PART 4

Structures, Unions, and Input/Output

- Casts
- `typedef`
- Structures
- Dynamic Memory Allocation
- Unions
- File I/O
- Understanding C Data Types
- More on Multidimensional Arrays
cast.c

/* CAST.C: Program illustrates the use of a cast to
   coerce a function argument to be of the correct form. */
#include <stdio.h>
#include <math.h>
/* The above include is present so that the return type
of the sqrt math function is properly declared, and the
value passed is converted, if necessary, to the correct type.
The function sqrt takes one double argument and returns a
double. We could have replaced the above #include with:
   extern double sqrt(double);
This would have the same effect. */

main()
{
    int n = 2;

    printf("%f\n", sqrt(n) ); /* int silently
converted
    to double because of declation in math.h */

    printf("%f\n", sqrt( (double)n ) ); /* better */
    return 0;
}
CASTS

- References: Kernighan and Ritchie, page 45; Kochan, pages 329-332.

- Explicit type conversions can be forced in any expression with a construct called a cast. In the construction
  
  \[(\text{type-name}) \text{ expression}\]

  the expression is converted to the named type.

- Example:
  
  ```c
  int n;
  double z;
  ...
  ...
  z = \text{sqrt}( (\text{double})n );
  /* n is cast as a double */
  ```

- In the above example, the cast is necessary in order to use the \text{sqrt} function properly. The function \text{sqrt} (and most other math functions) takes a double as an argument. An incorrect result will be obtained if an integer is passed to the \text{sqrt} function.

- Example:
  
  ```c
  char *p;
  double *z;
  ...
  ...
  z = (\text{double } *)p;
  /* the character pointer is cast into a pointer to a double */
  ```

- In the above example, the cast is necessary in order to avoid a type mismatch.
#include <stdio.h>

typedef int COUNT; /* define datatype = COUNT */
typedef float RATIO; /* define datatype = RATIO */
typedef char STRING[81]; /* define datatype = STRING*/

main()
{
    COUNT n = 3; /* declare n to be of type COUNT */
    RATIO x = 2.7; /* declare x to be of type RATIO */
    static STRING name = {"Johann Sebastian Bach"); /* declare name to be of type STRING and initialize*/

    printf("n = %d, x = %f\n", n, x);
    printf("%s\n", name);
    return 0;
}


**TYPEDEF**

- Reference: Kernighan & Ritchie - Pages 146-147; Kochan - Pages 326-329.

- C provides a capability that enables the programmer to assign an alternate name to a data type. This is done with a statement known as a **typedef**. The statement
  
  ```c
  typedef int COUNTER;
  ```
  
defines the name **COUNTER** to be equivalent to the C data type int. Variables can subsequently be declared to be of type **COUNTER**, as in the statement
  
  ```c
  COUNTER i,j;
  /* i j, are integer variables */
  ```

- The main advantage of the use of the **typedef** is that it adds readability to the definition of variables. The use of the **typedef** statement does not actually define a new type - only a new type name. Variables defined to be of the new type name are not treated any differently by the C compiler.

- Another example of the use of the **typedef** is
  
  ```c
  typedef char STRING[81];
  ```

  ```c
  STRING x; /* same as char x[81];*/
  ```
### PRECEDENCE CHART OF C OPERATORS

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>NAME</th>
<th>GROUPING</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>function call</td>
<td>left-to-right</td>
</tr>
<tr>
<td>[]</td>
<td>array element</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>structure, union member</td>
<td></td>
</tr>
<tr>
<td>-&gt;</td>
<td>structure, union member with pointer</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical not</td>
<td>right-to-left</td>
</tr>
<tr>
<td>~</td>
<td>one's complement</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>minus</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>indirection</td>
<td></td>
</tr>
<tr>
<td>(type)</td>
<td>type cast</td>
<td></td>
</tr>
<tr>
<td>sizeof</td>
<td>size in bytes</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>left-to-right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>remainder</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>left-to-right</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>left-to-right</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>left-to-right</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise and</td>
<td>left-to-right</td>
</tr>
<tr>
<td>^</td>
<td>bitwise exclusive or</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise or</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical and</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=</td>
</tr>
<tr>
<td>?::</td>
<td>conditional</td>
<td>right-to-left</td>
</tr>
<tr>
<td>=</td>
<td>assignment operator</td>
<td>right-to-left</td>
</tr>
<tr>
<td>+=</td>
<td>assignment replace add</td>
<td></td>
</tr>
<tr>
<td>-=</td>
<td>assignment replace subtract</td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td>assignment replace multiply</td>
<td></td>
</tr>
<tr>
<td>/=</td>
<td>assignment replace divide</td>
<td></td>
</tr>
<tr>
<td>%=</td>
<td>assignment replace remainder</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>assignment replace shift left</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>assignment replace shift right</td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td>assignment replace and</td>
<td></td>
</tr>
<tr>
<td>^=</td>
<td>assignment replace exclusive or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>assignment replace or</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>
STRUCTURES

- A structure (record in Pascal) is a composite object that consists of components of any type except functions. Structures have the form:
  ```
  struct [tag]
  {
    list of declarations
  }
  ```
  If the tag is included then specific structure variables can be defined further on in the code.

