RC 13972 (#66791) 9/18/89
Computer Science

Research Report

An Introduction to the C Programming Language

Charles C. Palmer

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

IBM
RESEARCH LIBRARY

'89 OCT -5 PM 2:44

NON-CIRCULATING
FILE COPY

NOTICE

This report will be distributed outside of IBM up to one year after the IBM publication date.

IBM Research Division
Almaden • T.J. Watson • Tokyo • Zurich

62

An Introduction to the C Programming Language

Charles C. Palmer
CPALMER at YKTVMZ

IBM Research
T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract: This course was developed at the Thomas J. Watson Research Center during the years 1984-1988 for the Research Professional Education Program. It was restructured into a two-day presentation for the Corporate Education Network as course #IYT0040I in September, 1988. This report consists of the class notes used in this course.

The course assumes that the student has some background in programming in a block-structured language such as PASCAL, PL/1, PL/S, REXX, etc.. The course covers all the features of the language and stresses portability, efficiency, and maintainability. Various C language processors are discussed and compared, including:

IBM C (DOS, VM, & MVS), Waterloo C (same), Microsoft C (DOS),
Turbo-C (DOS), DICE (IUO DOS, VM, Unix™), and Unix™ C compilers.

An Introduction to the C Programming Language

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

This course was developed at the Thomas J. Watson Research Center during the years 1984-1988 for the Research Professional Education Program. It was restructured into a two-day presentation for the Corporate Education Network. The twelve 50-minutes sessions will cover the following topics:

1 Day 1

- Introductions & paperwork
- Textbooks & other references
- C Philosophy
- Some available C language processors
- General C program structure
- C conventions
- C data types

2

- Character strings & arrays
- Examples: strcat() & strlen()
- Primitive I/O functions
- stdin and stdout
- Redirection
- Simple I/O
- printf()
- scanf()

3

- Usual Operators
- Unusual, Very Specialized Operators
- Expressions & Statements
- Automatic Conversions

4

- Conditional Statements (if-then-else)
- Relational Operators
- C Truth
- Logical Operators
- Conditional Operator

5

- *while* loop
- *for* loop and the comma operator
- *do while* loop
- *break* and *continue*
- *switch*
- *goto* (ugh!)

6

- Preprocessor directives
- Debugging Techniques

7

- Storage classes and scope
- First day discussion/chalk-talk

8 Day 2

- Writing your own functions
- Local variables
- Call-by-value .vs. call-by-name
- Basic pointer use

9

- More about arrays
- All about pointers to everything
- Multi-dimensional arrays

10

- Character strings and pointers
- String-oriented I/O
- Standard string functions
- Command-line arguments

11

- Fancy Declarations
- Pointers to Functions
- Structure Type Specifiers
- Union Type Specifiers

12

- Typedefs
- Enumerations
- Bit Fields
- The C Library
- File I/O
- Dynamic Memory Allocation
- Program Termination
- What is C++
- Where to Get Help

B#
Use C!

An Introduction to the C Programming Language

Class 1

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

★ Introductions & paperwork

★ Textbooks & other references

★ C Philosophy

★ Some available C language processors

★ General C program structure

★ C conventions

★ C data types

Suggested Prerequisites

- Some programming experience, preferably with procedural languages such as PASCAL, PL/1, ALGOL, etc.

- Access to VM or PC C language processors

- A sense of humor

Class Text

Harbison & Steele

- "A C Reference Manual"
- Best all-around reference
- Not a "how to" book
- Good examples and explanations
- Best available for portability questions
- Second edition includes proposed ANSI-C standard

Other Texts

- Kernighan & Ritchie (K & R)
  - "The C Programming Language"
  - The original "holy book" of C
  - Very concise; originally the standard definition
  - Rather vague on some points
  - New expanded second edition now available (some ANSI-C comments)
- Waite, et.al.
  - "C Primer Plus"
  - A gentler introduction
  - Lots of explanations & examples
  - Great pictures
  - Handy-dandy reference card
  - Second edition now very Microsoft oriented
- Piles of good & bad C books in the bookstores

Other References

- Unix(tm) or AIX manuals
- C language processor documentation
- Unix Review (magazine)
- Dr. Dobb's Journal (magazine)

C Philosophy

- Efficiency
- Access to hardware
- Availability
- Portability
- Flexibility

Some Available Language Processors

- All Unix (tm) systems
- Host Based
  - Waterloo C (VM & MVS)
  - PL.8 Front End (IUO)
  - IBM Host C Compiler (PO for VM & MVS)
  - AT&T C370 (Old, not recommended)
- PC Based
  - IBM Personal Computer C & C/2
  - Microsoft C (versions 3, 4, 5, ...)
  - Turbo-C
  - Waterloo C
- All of the above systems
  - DICE (IUO)

General C Program Structure

```
#include  "stdio.h"

main( )

{

    int answer;


    answer = 42;

    printf( " The answer is %d ! \n", answer  ) ;

}
```

Traditional C Style

- Variables are in lower case
  - int bologna, banana, bordeaux;
  - Variable name length significance depends on language processor
- Functions are in lower or mixed case
  - printf("Year End Report\n");
  - ShowMenu(menuname, color);
- Constants are in UPPER case
  - #define PI 3.1415
- Lots of white space
- Indentation
  - No standard indentation practice
  - Pick one that you like and stick with it

Indentation Style 1

(K & R)

```
/*  A program to print a Fahrenheit to Celsius Table
    for the values 0, 20, ..., 300 degrees F.  */

main()
{
    int    lower, upper, step;
    float  ftemp, ctemp;

    lower = 0;          /* lower limit of table */
    upper = 300;        /* upper limit */
    step  = 20;

    ftemp = lower;      /* start at the bottom */

    while (ftemp <= upper) {

            ctemp = (5.0 / 9.0) * (ftemp - 32.0);
            printf("%4.0f %6.1f\n", ftemp, ctemp);
            ftemp = ftemp + step;

    }

    exit (0);
}
```

Indentation Style 2

(CCP)

```
/*  A program to print a Fahrenheit to Celsius Table
    for the values 0, 20, ..., 300 degrees F.  */

main()

{
   int    lower, upper, step;
   float  ftemp, ctemp;

   lower = 0;          /* lower limit of table */
   upper = 300;        /* upper limit */
   step  = 20;

   ftemp = lower;      /* start at the bottom */

   while ( ftemp <= upper )
   {
       ctemp = ( 5.0 / 9.0 ) * ( ftemp - 32.0 );
       printf( "%4.0f %6.1f\n", ftemp, ctemp );
       ftemp = ftemp + step;

   }

   exit ( 0 );

}
```

Indentation Style 3

(Job Security)

```
/*  F2C  */

   main() {
     int l=0,u=300,s=20; float f;
     f=l;
     while(f<=u) {
       printf("%4.0f %6.1f\n",f,(5./9.)*(f-32.));
       f=f+s; }
     exit(0); }
```

This is NOT recommended style

C Data Types

int integer; can be further specified as short, long, or unsigned

- int orange;
- long int banana;
- short kiwi; /* the int is optional */

char character; a single byte that can hold at most one character (0-255)

- char letter;

float floating point number; usually 32 bits long

- float depth;

double double precision floating point number; usually 64 bits

- double width;

C Data Types

Sizes (in bytes)

Datatype	VM or MVS	PC or PS/2
char	1	1
int	4	2
short	2	2
long	4	4
float	4	4
double	8	8

Declaring Variables

Why do it at all?

- Some data is type-sensitive
- Promotes better programming practices
- Helps to prevent bugs (bozo .vs. b0zo)
- Simplifies the compiler
- Helps to prevent run-time surprises

Declaration Syntax

In general, the syntax is simply a data-type, followed by one or more variable names, separated by commas, followed by a semicolon.

```
int     score;
float   average;
char    grade;
```

The variables may also be initialized and several variables of the same type can be placed on one statement.

```
int       carnumber = 54, where, who = 0;
float     c = 2.997925e10;
char      bell = '\007';
double    height, weight, density;
```

Using Constants

- integers

Signed whole numbers	12, -12392, +32767
Hexadecimal numbers	0x0c, 0xFFFE, 0x434350
Octal numbers	014, 077, 01237

An additional *long* specification can be given to force long-sized constants:

528l, 0x0012L

- characters:

'a', '0', ' ', '!'
'\n', '\007', '\\'

- floating point numbers 1., .42, -2.17524, -4e16

Arrays

- An array is simply a series of elements of the same data type. They are declared like this: int a [3];
- Typically, they are stored in consecutive memory locations, each one large enough to hold a single variable of the specified data type.
- Arrays can have any number of dimensions.
- C arrays are always "zero origin", so the array above would have elements a[0], a[1], and a[2].
- Arrays of dimensions greater than one can be thought of as "arrays of arrays".
- Arrays can have the same data types and storage classes as ordinary values (scalars), with the same defaults.
- Arrays are stored in row-major order.
- No array bounds-checking is done at any time.

Arrays

How much to declare?

The length of an array, a constant expression, may be omitted as long as it is not needed to allocate storage :

1. The object being declared is a formal parameter of a function.
2. The declarator is accompanied by an initializer from which the size of the array can be deduced.
3. The declaration is not a defining occurrence, that is, it is an external declaration that refers to an object defined elsewhere.

Arrays

An exception to these cases is that the declaration of any *n*-dimensional array, where $n>1$, must include the sizes of the last $n-1$ dimensions so that the accessing algorithm can be established. For example, an array a[2][3] is thought of like this:

00	01	02
10	11	12

The array is actually stored like this:

00	01	02	10	11	12

So, knowing a[ ][3] is enough to define how to access the array as it is thought to be.

Arrays

Sample Declarations

```
int queue [ 5 ];          /* defining occurrence */

float weights [ ] = { 174.2, 115.0, 17.7 };
                            /* can deduce size */

functionname (pile)
int pile [ ]
{ ...                       /* a formal parameter
                               of a function */

extern double dip [ ];      /* external vector */

extern int two_d [ ] [ 50 ] ;
                            /* external, but last
                               dimension must be
                               supplied */
```

Example Program

```
/* Program to ask for & accept a char, and then
   print its decimal, hex, & octal equivalents.  */
main()
{
   char ch;

   /* prompt the user & accept the input */
   printf( "Hit a key, any key, then ENTER...\n" );
   scanf( "%c", &ch );
   /* show its decimal value */
   printf( "The code for the character %c is:\n",ch);
   printf( "%d (decimal), ", ch );
   printf( "%o ( octal ), ", ch );
   printf( "%x (  hex  ).\n ", ch );
   exit(0);
}

p122
Hit a key, any key, then ENTER...
A
The code for the character A is:
193 (decimal), 301 ( octal ), c1 (  hex  ).

p122
Hit a key, any key, then ENTER...
a
The code for the character a is:
129 (decimal), 201 ( octal ), 81 (  hex  ).
```

An Introduction to the C Programming Language

Class 2

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ

(CENET Course #IYT0040I)

T. J. Watson Research Center
Yorktown Heights, NY

Internal Use Only

Outline

- ★ Character strings & arrays
- ★ Examples: strcat() & strlen()
- ★ Primitive I/O functions
- ★ stdin and stdout
- ★ Redirection
- ★ Simple I/O
- ★ printf()
- ★ scanf()

Character Strings

- Defined as a run of consecutive memory locations the last of which is set to '\0' (NULL).
- One way to declare a character string is:

```
char  sing[ ] = "put the lime";
```

- This allocated the exact amount of memory needed.

p	u	t		t	h
e		l	i	m	e
\0					

- The NULL was automatically generated by the initialization
- The NULL makes keeping track of the string length unnecessary
- If you are building strings, YOU must supply the NULL

```
char  sing[ ] = {'p','u','t',' ','t','h',
          'e',' ','l','i','m','e','\0' };
```

strcat() and strlen()

These are standard library functions that are used with character strings. The programmer can assume that they will always be present in the run-time library.

- *strcat()* copies one string onto the end of another.
  - The copy starts at null of the target string, and continues up to, and including, the null of the string being copied.
  - It is the programmer's responsibility to insure there is enough room at the target. If there isn't, this function will happily overwrite whatever follows the target.
- *strlen()* returns as its value the number of characters in the given string, excluding the null.

Example: Building character strings with strcat()

```
/* Example program that appends one string onto another
   using the standard library function  strcat()   */

main()
{
   static char fee[]   = "in the coconut";
   static char fie[60] = "put the lime ";

   strcat( fie, fee );

   printf( " \"%s\" \n", fie );

   exit( 0 );
}
```

p24

```
 "put the lime in the coconut"
```

How strcat() works

fie [60] (before)							
p	u	t		t	h	e	
l	i	m	e		\0	\0	\0
\0	\0	\0	\0	\0	\0	\0	\0
\0	\0	\0	\0	\0	\0	\0	...

fee [ ] (before)							
i	n		t	h	e		c
o	c	o	n	u	t	\0	

fie [60] (after)							
p	u	t		t	h	e	
l	i	m	e		i	n	
t	h	e		c	o	c	o
n	u	t	\0	\0	\0	\0	...

fee [ ] (after)							
i	n		t	h	e		c
o	c	o	n	u	t	\0	

How strlen() works

```
/* character string initialization example */
main()
{
   char iswhat[40];
   static char quote[60] = {
               'J','a','m','b','a','l','a','y','a',
               ' ','i','s',' ' };

   printf( "Complete the sentence : '%s ...'\n",quote);
   scanf( "%s", iswhat );
   printf( "Your answer of %d characters ",
              strlen( iswhat ));
   printf( "makes the sentence read:\n ");
   printf( "%s%s\n", quote, iswhat );

   exit(0);
}
```

```
p26
Complete the sentence : 'Jambalaya is  ...'
great!
Your answer of 6 characters makes the sentence read:
 Jambalaya is great!
```

Single Character I/O

- *getchar()* : Gets one character from stdin and returns that character as the function's value.
- *putchar(c)*: takes one character from the executing program and sends it to stdout.

```
#include <stdio.h>

/* Program to echo one char from stdin to stdout */

main()
{
   char ch;

   ch = getchar();
   putchar( ch );

}

p27
z[enter]
z
```

Slick Echo Program

```
#include <stdio.h>

/* Program to echo one char from stdin to stdout */
main()
{
   putchar( getchar() );
}

p28
![enter]
!
```

Buffered .vs. Unbuffered I/O

When the echo program is run on some systems, the character is not accepted until you hit the enter key and only then is it written back to the screen. These are "buffered" systems (i.e. VM & MVS).

- It can be more efficient to send complete packages rather than one character at a time.
- The buffering allows the user to correct typos before they are sent to the program.

If the echo program is run on a system that uses unbuffered I/O, the character entered is immediately passed to the program which will immediately write it back to the screen (i.e. UNIX™ in raw mode).

- Interactive programs expecting many short input strings work best in an unbuffered system.

Some C library packages provide both buffered as well as unbuffered I/O functions.

So What Is stdio.h Anyway?

It is a file that is supplied with the language processor that contains information about input and output:

- Useful #define's like NULL and EOF
- Many I/O "functions" are actually macros defined here
- Usually some structure and type definitions
- YOU DON'T ALWAYS NEED IT - see your documentation

Echoing Lots of Chars #1

```
#include <stdio.h>
#define    QUIT  '#'

/* copy lots of chars from stdin to stdout */
main()
{
   char ch;
   int count=0;

   ch = getchar();
   while (ch != QUIT )
   {
       count = count + 1;
       putchar(ch);
       ch = getchar();
   }
   printf("\n%d chars were read.\n",count);
}

p211
abcdef[enter]
abcdef
BYEBYE!#[enter]
BYEBYE!
14 chars were read.

p211
Zz#abcdefabcdefabcdefabcdefabcdefabcdefabcdef[enter]
Zz
2 chars were read.
```

Echoing Lots of Chars #2

```
#include <stdio.h>
#define    QUIT  '#'

/* copy lots of chars from stdin to stdout */
main()
{
   char ch;
   int count=0;

   /* Make use of what getchar() returns */
   while( (ch = getchar()) != QUIT )
   {
       count = count + 1;
       putchar(ch);
   }
   printf("\n%d chars were read.\n",count);
}
```

```
p212
This is a test of the early warning ...#
This is a test of the early warning ...
39 chars were read.
```

The EOF Character

C processors #define (in stdio.h) a constant named EOF that is set to a character that the system input routines will return if they reach end-of-file. Most of the time, it happens to be set to -1.

- On PC's running PCDOS, the way you enter EOF is with control-z.
- On PC's running UNIX™, control-d sends EOF to your program.
- On VM, using CW, you enter EOF by setting a pfkey to a special EBCDIC character (0x03).
- Using IBM Host C (VM & MVS) as well as AT&T C370 (VM), you must enter '/*', starting in column 1, on a line by itself to signal EOF (JCL lives!).

Echoing Lots of Chars #3

```
#include <stdio.h>

/* copy lots of chars from stdin to stdout */
main()
{
   char ch;
   int count=0;

   /* Make use of what getchar() returns */
   while( (ch = getchar()) != EOF )
   {
       count = count + 1;
       putchar(ch);
   }
   printf("\n%d chars were read.\n",count);
}
```

p214

```
"Go ahead, make my day", said the burly operator to ...^Z[enter]
"Go ahead, make my day", said the burly operator to ...
54 chars were read.
```

STDIN and STDOUT

- Defaults: stdin == input device, stdout == output device
- scanf() always reads from stdin, printf always writes to stdout
- Most systems allow redirection of stdin and stdout
- Using printf() and scanf(), and redirection, you can accomplish limited file I/O
- Great for testing
- All UNIX™ systems, PCDOS, and VM & MVS(IBMC & CW only)

Redirection

Redirection of stdin and stdout provide great flexibility in how you can write, test, and use programs.

To redirect stdin, follow the program's name with a '<' followed by where the input is to come from (i.e. a file or device).

To redirect stdout, follow the program's name with a '>' followed by where the output is to go (i.e. a file or device). Some systems (PCDOS and UNIX™) provide a '>>' operator that will append the redirected stdout to the end of a file if it already exists.

Another stream, stderr, is where error messages are sent. It also defaults to the output device. Using this stream can keep error and other non-data messages from getting mixed up in the stdout stream. The stderr stream can only be redirected under UNIX™ systems.

Redirection Rules

- A redirection operator connects an executable program (command) with a file or device. It can not be used to connect one file to another or one program to another.
- Input can not be taken from more than one file or device, nor can output be directed to more than one file or device using these operators.
- Whether or not spaces are required around the operators is, unfortunately, operating system dependent.

A Character Counting Program

```
#include <stdio.h>

main()
{
   int count = 0;

   while( (getchar()) != EOF )
       count = count + 1;
   printf( "The input consists of %d characters.\n",
           count);
   exit(0);
}
```

```
p218
abcdefg[enter]
hijklmnop[enter]
qrstuv[enter]
wxyz^Z[enter]
The input consists of 29 characters.

p218 < p218.c
The input consists of 183 characters.

p218 < p218.c > whatever.fil

type whatever.fil
The input consists of 183 characters.
```

An Uppercasing Filter

A filter is a program that accepts stdin, changes it in some way, and the sends it to stdout. Examples are *sort* & *more*, or a program that would convert stdin to uppercase:

```
#include <stdio.h>
main()
{
   char ch;
   while( (ch = getchar()) != EOF )
       putchar( toupper( ch ) );

   exit(0);
}

p219 < p219.c
#INCLUDE <STDIO.H>
MAIN()
{
   CHAR CH;
   WHILE( (CH = GETCHAR()) != EOF )
       PUTCHAR( TOUPPER( CH ) );

   EXIT(0);
}
```

printf() & scanf()

- "standard" functions
- Both expect a control string and an optional list of arguments.
- Various conversion specifications are available
- There MUST be the right number of conversion specifications for the arg list.

```
/* RIGHT! */
   printf ("The sum of %d and %d is %d \n", a, b, a+b);


/* WRONG! */
   printf ("The sum of %d and %d is %d \n", a, b      );
```

printf() conversion modifiers

A conversion specification begins with the '%' and is followed by the following elements in the following order:

1. optional flag characters
   - '–' : left-justify the field
   - '0' : use '0' as the pad character
   - '+' : always produce a sign '+' or '–'
   - ' ' : always produce either a leading '–' sign or a space
   - '#' : try to identify the type of the output
2. an optional minimum field width expressed as an integer
3. an optional precision specification, given as a '.' followed by the number of digits to appear after the decimal
4. an optional long size specification specified as 'l' (lowercase L) to indicate that the argument is long.
5. a required conversion operation, one of the characters "cdeEfgGousxX%".

printf() conversion modifiers

%d signed decimal conversion from type int or long

%c the argument is printed as a single character

%s the argument is expected to be the address of an area which is printed as a character string

%e,%E

signed decimal floating-point conversion is performed. The output is in the form *[–]d.ddddde+dd* or *[–]d.dddddE+dd*. One digit appears before the decimal point, the precision specifies the number of digits to follow the decimal point.

