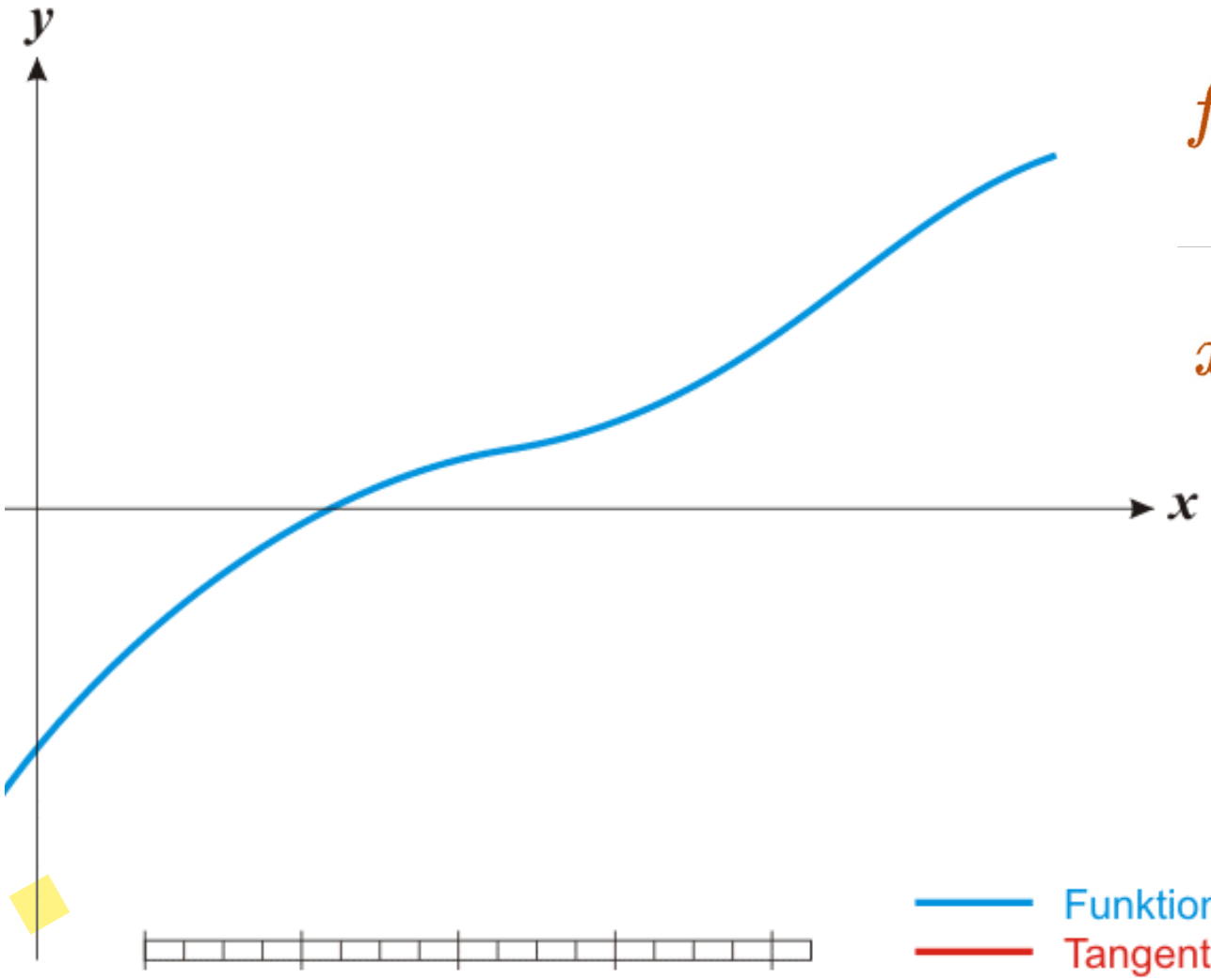
- Bisection method for computing function root
  - Is this code correct ?
- Analysis
  - Will loop terminate ?
  - Are we improving each iteration ?

```
main_program{                    // find root of f(x) = x*x - 2.
  float xL=0,xR=2; // invariant: f(xL),f(xR) have different signs
  float xM,epsilon;
  cin >> epsilon;
  bool xL_is_positive, xM_is_positive;
  xL_is_positive = (xL*xL - 2) > 0;
  // Invariant: xL_is_positive gives the sign of f(x_L).

  while(xR-xL >= epsilon){
    xM = (xL+xR)/2;
    xM_is_positive = (xM*xM -2) > 0;
    if(xL_is_positive == xM_is_positive)
      xL = xM;  // does not { xL = xM; xL_is_positive = xM_is_positive; }
    else
      xR = xM;  // does not upset any invariant!
  }
  cout << xL << endl;
}
```



