PROGRAMMING IN C

Digital Equipment Corporation
Educational Services

EY-1925E-HO-0002

Second Edition, August 1985

Programming in C
Course Outline

TEXT: "Learning to Program in C"

by Thomas Plum, Plum Hall Inc.

Monday:

Introduction to C
C Operands and Operators
C Control Flow - if, else if, switch, while, for

Tuesday:

C Control Flow - comma, do while, break, goto
C Functions
The C Preprocessor

Wednesday:

C Pointers and Arrays - to and including array arguments
The C Library - to and including sprintf and sscanf

Thursday:

Structures and Unions - to and including pointers to structures
The C Library - File I/O
C Pointers and Arrays - pointer arrays

Friday:

C Pointers and Arrays - cmd line args, pointers to functions
The C Library - system level I/O, heap allocation
Structures and Unions - arrays of structures, unions

TEXT ASSIGNMENTS

It is recommended that the following reading in the text "Learning to Program in C" be performed and the questions in that reading answered. Programming exercises are provided at the end of this workbook as a replacement for those in the text.

Monday:

Optional assignment - computer concepts (as needed)
"Learning to Program in C" - Chs. 1, 2.1

"Learning to Program in C"

Sections 2.6, 2.7, 2.8
Sections 3.4, 3.5, 3.6

Tuesday:

"Learning to Program in C"

Sections 3.8, 3.9, 3.11
Sections 5.4, 5.7

Wednesday:

"Learning to Program in C"

Section 3.12
Sections 7.2, 7.4

Thursday:

"Learning to Program in C"

Sections 7.3, 7.7
Sections 8.1, 8.2, 8.6

Friday: (or following the completion of the course)

"Learning to Program in C"
Ch. 6

"Programming in C Workbook"
Ch. 9

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

Table of ASCII characters: ASCII, decimal, octal, hexadecimal

nul	0	0000	0x00	+	43	0053	0x2b	V	86	0126	0x56
soh	1	0001	0x01	,	44	0054	0x2c	W	87	0127	0x57
stx	2	0002	0x02	-	45	0055	0x2d	X	88	0130	0x58
etx	3	0003	0x03	.	46	0056	0x2e	Y	89	0131	0x59
eot	4	0004	0x04	/	47	0057	0x2f	Z	90	0132	0x5a
enq	5	0005	0x05	0	48	0060	0x30	[	91	0133	0x5b
ack	6	0006	0x06	1	49	0061	0x31	\	92	0134	0x5c
bel	7	0007	0x07	2	50	0062	0x32	]	93	0135	0x5d
bs	8	0010	0x08	3	51	0063	0x33	^	94	0136	0x5e
ht	9	0011	0x09	4	52	0064	0x34	_	95	0137	0x5f
nl	10	0012	0x0a	5	53	0065	0x35	`	96	0140	0x60
vt	11	0013	0x0b	6	54	0066	0x36	a	97	0141	0x61
np	12	0014	0x0c	7	55	0067	0x37	b	98	0142	0x62
cr	13	0015	0x0d	8	56	0070	0x38	c	99	0143	0x63
so	14	0016	0x0e	9	57	0071	0x39	d	100	0144	0x64
si	15	0017	0x0f	:	58	0072	0x3a	e	101	0145	0x65
dle	16	0020	0x10	;	59	0073	0x3b	f	102	0146	0x66
dc1	17	0021	0x11	<	60	0074	0x3c	g	103	0147	0x67
dc2	18	0022	0x12	=	61	0075	0x3d	h	104	0150	0x68
dc3	19	0023	0x13	>	62	0076	0x3e	i	105	0151	0x69
dc4	20	0024	0x14	?	63	0077	0x3f	j	106	0152	0x6a
nak	21	0025	0x15	@	64	0100	0x40	k	107	0153	0x6b
syn	22	0026	0x16	A	65	0101	0x41	l	108	0154	0x6c
etb	23	0027	0x17	B	66	0102	0x42	m	109	0155	0x6d
can	24	0030	0x18	C	67	0103	0x43	n	110	0156	0x6e
em	25	0031	0x19	D	68	0104	0x44	o	111	0157	0x6f
sub	26	0032	0x1a	E	69	0105	0x45	p	112	0160	0x70
esc	27	0033	0x1b	F	70	0106	0x46	q	113	0161	0x71
fs	28	0034	0x1c	G	71	0107	0x47	r	114	0162	0x72
gs	29	0035	0x1d	H	72	0110	0x48	s	115	0163	0x73
rs	30	0036	0x1e	I	73	0111	0x49	t	116	0164	0x74
us	31	0037	0x1f	J	74	0112	0x4a	u	117	0165	0x75
sp	32	0040	0x20	K	75	0113	0x4b	v	118	0166	0x76
!	33	0041	0x21	L	76	0114	0x4c	w	119	0167	0x77
"	34	0042	0x22	M	77	0115	0x4d	x	120	0170	0x78
#	35	0043	0x23	N	78	0116	0x4e	y	121	0171	0x79
$	36	0044	0x24	O	79	0117	0x4f	z	122	0172	0x7a
%	37	0045	0x25	P	80	0120	0x50	{	123	0173	0x7b
&	38	0046	0x26	Q	81	0121	0x51	|	124	0174	0x7c
'	39	0047	0x27	R	82	0122	0x52	}	125	0175	0x7d
(	40	0050	0x28	S	83	0123	0x53	~	126	0176	0x7e
)	41	0051	0x29	T	84	0124	0x54	del	127	0177	0x7f
*	42	0052	0x2a	U	85	0125	0x55				

What is C?

- o A general purpose programming language.

- o Low level -- "portable assembler"

  Can access many computer objects directly.

  No storage allocation or heap mechanism.

  Weak typing rules.

  No I/O facilities.

  Cannot access composite objects as a whole.

- o Single thread control flow

- o Minimal run time environment

- o Modern machines:

  Byte addressing

  Address similar to integer

  Stack is cheap

Why C?

- o Efficient code generation on a variety of modern machines.

  UNIX consists of 12,000 lines of C and 800 lines of assembler language.

  C is on a variety of machines.

- o Portable

  Both the compiler and the library are easily ported to other machines.

- o Easy to learn and use.

A quick overview

- o Fundamental data types:
  characters, integers, and floatings.

- o Composite data types:
  pointers, arrays, structures, unions,
  functions.

- o Flow control:
  if, while, for, do, switch.

- o Recursion and reentrancy 'for free':
  automatic storage.

- o Scope of data:
  internal to a function or block,
  or
  global within a file,
  or
  global through all files.

- o Weakly typed:
  many data conversions
  permitted.

Identifiers, keywords, comments, constants

- Identifiers are strings of letters (including underscore) and digits, beginning with letter.
  Upper and lower case are distinct.

  Variables must be declared before use.

  First eight characters are significant (less for externals)

  Names that start with underscore should be reserved for system programs

- Keywords (reserved):

auto	double	if	static
break	else	int	struct
case	entry	long	switch
char	extern	register	typedef
continue	float	return	union
default	for	short	unsigned
do	goto	sizeof	while

- Comments consist of any text between /* and */ .

- Constants:

1	3.07	'x'	"help"

- Separators:
  `,` `;{}` `=` `(` `)` `:`

- Whitespace:
  blank or newline or horizontal tab,
  (Whitesmiths: any other non-printing character.)

- Suggestion: 80-character line limit.

Data types

Bytes	Type	Description
1	char	a single byte.
2	short	a short integer.
2 or 4	int	an integer (same size as pointer).
4	long	a long integer.
4	float	a single precision floating point number.
8	double	a double precision floating point number.

Compile and execute a simple C program

```
main()
        {
        printf("this is a C program.\n");
        }
```

To run using the UNIX operating system:

o EDITING

```
$ vi myprog.c
i<new text>ESC          insert new text (before cursor)
a<new text>ESC          append new text (after cursor)
h                       move cursor left one column
j                       move cursor down one line
k                       move cursor up one line
l                       move cursor right one column
x                       "gobble" character under cursor
:wq                     write output file and quit
```

o COMPILING & LINKING

```
$ cc -o myprog myprog.c
```

o RUNNING PROGRAM

```
$ myprog
```

To run using the VMS operating system:

o EDITING

```
$ EDIT MYPROG.C
*C                      to enter character mode
arrow keys              to move cursor on screen
delete key              to delete characters
PF2 key                 to get help on using keypad
<CTRL-Z>EX              to exit editor
```

o COMPILING

```
$ CC/LIST MYPROG        listing file MYPROG.LIS
                        object file  MYPROG.OBJ
$ PRINT MYPROG          to get hard copy of listing file
```

o LINKING

```
$ ASSIGN SYS$LIBRARY:CRTLIB.OLB  LNK$LIBRARY  !V1.0
$ ASSIGN SYS$LIBRARY:VAXCRTL.OLB LNK$LIBRARY  !V2.0

$ LINK MYPROG           image file MYPROG.EXE
```

o RUNNING PROGRAM

```
$ RUN MYPROG
```

A program to copy input to output

```
#include <stdio.h>

/* copy input to output
 */
main()
        {
        char c;

        c = getchar();
        while (c != EOF)
                {
                putchar(c);
                c = getchar();
                }
        exit (0);       /*exit(1) in VAX-11C*/
        }
```

At the top of each source file that performs I/O

```
#include <stdio.h>

        defines:        EOF     -1
                        getchar()
                        putchar()
                        ...
```

A program to count lines, words, chars

```
#include <stdio.h>

/* count lines, words, chars in input
 */
main()
        {
        int inword;       /* currently in a word? */
        short nc;         /* number of chars */
        short nl;         /* number of lines */
        short nw;         /* number of words */
        char c;           /* most recently read: char or EOF */

        inword = NO;
        nc = nl = nw = 0;

        while ((c = getchar()) != EOF)
                {
                ++nc;

                if (c == '\n')
                        ++nl;

                if (c  ==  ' ' || c == '\n' || c == '\t')
                        inword = NO;

                else if (inword == NO)
                        {
                        inword = YES;
                        ++nw;
                        }
                }

        printf("%d %d %d\n", nl, nw, nc);

        exit (0);         /*exit(1) in VAX-11C*/
        }
```

Sample formats for printf

Integer types

%d	integer (printed decimal, signed)
%x	hex integer
%o	octal integer
%03o	3-digit octal integer with 0-fill
%c	ASCII character

Strings of characters

%s	ASCII character string (null-terminated)
%.5s	A maximum of 5 ASCII characters from a string

Floating point

%8.2f	fixed-point, 8 wide, 2 places: -2345.78
%12.5e	e-format: -2.45678e-12

Buffering of input

On most systems, input from terminals is buffered one line at a time. This allows correction of typing mistakes on that one line.

Thus, the program does not see the input until the newline is typed. (Each operating system has its own method of over-riding this buffering to allow a program to see each character as typed.)

