FUNCTIONS – THE DETAILS

- C allows a block of code to be separated from the rest of the program and named.
- These blocks of code or modules are called functions.
- Functions can be passed information thru a parameter list. Any number of parameters can be passed to a function.
- Functions can pass back a result thru a return value. At most one return value can be produced.
- All the C data types are candidates for parameter types and return types.
- Ideally a function can be treated as a black-box. If you know what to pass it and what it will return; you don't need to, or sometimes want to, know how it works.
- C has a special keyword, *void*, that is used to explicitly state that there are no parameters or no return type.
- Using a function takes place in three steps:
    - Defining the function

      The definition is the C code that completely describes the function, what it does, what formal parameters it expects, and what it's return value and type will be.
    - Calling the function

      When the function is needed to do its work, it is "called" by its name and supplied actual parameters for the formal parameters it requires. Its return value is used if provided and needed.
    - Prototyping the function

      A prototype provides the communication information for the function, the parameter types and return value, to the compiler. This allows the compiler to more closely scrutinize your code. (This is a very, very good thing.) A prototype looks like the first line of the function definition, it identifies the parameter types and the return type of the function. A prototype should be placed within the source code at a point before the call is made. Often prototypes are placed near the top of the source code file. More often, the prototypes are placed into a .h file and *#include* is used to include them in the source code file.

POINTERS

- A pointer in C is a data type that can store the address of some other storage location.
- Pointers are used when a variable's location is of interest and not just it's value.
- A pointer is declared by using a data type followed by an asterisk, *.
- To produce the address of a variable, apply the address-of operator, & to a variable.
- Since the contents of a pointer variable are an address you need to dereference the pointer to access the value it references. That will be the value at the address the pointer contains, or the value the pointer references.

Ex:

```
/*     FILE: pointer.c     */

/* A pointer variable. */
#include <stdio.h>

int main( )
{
  int* ptr;
  int i;

  i = 7;

  ptr = &i;     /* ptr now knows where i is. */

  printf("i = %d and is at address %p\n", i, &i);

  printf("i = %d and is at address %p\n", *ptr, ptr);

  return 0;
}

/*    OUTPUT: pointer.c

          i = 7 and is at address 0022FF68
          i = 7 and is at address 0022FF68

*/
```

TEXT FILE I/O

- Basic text file I/O is only slightly more difficult than the I/O done to date.
- Every I/O function seen so far has a sister function that will read/write to a file on disk.
- The programmers connection to a file on disk is a file name. The C connection to a file on disk is a file pointer, `FILE *`. The first step in doing file I/O is to translate a filename into a C file pointer using `fopen( )`.
- The file pointer is then passed to the file I/O function we are using so that C can access the appropriate file.
- Finally the connection to the file is severed by calling `fclose( )` with the file pointer as a parameter.

Ex:

```
/*     FILE: FileIO.c     */

/* Basic output using printf( ) */
#include <stdio.h>

int main( )
{
  int x = 7;
  double y =7.25;

  printf("This data will be written to the screen.\n");
  printf("x = %d, y = %f\n", x, y);

  return 0;
}


/*    OUTPUT: FileIO.c

          This data will be written to the screen.
          x = 7, y = 7.250000

*/
```


BINARY FILE I/O

- Binary file I/O writes data from memory to disk in the same format as it is stored in memory.
- Generally is is not going to be human-readable but it should take up less space and can be done faster since it does not need to be translated into text.
- File pointers are used in the same manner as they are in text I/O.

