The C programming Language, Kernighan & Ritchie (1978)

- Basics:
  - variables and constants
  - arithmetic
  - control flow
  - functions
  - rudiments of input and output.
- Writing bigger programs:
  - pointers
  - structures
  - rich set of operators
  - several control-flow statements
  - and the standard library.

Introduction

> "A sequence of characters in double quotes, like `hello, world\n`, is called a character string or string constant"
>
> "An escape sequence like `\n` provides a general and extensible mechanism for representing hard-to-type or invisible characters."
>
> "If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the interger will be converted to floating point before the operation is done."
>
> "In any context where it is permissible to use the value of some type, you can use a more complicated expression of that type."
>
> "A character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set. This is called a character constant, although it is just another way to write a small integer"

Functions

> "With properly designed functions, it is possible to ignore how a job is done; knowing what is done is sufficient"

Arguments - Call by Value

> "Call by value is an asset... It usually leads to more compact programs with fewer extraneous variables, because parameters can be treated as conveniently initialized local variables in the called routine."

External Variables and Scope

> "[. . .] discusses the static storage class, in which local variables do retain their values between calls."
>
> "As an alternative to automatic variables, it is possible to define variables that are external to all functions, that is, variables that can be accessed by name by any function."
>
> "Before a function can use an external variables, the name of the variable must be made known to the function; the declaration is the same as before except for the added keyword extern."
>
> "In certain circumstances, the extern declaration can be omitted. If the defintion of the external variable occurs in the source file before its use in a particular function, then there is no need for an extern declaration in the function. The extern declarations in main, getline, and copy are thus redundant. In fact, commom practice is to place definitions of all external variables at the beginning of the source file, and then omit all extern declarations."
>
> "You should note that we are using the words definition and declaration carefully when we refer to external variables in this section. `Definition` refers to the place where the variables is created or assigned storage; `declaration` refers to the place where the nature of the variable is stated but not storage is allocated"

Types, Operators and Expressions

> "The type of an object determines the set of values it can have and what operations can be performed on it."
>
> "Declarations list the variables to be used, and state what type they haveand perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values."
>
> "We tend to use short names for local variables, especially loop indices, and longer names for external variables."
>
> "Be careful to distinguish between a character constant and a string that contains a single character: 'x' is not the same as"x". The former is an integer, used to produce the numeric value of the letter x in the machine's character set. The latter is an array of characters that contains one character (the letter x) and a '\0'."
>
> "There is one other kind of constant, the enumeration constant. An enumeration is a list of constant values, as in:

```
enum boolean { NO, YES };
```

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified."

If not all values are specified, unspecified values continue the progression form the last specified value:

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
```

> "The quatifier const can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered."
>
> "The const declaration can also be used with array arguments, to indicate that the function does not change that array: `int strlen(const char[]);`"
>
> "The expression `x % y` produces the remainder when x is devided by y, and thus is zero when y divides x exactly. For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 are leap years."

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d is a leap year\n", year);
else
    printf("%d is not a leap year\n", year);
```

> "The precedence of && is higher than that of ||, and both are lower than relational and equality operators"
>
> "By definition, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false."
>
> The unary negation operator ! converts a non-zero operand into 0, and a zero operand in 1. A common use of ! is in constructions like:

```
if (!valid)
/* rather than */
if (valid == 0)
```

Type conversion

> In general, if an operator like + or * that takes two operands has operands of different types, the "lower" type is promoted to the "highter" type before the operation proceeds.
>
> Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.
>
> Since an argument of a function call is an expression, type conversion also takes place when arguments are passed to functions. In the

> absence of a function prototype, char and short become int and float becomes double.
>
> Finally, explicit type conversions can be forced ('coerced') in any expression, with a unary operator called a cast. [...] The precise meaning of a cast is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction.

The <math.h> library routine sqrt expects a double argument, and will produce nonsense if inadvertently handled something else. So if "n" is an interger, we can use:

```
sqrt((double) n)
```

to convert the value of n to double before passing it to sqrt.