EXAMPLE

```
#include <stdio.h>

/* copy input to output
 */
main()
        {
        char c;

        c = getchar();
        while (c != EOF)
                {
                putchar(c);
                c = getchar();
                }
        }
```

THIS WILL HAPPEN

```
$ program
dog and cat
dog and cat
```

THIS WILL NOT HAPPEN

```
$ program
ddoogg  aanndd  ccaatt
```

Data types

o The fundamental C data types are:

```
char     : a byte.                        8 bits
short    : a 'short' integer             16 bits
long     : a 'long' integer              32 bits
float    : single-precision.             32 bits
double   : double-precision.             64 bits
int      : a pointer (address)           16 or 32 bits
```

o Signed/Unsigned

```
short             range of values -32768 -> +32767

unsigned short    range of values  0 -> 65535

unsigned long num;
unsigned int ab;
unsigned char x;
```

o Defined data types

```
#define  tiny  char

tiny x;
unsigned tiny y;


#define  ushort  unsigned short
#define  bool  int
```

Advantages to defined data types:

semantic distinctions

portable data (different defines - different hardware)

enhance readability

Constants

- Character constants:

  one char within single quotes

  ```
  'x', '\n', '\t', '\10'
  ```

- Integer constants:

  decimal:

  ```
  142, 17, 3421
  ```

  octal: a leading zero indicates an octal constant,

  ```
  042, 01, 0732
  ```

  hexadecimal: a leading 0x indicates hex constant,

  ```
  0x6f, 0x238, 0x17
  ```

  integer constants are represented in int (2/4 bytes)
  (on PDP - 200000L is long constant)

- Floating constants:

  ```
  1.23
   .23
  1.00
  17e-23
  ```

  floating constants are represented in double (8 bytes)

String constants

String constants: characters written within double quotes:

- "Hi there"
  ""

- Stored in memory as array of chars.

- By convention, the last character of a string is the null character, '\0'.

QUESTIONS:

- What is the size of these two strings?

  "hello" ____ "" ____

- What is the difference between "0" and '0' ?

Declarations

- Variables must be declared before use.

- Declarations specify a type, followed by a list of things having that type:

```
short a, b, c;

char q, r, s[100];
```

- The most readable format is an alphabetized list, one variable per line, with a comment:

```
short i;          /* buffer index counter */

int more;         /* is there more data? */

char tbuf[80];    /* terminal I/O buffer */

double x;         /* the unknown */
```

Arithmetic operators

+, -, *, /, % (remainder)

o % gives remainder;

5 % 2 = ___

4 % 2 = ___

8 % 3 = ___

```
a % b
```

gives the remainder of dividing a by b.

Not valid for double, float.

o * and / and %
have higher precedence than
+ and - .

o Unary -
has higher precedence than
any of the above.

o No guarantee of evaluation sequence:

```
funca() + funcb() * funcc()
```

could call a(), b(), or c() first.

o x + (y+z) = (x+y) + z = (x + z) +y
x * (y*z) = (x*y) * z = (x * z) *y

Compiler can rearrange across these parentheses. Parentheses are not adequate for specifying the order of calculation. Allows optimization:

(x + 1) + (y + 2) becomes x + y + 3

Value of: 23 + 4 * -5 + 1 - 6 % 5 ______

Relational operators

- The relational operators are:

  `> >= < <=`

  of lower precedence are:

  `== (is equal)   != (not equal)`

- produce 0 or 1 result (0=FALSE, 1=TRUE)

- Assignment operator is still lower precedence (and not relational):

  `= (assignment)`

- All are of lower precendence than arithmetic operators.

  `x + 1 < y + 2`

- Previously we wrote:

  ```
  while ((c = getchar()) != EOF)
  ```

  why not:

  ```
  while (c = getchar() != EOF)
  ```

  Always parenthesize embedded assignments.

QUESTIONS: What is the value of:

```
3 == 5   ____
1 >= 0   ____
4 > 4    ____
-1 < 0   ____
```

Expressions, operators, and operands

o Operators:

```
+ * - / % etc.
```

o Operands (the data being operated upon):

constant	1234	'x'	0xFF
variable	x	n	c

o Expression (examples):

operand operator operand	x + 1	n * 2
unary-operator operand	-40	&x
constant	40	'0'
variable	x	n

o Subexpression = an expression that is part of a larger expression

o x = (y + z) * 46

operator	+	*
operand	x	46
expression	y+z	x=(y+z)*46
subexpression	y+z	46

Logical operators

- "semi-Boolean":

```
        zero            means   NO (FALSE)
        non-zero        means   YES (TRUE)
```

- The logical operators are

```
        && (and)
        || (or) (pipe characters)
        !  (negation) (exclamation point)
```

- Precedence of && greater than that of ||.

  Both have lower precedence than relational operators:

```
        x < y && y < z
```

- Negation is monadic (unary): takes one operand.
  Converts YES (non-zero) into NO (zero),
  NO (zero) into YES (one).

- "Short-circuit": guarantee left-right sequence, stop evaluating when result is determined:

```
        if (j < MAX && ((c=getchar()) != '\n'))
```

- Sequence guarantees in C:

  full-expr   && || (more to come...)

- Truth Table

p	q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Type conversions

1) Widening of operands ("coercion"):

Register "int" sizes:

2-byte (16 bit):	PDP-11, 8080, Z80, ...
4-byte (32 bit):	68000, VAX, ...

Operands shorter than int are loaded into int-sized temporaries.

long remains long-size.

float operands are loaded into double temporaries.

2) Type balancing:

After coercion, if one operand is smaller than the other, it is further widened to equal size.

unsigned operand is "slightly wider" than signed.

2-byte machine

```
* double
  float
* unsigned long
* long
* unsigned int, unsigned short
* int, short
  unsigned char
  char
```

4-byte machine

```
* double
  float
* unsigned int, unsigned long
* int, long
  unsigned short
  short
  unsigned char
  char
```

* means a preferred type for this machine architecture.

Type conversion examples and cast

o For assignments, the right side value is converted to the type of the left side.

```
short i;
tiny t;
float f;

i = t;  /* t is sign extended */

t = i;  /* t gets low-order byte */

f = i;  /* i is converted to float */

i = f;  /* f is truncated */
```

o Types can be coerced by using a cast.

```
short i;
double d;

d = sqrt((double) i);
```

o From Standard library: sqrt()

Lvalue and rvalue

- Left side of an assignment is an object with a location in storage. This object is called an lvalue.

  An lvalue has:
  type, storage class, name, location.

  A simple case of an lvalue is an identifier.

- Right hand side of an assignment may be any object that has a value. This object is called an rvalue if it is not an lvalue.

  An rvalue has:
  type, name, value.

  A simple case of an rvalue is a constant.

- Making a value from an lvalue simply fetches its value from its location.

```
x = y;

x = 0;   /* legal */

0 = x;   /* illegal */
```

QUESTIONS: Which of the following are lvalues?

```
x              ____

2 * x + 3      ____

y = 0          ____
```

increment and decrement operators

o ++ adds one to a variable (lvalue).
-- subtracts one from a variable.

o ++ or -- used before the name (prefix):

```
b = a;
...++b...
```

value of expression is a + 1

++ or -- used after the name (postfix):

```
b = a;
...b++...
```

value of expression is a

o Fill in the missing parts:

```
short x, y;

x = 0;

y = x++;        /* y == _____ */

y = ++x;        /* y == _____ */
```

o Do not rely on exact time of ++, --.
It will be done by the next sequence-guarantee point.

```
s[i++] = t[j++];        /* GOOD */
s[i++] = t[i];          /* BAD  */
s[i]   = t[i++];        /* BAD  */
```

o Simpler rule: if you increment or decrement a variable, do not refer to it again in that statement.

Arrays

A composite data type

o The declaration:

```
short scores[30];
```

declares that scores is an array of 30 short integers.

o A declaration contains a "sandwich" of type + name.

```
char msg[80];
```

name is: msg

type is: char[80]

o Arrays are subscripted starting at zero
(just like birthdays and anniversaries!)

```
scores[0], scores[1], ... scores[29]
```

o To initialize the array scores to zero:

```
for (i=0 ; i<30 ; ++i)
        scores[i] = 0;
```

The for statement used above:

```
init:   i=0              - done once before loop

test:   i<30             - if YES, do body and step

body:   scores[i] = 0;

step:   ++i              - prepare for next test
```

o scores[i] is an lvalue.

Arrays of characters

- Strings are represented in C as arrays of characters.

- By convention, the null character, '\0', whose value is zero, is put on the end of all strings. This eliminates the need to store string lengths.

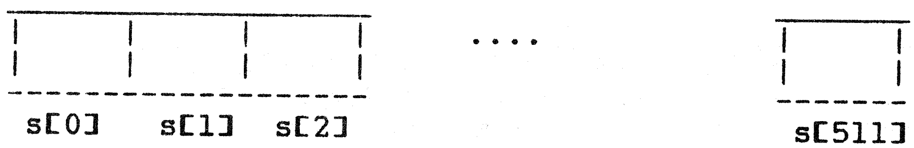
- For the string "april", the C complier generates:

  'a', 'p', 'r', 'i', 'l', '\0'

- Programs which build strings must append '\0' to those strings.

The array indexing formula

```
char s[512];                "type" of s is:  char[512]
```



Basic indexing formula:

```
address of jth element =
        address of zero-th element +
        j * (size of each element)
```

QUESTIONS:

Suppose &s[0] = 2000 (monadic "&" means address-of)

What is &s[101]? _______

Suppose a is declared:

```
short a[512];
```

and &a[0] = 4000

What is &a[101]? _______

String functions

```
   cpyastr (sl, s2, 3);              s2              sl
                                  |-----|         |-----|
                                  | 'D' |         |     |
                                  |-----|         |-----|
                                  | 'A' |         |     |
                                  |-----|         |-----|
                                  | '\0'|         |     |
                                  |-----|         |-----|
```

```
#include <stdio.h>

/*cpyastr - copy a string from s2 to sl
 */

cpyastr (sl, s2, n)

char sl[];                  /*destination string*/
char s2[];                  /*input string*/
unsigned int n;             /*number of characters to copy*/
        {
        short i;

        for (i=0 ; i<n ; ++i)

                sl[i] = s2[i];

        }
```

o Manipulation of strings must be done explicitly.
  In standard C, no statement will process an aggregate.

o Alternative to above:

```
        for (i=0 ; i<=n  &&  (sl[i] = s2[i]) != '\0' ; ++i)
                ;                /*null body*/
        sl[i] = '\0';
```

Bitwise operators

& bitwise and (bit-and)

```
&|   0    1
-|----------
0|   0    0
 |
1|   0    1
```

| bitwise or (bit-or)

```
(|)|   0    1
---|----------
  0|   0    1
   |
  1|   1    1
```

^ bitwise exclusive or

```
^|   0    1
-|----------
0|   0    1
 |
1|   1    0
```

o Example: char x = 0x16;

```
x & 0xF          0 0 0 1 0 1 1 0
                 0 0 0 0 1 1 1 1


x | 1            0 0 0 1 0 1 1 0
                 0 0 0 0 0 0 0 1
```

o What is the difference between && and & ?