- An example of a structure is:
  ```
  struct employee
  {
    char     name[20];
    int      age;
    float    salary;
  }
  ```
  The above defines the structure tag `employee`. No structure variables have been defined by the above. To define structure variables we use the tag as in:
  ```
  struct employee joe, bill, jane;
  /* 3 variables have been defined */
  ```
  The above defines structure variables `joe`, `bill`, and `jane`.

- We could have defined the structure variables `joe`, `bill`, and `jane` along with the structure as in:
  ```
  struct employee
  {
    char     name[20];
    int      age;
    float    salary;
  } joe, bill, jane;
  ```
  The tag is optional in the above. It must be used if we wish to define more structure variables later on in the program.
We can also use `typedef` to associate a name with a structure as in:

```c
typedef struct employee
    /* tag employee is optional */
    /* required if pointers are used */
{
    char name[20];
    int age;
    float salary;
} WORKER;
```

We can then define structure variables by:

```
WORKER joe, bill, jane;
```

(This form is the most elegant and generally preferred.)

Finally, we can first define the structure:

```c
struct employee
{
    char name[20];
    int age;
    float salary;
};
```

and then use `typedef` to associate a name with the structure:

```c
typedef struct employee WORKER;
```

and then define the structure variables:

```c
WORKER joe, bill, jane;
```
ACCESSING STRUCTURE COMPONENTS

• If x is a structure variable, then specific components are accessed by putting a dot after the x and continuing with the component name. e.g.,

```c
struct date
{
    int year,
    month,
    day,
    hour,
    minute,
    second;
} x;
```

The components of the structure variable x can be accessed by:

```
x.year, x.month, x.day, ..., x.second
```

• We can now define y to be of type pointer to structure by:

```c
struct date *y;
```

• We can assign y a value in the same way any other kind of pointer is handled:

```c
y = &x;
```

• If y is a pointer to the above structure, then the first component can be accessed as

```c
(*y).year
```

(The parentheses are needed because of the hierarchy of operators)

or by the shorter notation

```c
y->year
```
struct1.c

/* STRUCT1.C: Illustrates how complex (imaginary) arithmetic can be done by defining a structure to treat complex numbers as a single entity. */
#include <stdio.h>

typedef struct
{
    double real,
    imag;
} COMPLEX;

void product(COMPLEX *u, COMPLEX *v, COMPLEX *w);

main()
{
    COMPLEX z1, z2, z3;
    z1.real = -2.0;
    z1.imag =  3.0;
    /* z1 = -2 + 3i */
    z2.real =  4.0;
    z2.imag = -1.0;
    /* z2 =  4 - i */

    product( &z1, &z2, &z3 );
    /* Note that we are passing pointers to the structures rather than the structures themselves */

    printf("real = %f, imag = %f\n", z3.real, z3.imag);
    return 0;
}

void product(COMPLEX *u, COMPLEX *v, COMPLEX *w)
{
    w->real = (u->real)*(v->real) - (u->imag)*(v->imag);
    w->imag = (u->imag)*(v->real) + (u->real)*(v->imag);
}
struct2.c

/* STRUCT2.C: This program on complex arithmetic differs from the previous one in that the product function takes structures (not pointers to structures) as arguments. This program will not work on ancient C compilers. */
#include <stdio.h>

typedef struct
{
    double real,
    imag;
} COMPLEX;

COMPLEX product(COMPLEX, COMPLEX);

main()
{
    COMPLEX z1, z2, z3;

    z1.real = -2.0;
    z1.imag = 3.0;
    /* z1 = -2 + 3i */
    z2.real = 4.0;
    z2.imag = -1.0;
    /* z2 = 4 - i */

    z3 = product(z1, z2);

    printf("real = %f, imag = %f\n", z3.real, z3.imag);
    return 0;
}

COMPLEX product (COMPLEX u, COMPLEX v)
{
    COMPLEX w;

    w.real = (u.real)*(v.real) - (u.imag)*(v.imag);
    w.imag = (u.imag)*(v.real) + (u.real)*(v.imag);

    return(w);
}
triangle.c

/* TRIANGLE.C: Program illustrates the use of structures to do geometry of triangles. */
#include <stdio.h>
#include <math.h>

typedef struct
{
    double side1,
    side2,
    side3;
} TRIANGLE;

double perimeter(TRIANGLE), area(TRIANGLE);

main()
{
    static TRIANGLE t = {1.0, 1.0, 0.5};

    printf("perimeter = %f\n", perimeter(t) );
    printf("area = %f\n", area(t) );

