UNIVERZA EDVARDA KARDELJA V LJUBLJANI
FAKULTETA ZA ELEKTROTEHNIKO

Saša DIVJAK

PROGRAMIRANJE V JEZIKU C

Delovno gradivo

LJUBLJANA 1986

UNIVERZA EDVARDA KARDELJA V LJUBLJANI
FAKULTETA ZA ELEKTROTEHNIKO

Saša DIVJAK

PROGRAMIRANJE V JEZIKU C

Delovno gradivo

LJUBLJANA 1986

II 362003a

p

20-02-1987

0_ 602

Dr. Saša Divjak, dipl. ing.,
izredni profesor na Fakulteti za elektrotehniko
v Ljubljani

PROGRAMIRANJE V JEZIKU C
Delovno gradivo

Izdala in založila Fakulteta za elektrotehniko
v Ljubljani

Pripravlja Komisija za tisk — urednik Božidar Magajna
Tisk »F. Tori«, Ljubljana
Naklada 100 izvodov

PROGRAMMING IN C LANGUAGE

HANDOUT

3.1.1. An Example C Program

- The file hello.c:

```
main()
{
  printf("Hello world!");
}
```

- An example run:

```
Hello world!
```

3.1.1.3. About C Programs

- **Function definitions start with an interface section describing how they must be called.**
- **The main function can have a very simple interface, as it is automatically called by the system when the program begins, and can ignore any program arguments.**
- **After the interface section goes the implementation section, or outer block of the function.**
- **Blocks are enclosed in curly braces, and can enclose local declarations followed by statements.**

3.1.2.1. Using Character Constants

- The file CharConsts.c:

```
main()
{
putchar('a'); putchar('b');
putchar(' '); putchar('0');
putchar('1'); putchar(' ');
putchar('|'); putchar('@');
putchar(' '); putchar('\\');
putchar('\''); putchar('\n');
}
```

The output:

```
ab   01 |@ \'
```

1-2

3.1.2.2. About Character Constants

- Characters can be represented by their numeric character codes.
- For clarity and portability, another notation is normally used, involving the single quote (') and backslash (\) characters:
- To indicate their character codes, ordinary printable characters are simply enclosed between single quotes, for example:

  'a' 'b' 'c'   '0' '1' '2'   '-'
  '+' '='

- Nonprintable character codes can be represented by using a backslash character, \, followed by one to three octal digits, e.g.:

  '\0'   for an ASCII NULL
  '\7'   for an ASCII BELL

3.1.2.3. Special Character Constants

- For additional clarity and portability, C provides:

  '\n'   to represent a newline character
  '\r'   to represent a carriage return
  '\t'   to represent a tab
  unambiguously

- Finally, since backslash (\) and single quote (') have special meanings, they are represented as follows:

  '\\'   represents a single backslash
  '\''   represents a single quote

3.1.2.4. Using Character Variables

- The file chars.c:

```
main()
{
  char digit, special;
  int letter, whoKnows;

  digit = '8';  special = '$';
  letter = 'd'; whoKnows = getchar();

  putchar(digit);       putchar(special);
  putchar(letter);      putchar(whoKnows);
  putchar('\n');
}
```

An example run:

```
q
8$dq
```

1 - 3

3.1.2.5. About Character Variables

- Since characters are represented by (implementation dependent) character codes, they can be represented by the same variables used to hold integer numbers.
- C has several datatypes to represent integers, depending on how large the numbers are expected to be.
- The datatype char may be used to hold characters which will occupy only one byte, at least the values 0 through 127.
- The datatype int should be used if an extended character set with more than 128 codes is being used.
- int should also be used if characters are being interspersed with special codes for end of file, etc., which do not use any of the 128 codes reserved for normal characters.

3.1.3.1. Integer Constants & Output

The file: IntConsts.c

```
main()
{
  printf("%d is the same as %d\n", 010, 8);
  printf("%d is the same as %d\n", 0x10, 16);

  printf("Octal: %o, Decimal: %d, Hex: %x\n",
10, 10, 10);
  printf("Octal: %o, Decimal: %d, Hex: %x\n",
010, 010, 010);
  printf("Octal: %o, Decimal: %d, Hex: %x\n",
0x10, 0x10, 0x10);

  printf("%ld is a long integer\n", 123456789);
  printf("%ld just thought it was.\n", 123L);
}
```

3.1.3.2. About Integer Constants

- Integers may be represented in octal (base 8), decimal (base 10), hexadecimal (base 16) and as character codes.
- Octal constants are represented by the digits 0-7, and must have a leading zero:

  010　　　0177245　　　0177

- Decimal constants may be represented by the digits 0-9, but must not begin with an initial zero:

  10　　　45　　　100

- Hexadecimal constants may be represented by the digits 0-9, and a-f. They must begin with an initial 0x or 0X:

  0x10　　　0xffa0　　　0xff

1-4

3.1.3.1. Integer Constants & Output

The Output:

```
8 is the same as 8
16 is the same as 16
Octal: 12, Decimal: 10, Hex: a
Octal: 10, Decimal: 8, Hex: 8
Octal: 20, Decimal: 16, Hex: 10
123456789 is a long integer
123 just thought it was.
```

3.1.3.2. About Integer Constants

- Negative integer constants begin with a minus sign:

  -010 -123 -0xff

- Integer constants which need to be passed to functions expecting long integers can be followed by an l or L for long:

  1000L 0xabcL

- Integer constants which are too large to be single precision will automatically be represented as long integers, making the l or L optional:

  10000000L or just 10000000
  0xabcdefL or just 0xabcdef

3.1.3.4. About Integer Variables

- Integer values can be represented with or without sign, and may be of various im plementation dependent sizes.
  - <u>char</u> will provide one byte of storage, which may or may not include a sign bit. Only values 0..127 are safe.
  - <u>int</u> will provide some efficient but unspecified amount of storage, usually 16 or 32 bits.
  - <u>int</u> may be qualified with the words <u>short</u>, <u>long</u> and <u>unsigned</u>.
  - A <u>short int</u> provides at least 16 bits of precision.
  - A <u>long int</u> provides at least 32 bits of precision.

1-5

3.1.3.4. About Integer Variables

- The unsigned qualifier makes available all the bits in an integer variable for representing a non-negative value.

- short, long and unsigned may be used as abbreviations for short int, long int and unsigned int.

3.1.4.1. Floating Point Constants

- Floating point constants are always given in decimal fractional or scientific notation.

- A decimal point is used to indicate the start of the fraction part.

- The letter e or E is used to indicate the start of the exponent part.

  123.45   123.456e7   0.12e-3

- All floating point constants are represented with double precision.

3.14.2. Floating Point Variables

- C provides two sizes of floating point numbers, float and double, with implementation dependent range and precision.

  - float represents the available single precision floating point numbers, usually with at least 32 bits.