2 && 1 is 1 2 & 1 is 0

QUESTIONS: What are the values of the following expressions?

0x3F & 0x67 _____

0x3F | 0x67 _____

Bitwise operators (continued)

~ unary ones complement (bit-not)

~0 is 0xFFFF or 0177777 16 bit

~1 is 0xFFFE or 0177776 machine

<< bitwise left shift (zero fill)

>> bitwise right shift (signed fill)

```
|--------------------------------|
|   |   |   |   |   |   |   |   |
|--------------------------------|
```

007 << 3 is 070

07 >> 1 is 03

QUESTION: What is the value of:

5 >> 2 && 07 & 010 __________

Bitwise "right rotate" function

Right rotate - bits shifted off to the right are to be rolled into the left

Example using 8 bits:

```
|--------------------------------|
|   |   |   |   |   |   |   |    |
|--------------------------------|
```

```
/* right rotate function
 */

short rightrot (n, b)

short n;        /* 16 bit word to rotate */
short b;        /* number of bits to rotate */
        {

        for ( ; b>0 ; --b)      /* repeat b times */
           {
           if (n & 01)
                n = (((unsigned)n >> 1) | 0100000);
           else
                n = ((unsigned)n >> 1);
            }
        return (n);
        }
```

o Always parenthesize bitwise expressions; bitwise precedence is tricky.

Assignment operators

o Expressions of the form;

```
x = x + 2;
```

can be written in the compressed form;

```
x += 2;
```

o
```
x -= 5;          /* subtract 5 from x */

x *= z;          /* multiply x times z */

x /= y - 1;      /* x gets divided by (y - 1) */
```

o operators:

```
+ - * / % << >> & ^ |
```

o Usefulness:

```
a[100 * i + j] = a[100 * i + j] + n;
```

can be written:

```
a[100 * i + j] += n;
```

a[100 * i + j] is evaluated only once!

o The form += is preferred to the form =+

consider x=-1;

o
```
#define tiny char
tiny x, y;

x += y;          /* not widened to int */
```

o Expression result is value from operation, converted to type of left-hand side.

Operator precedence

a + b * c

- o Which binds more tightly, the + or the *?
- o By historical agreement, the *
- o Fully parenthsized: a + (b * c)
- o Or, in words, "multiply b times c, then add a"

QUESTION:

Put into words (a + b) * c + d

Table of precedence

Precedence Level	Operators
15	`()  []  ->  .`
14	`!  ~  ++  --  -  (type)  *  &`
13	`*  /  %`
12	`+  -`
11	`>>  <<`
10	`<  <=  >  >=`
9	`==  !=`
8	`&`
7	`^`
6	`|`
5	`&&`
4	`||`
3	`?:`
2	`=  +=  -=  (etc., op=)`
1	`,`

QUESTIONS: Parenthesize to show the binding:

```
a == b && c != d

x = y = 3.14 * - d
```

The conditional operator (thenelse)

- The 'thenelse' "?:" operator provides a conditional expression in C.

- Ternary (triadic) operator: takes three operands.

-
```
if (q > 25)
        x = z;
else
        x = y;
```

  is rewritten:

```
        x =     q > 25   ?   z   :   y  ;
```

- Examples:

```
absx =    x < 0 ? -x : x;

minxy =   x < y ? x : y;
```

Statements and blocks

- An expression plus a semi-colon
  makes a statement.

```
++x;

c = getchar();

x + 1;  /* useless but a statement*/
```

- Curly braces { }
  denote a block (compound statement)

```
{
++j;
x = y;
}
```

- Null statement:
  one lonely semicolon

```
;
```

If (else) statement

o

```
if (expression)                      0     => FALSE
                                     non-0 => TRUE
          statement1
  +-                    -+
  | else                 |
  |       statement2     |
  +-                    -+
```

o Statement can be simple or compound

```
if (i<4)                        if (i<4)

   x[i] = i;                        {

   y[i] = i;                        x[i] = i;

                                    y[i] = i;

                                    }
```

o The else clause is associated
with the closest un-elsed if statement.

```
if (i != 0)                     if (i != 0)

    if (b[j] == 0)                  {

        b[i] = 1;                   if (b[j] == 0)

else                                    b[i] = 1;

    printf ("error\n");             }

                                else

                                    printf ("error\n");
```

o Always put braces around a nested if.

Else if statement

o

```
if (expr1)
        statement1
else if (expr2)
        statement2
else if (expr3)
        statement3
else
        default statement
```

o Last else clause is optional.

o only one statement is executed

QUESTION: What does this program print?

```
for (i = 1; i <= 8; ++i)
        {
        if (i < 4)
                printf("A");
        else if (i % 2 == 0)
                printf("B");
        else if (5 < i)
                printf("C");
        else
                printf("D");
        }
putchar('\n');
```

1 2 3 4 5 6 7 8

_ _ _ _ _ _ _ _

Switch statement

o Example:

```
switch (cmdchar)

{

case 'a':
        add(nl);
        break;

case 'd':
        delete(nl, n2);
        break;

case 'c':
        change(nl, n2);
        break;

default:
        remark("?", "");
        break;

}
```

o Execution starts at the case label whose constant is equal to the expression, and continues til the end of the switch, or the next break.

o Default is optional.

o You should always escape the switch after each case with a break.

o Prefer switch to elseif unless different conditions are tested or tests must be in sequence.

while and for statements

o while (expr)

```
while (expr)
        statement
```

o test is at top of the loop

o for (exprl; expr2; expr3)

```
for (exprl; expr2; expr3)
        statement
```

same as

```
expr 1;
while (expr2)
        {
        statement
        expr 3;
        }
```

o Use for rather than while when:

loop to be performed a known number of times

there is loop initialization

o "Endless" loop

```
#define   FOREVER   for (;;)

FOREVER
        {
        wait 1 sec
        print time
        }
```

Comma operator

o j=k, num=i++, ct=i++;

o Evaluated left to right.

o Complete list of sequence-guarantee

   full-expression {} && || ,

o Function to reverse a string in place
   ("SPOON" becomes "NOOPS")

```
int reverse (s)

char s[];
        {
        char t;
        short i, j;

        for (i=0, j = strlen (s) - 1 ; i<j ; ++i, --j)

                t = s[i], s[i] = s[j] , s[j] = t;
        }
```

```
  |-----|           |-----|          |------------------------------|
i |     |         s |  o----------->|     |    |    |    |    |     |
  |-----|           |-----|          |------------------------------|
j |     |
  |-----|

  |-----|
t |     |
  |-----|
```

Do while statement

- do

  ```
  do
          statement
  while (expr);
  ```

- test at bottom of the loop

- The do while statement is desirable only when the problem dictates that statement be executed at least once.

```
do
        {
        printf("Answer y or n: ");
        ans = getchar();
        while (getchar() != '\n')
                ;
        }
while (ans != 'y' && ans != 'n');
```

Break and continue

o Break causes an early exit from for, while, do, or switch.

```
while (expr)
        {
        statement
        if (expr)
                break;
        statement
        }
```

o N + 1/2-time loop

```
FOREVER
        {
        statement(s)
        if (expr)
                break;
        statement(s)
        }
```

Continue statement

o Continue causes the next iteration of the for, while, or do.

```
while (expr)
        {
        statement
        if (expr)
                continue;
        statement
        }
```

Goto statement

- Goto is never needed.

- ```
  goto label;
              ...
  label:
  ```

- ```
  for (...)
          for (...)
                  for (...)
                          ...
                          if (error)
                                  goto error;
  ...

  error:
          /*code to fix the error*/
  ```

What is a function?

```
pgm.c              sub1.c             sub2.c
__________         _________          _________
|main()            |sub1()            | sub2()
| ...              | ...              | ...
|                  |                  |
| sub1(arg)        |                  |
| ...              ---------          ---------
| sub2(arg)
|
----------

  compile            compile            compile
   only               only               only


                   linker


                 executable
                  program
```

VMS

```
$ CC PGM

$ CC SUB1

$ CC SUB2

$ LINK PGM,SUB1,SUB2
```

ULTRIX

```
% cc -c pgm.c

% cc -c sub1.c

% cc -c sub2.c

% ld pgm.o sub1.o sub2.o -lc
```

Basic function syntax

```
/* pow - return x to the power y
 */
double pow(x, y)

double x;        /* base */
long y;          /* exponent */
        {
        ----            body
        ----
        ----
        return ( ... );
        }
```

```
[return type]  name  ([parmlist])

[parmlist declarations];

        {
        body
        }
```

- o More power and complexity than a single statement.
- o independent building block
- o Take time to become familiar with existing libraries, to avoid re-inventing the wheel.
- o default return type is "int"

```
        reverse (s)

        char s[];
                {
                --
                }
```

Non-integer functions

o Function must be declared in the calling function

```
#include <stdio.h>

main()
        {
        short i, convert();
        long j;

        ---
        ---
        i = convert (j);
        ---
        }

short convert (num)

long num;
        {
        ---
        ---
        return ( ... );
        }
```

o If a function return is not declared integer is assumed.

o The return statement expression will be converted to the type of the function.

QUESTION: What is the data type of

convert ____________________

convert(j) ____________________

Argument passing

```
/* test integer power function
 */
main()
        {
        short i;
        short power();

        for (i = 0; i < 10; ++i)
                printf("%d %d\n", i, power(2, i));
        }
```

```
AUTOMATIC STORAGE                PARAMETER STACK
                 ______                  ______
i               |  0   |                |  2   |  <- 1st param
                |______|                |______|
                                        |  0   |
                                        |______|
```

copy of variable i ==> parameter stack

o Precise usages of the terms "argument" and "parameter":

In calling function:
actual parameter, actual argument,
dummy argument;

In called function:
formal parameter, formal argument,
parameter, real argument;

Argument passing (continued)

```
/* power - raise integer x to integer n-th power
 */

short power(x, n)

int x;                       /*base*/
short n;                     /*exponent*/
        {
        short p;

        for ( p=1 ; n > 0; --n)
                p *= x;

        return(p);
        }
```

```
AUTOMATIC STORAGE                 PARAMETER STACK
                 _______                    _______
p               |       |         x        |       |
                |_______|                  |_______|
                                  n        |       |
                                           |_______|
```

o Width of actual arguments in parameter stack:

Always widened to int, long, or double.

