Enumerations

- In many programs, we'll need variables that have only a small set of meaningful values.
- C provides a special kind of type designed specifically for variables that have a small number of possible values.
- An ***enumerated type*** is a type whose values are listed ("enumerated") by the programmer.

  Each value must have a name (an ***enumeration constant***).
- For example, a variable that stores the suit of a playing card should have only four potential values: "clubs," "diamonds," "hearts," and "spades."

  enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
- The names of enumeration constants must be different from other identifiers declared in the enclosing scope.
- If an enumeration is declared inside a function, its constants won't be visible outside the function.

Enumeration Tags and Type Names

- As with structures and unions, there are two ways to name an enumeration: by declaring a tag or by using typedef to create a genuine type name.
- Enumeration tags resemble structure and union tags:

```
enum suit {CLUBS, DIAMONDS, HEARTS,
           SPADES};
```

- `suit` variables can be declared in the following way:

```
enum suit s1, s2;
```

- As an alternative, we could use typedef to make Suit a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS,
              SPADES} Suit_type;

Suit_type s1, s2;
```

Enumerations as Integers

- Behind the scenes, C treats enumeration variables and constants as integers.

  By default, the compiler assigns the integers 0, 1, 2, … to the constants in a particular enumeration.

  In the suit enumeration, CLUBS, DIAMONDS, HEARTS, and SPADES represent 0, 1, 2, and 3, respectively.
- The programmer can choose different values for enumeration constants:

```
enum suit {CLUBS = 1, DIAMONDS = 2,
           HEARTS = 3, SPADES = 4};
```

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant. The first enumeration constant has the value 0 by default.

Chp. 17 Advanced Uses of Pointers

- C supports *dynamic storage allocation:* the ability to allocate storage during program execution.
- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.
- Dynamic storage allocation is done by calling a memory allocation function.

  <stdlib.h> declares memory allocation functions:

  `malloc` — Allocates a block of memory but doesn't initialize it.

  `calloc` — Allocates a block of memory and clears it.

  `realloc` — Resizes a previously allocated block of memory.

  These functions return a value of type `void *`, a "generic" pointer.
- If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer.***

  After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

  An example of testing malloc's return value:

```
p = malloc(10000);
if (p == NULL) {// allocation failed
  . . .
}
else {
  . . .
}
```

  NULL is a macro (defined in various library headers) that represents the null pointer.
- Pointers test true or false in the same way as numbers.

  Statement `if (p == NULL) …`
  is equivalent to `if (!p) …`

  Statement `if (p != NULL) …`
  is equivalent to `if (p) …`

- Prototype for the **malloc** function:

```
void *malloc(size_t size);
```

malloc allocates a block of <u>size bytes</u> and returns a pointer to it. size_t is an unsigned integer type defined in the library.

- Prototype for **calloc**:

```
void *calloc(size_t nmemb, size_t size);
```

- Properties of calloc:
  - Allocates space for an array with nmemb elements, each of which is size bytes long.
  - Returns a null pointer if the requested space isn't available.
  - Initializes allocated memory by setting all bits to 0.
- Example: use malloc or calloc to allocate:

```
int *a, *b, n=100;
a = malloc(n*sizeof(int)) ;
b = calloc(n, sizeof(int));
```

- Example: use malloc or calloc to allocate space for structures:

```
struct point { int x, y; } *p, *q;
p = malloc(sizeof(struct point));
q = calloc(1, sizeof(struct point));
```

- The **realloc** function can resize a dynamically allocated array. Prototype for realloc:

```
void *realloc(void *ptr, size_t size);
```

`ptr` must point to a memory block obtained by a previous call of malloc, calloc, or realloc.

`size` represents the new size of the block, which may be larger or smaller than the original size.

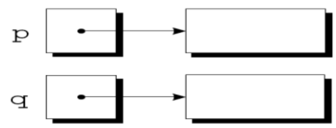
- Properties of realloc:
  - When it expands a memory block, realloc doesn't initialize the bytes that are added to the block.
  - If realloc can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
  - If realloc is called with a null pointer as its first argument, it behaves like malloc.
  - If realloc is called with 0 as its second argument, it frees the memory block.
- We expect realloc to be reasonably efficient:
  - When asked to reduce the size of a memory block, realloc should shrink the block "in place."
  - realloc should always attempt to expand a memory block without moving it.

    If it can't enlarge a block, realloc will allocate a new block elsewhere, then copy the contents of the old block into the new one.
- Once realloc has returned, be sure to update all pointers to the memory block in case it has been moved.
- Memory allocation functions obtain memory blocks from a storage pool known as the ***heap.***

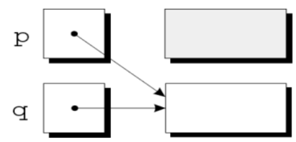
  Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.
- To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space.
- Example:

```
p = malloc(…);
q = malloc(…);
p = q; // memory leak!
```

A snapshot after the first two statements have been executed:



After the 3rd statement, both variables now point to the second memory block:



There are no pointers to the first block, so we'll never be able to use it again.

- A block of memory that's no longer accessible to a program is said to be ***garbage.*** A program that leaves garbage behind has a ***memory leak.***

  Some languages provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.

  Instead, each C program is responsible for recycling its own garbage by calling the free function to release unneeded memory.
- Prototype for **free**:

```
void free(void *ptr);
```

- For the example code above that has memory leak, a version without memory leak:

```
p = malloc(…);
q = malloc(…);
free(p) ;
p = q;
```

Calling free releases the memory that p points to.

- Using free leads to a new problem: *dangling pointers*.
- free(p) deallocates the memory block that p points to, but doesn't change p itself.
- If we forget that p no longer points to a valid memory block, chaos may ensue:

```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");/*** WRONG ***/
```

Pointers to Pointers

- One use of pointers to pointers is for two-dimensional arrays, including arrays of strings.

```
int **a, n=100, i;
unsigned int *b;

b = (int *)calloc(n, sizeof(int));
b[0] = ... ;
b[1] = ... ;
 . . .
b[99] = ...;

a = (int **)calloc(n, sizeof(int *));
for(i=0; i<n; i++)
   a[i]=(int *)calloc(b[i],sizeof(int));
. . .
```

- Another use is to allow the change the pointer argument to a new address for a function.

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

struct node *add_to_list(
            struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if(new_node == NULL) {
    printf("Error: malloc failed\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
 void main() {
   int n ;
   struct node *new_node, *first ;

   first = malloc(sizeof(struct node));
    . . .
   n = 123 ;
   new_node = add_to_list(first, n) ;
   first = new_node ;
    . . .
 }
```

- Another version:

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

void add_to_list(struct node **list,
                  int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if(new_node == NULL) {
    printf("Error: malloc failed\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = *list;
  *list = new_node;
}

void main() {
  int n ;
  struct node *first ;

  first = malloc(sizeof(struct node)) ;
   . . .
  n = 123 ;
  add_to_list(&first, n) ;
   . . .
}
```

Pointers to Functions

- It's also possible to have pointers to *functions.*
- Functions occupy memory locations, so every function has an address. We can use function pointers in much the same way we use pointers to data.
- Passing a function pointer as an argument is common.
- Example:

```
double integrate(double (*f)(double),
                 double a, double b) {
   int i, n=1000 ;
   double delta, result = f(a) ;
   delta = (b-a)/n ;
   for(i=1; i<=n; i++)
      result += f(a + i*dalta) ;
   return result ;
}

void main() {
  . . .
  result = integrate(sin, 0.0, 3.14159);
  . . .
}
```

- Within the body of integrate, we can call the function that f using either `f(x)` or `(*f)(x)`.
- C treats pointers to functions just like pointers to data.

  They can be stored in variables or used as elements of an array or as members of a structure or union.

  It's also possible for functions to return function pointers.
- Example: A variable that can store a pointer to a function with an double parameter and a return type of double:

  ```
  double (*pf)(double);
  ```

  We can make pf point to a function as in the following:

  ```
  pf = sin;
  ```

  We can now call f by writing either

  ```
  x = (*pf)(1.0);//or
  x = pf(1.0);
  ```

Restricted Pointers (C99)

`int * restrict p;` // p is a ***restricted pointer.***

- The intent is that if p points to an object that is later modified, then that object is not accessed in any way other than through p.
- Consider the following code:

  ```
  int * restrict p;
  int * restrict q;
  p = malloc(sizeof(int));
  ```

  Normally it would be legal to copy p into q and then modify the integer through q:

  ```
  q = p;
  *q = 0;  /* undefined behavior */
  ```

  Because p is a restricted pointer, the effect of executing the statement *q = 0; is undefined.

Chp. 18 Declarations

- Declarations furnish information to the compiler about the meaning of identifiers. General form of a declaration:

  declaration-specifiers declarators

  - ***Declaration specifiers*** describe the properties of the variables or functions being declared.
  - ***Declarators*** give their names and may provide additional information about their properties.

  Examples:

  ```
  int i;
  float f(float);
  ```

- Declaration specifiers fall into three categories:
  - Storage classes: auto, static, extern, register. At most one storage class may appear in a declaration; if present, it should come first.
  - Type qualifiers:

    In C89, two type qualifiers***:*** const and volatile

    C99 has a third type qualifier, *restrict*.

    A declaration may contain zero or more type qualifiers.
  - Type specifiers: *void, char, short, int, long, float, double, signed, unsigned* are type specifiers. The order in which they are combined doesn't matter.

    For instance, *int unsigned long* is the same as *long unsigned int*.

    Type specifiers also include specifications of structures, unions, and enumerations.

    typedef names are also type specifiers.
  - (C99 only) ***function specifier:*** keyword `inline`
- Type qualifiers and type specifiers should follow the storage class, but no other restrictions on their order.

- Declarators include:
  - Identifiers //names of simple variables
  - Identifiers followed by [] // array names
  - Identifiers preceded by * // pointer names
  - Identifiers followed by () // function names
- Declarators are separated by commas.

storage class — declarators

```
static float x, y, *p;
```

type specifier

- A declarator that represents a variable may be followed by an initializer.

type qualifier — declarator

```
const char month[] = "January";
```

type specifier — initializer

- Function declarations may have a storage class, type qualifiers, and type specifiers:

storage class — declarator

```
extern int square(int);
```

type specifier

Properties of Variables

- Every variable in a C program has three properties:
  - Storage duration: determines when memory is set aside for the variable and when that memory is released.
  - Scope: the portion of the program text in which the variable can be referenced.

    block scope, file scope
  - Linkage: determines the extent to which it can be shared.

    external, internal, no linkage.

```
int i;      static storage duration
            file scope
            external linkage

void f(void)
{
  int j;    automatic storage duration
            block scope
            no linkage
}
```

```
static int i;   static storage duration
                file scope
                internal linkage

void f(void)
{
  static int j;   static storage duration
                  block scope
                  no linkage
}
```

The Storage Class of a Function

- Function declarations and definitions may include a storage class. The options are extern and static:
  - extern specifies that the function has external linkage, allowing it to be called from other files.
  - static indicates internal linkage, limiting use of the function's name to the file in which it's defined.
  - If no storage class is specified, the function is assumed to have external linkage.
- Function parameters have the same properties as auto variables: automatic storage duration, block scope, and no linkage.

  The only storage class that can be specified for parameters is register.
- Example:

```
int a;
extern int b;
static int c;
void f(int d, register int e){
  auto int g;
  int h;
  static int i;
  extern int j;
  register int k;
}
```

Name	*Storage Duration*	*Scope*	*Linkage*
a	static	file	external
b	static	file	†
c	static	file	internal
d	automatic	block	none
e	automatic	block	none
g	automatic	block	none
h	automatic	block	none
i	static	block	none
j	static	block	†
k	automatic	block	none

† In most cases, b and j will be defined in another file and will have external linkage.

- static and extern are important storage classes. auto has no effect. register is less important nowadays.

Deciphering Complex Declarations

- ***When there's a choice, always favor* [] *and* () *over* *.** Parentheses can be used to override the normal priority of [] and () over *.
- Example 1:

```
int *ap[10];
```

ap is an *array* of *pointers*.

- Example 2:

```
float *fp(float);
```

fp is a *function* that returns a *pointer*.

- Example 3:

```
float (*pf)(float);
```

Since *pf is enclosed in parentheses, pf must be a pointer.

But (*pf) is followed by (float), so pf must point to a function with a float argument.

The word float at the beginning represents the return type of this function.

Certain things can't be declared in C.

- Functions can't return arrays:

  int f(int)[]; /*** WRONG ***/
- Functions can't return functions:

  int g(int)(int); /*** WRONG ***/
- Arrays of functions aren't possible, either:

  int a[10](int); /*** WRONG ***/
- A function can't return an array, but it can return a *pointer* to an array.

Initializers

- C allows us to specify initial values for variables as we're declaring them.

  To initialize a variable, we write the = symbol after its declarator, then follow that with an initializer.
- The initializer for a simple variable is an expression of the same type as the variable:

```
int i = 5/2;   // i is initially 2
```

- If the types don't match, C converts the initializer using the same rules as for assignment:

```
int j = 5.5;     // converted to 5
```

- The initializer for a pointer variable must be an expression of the same type or of type void *:

```
int *p = &i;
```

- The initializer for an array, structure, or union is usually a series of values enclosed in braces:

```
int a[5] = {1, 2, 3, 4, 5};
```

- In C99, brace-enclosed initializers can have other forms, thanks to designated initializers.
- An initializer for a variable with static storage duration must be constant. For example,

```
static int n = 100 + 3/2 - 1 ;
```

  Question: Is it possible to declare a variable of static storage without using the keyword static?
- If a variable has automatic storage duration, its initializer need not be constant:

```
int f(int n){
  int last = n - 1;
  . . .
}
void g(struct part part1){
  struct part part2 = part1;
  . . .
}
```

Chp. 22 Input/Output

- C's input/output library is an important part of the standard library.

  The <stdio.h> header is the primary repository of input/output functions, including printf, scanf, putchar, getchar, puts, and gets.

  Functions in <stdio.h> that read or write data are known as ***byte input/output functions.***
- In C, the term ***stream*** means any source of input or any destination for output.
- Some programs obtain all their input from one stream (the keyboard) and write all their output to another stream (the screen). Larger programs may need additional streams.
- Streams often represent files stored on various media. However, they could be associated with devices such as network ports and printers.
- <stdio.h> provides three standard streams:

File Pointer	*Stream*	*Default Meaning*
stdin	Standard input	Keyboard
stdout	Standard output	Screen
stderr	Standard error	Screen

- These streams are ready to use—we don't declare them, and we don't open or close them.

File Pointers

- Accessing a stream is done through a ***file pointer,*** which has type FILE *, and is declared in <stdio.h>.
- Certain streams are represented by file pointers with standard names, e.g. stdin, stdout, stderr.

  Otherl file pointers can be declared as needed:

```
FILE *fp1, *fp2;
```

Standard Streams and Redirection

- A typical technique for forcing a program to obtain its input from a file instead of from the keyboard:

```
demo < in.dat
```

  This technique is known as ***input redirection.***

- ***Output redirection*** is similar:

```
demo > out.dat
```

  All data written to stdout will now go into the out.dat file instead of appearing on the screen.

- Input redirection and output redirection can be combined:

```
demo < in.dat > out.dat
demo > out.dat < in.dat
```

- Writing error messages to stderr instead of stdout guarantees that they will appear on the screen even when stdout has been redirected.

File Operations

- Simplicity is one of the attractions of input and output redirection. Unfortunately, redirection is too limited for many applications.
  - When a program relies on redirection, it has no control over its files; it doesn't even know their names.
  - Redirection doesn't help if the program needs to read from two files or write to two files at the same time.

  When redirection isn't enough, we'll use the file operations that <stdio.h> provides.

- Opening a file for use as a stream requires a call of the fopen function. Prototype for fopen:

```
FILE *fopen(const char * filename,
            const char * mode);
```

  - filename is the name of the file to be opened. This argument may include information about the file's location, such as a drive specifier or path.
  - mode is a "mode string" that specifies what operations we intend to perform on the file.

- In Windows, be careful when the file name in a call of fopen includes the \ character. For example, the call

```
fopen("c:\project\test1.dat", "r");
```

  will fail, because \t is treated as a character escape.

  One way to avoid the problem is to use \\ instead of \:

```
fopen("c:\\project\\test1.dat", "r");
```

  An alternative is to use the / character instead of \:

```
fopen("c:/project/test1.dat", "r");
```

- fopen returns a file pointer that the program can (and usually will) save in a variable:

```
fp = fopen("in.dat", "r");
```

  When it can't open a file, fopen returns a null pointer.

- Factors that determine which mode string to pass to fopen:
  - Which operations are to be performed on the file
  - Whether the file contains text or binary data

- Mode strings for text files:

String	***Meaning***
"r"	Open for reading. The file must exist.
"w"	Open for writing (file need not exist)
"a"	Open for appending (file need not exist)
"r+"	Open for reading and writing, starting at beginning. The file must exist.
"w+"	Open for reading and writing (truncate if file exists)
"a+"	Open for reading and writing (append if file exists)

- Mode strings for binary files:

String	***Meaning***
"rb"	Open for reading
"wb"	Open for writing (file need not exist)
"ab"	Open for appending (file need not exist)
"r+b" or "rb+"	Open for reading and writing, starting at beginning
"w+b" or "wb+"	Open for reading and writing (truncate if file exists)
"a+b" or "ab+"	Open for reading and writing (append if file exists)

- It's possible to combine a call of fopen with the declaration of fp:

```
FILE *fp = fopen(FILE_NAME, "r");
```

- `freopen` attaches a different file to a stream that's already open.

  `freopen`'s normal return value is its third argument (a file pointer).

  If it can't open the new file, freopen returns a null pointer.

  The most common use of `freopen` is to associate a file with one of the standard streams (stdin, stdout, or stderr).

  Example: A call of freopen that causes a program to begin writing to the file `myfile`:

```
if(freopen("myfile","w", stdout)==NULL){
  . . .
}
else {
 . . .
}
```

Closing a File

- The `fclose` function allows a program to close a file that it's no longer using.
- The argument to `fclose` must be a file pointer obtained from a call of fopen or freopen.
- fclose returns zero if the file was closed successfully.
- Otherwise, it returns the error code EOF (a macro defined in <stdio.h>).
- Example code:

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

#define FILE_NAME "example.dat"

int main(void) {
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n",
FILE_NAME);
    exit(EXIT_FAILURE);
  }
      …
  fclose(fp);
  return 0;
}
```