## CSC 181F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC181F in the Fall term 2000/2001 at the University of Toronto





## CSC181F Introduction to Computer Programming

| | |
|---|---|
| **Instructor:** | Ray Ortigas |
| **Lectures:** | R11 in GB119, F12 in GB220, T12 in RS211 |
| **Tutorial:** | M12 in GB405, HA403 |
| **Practical:** | W4-6 in SF1106, SF1012 |
| **Office Hours:** | T2-4, F11 in SF3207 |
| **E-mail:** | rayo@dgp.toronto.edu |
| **News group:** | `ut.ecf.csc181` |
| **Web Page:** | `www.dgp.toronto.edu/people/rayo/csc181/` |



## CSC180F & CSC181F

| **CSC180F** | | **CSC181F** |
|---|---|---|
| no previous experience | | assumes previous experience |
| no previous programming | | assumes previous programming |
| covers introductory C | | covers more (advanced) C |
| | | covers C++ |
| basic programming skills | | covers advanced material |
| | | more emphasis on technique and style |
| | | teaches *effective programming* |
| FinalAverage( CSC180F ) | $<$ | FinalAverage( CSC181F ) |



## What's Important

- Writing and Communication Skills
- Writing and Communication Skills
- Mathematical Skills
- Computer Skills

A typical commercial software project involves creating more than 20 kinds of paper documents on such items as requirements and functional, logic, and data specifications. For civilian projects, at least 100 English words are produced for every source code statement in the software. For military software, about 400 words are produced for every source code statement. Many new software professionals are surprised when they spend more time producing words than code.[a]

[a] Capers Jones, *Gaps in programming education* , IEEE Computer Magazine, pp. 70–71, April 1995

## Books for CSC181F

- Required Text Books
  - K.N. King, C Programming: A Modern Approach, Norton, 1996
- Recommended Text Books
  - S.P Harbison and G.L. Steele Jr., C A Reference Manual, Prentice Hall, 4th edition, 1995
  - S. McConnell, Code Complete - A Practical Handbook of Software Construction, Microsoft Press, 1993
- Reference Texts
  - E. Roberts, The Art and Science of C, Addison-Wesley, 1995
  - E. Roberts, Programming Abstractions in C, Addison-Wesley, 1998
  - B.W. Kernighan & D.M. Ritchie, The C Programming Language (ANSI edition), Prentice-Hall, 1988
  - B.W. Kernighan and R. Pike, The Practice of Programming, Addison-Wesley, 1999



## Reading Assignment

```
        K.N. King,          Chapter 1

Supplemental reading

        S. McConnell        Chapters 1 to 3
```



## Grading Scheme

| | | |
|---|---|---|
| Assignment 1 | Due 26 Sep | 4% |
| Assignment 2 | Due 10 Oct | 4% |
| Term Test 1 | TBA (Week of 16 Oct) | 20% |
| Assignment 3 | Due 31 Oct | 4% |
| Assignment 4 | Due 14 Nov | 4% |
| Term Test 2 | TBA (Week of 20 Nov) | 20% |
| Assignment 5 | Due 5 Dec | 4% |
| Final Exam | TBA (December) | 40% |



### Assignments and Tests

- Assignments are due at the **START** of lecture on the due date.
- Assignments may be handed in early to the instructor.
- Late assignments will **NOT** be accepted except in case of *documented* illness or family problems. Similarly, midterms may not be missed except in case of *documented* illness or family problems. Consult the instructor if you need special consideration.

The test and exam are closed book, but a single page (8.5 by 11 inch, both sides) aid and non-programmable calculators will be allowed.

## Plagiarism and Cheating

Plagiarism is a kind of fraud: passing off someone else's work or ideas as your own in order to get a higher mark.

Plagiarism is a serious offense at U of T. It will **not** be ignored.

We have programs to compare students' programs for evidence of similar code. We shall ask you to submit electronic versions of all of your assignments, and we shall run our programs on these. Due to the way programs work, it does not help to change comments, variable names or even code organization.

**You can really screw up your career at U of T AND YOUR FUTURE by committing an act of plagiarism.**

**The assignments you hand in must be your own and must not contain anyone else's ideas.**

Refer to Appendix A in the U of T Code of Behavior on Academic Matters for a more detailed description of plagiarism.