    return 0;
}

double perimeter(TRIANGLE x)
{
    return( x.side1 + x.side2 + x.side3 );
}

double area(TRIANGLE x)
/* use hero's formula for area of a triangle */
{
    double s, a;

    s = perimeter(x)/2.;    /* s is semi-perimeter */
    a = sqrt(s*(s - x.side1)*(s - x.side2)*(s - x.side3));

    return(a);
}
INITIALIZING STRUCTURES

- Initialization of structures is similar to initialization of arrays; the elements are simply listed inside a pair of braces, with each element separated by a comma. As with arrays, automatic structure variables cannot be initialized. It is valid to initialize external and static structure variables. A structure variable is made static by placing the keyword static directly before the definition. An example of a structure initialization is:

```c
struct complex
{
    double real, imag;
};
static struct complex z = {-2.0, 4.7};
```

- In TurboC, to increase stack size (for structure passing), declare a global C variable in your program:

```c
unsigned _stklen = 5000;
/* set up 5000 byte stack */
```

the default is 4096 bytes.
Lab 1 for Part 4

1. Write a program, rectangl.c, similar to triangle.c, to define a structure RECTANGL and a set of processing functions, perimeter and area. The rectangle can be defined simply by the lengths of two perpendicular edges. The area is just the product of these two lengths, and the perimeter is twice the sum of the two lengths.

2. If a UNIX system is available, try lint on cast.c (% lint cast.c -lm). Be sure you understand what lint says.

3. For any available systems (including UNIX), try compiling cast.c with every error option enabled. Be sure you understand all the compiler warnings. (In Turbo C compile by using Alt-C, Return so that you see the error messages before execution.)

4. Rewrite struct1.c and struct2.c without the typedef. (But you should still use structures.)

5. Rewrite triangle.c so that the triangle information contained in the structure consists of two sides and the angle subtended by these two sides. Write the function that will compute the area of the triangle (area = 0.5*a*b*sin(C)). Write the area function in two ways:
   a. The structure is passed to the area function, and
   b. A pointer to the structure is passed to the area function.
COBOL vs. C STRUCTURES

COBOL:

01 EMPLOYEE.
  02 JOB-TITLE PICTURE X(7).
  02 EMP-NO PICTURE X(9).
  02 PAY-RATE PICTURE 9(2)V9(2).
  02 NAME.
    03 L-NAME PICTURE X(12).
    03 F-NAME PICTURE X(8).
    03 INIT PICTURE X.
  02 OFFICE-ADDR.
    03 BLDG PICTURE X(3).
    03 ROOM PICTURE 9(3).
  02 PROJECT-CODES OCCURS 5 TIMES PICTURE 9(2).
  02 TELEPHONE-NO OCCURS 2 TIMES.
    03 AREA-CODE PICTURE 9(3).
    03 LOCAL PICTURE 9(7).

C:

struct
{
  char  job_title[8],
  char  emp_no[10],
  float pay_rate,
  struct
  {
    char   l_name[13];
    char   f_name[9];
    char   init;
  } name,
  struct
  {
    char   bldg[4],
    int     room;
  } office_addr,
  int  project_codes[5],
  struct
  {
    int     area_code,
    long   int  local;
  } telephone_no[2];
} employee;
Suppose we wish to create a structure for the data processing record shown below:

<table>
<thead>
<tr>
<th>Job Title</th>
<th>Emp No</th>
<th>Pay Rate</th>
<th>Name</th>
<th>Office Addr</th>
<th>Project Codes</th>
<th>Telephone No</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Last</td>
<td>First</td>
<td>M I</td>
<td>Bldg</td>
</tr>
<tr>
<td>Analyst</td>
<td>123456789</td>
<td>15.93</td>
<td>Mud</td>
<td>Joe</td>
<td>N</td>
<td>CSC</td>
</tr>
</tbody>
</table>

Note that the record has alphabetic, integer, and real data types (job title, employee number, pay rate). It has embedded structures (name, office address). It has an array of integers (project codes), and it has an array of embedded structures (telephone numbers).

The code on the facing page shows how this would be coded in COBOL and shows the closest parallel structure in C. The C code is the most straightforward, but not the most elegant. More elegant code would involve using `typedef` rather than directly embedding the structures.