  - double represents the available double precision floating point numbers, usually with at least 64 bits.

- Despite the ability to store single precision floating point values, all floating point calculations are performed using double precision arithmetic.

3.1.5.1. Numerical Input/Output: The file numbers.c:

```
main()
{
  int netWorth, numKids;
  float weight, height;

  netWorth = 14;          weight = 14.007e2;

  /* next line could be deleted w/o effect */
  numKids = 10; height = 0.1;    /* to be
trashed */

  printf("Please enter a decimal integer ");
  printf("and a floating point number:\n");

  scanf("%d %f", &numKids, &height);

  printf("worth is %d, chidren, %d, ", netWorth, numKids);

  printf("weight = %f\nand height, %f.",
weight, height);
}
```

(pass the parameter addresses)

3.1.5.2. Using Arithmetic Operators: The File "NumOps.c":

```
main()
{
  int int1, int2;
  float flt1, flt2;

  int1 = 2 + 3 - 15; /* result is negative */
  int2 = -int1;      /* result is positive */

  int1 = 2+(3*5);    /* result is 17 */
  int1 = 2+3*5;      /* result is 17 */
  int2 = (2+3)*5;    /* result is 25 */

    int1 = int2 = 0;  /* = associates  */
                      /* right to left */
  int1 = 123 / 10;    /* yields quotient 12 */
  int2 = 123 % 10;    /* yields remainder 3 */

  flt1 = 123 / 10;      /* yields 12.3 */

  flt2 = int1;          /* integer to float   */
                        /* conversion implied */
  int2 = flt1;          /* result is truncated */
}
```

3.1.5.3. Arithmetic Operators

- Integer division yields only the quotient, the remainder can be obtained with the operator %, pronounced ``mod'':

  1234.0 / 100 is
  12.34 1234 / 100 is
  12
  1234 % 100 is 34

- The operators *, / and % are of higher precedence than + and -, but parentheses can be used to clarify or alter this pre cedence, thus:

  2+3*5 is the same as 2+(3*5), not
  (2+3)*5

An Example of a C Function With One Input

The file proc.c:

```
main()
{
  decode('a');
  decode('0');
}
                                  } interface part
decode(c)
  char c;
{
  printf("The character %c has code %d.\n", c, c); } implementation part
}
```

The output:

```
The character a has code 97.
The character 0 has code 48.
```


3.1.7.1. An Example of the if Statement

The file if1.c:

```
main()
{
  printf("Do you wish instructions?");
  if ( getchar() == 'y' )
    printf("Sorry, none are available");
}
```

An example run:

```
Do you wish instructions?
y
Sorry, none are available
```

Another example run:

```
Do you wish instructions?
n
```

3.1.7.2. An if Statement with an else

The file if2.c:

```
main()
{
  char c;

  printf("Do you wish instructions?");
  c = getchar();
  if ( c == 'y' )
      printf("Sorry, none are available!\n");
  else
     {
      printf("Well, its a good thing, ");
      printf("because none are available.");
     }
}
```

3.1.7.3. Nested if Statements

The file if3.c:

```
main()
{
  char c;

  printf("Do you wish instructions?");
  c = getchar();
  if ( c == 'y' )
      printf("Sorry, none are available!\n");
  else if ( c == 'n' )
     {
      printf("Well, its a good thing, ");
      printf("because none are available.");
     }
  else
     printf("Type y for Yes or n for No\n");
}
```

3.1.8.1. True & False in C

- Logical operations in C, such as the test part of an if statement, take integer values.
- A non-zero integer value is taken to be True, or a successful test, thus:

  `if (1) printf("hello");`

  will always print hello.
- The integer value of zero is taken to be False, or failure, thus:

  `if (0) printf("goodbye");`

  will never have any effect.

3.1.8.2. The logical operators

- C has a set of logical operators, yielding either true (1) or false (0) results.
- Integer and floating point values may be compared for strict equality using the operators:

  `==` `!=`

- They may be compared for order by using the relational operators:

  `>` `>=` `<` `<=`

- Logical expressions may be combined into conjunctions and disjunctions:

  `&&` (logical and) `||` (logical or)

3.1.8.3. Precedence of the logical operators

- Precedence can always be clarified or altered with parentheses, for example, these pairs have the same meaning:

```
a > b == c > d
        (a > b) == (c>d)

a &&b || c
  (a &&b) || c

a || b &&c
  a || (b &&c)
  a < b || c == 4 &&a > b
(a< b) || ( (c == 4) &&(a > b) )
```

3.18.4. A Test Expression Example

File test.c:

```
main()
{
  int a, b, c, d, same;

  a = 1;          b = 2;
  c = 3;          d = 4;
  same = (a < b) == (c < d);
  if (a < b)
    printf("%d is less than %d\n", a, b);
  if (same)
    printf("%d and %d are ordered the same\n",
    c, d);
}
```

Output:

```
1 is less than 2
3 and 4 are ordered the same
```

3.2.1.2. Accessing Standard I/O Library Definitions

- Accompanying the functions in the standard I/O library are a set of definitions, which can be included into your file with the line:

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

- This line causes the C preprocessor to fetch the file stdio.h from a special sys tem area (/usr/include on UNIX), and include its contents at this point in the current file.
- The C compiler proper never sees any C preprocessor requests; it just sees normal C code.

3.2.1.4. Variations with printf

The file printf.c:

```
main()
{
    printf("char %c, decimal %d, octal %o,
    hex %x\n", 'a', 'a', 'a', 'a');
    printf("float %f, string %s", 14.3,
    "hello");
}
```

Output of the program:

```
char a, decimal 97, octal 141, hex 61
float 14.300000, string hello
```

Parameter Values are Copies

Function parameters are actually local variables which start out with copies of the values handed in.

Thus in the following program, file "varCopy.c":

```
main()
{
    char c;

    c = 'a';
    func(c);
    putchar(c);
}

func(c)
char c;
{
    putchar(c);
    c = 'b';
    putchar(c);
}
```

The output is:

```
aba
```

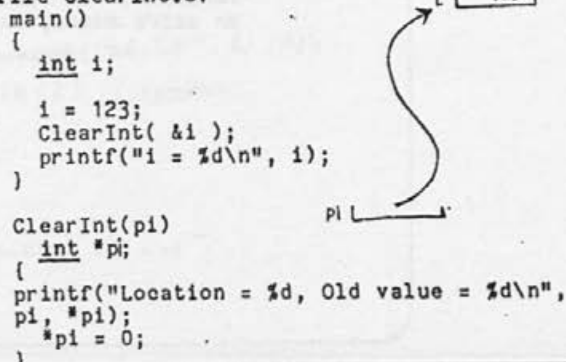

3.2.2.1. About the Standard I/O Library

- The standard I/O library provides a number of useful functions for doing I/O from C programs.
- The standard I/O library is not part of the C language proper, but should be present in any C program development environment.
- When you compile a C program, the standard library is automatically searched for any functions not defined by the files specified as arguments to the cc command.