## Guidelines for Avoiding Plagiarism

You may discuss assignments with friends and classmates, but only up to a point: You may discuss and compare general approaches and also how to get around particular difficulties, but you should not leave such a discussion with any written material. You should not look at another student's solution to an assignment on paper or on the computer screen, even in draft form. The actual coding of your programs, analysis of results, and writing of reports must be done individually.

If you do talk with anyone about an assignment, please state this in your assignment and state the extent of your discussion.

Note that it is also a serious offense to help someone commit plagiarism. Do not lend your printouts, reports or diskettes, and do not let others copy or read them. To protect yourself against people copying your work without your knowledge, retain all of your old printouts and draft notes until the assignments have been graded and returned to you. If you suspect that someone has stolen a printout or diskette, contact your instructor immediately.



## Helping Each Other

Although you must not solve your assignments with the help of others, there are still many ways in which students can help each other.

For instance, you can go over difficult lecture or tutorial material, work through exercises, or help each other understand an assignment handout.

You can ask the tutors to explain material that you are having difficulty with.

This sort of course collaboration can be done in study groups or through the newsgroup.



## Program Construction

- Understand the Problem!!!!
- Design Algorithms and Data Structures
- Design to Program
- Write the program
- Inspect the program for errors
- Compile and Debug the Program
- Test the program thoroughly
- Document the program

## Important Considerations

- Correctness
- Correctness
- Correctness
- Program maintainability and modifiability
- Program's efficiency
- Programmer's efficiency



## Knowing a Programming Language

- Syntax
  The *form* of legal constructs
- Semantics
  The *meaning* of legal constructs
- Technique
  How to use the language effectively and efficiently



## Program Development

- Stepwise refinement
- Choose data structures
- Choose algorithms
- **Think before programming**
- **KISS** - **K**eep **I**t **S**imple **S**tupid
- Solve most general instance of problem, minimize special cases



## Program Development

- Requirements
- Specification
- Design
- Implementation
- Debugging & Testing

## Requirements & Specification

- Understand the problem
  - General case
  - Special cases
    - Boundary conditions
  - Errors & Exceptions
- Requirements
  - What the user needs
  - Often described informally
- Specification
  - Formal & Precise description of Problem
  - Describe problem not the solution



## Program Design

- The Designer's Palette
  - Data Structures
  - Algorithms
  - Programming Language
- Design Goals
  - Always correctness
  - Time or space efficiency
  - Development time
  - Maintainability
- Designer's Resources
  - Experience
  - Books and articles on algorithms and design



## Design Techniques

- Top Down
  - Stepwise refinement
- Bottom Up
- Yo-yo, Chaotic
  - **BAD -** leads to rotten programs

*Program design should be systematic and methodical!*



## Problem Analysis

- What are the inputs?
  - domain & range? special cases?
- What are the outputs?
  - domain & range? special cases?
- How are outputs related to inputs?
- What is the general case of the problem?
- Have I solved this problem or a similar problem before?
- Has someone solved this problem or a similar problem before?

## Design Technique

- Data structures then algorithms
- Algorithms then data structures
- Try to find efficient method for the most general case
  Minimize special cases and exceptions
- Iterate on design
  more efficient or more general
- KISS - Keep It Simple Stupid
  Simplicity is **the** virtue



## Problem Analysis[a]

- Analogy, Conditions
- Decomposition & Recombination
- Use all the problem description?
- Solve subproblem & generalize?
- Induction, work backwards?
- Identify intermediate results
- Check the results?
- Iterate on simplifying solution

[a] G. Polya, *How to Solve It*, Princeton University Press



## Design by Stepwise Refinement

- Start with the whole problem
- Subdivide problem into several separate subparts
  e.g. input, compute, output
- Decide how to represent data and carry it between subparts
- Subdivide each subpart into simpler subsubparts
- Continue subdividing until sub*parts are small
- Combine sub*parts to make complete program