Increment and Decrement Operators In a context where no value is wanted, just the incrementing effect, prefix and postfix are the same.

But there are situations where one or the other is specifically called for:

```
/* squeeze: delete all c from s */
void squeeze(char s[], int c) {
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}

/* this is exactly equivalent to */
    if (s[i] != c) {
        s[j] = s[i];
        j++;
    }
```

Bitwise Operators

```
/* getbits: get n bits from position p */
unsigned getbits(unsigned x, int p, int n) {
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

This bitwise section is all about understanding this return statement.

First getbits, extracts `n` bits starting from position `p` in the integer `x`, these extracted bits are then "right adjusted", meaning that they are shifted to the right end of the integer (the rest of the bits to the left are set to zero).

Example:

`x = 10110100` and you want to extract 3 bits starting from position 4, you would extract `101`. Then adjusting, would end up with `00000101`

`x >> (p+1-n)`: this shifts the desired part at the right end.

`~0`: this is a common idiom to inverts every bit.

`~0 << n`: This shifts all-1s value `n` positions to the left, which sets the `n` rightmost bits to 0 and the rest to 1.

`~(~0 << n)`: This inverts the bits, producing a mask, where only the `n` rightmost bits are set to 1, and the rest to 0.

`&`: apply the mask produced above, leaving only the `n`righmost its of the shifted `x`

---

x &= x - 1

The expression `x &= x - 1` is a clever and efficient way to count the number of 1 bits (set bits) in an integer.

Let consider what happens when you subtract 1 from a binary number:

`x - 1` - If LSB is 1 `001`, subtrating 1 will turn it to 0 `000` - If LSB is 0, the subtraction borrows from a more significant bit, flippingall trailing 0s to 1s until it reaches a 1, which is flipped to 0.

`x &=`

Performs a bitwise AND on x with the result of `x - 1`. This operation clears the least significant set bit.

```
x = 12; /* 1100 */
/* x - 1 = 11;  1011 */

x &= x - 1; /* 1100 & 1011 = 1000 (8 in decimal) */

/* x - 1 = 7;  0111 */
x &= x - 1 /* 1000 & 0111 = 0000 */
```

Control Flow

> "... it's a good idea to use braces when there are nested 'ifs' "
>
> "... the last 'else' part handles the 'none of the above' or default case where none of the other conditions is satisfied."

Switch

> "The break statement causes an immediate exit from the switch. Because cases serve just as labels, after the code for one case is done, execution falls through to the next unless you take explicit action to escape. break and return are the most common ways to leave a switch."

Loops

> "One final C operator is the comma","", which most often finds use in the for statement. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand."

```c
#include <stdio.h>
void reverse(char s[]) {
    int c, i, j;
    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

```c
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

Break & Continue It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom (for/while loop or do/while loop). The break statement provides an early exit from for, while, and do. A break causes the innermost enclosing loop or switch to be exited immediately.

The continue statement is related to break, but less often used; it causes the next iterations of the enclosing for, while, or do loop to begin. In the while and do, this means that the test part is executed immediately; in the for, control passes to the increment step.

Goto and labels

> "The are a few situations where goto(s) may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop."

Function

"C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones"

"Coercion: in the context of function type declarations refers to the automatic conversion of a value from one type to another."

"[. . .] Because C arrays begin at position zero, indexes will be zero or positive, and so a negative value like -1 is convenient for signaling failure."

"A program is just a set of definition of variables and functions. Communication between the functions is by arguments and values returned by the functions, and through external variables."

External Variables

"A C program consists of a set of external objects, which are either variables or functions. The adjective 'external' is used in contrast to 'internal', which describes the arguments and variables defined inside functions."

"We will see later how to define external variables and functions that are visible only within a single source file. Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicationg data between functions. Any function may access an external variable by referring to it by name, if the name has been declared somehow."

"External variables are also useful because of their greater scope and lifetime. Automatic variables are internal to a function; they come into existence when then function is entered, and disappear when it is left. External variables, on the other hand, are permanent, so they can retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, it is often most convenient if the shared data is kept in external variables rather than being passed in and out via arguments."

Scope Rules It is important to distinguish between the declaration of an external variable and its definiton. A declaration announces the properties of a variable (primarly its type); a definiton also causes storage to be set aside.

If lines:

```
int sp;
double val[MAXVAL];
```

appear outside of any function, they define the external variables sp and val, cause storage to be set aside, and also serve as the declarations for the rest of that source file. On the other hand, the lines

```
extern int sp;
extern double val[];
```

declare for the rest of the source file that sp is an int and that val is a double array (whose size is determined elsewhere), but they do not create the variables or reserve storage for them.

There must be only one definition of an external variable among all the files that make up the source program; other files may contain extern declarations to acces it.

```
/* in file1: */

extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }

/* in file2: */
int sp = 0;
double val[MAXVAL];
```

source: page 73.

Static Variables

> "static storage class should be used only when a program requires the value of a varaible to persist between different function calls, like in recursive calls" Exploring C - YPK

The static declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled. External static thus provides a way to hide names like `buf` and `bufp` in the getch-ungetch combination, which must be external so they can be shared, yet which should not be visible to users(place where function is call) of getch and ungetch.

source: page 75.

Initialization In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic and register variables have undefined (i.e, garbage) initial values.

For external and static variables, the initializer must be a constant expression;

For automatic and register variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls.

In effect, initialization of automatic variables are just shorthand for assignement statements. Which form to prefer is largely a matter of taste. We have generally used explicit assignments, because initializes in declarations are harder to see and further away from the point of use.

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas.

Character arrays are a special case of initialization; a string may be used instead of the braces and commans notation:

```
char pattern[] = "ould";
/* is a shorthand for the longest but equivalent */
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

Recursion C functions may be used recursively; that is, a function may call itself either directly or indirectly.

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set.

The C Preprocessor C provides certain language facilities by means of a preprocessor, which is conceptionally a separate first step in compilation.

Macro Substitution

```
#define name replacement text
```

It calls for a macro substitution of the simplest kind - subsequent occurences of the token name will be replaced by the replacement text.

The scope of a name defined with #define is from its point of definition to the end of the source file being compiled.

It is also possible to define macros with arguments, so the replacement text can be different for different calls of the macro.

```
#define max(A, B) ((A) > (B) ? (A) : (B))

x = max(p+q, r+s);
/* will be replaced by the line */
x = ((p+q) > (r+s) ? (p+q) : (r+s))
```

Note, macros are essentially textual replacements, ensure proper precedence by adding parentheses to the macro definiton

```
#define square(x) x * x

square(z+1) /* => z + 1 * z + 1, wrong precedence */
```

Pointers The `void *` pointer is the proper type for a generic pointer.

As a pointer, `void *` can point to any data type, but you cannot directly dereference a `void *` without first casting it to another pointer type that points to a specific data type.

Before the adoption of `void *` as a generic pointer, `char *` was frequently used for this purpose. This is because a `char` in C is defined to be 1 byte, and `char *` can access each byte of memory individually.

Using `void *` pointer is like positioning a cursor at a starting point in memory. At this stage, the pointer simply indicates where something is located but doesn't provide information about what exactly is at that location or how it should be interpreted.

The unary operator `&` gives the address of an object. This operator only applies to objects in memory: variables and array elements.

The unary operator `*` is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to.

Pointers and Function arguments Since C passes argument to functions by value, there is no direct way for the called function to alter a variable on the calling function. The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed.

Pointers and Arrays Any operations that can be achieved by array subscripting can also be done with pointers.

The correspondence between indexing and pointer arithmetic is very close. By definiton, the value of a variable or expression of type array is the address of element zero of the array. Since the name of an array is a synonym for the location of the initial element, the assignment `pa=&a[0]` can also be written as `pa = a`;

Rather more surprising, at first sight, is the fact that a reference to a[i] can also be written as *(a+1). In evaluating a[i], C converts it to* (a+i) immediately; the two forms are equivalent.