3.2.2.2. Passing Variable Addresses

- If a function is to be able to change the values of variables, it must be passed the address of those variables.
- The prefix operator & takes the address of a variable.

The file ClearInt.c:




```
main()
{
  int i;

  i = 123;
  ClearInt( &i );
  printf("i = %d\n", i);
}

ClearInt(pi)
  int *pi;
{
printf("Location = %d, Old value = %d\n",
pi, *pi);
  *pi = 0;
}
```

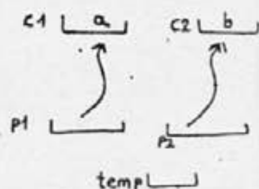
3.2.2.3. Using Multiple Function Outputs

Part of the file swap.c:

```
main()
{
  char c1, c2;

  c1 = 'a';
  c2 = 'b';
  printf("a = %c, b = %c\n", c1,
  c2);
  swap(&c1, &c2);
  printf("c1 = %c, c2 = %c\n",
  c1, c2);
}
```

3.2.2.4. Using Variable Addresses

- The rest of the file swap.c:




```
swap(p1, p2)
  char *p1, *p2;
{
  char temp;

  temp = *p1;
  *p1 = *p2;
  *p2 = temp;
}
```

- In order to use an address of, or pointer to, a variable, one must store it in a pointer variable.
- Pointer variables are declared by indicating what kind of value they may point to.

3.2.2.4. Using Variable Addresses

- In the definition of a variable, asterisks indicate that the variable in question is a pointer to an object of the given datatype. No runtime operation is implied.
- As part of a statement or an expression, the asterisk prefix operator, (*), refers to the data being pointed to by the pointer variable. At runtime, this corresponds to an extra memory fetch.

3.2.3.1. Reading Single Characters

- Characters may be read from the standard input by the function getchar.

  The file getchar.c:

```
main()
{
    char c;

    c = getchar();      /* save the 1st
char*/
    getchar();          /* chuck the 2nd
*/
    putchar( getchar() );/* print the 3rd
*/
    putchar(c);         /* print the 1st
*/
    }
```

3.2.3.2. Formatted Input

- The function scanf reads input according to the formats specified by its format string.
- scanf understands the same format codes that printf does.
- Whitespace characters (spaces, newlines and tabs) inbetween format codes in the format string will match any number of whitespace characters in the input stream. ← not true
- Any extra nonwhitespace characters in the format string are expected to literally match characters in the input stream.
- scanf will keep reading until it fills all the values called for, or until the input stream deviates from the form specified by the format string.

3.2.3.2. Formatted Input

- scanf returns as its function value the number of items successfully read, or the special value EOF (defined in <stdio.h>) if it reaches the end of the input stream before it has finished.

```
k = scanf("%d %d", &i, &j);
if (k < 2) ..... complain...
```

usually ... EOF = -1

Variations or scanf

The file scanf.c:

```
main()
{
  char c;
  int i, j;
  float f;

  printf("Please type a character, an integer");
  printf(" and a floating point number:\n");

  scanf(" %c %d %f", &c, &i, &f);
  printf("You typed %c, %d and %f\n", c, i, f);

  printf("Please type ``top 10 teams'':\n");

  i = scanf(" top %d teams", &j);
  printf("I read %d value(s), including the number %d.\n",
  i, j);
}
```

An example run:

```
Please type a character, an integer and a floating point number:
          x 123 3.14159
You typed x, 123 and 3.141590
Please type ``top 10 teams'':
top 10 teams
I read 1 value(s), including the number 10.
```

C-3-69

3.3.1.1. Printing a Numeric Digit

Part of the file PutInt.c:

```
PutDigit( d )
  char d;
  /* assertion: d is in [0..9] */
{
  putchar( d + '0' );
}
```

3.3.1.1. Printing a Numeric Digit

Part of the file PutInt.c:

```
PutPosInt( i )
  int i;
  /* assertion: i >= 0 */
{
  if (i > 0)
    {
      PutPosInt( i/10 );
      PutDigit( i%10 );
    }
}
```

4.1.1.4. Using Static Variables

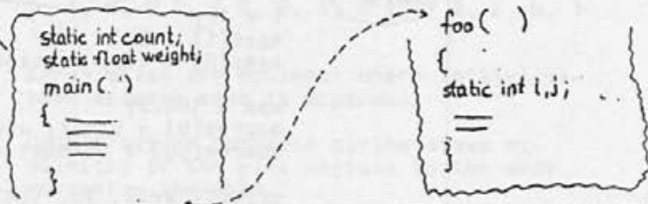
- Initialization clauses on variable definitions apply only when variable is created.
  - The file rand.c:

```
int rand()
{
static int seed = 31415;

seed := (seed + 7227) % 2345;
return seed;
}
```

- Only function rand can see the variable seed.
  - Without the keyword static, rand would always return (31415 + 7227) % 2345.

4.1.1.3. The Scope of Static Variables

- Externally declared static variables:
  - are visible through the rest of the
  - file (except within blocks which redeclare the same name)
  - do not require keyword static
- Internally declared variables (static or otherwise):
  - are visible within their block (except within nested blocks which redeclare the same name)
  - require the keyword static



4.1.2.1. Allocation of Automatic Storage

- New storage is automatically allocated when:
  - a function is called
  - a block is entered
- The same storage is automatically reclaimed when:
  - the function returns
  - control leaves the block
- Function calls & returns and block entry & exits are always nested, permitting a simple storage management scheme.

4.1.2.3. Declaring Automatic Variables

- Function parameters are automatic variables:

  example: fred (x,y)
  int x,y;

  - allocated & initialized at function call visible throughout the function body (except where name reused in nested block)
- Internal variables are automatic by default:
  - allocated at block entry
  - initially contain a random value (unless initialization clause used)
  - are visible throughout the block (except where name reused in nested block)

4.21.1. Allocating Array Variables

- The file arrays.c:

```
main()
{
char c, hold[4], name[20];
int i, ages[100];
float max, scores[100];

c = 'X';
hold[0] = 'U';      hold[1] = 'N';
hold[2] = 'I';      hold[3] = c;

printf("What's your name? ");
scanf("%s", name);

i = 10;
ages[i] = i;
ages[0] = 100;    ages[99] = 0;

max = 100.0;
scores[0] = 0/max; scores[1] = 1/max;
scores[98] = 98/max; scores[99] = 99/max;

printf("Well, %s, thats all!", name);
}
```

string ends with a noncharacter code (tipically '\0')

begins with index 0

without &, this is the name of array

4.2.1.2. Initialized Static Arrays

File dates.c:

```
int IsLeapYear(year)
    int year;
{
    return year%4 && (year%400 || !year%100);
}

long DayOfEpoch(year, month, day)
    int year, month, day;
{
    static int DayOffsets[] = { 0,  31,  59,  90, 120, 151;
         151, 212, 243, 273, 304, 334 };
    long days;

    days = year*365 + year/4 - year/100 + year/400
              + DayOffsets[month] + day;
    if (month > 1 /* Feb */ && IsLeapYear(year))
        days++;
    return days;
}
```