%f signed decimal floating-point conversion is performed. The output is in the form *[–]ddd.dddd*. The precision specifies the number of digits to follow the decimal point.

printf() conversion modifiers

%g,%G

signed decimal floating-point conversion is performed. If the value to be printed is not too large or small, then *f* format is used; otherwise, *e* or *E* is used. The output should be in whichever form takes the least amount of room.

%u unsigned decimal conversion from type unsigned int or unsigned long.

%o unsigned octal conversion from type unsigned int or unsigned long.

%x,%X

unsigned hexadecimal conversion from type unsigned int or unsigned long. The *x* form uses *0123456789abcdef* as digits, whereas the *X* form uses *0123456789ABCDEF*

% A single percent sign is printed.

printf() formatting examples

The following pages (from the text) were generated using various printf conversion modifiers. For example, the third line of the first page was generated with

```
printf ("%6s|%#5d|%#5o|%#5x|%#7.2f|%#10.2e|%#10.4g|\n",
          "%#",  45,  45, 45, 12.678, 12.678, 12.678);
```

```
----------+---------------------------------------------
Value   -> 45     45     45    12.678    12.678     12.678
Operation-> 5d    5o     5x     7.2f      10.2e      10.4g
Flags:
----------+---------------------------------------------
        % |   45|   55|   2d|  12.68| 1.27e+01|     12.68|
       %0 |00045|00055|0002d|0012.68|001.27e+01|0000012.68|
       %# |   45|  055| 0x2d|  12.68| 1.27e+01|     12.68|
      %#0 |00045|00055|0x02d|0012.68|001.27e+01|0000012.68|
       %  |   45|   55|   2d|  12.68| 1.27e+01|     12.68|
      % 0 | 0045|00055|0002d| 012.68| 01.27e+01| 000012.68|
      % # |   45|  055| 0x2d|  12.68| 1.27e+01|     12.68|
     % #0 | 0045|00055|0x02d| 012.68| 01.27e+01| 000012.68|
       %+ |  +45|   55|   2d| +12.68|+1.27e+01|    +12.68|
      %+0 |+0045|00055|0002d|+012.68|+01.27e+01|+000012.68|
      %+# |  +45|  055| 0x2d| +12.68|+1.27e+01|    +12.68|
     %+#0 |+0045|00055|0x02d|+012.68|+01.27e+01|+000012.68|
      %+  |  +45|   55|   2d| +12.68|+1.27e+01|    +12.68|
     %+ 0 |+0045|00055|0002d|+012.68|+01.27e+01|+000012.68|
     %+ # |  +45|  055| 0x2d| +12.68|+1.27e+01|    +12.68|
    %+ #0 |+0045|00055|0x02d|+012.68|+01.27e+01|+000012.68|
       %- |45   |55   |2d   |12.68  |1.27e+01  |12.68     |
      %-0 |45   |55   |2d   |12.68  |1.27e+01  |12.68     |
      %-# |45   |055  |0x2d |12.68  |1.27e+01  |12.68     |
     %-#0 |45   |055  |0x2d |12.68  |1.27e+01  |12.68     |
      %-  | 45  |55   |2d   | 12.68 | 1.27e+01 | 12.68    |
     %- 0 | 45  |55   |2d   | 12.68 | 1.27e+01 | 12.68    |
     %- # | 45  |055  |0x2d | 12.68 | 1.27e+01 | 12.68    |
    %- #0 | 45  |055  |0x2d | 12.68 | 1.27e+01 | 12.68    |
      %-+ |+45  |55   |2d   |+12.68 |+1.27e+01 |+12.68    |
     %-+0 |+45  |55   |2d   |+12.68 |+1.27e+01 |+12.68    |
     %-+# |+45  |055  |0x2d |+12.68 |+1.27e+01 |+12.68    |
    %-+#0 |+45  |055  |0x2d |+12.68 |+1.27e+01 |+12.68    |
     %-+  |+45  |55   |2d   |+12.68 |+1.27e+01 |+12.68    |
    %-+ 0 |+45  |55   |2d   |+12.68 |+1.27e+01 |+12.68    |
    %-+ # |+45  |055  |0x2d |+12.68 |+1.27e+01 |+12.68    |
   %-+ #0 |+45  |055  |0x2d |+12.68 |+1.27e+01 |+12.68    |
----------+---------------------------------------------
```

```
----------+----------------------------------------------
Value     ->"zap"  '*'   none -3.4567  -3.4567    -3.4567
Operation-> 5s     5c    5%    7.2f     10.2e      10.4g
Flags:
----------+----------------------------------------------
         %|  zap|    *|    %|  -3.46| -3.46e+00|    -3.457|
        %0|00zap|0000*|0000%|-003.46|-03.46e+00|-00003.457|
        %#|  zap|    *|    %|  -3.46| -3.46e+00|    -3.457|
       %#0|00zap|0000*|0000%|-003.46|-03.46e+00|-00003.457|
        % |  zap|    *|    %|  -3.46| -3.46e+00|    -3.457|
       % 0|00zap|0000*|0000%|-003.46|-03.46e+00|-00003.457|
       % #|  zap|    *|    %|  -3.46| -3.46e+00|    -3.457|
      % #0|00zap|0000*|0000%|-003.46|-03.46e+00|-00003.457|
        %+|  zap|    *|    %|  -3.46| -3.46e+00|    -3.457|
       %+0|00zap|0000*|0000%|-003.46|-03.46e+00|-00003.457|
       %+#|  zap|    *|    %|  -3.46| -3.46e+00|    -3.457|
      %+#0|00zap|0000*|0000%|-003.46|-03.46e+00|-00003.457|
       %+ |  zap|    *|    %|  -3.46| -3.46e+00|    -3.457|
      %+ 0|00zap|0000*|0000%|-003.46|-03.46e+00|-00003.457|
      %+ #|  zap|    *|    %|  -3.46| -3.46e+00|    -3.457|
     %+ #0|00zap|0000*|0000%|-003.46|-03.46e+00|-00003.457|
        %-|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
       %-0|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
       %-#|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
      %-#0|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
       %- |zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
      %- 0|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
      %- #|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
     %- #0|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
       %-+|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
      %-+0|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
      %-+#|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
     %-+#0|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
      %-+ |zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
     %-+ 0|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
     %-+ #|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
    %-+ #0|zap  |*    |%    |-3.46  |-3.46e+00 |-3.457    |
----------+----------------------------------------------
```

Aligned output with printf()

You can produce columns

These statements

```
int d=42, h3=12168, o2=3344;
printf( "     %d %d %d \n", d, h3, o2 );
printf( "     %d %d %d \n", h3, o2, d );
printf( "     %d %d %d \n", o2, d, h3 );
```

produce

```
42 12168 3344
12168 3344 42
3344 42 12168
```

while these

```
printf( "     %6d %6d %6d \n", d, h3, o2 );
printf( "     %6d %6d %6d \n", h3, o2, d );
printf( "     %6d %6d %6d \n", o2, d, h3 );
```

produce

```
    42  12168   3344
 12168   3344     42
  3344     42  12168
```

Aligned output with printf()

You can remove unneeded blanks

These statements

```
float pct = 0.255 ;
printf( "Ha! %9.2f%% of your horses lost!\n",pct*100.);
```

produce

```
Ha!      25.50% of your horses lost!
```

while these

```
float pct = 0.255 ;
printf( "Ha! %.2f%% of your horses lost!\n",pct*100.);
```

produce

```
Ha! 25.50% of your horses lost!
```

Data conversion using printf()

The statement

```
printf("    %d, %x, %o, %d, %u\n",
     511,511,511,-511,-511);
```

produces (on VM)

```
511, 1ff, 777, -511, 4294966785
```

and this statement

```
printf( "    %c is ascii %d ( %#x )\n",
     'C', 'C', 'C' );
```

produces

```
C is ascii 195 ( 0xc3 )
```

scanf() differences

- Uses whitespace to separate input values.
- Expects 'pointers' to variables
  - For basic data types, precede the name with &
  - For string variables, just use the variable name
- There is no *%g* option
- *%f* and *%e* are equivalent, both accept signs, digits, decimal points, and exponent fields
- To read strings not delimited by whitespace, a set of characters in brackets ([ ]) may be substituted for the s(string) type character. This causes the corresponding input field to be read up to the first character that is not in the bracketed set. If the first character in the set is '^', the effect is reversed.

scanf() control string

The control string is a picture of the expected form of the input. The contents of this string fall into three categories:

- Whitespace characters - a whitespace character causes whitespace characters to be read and discarded. The first input character that is not a whitespace character will remain as the first character to really be read into a variable. A sequence of whitespace characters acts just like a single one.

- Conversion specifications - a conversion specification begins with a "%" and is followed by one of the same conversion identifiers used by *printf* (except *%g*). The conversion operation processes characters until either

  1. end of file is reached
  2. a whitespace or other inappropriate character is encountered,
  3. the number of characters read equals the explicitly specified maximum field width.

scanf() control string

- Any other characters must match the next character of the input stream. If it does not match, the *scanf()* terminates and the conflicting input character remains in the input stream.

scanf conversion modifiers

A conversion specification begins with the '%' and is followed by the following elements in the following order:

1. an optional assignment suppression flag, written as an '*'. If this flag is present in a specification that would otherwise cause an assignment, then the input characters for that assignment are read as usual, but no assignment is done and no pointer variable is used.
2. an optional maximum field width expressed as an integer
3. an optional size specification: 'h' indicating the argument is "short" variable; 'l' (lowercase L) to indicate that the argument is long.
4. a required conversion operation, one of the characters "cdeEfousxX%".

scanf() example

```
main()
{
    char descr[80], partch[5],  partnum[5];
    int row, bin;

    printf ("enter part number: ");
    scanf ("%5[^1234567890]", partch);
    scanf ("%5[1234567890]", partnum);

    printf ("enter part description & row-bin: ");
    scanf ("%79s %d-%d", descr, &row, &bin);

    printf ("\npartch=%s partnum=%s descr=%s row=%d bin=%d\n",
                partch, partnum, descr, row, bin);
}
```

```
p234
enter part number: kmr4711
enter part description & row-bin: StraitJacket 35-016

partch=kmr  partnum=4711  descr=StraitJacket  row=35  bin=16
```

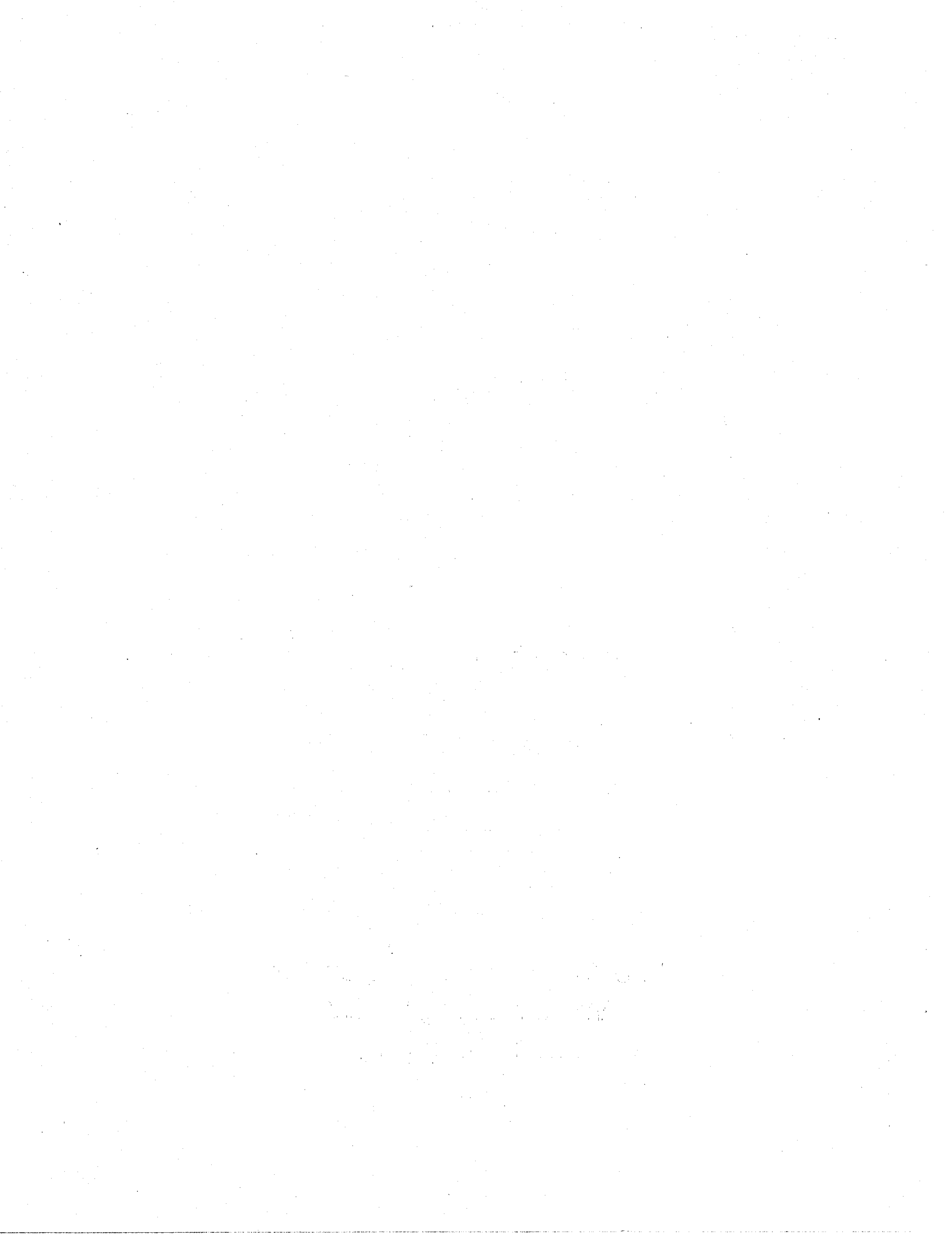
An Introduction to the C Programming Language

Class 3

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

IBM
T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only



Outline

- ★ **Usual Operators**
- ★ **Unusual, Very Specialized Operators**
- ★ **Expressions & Statements**
- ★ **Automatic Conversions**

Usual Operators

=	assignment
–	unary minus
+ – *	standard operators
/	integer or floating point divide, depending upon the types of the operands
( )	parentheses (aka ‘bananas’)

The usual (non-APL) precedence rules apply

strcat(): version one

```
/* strcat(): version one */

main()
{
   static int i = 0, j=0;
   static char line1[40] = "Whether ";
   static char line2[12] = "'tis nobler";

   printf( "line1 was '%s'\n", line1);
   printf( "line2 was '%s'\n", line2);

   while ( line1[i] != '\0' )
   {
       i = i + 1;
   }
   while ( line2[j] != '\0' )
   {
       line1[i] = line2[j];
       i = i + 1;
       j = j + 1;
   }
   line1[i] = line2[j];

   printf( "line1 is now '%s'\n", line1 );
   exit(0);
}

p33
line1 was 'Whether '
line2 was ''tis nobler'
line1 is now 'Whether 'tis nobler'
```

Integer .vs. Float Division

```
/* integer .vs. float division  */

main()
{
    printf( "integer division:  5/4 = %d\n", 5/4 );
    printf( "                   6/3 = %d\n", 6/3 );
    printf( "                  12/5 = %d\n\n",12/5 );
    printf( "float division: 12./5. = %2.2f\n", 12./5. );
    printf( "                 5./4. = %2.2f\n\n", 5./4. );
    printf( "mixed division:  5./4  = %2.2f\n", 5./4 );
    exit(0);
}

p34
integer division:  5/4 = 1
                   6/3 = 2
                  12/5 = 2

float division: 12./5. = 2.40
                 5./4. = 1.25

mixed division:  5./4  = 1.25
```

Modulus Operator

The % operator performs the modulo function. That is, it divides its left operand by its right and returns the remainder.

```
/* modulus operator */

main()
{
   printf( "  5 %% 4 = %d\n", 5%4 );
   printf( "  6 %% 3 = %d\n", 6%3 );
   printf( " 12 %% 5 = %d\n", 12%5 );
   printf( "  2 %% 5 = %d\n", 2%5 );
   exit(0);
}

p35
  5 % 4 = 1
  6 % 3 = 0
 12 % 5 = 2
  2 % 5 = 2
```

Increment & Decrement Operators

Very often a variable needs to be incremented or decremented by 1 unit. These special operators handle this quite well. There are two forms of each: prefix and postfix.

++z Adds one to z BEFORE it is used (prefix).

--z Subtracts one from z BEFORE it is used (prefix).

z++ Adds one to z AFTER it is used (postfix).

z-- Subtracts one from z AFTER it is used (postfix).

These operators have very high precedence; only parentheses are higher. So, x*y++ means (x)*(y++) and not (x*y)++ which is meaningless (to C anyway).

Increment & Decrement Example

```
/* increment/decrement prefix.vs. postfix  */

main()
{
   static int    a = 0, b = 0;
   static float  i = 4.5, k = 4.5;

   printf( "  a  a++  b  b++\n%3d %3d %3d %3d\n\n",
                                a, a++, b, b++ );

   printf( "  a  ++a b++  b \n%3d %3d %3d %3d\n\n",
                                a, ++a, b++, b );

   printf( " a++ ++a b++ ++b\n%3d %3d %3d %3d\n\n",
                               a++, ++a, b++,++b);

   printf( "a=%d, b=%d\n", a, b );
   exit(0);
}

p37
  a  a++  b  b++
  0   0   0   0

  a  ++a b++  b
  1   2   1   2

 a++ ++a b++ ++b
  2   4   2   4

a=4, b=4
```

Operator Precedence (so far)

Operators (Hi→Lo priority)	Grouping
( )	L-R
++ -- -(unary)	R-L
* / %	L-R
+ -	L-R

Grouping describes how the operands of an operator are determined when precedence isn't enough. For example, the expression `j*c%n` is treated as `(j*c)%n` due to the `L-R` grouping of both operators. The other case is shown in the expression `a---b` which would result in `a-(--b)`.