In C, `[ ]` is metaphorically equivalent to `* +` becoming `* (array_name + subscript)`

In short, an array-and-index expression is equivalent to one written as a pointer and offset.

An array name has a special behavior that distinguishes it from regular variables. Instead, it acts as a constant pointer to the first element of the array. The key distinction is that while 'a' behaves like a pointer, you cannot change its value. That is, 'a' is a constant pointer, 'a' is not a modifiable value.

The array name 'a' gives the address of the first element, but itself is not stored anywhere as a separate entity. It's more of a compile-time constructthat refers to the memory location where the array elements are stored.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.

```
/* as formal parameters in a function definition, */
char s[];
/* and */
char *s;
/* are equivalent; we prefer the latter because it says more
 * explicitly that the variable is a pointer */
```

Address Arithmetic

```
/* strlen: return length of string s */
int strlen(char *s) {
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

In the while loop the pointer increment until is own address is equal to `\0`

Character Pointers & Functions A string constant is accessed by a pointer to its first element.

```
char *pmessage;

/* assigns to pmessage a pointer to the character array */
pmessage = "now is the time";
```

There is an important difference between these defintions:

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time";  /* a pointer */
```

Arrays have their own storage, while pointers reference existing storage, typically read-only in the case of string literals. Arrays and pointers are fundamentally different in C, although they can be accessed similarly in many contexts. The main differences are the memory allocation and size. The similarities are, accessing elements (both can be used to access elements using the subscript operator `[]`, notation, and (pointer)arithmetic.

In C, an assignement operations returns the value that was assigned.

```c
/* strcpy: copy t to s; pointer version 2 */
void strcpy(char *s, char *t) {
    while ((*s++ = *t++) != '\0')
        ;
}
```

Interesting idiom, frequently used in C:

```c
/* the comparison against '\0' is redundant, since the question
* is merely whether the expresion is zero */

/* strcpy: copy t to s: pointer version 3 */
void strcpy(char *s, char *t) {
    while (*s++ = *t++)
        ;
}
/* '\0' has an integer value of 0 */
```

Standard idiom for pushing and popping a stack:

```c
    *p++ = val;      /* push val onto stack */
    val = *--p;     /* pop top of stack into val */
```

NOTE:

It's great practice to `return 0` indicating equality.

```c
int strncmp(char *s, char *t, int n) {
    for ( ; *s == *t; s++, t++)
        if (*s == '\0' || --n <= 0)
            return 0;
    return *s - *t;
}
```

In this case, if either condition is met, (`*s == '\0' || --n <= 0`), the function return 0, indicating that the strings are equal. Else we return the difference (`*s - *t`) indicating whether the character from 's' is greater than or less than.

Pointer Arrays; Pointers to pointers

```c
    char *lineptr[MAXLINES]
```

This line says that lineptr is an array of MAXLINES elements, each element of which is a pointer to a char.

Initialization of Pointer Arrays Ideal application for an internal `static` array.

```c
/* month_name: return name of n-th month */
char *month_name(int n) {
    static char *name[] = {
```

```
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

The `static` keyword ensures that the array `name` retains its value across multiple calls of the same function. This means that the array is only initialized once.

Command-line Arguments

> "In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When main is called, it is called with two arguments. The first (conventionally called argc, for argument count) is the number of command-line arguments the program was invoked with; the second (argv, for argument vector) is a pointer to an array of character strings that contain the arguments, one pers string."
>
> "A common convention for C programs on UNIX systems is that an arguments that begins with a minus sign introduces an optional flag or parameter. If we choose -x (for except) to signal the inversion, and -n (number) to request line numbering, then the command `find -x -n pattern` will print each line that doesn't match the pattern, preceded by its line number"

How the -x flag work. Here the table of truth: (code page 105)

`strstr(line, *argv) != NULL`	`except`	`!= except`	Result
1 (substring found)	0	1 != 0	1 (print line)
0 (substring not found)	0	0 != 0	0 (skip line)
1 (substring found)	1	1 != 1	0 (skip line)
0 (substring not found)	1	0 != 1	1 (print line)

Pointers to Functions

> "A sort often consists of three parts - a comparison that determines the ordering of any pair of objects, an exchange that reverses their order, and a sorting algorithm that makes comparisons and exchanges until the objects are in order."
>
> "Any pointer can be cast to void * and back again without loss of information, so we can call qsort by casting arguments to void *."

`int (*comp)(void *, void *)` says that comp is a pointer to a function that has two void * arguments and returns an int.

whereas `int *comp(void *, void *) /* wrong */` says that comp is a function returning a pointer to an int, which is very different.

Complicated Declarations

