Elements of a Computer System

1. Input Device
2. Output Device
3. Processor (CPU)
4. Memory
5. External Storage
Steps in Programming

1. Problem Analysis
2. Algorithm Design
3. Coding
4. Testing & Debugging
5. Documentation
A good program is:

1. Correct
2. Easy to read & understand
3. Easy to modify
4. Concise
5. Completed on time & within budget
Running a C Program

1. Editor
2. Preprocessor
3. Compiler
4. Linker
5. Load & Execute
A Simple C Program

/*
 * Simple addition program.
 */

#include <stdio.h>

main (){
    int A, B, Sum;

    printf ("Enter two integers ==> ");
    scanf ("%d %d", & A, & B);

    Sum = A + B;

    printf ("The sum of %d and %d is %d\n", A, B, Sum);
    return 0;
}
Flowchart Symbols

Start/Stop

Process

Decision

Input/Output
Program Structures

*Sequence*:

![Diagram of sequence structure]
Program Structures

Selection:

[Diagram of selection structure with decision diamond and flowchart]

T  F

[Diagram showing flowchart and decision process]

8 9 February 2000
Program Structures

Iteration (i.e., loop):

1. **Flowchart 1**
   - Start (oval)
   - Decision (diamond) with conditions: T or F
   - Processing block
   - Loop back to decision

2. **Flowchart 2**
   - Start (oval)
   - Decision (diamond) with conditions: T or F
   - Processing block
   - Loop back to decision
General Form of a C Program

Initial comments

#include’s

Global declarations

Function prototypes

main ()
{
    Local Declarations
    Statements
}

Function definitions
Data

Data is information.

Basic types of data:

**Integer**  Stored as a string of bits like a car odometer. Has limited range.

**Floating-Point**  Stored in “scientific-notation” form (e.g., \(0.123 \times 10^3\)). Has a limited number of significant digits, and limited exponent range.

**Character**  Typically encoded in 1 byte (8 bits) using the ASCII coding scheme.

**String**  A sequence of characters. Represented by contiguous characters in memory, with the special “null character” (’\0’) used to indicate the end of the sequence.
Primitive C Data Types

void
char
short or short int
int
long or long int
float
double
long double

The integer types (including char) can be specified as signed or unsigned, e.g., unsigned long int
Backus-Naur Form (BNF)

Names in *italics* represent sets of strings

::= indicates a definition of a name (there can be more than one)

| denotes alternatives

{...}_1 means choose one of the enclosed entries

{...}_0+ means repeat the enclosed zero or more times

{...}_1+ means repeat the enclosed one or more times

{...}_opt means the enclosed is optional
BNF Examples

\[
digit ::= 0 \mid 1 \mid \cdots \mid 9
\]
\[
lowercase ::= a \mid b \mid \cdots \mid z
\]
\[
uppercase ::= A \mid B \mid \cdots \mid Z
\]
\[
letter ::= lowercase \mid uppercase
\]
\[
identifier ::= \{letter \mid _\} \{letter \mid digit \mid _\}^{0+}
\]
\[
declaration ::= 
\quad \text{type variable} \ \{= \text{constant}\}^{0+}
\quad \{, \text{variable} \ \{= \text{constant}\}^{0+}\}
\]
\[
ifstatement ::= 
\quad \text{if} \ ( \text{integerexpression} )
\quad \text{statement}
\quad \{\text{else}
\quad \text{statement}\}^{0+}
\]
**Variables**

A *variable* is a name for a storage area in memory. It represents 2 things:

- The *type* of data stored. This includes both the size of the storage area (*i.e.*, number of bytes) and the type of bit-encoding to use.

- The *address* of the storage area, *i.e.*, where it is in memory.

In C/C++ variable names are composed of letters, digits and the underscore character (_), and must begin with a letter or underscore (why?).

Upper- and lower-case letters are different, *e.g.*, `fed_tax` and `Fed_Tax` are different names.
## C/C++ Keywords

The following identifiers have special meaning in C/C++ and should **NOT** be used as variable names

<table>
<thead>
<tr>
<th>asm</th>
<th>double</th>
<th>new</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>else</td>
<td>operator</td>
<td>template</td>
</tr>
<tr>
<td>break</td>
<td>enum</td>
<td>private</td>
<td>this</td>
</tr>
<tr>
<td>case</td>
<td>extern</td>
<td>protected</td>
<td>throw</td>
</tr>
<tr>
<td>catch</td>
<td>float</td>
<td>public</td>
<td>try</td>
</tr>
<tr>
<td>char</td>
<td>for</td>
<td>register</td>
<td>typedef</td>
</tr>
<tr>
<td>class</td>
<td>friend</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>const</td>
<td>goto</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>continue</td>
<td>if</td>
<td>signed</td>
<td>virtual</td>
</tr>
<tr>
<td>default</td>
<td>inline</td>
<td>sizeof</td>
<td>void</td>
</tr>
<tr>
<td>delete</td>
<td>int</td>
<td>static</td>
<td>volatile</td>
</tr>
<tr>
<td>do</td>
<td>long</td>
<td>struct</td>
<td>while</td>
</tr>
</tbody>
</table>
Declaring Variables

A variable declaration is the type of the variable followed by a comma-separated list of variable names of that type, ending with a semi-colon.

Example:

```plaintext
int Larry, Moe, Curly;
double Income;
double Tax;
char Initial, Name [50];
```

Variables can be given an initial value when they are defined by putting the value after an equal sign.

Example:

```plaintext
int Tom, Dick = 3, Harry;
double Batting_Average = 0.0;
char ID_String [50] = "Loyola Student";
```
Constants

**Integer Constants**: A string of digits, optionally preceded by a + or −. No decimal point allowed. If it begins with 0 it is regarded as octal (base-8). If it begins with 0x or 0X it is regarded as hexadecimal (base-16).

