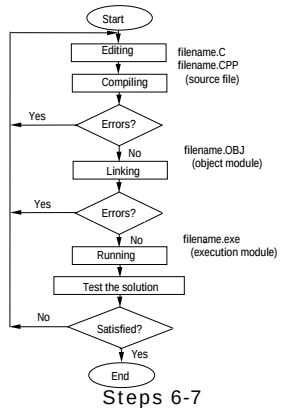
Introduction to Algorithms

Steps in Problem-solving

1. Understanding the problem.
2. Mathematical description.
3. Choose a numeric method.
4. Algorithm specification.
5. Coding the program.
6. Program execution.
   - Editing.
   - Compiling.
   - Linking.
   - Running.
7. Test the solution and fix problems.



Steps 6-7

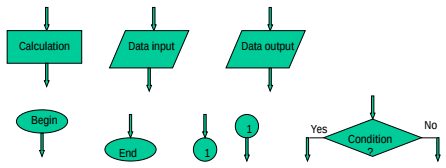
Algorithm - Definition, Characteristics, Types, Presentation

1. Definition

   Algorithm is a problem-solving method suitable for implementation as computer program.

2. Characteristics
   - Definiteness.
   - Discretion.
   - End.
   - Input data.
   - Output results.
   - Mass.

3. Presentation
   - Word presentation.
   - Block diagrams.



   - Pseudo code.

4. Types
   - Linear.
   - Selection.
   - Iterations.

Introduction to C Programming Language

Programming language defines a set of rules that determines exactly how a programmer can code the algorithms and data structures into a program.

C was designed for and implemented by Dennis Ritchie in the 1970s on a DEC PDP-11 that used the UNIX operating system.

C is a middle-level language – combines the best elements of high-level languages with the control and the flexibility of assembly language.

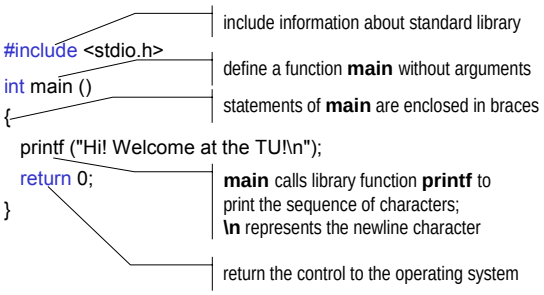
C code is portable.

C is a structured language.

Getting Started

Example: Print a sentence

Hi! Welcome at the TU!



Exercise: What is the output of the program?

```
#include <stdio.h>
int main ()
{
   printf ("Hi! ");
   printf ("Welcome ");
   printf ("at ");
   printf ("the ");
   printf ("TU!");
   printf ("\n");
   return 0;
}
```

Exercise: Try to call the function **printf** like

```
printf ("Hi! Welcome at the TU!
");
```

Exercise: Experiment to find out what happens when **printf**'s argument string contains **\a**.

Identifiers

Identifier
- sequence of letters, digits, and underscore (_)
- begins with a letter
- upper and lower case letters are distinct

product Product
gradeOfGroup grade_of_group
X1 x1
Max max

Comments

Comment begins with /* and terminates with */.

```
/* First program: Print a sentence */
```

Types, Operators, and Expressions

Data Types and Sizes

char	a single byte, capable of holding one character in the character set
int	an integer
float	single-precision floating point
double	double-precision floating point

Type	Size [bytes]	Range
unsigned short	2	0÷65535
short int	2	-32768÷32767
unsigned int	4 (1 word)	0÷4294967295
int	4 (1 word)	-2147483648÷ 2147483647
unsigned long	4	0÷4294967295
long int	4	-2147483648÷ 2147483647

Type	Size [bytes]	Range	Precision
float	4 (1 word)	+/-3.4E38	7 digits
double	8 (2 words)	+/-1.7E308	15 digits
long double	10	+/-1.2E4932	

Type	Size [bytes]	Region
char	1	-128÷127
signed char	1	-128÷127
unsigned char	1	0÷255

Constants

1234	int
1234L or 1234l	long int
1234U or 1234u	unsigned
1234UL or 1234ul	unsigned long
0123 0123L 0123U 0123UL	octal integer (leading 0)
0x123 or 0X123 0x123L 0x123U 0x123UL	hexadecimal integer (leading 0x or 0X)
123.4 or 1.234e2 or 1.234E2	double
123.4F or 123.4f	float
123.4L or 123.4l	long double
'x' '1' '+'	char

Escape sequences – two characters representing only one character

\a	alert (bell)	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	\0oo	octal number
\t	horizontal tab	\xhh	hexadecimal number
\v	vertical tab		

Define symbolic constant

```
#define name replacement_text
#define VTAB '\013' /* vertical tab as octal number */
#define VTAB '\xb'  /* vertical tab as hexadecimal number */
```

comments

Constant expression – involves constants

```
#define SIZE 100
int array[SIZE];      /* integer array with SIZE elements */
```

String constant (string literal) – sequence of zero or more characters surrounded by double quotes.

```
"I am a student"      /* string constant */
""                    /* empty string */
```

Enumeration constant – set of named constants; the first name has value 0, the next 1, and so on, unless explicit values are specified.

enum name {constant$_1$, …, constant$_n$};

```
enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
             SET, OCT, NOV, DEC};
```

Variables

Variable is a named location in memory that is used to hold a value.