QUESTION: What does the stack look like as we enter power and as we leave power after the call:

power(12, 3)

```
        before                         after
       _______                        _______
      |       |                      |       |
      |_______|                      |_______|
      |       |                      |       |
      |_______|                      |_______|
```

Recursive functions

```
#include <stdio.h>

main()
        {
        long factorial();

        printf ("3 factorial is %d\n",
                                factorial (3));
        exit (0);
        }

/* factorial - return n!
 */
long factorial(n)

long n;         /* parameter, local storage */
        {
        if (n <= 1)
                return (1);
        else
                return (n * factorial(n - 1));
        }
```

- Variables declared within a function are local to that function and come into being with the dynamic invocation of the function. They disappear at function termination.

- The parameter stack comes into being with the dynamic invocation of the function. It disappears at function termination.

Initializing automatic scalars

- An "initializer" may be attached to the declaration of an automatic scalar (but not an array).

  Automatic arrays CAN be initialized in VAX-11C.

```
main()
        {
        char c = 'x';
        short i = 1;
        short j = i * 2;

        printf("%d %d %c\n", i, j, c);

        exit (0);
        }
```

QUESTION: What does this program print?

- The intialization is done by instructions that are executed each time the function is entered.

```
main()
        {
        short receipt();

        printf("First = %d\n", receipt());
        printf("Second = %d\n", receipt());

        exit (0);
        }

short receipt()
        {
        short number = 1;

        return (number++);
        }
```

QUESTION: What does this program print?

Storage class

o Picture of C program in computer memory

TEXT	contains the machine instructions for the program
DATA	contains variables which remain in FIXED locations -- "static" storage
STACK	contains automatic variables arguments, and function-call bookkeeping; changes as functions are called and returned

Static storage class

o Internal static:

Declared inside a function or block, and is known only inside that block (private memory). Stays put; is not in the stack.

Remembers values between function calls.

Initialization is done only once, when the program is loaded into the machine.

```
main()
        {
        short receipt();

        printf("First = %d\n", receipt());
        printf("Second = %d\n", receipt());

        exit (0);
        }

short receipt()
        {
        static short number = 1;

        return (number++);
        }
```

QUESTION: What is the output of this program?

Static storage class (continued)

- o External static:

  Data that is common (global) to several functions

  Declared OUTSIDE the body of any function

  Shared by all functions that follow in that source file

```
#include <stdio.h>


static short rnum = 0;          /* random number */


/* rand - return a random short integer
 */

short rand()
        {
        rnum = rnum * 12047 + 13911; /*period=8192*/
        return (rnum >> 1);
        }


/* srand - set random seed
 */

int srand(seed)

short seed;
        {
        rnum = seed;
        }
```

QUESTION: In which memory segment does rnum reside: TEXT, DATA, or STACK?

____

Initializing arrays

- o UNIX static arrays can be initialized
  automatic arrays cannot be initialized

- o VMS both static and automatic arrays can be initialized

- o static data is initialized into the program file at link time

```
static short digits[10] = {0,1,2,3,4,5,6,7,8,9};
static char msg[13] = "hello, world";
```

- o If the array bound is bigger than the number of initializers, the extra elements are initialized to zero.

  If the array bound is less than the number of initializers, a compiler error is generated.

  If no bound is given, it is taken to be the number of initializers.

QUESTION: What are the intial values?

```
static char st[5] = "std";

static char s[2] = "abc";

static short a[5] = {1, 2, 3};

static short b[] = {1, 3, 5, 7};

static char x[] = "abc";
```

External variables

o data that is common (global) to several functions

declared outside the body of any function

functions that wish to share access to external data use the extern keyword

```
#include <stdio.h>

short a = 0;    /*external data can be initialized*/

main()
        {
        extern short a;
        short i = 17;
        long l;
        ...
        l = subfn (i);
        ...
        if (a <= 25)
        ...
        }


long subfn (arg)

short arg;
        {
        extern short a;
        ...
        a = 32 / arg;
        ...
        return ( ... );
        }
```

Register storage class

o register int x;

data will be allocated in general purpose registers, instead of memory

reduce execution time since a memory access is not needed

for variables that are used often, eg. loop index

VAX-11C will ignore - it will choose which variables to place into registers

o Register storage class may be assigned to formal parameters in a function or to automatic variables.

```
int power(x, n)

register int x, n;
        {
        register int p;

        ...
        }
```

o Cannot take address of (&) register.

o For maximum portability, register should be used only with int and pointer variables. However, most compilers will do sensible things with char and short register declarations, also.

Scope rules

Internal data (local)

- o data declared inside a function is known only within that function

- o data can be declared inside any compound statement (formed with curly braces) (BLOCK)

  data declared in a block is known only within that block

```
main()
        {
        short i;
        ...
        if (i <= 25)
                {
                float i;
                ...
                i = 3e10;
                ...
                }
        ...
        }
```

External data (global)

- o data declared outside the body of a function in a source file is known to all functions that follow in that source file

- o static data declared outside the body of a function in a source file canNOT be made known to a function in any other source file

- o non-static data declared outside the body of a function in a source file is made known to any function in any source file with the extern keyword

- o Function names are external by default.

Initialization summary

o External or static storage is initialized only once, into the program file at link time. They stay put in fixed locations.

Scalars - initialized to constants or constant expressions:

```
static short lim = BUFSIZ + 1;
static char separator = '\n';
```

Arrays - initialized to lists of constants, padded with zeros:

```
static short ar[5] = {1, 2, 3, 4, 5};
static char buf[512] = {0};
static char s[] = "dog&cat";
```

o Auto storage (stack), and register storage (register) are initialized every time the function is entered.

Scalars - initialized to expressions:

```
short b = a + 1;
register int c = 326 / b;
```

Arrays - cannot be initialized in auto storage (UNIX)

- can be initialized in auto storage (VMS) with some irregularities

```
char buf [3] = {'d','o','g'};  /*legal*/
char msg[] = "dog";           /*illegal - compiler error*/
short tim[5] = {1, 2, 3};     /*NOT padded with zeroes*/
```

Empty brackets: three cases in C

1) As parameter to function, they are a synonym for address

```
long setstr (s)

char s[];        <-- receives the address of array s
```

2) With array initializer, they mean "take the size from the count of initializers."

```
static short x[ ] = {0123, 0456, 0777};
                           ^
                           |
                           3
```

3) With an external array, they mean "the bound will be specified by the actual data declaration"

```
extern short y[];
```

Passes of C compiler

- Preprocessor: expand macros, compile-time constants, #include files, and conditional compilation

- Parser: translate program into a logical tree-structure language

- Code generator: translate this tree into assembler code

- Assembler: produce relocatable object code from the symbolic assembler code

- Linker: link the relocatable object code together with other object files

Define

o #define ID token-string

the preprocessor replaces all occurences
of ID with 'token-string' after this defining instance.
'token-string' is scanned for previously defined ID's.

o Example

```
#define FAIL     1
#define EOF     -1
        ...

if (EOF == getchar())
        exit(FAIL);
```

becomes (in-line code)

```
if (-1 == getchar())
        exit(1);
```

o Dangerous example:

```
#define RABBIT  (RABBIT * RABBIT)
```

o Define can also be done on command line in UNIX

```
cc -DRT11=1 pgm.c
```

o Style rules:
- put # in column 1
- use uppercase names
- put all #defines before any data declarations

Define and macros

o Example (macro):

```
#define SQUARE(n)        n * n
#include <stdio.h>

main()
        {
        char x[100];
        short i;
        ...

        y = SQUARE(x[i]);

        ...
        }
```

becomes (in-line code):

```
        y = x[i] * x[i];
```

QUESTION: Write the in-line code for SQUARE(x+1).

Fix the definition.

Macros (continued)

- ```
  #define MAX(x, y)        (((x) < (y)) ? (y) : (x))

  #define MIN(x, y)        (((x) < (y)) ? (x) : (y))

  #define ABS(x)           (((x) < 0) ? -(x) : (x))
  ```

- "Generic" - accept any data type

- Efficiency - in-line code, no call and return

- Preprocessor lines are taken one at a time; they can be continued by placing a '\' at the end of the line.

  ```
  #define MIN(x, y)        (((x) < (y)) ? \
                           (x) : (y))
  ```

- Continuation possible for any C statement

  ```
  static char msg[] = "very long... \
  line";
  ```

  But if string fits on one line, prefer

  ```
  static char msg[] =
       "very long ... line";
  ```

- Don't put side-effects on arguments

  ```
  ABS(++n) ==> (((++n) < 0) ? -(++n) : (++n))
  ```

- SUGGESTION: Write function first.
  Make macro only when needed.
  (Function is less prone to programming errors.)

- Undef - To remove the latest definition:

  ```
  #undef id
  ```

  Rarely used in practical programming.
  ```

Include

- #include "filename"

  Causes this line to be replaced
  with the entire file 'filename'.

  For personal or project header files

  The UNIX compiler searches
  (1) the directory containing the C program,
  (2) directories specified in the compile command,
  (3) "standard places."

  The VAX-11C compiler searches
  (1) the current default directory
  (2) the directory containing the C program

- #include <filename>

  For system-wide header files

  The UNIX compiler searches
  (1) directories specified in the compile command,
  (2) "standard places."

  The VAX-11C compiler searches
  (1) SYS$LIBRARY - a standard directory

- Header files are usually named:

  file.h where file is any filename.

- Includes may be nested (discouraged).

Conditional compilation

o #if constant-expression

or

#ifdef ID

or

#ifndef ID

(any C or preprocessor statements)

#else

(any C or preprocessor statements)

#endif

o #if constant-expression
is true if constant-expression
evaluates to nonzero

#ifdef ID
is true if 'ID' has been defined.

#ifndef ID
is true if 'ID' has not been defined.

Conditional compilation examples

- o Environment dependencies (adapted from stdtyp.h)

```
#ifdef USHORT
#define ushort unsigned short
#else
#define ushort short
#endif
```

- o Simulating hardware on mainframe

```
#ifdef UNIX
static char buffer [48][80] = 0;
static char *bufp = &buffer;
#else
static char *bufp = 0x8000;
#endif
```

- o Safe way to nest #include

```
/* "sandwich" around header */
#ifndef SOMENAME
 ... text of header
#endif
```

- o "Tuning" for size

```
#if MAXTOKEN < 128
#define TOKEN char
#else
#define TOKEN ushort
#endif
```

- o Including TRYOUT main with function file

```
... (code for function)
#ifdef TRYOUT
main()
        {
        ... (code to test function)
        }
#endif
```

Line

o #line line-number ID

can be used to reset the line-number
and/or ID which is passed to the compiler.

```
/*test.c - 0 and o are mixed up in string name
 */

main()
        {
        char s0 [25];
        ...

        strcpy (so, "test string");
        ...
        }


% cc test.c
"test.c", line 23: so undefined


/*test.c - 0 and o are mixed up in string name
        #line used to change compiler error msg
 */

main()
        {
        char s0 [25];
        ...