**Floating-Point Constants**: A number that has a decimal point or is in “scientific-notation” form, e.g., 3e-2 represents $3 \times 10^{-2}$.

**Character Constants**: A single character between single-quotes, e.g., ’a’. Can also specify an octal value after a \, e.g., ’\101’.

**String Constants**: A sequence of characters inside double-quotes, e.g., "my answer". Note that this example uses 10 bytes of memory (why?).
Common Escape Sequences

The following are special character-constant codes:

\n Newline. Go to beginning of next line.
\t Tab. Go to next tab stop.
\r Return. Go to beginning of current line.
\a Alarm. Make a noise.
\\ Backslash character.
\" Double quote character.
C Statements

• expression ;
  Includes assignments and function calls.

• if ( expression )
  statement
  else
  statement

• while ( expression )
  statement

• do
  statement
  while ( expression );
C Statements

• for ( expr₁ ; expr₂ ; expr₃ )

  \textit{statement}

• switch ( expression )

  \{
    \text{case expression}_1 : statements_1 \\
    \text{case expression}_2 : statements_2 \\
    \vdots \\
    \text{default : statements}
  \}

• \text{break;}

• \text{continue;}
C Statements

• `return expression;`

• `goto identifier;`

• `{ declarations statements }
  A compound statement
Operators

Grouped by precedence:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>::</td>
<td>scope resolution</td>
<td>left-to-right</td>
</tr>
<tr>
<td>()</td>
<td>function call</td>
<td>left-to-right</td>
</tr>
<tr>
<td>[]</td>
<td>array index</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>field in structure</td>
<td></td>
</tr>
<tr>
<td>-&gt;</td>
<td>field in pointed-to structure</td>
<td></td>
</tr>
</tbody>
</table>
## Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>not</td>
<td>right-to-left</td>
</tr>
<tr>
<td>~</td>
<td>one’s complement</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>negation</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>address of</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>contents pointed to</td>
<td></td>
</tr>
<tr>
<td>(type)</td>
<td>type casting</td>
<td></td>
</tr>
<tr>
<td>sizeof</td>
<td>storage size</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>left-to-right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>remainder (mod)</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>left-to-right</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td></td>
</tr>
</tbody>
</table>
## Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;&lt;</code></td>
<td>left shift</td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>right shift</td>
<td></td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
<td></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal</td>
<td></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal</td>
<td></td>
</tr>
<tr>
<td><code>==</code></td>
<td>equal</td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>not equal</td>
<td></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>bitwise AND</td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>^</code></td>
<td>bitwise XOR</td>
<td>left-to-right</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>bitwise OR</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>AND</td>
<td>left-to-right</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
</tr>
</tbody>
</table>
## Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>? :</td>
<td>conditional</td>
<td>right-to-left</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>*=</td>
<td>multiply and assign</td>
<td></td>
</tr>
<tr>
<td>/=</td>
<td>divide and assign</td>
<td></td>
</tr>
<tr>
<td>%=</td>
<td>mod and assign</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>add and assign</td>
<td></td>
</tr>
<tr>
<td>-=</td>
<td>subtract and assign</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>shift left and assign</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>shift right and assign</td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td>AND and assign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>OR and assign</td>
</tr>
<tr>
<td>^=</td>
<td>XOR and assign</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>compound</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>
Division

Division of two integers discards the remainder, \textit{i.e.}, it rounds down. If either value is a floating-point, then the result is floating-point and will include any fractional part.

\begin{verbatim}
int i = 9, j = 4;
double a = 9.0, b = 4.0, x, y;

x = i / j;
printf ("x = %.2f\n", x);

y = a / b;
printf ("y = %.2f\n", y);
\end{verbatim}

produces the following output:

x = 2.00
y = 2.25
Basic Formatted Output

The function to output values to the screen in C is `printf`. Its general format is:

```
printf (format string, value list)
```

The format string is printed as-is, except that any conversion codes in it have the corresponding value from the value list substituted for them.

Conversion codes begin with a percent sign and end with a letter. The letter indicates the type of value:

- c character
- d decimal integer
- e floating point (scientific notation)
- f fixed-point
- o octal integer
- s string
Basic Formatted Output

Between the % and the letter you can optionally specify numbers to describe the width of the corresponding output field. The number is the number of bytes to reserve for the output value. A positive number makes the output value align to the right side of the output field. A negative number makes it align to the left side. A leading zero makes unused positions zero instead of blanks.

For floating-point values a decimal-point followed by an integer can be used to specify the number of decimal places to use in the output.

Example:

```c
int i = -27;
double x = 3.453;

printf ("i: >%d< >%5d< >%-5d<\n", i, i, i);
printf ("x: >%f< >%5.1f< >%5.2f<\n", x, x, x);
```

produces the following output:

```
i: >-27< >␣␣-27< >-27␣␣<
x: >3.453000< >␣␣3.5< >␣3.45<
```
Basic Formatted Input

The function to input values from the keyboard in C is `scanf`. Its general format is:

```
scanf (format string, address list)
```

The format string describes how the input should look and contains conversion codes for the corresponding entries in the address list.

The conversion codes are essentially the same as those used for output. Usually less precise specifications are needed. The address list entries are usually variable names preceded by an `&`.

**Example:**

```c
int i, j, k;
double x, y;

scanf ("%d %d %d", & i, & j, & k);
scanf ("%f %f", & x, & y);
```

When reading values other than characters, any “white space” in the input (spaces, tabs, newlines) is ignored.