4.2.2.1. Declaring & Initializing

- Only statically allocated arrays can be initialized with an initialization clause.

  screen is an array of 24 arrays with 80 characters each.

  ```
  char screen[24][80];
  ```

  (Separate brackets) (screen[3][j]; screen[i,j] is legal but it is some other thing)

  I4X4 is an array of 4 subarrays, each containing 4 integers. I4X4 is being initialized to an identity matrix.

  ```
  static int I4X4[][] = { { 1, 0, 0, 0 }, {
  0, 1, 0, 0 }, { 0, 0, 1, 0 }, { 0, 0, 0, 1
  } };
  ```

- Array sizes are optional where initialization clauses make it apparent.
- Good practice suggests giving sizes explicitly if the size matters to the code accessing the data.

4.2.3.1. Using Multidimensional Arrays

The file "demos/multidim.c

```
main()
{
    int src, dst;
    float amount, x[3][3];

    x[0][0] = 1;          x[1][1] = 1;x[2][2] = 1;
    x[1][0] = 12;           x[0][1] = 1 / x[1][0];
    x[2][1] = 3;            x[1][2] = 1 / x[2][1];

    printf("Enter the source & destination   unit");
    printf("0 = inches, 1 = feet, 2 = yards");
    scanf("%d %d", &src, &dst);
    printf("Amount: ");
    scanf("%f", &amount);

    printf("Result is %f.", amount *       x  [src][dst]);
}
```

4.2.3.3. Arrays of Unknown Size

C does not check array index bounds, therefore:

The number of elements of an array merely being pointed to are irrelevant:

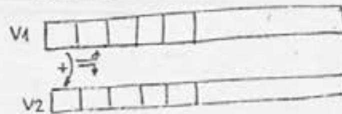
```
VectorAdd(v1, v2, size)
    int v1[], v2[], size;
{
    if ( size )
        {
         v2[0] = v1[0] + v2[0];
    /* modifies original caller's values */
         VectorAdd( &v1[1], &v2[1], size - 1 );
    /* passes array slice */
        }
}
```

Stylistically, the number of elements of an array parameter should appear if and only if the code is size dependent.

Although the number of elements is irrelevant, the element size is needed to perform indexing operations, therefore:

Dimensions of multidimensional arrays require all but the last array size.

recursion!!



4.2.4.1. The Increment & Decrement Operators

- The increment operator ++ increments an integer variable by 1:

  ++i is the same as i = i + 1

- The decrement operator -- decrements integers similarly.

- Pre-increments and pre-decrements (applied in front) happen before expression evaluation:

```
i = 0;
j = ++i + ++i;
/* i == 2, j == 4 */
```

wrong from the viewpoint of the compiler (must be in the parentheses) j= (++i + ++i)

- Post-increments and post-decrements (applied behind) happen after expression evaluation:

```
i = 0;
j = i++ + i++;
/* i == 2, j == 0 */
```

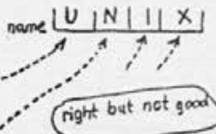
4.2.4.3. Arrays Are Pointer Constants

- Array identifiers refer to:
  - the address of the array, i.e., the address of the first element of the array, therefore they are simply pointer constants
- Pointer variables pointing into arrays can be stepped along sequential elements with increments & decrements.

```
char name[20], *pc;
pc = name;
  :
*pc = 5; /* has the same effect as name[0] = 5 */
pc++ ; /* now points to name[1] */
        /* name is a constant:
           so... name = ... is wrong  */
```

4.2.4.3. Arrays Are Pointer Constants

- The file unix.c:




```
main()
{
char *c, name[4];

name[0] = 'U';    name[1] = 'N';
name[2] = 'I';    name[3] = 'X';

putchar( *name );putchar( *(name + 1) );
putchar( *(name + 2) );putchar( *(name + 3)
);

c = name;
putchar( *c++ );   putchar( *c++ );
putchar( *c++ );   putchar( *c-);
}
```

(right but not good)

(better, using pointers)

(notation similar to assembly language)

4.2.4.6. Passing Array Slices

- The file slice.c:

```
main() {
    char DoubleBuf[2][80];

GetLine( DoubleBuf[0] );
GetLine( DoubleBuf[1] ); }

int GetLine( line )
    char line[]; {
    *line = getchar();
    if ( *line == '\n' )
        {
         *line = '\0';
         return 0;
        }

return 1 + GetLine( line + 1 ); }

/* Recursion!!
   at the end GetLine returns the number of accepted characters */
```

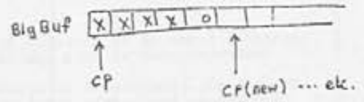
(0 ... 79 0 ... 79)

(the same as: char *line;)

(string terminator)

4.2.4.7. Sharing Array Storage

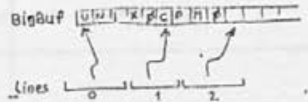
```
main()
{
    char BigBuf[160];
    char *cp;

    cp = BigBuf;
    cp = cp + GetLine( cp );
    cp = cp + GetLine( cp );
}
```



/* Better : less storage needed */

4.2.4.8. Arrays of Pointers

```
main()
{
    char BigBuf[160];
    char *lines[3];      <---- array of pointers !!

lines[0] = BigBuf;
lines[1] = lines[0] + GetLine( BigBuf );
lines[2] = lines[1] + GetLine( lines[1] );
GetLine( lines[2] );

    puts( lines[2] );    <---- put String
    puts( lines[1] );
    puts( lines[0] );
}
```



Note that lines[2][5] is the 6th element of the 3rd line, just as if lines were a two dimensional array.

lines[2][5] ≡ *(lines[2]+5)

5.1.1.1. The Form of the switch Statement

File switch.c:

```
main()
{
    char c;

    printf("Please type a character: ");
    c = getchar();

    switch( c )
        {
        case '0': case '1': case '2': case '3':
        case '4': case '5': case '6': case '7':
        case '8': case '9':
        printf("You typed a digit0);
        break;
        case 'a': case 'e': case 'i': case 'O':
        case 'u': case 'A': case 'E': case 'I':
        case 'O': case  'U':
        printf("You typed a vowel0);
        printf("In upper or lower case0);
        break;
        case ' ':
        printf("You typed a space0);
        break;
        case '':
        printf("You typed a tab0);
        break;
        default:
        printf("You typed a consonant");
        printf("or special character.");
        break;
        }
}
```

5.1.1.2. The Semantics of the switch Statement

- The switch statement can take any integer or character; that is, any discrete value, and use it to select statements to execute.
- Each case must be labelled with a constant expression, and must be unique.
- If the value being switched on matches any of the cases, execution will immediately transfer to that case.
- If none of the cases matches the switch value, control will transfer to the default case.
- If none of the cases match and there is no default case, no action will be performed.