## Stepwise Refinement Example

Problem: Solve quadratic equations

$$ax^2 + bx + c = 0.0$$

Analysis

general case 2 real or imaginary roots
special cases a, b and/or c = 0.0

Applicable algorithm

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

## Quadratic Equation Case Analysis

$$ax^2 + bx + c = 0.0$$

| | b = 0.0<br>c = 0.0 | b ≠ 0.0<br>c = 0.0 | b ≠ 0.0<br>c ≠ 0.0 | b = 0.0<br>c ≠ 0.0 |
|---|---|---|---|---|
| a = 0.0 | ∞ roots<br>$0.0 = 0.0$ | one real root<br>$bx = 0.0$ | one real root<br>$bx + c = 0$ | no roots<br>c = 0.0 |
| a ≠ 0.0 | two real roots<br>$ax^2 = 0.0$ | two real roots<br>$ax^2 + bx = 0.0$ | two roots<br>$ax^2 + bx + c = 0.0$ | two roots<br>$ax^2 + c = 0.0$ |

Applicable algorithm is only valid when a ≠ 0.0
Applicable algorithm is overkill if c = 0.0
Roots are imaginary if $b^2 < 4ac$



## First Refinement

$Input_1$: read quadratic coefficients
$Analysis_1$: identify type of roots
$Compute_1$: calculate roots
$Output_1$: print coefficients and roots



## Second Refinement

$Input_{1.1}$: read a
$Input_{1.2}$: read b
$Input_{1.3}$: read c

‘ $Analysis_{1.1}$: determine if equation has valid roots
$Analysis_{1.2}$: deterimine if special case or not
$Analysis_{1.3}$: determine if non-special case roots are real or imaginary.

$Compute_{1.1}$: compute roots for special cases
$Compute_{1.2}$: compute roots for non-special cases

$Output_{1.1}$: Print coefficients
$Output_{1.2}$: Print roots



## Third Refinement

$Analysis_{1.1.1}$: no valid roots if a = 0.0 and b = 0.0
$Analysis_{1.2.1}$: special case if a = 0.0 and b ≠ 0.0
$Analysis_{1.2.2}$: non-special case if a ≠ 0.0
$Analysis_{1.3.1}$: if non-special case, compute $disc = b^2 - 4ac$
$Analysis_{1.3.2}$: roots are real if $disc \geq 0.0$
$Analysis_{1.3.3}$: otherwise roots are imaginary

$Compute_{1.2.1}$: compute roots if special case
$Compute_{1.2.2}$: compute non-special case real roots
$Compute_{1.2.3}$: compute non-special case imaginary roots

$Output_{1.1.1}$: Print a, b, c
$Output_{1.2.1}$: if no valid roots, print error message
$Output_{1.2.2}$: if special case print roots
$Output_{1.2.3}$: if non-special case real roots, print roots
$Output_{1.2.4}$: if non-special case imaginary roots, print roots

## Reading Assignment

```
K.N. King          Chapter 2

K.N. King          Chapter 3

K.N. King          Sections 7.1 to 7.3, 7.6

Supplementary reading

S. McConnell       Chapter 19
```



## C Programming Language

- *Very* widely used general purpose programming language
- Available on many machines and operating systems
- Designed to be flexible, powerful, and unconstraining
- Originally a replacement for assembly language
- **C requires extreme care in programming**
- **C requires extreme care in programming**
- *Traditional* C and *ANSI* C
- C++ is a superset of C with Object Oriented features



## BE REALLY REALLY CAREFUL IN C

- C provides **no** run-time checking
  e.g. array subscripts, undefined variables
- Programmer must manage dynamic storage allocation
- Pointers are widely used but are **unchecked**
- **Program with extreme care**
- There are good software tools for developing C programs
  debuggers, program checking aids
  large libraries of existing software



## Good Style: , Good Technique: and WARNING:

- **Good Style:** indicates a preferred way of programming in C. Programs with **Good Style:** are easier to read, understand, modify and get correct.
  Markers just love programs that exhibit **Good Style:** .
- **Good Technique:** indicates a good way to do some particular programming task in C. The technique is good because its efficient, easy to understand, easy to get correct.
  An entire slide of **Good Technique:** usually has **HOW TO** in the title.
- **WARNING:** is used to indicate a particularly tricky or dangerous part of C. Good programmers avoid **WARNING:** constructs or are *extremely careful* about how they use them.

## Comments

- Comments start with the characters `/*`
  Comments end with the characters `*/`
  *Any* arbitrary text can be included in a comment.
  Comments can be placed anywhere that a blank is legal.
- **Good Style:** comments should add to the readers understanding of the program by providing information that is *not* available just by reading the program. **Just repeating the program in English is dumb and useless**.
- **Good Style:** use *lots* of comments to make program easy to read and easy to understand.