W.D.T.P. #1

```
/* WDTP */

main()
{
   static int this, that= 10, them = 5;

   this = 4 * 6 + 12 % 4 / 3;  printf( "%d\n", this);

   this = -4 / 6 + 12 - - 4;   printf( "%d\n", this);

   this = that++ + ++them;     printf( "%d\n", this);

   this = --that * them / ++them;
                               printf( "%d\n", this );
   exit(0);
}

p39
24
16
16
8
```

Bit-Fiddling Operators

~	One's complement, or bitwise negation
&	AND
|	OR
^	Exclusive OR
<<	Left shift
>>	Right shift

W.D.T.P. #2

```
/* WDTP */


main()
{
   int claw, hoof, fin, root, toes;
   claw = 3; hoof = 2; fin = 1; root = -1;

   toes = claw | hoof & fin   ; printf( "%d\n", toes);

   toes = claw | hoof & ~ fin ; printf( "%d\n", toes);

   toes = claw ^ hoof & ~ fin ; printf( "%d\n", toes);

   toes = claw >> hoof | fin      ; printf( "%d\n", toes);

   toes = fin << 3                    ; printf( "%d\n", toes);

   toes = root << 3                   ; printf( "%d\n", toes);

   toes = root >> 3                   ; printf( "%d\n", toes);

   exit(0);
}

p311
3
3
1
1
8
-8
-1
```

Assignment Operators

<table>
<tr><th>Op</th><th>Use</th><th>Effect</th></tr>
<tr><td>+=</td><td>a+=b;</td><td>a=a+b;</td></tr>
<tr><td>-=</td><td>a-=b;</td><td>a=a-b;</td></tr>
<tr><td>*=</td><td>a*=b;</td><td>a=a*b;</td></tr>
<tr><td>/=</td><td>a/=b;</td><td>a=a/b;</td></tr>
<tr><td>%=</td><td>a%=b;</td><td>a=a%b;</td></tr>
<tr><td>>>=</td><td>a>>=b;</td><td>a=a>>b;</td></tr>
<tr><td><<=</td><td>a<<=b;</td><td>a=a<<b;</td></tr>
<tr><td>&=</td><td>a&=b;</td><td>a=a&b;</td></tr>
<tr><td>|=</td><td>a|=b;</td><td>a=a|b;</td></tr>
<tr><td>∧=</td><td>a∧=b;</td><td>a=a∧b;</td></tr>
</table>

W.D.T.P. #3

```
/* more operators */

main()
{
   static int  it, cu = 8, fe = 5, pb = 4;

   it = cu % fe / pb ;               printf( "%d\n", it);

   it += pb + 2;                     printf( "%d\n", it);
   it %= cu++ + --cu;                printf( "%d\n", it);
   it >>= fe % 2;                    printf( "%d\n", it);
   it ^= ~cu | ~pb;                  printf( "%d\n", it);
   it /= ~(pb * pb % cu + 1);        printf( "%d\n", it);

   exit(0);
}
```

```
p313
0
6
6
3
-4
2
```

sizeof and casts

sizeof

Yields the size, in bytes, of the operand to its right. The operand can be a type-specifier inside bananas, as in sizeof(float), or it can be the name of a particular variable or array, as in sizeof shoe;

(type)

The cast operator: converts the following thing to the type specified by the enclosed type name. For example, (float) 42 converts the integer 42 to the floating point number 42.0.

sizeof and casts

```
/* sizeof  and   casts */

main()
{
   static int     x = 4, heap[100],dog;
   static char    broiled[ ]
                  = {'w','e','l','l',' ','d','o','n','e'};
   static double  bubble = 831.1956;

   printf( "My int is %d bytes long.\n", sizeof(int) );
   printf( "So, 100 of 'em take %d bytes.\n", sizeof heap);
   printf( "And the char array has %d bytes.\n", sizeof broiled);

   dog = 1.987 + bubble;
   printf( "Auto dog = %d, ",dog);
   dog = (int) 1.987 + (int) bubble;
   printf( "Cast dog = %d, ",dog);

   exit(0);
}

p315
My int is 4 bytes long.
So, 100 of 'em take 400 bytes.
And the char array has 9 bytes.
Auto dog = 833, Cast dog = 832,
```

Operator Precedence (so far)

Operators (Hi→Lo priority)	Grouping
( ) [ ]	L-R
++ -- -(unary) (cast) sizeof ~	R-L
* / %	L-R
+ -	L-R
<< >>	L-R
&	L-R
^	L-R
|	L-R
all assignment ops	R-L

Be Careful With Cleverness

- Don't use ++ or -- operators on a variable that is part of more than one argument of a function.
- Don't use ++ or -- operators on a variable that appears more than once in an expression.

```
/* Cleverness backfire example one */
main()
{
   static int num = 0;

   while( num < 4 )
   {
      printf( "%10d %10d\n", num, num*num++ );
   }
   exit(0);
}

p317  /* CW */
         0          0
         1          2
         2          6
         3         12
         4         20

p317  /* PC - MSC */
         1          0
         2          1
         3          4
         4          9
         5         16
```

Expressions & Statements

An expression consists of one or more operands and zero or more operators. Examples of expressions are:

```
42
-109
a*0777
x = flags >> 8 & 0xf0
```

A compound statement is two or more complete statements grouped together by enclosing them in braces, { & }. A compound statement can be used anywhere that a simple statement can be.

An important feature of expressions is that every expression has a value. To find the value, perform the operations in the expression.

```
expression              value

   -4                    -4
   4+10                  14
  c = a + b              a+b
  6+ ( c= 2*5 )          16
  a = b = c = 96;        96
```

Expressions & Statements

Because expressions always have a value, these two programs produce the same results.

```
/* The hard way */
main()
{
   static int num = 0, temp;
   while( num < 20 )
   {
       num++;  /* or, num = num + 1 or num += 1;  */
       temp = num * num;
       printf( "%10d %10d\n", num, temp);
   }
   exit(0);
}

/* =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-= */

/* Use the value for an expression */
main()
{
   static int num = 0, temp;
   while( ++num < 21 )
       printf( "%10d %10d\n", num, (temp=num*num) );
   exit(0);
}
```

Automatic Conversion

For arithmetic operations, the following sequence of conversion rules are applied in order:

- char & short are converted to int, and float is converted to double.
- if either operand is double, the other is converted to double, and the result is double.
- Otherwise, if either operand is long, the other is converted to long and the result is long.
- Otherwise, if either operand is unsigned, the other is converted to unsigned and the result is unsigned.
- Otherwise, the operands must be int and the result is int.

Automatic Conversion Example

```
/* automatic conversions */

main()
{
   int eger;  char actor;  float ers;

   eger = ers = actor = 'Q';
   printf( "actor='%c', ers=%f, eger=%d\n",
           actor, ers, eger );

   ++actor;
   eger = ers + 2 * actor;
   ers  = 2.0 * actor + eger;
   printf( "actor='%c', ers=%f, eger=%d\n",
           actor, ers, eger );

   actor = 2.123e4;
   printf( "actor= '%c'(%0x)\n", actor, actor );

   exit(0);
}

p321
actor='Q', ers=216.000000, eger=216
actor='R', ers=1084.000000, eger=650
actor= '□'(ee)
```

An Introduction to the C Programming Language

Class 4

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

★ Conditional Statements (if-then-else)

★ Relational Operators

★ C Truth

★ Logical Operators

★ Conditional Operator

if Statement

- Typical decision statement
- Syntax

```
        if (expression)

            statement
```

- The expression is usually a relational one, but any kind can be used.
- The statement can be a simple one or a compound statement or block.
- Indentation is strongly recommended, but not required.
- If the expression evaluates to zero the statement is NOT executed.
- If the expression evaluates to non-zero the statement will be executed.
- The parentheses are required.

if Statement

```
/* line & char counting program #1 */
#include <stdio.h>
main()
{
   char ch;
   int ccount = 0, lcount = 0;

   while( (ch = getchar()) != EOF )
   {
       ++ccount;
       if( ch == '\n' )
           ++lcount;
   }

   printf( "The input consists of %d lines,", lcount);
   printf( "and %d chars.\n", ccount);
   exit(0);
}
```

```
p43
this is[enter]
a test[enter]
this is[enter]
only a [enter]
test...[enter]
/*[enter]
The input consists of 5 lines, and 38 chars.
```

if-else

- **Syntax**

```
if (expression)
    statement1
else
    statement2
```

- **The expression is usually a relational one, but any kind can be used.**

- **Either statement can be a simple one or a compound statement or block.**

- **Indentation is strongly recommended, but not required.**

- **If the expression evaluates to non-zero statement1 will be executed.**

- **If the expression evaluates to zero statement2 will be executed.**

- **As far as C is concerned, this if-else pair is a single statement and may be the object of a for or while clause without enclosing braces.**

if-else

```
/* line & char counting program #2 */
#include <stdio.h>
main()
{
   char ch;
   int ccount = 0, lcount = 0;

   while( (ch = getchar()) != EOF )
   {
    /* don't count newlines as chars */
       if( ch == '\n' )
           ++lcount;
       else
           ++ccount;
   }

   printf( "The input consists of %d lines,", lcount);
   printf( "and %d chars.\n", ccount);
   exit(0);
}
```

```
p45
this is[enter]
a test[enter]
this is[enter]
only a [enter]
test...[enter]
/*[enter]
The input consists of 5 lines, and 33 chars.
```

if-else is one statement!

```
#include <stdio.h>

main()
{
   char ch;

   puts( " Enter a mixed case line:\n");

   while( (ch=getchar()) != '\n' )
                                                  /* { */
       if( isupper( ch ))
           putchar( ch );
       else
           putchar( '*' );
                                                  /* } */
}
```

```
p46
 Enter a mixed case line:
the Society To Abolish Basic
****S*******T**A*******B****
```

Nested if's

Since *if (expression) statement* is itself a statement, then it follows that we can use this construct anywhere we need a *statement*, i.e. in an *if-else* statement.

```
/* line & char counting program #3 */
#include <stdio.h>
main()
{
   char ch;
   int ccount = 0, lcount = 0;
   while( (ch = getchar()) != EOF )
   { /* don't count blanks or newlines as chars */
       if( ch == '\n' )
           ++lcount;
       else if ( ch != ' ' )
           ++ccount;
   }
   printf( "The input consists of %d lines,", lcount);
   printf( "and %d chars.\n", ccount);
   exit(0);
}

p47
this is[enter]
a test[enter]
this is[enter]
only a [enter]
test...[enter]
/*[enter]
The input consists of 5 lines, and 29 chars.
```

if-else Pairing Rule

An else works with the most recent if unless braces indicate otherwise.

```
/* if-else pairing tester #1 */
#include <stdio.h>
main()
{
   char ch;
   printf( " Gimme a char: ");
   if( (ch = getchar()) != EOF )
       if(ch > 'Z')
           printf( " Off the end\n" );
       else
           if (ch >= 'A')
               printf( " %c isa capital\n",ch );
   else
       puts( " EOF detected\n" );

   puts( " byebye\n" );
}
```

```
p48
 Gimme a char: D
 D isa capital
 byebye

p48
 Gimme a char: 3
 EOF detected
 byebye
```

Let's try again

```
/* if-else pairing tester #2 */
#include <stdio.h>
main()
{
   char ch;
   printf( " Gimme a char: ");
   if( (ch = getchar()) != EOF )
   {
       if(ch > 'Z')
           printf( " Off the end\n" );
       else
           if (ch >= 'A')
               printf( " %c isa capital\n",ch );
   }
   else
       puts( " EOF detected\n" );

   puts( " byebye\n" );
}
```

```
p49
 Gimme a char: D
 D isa capital
 byebye

p49
 Gimme a char: 3
 byebye

p49
 Gimme a char: ^Z
 EOF detected
 byebye
```

Relational Operators

<	is less than
<=	is less than or equal to
==	is equal to
>=	is greater than or equal to
>	is greater than
!=	is not equal to

The relational operators associate left-to-right.

Relational Operators

A Classic Error

Be very careful not to confuse the == operator with the = operator. The == operator performs equality comparison, while the = operator does simple assignment.

Statements like this

```
while ( token = next_one() )
{   ...
```

are very common (and efficient) in C programs, but programmers must use the correct operator to accomplish their goal. For example, the programmer could just as easily have meant to only perform a test rather than an assignment:

```
while ( token == next_one() )
{   ...
```

If the function `next_one()` happens to always return a non-zero value, the first while will never fail its condition!

Priority of Relational Operators

Their priority is less than that of + and – and greater than that of assignment.

```
    pair < pile + bunch  ≡  pair < (pile + bunch)

(ch = getchar() != EOF)  ≡  (ch = (getchar() != EOF))
                instead of  ((ch = getchar()) != EOF)
```

The relational operators themselves are grouped into two levels of priority :

```
    higher: < <= >= >
    lower:  == !=
```

So ...

```
    bay <= boy == buoy  ≡  (bay <= boy) == buoy
    ch != EOF == TRUE     ≡  (ch != EOF) == TRUE
```

C Truth

A value used in a relational expression is treated as "true" when that value is not zero.

As a result of this simple definition, you shouldn't depend upon the result of a relational expression being '0' or '1'; you can only be sure that it will be either '0' or it won't.

```
main()
{
   int a=4, b=2, c=4;

   printf ("(a==b) evaluates to %d\n",(a==b));
   printf ("(a==c) evaluates to %d\n",(a==c));
}

/* on some systems */
p413
(a==b) evaluates to 0
(a==c) evaluates to 1

/* on others */
p413
(a==b) evaluates to 0
(a==c) evaluates to -1
```

C Truth Example

```
#include <stdio.h>
main()
{
    static int johns = 9;
    static char terr  = 'F', ch;

    if( johns )     printf( " %d is true\n", johns );
    if( terr )      printf( " %d is true\n", terr );
    if( NULL )      printf( " %d is true\n", NULL );

    if( (ch = getchar()) )
      printf( " getchar returns true\n" );
    if( johns = (terr == 'F') )
      printf( " multiple assignments can also\n" );
    if( johns -= '1' )
      printf( " negative numbers (%d) are true\n",johns);
    if( johns = terr = NULL )
      printf( " multiple NULL assignments can ???\n" );
    else
      printf( " multiple NULL assignments are false\n" );
}
```

```
p414
 9 is true
 198 is true
A
 getchar returns true
 multiple assignments can also
 negative numbers (-240) are true
 multiple NULL assignments are false
```

Logical Operators

Logical operators allow you to combine two or more relational expressions.

&& and

|| or

! not

expr1 && expr2 is true iff both expr1 and expr2 are true:

```
5 > 2 && 4 > 100  is false.
```

expr1 || expr2 is true if either or both of expr1 and expr2 are true.

```
5 > 2 || 4 > 100  is true
```

!expr1 is true if expr1 is false and vice versa.

```
!( 4 > 100 )  is true
```

Priorities

The ! operator has a very high priority, above multiplication, the same as increment operators, and just below parentheses. The && operator ranks higher than the || and both rank below the relational operators and above assignment.

```
! i > t && c < h || e == s
```

would be interpreted as

```
( ( (!i) > t) && (c < h) ) || ( e == s )
```

Order of Evaluation

Standard C does not guarantee which parts of an expression will be evaluated first. So in the statement *ans = (a + 2) * (b - 2)* you can not depend upon *(a + 2)* being evaluated first. The only exception to this rule is the way logical operators are handled. Standard C does guarantee that (1) logical operations are evaluated from left to right, & (2) that as soon as a sub-expression is found that will force the whole expression to be false, the evaluation halts.

```
/* don't test an already known value */
    while( (ch = getchar()) != EOF && ch != '\n') ...

/* prevent division by zero */
    if (denom != 0 && numer/denom >= 60) ) ...
```

Conditional Operator ?:

A shorthand for *if-else* , the conditional operator is a two-part operator that has three operands.

```
knit = (wit < 0) ? -wit : wit ;
```

Everything between the '=' and the ';' is the conditional expression. This statement can be read "if wit is less than zero then assign -wit to knit, otherwise assign wit to knit. "

Example

```
main()
{
   static int index=0;
   static int list[24] = {0,1,2,3,4,5,6,7,
                          8,9,10,11,12,13,14,15,
                          16,17,18,19,20,21,22,23 };

   while( index < (sizeof list / sizeof(int)) )
   {
       printf( "list[%02d] = %02d", index, list[index] );
       printf( "%s", (++index % 3 ? ", " : "\n"));
   }
   exit(0);
}
```

```
p419
list[00] = 00, list[01] = 01, list[02] = 02
list[03] = 03, list[04] = 04, list[05] = 05
list[06] = 06, list[07] = 07, list[08] = 08
list[09] = 09, list[10] = 10, list[11] = 11
list[12] = 12, list[13] = 13, list[14] = 14
list[15] = 15, list[16] = 16, list[17] = 17
list[18] = 18, list[19] = 19, list[20] = 20
list[21] = 21, list[22] = 22, list[23] = 23
```

Operator Precedence (so far)

Operators (Hi→Lo priority)	Grouping
( ) [ ]	L-R
++ −− −(unary) (cast) sizeof ~	R-L
* / %	L-R
+ −	L-R
<< >>	L-R
< > <= >=	L-R
== !=	L-R
&	L-R
^	L-R
|	L-R
&&	L-R
||	L-R
? :	R-L
all assignment ops	R-L

An Introduction to the C Programming Language

Class 5

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

- *while* loop
- *for* loop and the comma operator
- *do while* loop
- *break* and *continue*
- *switch*
- *goto* (ugh!)

while Statement

- General form:

```
while( expression )
     statement
```

- The expression in parentheses that follows the while keyword is evaluated before the statement is executed.
- The statement (compound statement) is executed only if the expression evaluates to be true (non-zero). Otherwise, the program continues at the next statement.
- If the statement is executed, the expression within the parentheses will be evaluated again and one of the above actions will be taken.
- Either the expression or the statement must do something to cause the expression to eventually evaluate to false or the statement must contain a *break* statement to terminate the looping.
- The statement can be simple, compound, or the null statement ';'.

Example 1

```
#include  <stdio.h>
/* skip leading whitespace & print first word */
main()
{
   char ch;

   printf( "Enter a line\n" );
   while( (ch = getchar()) == ' ' ||
                      ch == '\t' || ch == '\n' ) ;

   putchar('\'');
   putchar(ch);
   while( (ch = getchar()) != ' ' &&
                     ch != '\t' && ch != '\n' )
       putchar(ch);
   printf( "' was the first word.\n" );
   exit(0);
}

p53
Enter a line
     APL tends to be a 'write only' language.
'APL' was the first word.

p53
Enter a line


         Thereweresseveral empty lines before this one.
'Thereweresseveral' was the first word.
```

Example 2

```
#include <stdio.h>

main()
{ /* cute strlen() */
   static int len = 0;
   char thing[100];

   printf( " Gimme a string \n" );
   scanf( "%s", thing );

   while( thing[len++] ) ;

   printf( " The length of '%s' is %d.\n", thing, --len );
   exit(0);
}

p54
 Gimme a string
ThisStringHasThirty-nineCharactersInIt.
 The length of 'ThisStringHasThirty-nineCharactersInIt.' is 39.
```

for Statement

- General form

```
for( initialize;  exit-test;  update )
   statement
```

- The initialize expression is executed once, before the statement is executed.
- If the exit-test expression is true (non-zero) the statement is then executed once. Otherwise, the statement is skipped.
- If the statement was executed, then the update expression is evaluated.
- The previous two steps are repeated until the evaluation of the exit-test expression becomes false.
- Any of the three expressions within the parentheses may be omitted.
- The statement can be simple, compound, or the null statement ';'.

Example 1

```
#include <stdio.h>
main()
{ /* strchr(), sort of  */
   char ch, pile[80];
   int len, i, num;

   printf( " Enter the string and the search char: " );
   scanf( "%s %c", pile, &ch );
   len = strlen(pile);
   num = 0;

   for ( i=0; i<len; ++i )
       if ( pile[i] == ch )
       {
           printf( " There is a %c at position %d\n", ch, i);
           ++num;
       }
   printf( " %d %c%s found in %s\n",
         num, ch, ( num==1 ? " was" : "'s were" ), pile );
   exit(0);
}
```

Example 1

```
p56
 Enter the string and the search char: thisisapile i
 There is a i at position 2
 There is a i at position 4
 There is a i at position 8
 3 i's were found in thisisapile
p56
 Enter the string and the search char: thisisapile p
 There is a p at position 7
 1 p was found in thisisapile
p56
 Enter the string and the search char: thisisapile q
 0 q's were found in thisisapile
```

Example 2

```
#include <stdio.h>
main()
{  /* slick character counter */
   int i;
   char ch;

   for( i=0; (ch=getchar()) != EOF; ++i);

   printf( " The input consisted of %d chars.\n", i);
   exit(0);
}

p58  <p58.c
 The input consisted of 193 chars.
```

Example 3

```
#include <stdio.h>
main()
{  /* obscure tab-expanding filter */
   char ch;
   static char exp[9] = "        ";

   for( ; ((ch=getchar()) != EOF);
          (ch=='\t' ? printf("%s",exp) : putchar(ch)) );

}

p59 < p59.fil > p59.out

type p59.fil
1→2→3→4→5
d→d→d→s→a
5→333^Z

type p59.out
1        2        3        4        5
d        d        d        s        a
5        333
```

Comma Operator

The comma operator can be used to allow more than one initialization or update expression in a *for* statement.

