What's Important

Atypical 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 English words are produced for every source code statement in the software. Many new software professionals are surprised when they spend more time producing words than writing code.


---

**CSC180F & CSC181F**

CSC180F (CSC181F) teaches effective programming more emphasis on concepts and style covers advanced material covers C++ covers introductory C more (advanced) C assumes previous programming knowledge assumes previous experience

---

CSC181F 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. Copying for purposes other than this use and all forms of distribution are expressly prohibited.


© Ray Ortigas, 2000

© Hiroshi Hayashi, 1997

Assignments and Tests

Assignments are due at the start of lecture on the due date. Assignments must 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. Consult the instructor if you need special consideration.

The test and exam are closed book, but a single page (6.5 by 11 inch, both sides) and a non-programmable calculator will be allowed.

Special consideration:
- Assignments may be handed in early to the instructor.
- Assignments are due at the start of lecture on the due date.

Assignments

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Due Date</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment 1</td>
<td>26 Sep</td>
<td>4%</td>
</tr>
<tr>
<td>Assignment 2</td>
<td>10 Oct</td>
<td>4%</td>
</tr>
<tr>
<td>Term Test 1</td>
<td>TBA (Week of 16 Oct)</td>
<td>20%</td>
</tr>
<tr>
<td>Assignment 3</td>
<td>31 Oct</td>
<td>4%</td>
</tr>
<tr>
<td>Assignment 4</td>
<td>14 Nov</td>
<td>4%</td>
</tr>
<tr>
<td>Term Test 2</td>
<td>TBA (Week of 20 Nov)</td>
<td>20%</td>
</tr>
<tr>
<td>Assignment 5</td>
<td>5 Dec</td>
<td>4%</td>
</tr>
<tr>
<td>Final Exam</td>
<td>TBA (December)</td>
<td>40%</td>
</tr>
</tbody>
</table>

Grading Scheme

Books for CSC181F

Required Textbooks

Recommended Textbooks

Supplemental Reading
Program Construction

1. Understand the Problem
2. Design Algorithms and Data Structures
3. Design to Program
4. Write the Program
5. Inspect the Program for Errors
6. Test the Program Thoroughly
7. Compile and Debug the Program
8. Document the Program

Guidelines for Avoiding Plagiarism

You may discuss assignments with friends and classmates, but only 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 or read their code. You should not look at another student's solution or read their code. No part of your final program should be taken from another person's work. No part of your final program should be taken from another person's work.

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.

Plagiarism and Cheating

Plagiarism is a kind of fraud: passing off someone else's work or ideas as your own.

The assignments you hand in must be your own and must not contain anyone else's work. You can reasonably expect that the assignments you hand in will be used to evaluate your performance in the course. The assignments you hand in must be your own and must not contain anyone else's work.
Program Development

Requirements Specification Design Implementation Debugging & Testing

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

Important Considerations
Correctness
Program maintainability and modifiability
Programmer's efficiency
Program's efficiency

Stepwise refinement
Choose data structures
Choose algorithms
Simplicity - keep it simple stupid
Think before programming
KISS - keep it simple stupid
Solve most general instance of problem, minimize special cases
Think before programming
Choose algorithms
Choose data structures
Simplicity - keep it simple stupid
Solve most general instance of problem, minimize special cases
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?

What are the special cases?

What are the inputs?

What are the outputs?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the special cases?

What are the general cases?

What are the inputs?

What are the outputs?

What are the general cases?

What are the inputs?

What are the outputs?

What are the general cases?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the outputs?

What are the general cases?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the outputs?

What are the general cases?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the outputs?

What are the general cases?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the outputs?

What are the general cases?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the outputs?

What are the general cases?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the outputs?

What are the general cases?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the outputs?

What are the general cases?

Have I solved this problem or a similar problem before?

Is the general case of the problem?

How are special cases related to inputs?

What are the outputs?

What are the inputs?

What are the outputs?

What are the general cases?
Stepwise Refinement Example

Problem: Solve quadratic equations

Analysis

Special cases a, b, and/or c = 0.

General case z real or imaginary roots

Applicable algorithm

Design by Stepwise Refinement

Start with the whole problem

Subdivide problem into several separate subparts

Decide how to represent data and carry it between subparts

e.g. input, complete output

Subdivide problem and several separate subparts

Start with the whole problem

Simplest is the virtue

KISS - Keep It Simple Stupid

More efficient or more general

Iterate on design

Minimize special cases and exceptions

Design Technique

Data structures then algorithms

Problem Analysis

Analogy, Conditions

Decomposition & Recomposition

Induction, work backwards

Solve subproblems & generalize

Use all the problem description

Analogies; Conditions

Iterate on simplifying solution

Check the results

Identify intermediate results

Induction, work backwards

Solve subproblems & generalize

Use all the problem description

Analogies; Conditions
Third Refinement

If no valid roots if \( a = 0.0 \) and \( b = 0.0 \):
- Special case if \( a = 0.0 \) and \( b = 0.0 \):
  - Non-special case if \( a = 0.0 \) and \( b = 0.0 \):
    - If non-special case, compute roots.
    - Roots are real if \( a = 0.0 \)
    - Otherwise, roots are imaginary.