- **WARNING:** An unterminated comment in C will **silently EAT part of your program**.
  This will almost certainly lead to **bugs** in your program.
  **Example:**

```
/* The programmer forgot to end this one line comment
    root1 = x * y -z ;
    root2 = root1 / ( a - c ) ;
/* the original comment REALLY ends here → */
```

- **Good Style:** Do **not** use comments to delete code from a program.
  Use the #if and #endif constructs as shown below:

```
#if 0
    This code is ignored
#endif
```

- **Good Style:** Use a comment style that leaves *no doubt* as to where the comment starts and ends.



## Block Comment Styles

- Simple, unadorned

```
/* first line
    following lines
*/
```

- Head and Trail Markers

```
/********************************************************
    first line of comment
    many more lines of comment
********************************************************/
```

- Full Block

```
/****************************************************************/
/* Comment that extends over several lines to explain */
/* some really vital concept about the program.        */
/****************************************************************/
```



## HOW TO Comment

- Declarations
  - Describe the purpose of the thing being declared
  - Include any knowledge about range of values, special encodings, etc.
  - Describe where the thing is used, if that's important.
- Statements
  - Describe the purpose of the statement or block of statements.
  - Describe any assumptions necessary for the correct execution of the statements
- Tricky Code
  - Any particularly tricky, clever or obscure piece of code should get a really large block comment that explains what's going on.
    **Tricky Code should really be rewritten.**

- Procedures and Functions
  - Describe what the procedure or function does
  - Describe the purpose of each parameter including any assumptions about parameter values or usage.
    Say if parameter is used for input, output or both.
- Data Structures
  - Describe the purpose of any complicated data structure
  - Describe any assumptions about how the data structure is used.
  - Describe how this data structure is linked to other data structures.

The purpose of comments is to make the program easier to understand. Use comments generously



## Identifiers (basic names)

- Identifiers start with a letter or _ ( underscore )
  Identifiers contain letters, digits or _
  Upper and lower case letters are distinct, e.g. $A \neq a$
  **Examples:** i , I , total , bigNumber , _DEBUG_ , testing_123
- Words that have a special meaning in C (*keywords* , See King Table 2.1 ) are reserved and can **not** be used as identifiers. **Examples: int, while, if**
- Identifiers are used to name variables, constants, types and functions in C.
- **Good Style:** Use mnemonic identifiers!!
  Mnemonic means that the identifier describes its purpose in the program, e.g. use *sum* and *index* instead of *Mary* and *fabulous*
  Mnemonic identifiers help you (and others) understand what the program is supposed to do.



## Basic Data Types

| Use | keyword(s) | constants |
|---|---|---|
| integers | **unsigned** , **int** , **short** , **long** | -237, 0, 23, 101467 |
| real numbers | **float** , **double** | -0.123, +417.6, 1234e+7, 0.23e-12 |
| characters | **char** | 'a', 'A', '3', '+' |

- Values of type **char** can be used in integer expressions.
- The character data type is for single characters.
  Character strings will be described later.
- **Notation:** the phrase *type-name* will be used to denote any valid type in C.
  **int** , **double** and **char** are instances of *type-name* .



## Integer Constants

| Type | Digits | Starts with | Examples |
|---|---|---|---|
| decimal integer | 0123456789 | 1..9 | 1 123L 456790U |
| octal integer | 01234567 | 0 | 01 0123 0456707U |
| hexadecimal integer | 01234567899 abcdef ABCDEF | 0x or 0X | 0x14 0x123 0XDEADBEEF |

Add an upper case L after a constant to force it to be treated as a **long** number.
**Good Style:** Don't use lowercase l (el), it looks like 1 (one).
Add an upper case U after a constant to force it to be treated as an **unsigned** number.