It can also be used to guarantee that the expressions that it separates will be evaluated in a left-to-right order. Without using the comma operator, you can not be sure of the order in which the expressions would be handled.

```
main()
{
    int i, sum;

    for( i=0, sum=i; i<100; sum += i, i++ );
    printf( "The sum of the ints 0 thru %d is %d\n",--i,sum);
    exit(0);
}

p510
The sum of the ints 0 thru 99 is 4950
```

However, commas in declaration statements and in function argument lists are only separators.

Comma Operator

A handy side effect of this operator is that while it separates two expressions and they are evaluated left-to-right, the overall value of the whole expression is the value of the last expression evaluated. This feature can be used to produce some pretty slick (and sometimes hard to understand) statements.

For example, a programmer had built the following statement to set the return code for a program:

```
rc = ( error_level > 10 ? 2 : 1 );
```

When the specification changed, causing the programmer to need to set an additional variable when `rc` was set to 2, this code resulted:

```
rc = ( error_level > 10 ? emsg=5, 2 : 1 );
```

do while Statement

This statement is similar to the *while* statement except that it is an exit-condition loop. That is, the condition that eventually evaluates to false and causes the termination of the loop is checked *after* the statement(s) of the loop are executed. So the loop will always be executed at least once.

The general form is:

```
do

  statement

while( expression );
```

Example

```
main()
{ /* prompter */

   char ch;

   printf( " does 4096 x 4096 = 1677216?\n" );
   do
   {
       printf( " answer with 'y' or 'n': ");
       scanf("%c", &ch);
   } while( ch != 'n' && ch != 'y' );

   if (ch == 'y')
       printf( " You're correct\n" );
   else
       printf( " You're wrong\n" );
   exit(0);
}
```

```
p513
 does 4096 x 4096 = 1677216?
 answer with 'y' or 'n':
a[enter]
 answer with 'y' or 'n':  answer with 'y' or 'n':
N[enter]
 answer with 'y' or 'n':  answer with 'y' or 'n':
n[enter]
 You're wrong
```

break Statement

This statement causes the program to break free of the *for*, *while*, *do while* or *switch*, statement that encloses it, and to continue at the next statement.

```
/* fragment to echo until EOF or '\n' */

    while( ( ch = getchar() ) != EOF )
    {
         if ( ch == '\n' )
             break;
         else
             putchar( ch );
    }
```

continue Statement

This statement can be used with the loop statements *for*, *while*, or *do while* only. Like *break*, it interrupts the usual flow of the program. Unlike *break*, instead of terminating the loop, the *continue* statement causes the rest of an iteration (pass) to be skipped and the next one to be started.

```
/* fragment to echo everything except '\n' until EOF */

    while( ( ch = getchar() ) != EOF )
    {
         if ( ch == '\n' )
             continue;
         else
             putchar( ch );
    }
```

switch & break Statements

If a program has to choose between several alternatives one way to write the program is to use a large *if-else if-else if-else ...* construction. A far more convenient and clear way is to use the switch and break statements provided by C.

The switch statement provides an orderly way to arrange the alternatives as well as a default or "catch-all" choice. It is similar to the case statement of other languages.

The break statement causes the program to get out of the switch construct and to continue at the statement following the whole switch construct.

switch Example #1

```
#include  <stdio.h>

main()
{
   char ch;

   printf( " Gimme the 1st letter of an instruction \n" );
   while( (ch = getchar()) != EOF )
   {
       if ( ch != '\n' )
       {
           if (ch >= 'a' && ch <= 'z')
               switch(ch)
               {
                   case 'a':
                       printf(" add instruction\n");
                       break;
                   case 'j':
                       printf(" jmp instruction\n");
                       break;
                   case 's':
                       printf(" sub instruction\n");
                       break;
                   case 'c':
                       printf(" cmp instruction\n");
                       break;
                   default:
                       printf( " a,c,j, or s please\n" );
                       break;
               }
           else
               printf( " lower case only, please\n" );

           printf( " Gimme another one\n" );
       }
   }
   printf( "EOF detected\n" );
}
```

switch Example #1

```
p517
 Gimme the 1st letter of an instruction
a
 add instruction
 Gimme another one
d
 a,c,j, or s please
 Gimme another one
J
 lower case only, please
 Gimme another one
j
 jmp instruction
 Gimme another one
^Z
 EOF detected
```

switch details

- The case values must be of integer type, including char, and can be either constants or expressions.
- The expression within the parentheses must evaluate to an integer type, including char.
- The statements within each case, including the *break*, are optional, allowing for groups of cases to be treated the same.
- The default case is optional.
- The break statements are optional, but without them the program will always execute all of the following case's statements up to the next break or the end of the switch block.

switch Example #2

```
#include <stdio.h>
main()
{
   char ch;

   printf( " gimme a single digit number: " );
   ch = getchar();
   switch(ch)
   {
     case '0':
     case '1':
     case '5':
     case '4':
     case '8':
         printf( " the word for %c has 2 different vowels\n",
                                                        ch);
     case '2':
     case '6':
         printf( " the word for %c has no repeated letters\n",
                                                         ch);
         break;
     case '3':
     case '7':
     case '9':
         printf( " the word for %c has repeated letters\n"
                                                      ch);
   }
}
```

switch Example #2

```
p520
 gimme a single digit number: 4
 the word for 4 has 2 different vowels
 the word for 4 has no repeated letters

p520
 gimme a single digit number: 3
 the word for 3 has repeated letters

p520
 gimme a single digit number: A
```

The “Infinitely Abusable” goto

Although K&R suggest that it “be used sparingly, if at all”, the *goto* statement is provided in C. However, with the many control and looping statements available in C, one has very few justifiable needs for the *goto* statement.

The use of goto to completely escape from very nested loops is usually not frowned upon, since *break* only gets out of the innermost loop.

A tolerable goto use

```
#include <stdio.h>

main()
{
   int i, j, k;

   /* ... */
   for( i=0; i<100; i++ )
   {
       for( j=0; j<200; j++ )
       {
           for( k=0; k<50; k++ )
           {
               /* lots of processing */
               if ( self-destruct order given )
                   goto harikari
               else ...
           }
           other statements ...
       }
       still other statements ...
   }
   a few more statements ...
   exit(0);

   harikari: printf( "we, who are about to die, salute you!\n" );
   exit(42);
}
```

An Introduction to the C Programming Language

Class 6

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

IBM
T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

★ **Preprocessor directives**

- **Macros**
- **File Inclusion**
- **Conditional Compilation**

★ **Debugging Techniques**

Preprocessor Directives

The preprocessor is the first part of the C language processor that gets your source file. There are several commands that can be put in a source file to direct the preprocessor to do different things.

- A line whose first character is '#' is treated as a preprocessor command. The name of the command must follow the # immediately. Most compilers also require that the # be the first character on the line.
- The rest of the line can contain arguments for the command if needed.
- Within a preprocessor command line, if a newline character is immediately preceded by a '\', then the newline and the '\' are ignored and the following line is treated as though it was part of the original line.

Preprocessor Directives

- A side effect of the preprocessor is that all comments are removed.
- C language processors provide a way to save the output from the preprocessor for debugging purposes.
- The preprocessor *does not know C*. It will happily do whatever it is directed to do, including the building of invalid statements and the deletion of source code.

Preprocessor Directives

The K&R standard directives are:

#define
: Define a preprocessor macro (constant)

#undef
: Remove a macro (constant) definition

#include
: Insert text from another file

#if
: Conditionally include some text, based upon the value of a constant expression

#ifdef
: Conditionally include some text, based upon whether a macro name is defined

#ifndef
: Conditionally include some text, with the test being opposite of the one for #ifdef

#else
: Alternatively include some text, if the previous #if, #ifdef, or #ifndef test failed.

#endif
: Terminate conditional text

#define

- It is used to define symbolic constants or macros. Quite handy for "magic" numbers like *PI* or *LINES_PER_PAGE*.
- The name that is defined by this command is traditionally in UPPERCASE to enhance readibility and to accent the fact that the name is from a #define.
- This command can appear anywhere in the source file, and its definition holds from its place of appearance to the end of the file or up to an associated #undef command.
- The value of a *define*d thing can be numbers, expressions, subexpressions, ... anything you want.
- The syntax of the command does not require an equal sign or any other special delimiter character after the name or definition. The body of the definition starts with the first non-blank following the name.
- The preprocessor does not know any C keywords, so you can redefine them if you choose to do so.

#define Macros Without Args

There are two forms of this command, depending upon whether or not a left parenthesis immediately follows the name to be defined.

```
#define   name   sequence of tokens
```

A macro of defined this way takes no arguments. It is invoked simply by mentioning its name. However, macro replacement is never performed within comments or constants. Here are some examples:

```
#define   BLOCK_SIZE    4096

#define   ERRMSG1       "Extraneous brouhaha in input line"

#define   PRNL          putchar('\n')

#define   BYTES_PER_REC       512

#define   RECORDS_PER_BLOCK   BLOCK_SIZE/BYTES_PER_REC

#define   do    "repeat"
```

#define Macros Without Args

C language processors provide a way to define preprocessor macros (without args) at compile time using a compilation option. For example, to tell the compiler on UNIX™ that the preprocessor macro `DEBUG:` should be defined and set to 3 before compiling the program `buggy`, the compile command would be

```
cc -DDEBUG=3 buggy.c
```

#define Macros Without Args

Incorrect usage

Here are some examples that are valid, but will probably cause some pain:

```
#define   NUMBER_OF_REPLIES = 5
#define   PIPESIZE   42;
#define   NEWLINE    "\n"
          ...
          count = NUMBER_OF_REPLIES;

          x = PIPESIZE * radius;

          printf( "%c", NEWLINE );
          ...
```

would produce

```
          ...
           count =    =  5 ;

           x = 42; * radius;

           printf( "%c","\n"  );
          ...
```

When does substitution occur

Substitution doesn't usually occur within quotes. For example:

```
/* example program  */

#define   HOOK      '?'

main()
{
   printf( "   A HOOK is a '%c' character\n", HOOK );
}

p69
   A HOOK is a '?' character
```

K & R did not define exactly what should happen to #defined names within quotes. There are some language processors that always do substitution. Check your documentation to be sure.

The ANSI-C standard provides a way to force substitution within quotes selectively.

#define Program Example

```
#define    MINIMS_INA_DRAM    60
#define    DRAMS_INA_OZ        8
#define    OZS_INA_PINT       16
#define    PINTS_INA_QUART     2

main()
{
   int quarts_ina_gallon = 4;
   printf( "Apothecary's Fluid Measures:\n" );
   printf( "%d quarts/gallon\n", quarts_ina_gallon);
   printf( "%d pints/quart\n", PINTS_INA_QUART);
   printf( "%d ounces/pint\n", OZS_INA_PINT);
   printf( "%d drams/ounce\n", DRAMS_INA_OZ);
   printf( "%d minims/dram\n", MINIMS_INA_DRAM);

   printf( "So, there are %d minims/gallon!\n",
     (quarts_ina_gallon * PINTS_INA_QUART *
     OZS_INA_PINT * DRAMS_INA_OZ * MINIMS_INA_DRAM ) );

   exit(0);
}

p610

Apothecary's Fluid Measures:
4 quarts/gallon
2 pints/quart
16 ounces/pint
8 drams/ounce
60 minims/dram
So, there are 61440 minims/gallon!
```

#define Macros With Args

The second form of this command allows for the definition of a macro that accepts arguments.

```
#define   name(arg1,arg2,...,argn)  sequence of tokens
```

- The left parenthesis must immediately follow the macro name or it will be treated as part of the definition.
- The names of the arguments must be valid identifiers, no two the same.
- Arguments names in the list do not have to be used in the body of the macro.
- The macro is invoked by mentioning its name immediately followed by a left parenthesis, then the list of arguments separated by commas, followed by a right parenthesis.
- The argument list may be defined as empty, but then the macro must be called with an empty argument list.

Macro Example Source

```
#include <stdio.h>
#define   TRACE   1
#define   NL   putchar('\n')

#define Skip2nb(p)   while (*p == ' ') p++
#define Nextc(p)     printf("-> nextc=<%c>.\n",*p)
#define Trcode(stmt) if (TRACE)  stmt
#define IsUpCase(c)  (('A'<=c && c<='I')||  \
                      ('J'<=c && c<='R')||  \
                      ('S'<=c && c<='Z'))
```

Macro Example Source

```
main()
{
    int i = 0;
    char string[80], *ptr;;

    printf("Gimme a string");
    NL;
    scanf("%s", string);

    ptr = string;
    Skip2nb(ptr);   /* skip leading blanks */

    for( ; *ptr != '\0'; ++ptr )
    {
        /* debugging statement */
        Trcode( Nextc(ptr) );
        if( IsUpCase(*ptr) )
            ++i;
    }
    printf( "There are %d uppercase chars in '%s'\n",
                                                    i, string);
    exit(0);
}
```

Macro Example

Partial preprocessor output

```
main()
{
    int i = 0;
    char string[80]. *ptr;;

    printf("Gimme a string");
    ( --( (&_iob(:1:))  )->_cnt >= 0 ? ( *( (&_iob(:1:))
     )->_ptr++ = ( '\n' ) ) :
    _flushbuf( ( '\n' ), ( (&_iob(:1:))  ) ) )    ;
    scanf("%s", string);

    ptr = string;
    while (* ptr  == ' ')  ptr ++ ;

    for( ; *ptr != '\0'; ++ptr )
    {

         if (1 )   printf("-> nextc=<%c>.\n",* ptr )   ;

         if( (('A'<= * ptr  &&  * ptr <='I')||
          ('J'<= * ptr  &&  * ptr <='R')||
           ('S'<= * ptr  &&  * ptr <='Z'))  )
              ++i;
    }
    printf( "There are %d uppercase chars in '%s'\n",
                                                  i, string);
    exit(0);
}
```

Macro Example

Program output

```
p614
Gimme a string
AbCDefgHIjk
-> nextc=<A>.
-> nextc=<b>.
-> nextc=<C>.
-> nextc=<D>.
-> nextc=<e>.
-> nextc=<f>.
-> nextc=<g>.
-> nextc=<H>.
-> nextc=<I>.
-> nextc=<j>.
-> nextc=<k>.
There are 5 uppercase chars in 'AbCDefgHIjk'
```

Macro Pitfalls

- **Don't send expressions with side effects (i.e. a++) into macros unless you are sure of what will happen.**

```
#define  Isdigit(n)    ( '0' <= (n) && (n) <= '9' )
     ...
         Isdigit( *p++ );     /* Watch out! */
```

- **There can be no spaces in the macro name or argument list. The preprocessor thinks the macro body begins at the first space, so anything after that is lumped into the replacement string.**

```
#define  SUM(x, y) ( (x) + (y) )  /* Wrong!!! */
```

Macro Pitfalls

- Use parentheses around each argument and around the definition as a whole. This ensures that the terms are grouped properly. For example

```
#define   SQUARE(x)    x * x
       answer = SQUARE(a+b) * 2;
```

  expands to

```
       answer = a+b * a+b * 2;         /*  Surprise! */
```

- Macros cannot be expected to define other preprocessor commands. A line is treated as a preprocessor command if and only if no macro processing has taken place and it starts with a #.

File Inclusion

If you have a lot of constants that you use often, or in several different source files, you can place the *#defines* in a separate file and have that file copied into each of the source files using *#include*.

When it recognizes an *#include* statement, the preprocessor searches for another file with the given name. If it is found, then the file is copied into the current file in the place of the *#include* statement.

File Inclusion

The file to be included can be specified in two ways:

<fileid>
: Which tells the preprocessor to look in the places on the disk where system-related or system-supplied include files are kept.

"fileid"
: Which tells the preprocessor to first look in your own areas (i.e. current directory and/or path) and then in the system-related places.

Under VM, these two forms are equivalent; the standard CMS minidisk search order is always used.

#include example

```
/* copy in our constant file */
#include "mynums"
main()
{
    printf( "Hidden within the mynums.h file are:\n");
    printf( "#defines for PI = %d,\n", PI );
    printf( "         for E  = %d,\n",  E );
    printf( "     and for C  = %d.\n",  C );
}
```

```
type mynums.h

#define  PI   3.14159
#define  E    2.151
#define  C    2.997925E10
```

Conditional Compilation

At times it may be necessary to have different statements compiled in your program depending upon certain situations. Since you would like to minimize the changes that would have to be made under the different circumstances, if would be handy to keep all of the statements in the program for all of the different situations and to be able to select which bunches of statements to use.

An example of this is if your program needs to run on a PC as well as on VM. The character sets are quite different, so statements like:

```
#define    Isupperc(c)    (('A' <=c && c <= 'Z'))
```

would work fine on the PC, but as well on VM or MVS.

The #if preprocessor statement can help us out. It expects a constant expression which will evaluate to either zero or non-zero. If the value is non-zero, then everything up to the next #else or #endif will be compiled. If the value is zero, then everything up to the next #else or #endif will be ignored.

Conditional Compilation

So, one solution would be:

```
#if   ('A'==0xc1)
#define    ONTHEPIG    1
#define    ONTHEPC     0
#define    Isupper(c) (('A'<=c && c<='I')||  \
                       ('J'<=c && c<='R')||  \
                       ('S'<=c && c<='Z'))

#else
#define    ONTHEPIG    0
#define    ONTHEPC     1
#define    Isupper(c)    (('A' <=c && c <= 'Z'))

#endif

main()
{
   ( ONTHEPC ? printf("on the PC") : printf("on VM"));

#if ONTHEPIG
  /* do some vm-specific stuff */
  system( "EXEC QLUNCH" );
#endif

}
p622
on the PC
```

Other uses

- Use *#if 0* as a quick way to comment out some code.
- Use *#undef* followed by *#define* to change the value of a magic number for different stages of a program.
- Use *#if* to conditionally include other files, i.e. a 8087 function .vs. a non-8087 function.
- You should be able to do this to find out where you are:

```
#define  ONTHEPC   ( 'A' == 0x41 )
```

Debugging Code #1

The preprocessor can be quite handy when debugging some code.

```
#ifdef      DEBUG
#define     DPRINTF(args)  printf args
#else
#define     DPRINTF(args)  /* do nothing */
#endif

            ...
            DPRINTF(("index is now %#8x\n", index));
            ...
```

When the preprocessor variable DEBUG is defined, the above statement will be changed by the preprocessor to

```
printf("index is now %#8x\n", index);
```

When the preprocessor variable is not defined, the above statement will be changed by the preprocessor to a ';' which will be discarded by the compiler.

Debugging Code #2

If you want a trace of the path taken through a certain bunch of code, you can use the preprocessor's builtin macros `__FILE__` and `__LINE__`. These variables hold the current file's name as a quoted string and line number as an integer, respectively.

```
#include <stdio.h>        /* defines stdin, stdout, and stderr */

#ifdef  DEBUG
#define DTRACE(var)       fprintf(stderr, "%s:%d  var = %d\n",   \
                                  __FILE__, __LINE__, var)
#else
#define DTRACE(var)       /* do nothing */
#endif
```

Note that these messages are being sent to stderr rather than stdout. This way, you can capture your trace messages in a file through redirection and still see your stdout messages on the screen (or in a different file).