#line 37 COPY
        strcpy (so, "test string");
        ...
        }


% cc test.c
COPY, line 37: so undefined
```

What is a pointer?

o A pointer holds the address of another variable.

o
```
short i, j;      /* i, j are short */
short *p;        /* p is a pointer to short */
        ...
i = 0;
p = &i;          /* p gets address of (&) i */
```

o
```
j = *p;          /* that which is pointed to by p*/
```

thus:

```
p = &i, j = *p;
```

is the same as:

```
j = i;
```

o
```
short *p;
```

is read "declare p as a short pointer"
declaration of the variable p

```
j = *p;
```

is read "set j to that which is pointed to by p"
assigning the variable j

*p has 2 meanings

o "Address-of" (&) can be applied only to lvalues, not rvalues.

QUESTION: Which of the following are ILLEGAL?

____ `p = &i;`

____ `p = &(i + 1);`

____ `p = &(i = 1);`

Declaring and using pointers

o

```
short *pi, *pj, t;  /*pi,pj are pointers to short*/
long *pl;           /*pl is pointer to long*/
double *pd;         /*pd is pointer to double*/
```

o pi, pl, pd are the pointers; they are lvalues. *pi, *pl, *pd are references to the objects pointed to; they are also lvalues.

variable	address	contents
	1100	9
pi	1300	1100
t	1350	14
		20
pj	1380	1350
pl	1400	1410
	1410	7
	1430	0.0
pd	1440	1430

Simple examples using pointers

```
short *pi, *pj, t;  /*pi,pj are pointers to short*/
long *pl;           /*pl is pointer to long*/
double *pd;         /*pd is pointer to double*/
```

```
1)      *pd += *pi;

2)      pi = &t;

3)      *pi = *pl;

4)      pj = pi;

5)      *pj /= 3;

6)      ++pj;

7)      (*pj)++;

8)      ++pl;
```

	1100	9
pi	1300	1100
t	1350	14
		20
pj	1380	1350
pl	1400	1410
	1410	7
	1430	0.0
pd	1440	1430

Pointers as function arguments: swap

o Call by value; C cannot directly
alter function arguments in caller.
To change the arguments in the caller,
pass pointers to the variables to be altered.

```
int badswap(i, j)

short i, j;
        {
        short t;

        t = i;
        i = j;
        j = t;
        }
```

This simply changes
the local i and j.

```
int swap(pi, pj)

short *pi, *pj;
        {
        short t;

        t = *pi;
        *pi = *pj;
        *pj = t;
        }
```

This is called:
swap(&x, &y);

```
   |-----|                 |-----|
 t |     |               t |     |
   |-----|                 |-----|

   |-----|                 |-----|
 i |     |              pi |     |
   |-----|                 |-----|

   |-----|                 |-----|
 j |     |              pj |     |
   |-----|                 |-----|


             |-----|                 |-----|
         800 |     |             900 |     |
             |-----|                 |-----|
```

Pointers as function arguments: scanf

```
int x;
short y;
float z;

nargs = scanf("%d%hd%f", &x, &y, &z);
```

Reads from standard input: 368 23 87.62
nargs tells how many successful assignments.
separators are whitespace: spaces, tabs, newlines

Input	Use this call:
FFFF7421	`scanf("%8x", &status)` 8-digit hex int number
ABC	`scanf("%c%c%c", &c1, &c2, &c3)` Three contiguous characters
A B C	`scanf("%s%s%s", s1, s2, s3)` Three separate characters (into strings)
hello	`scanf("%s", str)` One "word" (delimited by whitespace)
hello	`scanf("%80c", str)` At most 80 characters into str
499.95	`scanf("%3hd.%2hd", &dols, &cents)` Dollars and cents (2 shorts)

o Using scanf, there is no simple way to read one line of characters up to a newline.

Pointers and arrays

o All operations done by array subscripting can be done - usually faster - with pointers.

```
short q[100];
short *pq;
      ...

pq = &q[0]; /* pq gets address of the
                zeroeth element of q */
is equivalent to

pq = q;     /*q is equivalent to &q[0]*/
```

```
   700                    1200
   |-----|                |-----|
 q |  5  |             pq |     |
   |-----|                |-----|
   | 10  |
   |-----|                1300
   | ... |                |-----|
   |     |              i |     |
   |-----|                |-----|
   | 495 |
   |-----|
```

o If we then write: i = *pq;

What does i have in it?

o type of q = ________________

type of q[n] = ________________

o Declaration: short q[100] is read "array of 100 shorts"

Expression: q[n] is read "q sub n"

Pointers and arrays (con't.)

- o Whenever pointers are used in arithmetic expressions, integer constants and variables are scaled by the storage size of the pointer.

- o e.g.:

```
double *pd;
short *pi;
        ....

x = *(pi + 2);  /* the 2 is first multiplied by */
                /*      2 (the size of a short) */

d = *(pd - 7);  /* the 7 is first multiplied by */
                /*      8 (the size of a double) */
```

- o Example: short q[5];

```
                                  |----|
                            q[0]  |    |  1200
&q[3] is 1200 + 3*2               |----|
                            q[1]  |    |  1202
                                  |----|
q + 3 is 1200 + 3*2         q[2]  |    |  1204
                                  |----|
                            q[3]  |    |  1206
&q[3] is q + 3                    |----|
                            q[4]  |    |  1208
                                  |----|
q[3]  is *(q + 3)
```

- o Generally - q[n] is *(q + n)

Pointer and array examples: index.c

```
/* index - return index of first occurrence of char c
 *      in string s
 *      SUBSCRIPTED version
 */

#include <stdio.h>


int index(s, c)

char s[];        /*string to be searched*/
char c;          /*search character*/
        {
        short i = 0;

        while (s[i] != '\0' && s[i] != c)
                ++i;

        return (s[i] == c ? i : -1);
        }
```

```
/* index - return index of first occurrence of char c
 *      in string s
 *      POINTER version
 */


#include <stdio.h>


int index(s, c)

char *s;         /*string to be searched*/
char c;          /*search character*/
        {
        char *s0 = s;

        while (*s != '\0' && *s != c)
                ++s;

        return (*s == c ? s - s0 : -1);
        }
```

Array arguments: strncpy

- o When arrays are passed to functions, what C really passes is a pointer to the array.

```
/* strncpy - copy n characters from string  s2 to
        string  s1
 */

char *strncpy(s1, s2, n)

char *s1, *s2;
unsigned int n;
        {
        char *oldp = s1;

        while (n-- > 0)
                *s1++ = *s2++;

        return (oldp);
        }
```

- o strncpy will accept calls:

```
        (1)     strncpy(a1, a2, DIM);

                        or

        (2)     strncpy(&a1[0], &a2[0], DIM);
```

where a1 and a2 are declared as arrays:

```
        char a1[DIM], a2[DIM];
```

- o QUESTION: What is the type of

s1 ____________

strncpy ____________

Array arguments: a question

QUESTION: Assume the following machine state
just before calling strncpy(save, line, 4) :

```
VARIABLE        ADDRESS         STORAGE
                                 _____________________
line            800             | a  | b  | c  | \0 |
                                |____|____|____|____|
                                 _____________________
save            1800            | x  | y  | z  | w  |
                                |____|____|____|____|
```

What does the parameter stack look like when
strncpy(save, line, 4) is entered?

```
               _______
s1            |       |  <- 1st param
              |_______|
s2            |       |
              |_______|
n             |       |
              |_______|
```

What does the storage of save look like
when strncpy returns?

```
                                 _____________________
save            1800            |    |    |    |    |
                                |____|____|____|____|
```

Array arguments: read

```
#include <stdio.h>

/* read - read characters into an array
 */

unsigned int read (s, n)

char s[];                    /* where to store the bytes read */
unsigned int n;              /* max no. of bytes to read */
        {
        int i;
        char c;

        for (i = 0; i < n; )
                {
                c = getchar();

                if (c == EOF)
                        return (i);

                s[i++] = c;

                if (c == '\n')
                        return (i);
                }
        return (i);
        }
```

EXAMPLE:

```
                read (array, 10);


        s               ______
                       |      | <- 1st param
                       |______|
        n              |      |
                       |______|
```

```
char s[];         IS REWRITTEN BY C TO BECOME:

char *s;          (A POINTER TO CHARACTERS)
```

Array arguments: write

```
/* write - write the characters from an array
 */

unsigned int write (s, n)

char s[];                    /* location of bytes to write */
unsigned int n;              /* how many bytes to write */
        {
        unsigned int j;

        for (j = 0; j < n; ++j)

                putchar(s[j]);

        return (n);
        }
```

QUESTION: What does the parameter stack look like after the following function calls set up their arguments?

```
write ("abc", 3)
                                     _______
                                    |       | <- 1st param
                                    |_______|
                                    |       |
                                    |_______|


write ("0", 1)
                                     _______
                                    |       | <- 1st param
                                    |_______|
                                    |       |
                                    |_______|


                                     ____________________
"abc"             400               | a  | b  | c  | \0 |
                                    |____|____|____|____|
"0"               500               | 0  | \0 |
                                    |____|____|
```

Pointer arithmetic

- o Adding or subtracting pointers and integers will cause C to scale according to the storage size pointed to.
- o Pointers may be subtracted from each other (scaled).
- o Pointers to like types may be meaningfully compared with each other.
- o Pointers may be assigned or compared against 0. C guarantees that no data item will ever be at 0.

```
char *p;

if (p == NULL)
        return;
```

- o NULL: in stdio.h

QUESTION: If &s[0] == 1000, what address will receive 777?