C code is much more flexible, in that there are a great many ways in which this record can be coded.
dynamic.c

/* DYNAMIC.C: A very simple demo of dynamic memory allocation. */
#include <stdio.h>
#include <stdlib.h>

main()
{
    double *p;

    /* dynamically allocate space for 4 consecutive doubles */
p = (double *) calloc(4, sizeof(double));

    if ( p == NULL )
        exit(1); /* exit if space not available */

    /* the pointer p points to the first of 4 consecutive doubles */
    *p = 1.0;       /* or p[0] = 1.0 */
    *(p+1) = 2.0;
    *(p+2) = 3.0;
    *(p+3) = 4.0;       /* or p[3] = 4.0 */


    free( (char *)p );
    return 0;
}
DYNAMIC MEMORY ALLOCATION

- The following C functions are available for allocating and releasing dynamic memory. For each of these functions, \( n \) and \( size \) represent unsigned integers, and \( pointer \) represents a character pointer.

```c
char *malloc(size)
Allocates contiguous space of \( size \) bytes, returning a pointer to the beginning of the allocated block if successful, and the null pointer otherwise.
```

```c
char *calloc(n, size)
Allocates contiguous space for \( n \) items of data, where each item is \( size \) bytes in length. The allocated space is initially set to all zeroes. On success, a pointer to the allocated space is returned; on failure, the null pointer is returned.
(Both malloc and calloc use a lower-level function sbreak which gets space plus a header.)
```

```c
void free(pointer)
Releases a block of memory pointed to by \( pointer \) that was previously allocated by a calloc or malloc call. It is an error to free a block that was not previously allocated by a call to calloc or malloc.
```

```c
char *realloc(pointer, size)
Alters the length of the continuous space pointed to by \( pointer \) to be of length \( size \). The contents will be unchanged up to the minimum of the old and new sizes.
```

- Typically, the pointer returned by a call to calloc or malloc must be cast into a form appropriate for the application.
linked.c

/* LINKED.C: Example of a linked list. */
#include <stdio.h>
#include <stdlib.h>
typedef struct x
{
    int data; /* data component of NODE */
    struct x *next; /* pointer to next NODE */
} NODE;

NODE *insert (NODE *, int);
void delet(NODE *);
void list(NODE *);

NODE *insert(NODE *p, int d)
/* Insert a new node, with data d, immediately after the node pointed to by p. If successful, return a pointer to the new node, else return NULL. */
{
    NODE *newnode;

    newnode = (NODE *) calloc(1, sizeof(NODE));
    if (newnode == NULL)
        return(NULL);
    newnode->data = d;
    newnode->next = p->next;
    p->next = newnode;
    return(newnode);
}

void delet(NODE *p)
/* delete node immediately after node pointed to by p
   -node MUST have been allocated by the insert function */
{
    NODE *q;

    q = p->next;
    if( q == NULL )
        return;
    p->next = q->next;
    free((char *)q);
}
void list(NODE *p)     /* list data of linked nodes starting with node pointed to by p */
{
    while ( p != NULL ) /* or while (p) */
    {
        printf("%d\n",p->data);
        p = p->next;
    }
    printf("\n");
}

main()     /* purpose of main is to exercise (test) insert and delete routines */
{
    NODE first, *ptr;
    first.data = 0;  first.next = NULL;
    if( (ptr = insert(&first,10)) == NULL )
        exit(1);
    if( (ptr = insert(ptr,20)) == NULL )
        exit(1);
    list(&first);
    delet(&first);
    list(&first);
    return 0;
}
union1.c

/* UNION1.C: Illustrates the use of a union */
#include <stdio.h>

union sample
{
    int   i;
    float f;
    double d;
    char  c;
} x;

main()
{
    x.i = 2;
    printf("%d\n", x.i);

    x.f = -3.8;
    printf("%f\n", x.f);

    /* the next two lines are a misuse of the union */
    x.c = 'a';
    printf("%f\n", x.f);
    /* x.f is not currently active */

    return 0;
}
UNIONS

- Another powerful aggregate in the C programming language is the union. This construct is used mainly in applications where it is necessary to store different types of data into the same storage area. For example, if we wanted to define a single variable called x, which could be used to store a single character, a floating point number, or an integer, then we would define a union, called mixed, as follows:

  ```c
  union mixed
  {
    char   c;
    float  f;
    int    i;
  } x;
  ```

- The declaration for a union is identical to that of a structure, except the keyword union is used where the keyword struct is otherwise specified. The real difference between structures and unions has to do with the way memory is allocated. Declaring a variable to be of type mixed, as above, does not define x to contain three distinct members, but rather, a single member that is called either c, f, or i. We could store a real number into the variable x by the statement

  ```c
  x.f = -2.7;
  ```

  Since the float, char, and int members of x all co-exist in the same place in memory, only one value can be stored into x at a time. It is your responsibility to ensure that the value used in a union is consistent with the way it was last stored in the union.