Ex:

```
/*     FILE: FileIO_5.c     */

/* Binary I/O using fwrite( ) and fread( ) */
#include <stdio.h>

int main( )
{
  FILE *fptr;
  int i, x;

  x = 0;
  i = 7;

  printf("i = %d   x = %d\n", i, x);

  fptr = fopen("tmp.dat","w");

  if(fptr != NULL){
    fwrite(&i, 4, 1, fptr);
    fclose(fptr);
  }
  else
    printf("Unable to open file for write.\n");

  fptr = fopen("tmp.dat","r");

  if(fptr != NULL){
    fread(&x, sizeof(int), 1, fptr);
    fclose(fptr);
  }
  else
    printf("Unable to open file for read.\n");

  printf("i = %d   x = %d\n", i, x);

  return 0;
}


/*    OUTPUT: FileIO_5.c

          i = 7   x = 0
          i = 7   x = 7

*/
```

STRINGS

- The C definition of a string is: a set of characters terminated by a null character.
- A set of characters written inside of double quotes indicates to the compiler that it is a string.
- Placement of the null character gets handled by C itself, when C can identify that it is working with strings.
- A programmer can create and manipulate a string as a set of char locations. This set of locations can be created as an array. The programmer must then be sure that the set is used properly so that the terminating null gets placed at the end of the characters so that it represents a legitimate string.

ARRAYS

- C allows easy creation and access to sets of storage locations with arrays.
- An array is set of storage locations all referred to by the same name. Each individual location is uniquely identified by the array name and an index value, or offset, into the array.
- C arrays are indexed beginning with the value 0 for the index of the first location and ending with the size-1 for the index of the last location.
- Since the only difference between successive locations in an array is the index value, the computer can be used to generate the index values. This allows an entire array to be processed with very little programming effort.
- An array is homogeneous, that is all elements are of the same data type.

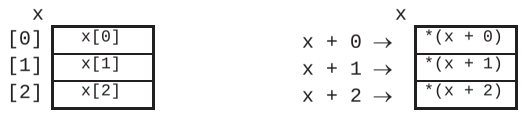
ARRAYS AND POINTERS

- With a 1-D array the array name is the address of the first thing in the array.

```
int  x[3];
```


```
  x  - address of the first thing in the integer array
x + 1- address of the second thing in the integer array
x + 2- address of the third thing in the integer array
```

- With a 1-D array dereferencing once, or indexing into the array once using the array access operator, gives a value in the array.

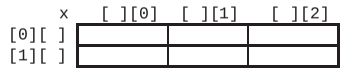



```
*x == x[0]
      - value of the first element in the array
*(x + 1)== x[1]
      - value of the second element in the array
*(x + 2)== x[2]
      - value of the third element in the array
```

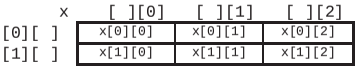

BASIC MULTI-DIMENSIONAL ARRAYS

- Basically, a 2-dimensional array can be thought of as a 2-D table of storage locations. The first index determines a row in the table and the second the column in that row.

```
int  x[2][3];   /* a 2-D set of ints */
```

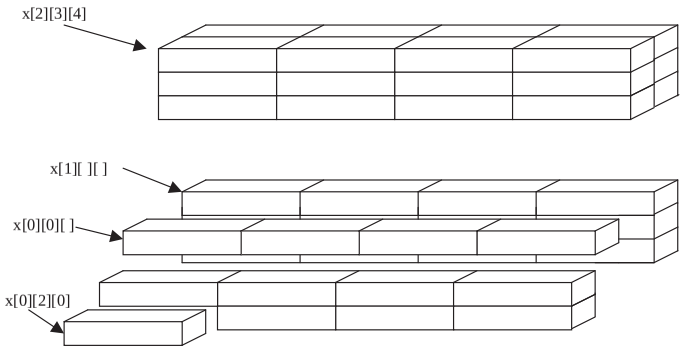


- To access a particular location 2 index values must be provided.



- A 3-dimensional array can be thought of as a 3-D set of storage locations. The first index determines a layer in the set, the second, a row in that layer, and the third, a particular element in that layer and row.