```
short *ptr;
char s[20];

ptr = s;

*(ptr+3) = 777;
```

Multidimensional arrays

```
static short scores[7][9] =
{
        0, 1, 0, 2, 0, 0, 0, 0, 1,
        0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 2, 3, 1, 0, 0, 0, 0, 0,
        0, 0, 7, 0, 1, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0,
        1, 0, 1, 0, 1, 0, 0, 0, 0,
        2, 3, 1, 0, 0, 0, 0, 0, 0,
};
```

o Arrays are stored in rows, that is, right subscripts vary the fastest.

scores [2][3] = ________________

o sizeof (scores) = 7 x 9 x 2

sizeof (scores[0]) = 9 x 2

sizeof (scores[0][0]) = 2

type of scores[2] = short[9]

QUESTIONS: If &scores[0][0] == 1200, what is

&scores[1][0] ____________________

scores[1] ____________________

o Passing a multidimensional array to a function:

```
x = sumup(scores, 7);
```

where sumup is declared

```
short sumup(arr, nrows)
short arr[][9];
short nrows;
```

Pointer arrays

o short *aptr[10];
declares aptr to be an array of ten pointers to short.

o static char *cities[] =
{"NY", "PHILA", "BOS", "LA", NULL};

o cities 1000

```
 _______
| 1100  |
|_______|
| 1103  |
|_______|
| 1109  |
|_______|
| 1113  |
|_______|
|     0 |
|_______|
```

```
1100  | N   | Y   | \0  |
      |_____|_____|_____|

1103  | P   | H   | I   | L   | A   | \0  |
      |_____|_____|_____|_____|_____|_____

1109  | B   | O   | S   | \0  |
      |_____|_____|_____|_____|

1113  | L   | A   | \0  |
      |_____|_____|_____|
```

QUESTION: Write down the TYPE and VALUE of

cities[2][2] ____________ __________

&cities[2][2] ____________ __________

*cities[2] ____________ __________

Command line arguments to C

o When main() is called it is passed two arguments:

```
main (ac, av)

unsigned int ac;
char *av[];
```

o ac is the count of the number of arguments passed to the main.

o av is a pointer to a list of name pointers.

One types:

ULTRIX interface	VMS interface
cmd a1 a2	cmd = "syslogin:cmd.exe" cmd a1 a2

and the program sees the following variables:

```
                                  ________
ac              1400             |    3   |
                                 |________|
av              1404             | 1440   |
                                 |________|

                  ________                  ____  ____  ____  ____
av[0]   1440     | 1662  |        1662     | c  | m  | d  | \0 |
                 |_______|                 |____|____|____|____|
av[1]   1444     | 1666  |        1666     | a  | 1  | \0 |
                 |_______|                 |____|____|____|
av[2]   1448     | 1669  |        1669     | a  | 2  | \0 |
                 |_______|                 |____|____|____|
av[3]   1452     |     0 |
                 |_______|
```

Using arguments in echo.c

```
#include <stdio.h>


main(ac, av)

unsigned int ac;
char *av[];
        {
        short i;

        for (i = 1; i < ac; ++i)
          printf(i < ac-1 ? "%s " : "%s\n",
                                        av[i]);

        exit (0);
        }
```

o Example

```
        $ echo  ab  xyz  12345

ac              2200              ______
                                 |    4 |
                                 |______|
av              2204             | 2630 |
                                 |______|

                  ______              ____  ____  ____  _________
av[0]   2630     | 3750 |   3750 | e  | c  | h  | o | \0 |
                 |______|        |____|____|____|_________|
av[1]   2634     | 3755 |   3755 | a  | b  | \0 |
                 |______|        |____|____|____|____
av[2]   2638     | 3758 |   3758 | x  | y  | z  | \0 |
                 |______|        |____|____|____|______________
av[3]   2642     | 3762 |   3762 | 1  | 2  | 3  | 4  | 5  | \0 |
                 |______|        |_____________________________
av[4]   2644     |    0 |
                 |------|
```

Variable number of function arguments: cpystr

Taking the address of the argument list allows a pointer to walk this list:

```
char *cpystr(olddest, s)

char *olddest;
char *s;
        {
        char **ps = &s;
        register char *dest = olddest;
        register char *src;

        for (src = *ps; src != NULL; src = *++ps)
                while (*src != '\0')
                        *dest++ = *src++;
        *dest = '\0';
        return (olddest);
        }
```

Walking the pointer ps along the arguments allow user to call the function with variable number of arguments:

```
        cpystr(target, "ab", "c", "de", NULL);
```

```
                  |-----|         |------------------------
olddest      1300 | 2000|    2000 |  |  |  |  |  |  |  |
                  |-----|         |------------------------
             1304 | 3000|
                  |-----|         |-----------|
             1308 | 3050|    3000 | a | b | \0|
                  |-----|         |-----------|
             1312 | 3130|
                  |-----|         |-------|
             1316 |   0 |    3050 | c | \0|
                  |-----|         |-------|

                                  |-----------|
                             3130 | d | e | \0|
                  |-----|         |-----------|
               ps |     |
                  |-----|
             dest |     |
                  |-----|
              src |     |
                  |-----|
```

Pointers to functions

o Functions themselves cannot be directly manipulated but pointers to the functions can be.

o
```
void f(g)

void (*g)();                /* pointer to function */
        {
                ...
        k = (*g)(i);                /* call g(i) */
                ...
        }
```

o For example:

```
short fnl (arg)             /*the first function*/
short arg;
        {
        if (arg < 5) return (10);
        else return (11);
        }


short fn2 (arg)             /*the second function*/
short arg;
        {
        if (arg < 5) return (20);
        else return (21);
        }


int func (g, i)             /*the calling function*/
short (*g)();
short i;
        {
        printf ("%d\n", (*g)(i));
        }


main()
        {
        func (fnl, 7);  /* will print 11 */

        func (fn2, 3);  /* will print 20 */
```

Structure basics

o A structure is a group of variables, of varying type, which is placed together for ease of manipulation.

o Formal definition of structure variable
   (define the pattern)

```
struct task
        {
        char job [20];
        char *plan;
        short start;
        float length;
        };
```

o Declare structure variables from the pattern
   (actually allocate storage)

```
        struct task t;

        +----------------------------------------+
job     | | | | | | | | | | | | | | | | | | | | |
        +----------------------------------------+
plan    |       |
        +-------+
start   |   |
        +---+---+
length  |       |
        +-------+
```

o struct task ti, tj, tk;
declares three variables: ti, tj, tk.

o on VAX: sizeof (ti) = __________ (bytes).

Members of structures

o The member of a structure is used in expressions:

```
structurename.member
```

e.g.:

```
t.plan
  or
tk.length
```

o

element	type	offset in structure
t.job	char [20]	0
t.plan	char *	20
t.start	short	24
t.length	float	26

o EXAMPLE:

in task.h header file:

```
struct task
        {
        char job [20];
        char *plan;
        short start;
        float length;
        };
```

in prog.c program file:

```
#include "task.h"
#include <stdio.h>

main()
        {
        static struct task tl = {"Hawaii vacation",
                "car-plane", 1210, 8.45};

        printf ("%s  %s  %d  %8.2f",
          tl.job, tl.plan, tl.start, tl.length);
        }
```

Members and nesting

o structure.member is an lvalue

Examples:

```
        if (tj.start < ti.start)
                ...
        tk.length = 12.3;
                ...
        tl.plan = ptr;
```

o One structure may be nested inside another.

```
struct time
        {
        char hh;
        char mm;
        char ss;
        };

struct task
        {
        char job [20];
        char *plan;
        struct time start;
        float length;
        };

struct task t;
```

o We can now reference the components of each time:

```
t.start.hh
t.start.mm
t.start.ss
```

Defined types for structures

o Common usage -

#define the structure as a new variable

o EXAMPLE:

in task.h header file:

```
#define TASK struct task
TASK
        {
        char job [20];
        char *plan;
        short start;
        float length;
        };
```

in prog.c program file:

```
#include "task.h"
#include <stdio.h>

main()
        {
        static TASK t1 = {"Hawaii vacation",
                "car-plane", 1210, 8.45};

        printf ("%s  %s  %d  %8.2f",
          t1.job, t1.plan, t1.start, t1.length);
        }
```

Pointers to structures

o Only a few operators are allowed upon structures:

t.plan	member
&t	address-of
sizeof (t)	size of

Structures cannot be operated upon as a unit

```
  e.g.   ti = tj;        /*generally not work*/
                         /* but works in VAX-11C*/
```

o The declaration:

```
struct task *ptask;
```

declares ptask to point to a structure of type task.

```
        ptask = &t;
```

o To access members of the structure pointed to by ptask:

```
        ptask->job
                or
        ptask->plan
                or
        ptask->start.mm
```

o `ptask->length` is the same as `(*ptask).length`

o `t.plan` is the same as `(&t)->plan`

Pointers to structures (continued)

o Pointers to structures are often used to pass structures to functions.

o EXAMPLE:

```
/*function to add a task structure to a task table
 */

#include "task.h"

int install (ptask)

struct task *ptask;
        {
        ...
        ... ptask->job ...
        ... ptask->plan ...
        ... ptask->start.mm ...
        ...
        return (...);
        }
```

called from the main() by:

```
    num = install (&ti);
```

Formats for structure definitions

o
```
struct task
        {
        char *desc;
        long plan;
        }
        tskl, tsk2;
```
STRUCTURE FORM and ACTUAL STRUCTURES

o
```
struct task
        {
        char *desc;
        long plan;
        };
```
STRUCTURE FORM only

o
```
struct
        {
        char *desc;
        long plan;
        }
        tskl, tsk2;
```
ACTUAL STRUCTURES only

o
```
struct  task  tskl,tsk2;
```
ACTUAL STRUCTURES from a previously defined STRUCTURE FORM

Unions

- structure-like variables, i.e.
  objects of varying types and widths in one variable

- variable values overlay one another
  (not follow one another as in a structure)

- One use: two or more ways of looking at the same storage.

```
union
        {
        long l;
        char c[4];
        }  parts;
```

  l and c are two objects which
  can be held in the variable parts.

```
If  parts.l = 0x87654321

then    parts.c[0] = 0x21
        parts.c[1] = 0x43
        parts.c[2] = 0x65
        parts.c[3] = 0x87
```

- Another use: saving space in data storage by using the same space for mutually-exclusive values.

```
union payeeno
        {
        char ssno[12];
        char taxidno[15];
        };
```

- A union will be large enough to hold the largest member. Alignment will satisfy all uses.

- It is the programmers task to keep
  track of how the union was most
  recently used.

Typedef

- Typedef is a method of creating synonyms for types. This is part of the C language <u>not</u> part of the preprocessor.

- Instead of:
  ```
  #define bool int
  ```
  We could say:
  ```
  typedef int bool;
  ```

- ```
  typedef char *STRING;
          STRING s, t;
  ```
  ```

Arrays of structures

Consider the problem of looking up
a keyword in a predefined table and
mapping it into an integer "token" for efficiency.

```
the book is on this desk
1   2    3  4  5    6
```

o In header file (token.h)

```
struct keytab
        {
        char *word;
        int token;
        };
```

```
word              ______
                 |  o---|---->
                 |______|
token            |      |
                 |______|
```

o In program:

```
static struct keytab dtab[] =
        {
        "define", 1,
        "include", 2,
        "undef", 3,
        "line", 4,
        "ifdef", 5,
        "ifndef", 6,
        "endif", 7,
        "elseif", 8,
        };
```

Arrays of structures (continued)

```
/*function to lookup a keyword in a table
  of keytab structures and return the token
 */

#include "token.h"

int lookup (keyword, table, tablesize)

char *keyword;              /*keyword to lookup*/
struct keytab *table;       /*ptr to table of structures*/
short tablesize;            /*number of entries in table*/
        {

        for (  ; tablesize > 0 ; --tablesize, ++table)

                if (cmpstr(keyword, table->word))

                        return (table->token);

                return (0);     /*failure*/
                }
```

o Called from main() as:

```
  typ = lookup("line", dtab, 8);
```

```
         +-----+         +-------------------+
  keyword|     |         | l | i | n | e | \0|
         +-----+         +-------------------+

         +-----+ +-----+         +---------------------------+
   table |     | |   ------------>| d | e | f | i | n | e | \0|
         +-----+ |-----|         +---------------------------+
                 |  1  |
         +-----+ |-----|     +-------------------------------+
tablesize|     | |   -------->| i | n | c | l | u | d | e | \0|
         +-----+ |-----|     +-------------------------------+
                 |  2  |
                 |-----|
                 |     |
                    .
                    .
                 |     |
                 |-----|         +---------------------------+
                 |   ------------>| e | l | s | e | i | f | \0|
                 |-----|         +---------------------------+
                 |  8  |
                 +-----+
```

Bit fields

o Represent data as bit field instead of bytes

o Useful if storage is limited
Useful for defining status words, hardware interfaces, ...

o
```
#define bits unsigned int
struct flags
        {
        bits    alloc:1;
        bits    type:3;
        bits    ref:2;
        bits    sc:3;
        };

struct flags f;
```

Each individual field is n bits long
and may be referenced:

```
        f.alloc
                or
        f.sc
                etc.
```

```
|-------------------------------------|
|   ....        |      |    |      |  |
|-------------------------------------|
                   sc    ref   type  alloc
```

Bit fields (continued)

- To set on:

```
        f.alloc = 1;
```

  to turn off:

```
        f.type = 0;
```

  to test:

```
        if (f.sc == 1)
                ...
```

- Can't take address of (&) field.