- The union construct in C is similar to the equivalence statement in FORTRAN, the redefines clause in COBOL, or the variant record in PASCAL.
union2.c

/* UNION2.C: Simple application of a union */
#include <stdio.h>
#include <string.h>

typedef struct
{
    char name[25];
    int age;
    char job_status; /* P = permanent, C = contract */
union
    {
        char employee_no[10]; /* employee number */
        int svc_time; /* months of service */
    } job_info;
} EMPLOYEE;

void display (EMPLOYEE *);

main()
{
    EMPLOYEE joe, irene;

    (void) strcpy(joe.name, "JOSEPH");
    joe.age = 21;
    joe.job_status = 'P';
    (void) strcpy(joe.job_info.employee_no, "654321");

    (void) strcpy(irene.name, "IRENE");
    irene.age = 16;
    irene.job_status = 'C';
    irene.job_info.svc_time = 1;

display(&joe);
display(&irene);

    return 0;
}
void display(EMPLOYEE *x)
   /* display information in employee structure */
{
   printf("%s\n",x->name);
   printf("age = %2d years\n",x->age);

   if (x->job_status == 'P')
   {
      printf("Permanent employee\n");
      printf("Employee number = %s\n",x->job_info.employee_no);
   }
   else if (x->job_status == 'C')
   {
      printf("Contract employee\n");
      printf("Service time = %d months\n",x->job_info.svc_time);
   }
   printf("\n");
}
union3.c

/* UNION3.C: Simple union using a header file */
#include <stdio.h>
#include <string.h>
#include "union3.h"

main()
{
   EMPLOYEE joe, irene;

   (void) strcpy(joe.name, "JOSEPH");
   joe.age = 21;
   joe.job_status = 'P';
   (void) strcpy(joe.job_info.employee_no, "654321");
   (void) strcpy(irene.name, "IRENE");
   irene.age = 16;
   irene.job_status = 'C';
   irene.job_info.svc_time = 1;

   display(&joe);
   display(&irene);
   return 0;
}

void display(EMPLOYEE *x)
/* display information in employee structure */
{
   printf("%s\n", x->name);
   printf("age = %2d years\n", x->age);
   if (x->job_status == 'P')
   {
      printf("Permanent employee\n");
      printf("Employee number = %s\n", 
             x->job_info.employee_no);
   }
   else if (x->job_status == 'C')
   {
      printf("Contract employee\n");
      printf("Service time = %d months\n", 
             x->job_info.svc_time);
   }
   printf("\n");
}
union3.h

/* Header file for employee union - union3.h */

/* First, protect against multiple definitions */

#ifndef union3
#define union3

/* Then, define the structures */

typedef struct
{
    char name[25];
    int age;
    char job_status; /* P = permanent, C = contract */
    union
    {
        char employee_no[10]; /* employee number */
        int svc_time; /* months of service */
    } job_info;
} EMPLOYEE;

/* Then declare the functions that use them */

void display (EMPLOYEE *);

#endif
bits.c

/* BITS.C: This program splits a "word" apart into its
bit components */
#include <stdio.h>

#define PATTERN 075462154

main()
{
    struct packed
    {
        unsigned int  f1:1;
        unsigned int  f2:2;
        unsigned int  f3:6;
        unsigned int  f4:4;
        unsigned int  f5:12;
        unsigned int  f6:11;
        unsigned int  f7:12;
    };

    union overlay
    {
        struct packed mydata;
        unsigned int bitpattern;
    };

    union overlay x;

    /* See the effect of breaking apart a stored constant */

    x.bitpattern = PATTERN;

    printf ("nDecoded constant n");
    printf ("%o\n", x.mydata.f1);
    printf ("%o\n", x.mydata.f2);
    printf ("%o\n", x.mydata.f3);
    printf ("%o\n", x.mydata.f4);
    printf ("%o\n", x.mydata.f5);
    printf ("%o\n", x.mydata.f6);
    printf ("%o\n", x.mydata.f7);
/* Now see the effect of storing data into the components */

x.mydata.f1 = 7;
x.mydata.f2 = 4;
x.mydata.f3 = 11;
x.mydata.f4 = 023456;
x.mydata.f5 = 01234567;
x.mydata.f6 = 7;
x.mydata.f7 = -3;

printf("\nIndividual constants\n");
printf("%o\n", x.mydata.f1);
printf("%o\n", x.mydata.f2);
printf("%o\n", x.mydata.f3);
printf("%o\n", x.mydata.f4);
printf("%o\n", x.mydata.f5);
printf("%o\n", x.mydata.f6);
printf("%o\n", x.mydata.f7);
return 0;
}
A Generalized Data Type

The following code can be used to generate a "generalized" data type:

```c
typedef struct gen
{
    int type;
    /* 0 = char
    1 = int
    2 = long
    3 = float
    4 = double */
union
{
    char xc;
    int xi;
    long xl;
    float xf;
    double xd;
} data;
} PARAM;
```

Now parameters can be passed as type `PARAM`, and calling routines can use any data type. The called routine can determine the data type passed and correctly interpret it.

Several commercial packages use this type of parameter passing mechanism.
Lab 2 for Part 4.

1. Modify `dynamic.c` so that we dynamically allocate an array of 5 integers. Assign values to this array and determine the sum of the members of this array.

2. Add two new routines to the file `linked.c`, one to search for a node by contents, the other to alter the contents of a node. The search routine should begin its search at the node pointed to by p, checking for matching data, d. If a match is found, the routine should return the pointer to the node; if no match is found, it should return NULL. The alter routine should replace the data in the node pointed to by p with data d. Now write a section of code in program main to search for all occurrences of the value 10 and alter them to 11.