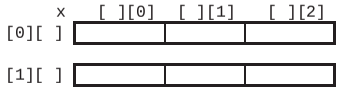
```
int  x[2][3][4];   /* a 3-D set of ints */
```



cont…

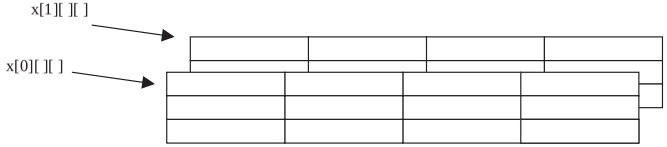
- Equivalently a 2-D array can be thought of as a set of 1-D arrays. Each row being a 1-D array of values.

```
int  x[2][3];    /* a set of 2, 1-D arrays containing 3 ints */
```



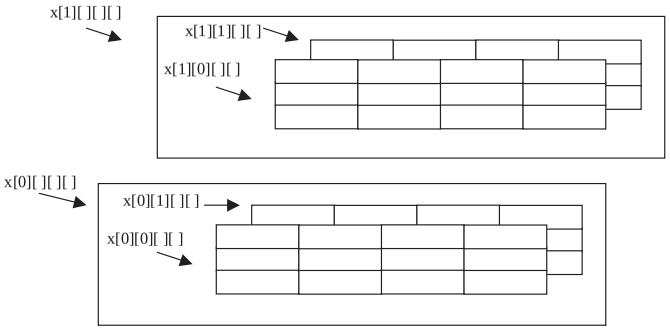
- A 3-D array can be thought of as a set of 2-D arrays. Each layer being a 2-D array of values.

```
int  x[2][3][4];    /* a set of 2, 3x4 2-D arrays */
```



- This conceptualization of arrays, as sets of sets, allows us to easily comprehend arrays of greater-than 3 dimensions. It also is a better model for understanding the relationship of pointers with n-dimensional arrays.

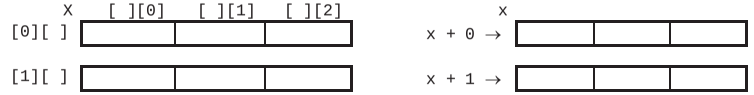
```
int  x[2][2][3][4];      /* a set of 2, 2x3x4 3-D arrays */
```



MULTIDIMENSIONAL ARRAYS AND POINTERS

- With a 2-D array the array name is the address of the first thing in the array. In this case the first thing in the array is a 1-D array:

```
int  x[2][3];  /* a set of 2 1-D arrays containing 3 ints */
```



- So the array name of this [2][3] array is the address of a set of 3 integers.

  `x` - address of the first array in the set of 2, 3 integer arrays.
  `x + 1`- address of the second array in the set of 2, 3 integer arrays.

  `*x == x[0]`
  - address of the first element in the first array
  (same as a 1-D array now that `x` has been dereferenced once.)

  `*(x + 1) == x[1]`
  - address of the first element in the second array
  (same as a 1-D array now that `x` has been dereferenced once.)

  `*(*(x + 0) + 1) == x[0][1]`
  - `x + 0` is the address of the first array
  - `*(x + 0)` is the address of the first element in the first array
  - `*(x + 0) + 1` is the address of the second element in the first array
  - `*(*(x + 0) + 1)` is the value of the second element in the first array

  `*(*(x + 1) + 2) == x[1][2]`
  - `x + 1` is the address of the second array
  - `*(x + 1)` is the address of the first element in the second array
  - `*(x + 1) + 2` is the address of the third element in the second array
  - `*(*(x + 1) + 2)` is the value of the third element in the second array

- With a 2-D array, dereferencing once, or indexing into the array once using the array access operator, gives an address. Dereferencing twice, or indexing into the array twice using the array access operator, gives a value from the array.
- Unless you dereference or index as many times as you have dimensions in an array, you still have an address, just a different kind of address.

COMMAND-LINE ARGUMENTS

- Information can be passed to a C program from the operating system's command line.
- The command-line arguments are packaged up into an array of strings, and the count of the number of strings and the array of strings are passed to *main( )*.
- The first line of the definition of *main( )* will now look like:

*int main(int argc, char *argv[ ])*

- *argc* is the argument count, *argv* is the array strings.
- Each comand-line argument can now be accessed within the program.