An Introduction to the C Programming Language

Class 7

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

IBM
T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

- ★ Storage classes and scope
- ★ First day discussion/chalk-talk

Local .vs. Global Variables

Variables are either local or global in their scope depending upon where they are declared.

- **If a variable declared within a function, it is a local variable. Only the function that declared it can access it directly. Most variables are local.**

- **If a variable is declared outside of any function then it is a global variable. Any function within the same source file has full access to that variable without declaring anything.**

```
int zimmers;

main()
{
    printf( "How many rooms do you need?\n ");
    scanf( "%d", &zimmers );
    printf( "Ok, that comes to %fDM\n", RoomCheck() );
}

#define   ROOMRATE     43.00
#define   TAXRATE       1.14
float RoomCheck()
{
    return( zimmers * ROOMRATE * TAXRATE );
}
```

Storage Class

In addition to a certain data type, each variable has a storage class. The storage class is defined by where the declaration is and what keyword, if any, is used.

The storage class determines two things:

- It controls the variable's scope, that is, which functions have access to the variable.
- It determines how long the variable will remain in memory.

The four keywords used to describe storage classes are:

- auto
- extern
- static
- register

Automatic Variables

- By default, variables declared within a function are automatic.
- Automatic variables have local scope, so only the function in which they are defined can access them.
- An automatic variable comes into existence each time the function which contains it is called. When the function returns to its caller its automatic variables disappear.
- Automatic variables other than arrays can have initializers. The initial value is given to the variable each time the containing function is called. If no initial value is given, the contents of the variable is undefined.
- Automatic arrays can NOT have initializers and their values are undefined until explicitly set by the program.
- The scope of an automatic variable is limited by the block (the { } pair) which contains it. For example, an automatic variable could be defined within the compound statement of a while statement.

Examples

```
#define   PI  3.14159
main()
{
   auto int radius=10;
   double circumf;

   circumf = calc(0);
   printf ("calc( 0) returned circumf = %g\n",
      circumf);

   circumf = calc(radius);
   printf ("calc(%d) returned circumf = %g\n",
      radius, circumf);

   circumf = calc(0);
   printf ("calc( 0) returned circumf = %g\n",
      circumf);
}

double calc(r)
int r;
{
   int radius = 5;

   if (r > 0)
      radius = r;

   return PI*radius*radius;
}

p75
calc( 0) returned circumf = 78.53975
calc(10) returned circumf = 314.159
calc( 0) returned circumf = 78.53975
```

External Variables

- A variable defined outside of a function is external. An external variable can also be defined using the *extern* keyword.
- The explicit declarations of external variables inside of functions may be omitted if the original definitions occur in the same file and before their use.
- An external variable exists throughout the execution of the program. Since it is not tied to any particular function, it does not come and go with the invocation and exit of any function.
- Any function in the program, whether it is defined within the same source file or not, can access any external variable
- Any external variable may be given initializers at its definition, including arrays.

Examples

```
double bubble = 1986.3;

main()
{
    /* def of bubble: unnecessary but ok */
    extern double bubble;
    ...
    printf( "In main, bubble = %g\n", bubble );
}

HisFun()
{
    /* no need to declare bubble at all, since
       we know about it here by default */
    ...
    printf( "In HisFun, bubble = %g\n", bubble );
}

-=-=-=-=-=-=- in a different file -=-=-=-=-=-=-=-
HerFun()
{
    /* If we don't declare it external here, but we do define
       it, we will get a new automatic variable allocated.
       If we don't declare it at all, we will get an error! */

    extern double bubble;
    ...
    printf( "In HerFun, bubble = %g\n", bubble );
}
```

Static Variables

- These variables have the same scope as automatic variables, however, they do *not* vanish when the function or block is exitted.
- The values of static variables are "remembered" across successive calls to the function in which they are defined.
- These variables can be initialized and the initialization occurs at compile time.
- Statis arrays can be initialized. In the absence of initializers, C guarantees that static arrays are initialized to zero.
- A variable can be declared static outside of a function, thereby making it an external static variable. Such a variable differs from a plain old external variable in that the ordinary external variable can be known to any function defined in any file, while the external static variable can only be used by functions defined (1) within the same file and (2) below the variable definition.

Example # 1

```
main()
{
    int kount;

    for( kount=0; kount < 3; ++kount)
    {
        printf( "Here is pass number %d\n", kount );
        CheckStat();
    }
}

CheckStat()
{
    int goaway = 96;
    static int passnumber = 0;

    printf( "goaway = %d, passnumber = %d\n",
                              goaway++, passnumber++ );
}
```

```
p79
Here is pass number 0 :
goaway = 96, passnumber = 0
Here is pass number 1 :
goaway = 96, passnumber = 1
Here is pass number 2 :
goaway = 96, passnumber = 2
```

Example # 2

italian is known to both ***main()*** and ***LAfood()***, but ***cajun*** is only known to LAFood(). ***italian*** could be known to NYFood(), NJFood().

Source file 1

```
int  italian;
main()
{
 ...
}

static int cajun;
LAFood()
{
 ...
}
```

It is impossible for ***NJFood()*** to know about ***creole*** and ***NYFood()***'s ***char italian*** hides the ***int italian*** of Source file 1.

Source file 2

```
NYFood()
{
  static int creole;
  char italian;
  ...
}


NJFood()
{
  extern int italian;
  ...
}
```

Register Variables

Variables are usually stored in the standard computer memory. Although fast, this memory is not as fast as what is called register memory. However, this latter type of memory is extremely limited.

- If you have a variable that will “fit” in one of your machine’s registers and that is used very heavily within a short function or block, you can “recommend” to the compiler that it should try to keep this variable in a register.
- You can not be guaranteed that this declaration will have any effect, since the compiler will have to decide based upon how many registers there are and how they can best be used.
- Their scope is local and their duration is temporary (just like automatic variables).
- You may not use the & operator on a variable of type register.
- Arrays of register variables are allowed, but seldom effective.

Example

```
/* sum(n) returns the sum of the first n integers as a long int */

long sum (n)
register int n;
{
  register long int sum=0;

  while (n<0)
      sum += n++;
  return sum;
}
```

An Introduction to the C Programming Language

Class 8

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

- ★ **Writing your own functions**
- ★ **Local variables**
- ★ **Call-by-value .vs. call-by-name**
- ★ **Basic pointer use**

Functions

The philosophy of C encourages a "toolbox" approach to programming. Some of the advantages to using a collection of tools, or functions, are:

- They help to prevent repetitious programming both within a single program as well as across several programs.
- They improve the modularity of a program, making it easier to understand and maintain and more portable.
- They can improve reliability by the fact that smaller modules are easier to write and test.

Defining Functions

The syntax is exactly like that for main():

```
#include ... /* preprocessor stuff the function needs */
 ...
functionname (arg1, ...)        /* the name of the function */
                                /* followed by a possibly empty */
                                /* list of args */
declarations for args;          /* declarations for optional args */
 ...
{

   ...    /* optional local variable declarations */

   ...  /* the body of the function */
   ...

   return( optional return value ); /* optional statement to */
                                    /* send a return value */
}
```

- **Functions are *NOT* nested within one another. Each function is defined completely separately.**
- **Functions can be in the same source file or in a different file that is compiled separately.**

Local Variables

- **All variables defined within the { } that enclose a function are known only within that function.**
- **Different functions can define local variables of the same name.**

```
main()
{
    int isotope, weight;
    char name[ ];

    isglowing( weight );
}

isglowing( mass )
int mass;
{
    int name;
    float weight;

    ...
}
```

Example #1 - No Args

```
#include <stdio.h>
/* function with no args, returning nothing */
main()
{
   int i;
   char ch;

   printf( "Do you want a decimal to hex table?\n");
   showprompt();
   if( (ch = getchar()) == 'y' )
       dec2hex();
   else
       printf( " Ok, never mind ...\n" );

   exit(0);
}

showprompt()
{
   printf( "\nPlease enter 'y' for yes, or 'n' for no.\n");
}

dec2hex()
{
   int i;

   printf( "\n.Dec..Hex.    .Dec..Hex.\n");

   for( i=0; i<16; ++i )
       printf( "  %02d   %02x       %02d   %02x\n",
           i, i, (i+16), (i+16) );

}
```

Example #1 - No Args

```
p85
 Do you want a decimal to hex table?

 Please enter 'y' for yes, or 'n' for no.
 y[enter]

 .Dec..Hex.    .Dec..Hex.
   00   00       16   10
   01   01       17   11
   02   02       18   12
   03   03       19   13
   04   04       20   14
   05   05       21   15
   06   06       22   16
   07   07       23   17
   08   08       24   18
   09   09       25   19
   10   0a       26   1a
   11   0b       27   1b
   12   0c       28   1c
   13   0d       29   1d
   14   0e       30   1e
   15   0f       31   1f
```

Functions with arguments

- Names for the arguments must be listed within the parentheses following the function name, separated by commas.
- Each argument must be declared following the right parenthesis and before the opening {.

```
sendmsg( buffer, buflength )
char buffer[ ];
int buflength;
{
```

- The function can use and/or modify its arguments just as though they were declared as local variables.

Functions with arguments

- Only the *value* of each of the caller's arguments is passed to the called function in the arguments. So if a function needs to change the caller's copy of the variable, the caller must give the address of the variable to the function (i.e. scanf()).
- The programmer must be sure that both the number of arguments and the types of the individual arguments agree both in the function definition and all of the function invocations. Programs like *lint* check this for you.

ANSI-C function declarations

The ANSI-C specification provides a means for checking argument type consistency called *function prototypes*. The function declaration syntax is extended to include declarations of the arguments. As an example of the use of this feature, consider the following code fragment:

```
int n;
double rootn, sqrt();

rootn=sqrt(n);
```

While this code initially looks correct, it will produce unpredictable results because the *sqrt()* function expects to be passed an argument of type *double*. Without the function prototype extension, you would have to find this error yourself or by using *lint*.

ANSI-C function declarations

However, if we use a function prototype the compiler can generate code to convert the integer to a double.

```
int n;
double rootn, sqrt(double);

rootn=sqrt(n);
```

The function definition also changes for consistency:

```
sqrt(double z)
{
    ...
}
```

Functions Returning a Value

The easiest way for a function to return information back to its caller is for the function itself to return a value.

- When a *return* statement is reached, the program goes back to where the function was originally called. The syntax is:

```
return ;                    /* return nothing */
return expression;          /* return something */
return ( expression );      /* return something */
```

- If the function has nothing to return to its caller, it can either simply "fall out the bottom" or use the first form of the return statement above.

- The function can send a value back to the caller by using the second form above. Whatever the expression evaluates to will be sent back to the caller and will logically replace the call of the function.

- The calling program *can* ignore the values that functions return.

Example #1

```
#include <stdio.h>

main()
{
   char str[80];

   printf( "Enter any string\n" );
   scanf( "%s", str );
   printf( "The string '%s' has %d characters.\n ",
                                  str, stlen(str) );
}

stlen(rope)
char rope[ ];
{
   int length;

   for( length=0; rope[length]; ++length );
   return (length);
}

p812
Enter any string
anystring[enter]
The string 'anystring' has 9 characters.
```

Example #2

```
main()
{
   static char prompt[ ] = "Gimme a mixed string\n";
   char inputline[80];
   int pos;

   Giveprompt( prompt );   /* assumes no value returned */
   scanf( "%s", inputline );
   pos = FindPunct( inputline );
   if ( pos>=0 )
       printf( "The first punctuation is character #%d.\n", pos);
   else
       printf( "No punctuation was found.\n" );
}

Giveprompt(stuff)
char stuff[ ];
{
   printf( "%s", stuff );
   return;  /* no return value */
}
```

Example #2

```
FindPunct(pile)
char pile[ ];
{
   int pos, pun;

   for( pos = 0, pun = -1; pun == -1 && pile[pos] != 0; ++pos )
       switch( pile[pos] )
       {
           case '.': case ';': case ',': case ':': case '?': case '!':
               pun=pos;
               break;
       }
   return( pun );
}
```

```
p813
Gimme a mixed string
asdbn.iirr?
The first punctuation is character #5.
```

Functions other than type int

Functions must have the same type as the value they return. Unless declared otherwise, functions are assumed to be of type int. If a function is to return another type, it must declared to do so before its use in the calling function as well as in the function definition.

Functions other than type int

```
#include <stdio.h>
main()
{
    double  frac(), fnum;

    fnum = 3.1423423;
    printf( "The fractional part of %g is %.8g \n",
            fnum, frac( fnum ) );
    printf( "And the next char, in upper case, is '%c'\n",
            GetUChar() );
}

double frac(thing)
double thing;
{
    return( thing - (int)thing );
}

char GetUChar()
{
    char it;
    printf( "gimme a char\n" );
    it = toupper( getchar()) ;
    return( it );
}

p816
The fractional part of 3.1423423 is 0.1423423
gimme a char
And the next char, in upper case, is 'P'
```

Other than type int

If the function has a declared return type, then the type of any expression appearing in a return statement must be convertible to that type by assignment, and that conversion in fact happens on return.

Any expression appearing in a return statement will be converted to the function's return type. For example, in a function declared as returning type int, the statement

```
return  42.174;
```

is equivalent to

```
return  (int) 42.174;
```

both of which are equivalent to

```
return  42;
```

Address Operator & and lvalues

The C language sends arguments to functions using "call-by-value". So the only way for a function to modify one of its caller's variables is for that function to be given the *address* of that variable.

- The address operator, & , when followed by a variable name, gives the address of that variable. For example, the expression

```
&house
```

  is the address of the variable named house.

- The variable following the & must be an *lvalue*. An *lvalue* is an expression that refers to an object in such a way that the object may be altered as well as examined. You can think of the 'l' as meaning that an *lvalue* is anything that can be on the left side of an assignment operator.

Example

```
main()
{
   static char it = 'z';
   static int bunch[5] = {1, 3, 5, 7, 9};

   printf( "it = '%c', &it = %d\n", it, &it );
   printf( "bunch[0] = %d, &bunch[0] = %d\n",
                              bunch[0], &bunch[0] );
   printf( "bunch[3] = %d, &bunch[3] = %d\n",
                              bunch[3], &bunch[3] );
}
```

```
p819
it = 'z', &it = 131232
bunch[0] = 1, &bunch[0] = 131480
bunch[3] = 7, &bunch[3] = 131492
```

Pointers

- A pointer is simply a symbolic representation of an address.
- When the & operator is used to determine the address of a variable *fred*, then the expression *&fred* is a "pointer to fred".
- C provides pointer variables that can hold an address just like a int variable can hold an integer. If we give a particular pointer variable the name fptr, then

  ```
  fptr = &fred;   /* assigns fred's address to fptr */
  ```

  Now, fptr is said to point to fred.
- The difference between *fptr* and *&fred* if that the former is a variable and the latter is a constant.
- A single pointer variable can be set and reset to any address.

Declaring Pointers

For any type *T*, a pointer type *pointer to T* can be used. A value of a pointer type is an address of an object of type *T*. The declaration syntax is

```
type *var;
```

where `type` is a C datatype and *var* is a variable name. So, for example, to declare a variable *ip* to be a *pointer to int* and another, *cp* to be a *pointer to char*, we could use something like:

```
int  *ip;
char *cp;
```

Yes, the * symbol is the same one used for multiplication. It is used both to declare pointers as well as in their use.

The Indirection Operator *

- The indirection operator, * , when followed by a pointer, fetches the value stored at the pointed-to address. This is called de-referencing the pointer.

- When de-referencing a pointer, the size (type) of the value fetched depends upon the datatype of the pointer.

- If we know that a pointer-to-int variable *finger* points to the int variable *nose*, then the indirection operator can be used to find the value of what is stored at *nose*.

```
int nose, schnozz, *finger;

nose = 109;            /* value of nose */
finger = &nose;        /* pointer to nose */
schnozz = *finger;     /* assign to schnozz the value of
                          what finger points to (nose) */
```

Using The & and * Operators

```
main()
{
   char eye;
   char jay;

   eye = 'i';
   jay = 'j';

   printf( "Before the swap, eye='%c', jay='%c'.\n",
                                      eye,jay );
   swap( &eye, &jay );

   printf( "After the swap, eye='%c', jay='%c'.\n",
                                      eye,jay );
}


swap( a, b )
char *a, *b;
{
   char temp;

   temp = *a;
   *a = *b;
   *b = temp;
}

p823
Before the swap, eye='i', jay='j'.
After the swap, eye='j', jay='i'.
```

An Introduction to the C Programming Language

Class 9

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

- More about arrays
- All about pointers to everything
- Multi-dimensional arrays

Arrays and Pointers

Since an array name is actually the beginning address of the consecutive memory locations making up the array, we can think of array names as a sort of pointer variable.

```
int bunch[ ] = {1, 2, 3, 4, 5};
...
if( bunch == &bunch[0] )  printf("YES!");
```

Both `bunch` **and** `&bunch[0]` **are constants which can be assigned to variables, but they themselves can not be changed.**

However, do not forget that the * operator binds more tightly than most others. So `*bunch + 2` **is two added to the value of the first element of the array, while** `*(bunch + 2)` **refers to the value of the third element of the array.**

So, it follows that all arrays and their elements can be accessed through pointers and this is in fact the way C compilers manage arrays.

Arrays and Pointers

```
/* A program to calculate the total number of days */
/* of the year that have passed at the end of each */
/* month (in a non-leap year).                    */

main()
{
   static int days[ ] = {31, 28, 31, 30, 31, 30,
                         31, 31, 30, 31, 30, 31};
   int daysofar[12], i;
   int *daysptr, *sofarptr;


   daysptr  = days;
   sofarptr = &daysofar[0];

   for( i=0; i<sizeof days/(sizeof (int)); ++i)
   {
       if (i == 0)
           *sofarptr++ = *daysptr++ ;
       else
           *sofarptr++ = *daysptr++ + daysofar[i-1];

       printf( "month %2d has %2d days, %3d days so far\n",
           (i + 1), days[i], daysofar[i] );
   }
}
```

Arrays and Pointers

```
p93
month  1 has 31 days,  31 days so far
month  2 has 28 days,  59 days so far
month  3 has 31 days,  90 days so far
month  4 has 30 days, 120 days so far
month  5 has 31 days, 151 days so far
month  6 has 30 days, 181 days so far
month  7 has 31 days, 212 days so far
month  8 has 31 days, 243 days so far
month  9 has 30 days, 273 days so far
month 10 has 31 days, 304 days so far
month 11 has 30 days, 334 days so far
month 12 has 31 days, 365 days so far
```

Passing Arrays to Functions

When a function is called that needs an array as its argument, the name of the array is used. Therefore, we are passing the address of the array to the function, rather than copying the whole array into a storage location that is local to the called function.

Because of this, functions called with array arguments can declare their formal parameters either as arrays of unknown length or as pointers to an item of the data type of the array.

Passing Arrays to Functions

```
#define   SIZE   5
main()
{
   static int scores[SIZE] = {80,90,30,92,88};

   printf( "high = %d, low = %d.\n", high(scores), low(scores));
}

high(list)
int list[ ];
{
   int i, high;
   for(i=0, high=0; i<SIZE; ++i)
       if (list[i] > high)  high = list[i];
   return(high);
}

low(list)
int *list;
{
   int i, low;
   for(i=0, low=101; i<SIZE; ++i)
       if (list[i] < low)  low = list[i];
   return(low);
}

p96
high = 92, low = 30.
```

Passing Arrays to Functions

The same program can be written without any array-like references other than the declarations. This convenience is due to the fact that arrays are actually implemented as pointers.

In general, any declaration of an array results in the compiler producing a pointer variable with the declared name and type, and setting its value to the beginning address of the allocated space. However, this pointer variable isn't variable at all - it is an "address constant".

Array names can be best thought of as just numbers that can be conveniently used as pointers.