Declaration

Variables must be declared before use.

type variable_name [[= expression]][[, ...]];

```
int age, top;
char c;
```

Variables may be initialized in its declaration.

```
int limit = 100;
char esc = '\\';
float eps = 1.0e-5f;
```

Assignment Operator =

Assignment operator sets the variable to the value of the expression.

variable = expression;

```
age = 18;
c = 'A';
```

Input and Output

Library functions provide input and output.

Text stream is a sequence of lines; each line ends with a newline character.

The library <stdio.h> defines the symbolic constant EOF (end of file).

```
#define EOF -1
```

```
int getchar (void);
```

getchar reads one character at a time from the standard input (keyboard) and returns the next input character, or EOF when it encounters end of file.

```
int putchar (int c);
```

putchar puts the character c on the standard output (screen) and returns the character written, or EOF if an error occurs.

```
int c;
c = getchar ();
putchar (c);
```

```
int printf (char *format, arg1, arg2...);
```

printf converts, formats, and prints its arguments arg_1, arg_2, ... on the standard output under control of the format string; returns the number of characters printed, or EOF if an error occurs.

format string
"% [[*flag*]] [[*width*]][[.*precision*]] [[*interpretation*]] *type*"

flag		right adjustment
	-	left adjustment
	+	prints the number with + or - sign
width	n	minimum field width
.precision	.n	maximum number of characters of a string, or number of digits after the decimal point of a floating-point value

interpretation	h	short int
	l	long int or double
	L	long double
type	d, i	int; decimal number
	o	int; unsigned octal number
	u	int; unsigned decimal number
	x, X	int; unsigned hexadecimal number
	f	float; [-]m.dddddd
	e,E	float; [-]m.dddddde±xx or [-]m.ddddddE±xx
	g,G	float; as %f or %e
	c	char; single character
	s	char *; character string
	%	symbol %
	p	void *; pointer

int scanf (char *format, &arg$_1$, &arg$_2$…);

scanf reads characters from the standard input, interprets them according to the specification in format, and stores the results through the arguments arg$_1$, arg$_2$, ... that must be pointers; returns the number of successfully matched items, or EOF when it encounters end of file or an error occurs.

scanf stops when it exhausts its format string, or when some input fails to match the control specification.

scanf ignores white spaces (blanks, tabs, newlines) in its format string.

```
int day, month, year;
scanf ("%d %d %d", &day, &month, &year);
printf ("Today is %d/%d/%d.\n", day, month, year);
```

20 1 2006
Today is 20/1/2006.

Literal characters can appear in the format string.

```
scanf ("%d/%d/%d", &day, &month, &year);
```

20/1/2006

```
char x, y, z;
scanf ("%c %c %c", &x, &y, &z);
printf ("%c%c%c", x, y, z);
```

1 2 3
1 2

Expressions and Operators

Expression consists of operands and operators.

Operands can be constants, variables, functions or their combinations.

Operators
- Arithmetic
- Relational
- Logical
- Bitwise

Operators
- Unary
- Binary

Arithmetic Operators

+ addition
- subtraction, also unary minus
* multiplication
/ division
% modulus
++ increment (adds 1 to its operand)
-- decrement (subtracts 1 from its operand)

++x (--x) – prefix form – the operand x is incremented/decremented by 1; the value of the expression is the value after the incrementation / decrementation

x++ (x--) – postfix form – the value of the expression is the value of the operand x; after the value is noted, the operand x is incremented / decremented by 1

Precedence of the arithmetic operators

Highest	++ --	
	- (unary minus)	
	* / %	
Lowest	+ -	

```
int x, y;
x = 14;
y = 4;
x / y        /* integer division */                3
x % y        /* remainder of integer division */   2
```

Example:

```
/* The program converts the velocity from miles per hour into
   kilometers per hour, where 1 mile = 1.60934 kilometer (km) */
#include <stdio.h>
/* Conversion constant */
#define MILES_INTO_KILOMETERS 1.60934f
int main ()
{
  float velocity_mph, velocity_kmph;
  printf ("Enter the velocity of the aircraft [miles/hour]: ");
  scanf ("%f", &velocity_mph);
  velocity_kmph = MILES_INTO_KILOMETERS * velocity_mph;
  printf ("The velocity of the aircraft = %.3f [km/h]\n",
          velocity_kmph);
  return 0;
}
```

```
int x, y;
x = 5;
y = 5;
printf ("++x = %d\n", ++x);
printf ("y++ = %d\n", y++);
printf ("x = %d\n", x);
printf ("y = %d\n", y);
```