3. Expand `union1.c` so that data is stored in one of the four interpretations of the union and printed in the other three. Repeat this for each of the four interpretations. Since you are deliberately misusing the union to see the effects, you should assume that your program (or entire system, if you are on a PC) could crash at any time. Take the appropriate precautions, including saving all files before starting this exercise. After completing the exercise, logout and log back in, or re-boot your PC, as appropriate.

4. Modify `union2.c` so that the name of the contract house, instead of the service time, is the relevant information in the union portion of the structure.

5. Determine the output of `bits.c` by tracing each bit group from assignment to printing. Modify the program as necessary to determine performance under cases that the current program may leave ambiguous. If more than one C compiler is available, try the program on several compilers and/or on different machines, noting any differences.
STANDARD I/O

- The following functions are used for input from stdin.

  ```c
  int getchar() returns next character from input
  char *gets(s) read a string into s from input and return pointer to the string
  scanf(format [, pointer] ...) read using a format control string from input
  ```

- The following functions are used for output to stdout.

  ```c
  putchar(c) write a character to output
  puts(s) write a string to output and append a newline character
  printf(format [, arg] ...) write using a format control string to output
  ```

- The header file `stdio.h` must be included at the front of any program that uses these functions.
FILE I/O

• Reference: Kochan - Pages 355 - 366.

• The following describes some of the commonly used I/O functions from the C library. The header file `stdio.h` must be included at the front of any program that uses these functions. Included in this file are definitions for `EOF`, NULL, stdin, stdout, stderr and the data type `FILE`.

• In the descriptions that follow,

  `file_name` is a pointer to a null-terminated string that contains the name of the file to be processed

  `access_mode` is a pointer to a null-terminated string that indicates the access mode for the file

  `format_string` is a pointer to a null-terminated string that provides the data conversion format

  `file_pointer` is a pointer to a `FILE` structure

  `n` is a positive integer

  `c` is a character
OPENING A FILE

• To open a file:
  ```c
  FILE *fopen(file_name, access_mode)
  ```
  Opens the specified file in the indicated access mode.

• Valid modes are
  - `r` for reading
  - `w` for writing
  - `a` for appending

• Any of the above mode strings may be appended with a plus sign `+` to indicate opening for update (both reading and writing). In this mode, both reads and writes may be performed on the file; in order to switch between reading and writing, however, an `fseek` or `rewind` must be executed.
  - if a file is opened for reading with a plus, then the file must already exist
  - if a file is opened for writing with a plus, the file will be created anew
  - opening for appending with a plus will permit reads to take place from any position in the file, but all write operations will occur at the end of the file.

• If an error occurs in attempting to open a file
  - `fopen` returns `NULL`
  - an external variable `errno` contains the code as to what actually happened
  - symbolic names for possible errors are in `<errno.h>`
READING FROM A FILE

• To read from an open file:

Single character at a time:

```c
int fgetc(file_pointer)
int getc(file_pointer)
```
Return the next character or EOF. Both functions work the same, `fgetc` is a function, `getc` is implemented as a macro.

Read characters into a buffer:

```c
char *fgets(buffer, n, file_pointer)
```
Reads characters from the indicated file, until either n-1 characters are read or until a newline character is read, whichever occurs first. Characters that are read are stored into the character array pointed to by `buffer`. If a newline character is read, then it will be stored in the array. If an EOF is reached and no characters are read, then the value `NULL` is returned, otherwise `buffer` is returned.

```c
int fread(buffer, size, n, file_pointer)
```
Reads n items of data from the identified file into `buffer`. Each item of data is `size` bytes in length. The function returns the number of characters that are successfully read.

Formatted read:

```c
int fscanf(file_pointer, format, arg1, ..., argn)
```
The function returns the number of items successfully read or EOF if the end of file is reached before the first item is read.
WRITING TO A FILE

- To write to an open file:

  Single character at a time:
  
  ```c
  int fputc(c, file_pointer)
  int putc(c, file_pointer)
  ```

  Writes the character `c` into the file identified by the file pointer, returning `c` if the write is successful, and the value `EOF` otherwise. The function `fputc` is a function, `putc` is implemented as a macro.

  Write characters from a buffer:
  
  ```c
  int fputs(buffer, file_pointer)
  ```

  Writes the characters in the array pointed to by `buffer` to the indicated file until the terminating null character in `buffer` is reached (which is not written). A newline character is not automatically written to the file by this function.

  ```c
  int fwrite(buffer, size, n, file_pointer)
  ```

  Writes `n` items of data from `buffer` into the identified file. Each item of data is `size` bytes in length. Returns the number of items successfully written.

