An example

```
/* hello.c     Jeff Shallit 3/29/89 */

/* This program just print some things and quits.
 * It illustrates the structure of C programs, and
 * how to run the compiler
 */

#include <stdio.h>

main()
{
int n;                  /* an integer variable */

    printf("Hello, world\n");
    n = 10;
    printf("The value of n is %d\n", n);
}
```

Comments delimited by `/*` and `*/`

`#include <stdio.h>` reads the header (interface) file for the standard I/O library.

Parameters to procedures enclosed in parens, `main()` expects no arguments.

Every program must have a procedure called `main`.

Procedure body delimited by `{` and `}`.

Some special characters: `\n` – newline, `\t` – tab, `\0` – end of string, `\f` – form feed.

Format specifications introduced by `%`, one for each additional argument to *printf*. Some useful formats: `%d` – decimal integer, `%s` – string, `%c` – character, `%f` – floating-point, `%%` – the character `%` itself.

Types, declarations, variables, literals

Basic types

```
int n;
char c;
float x;

int a,b,c,d;

int year = 1988;
char field_separator = '\t';
float pi = 3.14159;
```

Variables can be initialized in a declaration, any constant expression may be used.

Character constants delimited by single quotes.

Arrays

```
int dept[20];

char buffer[100];

int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19 };

float BigMatrix[10][10];
```

Elements of `dept[20]` are `dept[0]` . . . `dept[19]`. Indices always start at 0. `dept[20]` doesn't exist, but compiler doesn't catch range errors.

Initial array values delimited by `{` and `}`. Compiler will figure out size for you.

Multidimensional arrays are simply arrays of arrays of arrays.

Simple statements

Assignment statements

```
n = 3;

A[i][j] = A[i-1][j] + A[i][j-1];
```

Procedure/Function calls

```
invert_matrix(A, n);

initialize();
```

(Compound) Statement block

```
{
    first_thing();
    second_thing();
    third_thing();
}
```

Statements end with a semicolon. (stmt terminator, not separator!)

Blocks delimited by `{` and `}`, like Pascal's **begin** and **end**. No semicolon after `}`.

Control statements

If–then–else

```
if (temperature>212) {
    boil();
    }
else if (temperature>32) {
    pour();
    }
else {
    skate();
    }
```

while loop

```
while (x != A[i]) {
    i = i+1;
    }
```

Loop and conditional guards delimited by parens `()`.

Guards should evaluate to an integer; zero means false, nonzero means true.

do–while loop

```
do {
    c = get_next_input();
    done = process(c);
    } while (!done);
```

"Not" operator is `!`: `!=` – not equal, `!done` – false (zero) if done is true (nonzero), true otherwise.

for loop

```
for (k=0; k<n; k=k+1) {
    sum = sum + B[k];
    }
```

Guard on for loop has three parts separated by semicolons: initialization (done once), continue test (done before each iteration), increment (done after each iteration).

Loop terminates when continue test fails (evaluates to false).

Do–while loop performs at least one iteration. For and while loops may perform zero iterations.

switch (case) statement

```
switch (first_character) {
    case 'i':
        insert_text();
        break;
    case 'a':
        append_text();
        break;
    case 'q':
    case 'Q':
        quit();
        break;
    default:
        error();
        break;
    }
```

Case labels must be constant. Cases may have multiple labels.
End each case with a `break` statement.

iteration interruption

```
while (1) {
    r = get_next_record();
    if (is_fake(r))
        continue;           /* start next iteration */
    process(r);
    done = are_we_done_yet();
    if (done) break;        /* exit from middle of loop */
    do_more(r);
    }
```

`continue` goes immediately to the top of the loop to execute termination test. In for-loops, it executes the increment.

`break` immediately leaves the loop.

`break` applies to innermost loop or switch (if loops or switches are nested).

Pointers

```
int i, j, *ip;           /* ip is a pointer to an integer */
char c, *cp;             /* cp is a pointer to a character */

    j = 5;
    ip = &j;            /* ip is a "pointer to" j */
    i = *ip;            /* dereferencing ip.  Now i is 5 */
    i = 6;
    *ip = i;            /* j is 6 now!    */
    *cp = ’x’;          /* cp points to ’x’ in some unnamed mem loc */
    c = *cp;
    printf("%d, %c", *ip, *cp);  /* prints 6, x */

    cp = &c;            /* cp is a "pointer to" c */


int i, *ip, **ipp, ***ippp;

    ip = &i;
    ipp = &ip;
    ippp = &ipp;
    printf("%d", ***ippp);
```

You can create a pointer to any named variable using `&` (”ampersand”).
`*` dereferences a pointer in expressions (like Pascal’s `^` carat symbol).