- Unnamed fields are used for padding.

- Field of width 0 causes alignment on the next unsigned.

- Fields cannot overlap unsigned boundary; the field is aligned at the next unsigned.

- Do not depend on allocation order within word; it varies between machines. (some CPUs order bits left to right, not right to left as in VAX, PDP)

- Do not combine bit-field operations and mask-and-shift operations.

Linked lists

o A slight re-definition of our task structure will allow the creation of linked lists (chains) of tasks:

```
struct task
        {
        struct task *next;
        char job[20];
        char *plan;
        short start;
        float length;
        };


struct task t;

        +--------+
next    |  o----|------>
        +----------------------------------------+
job     | | | | | | | | | | | | | | | | | | | | |
        +----------------------------------------+
plan    |        |
        +--------+
start   |   |
        +---+----+
length  |        |
        +--------+
```

Linked list (continued)

```
struct task *tlist;          /* point to current head */

struct task *p;              /* point to new task */
```

o To add an element to a task linked list:

```
p = malloc(sizeof (struct task));

p->next = tlist, tlist = p;
```

o To delete an element from a task linked list:

```
p = tlist, tlist = p->next;

free(p);
```

```
         |-----|                 |-----|     |-----|     |-----|
   tlist |     |                 |  o------->|  o------->|  0  |
         |-----|                 |-----|     |-----|     |-----|
                                 |     |     |     |     |     |
                    |-----|      |     |     |     |     |     |
         |-----|    |     |      |     |     |     |     |     |
     p   |     |    |-----|      |     |     |     |     |     |
         |-----|    |     |      |     |     |     |     |     |
                    |     |      |-----|     |-----|     |-----|
                    |     |
                    |     |
                    |     |
                    |-----|
```

Standard input, standard output, and standard error

stdin Standard input

stdout Standard output

stderr Standard error file

- These three files are already opened for the main program.
- The default for all three files is the interactive terminal.

```
                         |-------|  stdout
                         |       |----------->   terminal
               stdin     |       |
 terminal------------->  |  cmd  |
                         |       |
                         |       |  stderr
                         |       |----------->   terminal
                         |-------|
```

- The stdin and stdout can be changed for individual commands

VMS

```
ASSIGN file1 SYS$INPUT
ASSIGN file2 SYS$OUTPUT
RUN CMD
```

UNIX

```
cmd <file1 >file2
```

```
                         |-------|  stdout
                         |       |----------->   file 2
               stdin     |       |
 file 1 ------------->   |  cmd  |
                         |       |
                         |       |  stderr
                         |       |----------->   terminal
                         |-------|
```

Character Input/Output: getchar, putchar

- Basic I/O facility:

  read a character at a time from the "standard" input
  write a character at a time to the "standard" output

- Get a character from the standard input:

  ```
  char getchar()
  ```

  getchar gets a character from stdin. (c >= 0)
  getchar returns EOF (-1) on end of file.

- Put a character to the standard output

  ```
  char putchar(c)
  ```

  putchar puts character c to stdout, returns c.

  (c must be >=0).
  putchar returns EOF on error.

- File copy:

  ```
  while ((c = getchar()) != EOF)
          putchar(c);
  ```

- getchar and putchar are typically implemented as macros, not functions.

Line Input/Output: gets, puts

gets - gets a text line from stdin.

puts - puts a text line to stdout.

o `char *gets(s)`

copies characters from stdin to the character string at s, until:
(a) newline
(b) EOF

A '\0' terminator is added.

The newline is deleted.

gets returns its argument.

o `int puts(s)`

copies from character string at s to stdout, appending a newline.

no value is returned.

o To copy input to output:

```
puts(gets(s));
```

Formatted output: printf

o printf(fmt, arg1, arg2, ...)

    fmt is a string specifying format
    arg1, ... are the variables to be output
        in that format

returns the number of characters written out

characters are output to the standard output

o EXAMPLES:

```
short i = 37;
static char s[] = "abc"
int j = 3;


printf ("%5s", s);       ==>   __abc

printf ("-5s", s);       ==>   abc__

printf ("-5.2s", s);     ==>   ab___

printf ("%5d", i);       ==>   ___37

printf ("%-5d", i);      ==>   37___

printf ("%*d", j, i);    ==>   _37
```

o If output is too wide for "output width", width is ignored.

Formatted input: scanf

o scanf(fmt, &arg1, &arg2 ... &argn)

fmt is a string specifying format
arg1, ... are the variables to be input
in that format

scanf returns the number of arguments
successfully assigned.
Characters are read from the standard input
The scan is terminated if the format
character does not match the input.
Codes for scanf are the same as for printf,
except that "hd", "ho", and "hx" read shorts.

o Input items are separated by whitespace,
which is ignored. The 'c' format is an exception;
the requested number of characters are always read
including whitespace characters.
(only EOF stops the scan)

o EXAMPLES:

```
int i;
short j;
char s1[20], s2[20];

scanf ("%d%hd", &i, &j);     <==   26  132

        will produce i==26, j==132

scanf ("%d%d", &i, &j);      <==   26  132

        will cause j value to overwrite adjacent i

scanf ("%2d%hd", &i, &j);    <==   356 241

        will produce i==35, j==6
        241 is still in terminal buffer

scanf ("%20c", s);           <==  test msg112$}*&^

        the next 20 characters go into s1

scanf ("%[abc]%[xyz]", s1, s2);  <== bacyxw

        will produce s1=="bac", s2=="yx"
```

I/O to and from strings: sprintf, sscanf

- o write args into a string according to fmt

```
char str [14];
static har month [10] = "November";
short day = 23;

sprintf (str, "%10s  %4d", month, day);
```

will produce str == "November 23"

- o read into args from string according to fmt

```
static char str [] = "Hammer 568";
char part [6];
long number;

sscanf (str, "%s%d", part, &number);
```

will produce part == "Hammer", number==568

File I/O

o A FILE is a structure specifying...

file descriptor: 0 STDIN, 1 STDOUT, 2 STDERR, ...
characters left in buffer
mode
next character in buffer
buffer

from stdio.h: #define FILE struct _iobuf

FILE pointers:	stdin,	stdout,	stderr
file descriptors:	0,	1,	2

o fopen - opens a file by name, in specified mode

```
FILE *fopen (fname, mode)
```

EXAMPLE:

```
FILE *fp;

fp = fopen ("data.file", "w");
```

mode == "w", open for seqential write

mode == "r" open for sequential read

mode == "a" append: open for writing at end

o fclose (fp)

Closes a file controlled by fp.

File I/O (continued)

- Character I/O

```
getc(fp)                /* macro */
putc(c, fp)             /* macro */
fgetc(fp)               /* function */
fputc(c, fp)            /* function */
```

- Line I/O

```
fgets(s, n, fp)
        read at most n-1 chars into s,
           including newline

fputs(s, fp)
        write s to file fp
```

- Formatted I/O

```
fscanf(fp, fmt, &arg1, ..., &argn)

fprintf(fp, fmt, arg1, ..., argn)
```

- Block I/O

```
fread (buf, size, num, fp)
        read num items of size each into buf

fwrite (buf, size, num, fp)
        write num items of size each from buf
```

```
  /*      Program to use C standard I/O to write a file
 *        containing one 1, two 2s, etc. up to 10
 */

#include <stdio.h>

main()
        {
        FILE *fptr;
        char string [10];
        register i, j;


        /*Create the file
         */

        if ((fptr = fopen ("FILE.DAT", "w")) == NULL)
                perror ("OPEN error"), exit (0);


        /*Place the correct numbers in the array string and
         *   write the array to the file
         */

        for (i=1 ; i<=10 ; i++)
                {

                for (j=0 ; j<i ; j++)
                        string [j] = i;

                if (fwrite (string, i, 1, fptr) == 0)

                        perror ("WRITE error"), exit (0);
                }

        /*Close the file
         */

        if (fclose (fptr) == EOF)

                perror ("CLOSE error"), exit (0);
        }
```

Error output: perror, fprintf

o perror ("file open error");

write string to stderr -and-

write system message to stderr that corresponds to the error code in the external int errno

o fprintf (stderr, "can't open file %s\n", fname);

write formatted output to stderr

System level I/O

- an alternative to standard I/O (fopen, fwrite, ...)