C STORAGE CLASSES

Automatic

* variables defined in a function or block of code are automatic by default
  * can be explicitly declared using the *auto* keyword
  * known only in the function or block of code they are defined in
  * exist only while the function or block is executing
  * not initialized by default

External

* variables defined outside any function
* known to all functions defined in the source file after the variable definition
* *extern* keyword declares an external and makes it known to a function or file regardless of where the external variable is actually defined
* exist for the entire duration of the program
* initialized to zero by default

Static automatic

* known only to the function in which they are defined
* *static* keyword defines a static automatic variable
* exist for the entire duration of the program
* initialized once, to zero by default
* retain their value between function calls

Static External

* external variable restricted to the file it is defined in
* *static* keyword declares an external variable to be static external

Dynamic memory

* allocated using *malloc( )*
* exists until released by *free( )*
* accessed by address

Function scope

External

* function that can be accessed by other files
* functions are external by default

Static

* function accessible only in the defining file
* *static* keyword declares a function to be static

STRUCTURES

- An array in C is a set or group of storage locations that are all of the same type.
- A structure in C allows a group of storage locations that are of different types to be created and treated as single unit.
- Structures are termed a data “aggregate”, since pieces of differing types are grouped together in a structure. These pieces are referred to as “members” of the structure.
- Structures are NOT the same as arrays because a structure itself is treated as a single entity and a structure’s name refers to the entire structure. (With an array, the array name is just the address of the first element in the set.)
- A structure definition creates the equivalent of a new data type. Any place you use a basic C data type you can use a structure. They can be passed as parameters, used as return values, you can take the address of one, C can compute the *sizeof* one.
- Since a structure we define is essentially a new data type, no existing C functions or operators were designed with our definitions in mind. So printf( ) and scanf( ) have no conversion specifiers for them, and the arithmetic operators won’t operate on them. But we can write our own functions to perform any of these operations.
- Some basic operators do still work with structures, & address-of, sizeof( ), * dereference, = assignment, (*type*) type cast.
- There are also two operators just for structure operations. The . member access operator and the –> member access thru a pointer operator.

ENUMERATED TYPES

- enumerated types can be created to give symbolic names to integer values and enlist the compiler for type checking.

Ex:

```
/*     FILE: enum.c     */

/* Enumerated types give symbolic constants with type
          checking.                                          */

#include <stdio.h>

enum GPA{F,D,C,B,A};

int main( )
{
          enum GPA grade;

          grade = A;
          printf("Score for an 'A': %d\n", grade);

          grade = B;
          printf("Score for an 'B': %d\n", grade);

          grade = C;
          printf("Score for an 'C': %d\n", grade);

          grade = D;
          printf("Score for an 'D': %d\n", grade);

          grade = F;
          printf("Score for an 'F': %d\n", grade);

          return 0;
}


/*    OUTPUT: enum.c

          Score for an 'A': 4
          Score for an 'B': 3
          Score for an 'C': 2
          Score for an 'D': 1
          Score for an 'F': 0

*/
```

UNIONS

- A union allows multiple mappings of the same piece of storage.
- Only one is in effect at any given time, but the same piece of memory can be utilized differently using a different mapping defined by the union.

Ex:

```
/*     FILE: union.c     */

/* A union that be either an array of
   ints or an array of floats, as
   needed.                          */

#include <stdio.h>
union intFloatArray{
  int iarray[5];
  float farray[5];
};

int main( )
{
  union intFloatArray ar;
  int i;

  for(i=0; i<5; i++)
    ar.iarray[i] = i*11;

  for(i=0; i<5; i++)
    printf("int[%d] = %d\n", i, ar.iarray[i]);

  for(i=0; i<5; i++)
    ar.farray[i] = i*1.1;

  for(i=0; i<5; i++)
    printf("float[%d] = %f\n", i, ar.farray[i]);

  printf("size of union int_float_array = %d\n",
                  sizeof(union intFloatArray));
  printf("size of ar = %d\n",sizeof(ar));

  return 0;
}


/*    OUTPUT: union.c

          int[0] = 0
          int[1] = 11
          int[2] = 22
          int[3] = 33
          int[4] = 44
          float[0] = 0.000000
          float[1] = 1.100000
          float[2] = 2.200000
          float[3] = 3.300000
          float[4] = 4.400000
          size of union int_float_array = 20
          size of ar = 20

*/
```


TYPEDEF

- C allows a type name to be defined using the *typedef* mechanism.
- The type defined used in situations where a standard C type would be used.
- *typedef* is often used to shorten the *struct name* type associated with a structure definition.