- Computer roots if special case.

Second Refinement

First Refinement

Quadratic Equation Case Analysis

Applicable algorithm is only valid when \( a \neq 0.0 \).

<table>
<thead>
<tr>
<th>( a = 0.0 )</th>
<th>( a \neq 0.0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0.0 = c + ax )</td>
<td>two real roots</td>
</tr>
<tr>
<td>( 0.0 = c + bx )</td>
<td>two real roots</td>
</tr>
<tr>
<td>( 0.0 = c + cx )</td>
<td>two real roots</td>
</tr>
<tr>
<td>( 0.0 = c )</td>
<td>no roots</td>
</tr>
<tr>
<td>( 0.0 = c + ax )</td>
<td>one real root</td>
</tr>
<tr>
<td>( 0.0 = c + bx )</td>
<td>one real root</td>
</tr>
<tr>
<td>( 0.0 = c + cx )</td>
<td>one real root</td>
</tr>
<tr>
<td>( 0.0 = c )</td>
<td>( \infty ) roots</td>
</tr>
<tr>
<td>( 0.0 = c + ax )</td>
<td>( \infty ) roots</td>
</tr>
<tr>
<td>( 0.0 = c + bx )</td>
<td>( \infty ) roots</td>
</tr>
<tr>
<td>( 0.0 = c + cx )</td>
<td>( \infty ) roots</td>
</tr>
</tbody>
</table>

If \( c = 0.0 \), then algorithm is overkill.

If \( c = 0.0 \), then algorithm is overkill.

If \( c = 0.0 \), then algorithm is overkill.

If \( c = 0.0 \), then algorithm is overkill.
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 it is efficient, easy to understand, and easy to get correct. An entire slide of Good Technique usually has a 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.

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

There are good software tools for developing C programs

Debuggers, program checking aids

Large libraries of existing software

Supplementary reading

K. N. King, Chapter 3

Chapter 2

Reading Assignment

K. N. King, Sections 7.1 to 7.3, 7.6

Chapter 19

S. McConnell
HOWTOComment

Declarations
- Describe the purpose of the thing being declared
- Include any knowledge about range of values, special encodings, etc.
- Describe the purpose of the thing being declared

Statements
- Describe the purpose of the statement or block of statements
- Describe any assumptions necessary for the correct execution of the statements

Tricky Code
- An particularly tricky, clever or obscure piece of code should get a really large block comment that explains what's going on.
- Block comment that explains what's going on

WARNING:
An unterminated comment in C will silently delete part of your program.

Block Comment Styles

Simple, unadorned

/*first line
following lines*/

Head and Trail Markers

/**************************************************************/
first line of comment
manymorelinesofcomment
**************************************************************/

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

Example:
This code is ignored
#if 0
 This code is ignored
#endif

Good Style: Do not use comments to delete code from a program.

Programmers commonly use comments to explain the program in English instead of English
Programmers commonly use comments to explain the program in English instead of English
Programmers can print comments anywhere in a blank, including
Comments should be added to the readers understanding of the program

Comments

/* Example */
root1 = x * y - z;
root2 = root1 / (a - c);

Good Style: Use comments with the characters /* and */

/* Example */
root1 = x * y - z;
root2 = root1 / (a - c);

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

Example:
This code is ignored
#if 0
 This code is ignored
#endif
Integer Constants

TypeDigitsStartswithExamples

decimal integer: 01234567891..91123L456790U

octal integer: 0123456700101230456707U

hexadecimal integer: 01234567899abcdefABCDEF0xor0X0x140x1230XDEADBEEF

Add an uppercase L after a constant to force it to be treated as a long number.

Good Style:
Don’t use lowercase (el), it looks like 1 (one).

Add an uppercase 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 = 8310)

Identifiers (basic names)

Identifiers start with a letter or (underscore).
Identifiers contain letters, digits or upper and lower case letters are distinct; e.g. G, g
Examples: i, I, total, bigNumber, DEBUG, testing

Words that have a special meaning in C (keywords) are reserved and cannot be used as identifiers.
Examples: int, while, if, do, else, return, break, continue, return, for, while, do, etc.

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

real numbers: float, double

characters: char
Values of type char can be used in integer expressions.

Values or type char can be used in integer expressions.

use keywords (type constants)

Procedures and Functions

– Describe what the procedure or function does
– Describe the purpose of each parameter including any assumptions about parameters and return values
– Describe how the data structure is linked to other data structures
– Describe how the data structure is used
– Describe the purpose of any complicated data structure

Data Structures

– Describe how the data structure is linked to other data structures
– Describe how the data structure is used
– Describe the purpose of any complicated data structure
– Describe how the data structure is linked to other data structures

Notation:
The phrase type-name will be used to denote any valid type in C.