Structures (like Pascal's "records")

```
struct person {             /* person is a "structure tag" */
    char name[20];
    int age;
    int height;
    } aPerson;             /* aPerson is a variable of type person */

struct person anotherPerson;
    aPerson.height = 72;
    if (aPerson.age>30) distrust(&aPerson);
```

Select fields with a dot (like Pascal).

Don't pass structures in and out of procedures, pass pointers to structures.
It is more efficient.

Example: Linked list

```
struct listnode {
    int value;
    struct listnode *next;
    };

struct listnode *header, *p;
int key;

/* search a linked list */
    p = header;
    while (p->value != key)  /* b/c p is a pointer; dereferencing */
        p = p->next;
```

Structures may contain pointers to structures.

Dereference and select using `->` operator (like Pascal `^.`).

Example: Doubly-linked list

```
struct dlistnode {
    int value;
    struct dlistnode *prev;
    struct dlistnode *next;
    };

typedef struct dlistnode *DLIST;
/* DLIST is defined as a shorter name for the structure */

/* insert *newnode before *p in a doubly-linked list */
DLIST p, newnode;
/* Note its usage - compare with previous slide */
    newnode->next = p;
    newnode->prev = p->prev;
    p->prev->next = newnode;
    p->prev = newnode;
```

Use `typedef` to shorten and clarify program text, and to hide details of a representation.

By convention, user-defined type names are all caps.

Expressions and operators

Arithmetic	`+` `-` `*` `/` `%` `++` `--`
Logical	`<` `<=` `>` `>=` `==` `!=` `||` `&&` `!`
Bit	`&` `|` `^` `~` `<<` `>>`
Assignment	`=` `+=` `-=` `*=` etc.
Conditional	`(` *expr* `)` `?` *value1* `:` *value2*
Pointer	`*` `&` `->` `.` `sizeof`
Type	`(` *type* `)` *var* or `(` *type* `)(` *expr* `)`

Division `/` produces an `int` if both operands are `int`, otherwise `float`.

`a%b` is *a* mod *b*, the remainder after dividing *a* by *b*.

`++b` means preincrement; `a = ++b;` is equivalent to `b=b+1; a=b;`

`b++` means postincrement; `a = b++;` is equivalent to `a=b; b=b+1;`

Use `==` to test for equality, not `=` (assignment operator).

`&&` is short-circuit and; `a && b` means "evaluate *a*; if it's false, ignore *b* and produce a false value, otherwise evaluate *b* and return its (logical) value".

The first four bit operators are, in order, And, Or, Exclusive Or, Complement, each taken bitwise.

`a<<3` means *a* shifted left 3 bits (multiplied by 8). `>>` is right shift.

`a += 5;` is equivalent to `a = a+5;`.

A conditional evaluates *expr*; if it's true, it evaluates *value1*, otherwise it evaluates *value2*.

`sizeof`(*type name*) computes the number of bytes needed to represent a value of the named type.

A type cast `(int) (x/y)` coerces a value into the named type.

Examples

Arithmetic

```
     a+1
     offset%2
     ++n
     n++
```

Logical

```
     x>0
     (k<n) && (A[k]==key)
     !found
```

Bit

```
     status & mask
     1 << offset
```

Assignment

```
     sum += A[k];
     status |= OVERFLOW;
```

Conditional

```
     max = (a>b)?  a :  b;
     tax = (income>100000)?  0 :  (income>50000)?  10000 :  income;
```

Pointer

```
     p = p->next;
     *cp++ = c;
     next_char = *++cp;
     malloc(sizeof (struct listnode))
```

Type

```
     ratio = (float) m/n;
     p = (struct listnode *) malloc(sizeof (struct listnode));
```

Pointers and Arrays

```
char *cp;
char buffer[100];
char c;

    c = search_character();
    cp = buffer;
    while (*cp != c)  ++cp;
```

```
#define n 20
struct bigstruct A[n];
struct bigstruct *bsp;

    for (bsp=A, k=0;  k<n;  ++bsp, ++k) {
        total += bsp->amount;
        }
```

Arrays are pointers.

`buffer` is really of type `char *`, and it points to the first character of a block of 100 characters.

`A` points to the first element in a block of 20 `bigstructs`.

`++bsp` increments the pointer `bsp` by one unit — here the unit is the number of bytes in a `bigstruct`.

Arithmetic on pointers is done in the relevant units — the size of the object being pointed to.

Strings

```
#include <strings.h>

char message[] = "This is a message";
char buffer[100];
char *cp1, *cp2;

    for (cp1=buffer, cp2=message;  *cp2!='\0';  *cp1++ = *cp2++) ;
    *cp1 = '\0';

    strcpy(buffer, message);
    strcat(buffer, " of hope");
```

A string is an array of characters (bytes), terminated by a byte with value 0 (`'\0'`). Alternatively, a string is a pointer to the first character of the array.