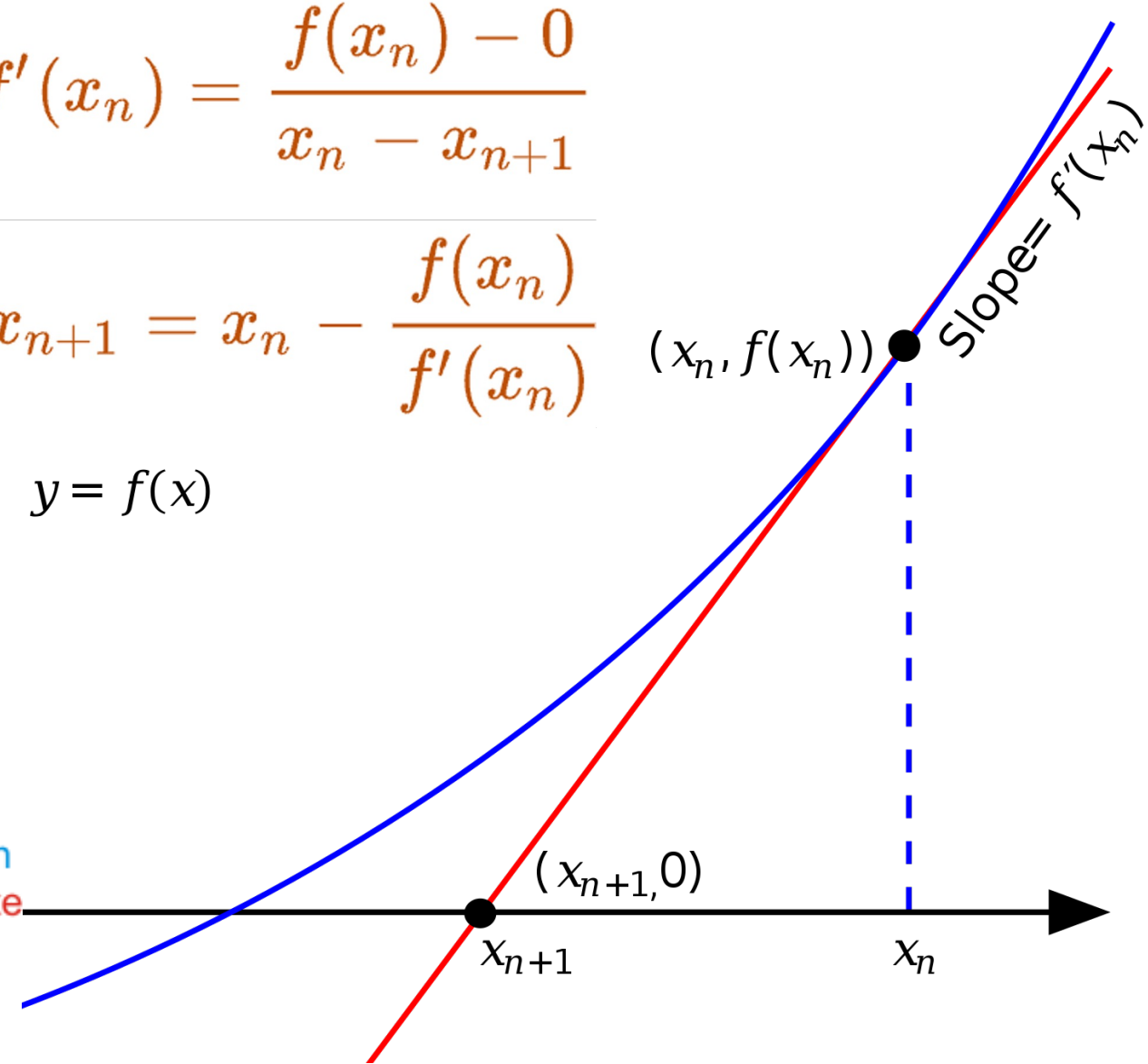
Computing Mathematical Functions

- Newton Raphson method for computing