5.1.1.3. Using break Inside switch Cases

- Normally, each alternative sequence of statements in the switch statement will end with a break statement.
- The break statement terminates the switch statement immediately.
- Without the break statement, control would pass into the next group of statements; this is:
  - usually confusing; hence, rarely desirable
  - but is sometimes done to optimize a highly time-critical section of code.

5.2.1.1. The Usage of the while Statement

Part of the file GetPutStr.c:

```
main()
{
    char s[80];

    GetStr(s, 80, '\0');
    PutStr(s);
}
int PutStr( s )
    char *s;
{
    while ( *s != '\0' )
        {
            putchar( *s );
            s++;
        }
}
```

5.2.1.3. The do Variant

Part of the file GetPutStr.c:

```
int GetStr( s, max, ender )
    char *s, ender;
    int max;
{
    int c, len = 0;

    do
        {
            c = getchar();
            len++;
            if (len < max)
              *s++ = c;
        } while ( c != ender && c != EOF );
    *s = '\0';
    return len;
}
```

defined in STDIO.H

5.2.2.1. The Usage of the for Statement

The file for.c:

```
int GetStr( s, max, ender )
    char *s, ender;
    int max;
{
    int c, len;

    max--; /* for '\0' */
    for (len = 0; (c = getchar()) != ender && c
    != EOF ; len++)
        if (len < max)
            *s++ = c;
    *s = '\0';
    return len;
}
```

this is possible because C-language guarantees testing from left to right. so c is defined before "c!=EOF" test.

C-5-14

5.2.3.1. Exiting Loops Prematurely

The break statement will prematurely break the innermost loop in which it is enclosed. File UnCtrl.c:

```
/* Untranslate Control Characters */
main()
{
    int c;
    while ( (c = getchar()) != EOF )
        {
                if ( c == '^' )
                  {
                  c = getchar();
                  if ( c >= '@' && c <= '_' )
                    c = c - '@';
                  else if ( c == '?' )
                    c = '\177'; /* ASCII DEL */
                  else if ( c == ' ' /* SP */ )
                    c = '^',
                  else
                    putchar('^');
                  if ( c == EOF )
/* ---> */        break;
                  }
                putchar( c );
        }
}
```

C-5-17

5.2.3.2. Skipping to the Next Iteration

The file NonBlank.c:

```
/* Copy nonblank characters */
main()
{
  int c;

  while ( (c = getchar()) != EOF )
    {
      if ( c == ' ' || c == '\t' )
/* ---> */      continue;
      putchar( c );
    }
}
```

(handwritten annotations: "TAB"; "Text will be compressed, without blanks and tabs"; "This is just an example. Could be better programmed")

5.2.3.3. Cautions with break and continue

- Ideally, the loop control structure should express the entire context of the evaluation of the statements in between.
- break destroys the "one entrance and one exit" principle of control constructs.
- Both break and continue have a tendency to be overlooked by readers.
- Good practice is to use break and continue sparingly, and always with a comment calling attention to their use.

5.3.2.1. Spaghetti Code

```
/* Untranslate Control Characters */
main()
{
   int c;

   start:
   if ( (c = getchar()) == EOF ) goto end;
   if ( c != '^' ) goto doit;
   c = getchar();
   if ( c >= '@' && c <= '_' )
        c = c - '@';
   else if ( c == '?' )
        c = '177'; /* ASCII DEL */
   else if ( c == ' ' /* SP */ )
        c = '^'
   else
        putchar('^');
   if ( c == EOF ) goto end;
   doit: putchar( c );
   goto start;
   end:
}
```

6.1.1.1. Declaration & Use of Structures

The file: you.c:

```
main()
{
  struct
    {
      char name[20];
      long SSN;
      float weight;
    } you;

    printf("Please type your first name, SSN & weight:0);

    scanf("%s %ld %f", you.name, &you.SSN, &you.weight);

    printf("Name = %s, SSN = %ld, weight = %f0,

    you.name, you.SSN, you.weight); }
```

Structure name

address, because these are variables

this is already an address

SSN ... Social Security Number

6.1.1.2. Use of the Structure Tag

From the file SayOlder.c:

```
struct person
   {
     char name[20];
     int age;
     float height, weight;
   };

main()
{
  struct person p1, p2;
  float ratio;

  printf("Enter first name, age, height & weight of 2 persons:0);
  GetPerson(&p1);
  GetPerson(&p2);

  if (p1.age > p2.age)
    SayOlder(p1, p2);
  else if (p2.age > p1.age)
    SayOlder(p2, p1);
  else
    printf("%s is the same age as %s ",p1.name, p2.name);
}
```

p1 and p2 are 'variables' of type 'structure person.'

storage reservation for p1 and p2 of this type

An example run:

```
Enter first name, age, height & weight of 2 persons:
John 32 6.2 182.5
Mary 33 5.8 142.0

Mary is older than John
```

6.1.1.3. Passing Structures to Functions

From the file SayOlder.c:

```
SayOlder(older, younger)
    struct person older, younger;
{
    printf("%s is older than %s", older.name,
    younger.name);
}

GetPerson(p)
  struct person *p;
{
  scanf("%s %d %f %f", (*p).name, &(*p).age,
  &(*p).height, &(*p).weight);
}
```

Bad form:
by call the whole structure
is copied

attention to the brackets

5.1.1.4. Efficiency & Abbreviations

The file SayOlder2.c:

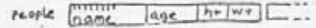
```
main()
{
    struct person p1, p2;
    float ratio;

    printf("Enter first name, age, height % weight of 2
    persons:0);
    GetPerson(&p1); GetPerson(&p2);

    if (p1.age > p2.age)
        SayOlder(&p1, &p2);
    else
        if (p2.age > p1.age) SayOlder(&p2, &p1);
    else
        printf("%s is the same age as %s0, p1.name, p2.name);
}

SayOlder(older, younger)
    struct person *older, *younger;
{   printf("%s is older than %s", older-> name, younger-> name);
}

GetPerson(p)
    struct person *p;
{
    scanf("%s %d %f %f", p->name, &p->age, &p->height, &p-
    >weight);
}
```

Better:
only pointers to the structures
are transmited

younger -> name is the same as (younger).name , but it is more clear.