**WARNING:** numbers starting with 0 (zero) are *octal* constants.
123 and 0123 have *different* values. ( $0123_8 = 83_{10}$ )

## HOW TO Use Integer Types

- For almost all integer variables, use **int**
- Use **short** or **char** when saving space is *really* important AND IT'S KNOWN that the range of values for the variable will fit in -32768 **..** 32767.
  Short or char integer variables *may* be slower to access.
- On a few machines **long** is 64 bits and provides a much larger range of values. Don't assume **long** is larger than **int** unless you check.
- Use **unsigned** for STRICTLY NON-NEGATIVE integer values.
- For maximum portability use:
  **int** or **short int** for integers in the range -32768 **..** 32767
  **long int** for all other integers
  C standard only requires: sizeof( **short** ) $\leq$ sizeof( **int** ) $\leq$ sizeof( **long** )



## Character and String Constants

| Type | Contains | Starts and ends with | Examples |
|---|---|---|---|
| character | a single character | ' (single quote) | 'a' '@' '1' 'C' |
| string | arbitrary characters | " (double quote) | "abc" "CSC181F" "arbitrary" |

The backslash ( \ ) notation can be used to create character or string constants containing arbitrary non-printable characters.
See "Escape Sequences" in King Section 7.3.

**WARNING:** be careful to use character constants where a single character is required and string constants where a string is required.



## Variables and Types

- Variables are the basic containers used to hold data in C.
- Every variable must be declared before it is used.
- Every variable is associated with a specific type.
- The type of a variable detrermines the kind of values that it can hold and the amount of storage that is allocated for the variable.
- *scalar* variables can hold exactly one variable at a time. Non-scalar variables (e.g. arrays) can hold many different values at the same time.



## Declarations for Scalar Variables

- The declaration for a scalar variables has the form
  *type-name identifierList* [a] **;**
  **Examples:** **int** I **,** J **,** K **;**
  **char** tempChar **,** nextChar **;**
- A variable can be declared and initialized at the same time using
  identifier = expression
  **Example:** **int** xPosition = 25 **,** yPosition = 30 **;**
  **WARNING:** Each variable must be individually initialized.
  **int** M **,** N = 0 **;** /* *only* initializes N. */
- **Good Style:** All variables should be initialized with some value **before** they are *used* in a program.
- **BAD Style:** do **not** depend on the "system" to automatically initialize variables for you. This is *lazy and dangerous.* Someday the variables will be initialized to RUBBISH and your program will CRASH.

[a] *identifierList* is a comma separated list of identifiers

## Named Constants

- A *named constant* is an identifier that serves as a synonym (name) for a constant value.
- Named constants are often used to provide a *single point of definition* for a constant value that is used throughout a program.
- Using named constant makes programs more easily modifiable and easier to understand.
- Named constants makes program more readable, use mnemonic name for constant.
- Named constants makes program correctness easier to achieve.
- **Good Style: Avoid Magic Numbers**
  Use named constants for **all** values that have any significant impact on the program's operation.



## Defining Named Constants

- Use the **#define** construct to create named constants
  **#define** *identifier expression*
- The identifier becomes a synonym for the expression in the rest of the program. If the expression is a *constant expression* then the identifier can be used anywhere that a constant can be used
- **Good Technique:** ALWAYS enclose the expression in parentheses.
  **Good Style:** Use UPPER CASE names for defined constants to make them stand out in the program.
- **Examples:** **#define** CUBIC_IN_PER_LB ( 166 )
  **#define** SCALE_FACTOR (5.0 / 9.0)
  **#define** ARRAY_SIZE ( 100 )
- **WARNING:** common errors
  **#define** N = 100 /* WRONG, defines N to be "= 100" */
  **#define** N 100 ; /* WRONG, defines N to be "100 ;" */



## Named Types

- A *named type* is an type that has been associated with a specific identifier. Named types are created using the **typedef** declaration.
- Named types make program more easily modifiable, since there is a single point of definition for the type.
- If mnemonic names are used for the types, named types make programs more readable.
- Named types make it easier to write a correct program.
- **Good Style: Avoid Magic Types**
  Use named types for **all** types that have any significant impact on the program's operation.



## Typedef Declaration

- A named type is created with the declaration
  **typedef** *type-name identifier* **;**
- *type-name* can be any valid type including compound types or a new type definition.
- *identifier* becomes a new name for this type
- **Good Style:** Use typedefs for all complicated types.

**Examples:**

**typedef long int** portableInt **;**
**typedef float** realType **;**
portableInt I **,** J **,** A[ 100 ] **;**
realType xAxis **,** yAxis **,** zAxis **;**

## Reading Data and Printing in C[a]

- Input and Output are *not* part of the C language.
- Builtin library functions are used for all reading and printing.
- Put the construct

  **#include** $<$ stdio.h $>$

  at the start of every program to make the builtin input and output functions available.
- The *printf* function does simple printing
- The *scanf* function is used to read values into variables.