```
int *f();   /* f: function returning pointer to int */

int (*pf)();    /* pf: pointer to function returning int */
```

'dcl' -> declaration

> "It's good to know that 'declaration' can be quite an art to understand, with all their subtleties."

```
char **argv;    /* equivalent of char *argv[];  */

int array1[13];
int array2[13];
int (*daytab)[13];  /* daytab is a pointer to array of 13 integers */
daytab = &array1;
daytab = &array2;

int *daytab[13];    /* daytab: array[13] of pointer to int */

void *comp() {} /* comp: function returning pointer to void */

void (*comp)() {}   /* comp: pointer to function returning void */

char (*(*x())[])();
/* x: function returning pointer to array[] of pointer to function returning char */
char (*(*x[3])())[5];
/* x: array[3] of pointer to function returning pointer to array[5] of char */
```

`char (*(*x())[])()` "x is a function returning a pointer to an array of pointers to functions returning char"

Structures

> "A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures help to organize complicated data, particulary in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities."

```
/* Basics of Structures
 * Let us create a basic point object
 * Two components can be placed in a structure declared like this:
 */
```

```c
struct point {
    int x;
    int y;
};
```

The keyword `struct` introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a **structure tag** may follow the word `struct`. The tag names this king of structure, and can be used subsequently as a shorthand for the part of the declaration in braces. The variables named in a structure are called members.

A `struct` declarations defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```c
struct { ... } x, y, z;
/* is syntactically analogous to */
int x, y, z;
/* in the sense that each statement declares x, y and z to be
 * variables of the named type and causes space to be set aside for them
 */
```

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or shape of a structure. If the declaration is tagged, however, the tag can be used later in definitions of instances of the structure.

For example, `struct point pt;` defines a variable pt which is a structure of type `struct point`

A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members: `struct maxpt = { 320, 200 };`

The structure member operator `.` connects the structure name and the member name. To print the coordinates of the point pt:

```c
printf("%d, %d", pt.x, pt.y);
```

Structures can be nested.

```c
struct rect {
    struct point pt1;
    struct point pt2;
};

struct rect screen;

screen.pt1.x;   /* refers to the x coordinate of the pt1 member of screen */
```

Structures and Functions

"The only legal operations on a structure are copying it (as a unit/member by member), taking its address with &, and accessing its members."

"The . operator is used to access members of a structure, by adding the specific offset on the base address."

"A structure name ('s') represents the entire block of memory for the structure" "You cannot use ('s') as a pointer to the first member unlike array. Instead, you use the address-of operator ('&') to get a pointer to the structure."

"Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then `p->member-of-structure`

```
struct rect r, *rp = &r;
/* then these four expressions are equivalent: */
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Arrays of Structures

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];

/* this could also be written */
struct key {
    char *word;
    int count;
};
struct key keytab[NKEYS];
```

Since the structure `keytab` contains a constant set of names, it is easiest to make it an external variable and initialize it once for all when it is defined. The structure initialization is analogous to earlier ones - the definition is followed by a list of initializers enclosed in braces:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
```

```
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0
};
/* as usual, the number of entries in the array will be computed
 * if the initializers are present and the [ ] is left empty. */
```

Self-referential Structures The binary tree data structure use a self-reference.