C-6-7

6.1.1.5. Combining Structures with Arrays



The file: eldest.c:

```
main()
{
    struct person people[10], *p;
    int i;

    printf("Please input 10 person records,
    each with:0);
    printf("first name, age, height & weight :" );
    for (i = 0; i < 10; i++)
        GetPerson(&people[i]);

    p = &people[0];
    for (i = 1; i < 10; i++)
        if ( people[i].age > p->age )
            p = &people[i];

    printf("%s is the eldest0, p->name); }
```

Pointer → p

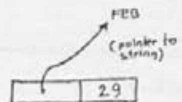
6.1.1.5.a Combining Structures with Arrays

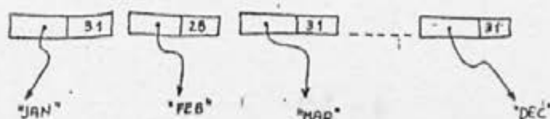
```
/* Better solution */
main( )
{
struct person *q, *p, people[10];
 for (q = people ; q < people + 10 ; q++)
    GetPerson (q);

 p = people
 for (q = people + 1 ; q < people + 10 ; q++)
     if (q -> age > p -> age
          p = q ;
```

6.1.1.6. Initializing Structures

From the file: months.c:

```
struct month
    {
        char *name;
        int numDays;
    };

static struct month LeapFeb = {"Feb", 29};

static struct month Months[12] =
        { {"Jan", 31}, {"Feb", 28}, {"Mar", 31},
        {"Apr", 30},{"May", 31}, {"Jun", 30},
        {"Jul", 31}, {"Aug", 31}, {"Sep", 30},
        {"Oct", 31}, {"Nov", 30}, {"Dec", 31} };
```





6.1.1.7. Limitations on Whole Structure Operations

- There are only three operations permissible on structures:

  (1) Taking the address of a structure,

  (2) Referencing one of the fields of a structure, and

  (3) Assigning (copying) a structure.

  In particular, this means that it is impossible to compare structures for equality or order with the relational operators of C.

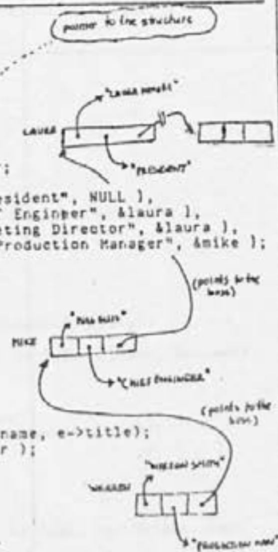
- In older versions of C it is impossible to copy structures, or return them as the values of functions.

  In particular, this means the inability to assign structure values or pass a structure as the value of a function parameter.

6.2.2.1. Employees and their managers.

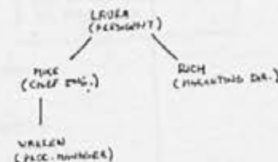
The file managers.c:

```
#include <stdio.h>

struct employee
    {
        char *name, *title;
        struct employee *manager;
    }
    laura = { "Laura Henry", "President", NULL },
    mike = { "Mike Ellis", "Chief Engineer", &laura },
    rich = { "Rich Rocco", "Marketing Director", &laura },
    warren = { "Warren Smith", "Production Manager", &mike };

main()
{
    ShowChain( &warren );
}

ShowChain( e )
    struct employee *e;
{
    if ( e != NULL )
        {
            printf("%s, %s0, e->name, e->title);
            ShowChain( e->manager );
        }
}
```



The output:

```
Warren Smith, Production Manager
Mike Ellis, Chief Engineer
Laura Henry, President
```

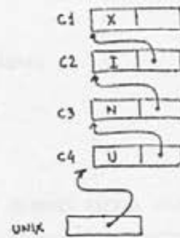


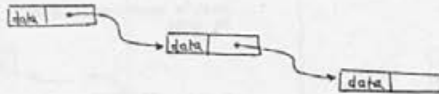
C-6-20

6.2.2.2. Declaring & Initializing Linked Lists

From the file Unix.c:

```
struct charList
    {
        char data;
        struct charList *next;
    }
        C1 = { 'X', NULL },
        C2 = { 'I', &C1 },
        C3 = { 'N', &C2 },
        C4 = { 'U', &C3 },
        *Unix = &C4;
```





6.2.2.3. Traversing Linked Lists

From the file Unix.c:

```
main()
{
  printf("%c at %x0", Unix->data, Unix);
  printf("%c at %x0", Unix->next->data, Unix->next);
  printf("%c at %x0", Unix->next->next->data, Unix->next-
>next);

    CLput(Unix); }

CLput( cl )
    struct charList *cl;
{
    while (cl != NULL)
        {
         putchar( cl->data );
         cl = cl->next;
        }
}
```

An example run:

```
U at e20
N at e1c
I at e18
UNIX
```

6.2.2.4. Creating List Structures

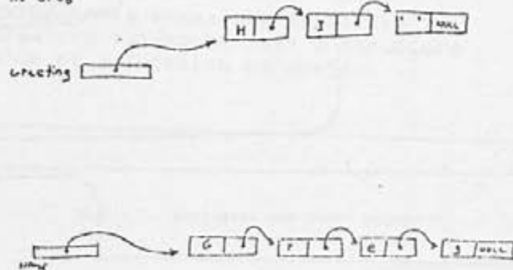
From the file greeting.c:

```
main()
{
    struct charList *greeting, *name;

    greeting = CLmake('H', CLmake('i', CLmake(' ', NULL)));
    printf("What's your name? ");
    name = CLget();
    CLput( greeting );
    CLput( name );
}
struct charList *CLget()
{
    int c;

    if ((c = getchar()) == '\n' || c == EOF)
        return NULL;
    return CLmake( c, CLget() );
}
```

RECURSION!!
REVERSE ORDER!!

An example run:

```
What's your name? Greg
Hi Greg
```



6.2.2.5. Creating New Storage

```
struct charList *CLmake( d, n )
    char d;
    struct charList *n;
{
    struct charList *temp;

temp = (struct charList *) malloc( sizeof(
struct charList ) );

    temp->data = d;
    temp->next = n;
    return temp;
}
```

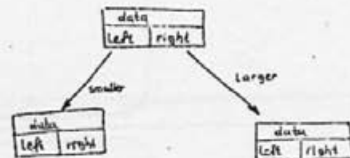
This is specific for UNIX (O.System call)

6.2.2.6. Declaring & Creating Binary Trees

From the file binTree.c:

```
struct binTree
    {
        int data;
        struct binTree *left, *right;
    };

struct binTree *BTmake( d, l, r )
    int d;
    struct binTree *l, *r;
{
    struct binTree *temp;

temp = (struct binTree *) malloc( sizeof( struct binTree ) );

    temp->data = d;
    temp->left = l;
    temp->right = r;
    return temp;
}
```

reservation of storage



6.2.2.7. Sorting With Binary Trees

From the file binTree.c:

```
main()
{
    struct binTree *t;
    int i;

    printf("Enter numbers to be sorted, followed by ^D");
    t = NULL;
    while ( scanf("%d", &i) != EOF )
        {
            BTinsert( &t, i );
            BTinorder( t );
            putchar('\n');
        }
}

BTinorder( root )
    struct binTree *root;
{
    if (root != NULL)
        {
            BTinorder( root->left );
            printf("%d ", root->data);
            BTinorder( root->right );
        }
}
```