function root



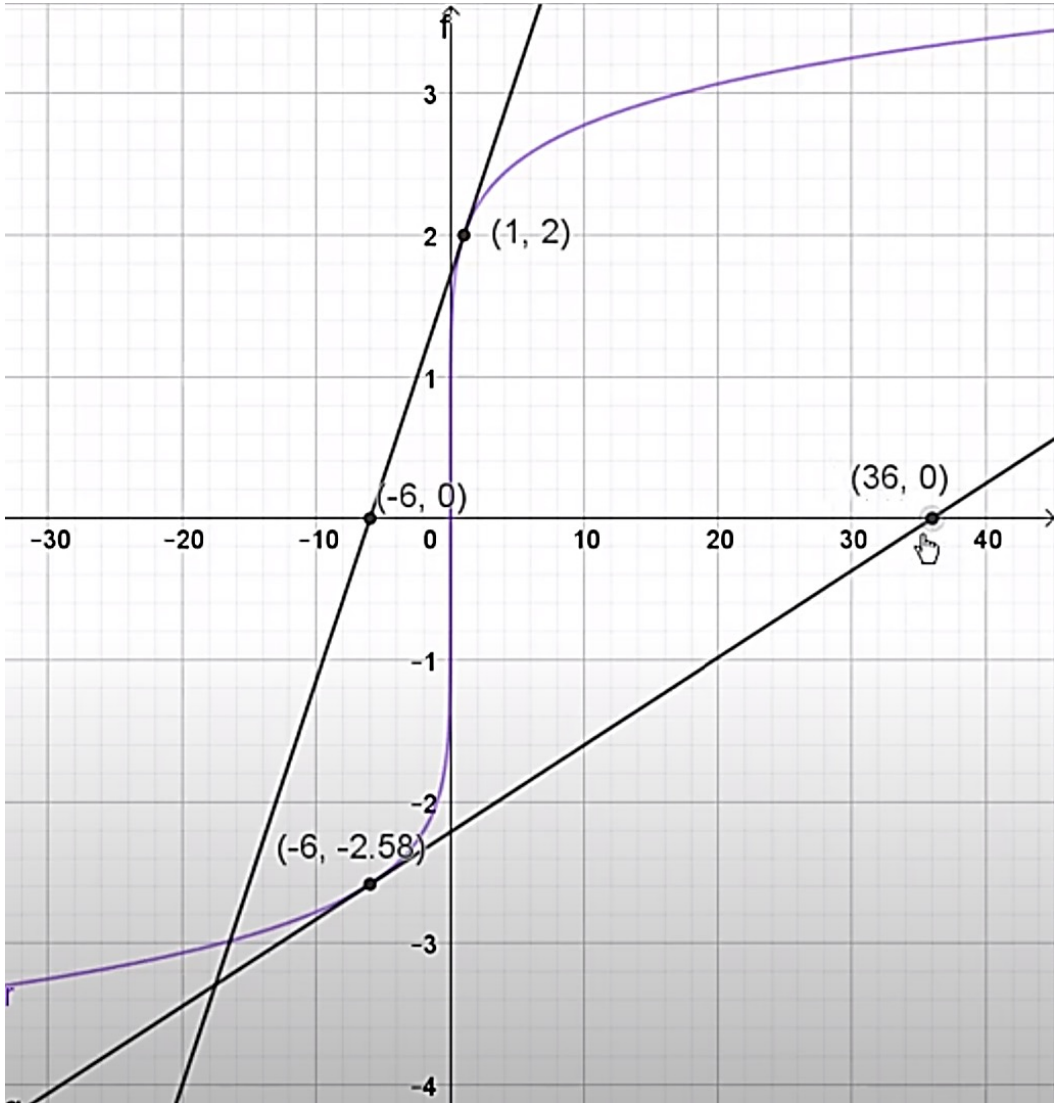
$$f'(x_n) = \frac{f(x_n) - 0}{x_n - x_{n+1}}$$