```
struct tnode {              /* the tree node:           */
    char *word;             /* points to the text       */
    int count;              /* number of occurrences    */
    struct tnode *left;     /* left child               */
    struct tnode *right;    /* right child              */
```

Table Lookup When a line like `#define IN 1` is encountered, the name IN and the replacement text 1 are stored in a table. Later, when IN appears in a statement like `state = IN;` it must be replaced by 1.

There are two routines that manipulate the names and replacement texts.

- `install(s,t)` records the name s and the replacement text in a table
- `lookup(s)` search for s in the table

A block in the list is a structure containing pointers to the name, the replacement text, and the next block in the list.

```
struct list {               /* table entry:         */
    struct nlist *next;     /* next entry in chain  */
    char *name;             /* defined name         */
    char *defn;             /* stand for definiton: replacement text */
};
```

Typedef C provides a facility called `typedef` for creating new data type names.

```
/* for example, the declaration */
typedef int Length;
/* makes the name Length a synonym for int */
/* the type Length can be used in declarations,
* casts, etc, in exaclty the same ways that the int type can be */
```

```
Length len, maxlen;
Length *lengths[s];
```

Notice that the type being declared in a `typedef` appears in the position of a variable name, not right after the word `typedef`. Syntactically, `typedef`is like the storage classes `extern`, `static`, etc. We have used capitalized names for `typedef`s, to make them stand out.

> "It must be emphasized that a `typedef` declaration does not create a new type in any sense; it merely adds a new name for some existing type."

Besides purely aesthetic issues, there are two main reasons for using `typedef`s:

- The first is to parameterize a program against portability problems.
- The second is to provide better documentation for a program.

Unions A **union** is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage.

> "In effect, a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the"widest" member, and the alignment is appropriate for all of the types in the union."
>
> "A union may only be initialized with a value of the type of its first member."
>
> "Unions, like structures, are derived datatypes which group together a number of variables. However, the way the two treat these variables is totally different. While the elements of a structure enable us to access different locations in memory, the elements of a union serve as different names by which the same portion of memory can be accessed."

Bit-fields

> "Using bit fields in C is analogous to creating a custom-tailored word"
>
> "The `:` operator is known as the bit field width specifier, it's used to specify the number of bits that a particular field within a structure will occupy"

```
struct {
    unsigned int field1 : 3;
    unsigned int field2 : 5;
```

```
    unsigned int field3 : 6;
};
```

- `field1` will use 3 bits, `field2` will use 5 bits etc...

  "When you use the `:` operator to define bit fields in a struct, the compiler packs these field tightly together"

Chapter 7 - Input and Output

> "We will not present the entire library here, since we are more interested in writing C programs that use it."
>
> "The library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the system doesn't operate that way, the library does whatever necessary to make it appear as if it does."

Variable-length Argument Lists This section contains an implementation of a minimal version of `printf`, to show how to write a function that processes a variable-length argument list in a portable way.

```
/* the proper declaration for printf is */
int printf(char *fmt, ...);
/* where the declaration ... means that the number and types of these
 * arguments may vary */

/* minprintf is declared as */
void minprintf(char *fmt, ...);
/* since we will not return the character count that printf does. */
```

The tricky bit is how `minprintf` walks along the argumnent list when the list doesn't even have a name. The standard header `<stdarg.h>` contains a set of macro definitions that define how to step through an argument list. The implementation of this header will vary from machine to machine, but the interface it presents is uniform.

The type `va_list` is used to declare a variable that will refer to each argument in turn; in `minprintf`, this variable is call `ap` for "argument pointer". The macro `va_start` initializes `ap` to point to the first unnamed argument. It must be called once before `ap` is used. There must be at least one named argument; the final named argument is used by `va_start` to get started.

See: minprintf.c

Formatted Input - Scanf

> "`scanf` stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the

number of succesfully matched and assigned input items. This can be used to decide how many items were found."

"There is also a function `sscanf` that reads from a string instead of the standard input"

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

It scans the `string` according to the format in `format` and stores the resulting values through `arg1`, `arg2`, etc. These arguments must be pointers.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are not ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream
- Conversion specifications, consisting of the characters %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l or L indicating the width of the target, and a conversion character.