6.2.2.9. Inserting Into Binary Trees

From the file binTree.c:

```
BTinsert( pp, d )
    struct binTree **pp;
    int d;
{
    if (*pp == NULL)
        *pp = BTmake( d, NULL, NULL );
    else
        if ( d < (*pp)->data )
            BTinsert( &(*pp)->left, d );
        else
            BTinsert( &(*pp)->right, d );
}
```

```
struct complex {float real, imag;}

struct complex add (c1, c2)
  struct complex c1, c2;
  {
    c1.real = c1.real + c2.real;
    c1.imag = c1.imag + c2.imag;
    return c1;
  }
```

7.1.1.1.1. The Function fopen

```
FILE *fopen(filename, type)
    char *filename, *type;
```

fopen takes:

filename an operating system specific filename
type a string determining access permissions

and returns:

a stream descriptor (of type FILE *) for doing I/O.

Access Permissions

The string type can contain various character codes, including:

r for read access
w for write access
a for append access

example:

```
File *fp
fp = fopen("fred", "r");
```

7.1.1.2. Writing to an External File

The file out.c:

```
#include <stdio.h>

main()
{
    FILE *fp;

fp = fopen("foo", "w");

fprintf(fp, "Hello world!");

    fclose(fp); }
```

7.1.1.3. Copying a File of Characters

The file copy.c:

```
#include <stdio.h>

main()
{
    FILE *in, *out;
    int c;

 in = fopen("foo", "r");
 out = fopen("bar", "w");

 while ((c = getc(in)) != EOF)
        putc(c, out);

        fclose(in);
        fclose(out);
}
```

7.1.2.1. Single Character I/O

int getc(stream) Returns a character from the FILE *stream; given stream, EOF on End Of File.

int getchar()	An abbreviation for getc(stdin).
int putc(c, stream)	Puts c on the given stream.
char c;	putc returns the character c
FILE *stream;	for convenience in expressions.
putchar(c)	An abbreviation for putc(c, stdout).

int, not char!!! because of this the function can return the value EOF

stdin, stdout ... standard input (output) stream !

7.1.2.2. Single String I/O

```
char *gets(s)
    char *s;
```

gets reads a string from stdin into s.

The string from stdin is terminated with a newline ('\n').

The string in s is terminated with an ASCII NUL ('\0').

```
char *fgets(s, n, stream)
    char *s;
    int n;
    FILE *stream;
```

fgets reads a string from stream into s until either n-1 characters are read, or a newline is read.

The string will be NUL terminated.

For convenience in forming expressions, fgets returns s.

7.1.3.1. Formatted File Input

```
int scanf(format, p1, ..., pn)
    char *format;
```

Scans stdin, picking up values and putting them into the pointers p1 through pn according to the format codes embedded in the string format.

The number of items successfully read is returned, unless End Of File occurs, in which case EOF is returned.

```
int fscanf(stream, format, p1, ..., pn)
    FILE *stream;
    char *format;
```

fscanf is the same as scanf, except that the named stream is scanned, instead of stdin.

7.1.3.2. Formatted File Output

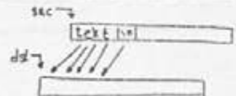
```
printf(format, exp1, ..., expn)
    char *format;

fprintf(stream, format, exp1, ..., expn)
    FILE *stream;
    char *format;
```

In the case of printf, characters are written to the stream stdout. In the case of fprintf, they are written to the named stream.

In either case, the values of the given expressions are written according to the corresponding format codes in format, and surrounded by any literal characters in format.

The format codes are multitudinous, and compatible with those used by scanf and fscanf.

7.2.2.1. Copying Strings



```
char *strcpy(dst, src)
char *dst, *src;
```

strcpy copies the characters of src to dst, stopping after copying the terminating NUL byte.

For convenience, strcpy returns dst.

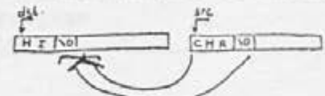
```
char *strncpy(dst, src, n)
   char *dst, *src;
   int n;
```

strncpy copies exactly n characters from src to dst.

If src has fewer than n characters, it will be NUL padded.

If src has n or more characters (not counting its terminating NUL byte), then it will be truncated, AND dst WILL NOT BE NUL TERMINATED!

7.2.2.2. Concatenating Strings



```
char *strcat(dst, src)
   char *dst, *src;
```

Copies src onto the end of dst. dst needs to have room to store the extra characters. dst is returned.

```
char *strncat(dst, src, n)
   char *dst, *src;
   int n;
```

Like strcat, strncat copies src onto dst, but will copy at most n characters, preventing overflowing dst.

Even if not all of dst is copied, the resulting string will be NUL terminated.

7.2.2.3. Inspecting Strings

int strlen(s)	Returns the length of the char *s; NUL terminated string s.
char *index(s, c)	Returns a pointer to the first char *s, c; occurrence of c in s, or 0.
char *index(s, c)	Returns a pointer to the last char *s, c; occurrence of c in s, or 0.

7.2.2.4. String Comparison

```
int strcmp(s1, s2)
    char *s1, s2;
```

Compares s1 and s2 lexicographically, according to the collating sequence defined by the character codes.

strcmp will return a positive number if s1 > s2, zero if s1 = s2 and a negative number if s1 < s2.

```
int strncmp(s1, s2, n)
    char *s1, *s2;
    int n;
```

strncmp compares s1 and s2 as if they had been truncated to at most n characters.

Neither strcmp nor strncmp will compare strings with embedded

digit strings in the human fashion, for example, "12" is considered to be less than "3".

9.1.3.1. Controlling Compilation with Macros

- The C Preprocessor has the facility to select sections of code based upon the value of macros.

```
#ifdef Berk
        vfork();
#else
        fork();
#endif
```

  - The code is selected by either #ifdef or #ifndef (if undefined), and the #else clause is optional.
  - If necessary, macros can be selectively undefined with #undef.

10.1.1.1. The Concept of a Process `fork'

- The UNIX operating system has only one way to generate multiple parallel processes.
- When a UNIX process issues the fork system call, it is split into two identical processes, the original and a copy.
- All UNIX processes have unique process ID numbers. The original process retains it's process ID, and the copy receives a new one.
- When first split, both processes are executing the same code in the same place, and have identical copies of all variables.
- Only their different process id numbers and the values returned by the fork systems call differentiate parent from child.
- The parent receives the child's process id as the value of the fork systems call. The child receives the value zero.