- direct calls to the ULTRIX operating system (emulated in VMS)

- creat will create a new file:

```
fd = creat (name, mode)
```

returns a file descriptor (positive integer)

mode specifies UNIX access permissions:

	owner	group	others
read	0400	040	04
write	0200	020	02
execute	0100	010	01

- open will open an already existing file:

```
fd = open (name, mode);
```

returns a file descriptor (positive integer)

```
mode    == 0 for read,
        == 1 for write,
        == 2 for read/write.
```

- close will close a file:

```
close(fd);
```

System I/O (continued)

o read and write:

```
read(fd, buf, size);
```

read size bytes into buf from fd

returns the number of bytes read
0 if end-of-file
-1 if error

```
write (fd, buf, size);
```

write size bytes from buf to fd

returns the number of bytes written
-1 if error

o EXAMPLE: To copy INPUT to OUTPUT

```
#include <stdio.h>

main()
        {
        char b[BUFSIZ];
        short i;
        int fdin, fdout;

        if ((fdin = open ("INPUT", 0)) == -1)
                perror ("open error"), exit (1);

        if ((fdout = creat ("OUTPUT", 0)) == -1)
                perror ("creat error"), exit (1);

        while ((i = read (fdin, b, BUFSIZ)) != 0)
                {
                if (i < 0)
                   perror("read error"), exit (1);

                else if (i != write(fdout, b, i))
                   perror("write error"), exit (1);
                }
        exit (0);
        }
```

From stdio.h: BUFSIZ (512 on most systems)

lseek

o lseek will position within an open file for read/write

```
lseek (fd, offset, origin);
```

offset is number of bytes from the origin

```
origin  == 0    byte offset from the beginning,
        == 1    byte offset from the current position,
        == 2    byte offset from the end.
```

returns the resulting offset location from the beginning

o lseek does not physically move the disk arm; it only specifies the byte position for the next I/O operation.

o EXAMPLE:

```
/*function to read randomly a block from a file
 */

#include <stdio.h>

int getblock(fd, buf, blkno)

int fd;                     /*file desc of open file*/
char *buf;                  /*address to read into*/
short blkno;                /*block number to read*/
        {

        lseek (fd, blkno * BUFSIZ, 0);

        /*return T or F value:
                F == 0 if end of file
                T >  0 for number of bytes read
         */
        return (read(fd, buf, BUFSIZ) == BUFSIZ);
        }
```

Heap allocation: malloc, free

o malloc - allocates space on the heap.

```
char *malloc(nbytes)
unsigned nbytes;
```

An element of size nbytes is
allocated, and its address is returned.

malloc() returns NULL pointer on failure.

o free - frees a previously allocated cell.

```
free(pcell)
char *pcell;
```

Free the space pointed to by pcell.

Be careful to free only those cells previously
malloc'ed!

C programming style

Data and variables

- Consistent and meaningful names

- Standard defined-types: ushort, tiny, ...

```
#define ushort unsigned short
#define tiny char
...
```

- Manifest constants: EOF, NULL, ...

C programming style

Operators

o No blank for primary and unary ops:

```
*p  p[]  s.m
```

o No blank for parens: (x + y)

o No blank for functions f(x)

o One blank for binary ops: x + y

o One blank for key words: if (...)

o Do not assume left-to-right evaluation:

```
a() + b() * c()
```

o Do not assume timing of side-effects within an expression:

```
a[i++] = b[j++];        OK

a[i++] = b[i];          BAD
```

o The only guarantees for sequence and side-effects are the sequence guarantees of C:

```
full-expr    &&    ||    ,    ?:
```

C programming style

Control structures

- o Braces above and below body

```
{
remark("bad value", code);
++nerrs;
}
```

- o One-tab uniform indents

- o 80-char line limit: no "wrapped lines"

- o "else-if" only when necessary; prefer "switch"

- o Avoid "goto" and "continue"

C programming style

Functions

o Layout:

```
#include <stdio.h>
#include <stdtyp.h>
#include "proj.h"
#define TOK short
TYPEX varx = NNN;       /* commented */

/* comment describing func
 */
TYPE func(a1, a2, a3)

TYPE1 a1;       /* describe */
TYPE2 a2;       /* describe */
TYPE3 a3;       /* describe */
        {
        extern TYPEX varx;
        <local declarations>

        <statements>
        }
```

o Build and use standard headers

o Source files no bigger than 500 lines; functions no bigger than 50 lines

o #includes, then #defines, then rest of file

o No initializations in header files; they should contain nothing but #define, typedef, structure declarations, and externs.

o Prefer static to external

o "Defensive programming": each source file responsible to avoid out-of bounds references. Professional code is not allowed to "bomb-out".

Common C bugs

1. General
   Uninitalized variables
   Off-by-one errors.
   Treating an array as though it were
   1-origin (instead of 0-origin).
   Unclosed comments.
   Forgetting semi-colons.
   Misplaced braces.

2. Types, Operators, and Expressions
   Using "char" instead of "int" for the
   returned value from getch.
   "Backslash" typed as "Slash"; e.g.,
   '/n' instead of '0.
   Declaring function arguments after the
   function brace, creating spurious
   local variables.
   Arithmetic overflow.
   Using relational operators on strings;
   e.g. s == "end" instead of
   strcmp(s, "end").
   Using "=" instead of "==".
   Multiple side-effects to the same memory
   in the same expression;
   e.g. sec = ++sec % 60;
   False assumptions about the time at which
   post-increment is done.
   Off-by-one errors in loops with increment.
   Precedence of bitwise logical operators.
   (Always parenthesize them.)
   Right-shifting negative numbers
   (Not equivalent to division).
   Assuming the order of evaluation of expressions.
   Forgetting null-terminator on strings.

Common C bugs (continued)

3. Control flow

Misplaced "else"
Missing "break" in "switch".
Loop with first or last case abnormal in some way.
Loop mistakenly never entered.

4. Functions and program structure

Wrong type of arguments
(relying on memory instead of manual).
Wrong order of arguments.
Omitting static on subfunction's abiding storage.
Assuming that static storage is re-initialized at
each re-entry.
Macro written without full parenthesization of
arguments
and result.

5. Pointers and arrays

Passing pointer instead of value -- or value
instead of pointer.
Confusing "char" with "char *".
Using pointers for strings without allocating
storage for the string.
Dangling pointer references -- references to
storage no longer used.
Confusing single quotes ('\n')
with double quotes ("\n").

MONDAY PROGRAMMING ASSIGNMENTS

In the following exercises, avoid explanding the scope of the exercises so as to involve sophisticated terminal input. For example in exercise 1, avoid generalizing so as to form the sum of numbers up to that input from the terminal.

1. Write a program to form the sum of the numbers from 1 to 25 inclusive. Print to the terminal the sum and the integer average of the numbers.

2. Write a program that reads 5 characters from the terminal and prints them back to the terminal in reverse order.

3. Write a program that will read characters from the terminal until newline and print back to the terminal a line of asterisks proportional in length to the binary value of each character typed. Apply a scaling factor, so that the largest ASCII character will still fit onto an 80 character line. This program functions as a simple plotter, treating the input line as an analog input signal.

4. Write a program that reads 2 numbers from the terminal and prints back to the terminal the larger. What happens if a letter is typed as input to your program?

5. Write a program which tells the size of a machine word in bits, i.e. tells how many bits exist in an int on the computer on which you are running.

6. Write a program which reads an line of input from the terminal and prints each word on a separate line. A word, for our purposes, is a sequence of non-whitespace characters. Along with each word, print its hash-sum (the sum of the characters in the word), once as a 4-digit hex number and once as a 5-digit octal number. Print the hex number with leading zeroes and the octal number with leading blanks. An empty line of input should produce no output.

TUESDAY PROGRAMMING ASSIGNMENTS

In the following exercises, avoid expanding the scope of the exercises so as to involve sophisticated terminal input.

√1. Write a function that compares 2 shorts (passed as arguments) and returns the larger. Test the function with a program.

√2. Write a macro DO which will duplicate the syntax of a FORTRAN DO loop, e.g.

DO i=3,11,2 written as DO(i,3,11,2)

meaning a loop from an initial value of i = 3 to a final value = 11 in increments of 2. The variable name, the limits of the loop and the increment are arguments. Test the macro with a program that prints to the terminal the values of the loop during each pass.

3. a. Write a function cmpstr (s1, s2) which returns a true value if strings s1 and s2 are equal, a false value otherwise. Compile cmpstr into an object file.

b. Write a program to test cmpstr. Compile it and link it with cmpstr.

4. Write a macro TOUPPER which will translate a lower case character into upper case using the conditional operator e.g. a ? x : y Test the macro with a program which reads characters from the terminal and prints back to the terminal the result of the TOUPPER macro on them.

5. The function nfrom (low,high) produces a random number between low and high inclusive. See page 5-24 of the text. Modify nfrom to generate a long value rather than a short one. Write a program that calls nfrom 10,000 times to generate random numbers from 1 to 6. Print to the terminal a summary showing how many 1s, 2s, etc. were generated.

6. Modify the program calling nfrom in the prior exercise to simulate 10,000 rolls of two six-sided dice. Print a summary showing each possible sum and how many times it occurred.

WEDNESDAY PROGRAMMING ASSIGNMENTS

1. Write a program that will populate a 50 element char array with the integers 1-50 using pointers, not subscripts. Print the array to the terminal on five lines

2. Write a program that reads your first name and age from the terminal using a single scanf and forms a character string using sprintf with your age at your next 3 birthdays. Print the character string back to the terminal.

3. Write the function rindex described in Exercise 7-1 on page 7-10 of the text "Learning to Program in C". Test with an appropriate program.

4. Modify the function cmpstr written in a previous exercise to use pointers rather than subscripts. Test with a program.

5. Multidimensional arrays will be needed for this exercise. See the appropriate pages of the text and this workbook for assistance. The program tokens.c converts its input into a table of tokens. See the supplied listing. A token is defined as a unique number assigned to a word. The first word from the input is assigned to token number 1, the next is assigned to token number 2, etc. When a word is found in the input that is identical to one encountered previously, it is given the same token number previously assigned. After reading all the input, the program prints out the table of token numbers and words. If a word is longer than 8 characters, only the first 8 are stored.

Write the functions required by tokens. Your solution consists of only one source file, with 2 functions:

```
short install (s)
char s[];
        /*Look for the word  s  in the table.  If found,
        return its token number.  If not found, insert
        s in the table and return a new token number.
        If no space left, exit (FAIL);*/

int dumptok()
        /*Print out the token table*/
```

Sample execution of tokens.c:

```
$ run tokens
ABC 123 ABC 1234567890
1 2 1 3

Token table:
  1 ABC
  2 123
  3 12345678
```

THURSDAY PROGRAMMING ASSIGNMENTS

1. Write a program to prompt for and read from the terminal the values for part name (maximum chars 10), part number (6 digits) and amount in stock. Obtain and write 4 such records into a disk file using a structure.

2. Write a program that uses an array of pointers to read from the terminal your first name, middle name and last name. Print on successive lines using the pointer array your last name, middle name and first name.

3. Write a program that will read the records from the part name file created above and print to the terminal a report showing the part name, part number and amount in stock of each and the total amount in stock of all parts.

4. Make the program revisions described in Exercise 8-1 on page 8-5 of the text "Learning to Program in C".

5. Write the program runtt described in Exercise 8-2 on page 8-7 of the text "Learning to Program in C". Ignore the last sentence of the exercise and print out the structure as in gettt.c.

FRIDAY PROGRAMMING ASSIGNMENTS

1. Write a program that reads its 2 arguments from the command line. If the strings are equal, print EQUAL and the string. If the strings are not equal, print NOT EQUAL. If less than or more than 2 arguments are supplied, print an appropriate error message.

2. Revise the program previously written to create a part number file to create and write to the file using system level I/O.

3. Write a program that will read the records using system level I/O from the part name file created above and print to the terminal a report showing the part name, part number and amount in stock of each and the total amount in stock of all parts.