Ex:

```
/*     FILE: struct14.c     */

/* Typedef - simplified naming */

#include <stdio.h>

#define SIZE 5

struct part{
  char name[124];
  long no;
  double price;
};

typedef struct part part;     /* part becomes the type of "struct part" */

void print_part(const struct part * const);

int main( )
{
  part board;               /* One "part" */
  part inventory[SIZE];     /* Array to hold SIZE "part"s */

  FILE * fp;
  char * filename = "structBin.bin";
  int i;

  fp = fopen(filename,"w");

  if(fp != NULL){
    for(i=0; i < SIZE; i++){        /* Write the structures. */
      sprintf(board.name,"I/O card #%d", i);
      board.no = 127356 + i;
      board.price = 99.50 + i*3;

      fwrite(&board, sizeof(part), 1, fp);
    }

    fclose(fp);

    fp = fopen(filename,"r");
```

cont…

BIT OPERATORS

- C has a set of operators that can be used to perform bit-level operations.
- There are a pair of shift operators, and bitwise OR, AND, XOR, and NOT.
- All the operators are applied to each bit in the entire bit pattern. That is, all bits get shifted, all bits get ANDed, etc.

Ex:

```
/*     FILE: bitop.c     */

/* Exercises several C bit operators */

#include <stdio.h>

void setOneBit(int* ptr, int bit);
void setBit(int* ptr, int bit);
void clearBit(int* ptr, int bit);

int main( )
{
  int x;

  setOneBit(&x, 3);
  printf("x = %8.8X\n", x);

  clearBit(&x, 3);
  printf("x = %8.8X\n", x);

  x = 3;
  printf("\nx = %8.8X\n", x);
  setBit(&x, 3);
  printf("x = %8.8X\n", x);

  clearBit(&x, 3);
  printf("x = %8.8X\n", x);

  return 0;
}

void setOneBit(int* ptr, int bit)  /* sets the specified bit on,  */
{                                  /* ... all others will be off. */
    *ptr = 1 << bit;
}

void setBit(int* ptr, int bit) /* sets specified bit on and    */
{                               /* ... leaves all others as-is. */
    *ptr = (*ptr)| (1 << bit);
}

void clearBit(int* ptr, int bit) /* turns specified bit off. */
{
    *ptr = (*ptr)&(~(1 << bit));
}
```

cont…

DYNAMIC MEMORY ALLOCATION

- Up until now, the storage requirements of a C program needed to be established at compile time. Variables needed to be defined so that their storage requirements would be known. Arrays needed to be sized so that their storage requirements would be known.
- Dynamic memory allocation allows storage requirements to be determined at run-time.
- When the requirements are known, a request is made for the required amount of storage and the program can then proceed. This allows the program to tailor its storage use to exactly fit its needs during each run, or at any given point in time during a run.
- There is a cost. There is overhead incurred while the request for storage is being met.
- The key to dynamic memory allocation is pointers and the appropriate allocation function.
- *malloc( )* is the basic m-emory alloc-ation function. *Malloc* is told how many bytes of storage are needed and if the allocation can be satisfied *malloc* returns the address of the storage. If the allocation fails, *NULL* is returned.
- Dynamically allocated memory can and should be deallocated using *free( )*.

DYNAMIC MULTIDIMENSIONAL ARRAYS

- Multidimensional arrays can be allocated dynamically.
- Pointers of the correct type must be defined in order to utilize the multidimensional array using standard array notation.
- Only the first dimension of a dynamic multidimensional array can be variable.
- *Truly dynamic multidimensional arrays can be created; but then the compiler cannot be as helpful with the offsets computed when indexing. The programmer is therefore completely responsible for computing positions from indices. (We will not look at these types of arrays here.)*