[a] The description of printf and scanf below is intended to get you started. Reading and Printing will be discussed in more detail later in the course.



## Format Strings

- A *format string* is used to specify how printf and scanf should operate.
  For printf the format string specifes exactly how the printed output should look.
  For scanf the format string specifes the exact form of the input that will be read.
- The format string consists of
  - *conversion specifications* a percent sign ( **%** ) followed by some optional information, followed by a *conversion specifier character* that indicates the *type* of data to be printed or read.
  - *ordinary text* Everything else. Printed as is by printf. Matched against the data being read by scanf[a]

[a] This feature is rarely used



## Conversion Specifier Characters

| Type | Conversion Specifier | |
|---|---|---|
| **int** | %d or %i | |
| **char** | %c | |
| Strings | %s | |
| **double** | %f or %e or %g | printf |
| **double** | %lf or %le or %lg | scanf |
| **float** | %f or %e or %g | |
| **short** | %hd | |
| **long** | %ld | |

Note that printf and scanf use different specifiers for **double** .
Use %% to print a %. Use \n to print a newline.



## The printf Function

**printf** ( *format-string* **,** *expressionList* [a] ) **;**

- The *format-string* controls how the information is printed.
- The expressions in the *expressionList* are printed in the order given.
  The type of each expression must be compatible with the % item used in the *format-string*
- By default each expression is printed using the minimum number of characters required to express its value.
  All formatting and spacing must be provided by the programmer.

[a] An *expressionList* is a list of expressions separated by commas

## Printing Technique

- In a format string a constant between the % and the following control character, specifies that the expression is to be printed using the number of characters specified by the constant. This feature can be used to print columns of values. **Examples:** `%10d` `%16e`
- The printf prints to *standard output*.
  If you are working at a terminal, this means printing to your screen.
  There are Unix/Windows commands that let you redirect standard out to a file.
- **WARNING:** make really sure that the type of the expression matches the type of % character that you use to print it.
- **WARNING:** make really sure that you have provided a % character for each expression in the expression list.



## Examples Printing the Value of a Scalar Variable

Examples:

```
int i , height , width ;
char c ;
float x ;
double y ;

printf("%d", i) ;
printf("%c", c) ;
printf("%f", x) ; /* decimal form               */
printf("%e", x) ; /* scientific form            */
printf("%g", y) ; /* decimal or scientific form */

printf("height is %d and width is %d\n", height , width ) ;
printf("i = %8d, x = %14f\n", i, x) ;
printf("\n") ;    /* blank line */
```



## The scanf Function

**int scanf** ( *format-string* **,** *variable-address-list*[a] ) **;**

- The *format-string* controls how the information is read
  Any ordinary text in the format string must match the input *exactly*.
- scanf attempts to read values for each of the variables in the order given.
  The type of each variable **must** be compatible with the % item used in the *format-string*
- scanf automatically skips *white space* between input values

[a] A *variable-address-list* is a list of **addresses** of variables separated by commas



- scanf returns the number of varaibles that it successfully read and stored.
  Returns special value EOF if an error or end of input was detected.
- Use the address-of operator & to create the addresses of variables for the arguments to scanf.
  The address-of operator is **almost always REQUIRED**.
- **WARNING:** forgetting the address of operator in a call to scanf will almost certainly cause your program to CRASH.
- **Good Style: always** check the value returned by scanf to make sure that you read as many varaibles as you expected to[a].

[a] We may sometimes not do this check in these slides in order to keep the examples simple, but it **should always be done**

## Examples - Reading Input

Examples:

```
int i , k ;
char c ;
double y ;

scanf("%d", &i) ;   /* read one integer */
scanf("%c", &c) ;   /* read one character */
scanf("%lf", &y) ;  /* read one double value */
scanf("%d%d", &i, &j );  /* read two integers */
```



## Main Program in C

- In C the main program is a function called *main*
- The body of the function is enclosed in left ( { ) and right ( } ) curley braces.
- Minimal main program example:

```
#include <stdio.h>
main()
{
   /* declarations and statements go here */
}
```