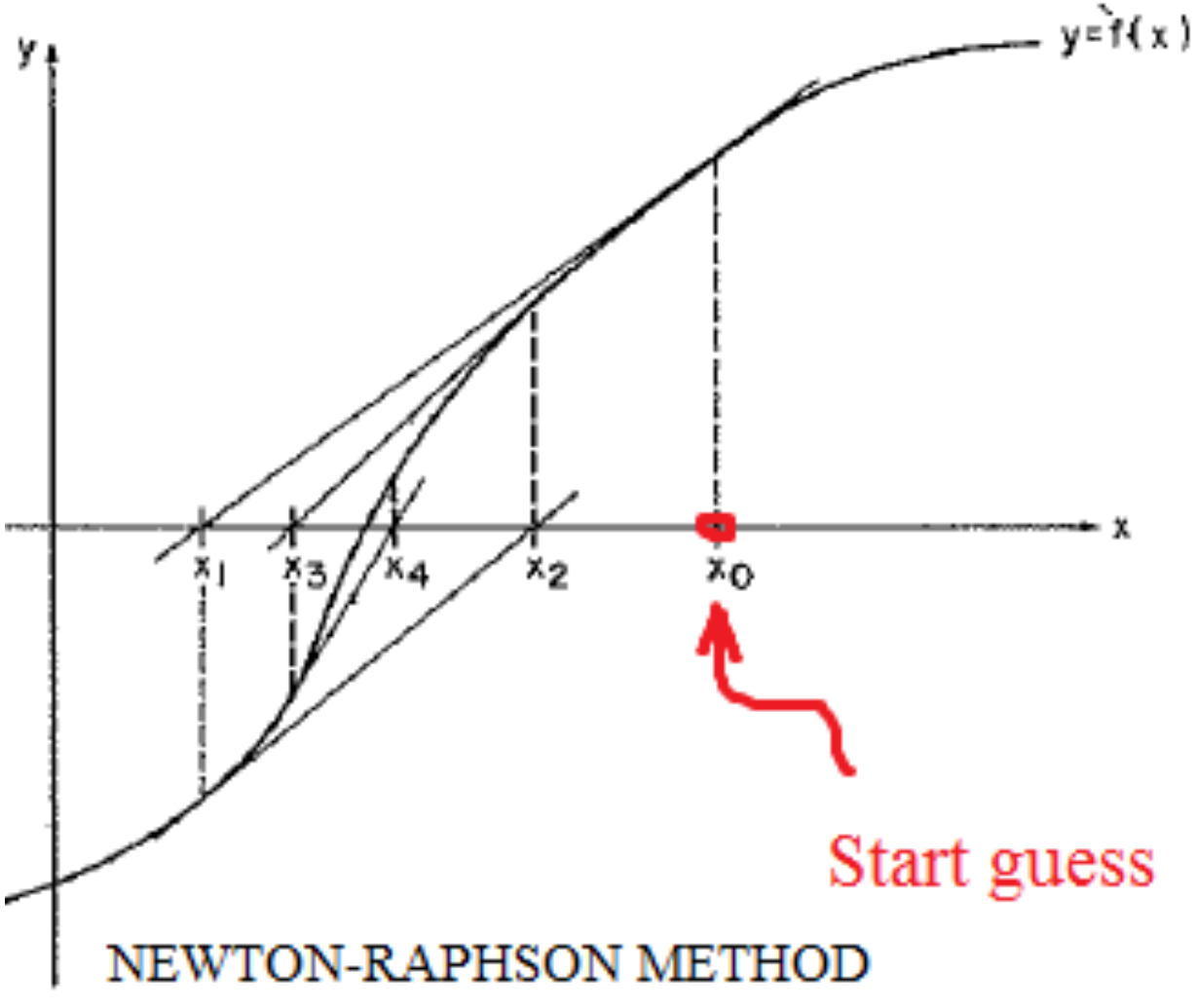
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



Computing Mathematical Functions

- Newton Raphson method for computing function root
  - Can fail to converge (depends on function and initialization), e.g., $f(x) = 2x^{(1/7)}$





Computing Mathematical Functions

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- Newton Raphson method for computing function root
  - Find root of $f(x) = x^2 - y$, where y is some given (positive) constant
  - Update step: $x_{i+1} = x_i - \frac{x_i^2 - y}{2x_i} = \frac{1}{2}(x_i + \frac{y}{x_i})$

```
main_program{
  double xi=1, y;  cin >> y;
  repeat(10){
    xi = (xi + y/xi)/2;
  }
  cout << xi << endl;
}
```

```
main_program{
  float y; cin >> y;
  float xi=1;
  while(abs(xi*xi - y) >0.001){
    xi = (xi + y/xi)/2;
    cout << xi << endl;
  }
}
```

Computing Mathematical Functions

- Secant method
  - Start with a pair of points x0 and x1
  - Equation of “secant” line is:

$$y = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0) + f(x_0)$$

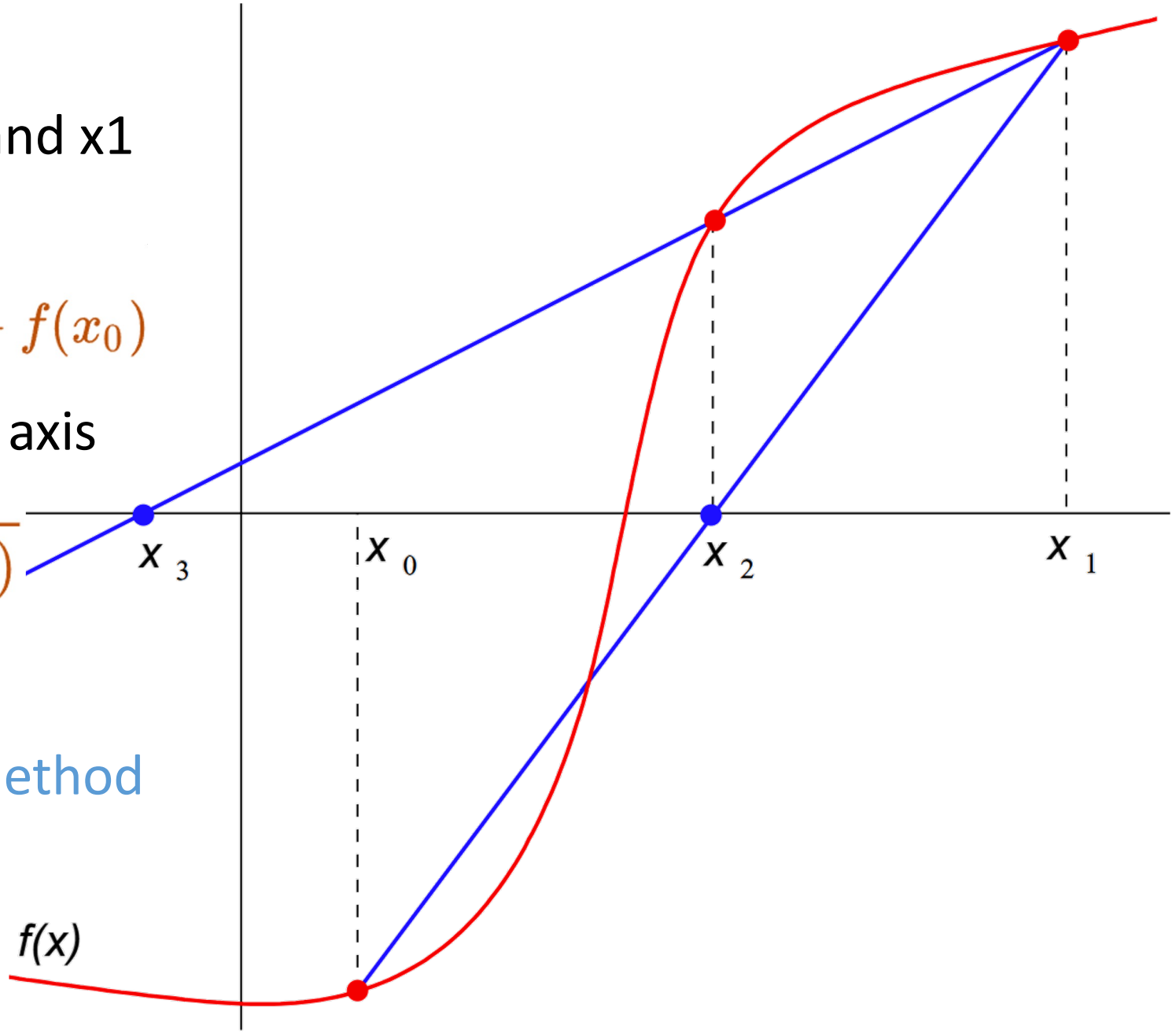
  - Find intersection of line withx axis

$$x = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

    - Call that point x2
  - Restart with pair x1 and x2
  - Similar to Newton-Raphson method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

  - Can fail to converge



Practice Examples for Lab: Set 7

- 1

A more accurate estimate of the area under the curve is to use trapeziums rather than rectangles. Thus the area under a curve $f(u)$ in the interval $[p, q]$ will be approximated by the area of the trapezium with corners $(p, 0)$, $(p, f(p))$, $(q, f(q))$, $(q, 0)$. This area is simply $(f(p) + f(q))(q - p)/2$. Use this to compute the natural logarithm.

- 2

Write a program to find $\arcsin(x)$ given x.

- 3

Write a program that takes as input a natural number x and returns the smallest palindrome larger than x.

- 4

Add checks to the GCD code to ensure that the numbers typed in by the user are positive. For each input value you should prompt the user until she gives a positive value.

Practice Examples for Lab: Set 7

- 5

Simpson's rule gives the following approximation of the area under the curve of a function f:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right)$$

Use this rule for each strip to get another way to find the natural log.

- 6

Children often play a guessing game as follows. One child, Kashinath, picks a number between 1 and 1000 which he does not disclose to another child, Ashalata. Ashalata asks questions of the form "Is you number between x and y?" where she can pick x, y as she wants. Ashalata's goal is to ask as few questions as possible and determine the number that Kashinath picked. Show that Ashalata can guess the number correctly using at most 10 questions. Hint: Use ideas from the bisection method.