The string `"This is a message"` occupies 18 bytes — 17 visible characters plus one byte that terminates the string.

There is a library of functions that manipulate strings. Be sure to `#include <strings.h>` before using it.

Procedures and functions

```
void                            /* return-type (ANSI C) */
print_maximum(int a,int b)
{
int max;
    max = (a>b)?  a :  b;
    printf("The max is %d\n", max);
}



    print_maximum(this_one, that_one);
```

Declare the types of parameters before beginning the body.

Parameters have local scope, arguments are called by value.

No semicolon after the procedure heading.

```
int                             /* returns an integer */
find_minimum(int A[], int n)
{
int min=INFINITY;
int k;
    for (k=0; k<n; ++k) {
       if (A[k]<min) min = A[k];
       }
    return min;                 /* don't miss it!  */
}



    optimum = find_minimum(myArray, mySize);
```

Don't specify the length of parameter arrays.

Arrays are passed by value, but an array is a pointer so changing `A[3]` within the procedure will change an element of the argument array.

```
void
get_min_and_max(int A[], int n, int *minp, int *maxp)
{
int small, large;
int k;

    *minp=INFINITY;
    *maxp=NEG_INFINITY;
    if (n%2) {
       *minp = *maxp = A[--n];
       }

    for (k=0; k<n; k+=2) {
       if (A[k]<A[k+1]) {
           small = A[k];  large = A[k+1];
           }
       else {
           small = A[k+1];  large = A[k];
           }
       if (small<*minp) *minp = small;
       if (large>*maxp) *maxp = large;
       }
}


    get_min_and_max(ycoords, n, &ymin, &ymax);  /* Call-by-reference */
```

Pass pointers to achieve the effect of Pascal **var** parameters, or Ada **out** parameters.

```
char
first_nonwhite(char *s)
{
char *cp=s;
    while (*cp==’ ’ || *cp==’\t’ || *cp==’\n’) ++cp;
    return *cp;
}
```

```
    c = first_nonwhite("  West Side Story  ");
    c = first_nonwhite(buffer);
```

By default, every procedure (function) returns an `int`. Precede the procedure heading with a type name to indicate return values of a different type.

The `return` statement supplies the returned value.

```
struct person *
elect_president(candidate, n)
struct person candidate[];
int n;
{
int k;
    for (k=0; k<n; ++k) {
       if (candidate[k].votes > MAJORITY)
           return &(candidate[k]);
       }
}
```

```
    the_pres = elect_president(slate, num_parties);
```

Don’t return a structure, return a pointer to a structure.

Input and output

```
#include <stdio.h>

    printf("Decimal %d = Octal 0%o = Hexadecimal 0x%x\n",
       a, a, a);

    fprintf(stderr, "Error %d:  %s\n",
       error_number,  error_message[error_number]);

    scanf("%s %d", last_name, &age);
```

The shell sets up three standard I/O streams when you run a program: `stdin` – the standard input, `stdout` – the standard output, and `stderr` – the standard error output. Normally `stdin` is your keyboard, `stdout` is your screen, and `stderr` is also your screen, but this can be changed using pipes and redirection.

The first argument to `fprintf` is a stream.

The arguments to `scanf` must be pointers. `scanf("%d", age);` will cause a mysterious runtime error.

```
/* copy input to output */

#include <stdio.h>

main()
{
int c;

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

`getchar()` returns an `int`, not a `char`. This allows it to return a flag value `EOF` when you try to read past the end of the file.

`EOF` is defined in `stdio.h`.

To produce `EOF` from your keyboard, type control-D.

Files

```
/* head.c       Sam Bent 1/14/88 */

/* print the first 10 lines of a named file */

#include <stdio.h>

#define MAXLINELENGTH   200
#define NUM_LINES       10

main()
{
char name[50];
FILE *fp;
int k;
char *t;
char buffer[MAXLINELENGTH];

    printf("File name:    ");
    scanf("%s", name);
    fp = fopen(name, "r");   /* "r" means read-only */

    if (fp==NULL) {          /* NULL means error from fopen */
        printf("Can’t open %s\n", name);
        exit(-1);            /* sets the shell variable ’status’ */
        }

    for (k=0; k<NUM_LINES; ++k) {
        t = fgets(buffer, MAXLINELENGTH, fp);
        if (t==NULL) break;
        printf("%5d  %s", k+1, buffer);
        }

    exit(0);
}
```

```
/* hunt.c       Sam Bent        1/14/88 */
/* browse around a file at random */