Similarly, but unportably, statements like

```
if ( *(0x005cL << 4) )
    /* the interrupt vector is defined */
    ...
```

can be used to look directly into storage at predefined or "magic" memory locations.

Passing Arrays to Functions

```
#define   SIZE   5

main()
{
   static int scores[SIZE] = {80,90,30,92,88};

   printf( "high = %d, low = %d.\n", high(scores), low(scores));
}

high(list)
int *list;
{
   int i, high;

   for(i=0, high=0; i<SIZE; ++i)
       if (*(list+i) > high)  high = *(list+i);
   return(high);
}

low(list)
int *list;
{
   int i, low;

   for(i=0, low=101; i<SIZE; ++i)
       if (*(list+i) < low)  low = *(list+i);
   return(low);
}

p98
high = 92, low = 30.
```

strcpy() Uses Pointers

The *strcpy()* function uses pointers in registers to quickly copy strings and also gives its first argument as it's return value.

```
strcpy(left,right)
register char *left, *right;
{
    char *leftstart;

    for( leftstart=left; *left++ = *right++; );

    return leftstart;
}
```

Pointer Operations

There are five basic operations that can be performed on pointers.

1. Assignment : assigning a particular address to a pointer.
2. Value-finding : finding the value of what a pointer points to, i.e. dereferencing.
3. Take a pointer address : Since the pointer itself is a variable we can find out the address of the pointer.
4. Increment & decrement : These operations add or subtract from the pointer the size of the kind of element to which the pointer points.
5. Differencing : The difference between two pointers can be determined, with the results being in the same units as the type size.

Pointer Operations Example 1

```
main()
{
   static int bag[ ] = {1, 3, 9, 16, 25};
   int *ptr1, *ptr2;

   ptr1 = bag + 1;      /* assignments */
   ptr2 = &bag[4];


   printf ("ptr1=%u, *ptr1%d, &ptr1=%u\n", ptr1, *ptr1, &ptr1);
   /* move over */
   ++ptr1;
   printf ("ptr1=%u, *ptr1%d, &ptr1=%u\n", ptr1, *ptr1, &ptr1);
   /* back up */
   --ptr1;
   printf ("ptr1=%u, *ptr1%d, &ptr1=%u\n", ptr1, *ptr1, &ptr1);

   printf ("ptr2=%u, *ptr2%d, &ptr2=%u\n", ptr2, *ptr2, &ptr2);
   /* past the end */
   ptr2++;
   printf ("ptr2=%u, *ptr2%d, &ptr2=%u\n", ptr2, *ptr2, &ptr2);

   /* find the difference, in units of sizeof(int) */
   printf( "ptr2-ptr1 = %u\n", ptr2-ptr1 );
}
```

```
p911
ptr1=244, *ptr1=3, &ptr1=3616
ptr1=246, *ptr1=9, &ptr1=3616
ptr1=244, *ptr1=3, &ptr1=3616
ptr2=250, *ptr2=25, &ptr2=3614
ptr2=252, *ptr2=1926, &ptr2=3614
ptr2-ptr1 = 4
```

Pointer Operations Example 2

```
main()
{
   static char name[4]="EVE";
   static int dates[2]={1981,1962}, height=65, *intptr;
   static float weight_in_grams = 2.42506E5;
   char *nameptr, *endnameptr, **ptrnameptr;
   float *weightptr;

   nameptr = &name[0];
   endnameptr = nameptr+strlen(nameptr);
   intptr = dates;
   weightptr = &weight_in_grams;
   ptrnameptr = &nameptr;

   printf (" name=%d, name=>'%s'\n", name, name);
   printf (" nameptr=%d, nameptr=>'%s'\n",
            nameptr, nameptr);
   printf (" ptrnameptr=%d, *ptrnameptr=%d, **ptrnameptr=%d\n",
            ptrnameptr, *ptrnameptr, **ptrnameptr);
   printf (" weight_in_grams=%g, weightptr=%d, ",
            *weightptr, weightptr);
   printf (" ++weightptr = %d\n", ++weightptr);
   printf (" intptr=%d, intptr=>%d\n", intptr, *intptr);
   ++intptr;
   printf (" intptr=%d, intptr=>%d\n", intptr, *intptr);

}
p912
 name=100, name=>'EVE'
 nameptr=100, nameptr=>'EVE'
 ptrnameptr=108, *ptrnameptr=100, **ptrnameptr=197
 weight_in_grams=242506, weightptr=112, ++weightptr = 120
 intptr=104, intptr=>1981
 intptr=106, intptr=>1962
```

Pointer Operations Example 2

Memory layout

100	101	102	103	104	105	106	107
E	V	E	000	1981		1962	
name				dates			

108	109	110	111	112	113	114	115
100		65		2.42506E5			
nameptr		height		weight_in_grams			

116	117	118	119	120	121	122	123
112		103		108		104	
weightptr		endnameptr		ptrnameptr		intptr	

Multi-dimensional Arrays

In C, arrays of many dimensions are stored in row-column order. This means that the right-most index varies the fastest as you move through the memory of the array.

Pointers can be used with arrays of any dimension, but they access the array as though it was a long vector, or singly-dimensioned array.

Multi-dimensional Arrays

Example 1

```
int fingers [2][5] = { {1,2,3,4,5},{6,7,8,9,10} };

main()
{
   int *p;
   int i,j;

   for(i=0; i<2; i++)
   {
       for( j=0; j<5; j++)
           printf ("fingers[%d][%d] = %d\n",
                        i, j, fingers[i][j]);
       printf ("\n");
   }

   for( p = fingers;
        p<(&fingers[0][0] + sizeof fingers / sizeof(int) );
        ++p )

      printf ("p=%u, *p=%2d, &p=%u\n",  p, *p, &p);
}
```

Multi-dimensional Arrays

Example 1 results

```
fingers[0][0] = 1
fingers[0][1] = 2
fingers[0][2] = 3
fingers[0][3] = 4
fingers[0][4] = 5

fingers[1][0] = 6
fingers[1][1] = 7
fingers[1][2] = 8
fingers[1][3] = 9
fingers[1][4] = 10

p=124, *p= 1, &p=3504
p=126, *p= 2, &p=3504
p=128, *p= 3, &p=3504
p=130, *p= 4, &p=3504
p=132, *p= 5, &p=3504
p=134, *p= 6, &p=3504
p=136, *p= 7, &p=3504
p=138, *p= 8, &p=3504
p=140, *p= 9, &p=3504
p=142, *p=10, &p=3504
```

Multi-dimensional Arrays

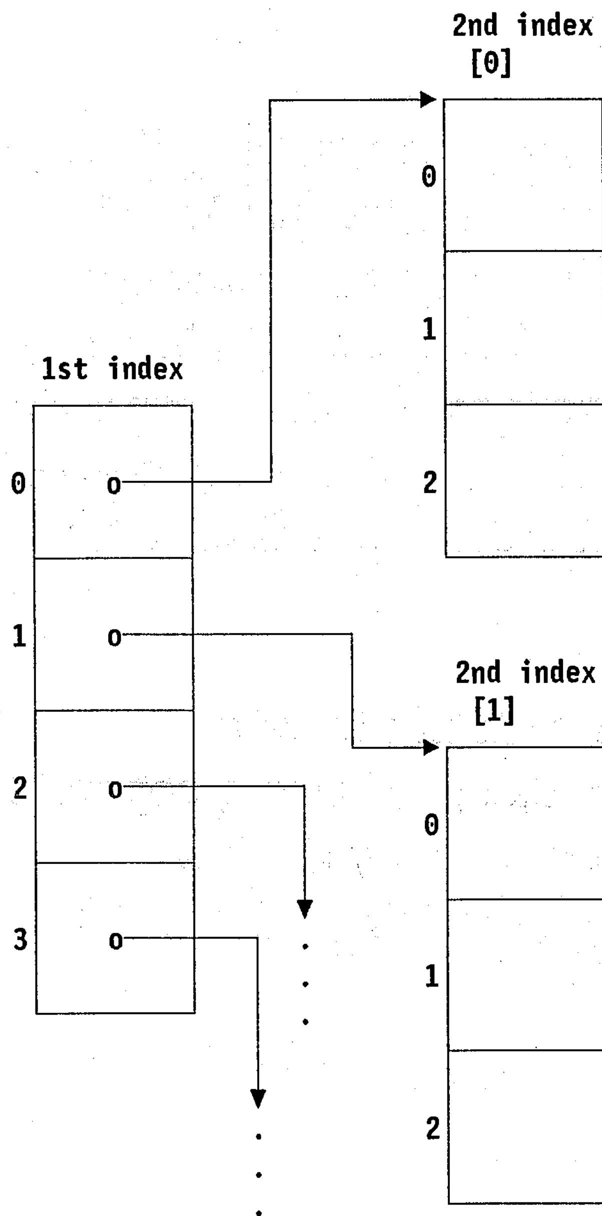
If `fingers` is the name of our two-dimensional array, can we refer to the individual rows as distinct vectors?

Since multi-dimensional arrays are actually "arrays of arrays of ...", a convenient representation of them is to have the first index be a displacement into an array of pointers. These pointers, in turn, point to an array of the next dimension's elements which will either be actual elements, in the case of two dimensions, or another array of pointers for higher dimensions.

So, if we want to access a given row of a two-dimensional array as though it was a separate array, we can do this by using only the first subscript to obtain a pointer to the beginning of that row.

Multi-dimensional Arrays

Memory layout for array[4] [3]



Multi-dimensional Arrays

Example 2

```
/* fingers is a 2 element array of pointers to int */
int fingers [2][5] = { {1,2,3,4,5},{6,7,8,9,10} };

main()
{
   int *p;
   int i;

   printf( "fingers[0]=%u, *fingers[0]=%d, &fingers[0]=%u\n",
           fingers[0],*fingers[0],&fingers[0]);
   p = fingers[1];
   printf( "Here is the second row: ");
   for( i=0; i<5; ++i)
      printf( "%d%c", *(p+i), (i<4 ? ',' : '\n') );
}
```

```
p919
fingers[0]=158, *fingers[0]=1, &fingers[0]=158
Here is the second row: 6,7,8,9,10
```

Note that `p = fingers[1]` is valid since both sides are of type “pointer to int”, whereas `p = fingers` would cause an error because `p` is type “pointer to integer” and `finger` is is type “array of pointers to integers”.

Functions & Multi-dimensional Arrays

To pass multi-dimensional arrays as arguments to functions, the receiving function must know (1) that it is getting an array, and (2) how to break up the array into dimensions.

Because of this, the function must specify the last `n-1` of the `n` dimensions of the passed array.

Functions & Multi-dimensional Arrays

Example

```
main()
{
   static int junk[5][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12},
                             {13,14,15,16}, {17,18,19,20} };
   double avg_row();
   int i;

   for( i=0; i<5; ++i)
       printf( "the average for row %d is %f\n",
           i, avg_row(junk,i));
}

double avg_row(arr,n)
int arr[ ] [4];
int n;
{
   int i, ans;
   for( i=0, ans=0; i<4; ++i)
       ans += arr[n][i];
   return ((double)ans / 4.0);
}

p921
the average for row 0 is 2.500000
the average for row 1 is 6.500000
the average for row 2 is 10.500000
the average for row 3 is 14.500000
the average for row 4 is 18.500000
```

An Introduction to the C Programming Language

Class 10

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

IBM

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

- ★ **Character strings and pointers**
- ★ **String-oriented I/O**
- ★ **Standard string functions**
- ★ **Command-line arguments**

Character Strings

A character string is simply an array of type *char* that has at least one element set to '*\0*'.

Character strings can be defined in several ways:

- string constants
- char arrays
- char pointers
- arrays of character strings

string constants

Whenever the language processor runs into something contained within double quotes, that something is recognized as a string constant.

- The enclosed characters, with a '\0' always added to the end, are stored in adjacent memory locations
- To get a double quote character in a string, it should be preceded with a backslash.
- Character string constants are placed in storage class static.
- The whole quoted string acts as a pointer to where the string is really stored.
- The string constant can be used as though it was a variable, in that its contents can be changed using pointers. However, if you need a variable you should *use* a variable to to enhance maintainability.

string constants example

```
main()
{
   printf( "This is a character string\n" );

   printf( "As is this \"%s\"\n", "Hi there!" );

   printf( "There are many ways to use a string constant\n" );

   printf( " %s, %u, %c\n" , "bananas", "are", *"hairy" );

}
```

```
p104
This is a character string
As is this "Hi there!"
There are many ways to use a string constant
 bananas, 186, h
```

Character Arrays

We can define an array of type char and use it as a character string.

The compiler must know how big the array is, so it must either be explicitly stated, or it can be initialized with a string constant rather than using the standard array initialization form (remember that only static or extern/global arrays can be initialized).

```
char easy[ ] = "userid";
char hard[ ] = { 'u','s','e','r','i','d','\0' };
```

In the first example, the trailing null is generated automatically by the string constant. In the second, the trailing null must be explicitly given. The first method is nearly always preferred for its ease of understanding and maintenance.

Character Arrays

If the storage for an array is given, the array can still be initialized using either of the above methods. However, keep these two points in mind:

- The number of elements in the array must be at least one more than the number of characters in the initialization string.
- As in other static or external/global variables, any uninitialized elements are automatically initialized to '\0'.

Character Pointers

If a character array has been defined and there is at least one *null* character in it, then the whole thing can be treated as a character string. If only a part of this string is wanted, a pointer can be set to some arbitrary position within the array. As long as there is a *null* eventually following it, that pointer is "a pointer to a character string".

Character Pointers Example

```
#define  Skip2nb(p)   while( *p == ' ' ) ++p
#define  Skip2ws(p)   while( *p != ' ' && *p != '\0' )  ++p

main()
{
   static char junk[ ] = "This  is    a  sentence...";
   char *csp, *nsp;

   csp = junk;
   while( *csp != '\0' )
   {
       Skip2nb(csp);
       nsp = csp;
       Skip2ws(nsp);
       if( *nsp == '\0' )
           break;
       *nsp = '\0';
       printf( "the next word is \"%s\"\n", csp );
       csp = nsp + 1;
   }
   printf( "the last word is \"%s\"\n", csp );
}
```

```
p108
the next word is "This"
the next word is "is"
the next word is "a"
the last word is "sentence..."
```

How not to copy strings

Strings can *NOT* be copied by simple assignment.
A loop of some sort must be used.

```
main()
{
   static char *wmsg = "Be careful, fatfingers!";
   static char *wptr;

   wptr = wmsg;
   printf( "string is \"%s\"\n", wptr );

   printf("wmsg=%s, value=%u, &wmsg=%u\n",wmsg,wmsg,&wmsg);
   printf("wptr=%s, value=%u, &wptr=%u\n",wptr,wptr,&wptr);
}
```

```
p109
string is "Be careful, fatfingers!"
wmsg=Be careful, fatfingers!, value=80, &wmsg=178
wptr=Be careful, fatfingers!, value=80, &wptr=636
```

Arrays of Character Strings

Many times it is convenient to collect a number of character strings together in an array. We can either (1) define a 1-D array of pointers to character string constants, or (2) define a 2-D char array with fixed dimensions.

The first of these choices makes better use of memory because it allocates just the right amount of memory to hold the characters. The second choice will waste memory for other than the longest strings.

Arrays of Character Strings

```
main()
{
   static char *errmsgs1[5] =
             { "Extraneous brouhaha in input line",
               "Silly operator usage suspected",
               "Undefined variable, stupid",
               "So what is this?",
               "Surely you must be joking?"   };

   static char errmsgs2[5][10] =
             { "Garbage", "Silly Op", "??? Var",
               "whatisit?", "snicker!"  };

   int i;

   for( i=0; i<sizeof errmsgs1 / (sizeof (char *)); i++)
       printf( "msg=\"%s\", len=%d\n",
           errmsgs1[i], strlen(errmsgs1[i]));

   printf("\n");

   for( i=0; i<sizeof errmsgs2 / (sizeof (errmsgs2[0])); ++i)
       printf( "msg=\"%s\", len=%d\n",
           errmsgs2[i], sizeof errmsgs2[i] );

}
```

Arrays of Character Strings

```
p1011
msg="Extraneous brouhaha in input line", len=33
msg="Silly operator usage suspected", len=30
msg="Undefined variable, stupid", len=26
msg="So what is this?", len=16
msg="Surely you must be joking?", len=26
msg="Garbage", len=10
msg="Silly Op", len=10
msg="??? Var", len=10
msg="whatisit?", len=10
msg="snicker!", len=10
```

String-Oriented I/O

To accept a string as input, two things must be done:

- space must be allocated for the string
- an input function must fetch the string.

You can not expect the program to allocate space for the input "on the fly". The space must be itself an array of char. The following declaration and input *will not work*

```
char *name;
...
scanf( "%s", name );
```

This declares a char pointer, possibly initializes it to zero, accepts an input string, and happily stores it at memory location zero! The following code would allocate the string space and produce the desired results.

```
char name[81];
...
scanf( "%80s", name );
```

gets( )

The gets() library function is available on most systems. It gets characters from stdin until it finds a newline ('\n') or EOF, and returns all of them , appended with a null ('\0') at the address it was given.

```
main()
{
    char name[81];

    printf( "Enter the author's name:" );
    gets(name);
    printf( "Searching for author = '%s'\n", name );
    ...
}
```

The function also returns a char pointer set to the address of its argument, which can be used or ignored.

```
main()
{
    char name[81], *nameptr, *gets();

    printf( "Enter the author's name:" );
    nameptr = gets(name);
    printf( "Searching for author = '%s'\n", nameptr );
}
```

A stdin line echo program

```
main()
{
    char line[81], *gets();

    while( gets(line) != '\0' )
        printf( "'%s'\n", line );
    printf( "(EOF)\n" );
}

p1015  <p1015.c
'main()'
'{'
'    char line[81], *gets();'
'    while( gets(line) != '\0' )'
'        printf( "'%s'\n", line );'
'    printf( "(EOF)\n" );'
'}'
(EOF)
```

puts()

As you would expect, there is another function for the output of strings: puts(str). It expects a pointer to a character string as its argument and returns nothing. It is particularly handy for messages because it appends a newline ('\n') to the output.

Here's how it might be defined:

```
myputs(s)
char *s;
{
    while( *s != '\0' )
         putchar (*s++);
    putchar('\n');
}
```

Since it is usually a part of the system, you won't need to define it yourself, just use it.

Other Handy String Functions

- strcat(s1, s2) : takes two character strings, finds the end of the first one ('\0'), then appends the second string onto the first, starting at the first string's null.
- strcmp(s1, s2) : takes two character strings and compares their corresponding characters. It returns *zero* if the strings are the same or *non-zero*, or true, if they are not. These rather odd return values are typically the difference between the last two characters that were compared.
- strcpy(s1, s2) : takes two character strings and replaces the first with a copy of the second.
- strlen(s1) : takes a character string and returns the number of characters it contains before the first null.

Strings Example

```
#define  STAR  "*"
char *users[ ] = { "susie*swiss",
                    "johnny*edam",
                    "sven*havarti",
                    "bruce*american",
                    "brenda*brie" };

main()
{
   char inuser[9], inpswd[9], teststring[18];
   int i, nomatch;

   *inuser = *inpswd = '\0';

   /* get the userid & password */

   while( strlen(inuser) == 0 )
   {
       printf( "Enter your userid: " );
       scanf("%8s", inuser);
   }
   while( strlen(inpswd) == 0 )
   {
       printf( "Enter your password: " );
       scanf("%8s", inpswd);
   }
```

Strings Example

```
    /* assemble the test string */

    strcpy(teststring, inuser);
    strcat(teststring, STAR);
    strcat(teststring, inpswd);

    /* look for the test string in the valid id*password table */

    for( i=0; i< sizeof users / (sizeof (char *)); ++i)
        if( !(nomatch = strcmp( teststring, users[i])) )
            break;

    if (nomatch)
        printf( "You are not allowed!\n" );
    else
        printf( "Welcome\n" );
}
```

Strings Example

```
p1018
Enter your userid:
bruce
Enter your password:
american
Welcome

p1018
Enter your userid:
brenda
Enter your password:
brie
Welcome

p1018
Enter your userid:
brenda
Enter your password:
brei
You are not allowed!
```

Command Line Arguments

Many times it is more natural for a program to accept its input from the command line, rather than having to ask for it. C provides a very standard way of handling this need.