10.1.1.3. Synchronization with Child Processes

The file ~demos/chap10/wait.c:

```
main()
{
   int pid1, pid2, status;
   if ( (pid1 = fork()) == 0 )
      {
       printf("I am the child process." );
       exit(5);
      }
   pid2 = wait(&status);
   printf("I am the parent." );
   printf("My child with pid %d just died
with status %d.", status%256);
   printf("It returned exit code %d.",
status/256);
   }
```

child — parent — waits for any child to die

10.1.2.2. An Example Program Using `fork' & `execl'

The file ~demos/chap10/execl.c:

```
main()
{
   int status;

   puts("The people currently using the sys-
   tem are:");
   if ( fork() == 0 )
      execl("/bin/who", "who", 0);
   wait(&status);
   puts("Courtesy of the program ``who''.");
}
```

10.2.1.1. Creating Binary Files

```
struct person
{
    char name[10];
    int age;
    float height, weight;
};

main()
{
    int fd;          <-- file descriptor
    static struct person
        pl = {
{'O','e','o','r','g','e','\0','\0','\0','\0'},
           21, 6.2, 185.3 },
        pl = {
{'M','a','r','s','h','a','\0','\0','\0','\0'},
           27, 5.2, 105.3 };

    fd = open("somedata", 1);      /* write access */

    write(fd, &p1, sizeof( struct person ));
    write(fd, &p2, sizeof( struct person ));
    close(fd);
}
```

```
fd = open("fred", 0/1/2)      0 ... reading
                              1 ... writing
                              2 ... both

read(fd, buffer, bytecount)
write (fd, buffer, bytecount)
close (fd)
```

C-10-11A

10.2.1.2. Random Access Files

```
struct person
{
    char name[10];
    int age;
    float height, weight;
};

main()
{
    int fd;
    struct person p1, p2;

fd = open("somedata", 2);
    /* read/write access */
read(fd, &p1, sizeof( struct person ));
read(fd, &p2, sizeof( struct person ));
lseek(fd, 0L, 0);
    /* relative to beginning of file */
write(fd, &p2, sizeof( struct person ));

write(fd, &p1, sizeof( struct person )); }
```

```
lseek(fd, offset, wherefrom)
          |        0 .. startOfFile
          |        1 .. currentPosition
          |        2 .. endOfFile
          how many byts of displacement (Longinteger)
```

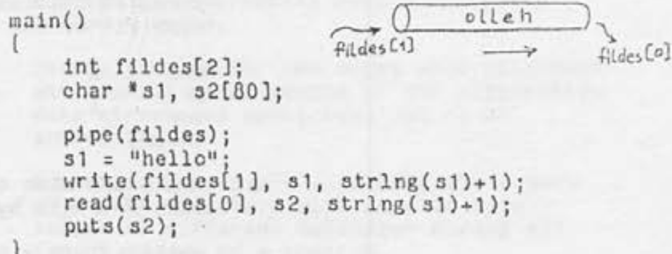
C-10-12

10.2.2.1. The Concept of a Pipe

- A UNIX pipe is a connection between an output stream and an input stream, such that data written to the output stream may be read from the associated input stream.
- Pipes may connect separate processes in UNIX. In fact, programs have no way of knowing in UNIX whether a file descriptor is associated with a pipe, a file, or a device (such as a tty).
- Like buffered files, pipes have a fixed sized buffer. When the process writing to the pipe files the buffer, it is suspended.
- When a process reads from a pipe, it is suspended until there is data to read.

10.2.2.2. Creating Pipes

The file ~demos/chap10/pipe.c:




```
main()
{
	int fildes[2];
	char *s1, s2[80];

	pipe(fildes);
	s1 = "hello";
	write(fildes[1], s1, strlng(s1)+1);
	read(fildes[0], s2, strlng(s1)+1);
	puts(s2);
}
```

10.2.2.3. Connecting Process Families with Pipes

The file ~demos/chap10/whonum.c:

0... standard input
1... standard output

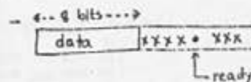
```
main()
{
	int fildes[2];

pipe(fildes);

if ( fork() == 0 )
	{
		dup2(fildes[0], 0);
	/* Install fildes[0] as stdin */
		execl("/usr/bin/num", "num", 0);
	}

	dup2(fildes[1], 1);
	/* Install fildes[1] as stdout */
	execl("/usr/bin/who", "who", 0);
}
```

1-41

10.3.1.1. Bit Structures

- Structure declarations can specify the exact number of bits to be used by each field, as well as where extra padding bits should be inserted:

```
struct port
  {
    :3;
    unsigned ready:1;
    :4;
    unsigned data:8;
  };
```



10.3.1.2. Bit Operators

- In addition to the logical operators, C provides bitwise operators which operate on each bit of the words they are applied to:

```
and     &
or      |
xor     ^       (exclusive or)
invert  ~       (one's complement)
```

- There are also operators for shifting words left or right by a given number of bits:

```
shift x left y bits:        x << y
shift x right y bits:       x >> y
```

```
int a,b,c;
c = a&b;
c = a|b;
c = ~b;
```

10.3.2.2. Peeking & Poking with Casts

- Normally, integers and pointers are incompatible. But sometimes its necessary to specify pointer values as integer addresses:

```
int *p;

p = (int *) 0xff56;

printf("Memory location %x contains %d ",
0xff56, *p);

*p = 0;

printf("Now it contains zero." ;
```

(operator - used by the compiler - tells to the compiler the type of pointer p)

(This corresponds to POKE(FF56,0))

(This corresponds to PEEK(FF56))

1-42

10.3.2.3. Hardware Data Structure Manipulation

- Hardware data structures, especially device registers in memory mapped computer address spaces, can be described as bit structures and then manipulated quite flexibly:

```
struct port *p;

p = (struct port *) 0177644;
    /* address of serial port */

    while (! p->ready)
    /* device not ready */ ;
    /* busy wait */

p->data = 'A';
    /* send an 'A' out the port */
```

10.3.2.4. Free Unions

- C provides an explicit way to overlap storage of alternative data structures, called a union.
- Unions provide for the worst case alignment and storage requirements of the alternative data structures specified, but do no conversion.
- Normally, unions are used only where part of a data structure will need to have values of different datatypes during different phases of a program.
- However, sometimes it is necessary to access a data structure with a datatype description different from that used to create it.
- This usage should be used quite sparingly, as it is highly dependent on non-portable assumptions about low level data representations.

10.3.2.4.a Equivalents

x = x + 6	is the same as	x += 6
a = a * y	is the same as	a *= y
x = x & 3	is the same as	x &= 3

NAR. IN UNIV. KNJIŽNICA
Ljubljana

362003