```c
#include <stdio.h>

int main() {
    const char *input = "123456 789 1234.56 7890";
    int first;
    double third;
    long int fourth;

    // Using sscanf to read the integers and floating-point numbers with specified width and
    // Suppress assignment of the second number
    if (sscanf(input, "%5d %*4hd %lf %ld", &first, &third, &fourth) == 3) {
        printf("First number (max width 5): %d\n", first);
        printf("Third number (double): %lf\n", third);
        printf("Fourth number (long int): %ld\n", fourth);
    } else {
        printf("Error reading input string.\n");
    }

    return 0;
}
```

File Access

"The examples so far have all read the standard input and written the standard ouput, which are automatically defined for a program by the local operating system."

> "The next step is to write a program that accesses a file that is not already connected to the program."

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

This say that `fp`is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. Notice that `FILE` is a type name, like `int`, not a structure tag; it is defined with a `typedef`

```
/* the call of fopen in a program is: */
fp = fopen(name, mode);

/* getc returns the next character from the stream referred by fp
 * it returns EOF for end of the file or error */
int getc(FILE *fp);

/* putc writes the character c to the file fp and returns the
 * character written, or EOF if an error occurs. */
int putc(int c, FILE *fp)
```

When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them. These files are the standard input, the standard ouput, and the standard error; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`, and are declared in `<stdio.h>`. Normally `stdin` is connected to the keyboard and `stdout` and `stdout` are connected to the screen, but `stdin` and `stdout` may be redirected to files or pipes...

`getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin` and `stdput` as follows:

```
#define getchar()   getc(stdin)
#define putchar()   putc((c, stdout)
```

For formatted input or output of files, the functions `fscanf` and `fprintf` may be used. These are identical to `scanf` and `printf`, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Error Handling - Stderr and Exit

> "`stderr`is not a traditional file; it is a stream."

> "`exit` terminates program execution when it is called. The argument of `exit` is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process. Conventionally, a return value of 0 signals

that all is well; non-zero values usually signal abnormal situtations. `exit` calls `fclose` for each open output file, to flush out any buffered output.

"within `main`, `return` expr is equivalent to `exit(expr)`."

We have generally not worried about exit status in our small illustrative programs, but any serious program should take care to return sensible, useful status values.

Line Input and Output The standard library provides an input and output routine `fgets` that is similar to the `getline` function that we have used in earlier chapter.