- The function main is declared as having two arguments: an integer number of tokens in the command line, usually called argc, and an array of string pointers to the command line tokens, usually called argv.
- The programmer can assume that these arguments will be passed if they are declared.
- If used, they *must* occur in the correct order: `main(argc,argv)`. Remember the correct order by always specifying them alphabetically.
- If there are no command line arguments, these variables will be defined and set accordingly.
- The variable argc is always at least one, because the variable argv always contains at least one string, the name of the program.

Command Line Arguments Example

```
#define  STAR  "*"
char *users[ ] = { "susie*swiss",
                    "johnny*edam",
                    "sven*havarti",
                    "bruce*american",
                    "brenda*brie" };

main( argc, argv )
int argc;
char *argv[ ];
{
   char inuser[9], inpswd[9], teststring[18];
   int i, nomatch;

   *inuser = *inpswd = '\0';

   if (argc != 3)
   {
       printf( "userid & password required \n" );
       exit(100);
   }
```

Command Line Arguments Example

```
/* get the userid & password */

   for( i=0; i<8 && argv[1][i] != '\0'; ++i)
       inuser[i] = argv[1][i];

   inuser[i] = '\0';

   for( i=0; i<8 && argv[2][i] != '\0'; ++i)
       inpswd[i] = argv[2][i];

   inpswd[i] = '\0';

/* assemble the test string */

   strcpy(teststring, inuser);
   strcat(teststring, STAR);
   strcat(teststring, inpswd);

/* look for the test string in the valid id*password table */

   for( i=0; i< sizeof users / (sizeof (char *)); ++i)
       if( !(nomatch = strcmp( teststring, users[i])) )
           break;

   if (nomatch)
       printf( "You are not allowed!\n" );
   else
       printf( "Welcome\n" );
}
```

Results

```
p1022
userid & password required
R(00100);
p1022  cpalmer zxcvbn
You are not allowed!
R;
p1022  brenda brie
Welcome
R;
p1022  bruce edam
You are not allowed!
R;
p1022  bruce american
Welcome
R;
```

A More Obscure Example

Both of these programs echo their command line arguments.

```
/* echo pgm 1 */
main(argc, argv)
int argc;
char *argv[ ];
{
   int i;

   for( i=1; i<argc; ++i )
       printf( "%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}

/* -------------------------------------------------- */

/* echo pgm 2 */
main(argc, argv)
int argc;
char *argv[ ];
{
   while( --argc > 0 )
       printf( (argc>1) ? "%s " : "%s\n", *++argv );
}
```

An Introduction to the C Programming Language

Class 11

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

IBM

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Fancy Declarations

In a declaration, the basic type of the variable can be augmented by the addition of modifiers to its name:

*	adds the modifier "a pointer to"
( )	adds the modifier "a function returning"
[ ]	adds the modifier "an array of"

Since C allows the use of more than one identifier at a time, we can create many combinations of types.

The method used to "read" these compound declarations is as follows:

- The [ ] and ( ) modifiers have higher priority than *.
- Parentheses used to group parts of the expression have the highest priority. Empty parentheses indicate a function.
- "Read" the declaration from the inside out.

Fancy Declarations

```
int bored[8][8]          /* an 8 item array of
                            8 item arrays of int */

int *ipar[10]            /* a 10 item array of pointers to int */

char *flavor[4]          /* a 4 item array of pointers to char */

char (*pflavor)[4]       /* a pointer to a 4 item array of char */

char *msgs[4][5]         /* a 4 item array of
                            5 item arrays of pointers to char */

int **finger             /* a pointer to a pointer to int */

int (*tic)[3][3]         /* a pointer to a 3 item array of
                            3 item arrays of int */
```

Pointers to Functions

A variable can be declared as "a pointer to a function returning a type". An occurrence of ( ) together, with no intervening declarations, denotes "a function". Then, using the priority rules mentioned earlier, we can have declarations like the following:

```
char *yesno();            /* function returning pointer to char */

char (* terse) ();        /* pointer to function returning char */

int (*cmdprocs [5]) ();   /* array of 5 pointers to
                             functions returning int */

int (*(*ugly[5])())();    /* array of 5 pointers to
                             functions returning
                             pointers to functions
                             returning int          */
```

The proposed ANSI C standard allows arg types within the ( ) of the function definitions.

Pointers to Functions Example

```
main()
{
   int (*funptr)();                   /* funptr is a pointer to
                                         a function returning int */

   extern int (*funretfptr())();      /* funretfptr is a function
                                         returning a pointer to a
                                         function returning int   */

   funptr = (*funretfptr)(0);         /* set the pointer */

   (*funptr)();                       /* invoke the returned function
                                         via a pointer to it */

   exit(0);
}


int foop()                            /* little test function */
{
   puts ("hi there");
   return 0;
}

int (*funretfptr()) ()                /* function returning a pointer
                                         to a function returning int */
{
   printf( "funretfptr:  fp = %08x\n", foop);
   return (int (*)()) foop;
}
```

Pointers to Functions Example

```
p115
 funretfptr:  fp = 00000032
 hi there
```

Structure Type Specifiers

C provides a way to declare data types that can contain varied types of data. This type is similar to the Pascal record or the PL/1 structured data type.

A structure declaration consists of a template and a variable list.

```
struct  {                                           /* T */
            int day;                                /* E */
            int month;                              /* M */
            int year;                               /* P */
            int daynumber;                          /* L */
            char dayofweek[10];                     /* A */
            char monthname[10];                     /* T */
        }                                           /* E */
        birthdate, marriage_date;                   /* var list */
```

This declares the variables birthdate and marriage_date to consist of 4 integers and two 10-character arrays.

Structure Type Specifiers

If desired, a structure can be given a tag, or name, thus allowing the same structure to be used in declarations simply by name. This can save some typing as well as making it possible to have globally defined structure templates or to place structure templates in a header file that can be included in the source files that need it.

```
struct  date  {                                     /* T */
            int day;                                /* E */
            int month;                              /* M */
            int year;                               /* P */
            int daynumber;                          /* L */
            char dayofweek[10];                     /* A */
            char monthname[10];                     /* T */
        } ;                                         /* E */
...

struct date birthdate, marriage_date;               /* var list */
```

Structure Type Specifiers

Initialization

If a structure variable is external/global or static, it can be given initial values. The scope of a structure variable depends on where the *variable* is defined, not where the *template* is defined.

```
/* global scope */
struct graddate {
        int mm;
        int dd;
        int yy;
        char degree[4];
        } BSdate = { 5, 21, 77, "BS" };

   main()
   {
       /* local scope */
       static struct graddate MSdate = { 8, 21, 86, "MS" };
       ...
```

Structure Components

To gain access to the "insides" of structures, we use yet another operator, the structure member operator '.' . To use a particular member of a structure variable, you use the name of the structure variable (not the template), followed by '.', followed by the name of the desired member inside the structure template. You can then use this rather longish name anywhere you could use a plain variable of the same type as the structure member.

```
struct gradate {
            int mm;
            int dd;
            int yy;
            char degree[4];
            } MSdate = { 8, 21, 86, "MS" };
...
printf( "She received her %s degree on %02d/%02d/%02d\n",
        MSdate.degree, MSdate.mm, MSdate.dd, MSdate.yy );
```

Structure Components

- A component of a structure can have any type except "function returning ...".
- Component names within a structure must be distinct, but they may be the same as component names of other structures and may be the same as regular variables or functions.

```
struct pool { float length; float width; } hispool;

struct pooltable { int length; int width; } mypool;

double length;  char width[ ] = "width";
```

Structure Components

- **If a structure tag is defined as one of the components of a structure, the scope of the tag extends to the end of the block in which the outer structure is defined.**

```
struct  someone { char name[80];
                  char address[80];
                  struct date {
                         int mm; int dd; int yy;
                         } birthdate;
                } me;
/* ... */
struct  date  employment_date;
```

Structure Components

- **Structures within structures (nested structures) are initialized the same way that plain structures are, including the { and }.**

```
struct someone me = { "John Q. Luudii",
                      "PO Box 218",
                      { 8, 31, 1956 },
                    };
```

- **Arrays of structures are possible: struct someone group[10]. Each item in the array is a complete "someone" struct.**

```
struct someone us[2] = {
                         {"Chester G. McChew",
                          "PO Box 218",
                          { 1, 15, 84 } },

                         {"Chelsea Gyland",
                          "PO Box 219",
                          { 5, 22, 86 } }
                       };
```

Structure Components

- The sizeof a structure is the sum of the storage required to store all its components and whatever padding of unused space is required by the compiler or machine architecture. You may not always assume that structure members are in consecutive memory locations. For example, the structure

```
struct object {
                long int length;
                long int width;
                char     color;
             };
```

would have a `sizeof (struct object)` equal to nine bytes on a PS/2 as opposed to twelve on an RT.

Structure Components

- **If a structure has various types within it, take advantage of boundary requirements if they exist.**

```
/* Use this, taking up no more than 12B */
struct object {
                 long int  length;  /* 4B boundary */
                 long int  width;   /* 4B boundary */
                 short int          /* 2B boundary */
                 char      color;   /* 1B boundary */
              };

/* ... instead of this, taking up to 16B */
struct object {
                 char      color;   /* 1B boundary */
                 long int  length;  /* 4B boundary */
                 short int          /* 2B boundary */
                 long int  width;   /* 4B boundary */
              };
```

Structures Example

```
#define    MAX      5
#define    FEMALE   'f'
#define    MALE     'm'

struct ename { char last[40];  char mi;  char first[20]; };
struct IBMer {
           struct ename name;
           int sernumber;
           char sex;
   };

main()
{
   struct IBMer employee[MAX];
   int i,notright;
   double junk;
   char ch;

   for( i=0; i<MAX; ++i)
   {
       printf( "\n********* Last name: " );
       scanf( "%s", employee[i].name.last );
       if( *employee[i].name.last == '*' )
           break;
       *employee[i].name.last = toupper(*employee[i].name.last);

       printf( "\n        * First name: " );
       scanf( "%s", employee[i].name.first );
       *employee[i].name.first = toupper(*employee[i].name.first);
```

Structures Example

```
    printf( "\n        * middle initial: " );
    for( ch=getchar(); !(isalpha(ch)); ch=getchar() );
    employee[i].name.mi = toupper(ch);

    printf( "\n        * sex (m/f): ");
    for( employee[i].sex=getchar();
         (employee[i].sex!=FEMALE && employee[i].sex!=MALE);
         employee[i].sex=getchar()
      );

    employee[i].sernumber = i;
    ++employee[i].sernumber;  /* don't want any zeros */

    printf( "Ok, %s. %s %c. %s has serial # %d\n",
        (employee[i].sex==FEMALE ? "Ms" : "Mr" ),
        employee[i].name.first, employee[i].name.mi,
        employee[i].name.last, employee[i].sernumber) ;
    }
  printf( "no more new folks, for now\n" );
}
```

Structures Example Results

```
p1118

********* Last name: ride
        * First name: sally
        * middle initial: r
        * sex (m/f): f
Ok, Ms. Sally R. Ride has serial # 1

********* Last name: hutt
        * First name: Jabba
        * middle initial: t
        * sex (m/f): e
r
m
Ok, Mr. Jabba T. Hutt has serial # 2

********* Last name: *
no more new folks, for now
```

Pointers to Structures

Pointers to structures are declared similarly to the way in which pointers to anything else are declared:

```
struct  flavors  {
            char name[30];
            int price;
            double calories;
        } IC_menu[31];

struct  flavors  *current_choice;
```

The last operator, the indirect membership operator "–>", is used with a pointer to a structure to identify a member of that structure.

```
printf( "What flavor would you like?" );
scanf( "%s", current_choice->name );
...
printf( "Sorry, we're out of %s\n", current_choice->name );
...
```

Pointers to Structures Example

```
struct  flavors  {
            char name[10];
            int instock;
            double calories;
   };

struct flavors IC_menu[] = { {"raspberry",1,5500.},
                             {"cranberry",1,1200.},
                             {"prunberry",0, 300.},
                             {"strwberry",0,3400.},
                             {"blueberry",1,2800.},
                             {"blakberry",0,1660.} };

struct flavors SH_menu[] = { {"chocolate",0,8400.},
                             {"vanilla",  1,7200.},
                             {"huckberry",0,5400.} };

struct  flavors  *current_choice;
```

Pointers to Structures Example

```
main()
{
   char choice[10];
   int i, menusize;

   for( menusize = 0; menusize==0; )
   {
       printf( "\n what do you want?\n" );
       scanf( "%s", choice );
       if( !(strcmp( choice, "ic" )) )
       {
           menusize = sizeof IC_menu / (sizeof IC_menu[0]);
           current_choice = &IC_menu[0];
       }
       else if( !(strcmp( choice, "sh" )) )
       {
           menusize = sizeof SH_menu / (sizeof SH_menu[0]);
           current_choice = &SH_menu[0];
       }
       else
           printf ("we only have \"ic\" or \"sh\".\n");
   }
```

Pointers to Structures Example

```
    printf( "\n and what flavor do you want?" );
    scanf( "%s", choice );
    for( i=0; i<menusize; ++i, ++current_choice)
        if( !(strcmp( choice, current_choice->name )) )
            break;

    if (i >= menusize)
        printf( "\n that ain't one of our choices! \n");
    else if( current_choice->instock > 0)
        printf( "\n that'll be %e calories.\n",
                       current_choice->calories );
    else
        printf( "\n too bad, we ran out of %s\n",
                                       (*current_choice).name );
}
```

Results

```
p1121
 what do you want?
ic
 and what flavor do you want?
blueberry
 that'll be 2.800000e+003 calories.

p1121
 what do you want?
hs
 we only have "ic" or "sh".
 what do you want?
sh
 and what flavor do you want?
banana
 that ain't one of ur choices!

p1121
 what do you want?
sh
 and what flavor do you want?
chocolate
 too bad, we run out of chocolate
```

Recursive Structures

Structures may *not* contain instances of themselves, but may contain pointers to instances of themselves:

```
/*   Illegal !!!   */

struct CardCatEntry {
            char[80] title;
            char[80] author;
            struct CardCatEntry OtherBooks;
            };


/*    just fine    */

struct CardCatEntry {
            char[80] title;
            char[80] author;
            struct CardCatEntry *OtherBooks;
            };
```

Hairy Example: TREESORT

```
#define  MAXNODES  20

struct  tnode  {
                   char word[20];
                   struct tnode *left;
                   struct tnode *right;
               } forest[MAXNODES];

main()
{
   char nextword[20];
   int i;

   printf( "Let me alphabetize up to %d words\n", MAXNODES );
   printf( "Enter '.' for a word if you have less than that.\n" );

   treeclear( forest );

   for( i=0; i<MAXNODES ; ++i)
   {
       printf( "Enter the next word\n" );
       gets(nextword);
       if (*nextword == '.')
           break;
       else
           treein( nextword, forest );
   }

   printf( "Here's the sorted list of words :\n" );
   treeout(&forest[0]);

   printf( "\nBye now\n" );
}
```

Hairy Example (con't)

```
static int nextslot = 0;  /* known only from this point on */

/* treeclear - initializes the tree structure */

treeclear (tree)
struct tnode tree[ ];
{
   int i;
   for( i=0; i<MAXNODES; ++i)
   {
       tree[i].word[0] = '\0';
       tree[i].left = 0;
       tree[i].right= 0;
   }
}
```

Hairy Example (con't)

```
/* treein - inserts a new word into the tree */
/*          in the appropriate place.        */

treein (newword, treenode)
char newword[ ];
struct tnode *treenode;
{
   if( *treenode->word == '\0' )
   {
       strcpy( treenode->word, newword );
       ++nextslot;
   }
   else
   {
       if( strcmp( newword, treenode->word ) == 1 )
       { /* new word greater than current */
           if ( treenode->right == 0 )
               treenode->right = &forest[nextslot];

           treein( newword, treenode->right );
       }
       else if( strcmp( newword, treenode->word ) == -1 )
       { /* new word less than current */
           if ( treenode->left == 0 )
               treenode->left = &forest[nextslot];

           treein( newword, treenode->left );
       }
   }
}
```

Hairy Example (con't)

```
/* treeout - traverses the tree in an inorder   */
/*           fashion, printing the sorted tree. */

treeout (treenode)
struct tnode *treenode;
{
   if( treenode->left != 0 )
       treeout( treenode->left );
   printf( "%s\n", treenode->word );
   if( treenode->right != 0 )
       treeout( treenode->right );
}
```

Hairy Example Results

```
p1125
Let me alphabetize up to 20 words
Enter '.' as the last word if you have less than that.
Enter the next word
C
Enter the next word
is
Enter the next word
an
Enter the next word
algebraic
Enter the next word
programming
Enter the next word
language
Enter the next word
.
Here's the sorted list of words :
C
an
algebraic
is
language
programming

Bye now
```

Union Type Specifiers

This specification allows you store different data types in the same memory space.

- The union type specification of C can be compared to the EQUIVALENCE statement of FORTRAN or the BASED ATTRIBUTE of PL/1.

- Unions are defined in the same format as structures: templates with optional tags and required member names that are unique within the template.

```
union  pacific {  long  engine;
                  char  coal[4];
                  double caboose;
               };
...
union pacific choo_choo;
```

Union Type Specifiers

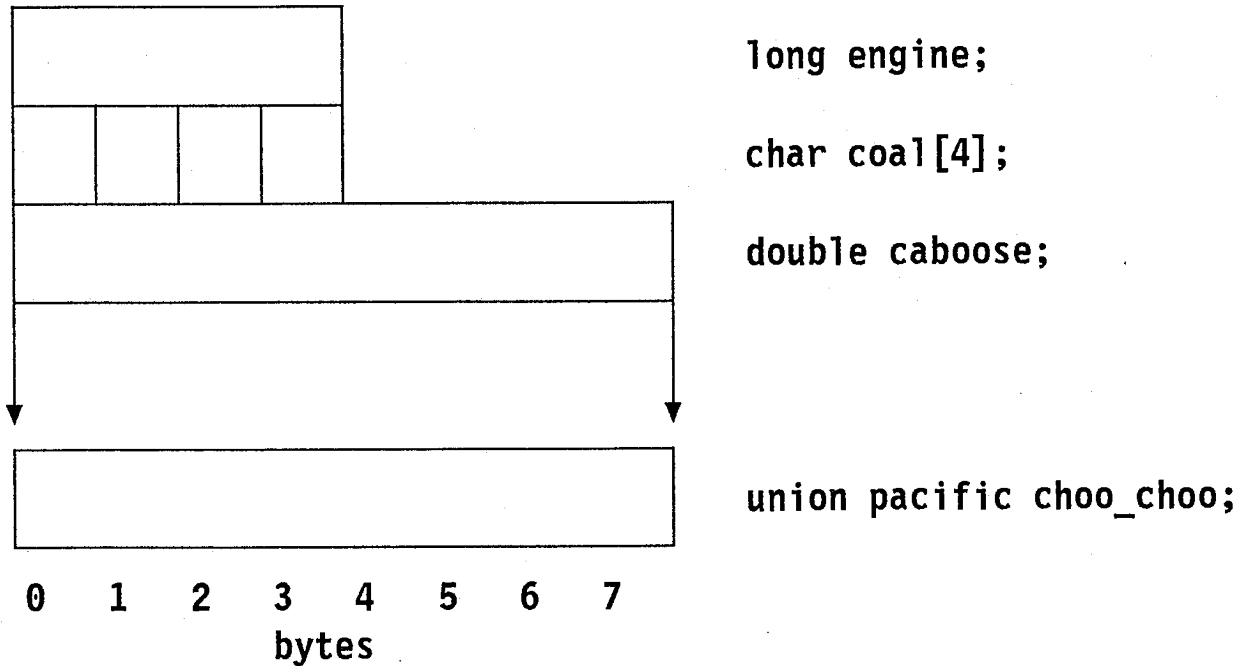
- When you declare a union variable, the compiler allots enough space to hold the largest component of the union.
- If a union variable is external/global or static, it can be given initial values. The scope of a union variable depends on where the *variable* is defined, not where the *template* is defined.