  Formatted write:
  
  ```c
  int fprintf(file_pointer, format, arg1, ..., argn)
  ```

  The number of characters written is returned if the call is successful; otherwise a negative value is returned.
FILE FUNCTIONS

• Error handling:
  Once a file has been opened, errors can be obtained by a call to:
    ferror(file_pointer)
  A non-zero value indicates an error. Error indications last until the stream is closed unless cleared by a call to
    clearerr(file_pointer)

• Other file functions:
  int fclose(file_pointer)
  Closes the file identified by file_pointer, and returns zero if the close is successful, EOF if an error occurs.
  long offset;
  int mode;
  int fseek(file_pointer, offset, mode)
  Positions the indicated file to a point that is offset bytes from the beginning of the file, from the current position in the file, or from the end of the file, depending on whether the value of mode is 0, 1, or 2, respectively. If the fseek call is successful, then zero is returned, otherwise a non-zero value is returned.
  long ftell(file_pointer)
  Returns the relative offset in bytes of the current position, or -1 on error.
fileio.c

/* FILEIO.C: Program to copy one file to another. */
#include <stdio.h>
#include <stdlib.h>

main()
{
    char in_name[25], out_name[25];
    FILE *in_file, *out_file;
    int c;

    printf("Source file??\n");
    scanf("%24s", in_name);

    printf("Destination file??\n");
    scanf("%24s", out_name);

    if ( (in_file = fopen(in_name, "r")) == NULL )
    {
        fprintf(stderr, "Can't open source file.\n");
        exit(1);
    }

    if ( (out_file = fopen(out_name, "w")) == NULL )
    {
        fprintf(stderr,
                "Can't open destination file.\n");
        exit(1);
    }

    /* this is the main loop */
    while ( (c = fgetc(in_file)) != EOF )
    {
        fputc( c, out_file );

        fclose(in_file);
        fclose(out_file);

        printf("Finished!!\n");
        return 0;
    }
}
filerror.c

/* FILERROR.C: Program to copy one file to another. */
/* Error processing is included. */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
extern int errno;

int main()
{
    char in_name[25], out_name[25];
    FILE *in_file, *out_file;
    int c;

    printf("Source file??\n");
    scanf("%24s", in_name);
    printf("Destination file??\n");
    scanf("%24s", out_name);
    if ( (in_file = fopen(in_name, "r")) == NULL )
    {
        if (errno == ENOENT)
            fprintf(stderr,
                "No such file or directory!!\n");
        else
            fprintf(stderr,
                "Can't open source file!!\n");
            exit(1);
    }
    if ( (out_file = fopen(out_name, "w")) == NULL )
    {
        perror("filerror.c");
        exit(1);
    }

    while ( (c = fgetc(in_file)) != EOF ) /* main loop */
    { /* main loop */
        fputc( c, out_file );
        fclose(in_file);
        fclose(out_file);
        printf("Finished!!\n");
        return 0;
    }
}
reverse.c

/* REVERSE.C: Read a file whose name is entered on the
command line and print the file in reverse order to
<stdout>. This program is not portable, and works
only on selected computers and file organizations.
It will generally work on files written by
a C program (such as fileio.c) */
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    FILE *ptr;
    long n;
    if ( argc != 2 || (ptr=fopen(argv[1],"r")) == NULL )
        exit(1);

    fseek(ptr, -1L, 2);
    /* position at last byte of file */
    n = ftell(ptr); /* n + 1 = file size */
    while ( n-- >= 0 )
    {
        putchar( fgetc(ptr) );
        fseek(ptr,n,0);
    }

    printf("\n");
    return 0;
}
UNDERSTANDING C DATA TYPE DECLARATIONS

• Reference: K/R - Page 195, Appendix A.

• In the following T is a C type specifier. The possible choices that T may have are: char, int, float, double, structure, union. In addition, the adjectives long, short, unsigned may be used as is appropriate.

• A simple declaration is of the form
  
  T identifier

For example,

  int x  (declares x to be of type int)

• A simple or complex declaration can be viewed in the form

  T D

where D is an identifier or something more complex as in the following.

• If T D is a declaration, then

  T *D identifier is ... pointer to T.
  T D() identifier is ... function returning T.
  T D[] identifier is ... array of T.

• We can start with a simple declaration, use the above several times to get a compound declaration. The only compositions that are ILLEGAL are

  T ( D() )[] function returning array of ..
  T ( D[] )() array of function(s) returning ..
  T ( D() )() function returning function returning ..