```
char *gets(char *line, int maxline, FILE *fp)
```

`fgets` reads the next input line (including the newline) from file `fp` into the character array `line`; at most `maxline-1` characters will be read.

Normally `fgets` return `line`; on end of file or error it return `NULL`.

Miscellaneous functions The standard library most useful functions:

- `s`, `t` are `char*`
- `c`, `n` are `int`

`<string.h>` Functions Summary

Function	Description
`strcat(s, t)`	Concatenate `t` to the end of `s`
`strncat(s, t, n)`	Concatenate `n` characters of `t` to the end of `s`
`strcmp(s, t)`	Return negative, zero, or positive for `s < t`, `s == t`, `s > t`
`strncmp(s, t, n)`	Same as `strcmp` but only for the first `n` characters
`strcpy(s, t)`	Copy `t` to `s`
`strncpy(s, t, n)`	Copy at most `n` characters of `t` to `s`
`strlen(s)`	Return length of `s`
`strchr(s, c)`	Return pointer to the first occurrence of `c` in `s`, or `NULL` if not present
`strrchr(s, c)`	Return pointer to the last occurrence of `c` in `s`, or `NULL` if not present

`<ctype.h>` Functions Summary

- `c` is `int` that can be represented as an unsigned char or `EOF`
- Function returns `int`

Function	Description
`isalpha(c)`	Non-zero if `c` is alphabetic, 0 if not
`isupper(c)`	Non-zero if `c` is upper case, 0 if not
`islower(c)`	Non-zero if `c` is lower case, 0 if not
`isdigit(c)`	Non-zero if `c` is a digit, 0 if not
`isalnum(c)`	Non-zero if `isalpha(c)` or `isdigit(c)`, 0 if not
`isspace(c)`	Non-zero if `c` is blank, tab, newline, return, formfeed, or vertical tab
`toupper(c)`	Return `c` converted to upper case
`tolower(c)`	Return `c` converted to lower case

`<math.h>` Functions Summary

- All functions take one or two `double` arguments and return a `double`.

Function	Description
`sin(x)`	Sine of `x`, where `x` is in radians
`cos(x)`	Cosine of `x`, where `x` is in radians
`atan2(y, x)`	Arctangent of `y/x`, in radians
`exp(x)`	Exponential function (e^x)
`log(x)`	Natural (base e) logarithm of `x` (x > 0)
`log10(x)`	Common (base 10) logarithm of `x` (x > 0)
`pow(x, y)`	(x^y)
`sqrt(x)`	Square root of `x` (x > 0)
`fabs(x)`	Absolute value of `x`

The UNIX System Interface The UNIX operating system provides its services through a set of system call, which are in effect functions within the operating system that may be called by user programs.

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even keyboard and screen, are files in the file system. This means that a single homogeneous interface handles all communication between a program and peripheral devices.

Since input and output involking keyboard and screen is so commom, special arragements exist to make this convenient. When the command interpreter (the "shell") runs a programs, three files are open, with file descriptos 0, 1, and 2, called the standard input, the standard output, and the standard error. If a program reads 0, and writes 1 and 2, it can do input and output without worrying about opening files.

The user of a program can redirect I/O to and from files with `<` and `>`: `prog <infile >outfile`

In this case, the shell changes the default assignements for the file descriptors 0 and 1 to the named files.

> "In all cases, the file assignements are changed by the shell, not by the program"

Low level I/O - Read and Write In the UNIX file system, there are nine bits of permission information associated with a file that control read, write and execute access for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is convenient for specifying the permissions.

Mapping Permissions to Binary and Octal:

- r (read) = 4 (100 in binary)
- w (write) = 2 (010 in binary)
- x (execute) = 1 (001 in binary)

System Call	Return Value	Description
`open`	>= 0	On success, returns a file descriptor (a non-negative integer).
	-1 and sets `errno`	On failure, returns -1 and sets `errno` to indicate the error.
`read`	> 0	On success, returns the number of bytes read.
	0	On end-of-file, returns 0.
	-1 and sets `errno`	On failure, returns -1 and sets `errno` to indicate the error.
`creat`	>= 0	On success, returns a file descriptor (a non-negative integer).
	-1 and sets `errno`	On failure, returns -1 and sets `errno` to indicate the error.
`write`	>= 0	On success, returns the number of bytes written.
	-1 and sets `errno`	On failure, returns -1 and sets `errno` to indicate the error.

> "Termination of a program via `exit` or return from the main program closes all open files"

Random Access - Lseek The system call `lseek` provides a way to move around in a file without reading or writing any data;

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

sets the current position in the file whose descriptor is `fd` to `offset`, which is taken relative to the location specified by `origin`.

For example, the following function reads any number of bytes from any arbitrary place in a file. It returns the number read, or -1 on error

```
#include "syscalls.h"

/* get: read n bytes from position pos */
int get(int fd, long pos, char *buf, int n) {
    if (lseek(fd, pos, 0) >= 0) /* get to pos */
        return read(fd, buf, n);
    else
        return -1;
}
```

The return value from `lseek` is a long that gives the new position in the file, or -1 if an error occurs.

Listing Directories

> "Let us begin with a short review of UNIX file system structure. A directory is a file that contains a list of filenames and some indication of where they are located. The 'location' is an index into another table called the 'inode list'. The *inode* for a file is where all information about the file except its name is kept. A directory entry generally consists of only two items, the filename and an inode number."