Union Type Specifiers

For this example union,

```
union  pacific {  long  engine;
                  char  coal[4];
                  double caboose;
               };
...
union pacific choo_choo;
```

the memory allocation on an IBM 370 would look like this



Union Type Specifiers

- A union variable can only contain *one* item at a time, since the components are effectively overlaid in the memory allocated to the union.
- The membership operator '.' and the indirect membership operator '->' can be used just as for structures.
- The scope of union templates and variables as well as what data types unions may contain are the same as for structures.

Union Type Specifiers

- It is up to you to remember what kind of data was last stored in a union variable. The best way is to embed the union inside a structure along with a flag variable to identify what was last put there.

```
struct unionkeeper {
            int  which;
            union  pacific {
                 long  engine;
                 char  coal[4];
                 double caboose;
              };
       };
```

Union Examples

```
union manualkey {  long   partnumber;
                   char   manualnumber[12];
                   double weight;
                };

#define    ITSA_PARTNUM   0
#define    ITSA_MANNUM    1
#define    ITSA_WEIGHT    2

struct keyunion {  int  whatitis;
                   union manualkey key;
                } DBrequest;

main()
{
    /* assigning the char array */
    DBrequest.whatitis = ITSA_MANNUM;
    strcpy( DBrequest.key.manualnumber, "GA2270009");
    printkey(&DBrequest);
}

/* function to print the union variable whatever it is */
printkey(mkey)
struct keyunion *mkey;
{
    switch(mkey->whatitis) {
         case ITSA_PARTNUM:  printf( "%dl",mkey->key.partnumber);
                             break;
         case ITSA_MANNUM:   printf( "%s",mkey->key.manualnumber);
                             break;
         case ITSA_WEIGHT:   printf( "%e",mkey->key.weight);
                             break;
         }
}
```

A Common PC Use of Unions and Structures

The PC has four general purpose 16-bit registers, AX, BX, CX, and DX. Each of these registers can be used 16-bits at a time by using the register's name. However, each is divided into two 8-bit registers called the 'high' and 'low' registers. These are referenced using the names AH & AL for AX, BH & BL for BX, etc.

If an assembly language subroutine needs or returns values for certain of these 12 registers, the following structure/union combination would be useful:

Union of Structures Example

```
struct WORDREGS {
  unsigned int ax;
  unsigned int bx;
  unsigned int cx;
  unsigned int dx;
  };

struct BYTEREGS {
  unsigned char al, ah;
  unsigned char bl, bh;
  unsigned char cl, ch;
  unsigned char dl, dh;
  };

/*          general purpose registers union          */
/* overlays the corresponding word and byte registers. */
/*                                                   */

union REGS {
  struct WORDREGS x;
  struct BYTEREGS h;
  } regs_in, regs_out;
```

Union of Structures Example

```
main()
{
    int month, day, year;

    /* get the current date */
    regs_in.x.ax = 0x2a00;/* set AX to the function code */
    regs_in.x.bx = 0x21;  /* set BX to the DOS INT number */

    interrupt21(regs_in, regs_out);

    year  = regs_out.x.cx;
    month = regs_out.h.dh;
    day   = regs_out.h.dl;

    printf ("today's date (in Europe) is %d/%d/%d\n",
             year, month, day);
}
```

Casting Comments

In past examples the cast operator has been used to force a data conversion whenever we wanted one. However, it should be noted that the cast operator is not limited to causing conversions to/from the basic data types. Any data type can be used as a cast.

```
/* floating to int */
i = (int) f;

/* from ptr to ? to ptr to int */
ip = (int *) p;

/* from ptr to ? to ptr to a mystruct structure */
mp = (struct mystruct *) sp;

/* from ptr to ? to ptr to an onion union */
op = (union onion *) sp;
```

An Introduction to the C Programming Language

Class 12

September 19-20, 1988

Charles Palmer
CPALMER at YKTVMZ
(CENET Course #IYT0040I)

T. J. Watson Research Center
Yorktown Heights, NY
Internal Use Only

Outline

- ★ Typedefs
- ★ Enumerations
- ★ Bit Fields
- ★ The C Library
- ★ File I/O
- ★ Dynamic Memory Allocation
- ★ Program Termination
- ★ What is C++?
- ★ Where to Get Help

typedef

The typedef keyword allows you to define you own name for a data type. It is similar to #define, with these three differences:

1. typedef is limited to giving symbolic names to data types only.
2. the typedef is handled by the compiler, not the preprocessor.
3. typedef is somewhat more flexible.

typedef examples

```
typedef  unsigned char BYTE;
typedef  char   *STRING;
typedef  int    fixed;
typedef  float *fptr, (*ffunc)();
typedef  struct comp { float real; float imag; } COMPLEX;

STRING  ofchars;          /* a ptr to char */
fixed point;              /* an integer */
fixed abs();              /* a function returning integer */
fptr  fp;                 /* pointer to float */
fptr  *indfp;             /* pointer to pointer to float */
ffunc sqrt;               /* pointer to function returning float */
ffunc mathsubs[10];       /* 10 element array of pointers
                             to functions returning float */

COMPLEX filter[1024];     /* 1024 element array of
                             struct comp variables  */
```

Enumeration Types

A recent addition to C, an enumeration type is a set of integer values represented by identifiers called *enumeration constants*. These constants are specified when the type is specified:

```
enum gumbotype { okra, seafood, crawfish } gumbo;
```

This declaration defines a new enumeration type *gunbo*, whose values are *okra, seafood,* and *crawfish*. A variable of this type is also declared, which can be given the specified values:

```
gumbo = crawfish;
```

Enumeration Types

The compiler implements these types by picking integer values to associate with the enumeration constants. If necessary, the programmer can specify these values in the declaration:

```
enum meltpoint { lead=120, water=32, tin=lead+5 } mpts;
```

One would expect enumeration types to enforce strict type-checking. Unfortunately, this is not the case. ANSI-C and most implementations treat all enumeration types as plain integers. As a result, enumeration constants are little more than a way to name integer constants.

Bit Fields

It is sometimes necessary, usually for machine dependent reasons, to be able to define variables of certain numbers of bits. In most cases, the programmer can make do with an integer variable and the logical operators for *shift, and, & or* operations. A typical example of such a need (probably the only justifiable one) is that of accessing system-dependent control blocks.

- The declaration of a bit-field is simply a structure whose components are rather strangely specified unsigned integers.

```
struct BitsNpieces{
  unsigned     top:1;
  unsigned        :2;  /* unused bits */
  unsigned    mask:4;
  unsigned        :0;  /* force "appropriate" boundary" */
  unsigned   extra:2;
} BitBucket;
```

Bit Fields

- The individual components can be used like other structure components, as in

```
BitBucket.top = 1;
```

- Bit field components may not be used with the & operator.
- A component name may be omitted in order to force unused space. The setting of the bits in such a space is undefined.
- An unnamed component of width zero indicates that the following component should be aligned to an "appropriate" boundary.

Example

```
/* lt.psw_ec.h>                                                   */
/*                                                               */
/*  The 370 EC mode non-XA PSW bit-field structure.              */
/*                                                               */
typedef  struct   {
  unsigned int              :  1; /*   (zero)                    */
  unsigned int PerMask      :  1; /* Pgm Event Rec Mask          */
  unsigned int              :  3; /*   (zero)                    */
  unsigned int XlateMode    :  1; /* Translation Mode (1)        */
  unsigned int IOMask       :  1; /* I/O Mask                    */
  unsigned int ExtMask      :  1; /* External Mask               */
  unsigned int PSWKey       :  4; /* PSW Storage Keys            */
  unsigned int ECMode       :  1; /* EC Mode (1)                 */
  unsigned int MCheckMask   :  1; /* Machine Check Mask          */
  unsigned int WaitState    :  1; /* Wait State                  */
  unsigned int ProbState    :  1; /* Problem State               */
  unsigned int SecSpaceMode :  1; /* Secondary Space (1)         */
  unsigned int              :  1; /*   (zero)                    */
  unsigned int CondCode     :  2; /* Condition Code              */
  unsigned int FixPtOvfl    :  1; /* Fixed Pt Overflow           */
  unsigned int DecOvfl      :  1; /* Decimal Overflow            */
  unsigned int ExpUnderfl   :  1; /* Exponent underflow          */
  unsigned int SignifMask   :  1; /* Significance Mask           */
  unsigned int              : 16; /*   (zero)                    */
  unsigned int InstrAddr    : 24; /* Instruction Address         */
  } PSW_EC;

  PSW_EC mypsw;


  if( mypsw.SignifMask )
     mypsw.SignifMask = 0;
```

Why You Should Avoid Them

- The use of bit-fields is likely to be non-portable. Their only only reasonable uses are when memory is very scarce or when a system-dependent data structure must be matched exactly.

- Some machines have a 16-bit word size, which limits the maximum width of a bit field. Other machines may have a 32-bit word size.

- The order in which different machines will pack bit fields into a word will vary. For example, on an IBM 370 the bits are packed left to right, i.e. from the most significant bit to the least. However, on an IBM PC the bits are packed in just the opposite manner.

- The use of an unnamed bit field of length zero to force an "appropriate" boundary may produce different alignments on different machines.

Use Masks Instead of Bit Fields

Most programs that need to work with bit-oriented data use a combination of *unsigned ints* and *#define* rather than bit fields. This results in more readable as well as portable code.

```
unsigned long int status;

#define ASLEEP  0x80000000
#define DOZING  0x40000000
#define TIRED   0x20000000
#define BORED   0x10000000
#define AWAKE   0x08000000
/* ... */

#define zzzz(s)   (s & (ASLEEP | DOZING) )

   if (zzzz(status))
     ...

   status &= ~ASLEEP;  /* Wake him up! */
   status |= AWAKE;
```

The C Library

Most serious C language processors provide a library of commonly used functions for the programmer. They are supplied in a combination of two ways:

- as compiled/translated functions in a library that is used at link time.
- as macros defined in header files that can be #include'd by the preprocessor, i.e. in *stdio.h*.

The C Library

A few functions can almost always be assumed to be available in one form or another:

```
                /* I/O functions we've used           */
getchar()       /* get a char from stdin              */
putchar()       /* put a char to stdout               */
gets()          /* get a line from stdin              */
puts()          /* put a line to stdout               */
scanf()         /* get formatted input from stdin     */
printf()        /* write formatted output to stdout   */
                /* string functions we've used        */
strlen()        /* find the length of a string        */
strcmp()        /* compare two strings                */
strcpy()        /* copy a string                      */
strcat()        /* combine two strings into one       */
strchr()        /* searches a string for a char       */
                /* and some others we haven't used    */
atoi()          /* convert string to int              */
atof()          /* convert string to float            */
sprintf()       /* do a printf into a string          */
```

The proposed ANSI-C standard does define a minimum library subset.

The C Library

A number of useful macros that handle the testing and conversion of characters are also usually defined for the programmer. They can usually be found in either *stdio.h* or *ctype.h*.

```
                /* returns true if c is :          */
isalpha(c)      /*    alphabetic                   */
isdigit(c)      /*    a digit                      */
islower(c)      /*    lower case                   */
isupper(c)      /*    upper case                   */
isspace(c)      /*    whitespace                   */
isalnum(c)      /*    alphanumeric (isalpha|isdigit)*/
isxdigit(c)     /*    a hexadecimal digit          */
iscntrl(c)      /*    a control character          */
ispunct(c)      /*    a punctuation character      */
isprint(c)      /*    a printable character        */

                /* returns c converted to          */
toupper(c)      /*    uppercase                    */
tolower(c)      /*    lowercase                    */
```

The C Library

There will usually be collections of other very specialized functions available:

- mathematical functions (trig, logs, etc.)
- system functions (time, date, interrupts,e etc.)
- data communications (comgetc, inport, outport, etc.)
- graphics functions (line, polygon, fill, etc.)

Since many of these special functions return values of other than type int, they must be declared as such. There will usually be some header files associated with these general groups of library functions that contain these declarations. For example, all of the math functions are declared in the file *math.h*.

File I/O

So far, all of our example programs that performed I/O to/from a file did so through our redirecting of *stdin* and *stdout*. However, this method is limited. For example, if output is redirected, then prompts written using printf() go into the file instead of to the screen.

C provides another family of library functions that handle file I/O. To use them the header file *stdio.h* must be #include'd as it contains some special declarations these functions need.

```
fopen()   /* prepares a file for I/O          */
fclose()  /* ends I/O to a file               */
getc()    /* single character file input      */
putc()    /* single character file output     */
fscanf()  /* scanf() input from a file        */
fprintf() /* printf() output to a file        */
fgets()   /* gets() input from a file         */
fputs()   /* puts() output to a file          */
fseek()   /* random access into a file        */
```

File I/O

The fopen() function returns what is called a *handle* for the opened file.

- The handle identifies the file with a system dependent data structure that contains information about the file. Examples of this would be FCB's under DOS or VM. One of these data structures is allocated for you by fopen() and a pointer to it is returned.
- The handle is one of the parameters to all of the other functions.
- Many times the handle will be of a special data type defined in *stdio.h*. As a result, the declarations for fopen() and the variable in which the handle is kept will need to be declared with this special type, usually like *FILE *fopen(), *filehandle;*.
- If the fopen() fails, it returns zero.

File I/O

stdin, stdout, stderr

- When a program is started, three "files" are opened automatically and file handles are provided for them. These files are the standard input, output, and error output, or *stdin, stdout, stderr*.
- These file pointers are defined for you in *stdio.h*.
- Although they are usually connected to the terminal, they can be treated as files since in fact they could be redirected to a file.

As an example, the *getchar()* and *putchar()* functions are usually macros defined in *stdio.h* as follows:

```
#define  getchar()     getc(stdin)
#define  putchar(c)    putc(c,stdin)
```

File I/O Example

```
#include <stdio.h>

/* concatenate the files passed as arguments, all onto stdout */

main(argc, argv)
int argc;
char *argv[ ];
{
   FILE  *fopen(), *handle;

   if (argc == 1)  /* no args, so use stdin */
       copyfile( stdin );
   else
       while ( --argc > 0 )
       {
         if ( (handle = fopen(*++argv, "r")) == NULL)
         {
           fprintf( stderr, "fopen failed on '%s'\n", *argv);
           break;
         }
         else
         {
           copyfile(handle);
           fclose(handle);
         }
       }
}

copyfile( fileptr )
FILE *fileptr;
{
   int c;

   while( (c = getc(fileptr)) != EOF )
       putc( c, stdout );
}
```

File I/O Results

```
p1218  test.h test.c  >  test.all

t test.all
/* test.h  */

#define FEE 1024
#define FIE  512
#define FOE  256
#define FOO   64

/* start a comment
#include <test.h>
end a comment     */

main(argc, argv)
int argc;
char **argv;
{
   printf( "->%d\n", FEE );
   exit(0);

}
```

Dynamic Memory Allocation

In some of our example programs, we allocated a large amount of memory to hold an input whose size we could not predict or was not constant. This is many times a waste of storage since you must allocate the largest possible memory size that you will accept.

A better approach is to allocate memory at run time, or "on the fly", when you know how much you need.

Dynamic Memory Allocation

The C library function malloc( ) takes one argument, the number of bytes, or chars, that you want. If the memory is available, it returns a pointer-to-char that is pointing to the newly allocated memory. This pointer must then be cast into whatever type of memory you need it to be. If no memory was available, this function returns NULL (zero).

The memory is not initialized in any way, so the caller must assume that it will contain garbage information.

Another function, free( ), gives back storage that was previously malloc'ed. It requires a single argument, a char pointer to the previously allocated memory block.

Dynamic Memory Allocation

```
#include <stdio.h>
#define  MAXSTRINGS   100

main()
{
   char *malloc();
   char *memp, instring[80], *strings[MAXSTRINGS];
   int i=0, j=0, size=0;

   for( ;; )
   {
       printf( "Enter the next string, or '.' to end\n" );
       gets( instring );
       if ( *instring == '.' )
           break;
       else
       {
           if (memp = malloc (strlen (instring) ) )
           {
               printf( "New memory allocated at %u.\n", memp);
               strcpy(memp, instring );
               strings[i++] = memp;
           }
           else
           {
               printf( "out of memory\n" );
               exit(1);
           }
       }
   }
```

Dynamic Memory Allocation

```
    for(j=0; j<i; ++j)  /* find out how much room we need */
        size += strlen( strings[j] );

    if( !( memp = malloc( size )) )         /* allocate it */
    {
        printf( "out of memory\n" );
        exit(1);
    }
    *memp = NULL;       /* make it a zero-length string */

    for(j=0; j<i; ++j)  /* & build up the sentence there */
    {
        strcat( memp, strings[j] );
        free( strings[j] );
    }

    printf( "here is the whole sentence: \n" );
    printf( "'%s'\n", memp );
    free(memp);
    exit(0);
}
```

Dynamic Memory Allocation

```
p1222
Enter the next string, or '.' to end
This
New memory allocated at 3782.
Enter the next string, or '.' to end
Is a
New memory allocated at 3790.
Enter the next string, or '.' to end
Test
New memory allocated at 3798.
Enter the next string, or '.' to end
of the
New memory allocated at 3806.
Enter the next string, or '.' to end
early warning system.
New memory allocated at 3816.
Enter the next string, or '.' to end
.
here is the whole sentence:
'This Is a Test of the early warning system.'
```

calloc( )

Another common library function used for dynamic memory allocation is calloc().

- It wants two arguments: the first is the number of chunks of memory you want, and the second is how many bytes are in each chunk.
- It returns a pointer-to-char (like malloc()) that points to the newly allocated block of memory.
- It returns NULL if it fails.
- The memory is *cleared*, that is, set to all zeros.
- The free() function will free calloc'ed memory also.

Program Termination

Most systems provide some way for a terminating program to communicate back to the system some indication of whether it ran okay or not. Then, using REXX (VM) or BATCH (PCDOS) or the UNIX™ shell, the system can check this 'return code' and decide what to do next.

```
/* rexx example */
'CW 'programname
if (rc <> 0)
  then say 'Compile Errors!';
else
  'LINKC 'programname
```

The C library provides a function `exit()` to set this return code. It takes one argument, the desired return code. After calling this function, the program will terminate, i.e. it does not get control back from the invocation of `exit(..);`.

If no call to exit is made, all C language processors should set the return code to zero.

What is C++ ?

C++ is a superset of C that retains the efficiency and notational convenience of C, while providing facilities for

- type checking
- data abstraction
- operator overloading
- object-oriented programming

The definitive text is "The C++ Programming Language", by Bjarne Stroustrup who designed the language.

The August 1988 issue of BYTE magazine also had a good introductory article.

Where to Get Help

The IBMPC, IBMVM, and IBMUNIX conferencing disks are an invaluable source of help with everything from programming problems to portability concerns. Currently there are several forums whose discussions are related to C:

- IBMPC
  - C-ANSI
  - C-C++
  - C-DEBATE
  - C-DEVELO
  - C-IBM
  - C-LANG
  - C-LAT
  - C-MS
  - C-MSQC
  - C-PITFAL
  - C-TURBO
- VMIBM disk
  - C-IBM370
  - C-WISH
- IBMUNIX disk
  - C

Copies may be requested from:

IBM Thomas J. Watson Research Center
Distribution Services F-11 Stormytown
Post Office Box 218
Yorktown Heights, New York 10598