• Some easy examples of compound declarations are:

  int (*x)() x is a pointer to a function returning an int.
  int x is a function returning a pointer to an int.
  *( x() ) (Because of precedence rules, int
           *x() is the same.)
  int (*x)[] x is a pointer to an array of integers.
  int x is an array of pointers to integers.
  *( x[] ) (Because of precedence rules, int
           *x[] is the same.)
  int x is an array of pointers to functions
  (*x[])() returning int.
enum.c

/* ENUM.C: Examine the enumerated data type in C */
#include <stdio.h>

main()
{
    enum Gender { female, male, unknown = 4, couple }; /* female=0, male=1, unknown=4, couple=5 */
typedef enum { yes, no } Customer; /* adding "unknown" as a third possibility would be an error */
    enum Gender mary = female, mike = male;
    enum Gender pat, ted_and_alice;
    Customer joe = yes;

    pat = unknown;
    ted_and_alice = couple;

    printf ("Mary's gender = %d\n", mary);
    printf ("Mike's gender = %d\n", mike);
    printf ("Pat's gender = %d\n", pat);
    printf ("Ted&Alice's gender = %d\n", ted_and_alice);

    switch (joe)
    {
        case yes:
            printf ("Joe is a customer\n");
            break;
        case no:
            printf ("Joe is not a customer\n");
            break;
        default:
            printf ("Joe has an error\n");
            break;
    }
    return 0;
}
## SUMMARY OF TYPE CONSTRUCTION

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>Simple/Aggregate</th>
<th>Terminating/Intermediate</th>
<th>Resolved to</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>S</td>
<td>T</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>short</td>
<td>S</td>
<td>T</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>int</td>
<td>S</td>
<td>T</td>
<td>int</td>
<td>no</td>
</tr>
<tr>
<td>long</td>
<td>S</td>
<td>T</td>
<td>long</td>
<td>no</td>
</tr>
<tr>
<td>enum</td>
<td>S</td>
<td>T</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>float</td>
<td>S</td>
<td>T</td>
<td>double</td>
<td>yes</td>
</tr>
<tr>
<td>double</td>
<td>S</td>
<td>T</td>
<td>double</td>
<td>no</td>
</tr>
<tr>
<td>void</td>
<td>S</td>
<td>T</td>
<td>(none)</td>
<td>-</td>
</tr>
<tr>
<td>pointer</td>
<td>S</td>
<td>I</td>
<td>pointer</td>
<td>no</td>
</tr>
<tr>
<td>function</td>
<td>S</td>
<td>I</td>
<td>pointer</td>
<td>yes</td>
</tr>
<tr>
<td>array</td>
<td>A</td>
<td>I</td>
<td>pointer</td>
<td>yes</td>
</tr>
<tr>
<td>structure</td>
<td>A</td>
<td>T</td>
<td>error</td>
<td>-</td>
</tr>
<tr>
<td>union</td>
<td>A</td>
<td>T</td>
<td>error</td>
<td>-</td>
</tr>
</tbody>
</table>
/* DEMO.C: Example of array of pointers to functions 
   returning integers. */
#include <stdio.h>
#define P(x) printf("%d\n",x)

int f0 (int);
int f1 (int);
int f2 (int);

main()
{
    static int ( *a[] )(int) = { f0, f1, f2 }; /* array size = 3 */
    int arg = 10; /* test argument */
    P( ( *a[0] )(arg) ); /* print each function */
    P( ( *a[1] )(arg) ); /* applied to arg */
    P( ( *a[2] )(arg) );
    printf("\n\n");
    P( ( **a )(arg) ); /* do it another way */
    P( ( **(a+1) )(arg) );
    P( ( **(a+2) )(arg) );
    return 0;
}

int f0(int x)
{
    return( x + 0 );
}

int f1(int x)
{
    return( x + 1 );
}

int f2(int x)
{
    return( x + 2 );
}
array2.c

/* ARRAY2.C: Program shows relation between pointers and two dimensional arrays. */
#include <stdio.h>

#define P(y) printf("%d\n",y)

main()
{
    static int x[3][4] = { {1, 2, 3, 4},
                          {5, 6, 7, 8},
                          {9, 10, 11, 12} };
    /* x is an array of size 3 whose elements are arrays of size 4 */

    P( **x );
P( *(x[1] + 2) );
P( *(x + 2) );
P( **(x + 2) );
P( x[2][2] );
P( *( *(x + 1) + 2 ) );
    return 0;
}
/* ARRAY3.C: Sizeof and two dimensional arrays. */
#include <stdio.h>

main()
{
    static int x[2][4] = { {1, 2, 3, 4},
                           {5, 6, 7, 8} };

    printf("size of int = %d\n", sizeof(int));
    printf("size of row = %d\n", sizeof(x[0]));
    printf("size of row = %d\n", sizeof(*x));
    printf("size of array = %d\n", sizeof(x));
    return 0;
}
Lab 3 for Part 4.

1. Experiment with various formatting options using the `printf` function, including the printing of preceding zeroes.
2. Write a C program that determines the size of a file by reading and counting every character in the file.
3. Write a C function that implements one of the string functions. Test the function.
4. Write a C function that implements one of the character functions. Test the function.
5. Write a C function that compares two files, finding either the line number and character position of the first difference between the two files, or verifying that the two files are character-for-character identical. Print the line number and character position of the first difference between the two files.