Characters and chars

Characters: a, A, 0, +

Real numbers: float, double

-0.123, +417.6, 1234e+7, 0.23e-12
Declarations for Scalar Variables

The declaration for a scalar variable has the form

```
type-name identifierList
```

Examples:
```
int I, J, K;
char tempChar, nextChar;
```

A variable can be declared and initialized at the same time using

```
identifier = expression
```

Examples:
```
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.

```
identifierList
```

is a comma-separated list of identifiers

**Character and String Constants**

Type Contains Start/End Examples
```
character as single character 'a' '@' '1' 'C'
```

```
string arbitrary characters "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:**
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 in a program.

```
```

The types are the basic containers used to hold data in C.

**HOW TO Use Integer Types**

For almost all integer variables, use:

```
int (long) (unsigned)
```

For maximum portability use:

```
short (int) (long)
```

For maximum efficiency use:

```
char (short) (int) (long)
```

Long int or long integer is best if available in the range ~37768 .. 37767.

If short int is available, use it if numbers you work with do not exceed the range of short int (16 bits) and provides a much larger range of values. Don't assume long is larger than int unless you check.

**NOTE:**
Short or char integer variables may be slower to access.

For most integer variables, use int.

C standard only requires: sizeof (short) <= sizeof (int) <= sizeof (long)
Typedef Declaration

An named type is created with the declaration:

```
type-name identifier;
```

*type-name* can be any valid type including compound types or a new type.

*identifier* becomes a new name for this type.

**Good Style:** Use typedefs for all types that have any significant impact on the program.

**Examples:**

```c
typedef long int portableInt;
typedef float realType;
```

**Named Types**

A named type is a type that has been associated with a specific name.

```
typedef type-name identifier;
```

**Example:**

```c
typedef int sqtn;
```

**WARNING:** Common errors:

- `#define N=100 /* WRONG, defines N to be "=100" */`
- `#define N100 /* WRONG, defines N to be "100;" */`

**Named Constants**

A named constant is an identifier that serves as a synonym for a constant value.

```
#define identifier expression
```

The identifier becomes a synonym for the expression in the rest of the program.

**Good Style:**

- Avoid Magic Constants
- Use identifiers for all values that have any significant impact on the program.

**Examples:**

```c
#define SCALE_FACTOR (5.0/9.0)
#define ARRAY_SIZE (100)
```

**Named Constants**

- Use uppercase names for defined constants to make them more readable.
- Use mnemonic names for defined constants to make them easily understandable.
- Avoid using constants that are used throughout a program.
- Avoid using constants that are clear and easy to maintain.
- Avoid using constants that are defined in a single file.

**Named Constants**

A named constant is a constant that has been associated with a specific name.
The printf Function

printf(format-string, expressionList);

The format-string controls how the information is printed. The expressions in the expressionList are printed in the order given. 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. The type of each expression must be compatible with the % item used in the format-string. The expressions in the expressionList are printed in the order given. The format-string consists of:

- conversion specifiers

Format Strings

A format string is used to specify how printf and scanf operate. 

The printf Function

Format Strings

Conversion Specifier Characters

<table>
<thead>
<tr>
<th>TypeSpecifier</th>
<th>%d or %i</th>
<th>%c</th>
<th>%s</th>
<th>%f for %e or %g printf</th>
<th>%l for %le or %lg scanf</th>
<th>%f for %e or %g scanf</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strings</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that printf and scanf use different specifiers for double:

Use \% to print a %. Use \n to print a newline.

Conversion Specifier Characters

Reading Data and Printing in C

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

The scanf function is used to read values into variables.

The printf function does simple printing.

The scanf function is used to read values into variables.

The printf function does simple printing.

Builtin library functions are used for all reading and printing.

Input and Output are not part of the C language.
Scanning variables

**typedef**: A scanf in the C family

- **scanf** reads from stdin.
- **printf** writes to stdout.

**scanf**

- **scanf** reads from stdin.
- **printf** writes to stdout.

scanf takes a format string and a variable list.

**Printf**

- **printf** writes to stdout.
- **scanf** reads from stdin.

Example:

```c
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*/
```

**Printf**

- **printf** writes to stdout.
- **scanf** reads from stdin.

Example:

```c
int scanf(int format-string, variable-address-list);
```

**Printf**

- **printf** writes to stdout.
- **scanf** reads from stdin.

Example:

```c
/* printf prints to standard output. */
/* scanf reads from stdin. */
```

**Printf**

- **printf** writes to stdout.
- **scanf** reads from stdin.

Example:

```c
int scanf(int format-string, variable-address-list);
```

**Printf**

- **printf** writes to stdout.
- **scanf** reads from stdin.

Example:

```c
int scanf(int format-string, variable-address-list);
```

**Printf**

- **printf** writes to stdout.
- **scanf** reads from stdin.

Example:

```c
int scanf(int format-string, variable-address-list);
```
In C the main program is a function called main.

Examples:

Minimal main program example:

```
#include <stdio.h>

int main() {
    /* declarations and statements go here */
    
    return 0;
}
```

Examples - Reading Input

```
#include <stdio.h>

int main() {
    /* declarations and statements go here */
    
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    /* declarations and statements go here */
    
    return 0;
}
```