#include <stdio.h>
#define MAXLINELENGTH   200

main()
{
char name[50], buffer[MAXLINELENGTH];
FILE *fp;
float percent;
long length, address;

    printf("File name:    ");
    scanf("%s", name);
    fp = fopen(name, "r");

    if (fp==NULL) {
        printf("Can’t open %s\n", name);
        exit(-1);
        }
    fseek(fp, 0, 2);    /* 2 means offset from end of file */
    length = ftell(fp);

    for (;;) {
        printf("Percentage:    ");
        scanf("%f", &percent);
        if (percent<0) break;
        address = percent * length / 100;
        fseek(fp, address, 0);      /* offset from beginning */
        while (getc(fp) != ’\n’) ;  /* skip to end of line */
        address = ftell(fp);
        fgets(buffer, MAXLINELENGTH, fp);
        printf("0x%06x  %s", address, buffer);
        }

    exit(0);
}
```

Command line arguments

```
/* echo.c      Sam Bent        1/14/88 */

/* print the command line arguments */

#include <stdio.h>

main(int argc, char **argv)
{
    printf("There are %d arguments:\n", argc);
    while (argc-- > 0)
       printf("’%s’ ", *argv++);
    printf("\n");
}
```

The shell passes two arguments to `main` when you run a program: `argc` – the number of words in the command line (including the name of the program), `argv` – an array of strings containing the words.

The C preprocessor

Compile-time constants

```
#define PI              3.14159
#define MAX_ARRAY_SIZE  50
#define O_READONLY      0x40
```

Inserting files

```
#include <stdio.h>    /* <...> for system include files */
#include "mydefs.h"   /* "..." for local include files */
```

Conditional compilation

```
#define DEBUG_ENABLED

#ifdef DEBUG_ENABLED
    dump_current_contents_of_data_structure();
#endif
```

Inline Macros

```
#define MAX(x,y)        ( ( (x)>(y) )?    (x) :  (y)  )
#define NEW(x)          (x *) malloc(sizeof x)

    best = MAX( one, the_other);
    very_best = MAX( MAX(a,b), c);
    p = NEW(struct listnode);
```

Separate compilation

The interface file `hash.h`

```
typedef  .  .  .  HASHTABLE;

HASHTABLE create();
HASHSLOT  enter();
HASHSLOT  lookup();

#define HT_MAXTABLESIZE 50

external int global_hash_data;
```

The supplier file `hash.c`

```
#include "hash.h"

int global_hash_data;

HASHSLOT
lookup(ht, name)
HASHTABLE ht;
char *name;
{
    .  .  .
}
```

```
% cc -c hash.c
```

This produces an object file `hash.o` whose symbol table defines the external symbols `global_hash_data` and `lookup`.

The client file hashtest.c

```
#include "hash.h"

HASHTABLE my_table;

main()
{
HASHSLOT aSlot;
    my_table = create();
    enter(my_table, "Look Homeward, Angel");
    aSlot = lookup(my_table, "Eugene Gant");
    printf("%d", global_hash_data);
}
```

```
% cc -c hashtest.c
```

This produces an object file hashtest.o containing unresolved references to external symbols lookup and global_hash_data.

```
% cc -o hashtest   hash.o  hashtest.o
```

This resolves the dangling references in hashtest.o using the definitions in hash.o, and produces an executable file hashtest. No compilation is done, only linking and loading.

Common errors

- Dereferencing a null pointer. Segmentation fault (core dumped)
- Array index error: `A[10]`. An unrelated variable changes value, with unpredictable consequences.
- `if (x=0)`. Assigns 0 to x, and takes false branch.
- Missing arguments to procedures: `strcmp(buffer)`.
- Missing declarations: `sin(x)` without `#include <math.h>`.
- Semicolon at end of procedure or function header:

  ```
  myproc(int a, int b, int c);
  {
  }
  ```

  Causes a million syntax errors.
- Referring to math library routines without loading the math library. Compile or load command must end with `-lm`.
- Forgetting to declare types of arguments
- No “then” keyword with “if”.
- If-else statements must have BOTH parts end with semicolons.
- `if (x==0) statement`, NOT `if x==0 statement` .
- C source filename must have “.c” suffix
- Use double quotes (`"`), not single quotes to denote char strings

System calls

Sections 2 and 3 of the manual contain dozens of useful procedures available in standard libraries, including procedures for

Mathematical functions (sin, cos)
Random numbers (rand, random)
Sorting (qsort)
String functions (strcpy, strcat)
Input and output (gets, fseek, fopen)
OS information (getpid, getrusage)
Interprocess communication (socket, send, recv)
File system information (stat, link, unlink)
Multitasking (fork, wait)
Directories (opendir, readdir)
Parsing command lines (getopt)

The online manual also explains these.