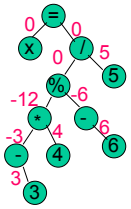
Results

++x = 6
y++ = 5
x = 6
y = 6

```
int x = -3 * 4 % -6 / 5;
```

Result:

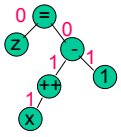
x = (-3) * 4 % (-6) / 5 = (((-3) * 4) % (-6)) / 5 = ((-12) % (-6)) / 5
= 0 / 5 = 0




```
int x, z;
x = 1;
z = x++ - 1;
```

Result:

z = 1 - 1 = 0
x = 2



Relational and Logic Operators

Relational operators

- `>` greater than
- `>=` greater than or equal
- `<` less than
- `<=` less than or equal
- `==` equal
- `!=` not equal

Logical operators

- `&&` AND
- `||` OR
- `!` NOT

Logical expression uses relational or logical operators and return 0 for false and 1 for true.

In C, true is any value other than 0. False is 0.

Truth table for the logical operators

x	y	x && y	x || y	!x
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

&& groups left-to-right: the first operand is evaluated, if it is equal to 0, the value of the expression is 0; otherwise the right operand is evaluated , and if it is equal to 0, the expression's value is 0, otherwise 1.

|| groups left-to-right: the first operand is evaluated, if it is unequal to 0, the value of the expression is 1; otherwise the right operand is evaluated, and if it is unequal to 0, the expression's value is 1, otherwise 0.

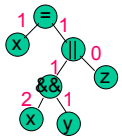
Precedence of the relational and logical operators

Highest	!
	> >= < <=
	== !=
	&&
Lowest	||

```
int x, y, z;
x = 2;
y = 1;
z = 0;
x = x && y || z;
```

Result:

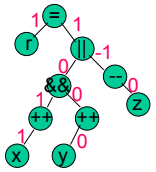
x = (x && y) || z = (2 && y) || z = (2 && 1) || z = 1 || z = 1
y = 1
z = 0




```
int x = 1, y = 0, z = 0;
r = x++ && y++ || --z;
```

Result:

r = (((x++) && (y++)) || --z) = ((1 && (y++)) || --z)
= ((1 && 0) || --z) = (0 || --z) = (0 || -1) = 1
x = 2
y = 1
z = -1
r = 1



Bitwise Operators

& AND
| OR
^ exclusive OR (XOR)
~ one's complement (NOT)
<< shift left
>> shift right

Assignment Operators

= x = y

Shorthand operators

variable operator= expression

can be rewritten as

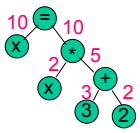
variable = variable operator expression

+=	x += y	x = x + y
-=	x -=y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

```
int x;
x = 2;
x *= 3 + 2;
```

Result:

x *= x * (3 +2)
x = 2 * 5
x = 10



Precedence and Associativity of Operators

Operators	Associativity
() [] . ->	left to right
! ~ + - ++ -- & * (*type*) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= >>= <<= &= ^= !=	right to left
,	left to right

Exercise: Compute the amount of soda (in liters) in a refrigerator that is field with two-liter bottles and 12-ounce cans. Use the conversion:

1 ounce [oz] = 29.586 milliliters [mL]

1. Define constant BOTTLE_VOLUME.

```
#define BOTTLE_VOLUME 2.0f
```

2. Define constant LITER_PER_OZ.

```
#define LITER_PER_OZ 0.029586f
```

3. Define constant CAN_VOLUME.

```
#define CAN_VOLUME 12 * LITER_PER_OZ
```

4. Declare variables in main function:

```
int   bottles,          // number of bottles
      cans;             // number of cans
float total;            // total value
```

5. Input number of bottles.

```
scanf ("%d", &bottles);
```

6. Input number of cans.

```
scanf ("%d", &cans);
```

7. Compute the total volume.

```
total = bottles * BOTLE_VOLUME + cans * CAN_VOLUME;
```

8. Print the results.

```
printf ("The total volume is %.3f [L]\n", total);
```

```
/* Compute the amount of soda [L] in a refrigerator that is
    field with two-liter bottles and 12-ounce cans.
   1 ounce [oz] = 29.586 milliliters [mL]
*/
#include <stdio.h>
/* Conversion constants */
#define BOTTLE_VOLUME 2.0f          /* 2-liter bottles */
#define LITER_PER_OZ 0.029586f      /* 1 oz = 29.586 mL */
#define CAN_VOLUME 12 * LITER_PER_OZ   /* 12-oz. cans */
```

```
int main ()
{
  int   bottles,          // number of bottles
        cans;             // number of cans
  float total;            // total value
  printf ("Enter the number of bottles: ");
  scanf ("%d", &bottles);
  printf ("Enter the number of cans: ");
  scanf ("%d", &cans);
  /* compute total volume */
  total = bottles * BOTLE_VOLUME + cans * CAN_VOLUME;
  /* print result */
  printf ("The total volume is %.3f [L]\n", total);
  return 0;
}
```